IGOR Pro

Version 6.2

WaveMetrics, Inc.

Updates

Please check our website at <http://www.wavemetrics.com/> or the Igor Pro Help menu for minor updates, which we make available for you to download whenever bugs are fixed.

If there are features that you would like to see in future versions of Igor or if you find bugs in the current version, please let us know. We're committed to providing you with a product that does the job reliably and conveniently.

Notice

Manual Revision: April 21, 2011 (6.22)

Printed in the United States of America.

WaveMetrics, Inc.
PO Box 2088
Lake Oswego, OR 97035
USA

Voice: 503-620-3001
FAX: 503-620-6754
Email: support@wavemetrics.com, sales@wavemetrics.com
Web: http://www.wavemetrics.com/

Table of Contents

Volume I Getting Started

Volume II User's Guide: Part 1

Volume III User's Guide: Part 2

Volume IV Programming

Volume V Reference

Volume I Getting Started

Table of Contents

Chapter
I-1

Introduction to Igor Pro

Introduction to Igor Pro

Igor Pro is an integrated program for visualizing, analyzing, transforming and presenting experimental data.

Igor Pro's features include:

- Publication-quality graphics
- High-speed data display
- Ability to handle large data sets
- Curve-fitting, Fourier transforms, smoothing, statistics, and other data analysis
- Waveform arithmetic
- Image display and processing
- Combination graphical and command-line user interface
- Automation and data processing via a built-in programming environment
- Extensibility through modules written in the C and C++ languages

Some people use Igor simply to produce high-quality, finely-tuned scientific graphics. Others use Igor as an all-purpose workhorse to acquire, analyze and present experimental data using its built-in programming environment. We have tried to write the Igor program and this manual to fulfill the needs of the entire range of Igor users.

Igor Objects

The basic objects that all Igor users work with are:

- Waves
- Graphs
- Tables
- Page layouts

A collection of objects is called an "experiment" and is stored in an experiment file. When you open an experiment, Igor recreates the objects that comprise it.

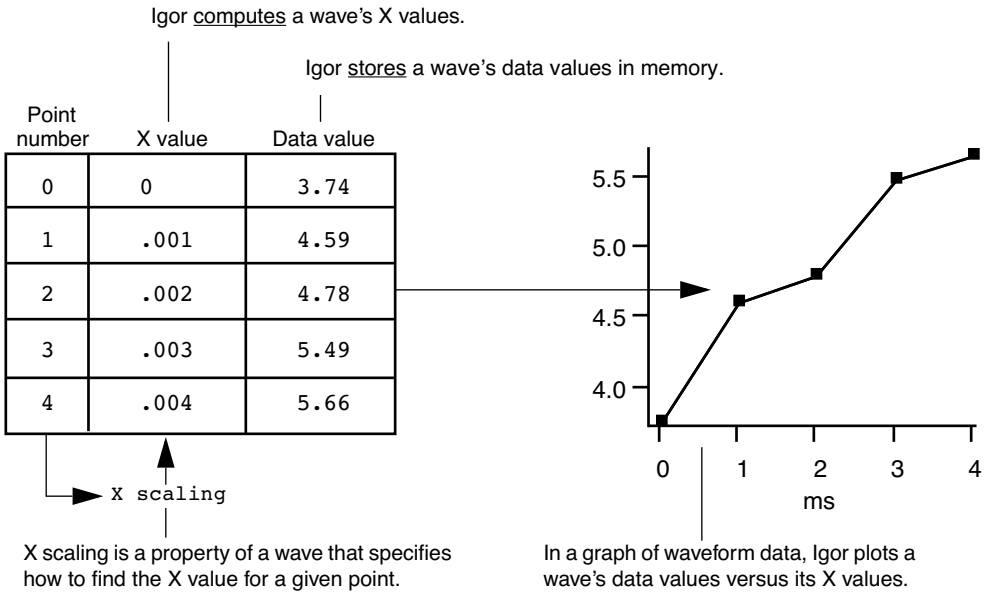
Waves — The Key Igor Concept

We use the term "wave" to describe the Igor object that contains an array of numbers. Wave is short for "waveform". The wave is the most important Igor concept.

Igor was originally designed to deal with waveform data. A waveform typically consists of hundreds to thousands of values measured at evenly spaced intervals of time. Such data is usually acquired from a digital oscilloscope, scientific instrument or analog-to-digital converter card.

The distinguishing trait of a waveform is the *uniform spacing* of its values along an axis of time or other quantity. An Igor wave has an important property called "X scaling" that you set to specify the spacing of your data. Igor *stores* the Y component for each point of a wave in memory but it *computes* the X component based on the wave's X scaling.

In the following illustration, the wave consists of five data points numbered 0 through 4. The user has set the wave's X scaling such that its X values start at 0 and increment by 0.001 seconds per point. The graph displays the wave's stored data values versus its computed X values.



Waves can have from one to four dimensions and can contain either numeric or text data.

Igor is also capable of dealing with data that does not fit the waveform metaphor. We call this XY data. Igor can treat two waves as an XY pair. In an XY pair, the data values of one wave supply the X component and the data values of another wave supply the Y component for each point in the pair.

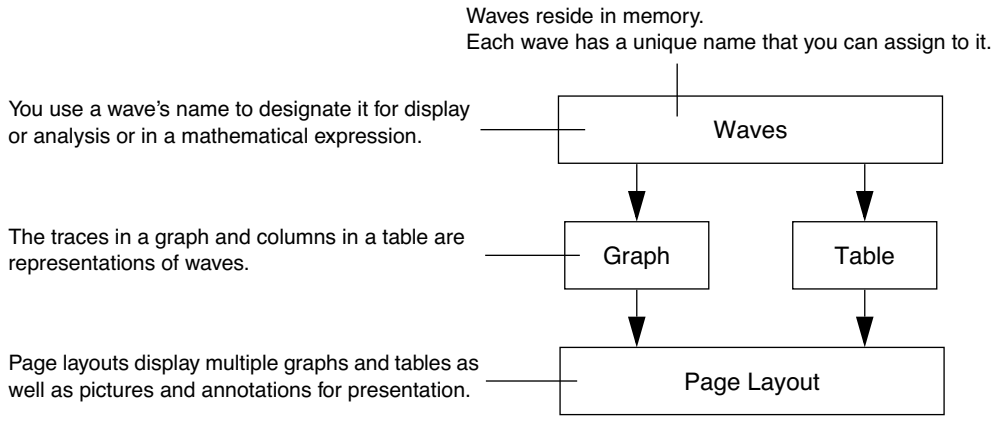
A few analysis operations, such as Fourier transforms, inherently work only on waveform data. They take a wave's X scaling into account.

Other operations work equally well on waveform or XY data. Igor can graph either type of data and its powerful curve fitting works on either type.

Most users create waves by loading data from a file. You can also create waves by typing in a table, evaluating a mathematical expression, acquiring from a data acquisition device, and accessing a database.

How Objects Relate

This illustration shows the relationships among Igor's basic objects. Waves are displayed in graphs and tables. Graphs and tables are displayed in page layouts. Although you can display a wave in a graph or table, a wave does not need to be displayed to exist.



Each object has a name so that it can be referenced in an Igor command. You can explicitly set an object's name or accept a default name created by Igor.

Graphs are used to visualize waves and to generate high-quality printouts for presentation. The traces in a graph are representations of waves. If you modify a wave, Igor automatically updates graphs. Igor labels the axes of a graph intelligently. Tick marks never run into one another and are always "nice" values no matter how you zoom in or pan around.

In addition to traces representing waveform or XY data, a graph can display an image or a contour plot generated from 2D data.

Tables are used to enter, inspect or modify wave data. A table in Igor is not the same as a spreadsheet in other graphing programs. A column in a table is a *representation* of the contents of a wave. The wave continues to exist even if you remove it from the table or close the table entirely.

Page layouts permit you to arrange multiple graphs and tables as well as pictures and annotations for presentation. If you modify a graph or table, either directly or indirectly by changing the contents of a wave, Igor automatically updates its representation in a layout.

Both graphs and layouts include drawing tools for adding lines, arrows, boxes, polygons and pictures to your presentations.

More Objects

Here are some additional objects that you may encounter:

- Numeric and string variables
- Data folders
- Notebooks
- Control panels
- 3D plots
- Procedures

A numeric variable stores a single number and a string variable stores a text string. Numeric and string variables are used for storing bits of data for Igor procedures.

A data folder can contain waves, numeric variables, string variables and other data folders. Data folders provide a way to keep a set of related data, such as all of the waves from a particular run of an experiment, together and separate from like-named data from other sets.

A notebook is like a text-editor or word-processor document. You can use a notebook to keep a log of results or to produce a report. Notebooks are also handy for viewing Igor technical notes or other text documentation.

A control panel is a window containing buttons, checkboxes and other controls and readouts. A control panel is created by an Igor user to provide a user interface for a set of procedures.

A 3D plot displays three–dimensional data as a surface, a scatter plot, or a path in space.

A procedure is a programmed routine that performs a task by calling Igor's built-in operations and functions and other procedures. Procedures range from very simple to very complex and powerful. You can run procedures written by WaveMetrics or by other Igor users. If you are a programmer or want to learn programming, you can learn to write your own Igor procedures to automate your work.

Igor's Toolbox

Igor's toolbox includes a wide range of built-in routines. You can extend it with user-defined procedures written in Igor itself and separately-compiled Igor extensions (plug-ins) that you obtain from WaveMetrics, from a colleague, from a third-party, or write yourself.

Built-In Routines

Each of Igor's built-in routines is categorized as a function or as an operation.

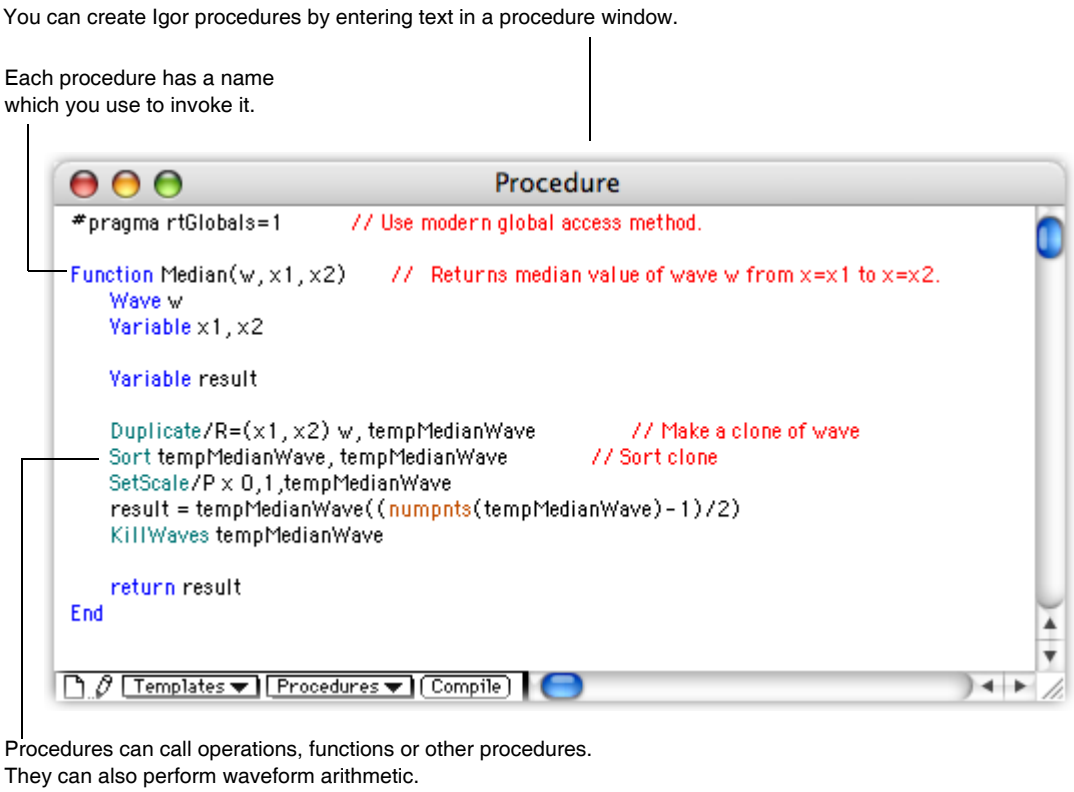
A built-in function is an Igor routine, such as sin, exp or ln, that directly returns a result. A built-in operation is a routine, such as Display, FFT or Integrate, that acts on an object and may create new objects but does not directly return a result.

The best way to get a sense of the scope of Igor's built-in routines is to scan the sections **Built-In Operations by Category** on page V-1 and **Built-In Functions by Category** on page V-6 in the reference volume of this manual.

For getting reference information on a particular routine it is usually most convenient to choose Help→Command Help and use the Igor Help Browser.

User-Defined Procedures

A user-defined procedure is a routine written in Igor's built-in programming language by entering text in a procedure window. It can call upon built-in or external functions and operations as well as other user-defined procedures to manipulate Igor objects. Sets of procedures are stored in procedure files.



Igor Extensions

An extension is a "plug-in" - a piece of external code that adds functionality to Igor. We use the terms "external operation" or "XOP" and "external function" or "XFUNC" for operations and functions added by extensions. An extension can add menus, dialogs and windows to Igor as well as operations and functions.

To write an extension, you must be a C or C++ programmer and you need the optional **Igor External Operations Toolkit**. See **Creating Igor Extensions** on page IV-181.

Although *creating* an extension is a job for a programmer, anyone can *use* an extension. The Igor installer automatically installs commonly used extensions in "Igor Pro Folder/Igor Extensions". These extensions are available for immediate use. An example is the Excel file loader accessible through Data→Load Waves.

Less commonly used extensions are installed in "Igor Pro Folder/More Extensions". Available extensions are described in the "XOP Index" help file (choose Windows→Help Windows→XOP Index.ihf). To activate an extension, see **Activating Extensions** on page III-424.

Igor's User Interface

Igor uses a combination of the familiar graphical user interface and a command-line interface. This approach gives Igor both ease-of-use and programmability.

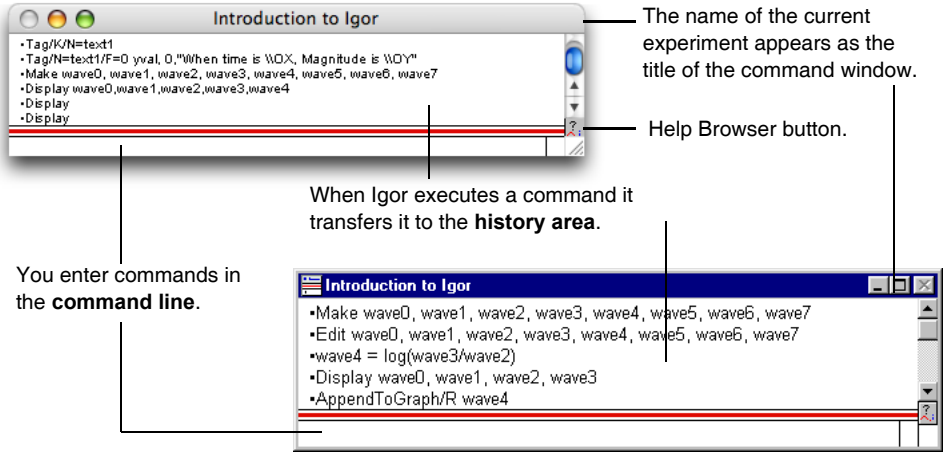
The job of the user interface is to allow you to apply Igor's operations and functions to objects that you create. You can do this in three ways:

- Via menus and dialogs
- By typing Igor commands directly into the command line
- By writing Igor procedures

The Command Window

The command window is Igor's control center. It appears at the bottom of the screen.

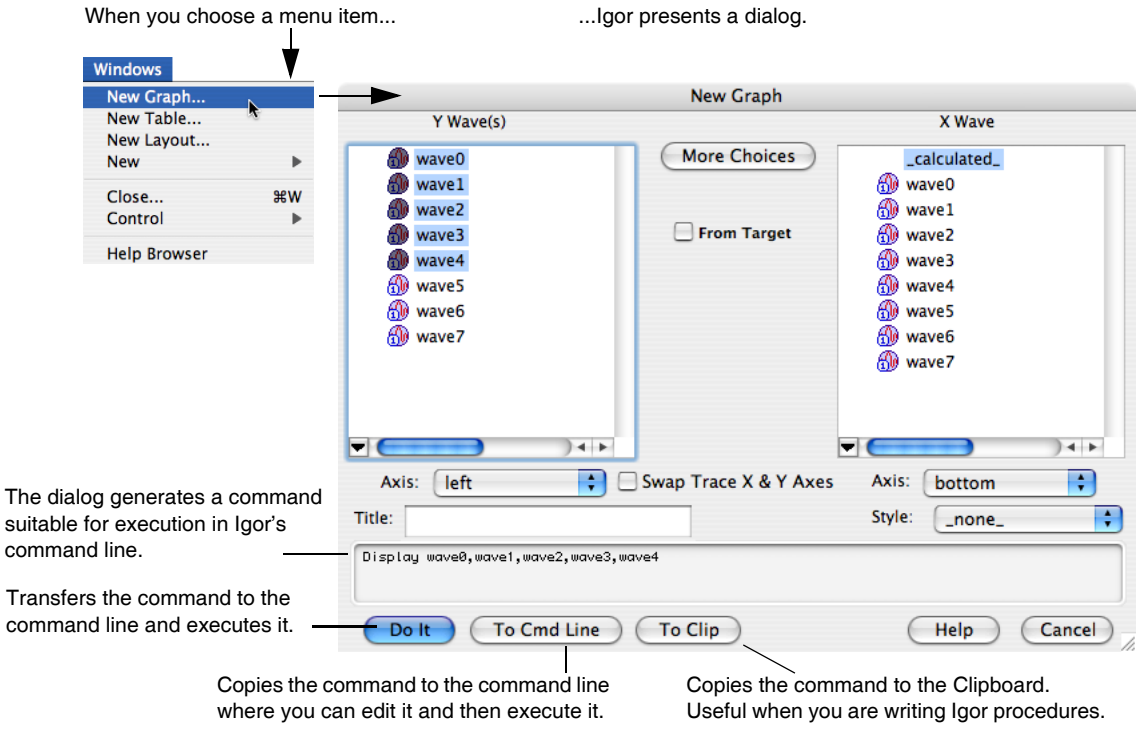
At the bottom of the command window is the command line. Above the red divider is the history area where executed commands are stored for review. Igor also uses the history area to report results of analyses like curve-fitting or waveform statistics.



Menus, Dialogs and Commands

Menus and dialogs provide easy access to the most commonly-used Igor operations.

When you choose a menu item associated with an Igor operation, Igor presents a dialog. As you use the dialog, Igor generates a command and displays it in the **command box** near the bottom of the dialog. When you click the Do It button, Igor transfers the command to the command line where it is executed.



As you get to know Igor, you will find that some commands are easier to invoke from a dialog and others are easier to enter directly in the command line.

There are some menus and dialogs that bypass the command line. Examples are the Save Experiment and Open Experiment items in the File menu.

Using Igor for Heavy-Duty Jobs

If you generate a lot of raw data or need to do custom technical computing, you will find it worthwhile to learn how to put Igor to heavy-duty use. It is possible to automate some or all of the steps involved in loading, processing and presenting your data. To do this, you must learn how to write Igor procedures.

Igor includes a built-in programming environment that lets you manipulate waves, graphs, tables and all other Igor objects. You can invoke built-in operations and functions from your own procedures to build higher-level operations customized for your work.

Learning to write Igor procedures is easier than learning a lower-level language such as FORTRAN or C. The Igor programming environment is interactive so you can write and test small routines and then put them together piece-by-piece. You can deal with very high level objects such as waves and graphs but you also have fine control over your data. Nonetheless, it is still programming. To master it requires an effort on your part.

The Igor programming environment is described in detail in **Volume IV Programming**. You can get started by reading the first three chapters of that volume.

You can also learn about Igor programming by examining the WaveMetrics Procedures and example experiments that were installed on your hard disk.

Igor Documentation

Igor includes an extensive online help system and a comprehensive PDF manual.

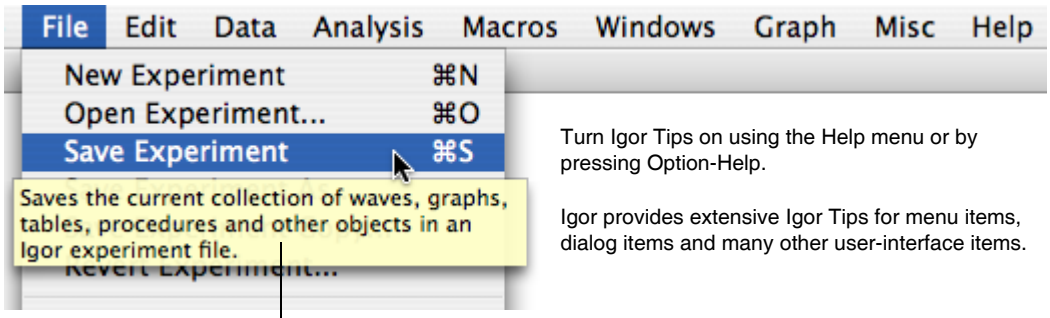
The online help provides guided tours, context-sensitive tips, general usage information for all aspects of Igor, and reference information for Igor operations, functions and keywords.

The PDF manual contains mostly the same information except for the context-sensitive tips.

The PDF manual, being in book format, is better organized for linear reading while the online help is usually preferred for reference information.

Igor Tips (*Macintosh only*)

Igor Tips present brief explanations of menus, dialogs and other user interface items.You turn Igor Tips on and off using the Help menu. When on, tips appear as you move your mouse over icons, menu items and dialog items.



Status Line Help, Tool Tips and Context-Sensitive Help (*Windows only*)

Igor's Windows help system provides three ways to get immediate help for an icon, a menu item, or a dialog item.

Status line help automatically shows brief descriptions of menu items and icons in the status line area at the bottom of the main Igor Pro window.

Tool tips provide very brief descriptions of icons when you point the cursor at the item.

Context-sensitive help displays a pop-up window containing information about the menu item, icon or dialog item of interest. Context-sensitive help is accessed as follows::

Menus and Icons:	Press Shift+F1 to get the question-mark cursor and click the item of interest.
Dialogs:	Click the question-mark button in the upper-right corner of dialog to get the question-mark cursor, then click a dialog item.

The Igor Help System

The Help menu provides access to Igor's help system, primarily through the Igor Help Browser.

- Use the Help menu or press Help (*Macintosh*) or F1 (*Windows*) to display the Igor Help Browser.
- Use the Igor Help Browser Help Topics tab to browse help topics.
- Use the Igor Help Browser Shortcuts tab to get a list of handy shortcuts and techniques.

- Use the Igor Help Browser Command Help tab to get reference information on Igor operations and functions. You can also right-click operation and function names in Igor windows to access the reference help.
- Use the Igor Help Browser Search tab to search Igor help, procedure and example files.

Most of the information displayed by the help browser comes from help files that are automatically loaded at launch time. The Windows→Help Windows submenu provides direct access to these help files.

The Igor Manual

The Igor PDF manual resides in "Igor Pro Folder/Manual". You can access it by choosing Help→Manual.

The manual consists of five volumes and an index.

Volume I contains the Getting Started material, including the Guided Tour of Igor Pro.

Volumes II and III contain general background and usage information for all aspects of Igor other than programming.

Volume IV contains information for people learning to do Igor programming.

Volume V contains reference information for Igor operations, functions and keywords.

Hard copy of the manual can be purchased from **http://www.lulu.com/wavemetrics**.

Learning Igor

To harness the power of Igor, you need to understand its fundamental concepts. Some of these concepts are familiar to you. However, Igor also incorporates a few concepts that will be new to you and may seem strange at first. The most important of these are *waves* and *experiments.*

In addition to this introduction, the primary resources for learning Igor are:

- The **Guided Tour of Igor Pro** in Chapter I-2

  The guided tour shows you how to perform basic Igor tasks step-by-step and reinforces your understanding of basic Igor concepts along the way. It provides an essential orientation to Igor and is highly-recommended for all Igor users.
- The Igor Pro PDF manual and online help files

  You can access the PDF manual through Igor's Help menu or by opening it directly from the Manual folder of the Igor Pro Folder where Igor is installed.

  You can access the help files through the Igor Help Browser (choose Help→Igor Help Browser) or directly through the Windows→Help submenu.
- The "Examples" experiments

  The example experiments illustrate a wide range of Igor features. They are stored in the Examples folder in the Igor Pro Folder. You can access them using the File→Example Experiments submenu or directly from the Examples folder of the Igor Pro Folder where Igor is installed.

You will best learn Igor through a combination of doing the guided tour, reading select parts of the manual (see suggestions following the guided tour), and working with your own data.

Getting Hands-On Experience

This introduction has presented an overview of Igor, its main constituent parts, and its basic concepts. The next step is to get some hands-on experience and reinforce what you have learned by doing the **Guided Tour of Igor Pro** on page I-11.

Chapter
I-2

Guided Tour of Igor Pro

Overview

In this chapter we take a look at the main functions of Igor Pro by stepping through some typical operations. Our goal is to orient you so that you can comfortably read the rest of the manual or explore the program on your own. You will benefit most from this tour if you actually do the instructed operations on your computer as you read this chapter. Screen shots are provided to keep you synchronized with the tour.

Terminology

If you have read Chapter I-1, **Introduction to Igor Pro**, you already know these terms.

Experiment	The entire collection of data, graphs and other windows, program text and whatnot that make up the current Igor environment or workspace.
Wave	Short for waveform, this is basically a named array of data with optional extra information.
Name	Because Igor contains a built-in programming and data transformation language, each object must have a unique name.
Command	This is a line of text that performs some task. Igor is command-driven so that it can be easily programmed.

About the Tour

This tour consists of three sections: **Guided Tour 1 - General Tour** on page I-14, **Guided Tour 2 - Data Analysis** on page I-48, and **Guided Tour 3 - Histograms and Curve Fitting** on page I-55.

The General Tour is a rambling exploration intended to introduce you to the way things work in Igor and give you a general orientation. This tour takes 2 to 4 hours but does not have to be performed in one sitting.

The second and third tours guide you through Igor's data analysis facilities including simple curve fitting.

When you've completed the first tour you may prefer to explore freely on your own before starting the second tour.

Guided Tour 1 - General Tour

In this exercise, we will generate data in three ways (typing, loading, and synthesizing) and we will generate graph, table, and page layout windows. We will jazz up a graph and a page layout with a little drawing and some text annotation. At the end we will explore some of the more advanced features of Igor Pro.

Launching Igor Pro

The Igor Pro application is typically installed in:

`/Applications/Igor Pro Folder (`*`Macintosh`*`)`

`C:\Program Files\WaveMetrics\Igor Pro Folder (`*`Windows 32-bit`*`)`

`C:\Program Files (x86)\WaveMetrics\Igor Pro Folder (`*`Windows 64-bit`*`)`

1. **Double-click the Igor Pro application file on your hard disk.**

   On Windows you can also start Igor using the Start menu.

   If Igor was already running, choose the File→New Experiment menu item.

2. **Use the Misc menu to turn preferences off.**

   Turning preferences off ensures that the tour works the same for everyone.

Entering Data

1. **If a table window is showing, click in it to bring it to the front.**

   When Igor starts up, it creates a new blank table unless this feature is turned off in the Miscellaneous Settings dialog. If the table is not showing, perform the following two steps:

   1a. Choose the Windows→New Table menu item.

   The New Table dialog appears.

   1b. Click the Do It button.

   A new blank table is created.

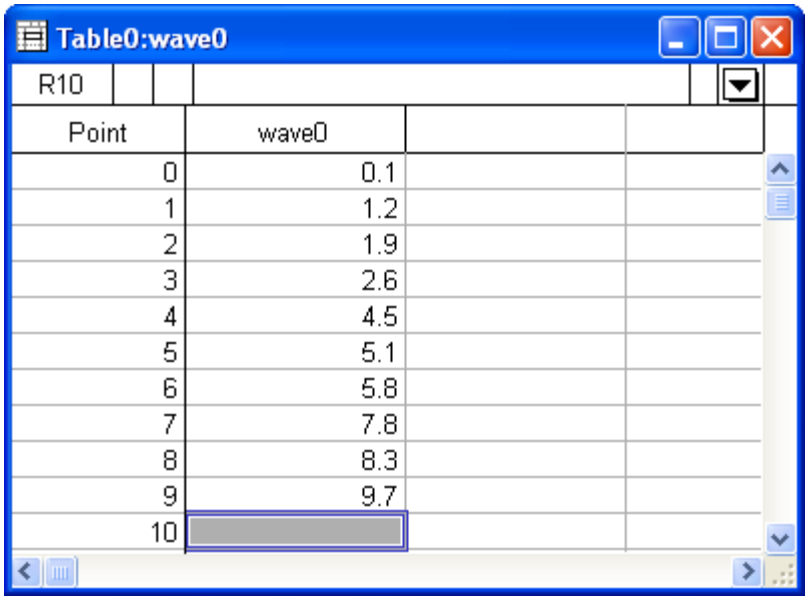
2. **Type "0.1" and then press Return or Enter on your keyboard.**

   This creates a wave named "wave0" with 0.1 for the first point. Entering a value in the first row (point 0) of the first blank column automatically creates a new wave.

3. **Type the following numbers, pressing Return or Enter after each one:**

   1.2
   1.9
   2.6
   4.5
   5.1
   5.8
   7.8
   8.3
   9.7

   Your table should look like this:



4. **Click in the first cell of the first blank column.**

5. **Enter the following numbers in the same way:**

   -0.12
   -0.08
   1.3
   1
   0.54
   0.47
   0.44
   0.2
   0.24
   0.13

6. **Choose Data→Rename.**

7. **Click "wave0" in the list and then click the arrow icon.**

8. **Replace "wave0" with "time".**

   Notice that you can't use the name "time" because it is the name of a built-in string function. We apologize for usurping such a common name.

9. **Change the name to "timeval".**

10. **Select "wave1" from the list, click the arrow icon, and type "yval".**

11. **Click Do It.**

    Notice the column headers in the table have changed to reflect the name changes.

Making a Graph

1. **Choose the Windows→New Graph menu item.**

   The New Graph dialog will appear. This dialog comes in a simple form that most people will use and a more complex form that you can use to create complex multiaxis graphs in one step.

2. **If you see a button labeled Fewer Choices, click it.**

   The button is initially labeled More Choices because the simpler form of the dialog is the default.

3. **In the Y Wave(s) list, select "yval".**

4. **In the X Wave list, select "timeval".**

5. **Click Do It.**

   A simple graph is created.

Touching up a Graph

1. **Position the cursor directly over the trace in the graph and double-click.**

   The Modify Trace Appearance dialog appears. You could also have chosen the corresponding menu item from the Graph menu.

   Note: The Graph menu appears only when a graph is the target window. The *target* window is the window that menus and dialogs act on by default.

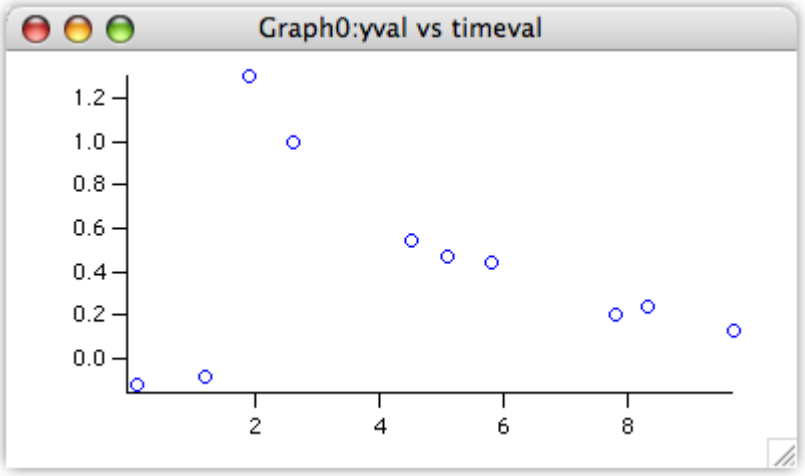
2. **Choose Markers from the Mode pop-up menu.**

3. **Select the open circle from the pop-up menu of markers.**

4. **Set the marker color to blue.**

5. **Click Do It.**

   Your graph should now look like this:



6. **Position the cursor over the bottom axis line.**

   The cursor changes to this shape: ↕. This indicates the cursor is over the axis and also that you can offset the axis (and the corresponding plot area edge) to a new position.

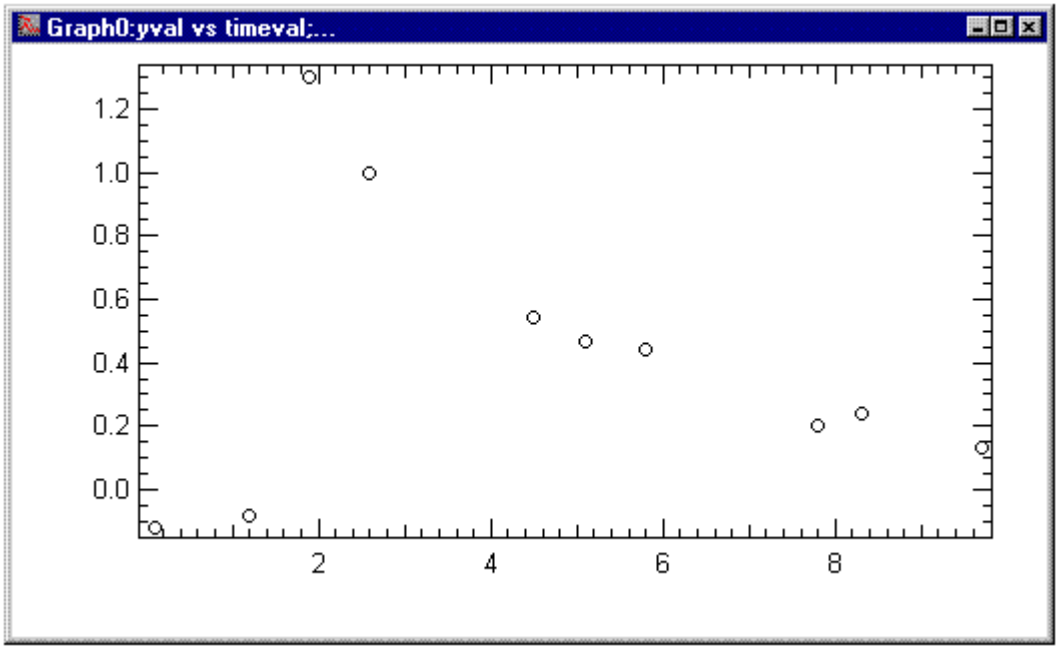
7. **Double-click directly on the axis.**

   The Modify Axis dialog appears. If another dialog appears, click cancel and try again, making sure the ↕ cursor is showing.

   Note the Live Update checkbox in the top/right corner of the Modify Axis dialog. When it is checked, changes that you make in the dialog are immediately reflected in the graph. When it is unchecked, the changes appear only when you click Do It. The Modify Axis dialog is the only one with a Live Update checkbox.

8. **If it is not already showing, click the Axis tab.**
9. **Choose On from the Mirror Axis pop-up.**
10. **Click the Auto/Man Ticks tab.**
11. **Click the Minor Ticks checkbox so it is checked.**
12. **Click the Ticks and Grids tab.**
13. **Choose Inside from the Location pop-up.**
14. **Choose the left axis from the Axis pop-up menu in the top-left corner of the dialog and then repeat steps 8 through 13.**
15. **Click Do It.**

    Your graph should now look like this:



16. **Again double-click the bottom axis.**

    The Modify Axis dialog appears again.
17. **Click the Axis tab.**
18. **Uncheck the Standoff checkbox.**
19. **Choose the left axis from the Axis pop-up menu and repeat step 18.**
20. **Click Do It.**

    Notice that some of the markers now overlap the axes. The axis standoff setting pushes out the axis so that markers and traces do not overlap the axis. You can use Igor's preferences to ensure this and other settings default to your liking, as explained below.
21. **Double-click one of the tick mark labels (such as "6") on the bottom axis.**

    The Modify Axis dialog reappears, this time with the Axis Range tab showing. If another dialog or tab appears, cancel and try again, making sure to double click one of the tick mark labels on the bottom axis.
22. **Choose "Round to nice values" from the pop-up menu that initially reads "Use data limits".**
23. **Choose the left axis from the Axis pop-up menu and repeat step 22.**
24. **Click Do It.**

    Notice that the limits of the axes now fall on "nice" values.

Adding a Legend

1. **Choose the Graph→Add Annotation menu item.**

   The Add Annotation dialog appears.

2. **Click the Text tab if it is not already selected.**

3. **Choose Legend from the pop-up menu in the top-left corner.**

   Igor inserts text to create a legend in the Annotation text entry area. The Preview area shows what the annotation will look like. Note that the text `\s(yval)` generates the symbol for the yval wave. This is an "escape sequence", which creates special effects such as this.

4. **In the Annotation area, change the second instance of "yval" to "Magnitude".**

5. **Click the Frame tab and choose Box from the Annotation Frame pop-up menu.**

6. **Choose Shadow from the Border pop-up menu.**

7. **Click the Position tab and choose Right Top from the Anchor pop-up menu.**

   Specifying an anchor point helps Igor keep the annotation in the best location as you make the graph bigger or smaller.

8. **Click Do It.**

Adding a Tag

1. **Choose the Graph→Add Annotation menu item.**

2. **Choose Tag from the pop-up menu in the top-left corner.**

3. **In the Annotation area of the Text tab, type "When time is ".**

4. **Choose "Attach point X value" from the Dynamic pop-up menu in the Insert area of the dialog.**

   Igor inserts the `\0X` escape code into the Annotation text entry area.

5. **In the Annotation area, add ", Magnitude is ".**

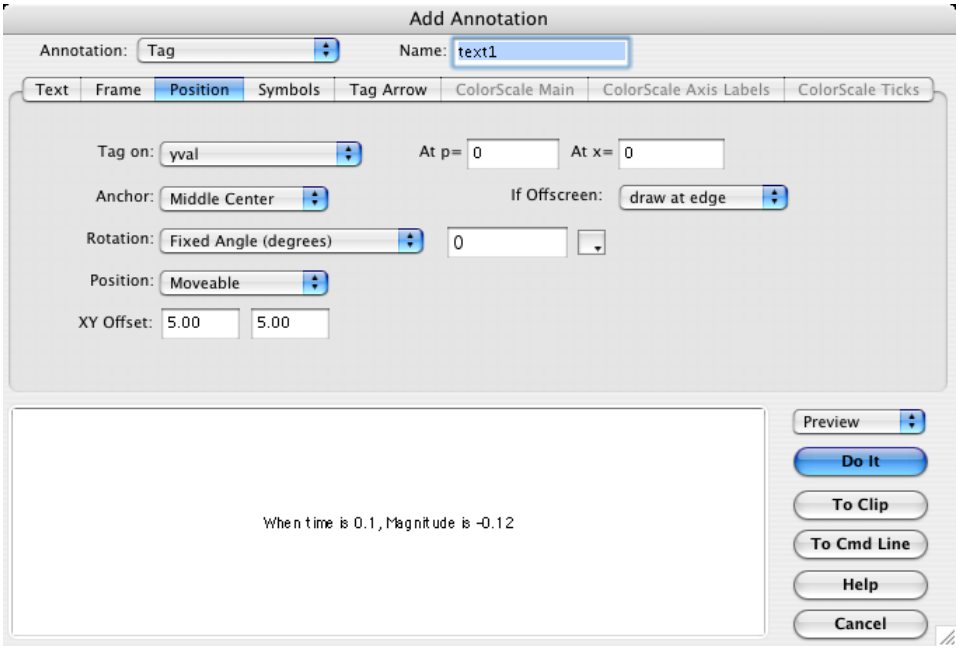
6. **Choose "Attach point Y value" from the Dynamic pop-up menu.**

7. **Switch to the Frame tab and choose None from the Annotation Frame pop-up menu.**

8. **Switch to the Tag Arrow tab and choose Arrow from the Connect Tag to Wave With pop-up menu.**

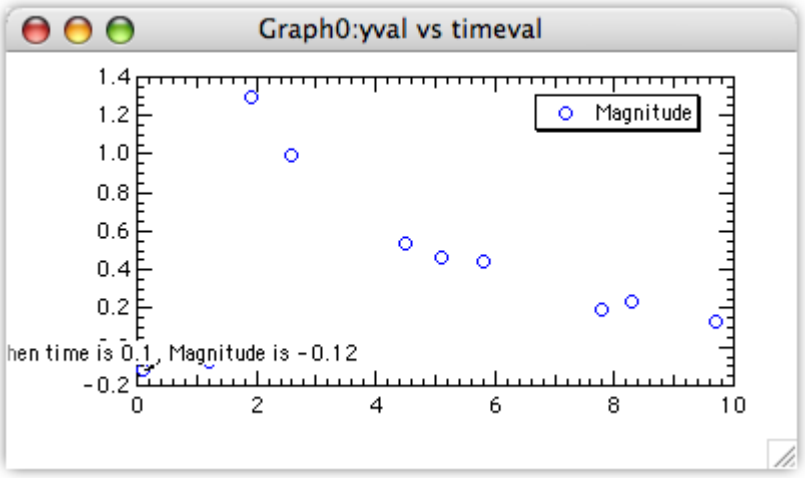
9. **Click the Position tab and choose "Middle center" from the Anchor pop-up menu.**

   The dialog should now look like this:



10. **Click Do It.**

    Your graph should now look like this:



    The tag is attached to the first point. An arrow is drawn from the center of the tag to the data point but you can't see it because it is hidden by the tag itself.

11. **Position the cursor over the text of the tag.**

    The cursor changes to a hand. This indicates you can reposition the tag relative to the data point it is attached to.

12. **Drag the tag up and to the right about 1 cm.**

    You can now see the arrow.

13. **With the cursor over the text of the tag, press Option (*Macintosh*) or Alt (*Windows*).**

    The cursor changes to this shape:  . (You may need to nudge the cursor slightly to make it change.)

14. **While pressing Option (*Macintosh*) or Alt (*Windows*), drag the box cursor to a different data point.**

    The tag jumps to the new data point and the text is updated to show the new X and Y values. Option-drag (*Macintosh*) or Alt-drag (*Windows*) the tag to different data points to see their X and Y values.

    Notice that the tip of the arrow touches the marker. This doesn't look good, so let's change it.

15. **Double-click the text part of the tag.**

    The Modify Annotation dialog appears.

16. **Click the Tag Arrow tab and change the Line/Arrow Standoff from "Auto" to "10".**

17. **Click the Change button.**

    The tip of the arrow now stops 10 points from the marker.

Using Preferences

If you have already set preferences to your liking and do not want to disturb them, you can skip this section.

1. **Use the Misc menu to turn preferences on.**
2. **Click the graph window if it is not already active.**
3. **Choose the Graph→Capture Graph Prefs menu item.**

    The Capture Graph Preferences dialog appears.

4. **Click the checkboxes for XY plot axes and for XY plot wave styles.**
5. **Click Capture Preferences.**
6. **Choose Windows→New Graph.**
7. **Choose "yval" as the Y wave and "timeval" as the X wave.**
8. **Click Do It.**

    The new graph is created with a style similar to the model graph.

9. **Press Option (*Macintosh*) or Alt (*Windows*) while clicking the close button of the new graph.**

    The new graph is killed without presenting a dialog.

10. **Choose Graph→Capture Graph Prefs.**
11. **Click the checkboxes for XY plot axes and for XY plot wave styles.**
12. **Click Revert to Defaults.**
13. **Use the Misc menu to turn preferences off.**

    We turn preferences off during the guided tour to ensures that the tour works the same for everyone. This is not something you would do during normal work.

Making a Page Layout

1. **Choose the Windows→New Layout menu item.**

    The New Page Layout dialog appears. The names of all tables and graphs are shown in the list.

2. **In the Objects to Lay Out list, select Graph0.**
3. **Command-click (*Macintosh*) or Ctrl-click (*Windows*) on Table0.**
4. **Click Do It.**

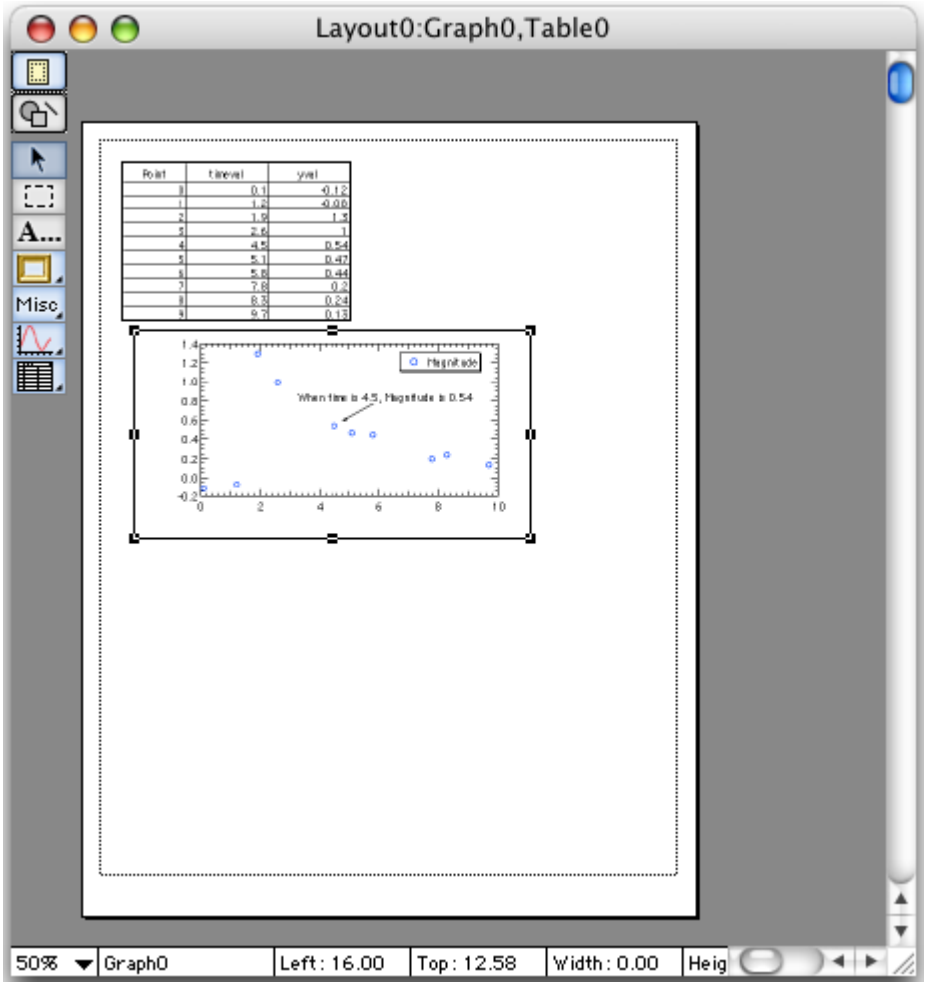
    A page layout window appears with a Table0 object on top of a Graph0 object.

    The layout initially shows objects at 50% but you may prefer to work at 100%. You can use the pop-up menu in the lower left corner of the window to change magnification.

5. **Click the Table0 object in the layout window.**

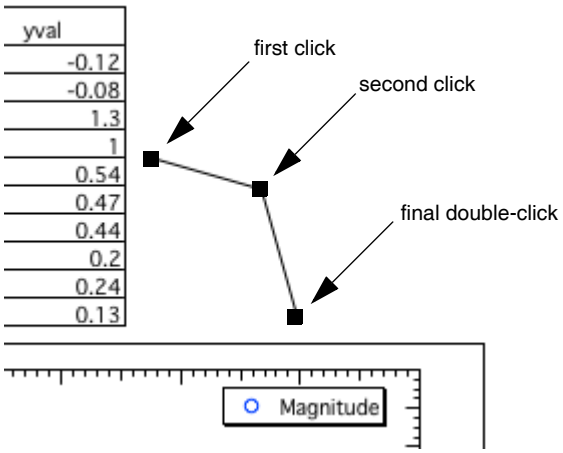
    The table object becomes selected, resize handles are drawn around the edge and the cursor becomes a hand when over the table.

6. **Click in the middle of the table and drag it so you can see the right edge of the table.**
7. **Position the cursor over the small black square (handle) in the middle of the right side of the table.**
   The cursor changes to a two headed arrow indicating you can drag in the direction of the arrows.
8. **Drag the edge of the table to the left until it is close to the edge of the third column of numbers.**
   You need only get close — Igor snaps to the nearest grid line.
9. **In a similar fashion, adjust the bottom of the table to show all the data but without any blank lines.**
10. **Drag the table and graph to match the picture:**



11. **Click this icon in the tool palette:**  .
    This activates the drawing tools.
12. **Click this icon in the drawing tool palette:**  .
    This is the polygon tool.

13. **Click once just to the right of the table, click again about 2 cm right and 1 cm down and finally double-click a bit to the right of the last click and just above the graph.**



The double-click exits the "draw polygon" mode and enters "edit polygon mode". If you wish to touch up the defining vertices of the polygon, do so now by dragging the handles (the square boxes at the vertices).

14. **Click the Arrow tool in the palette.**

This exits polygon edit mode.

15. **Click the polygon to select it.**
16. **Click the draw environment pop-up icon,  , and choose At End in the Line Arrow submenu.**
17. **Click this icon in the tool palette:  .**

This is the operate icon. The drawing tools are replaced by the normal layout tools.

We are finished with the page layout for now.

18. **Choose Windows→Send To Back.**

Saving Your Work

1. **Identify or create a folder on your hard disk for saving your Igor files.**

For example, you might create a folder for your Igor files in your user folder.

Don't save your Igor files in the Igor Pro folder as this complicates updating Igor and making backups.

2. **Choose File→Save Experiment.**

The save file dialog appears.

3. **Make sure that Packed Experiment File is selected as the file format.**
4. **Type "Tour #1 a.pxp" in the name box.**
5. **Navigate to the folder where you want to keep your tour files.**
6. **Click Save.**

The "Tour #1a.pxp" file contains all of your work in the current experiment, including waves that you created, graphs, tables and page layout windows.

If you want to take a break, you can quit Igor Pro now.

Loading Data

Before loading data we will use a Notebook window to look at the data file.

0. **If you are returning from a break, launch Igor and open your "Tour #1 a.pxp" experiment file. Then turn off preferences using the Misc menu.**

   Opening the "Tour #1 a.pxp" experiment file restores the Igor workspace to the state it was in when you saved the file. You can open the experiment file by using the Open Experiment item in the File menu or by double-clicking the experiment file.

1. **Choose the File→Open File→Notebook menu item.**

2. **Navigate to the folder "Igor Pro Folder:Learning Aids:Sample Data" folder and open "Tutorial Data #1.txt."**

   A Notebook window showing the contents of the file appears. If desired, we could edit the data and then save it. For now we just observe that the file appears to be tab-delimited (tabs separate the columns) and contains names for the columns. Note that the name of the first column will conflict with the data we just entered and the other names have spaces in them.

3. **Click the close button or press Command-W (*Macintosh*) or Ctrl+W (*Windows*).**

   A dialog appears asking what you want to do with the window.

4. **Click the Kill button.**

   The term Kill means to "completely remove from the experiment". The file will not be affected.

   Now we will actually load the data.

5. **Choose Data→Load Waves→Load Delimited Text.**

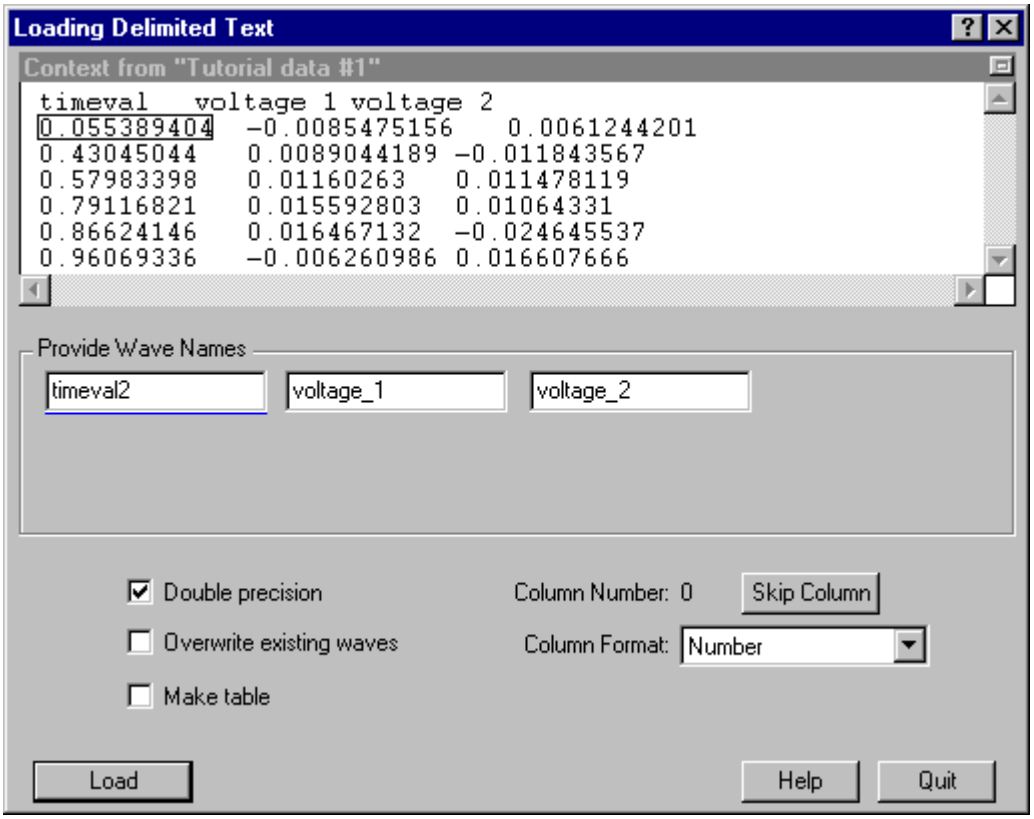
   An Open File dialog appears.

6. **Again choose "Tutorial Data #1.txt" and click Open.**

   The Loading Delimited Text dialog appears. The name "timeval" is highlighted and an error message is shown. Observe that the names of the other two columns were fixed by replacing the spaces with underscore characters.

7. **Change "timeval" to "timeval2".**

   The dialog should now look like this:



8. **Click the Make Table box to select it and then click Load.**

   The data is loaded and a new table is created to show the data.

9. **Click the close button of the new table window.**

   A dialog is presented asking if you want to create a recreation macro.

10. **Click the No Save button.**

    The data we just loaded is still available in Igor. A table is just a way of viewing data and is not necessary for the data to exist.

The Load Delimited Text menu item that you used is a shortcut that uses default settings for loading delimited text. Later, when you load your own data files, choose Data→Load Waves→Load Waves so you can see all of the options.

Appending to a Graph

0. **If necessary, click in Graph0 to bring it to the front.**

   The Graph menu is available only when the target window is a graph.

1. **Choose the Graph→Append Traces to Graph menu item.**

   The Append Traces dialog appears. It is very similar to the New Graph dialog that you used to create the graph.

2. **In the Y Wave(s) list, select voltage_1 and voltage_2.**

3. **In the X Wave list, select timeval2.**

4. **Click Do It.**

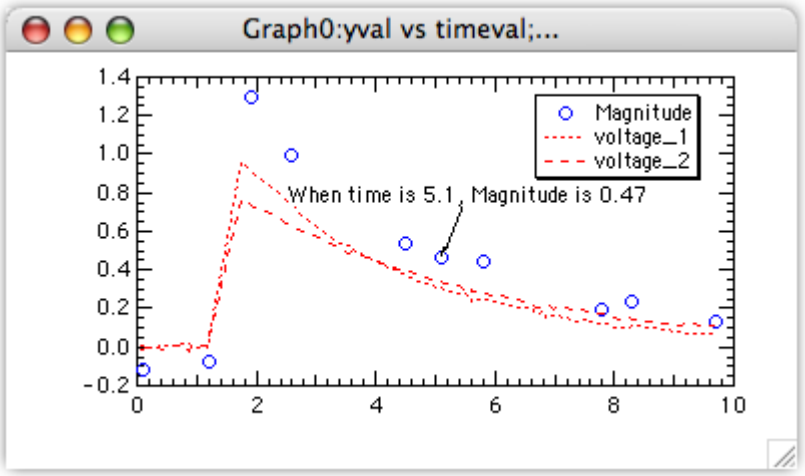
   Two additional traces are appended to the graph. Notice that they are also appended to the Legend.

5. **Position the cursor over one of the traces in the graph and double-click.**

   The Modify Trace Appearance dialog appears with the trace you clicked on already selected.

6. **If necessary, select voltage_1 in the list of traces.**
7. **Choose dashed line #2 from the Line Style pop-up menu.**
8. **Select voltage_2 in the list of traces.**
9. **Choose dashed line #3 from the Line Style pop-up menu.**
10. **Click Do It.**

    Your graph should now look like this:



Offsetting a Trace

1. **Position the cursor directly over the voltage_2 trace.**

   The voltage_2 trace has the longer dash pattern.

2. **Press and hold the mouse button for about 1 second.**

   An XY readout appears in the lower-left corner of the graph and the trace will now move with the mouse.

3. **With the mouse button still down, press Shift while dragging the trace up about 1 cm and release.**

   The Shift key constrains movement to vertical or horizontal directions.

   You have added an offset to the trace. If desired, you could add a tag to the trace indicating that it has been offset and by how much.

Unoffsetting a Trace

1. **Choose the Edit→Undo Trace Drag menu item.**

   You can undo many of the interactive operations on Igor windows if you do so before performing the next interactive operation.

2. **Choose Edit→Redo Trace Drag.**

   The following steps show how to remove an offset after it is no longer undoable.

3. **Double-click the voltage_2 trace.**

   The Modify Trace Appearance dialog will appear with voltage_2 selected. (If voltage_2 is not selected, click it to select it.) The Offset checkbox will be checked.

4. **Click the Offset checkbox.**

   This turns offset off for the selected trace.

5. **Click the Offset checkbox again.**

   The Trace Offset dialog appears showing the offset value you introduced by dragging.

6. **Click the Cancel button or press Escape.**

   The Offset checkbox should still be unchecked.

7. **Click Do It.**

   The voltage_2 trace is returned to its original position.

Drawing in a Graph

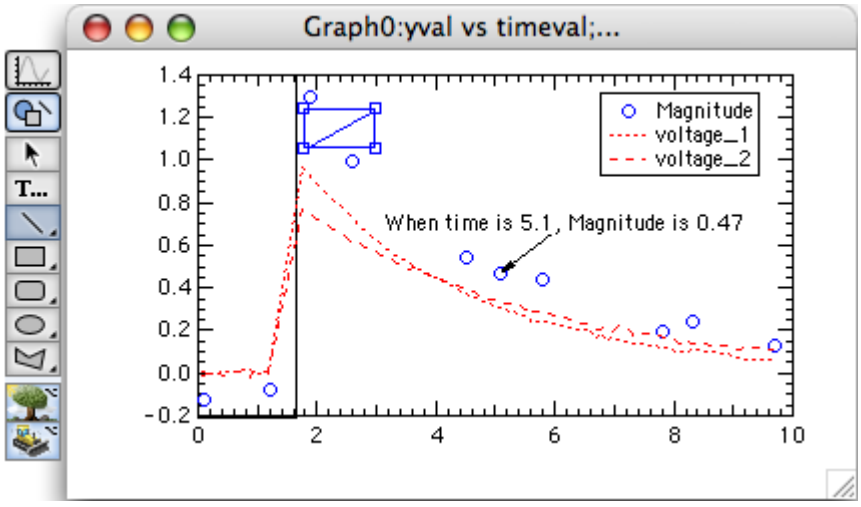
1. **If necessary click in Graph0 to bring it to the front.**
2. **Choose the Graph→Show Tools menu item or press Command-T (*Macintosh*) or Ctrl+T (*Windows*).**

   A toolbar is added to the graph. The second icon from the top ( ) is selected indicating that the graph is in drawing mode as opposed to normal (or "operate") mode.
3. **Click the top icon ( ) to go into normal mode.**

   Normal mode is for interacting with graph objects such as traces, axes and annotations. Drawing mode is for drawing lines, rectangles, polygons and so on.
4. **Click the second icon to return to drawing mode.**
5. **Press Option (*Macintosh*) or Alt (*Windows*) and press and hold down the mouse button while the cursor is in the draw environment icon (tree and grass).**

   A pop-up menu showing the available drawing layers and their relationship to the graph elements appears (the items in the menu are listed in back-to-front order).
6. **Choose UserBack from the menu.**

   We will be drawing behind the axes, traces and all other graph elements.
7. **Click the rectangle tool and drag out a rectangle starting at the upper-left corner of the plot area (y= 1.4, x=0 on the axes) and ending at the bottom of the plot area and about 1.5 cm in width (y= -0.2, x= 1.6).**
8. **Click the line tool and draw a line as shown, starting at the left (near the peak of the top trace) and ending at the right:**



9. **Click the draw environment icon and choose At Start from the Line Arrow item.**
10. **Click the Text tool icon T....**
11. **Click just to the right of the line you have just drawn.**

    The Create Text dialog appears.

12. **Type "Precharge".**
13. **From the Anchor pop-up menu, choose "Left center".**
14. **Click Do It.**
15. **Click the graph's zoom button (*Macintosh*) or maximize button (*Windows*).**

    Notice how the rectangle and line expand with the graph. Their coordinates are measured relative to the plot area (rectangle enclosed by the axes).
16. **Click the graph's zoom button (*Macintosh*) or restore button (*Windows*).**
17. **Click the Arrow tool and then double-click the rectangle.**

    The Modify Rectangle dialog appears showing the properties of the rectangle.
18. **Enter 0 in the Thickness box in the Line Properties section.**

    This turns off the frame of the rectangle.
19. **Choose Light Gray from the Fill Mode pop-up menu.**
20. **Choose black from the Fore Color pop-up menu under the Fill Mode pop-up menu.**
21. **Click Do It.**

    Observe that the rectangle forms a gray area behind the traces and axes.
22. **Again, double-click the rectangle.**

    The Modify Rectangle dialog appears.
23. **From the X Coordinate pop-up menu, choose Axis Bottom.**

    The X coordinates of the rectangle will be measured in terms of the bottom axis — as if they were data values.
24. **Press Tab until the X0 box is selected and type "0".**
25. **Tab to the Y0 box and type "0".**
26. **Tab to the X1 box and type "1.6".**
27. **Tab to Y1 and type "1".**

    The X coordinates of the rectangle are now measured in terms of the bottom axis and the left side will be at zero while the right side will be at 1.6.

    The Y coordinates are still measured relative to the plot area. Since we entered zero and one for the Y coordinates, the rectangle will span the entire height of the plot area.
28. **Click Do It.**

    Notice the rectangle is nicely aligned with the axis and the plot area.
29. **Click the operate icon,  , to exit drawing mode.**
30. **Press Option (*Macintosh*) or Alt (*Windows*), click in the middle of the plot area and drag about 2 cm to the right.**

    The axes are rescaled. Notice that the rectangle moved to align itself with the bottom axis.
31. **Choose Edit→Undo Scale Change.**

Making a Window Recreation Macro

1. **Click the graph's close button.**

   Igor presents a dialog which asks if you want to save a window recreation macro. The graph's name is "Graph0" so Igor suggests "Graph0" as the macro name.
2. **Click Save.**

   Igor generates a window recreation macro in the currently hidden procedure window. A window recreation macro contains the commands necessary to recreate a graph, table, or page layout. You can invoke this macro to recreate the graph you just closed.

3. **Choose the Windows→Procedure Windows→Procedure Window menu item.**

   The procedure window is always present but is usually hidden to keep it out of the way. The window now contains the recreation macro for Graph0. You may need to scroll up to see the start of the macro. Because of the way it is declared, `Window Graph0() : Graph`, this macro will be available from the Graph Macros submenu of the Windows main menu.

4. **Click the procedure window's close button.**

   This hides the procedure window. Most other windows will put up a dialog asking if you want to kill or hide the window, but the built-in procedure window and the help windows simply hide themselves.

Recreating the Graph

1. **Choose the Windows→Graph Macros→Graph0 menu item.**

   Igor executes the Graph0 macro which recreates a graph of the same name.

2. **Repeat step #1.**

   The Graph0 macro is executed again but this time Igor gave the new graph a slightly different name, Graph0_1, because a graph named Graph0 already existed.

3. **While pressing Option (*Macintosh*) or Alt (*Windows*) click the close button of Graph0_1.**

   The window is killed without presenting a dialog.

Saving Your Work

1. **Choose the File→Save Experiment As menu item.**
2. **Navigate back to the folder where you saved the first time.**
3. **Change the name to "Tour #1 b.pxp" and click Save.**

   If you want to take a break, you can quit from Igor now.

Using Igor Documentation

Now we will take a quick look at how find information about Igor.

In addition to guided tours such as this one, Igor includes context-sensitive help, general usage information and reference information. The main guided tours as well as the general and reference information are available in both the online help files and in the Igor Pro PDF manual.

We'll start with context-sensitive help.

1. **On Macintosh only, turn Igor Tips on by choosing Help→Show Igor Tips.**

   On Macintosh Igor tips for icons, menu items and dialog items appear in yellow textboxes.

   On Windows tips for icons and menu items appear in the status line at the bottom of the Igor Pro frame window.

2. **Click the Data menu item and move the cursor over the items in the menu.**

   Notice the tips in yellow textboxes on Macintosh and in the status line on Windows.

3. **Choose Data→Load Waves→Load Waves.**

   Igor displays the Load Waves dialog. This dialog provides an interface to the LoadWave operation which is how you load data into Igor from text data files.

4. **On Macintosh only, move the cursor over the Load Columns Into Matrix checkbox.**

   An Igor tip appears in a yellow textbox. You can get a tip for most dialog items this way.

5. **On Windows only, click the question-mark icon in the top-right corner of the dialog window and then click the Load Columns Into Matrix checkbox.**

   Context-sensitive help appears in a yellow textbox. You can get a tip for most dialog items this way.

6. **Click the Cancel button to quit the dialog.**

   Now let's see how to get reference help for a particular operation.

7. **Choose Help→Command Help.**

   The Igor Help Browser appears with the Command Help tab displayed.

   The information displayed in this tab comes from the Igor Reference help file - one of many help files that Igor automatically opens at launch. Open help files are directly accessible through Windows→Help Windows but we will use the Igor Help Browser right now.

8. **Make sure all three checkboxes in the Command Help tab of the help browser are checked and that all three pop-up menus are set to All.**

   These checkboxes and pop-up menus control which operations, functions and keywords appear in the list.

9. **Click any item in the list and then type "Loa".**

   Igor displays help for the LoadData operation. We want the LoadWave operation.

10. **Press the down-arrow key a few times until LoadWave is selected in the list.**

    Igor displays help for the LoadWave operation in the Help windoid.

    Another way to get reference help is to Ctrl-click (Macintosh) or right-click (Windows) the name of an operation or function and choose the "Help For" menu item. This works in the command window and in procedure, notebook and help windows.

    While we're in the Igor Help Browser, let's see what the other tabs are for.

11. **Click each of the Help Browser tabs and note their contents.**

    You can explore these tabs in more detail later.

    Next we will take a quick trip to the Igor Pro PDF manual. If you are doing this guided tour using the PDF manual, you may want to just read the following steps rather than do them to avoid losing your place.

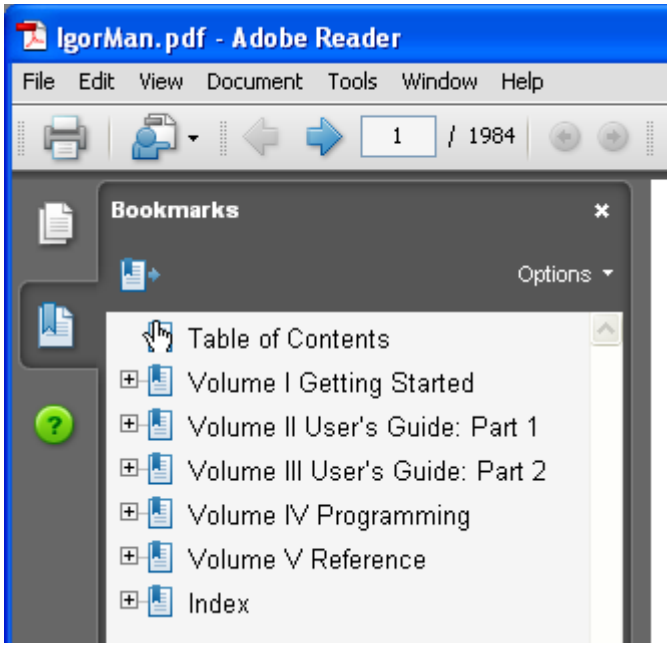
12. **Click the Manual tab and then click the Open Online Manual button.**

    Igor opens the PDF manual in your PDF viewer - typically Adobe Reader or Apple's Preview program.

    If you use Adobe Reader for viewing PDF files, you should have a Bookmarks pane on the left side of the PDF manual window. If not, choose View→Navigation Panel→Bookmarks in Reader.

    If you use Apple's Preview for PDF files, you should have a drawer on one side of the main page. If not, choose View→Drawer in Preview.

    Note in the Reader Bookmarks pane or the Preview drawer that the PDF manual is organized into five volumes plus an index.



13. **Use the Bookmarks pane to get a sense of what's in the manual.**

    Expand the volume bookmarks to see the chapter names.

    You may have noticed that the Igor PDF manual is rather large - about 2,000 pages at last count. You'll be happy to know that we don't expect you to read it cover-to-cover. Instead, read chapters as the need arises.

    The information in the manual is also in the online help files. The manual, being in book format, is better organized for linear reading while the online help is usually preferred for accessing reference information.

    In case you ever want to open it directly, you can find the PDF manual in "Igor Pro Folder/Manual".

That should give you an idea of where to look for information about Igor. Now let's get back to our hands-on exploration of Igor.

Graphically Editing Data

0. **If you quit Igor after the last save, open your "Tour #1 b.pxp" experiment and turn off preferences.**
1. **Adjust the positions of the graph and table so you can see both.**

   Make sure you can see the columns of data when the graph is the front window.
2. **If necessary, click in the Graph0 window to bring it to the front.**
3. **Click the drawing mode icon,  , to activate the drawing tools.**

4. **While pressing Option (*Macintosh*) or Alt (*Windows*) on the keyboard, move the cursor over the polygon icon ( ) and click and hold the mouse button.**

   A pop-up menu appears.

5. **Choose the Edit→Edit Wave menu item.**

6. **Click one of the open circles of the yval trace.**

   The trace is redrawn using lines and squares to show the location of the data points.

7. **Click the second square from the left and drag it 1 cm up and to the right.**

   Notice point 1 of yval and timeval changes in the table.

8. **Press Command-Z (*Macintosh*) or Ctrl+Z (*Windows*) or choose Edit→Undo.**

9. **Click midway between the first and second point and drag up 1 cm.**

   Notice a new data point 1 of yval and timeval appears in the table.

10. **Press Option (*Macintosh*) or Alt (*Windows*) and click the new data point with the tip of the lightning bolt.**

    The new data point is zapped.

    You could also have pressed Command-Z (*Macintosh*) or Ctrl+Z (*Windows*) to undo the insertion.

11. **Press Command (*Macintosh*) or Ctrl (*Windows*), click the line segment between the second and third point and drag a few cm to the right.**

    The line segment is moved and two points of yval and timeval are changed in the table.

12. **Press Command-Z (*Macintosh*) or Ctrl+Z (*Windows*) or choose Edit→Undo.**

13. **Click in the operate icon, , to exit drawing mode.**

14. **Choose File→Revert Experiment and answer *Yes* in the dialog.**

    This returns the experiment to the state it was in before we started editing the data.

Making a Category Plot (Optional)

Category plots show continuous numeric data plotted against non-numeric text categories.

1. **Choose the Windows→New Table menu item.**

2. **Click in the Do It button.**

   A new blank table is created. We could have used the existing table but it is best to keep unrelated data separate.

3. **Type "Monday" and then press Return or Enter.**

   A wave named "textWave0" was created with the text Monday as the value of the first point. Entering a non-numeric value in the first row of the first blank column automatically creates a new text wave.

4. **Type the following lines, pressing Enter after each one:**

   Tuesday
   Wednesday
   Thursday

5. **Click in the first cell of the next column and enter the following values:**

   10
   25
   3
   16

6. **Click in the first cell of the next column and enter the following values:**

   0
   12
   30
   17

7. **Choose Windows→New→Category Plot.**

   A dialog similar to the New Graph dialog appears. This dialog shows only text waves in the right-hand list.

8. **Click the From Target checkbox to select it.**

   This limits the list of waves to those in the target window. The target window is the table we just made.

9. **In the Y Wave(s) list, select both items and select textWave0 in the X Wave list.**

10. **Click Do It.**

    A category plot is created.

11. **Double-click one of the bars.**

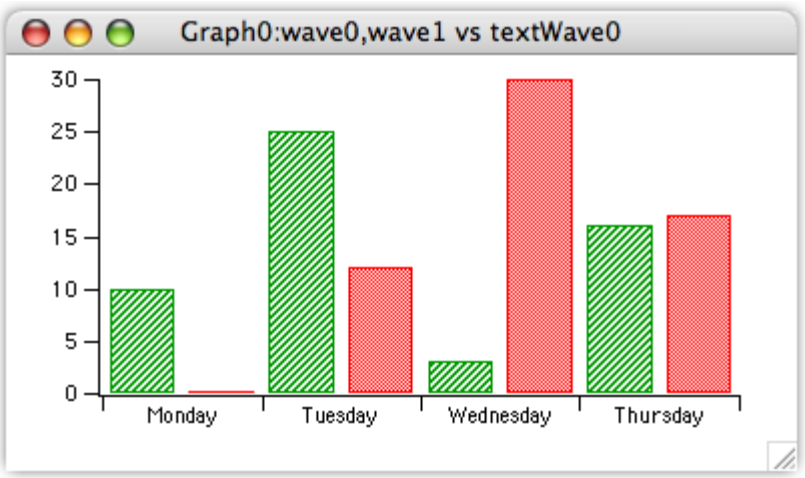
    The Modify Trace Appearance dialog appears.

12. **Using the "+Fill Type" pop-up menu, change the fills of each trace to any desired pattern.**

    You might also want to change the colors.

13. **Click Do It.**

    Your graph should now look like this:



Category Plot Options (Optional)

This section explores various category-plot options. If you are not particularly interested in category plots, you can stop now, or at any point in the following steps, by closing the graph and table and skipping forward to the next section.

1. **Double-click one of the bars and, if necessary, select the top trace in the list.**

2. **From the Grouping pop-up menu, choose Stack on Next.**

3. **Click Do It.**

   The left bar in each group is now stacked on top of the right bar.

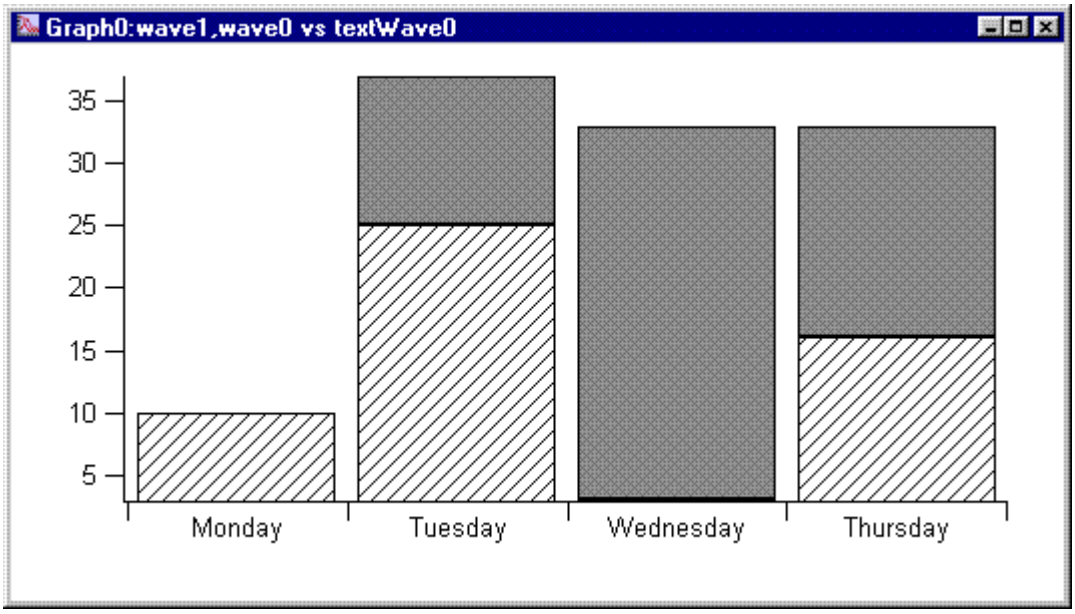
4. **Choose the Graph→Reorder Traces menu item.**

5. **Reverse the order of the items in the list by dragging the top item down. Click Do It.**

   The bars are no longer stacked and the bars that used to be on the left are now on the right. The reason the bars are not stacked is that the trace that we set to Stack on Next mode is now last and there is no next trace.

6. **Again using the Modify Trace Appearance dialog, set the top trace to Stack on next. Click Do It.**

   The category plot graph should now look like this:



7. **Enter the following values in the next blank column in the table:**

   7
   10
   15
   9

   This creates a new wave named wave2.

8. **Click in the graph to bring it to the front.**

9. **Choose Graph→Append to Graph→Category Plot.**

10. **In the Y list, select wave2 and click Do It.**

    The new trace is appended after the previous two. Because the second trace was in Stack on Next mode, the new trace is on the bottom of each set of three stacked bars.

11. **Using the Modify Trace Appearance dialog, change the grouping mode of the middle trace to none.**

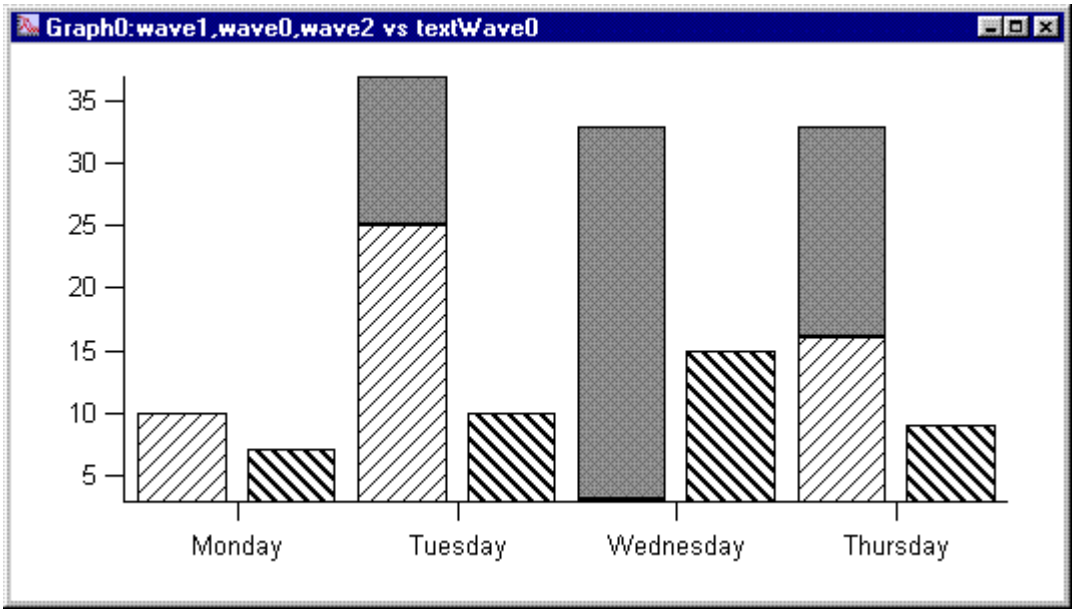
    Now the new bars are to the right of a group of two stacked bars. You can create any combination of stacked and side-by-side bars.

12. **Double-click directly on the bottom axis.**

    The Modify Axis dialog appears with the bottom axis selected.

13. **Click the Auto/Man Ticks tab.**

14. **Select the Tick In Center checkbox and then click Do It.**
    Notice the new positions of the tick marks.

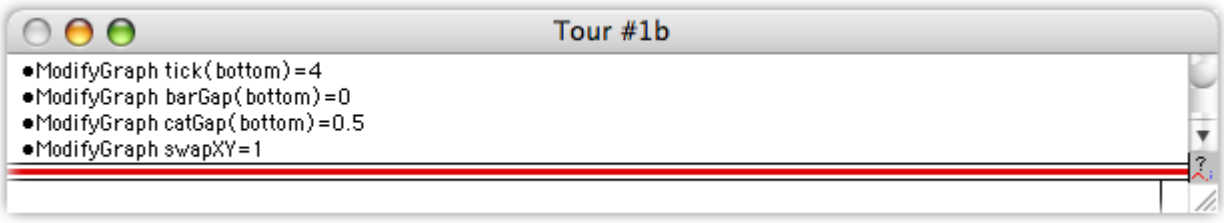


15. **Again double-click directly on the bottom axis.**
16. **Click the Axis tab.**
17. **Change the value of Bar Gap to zero and then click Do It.**
    Notice that the bars within a group are now touching.
18. **Use the Modify Axis dialog to set the Category Gap to 50%.**
    The widths of the bars shrink to 50% of the category width.
19. **Choose Graph→Modify Graph.**
20. **Select the "Swap X & Y Axes" checkbox and then click Do It.**
    This is how you create a horizontal bar plot.
21. **Close both the graph and table windows without saving recreation macros.**

The Command Window

Parts of this tour make use of Igor's command line to enter mathematical formulae. Let's get some practice now.

Your command window should look something like this:



The command line is the space below the separator whereas the space above the separator is called the history area.

1. **Click in the command line, type the following line and press Return or Enter.**

```
Print 2+2
```

    The Print command as well as the result are placed in the history area.

2. **Press the Up Arrow key.**

   The line containing the print command is selected, skipping over the result printout line.

3. **Press Return or Enter.**

   The selected line in the history is copied to the command line.

4. **Edit the command line so it matches the following and press Return or Enter.**

   ```
   Print "The result is ",2+2
   ```

   The Print command takes a list of numeric or string expressions, evaluates them and prints the results into the history.

5. **Choose the Help→Igor Help Browser menu item.**

   The Igor Help Browser appears.

   You can also display the help browser by pressing Help (*Macintosh*) or F1 (*Windows*), or by clicking the question-mark icon near the right edge of the command window.

6. **Click the Command Help tab in the Igor Help Browser.**

7. **Deselect the Functions and Programming checkboxes and select the Operations checkbox.**

   A list of operations appears.

8. **In the pop-up menu next to the Operations checkbox, choose About Waves.**

9. **Select PlaySound in the list.**

   Tip: Click in the list to activate it and then type "p" to jump to PlaySound.

10. **Click the Help windoid, scroll down to the Examples section, and select the first four lines of example text (starting with "Make", ending with "PlaySound sineSound).**

11. **Choose the Edit→Copy menu to copy the selection.**

12. **Close the Igor Help Browser.**

13. **Choose Edit→Paste.**

    All four lines are pasted into the command line area. You can view the lines using the miniature scroll arrows that appear at the right-hand edge of the command line.

14. **Press Return or Enter to execute the commands.**

    The four lines are executed and a short tone plays. (*Windows*: You may see an error message if your computer is not set up for sound.)

15. **Click once on the last line in the history area (PlaySound sineSound).**

    The entire line (less the bullet) is selected just as if you pressed the arrow key.

16. **Press Return or Enter once to transfer the command to the command line and a second time to execute it.**

    The tone plays again as the line executes.

    We are finished with the "sineSound" wave that was created in this exercise so let's kill the wave to prevent it from cluttering up our wave lists.

17. **Choose Data→Kill Waves.**

    The Kill Waves dialog appears.

18. **Select "sineSound" and click Do It.**

    The sineSound wave is removed from memory.

19. **Again click once on the history line "PlaySound sineSound".**

20. **Press Return or Enter twice to re-execute the command.**

    An error dialog is presented because the sineSound wave no longer exists.

21. **Click OK to close the error dialog.**

22. **Choose Edit→Clear Command Buffer or press Command-K (*Macintosh*) or Ctrl+K (*Windows*).**

    When a command generates an error, it is left in the command line so you can edit and re-execute it. In this case we just wanted to clear the command line.

Browsing Waves

1. **Choose the Data→Browse Waves menu item.**

   The Browse Waves dialog appears. You can view the properties of the waves that are in memory and available for use in the current experiment and can also examine waves that are stored in individual binary files on disk.

2. **Click "timeval" in the list.**

   The dialog shows the properties of the timeval wave.

3. **Click in the Wave Note area of the dialog.**

   A wave note is text you can associate with a wave. You can both view and edit the text of the note. The other fields in this dialog are read-only.

4. **Type the following line:**

   This wave was created by typing data into a table.

5. **Click the other waves in the list while observing their properties.**

6. **Click the done button to exit the dialog.**

Using the Data Browser

The Data Browser provides another way to browse waves. You can also browse numeric and string variables.

1. **Choose the Data→Data Browser menu item.**

   The Data Browser appears.

2. **Make sure all of the checkboxes in the top-left corner of the Data Browser are checked.**

3. **Click on the timeval wave icon to select it.**

   Note that the wave is displayed in the plot pane at the bottom of the Data Browser and the wave's properties are displayed just above in the info pane.

4. **Control-click (*Macintosh*) or right-click (*Windows*) on the timeval wave icon.**

   A contextual menu appears with a number of actions that you can perform on the selection.

5. **Press Escape to dismiss the contextual menu.**

   You can explore that and other Data Browser features later on your own.

6. **Click the Data Browser's close box to close it.**

Synthesizing Data

In this section we will make waves and fill them with data using arithmetic expressions.

1. **Choose the Data→Make Waves menu item.**

   The Make Waves dialog appears.

2. **Type "spiralY", Tab, and then "spiralX" in the second box.**

3. **Change Rows to 1000.**

4. **Click Do It.**

   Two 1000 point waves have been created. They are now part of the experiment but are not visible because we haven't put them in a table or graph.

5. **Choose Data→Change Wave Scaling.**

6. **If a button labeled More Options is showing, click it.**

7. **In the Wave(s) list, click spiralY and then Command-click (*Macintosh*) or Ctrl-click (*Windows*) spiralX.**
8. **Choose Start and Right for the SetScale Mode pop-up menu.**
9. **Enter "0" for Start and "50" for Right.**
10. **Click Do It.**

    This executes a **SetScale** command specifying the X scaling of the spiralX and spiralY waves. X scaling is a property of a wave that maps a point number to an X value. In this case we are mapping point numbers 0 through 999 to X values 0 through 50.
11. **If necessary, click in the command window to bring it to the front.**
12. **Type the following on the command line and then press Return or Enter:**

    ```
    spiralY= x*sin(x)
    ```

    This is a waveform assignment statement. It assigns a value to each point of the destination wave (spiralY). The value stored for a given point is the value of the righthand expression at that point. The meaning of *x* in a waveform assignment statement is determined by the X scaling of the destination wave. In this case, *x* takes on values from 0 to 50 as Igor evaluates the righthand expression for points 0 through 999.
13. **Execute this in the command line:**

    ```
    spiralX= x*cos(x)
    ```

    Now both spiralX and spiralY have their data values set.

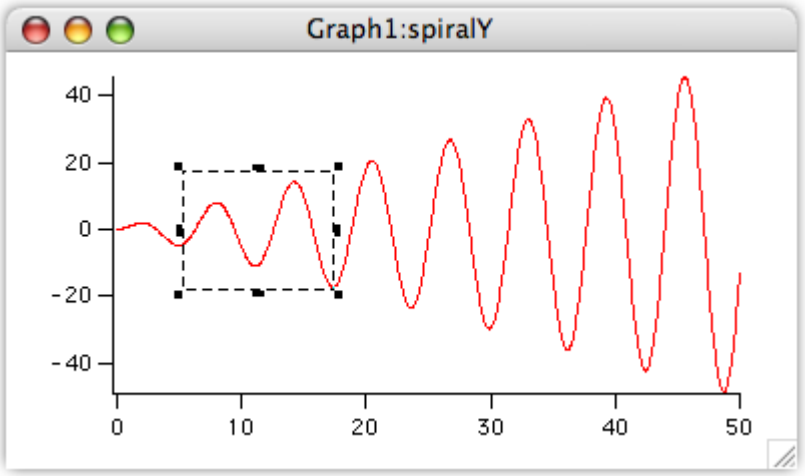
Zooming and Panning

1. **Choose the Windows→New Graph menu item.**
2. **If necessary, uncheck the From Target checkbox.**
3. **In the Y Wave(s) list, select "spiralY".**
4. **In the X Wave list, select "_calculated_".**
5. **Click Do It.**

    Note that the X axis goes from 0 to 50. This is because the **SetScale** command we executed earlier set the X scaling property of spiralY which tells Igor how to compute an X value from a point number. Choosing _calculated_ from the X Wave list graphs the spiralY data values versus these calculated X values.
6. **Position the cursor in the interior of the graph.**

    The cursor changes to a cross-hair shape.

7. **Click and drag down and to the right to create a marquee as shown:**



You can resize the marquee with the black squares (handles). You can move the marquee by dragging the dashed edge of the marquee.

8. **Position the cursor inside the marquee.**

The mouse pointer changes to this shape: , indicating that a pop-up menu is available.

9. **Click and choose Expand from the pop-up menu.**

The axes are rescaled so that the area enclosed by the marquee fills the graph.

10. **Choose Edit→Undo Scale Change or press Command-Z (*Macintosh*) or Ctrl+Z (*Windows*).**
11. **Choose Edit→Redo Scale Change or press Command-Z (*Macintosh*) or Ctrl+Z (*Windows*).**
12. **Press Option (*Macintosh*) or Alt (*Windows*) and position the cursor in the middle of the graph.**

The cursor changes to a hand shape. You may need to move the cursor slightly before it changes shape.

13. **With the hand cursor showing, drag about 2 cm to the left.**
14. **While pressing Option (*Macintosh*) or Alt (*Windows*), click the middle of the graph and gently fling it to the right.**

The graph continues to pan until you click again to stop it.

15. **Choose Graph→Autoscale Axes or press Command-A (*Macintosh*) or Ctrl+A (*Windows*).**

Continue experimenting with zooming and panning as desired.

16. **Press Command-Option-W (*Macintosh*) or Ctrl+Alt+W (*Windows*).**

The graph is killed. Option (*Macintosh*) or Alt (*Windows*) avoided the normal dialog asking whether to save the graph.

Making a Graph with Multiple Axes

1. **Choose the Windows→New Graph menu item.**
2. **If you see a button labeled More Choices, click it.**

We will use the more complex form of the dialog to create a multiple-axis graph in one step.

3. **In the Y Wave(s) list, select "spiralY".**
4. **In the X Wave list, select "spiralX".**
5. **Click Add.**

The selections are inserted into the lower list in the center of the dialog.

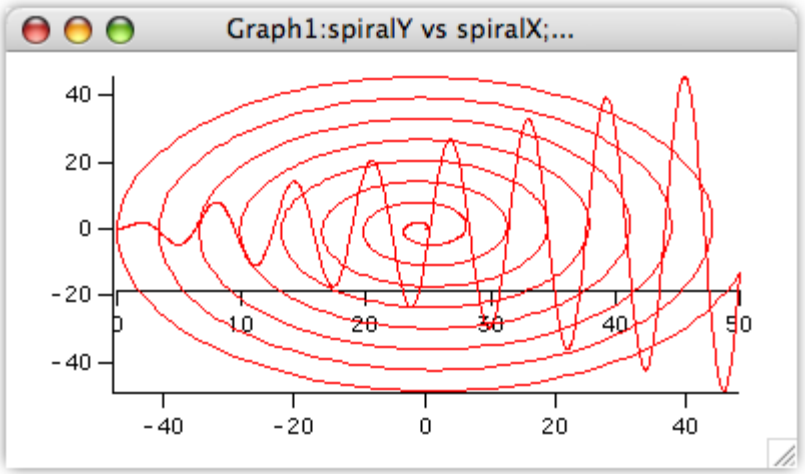
6. **In the Y Wave(s) list, again select "spiralY".**

7. **In the X Wave list, select "_calculated_".**
8. **Choose New from the Axis pop-up menu under the X Wave(s) list.**
9. **Enter "B2" in the name box.**
10. **Click OK.**

    Note the command box at the bottom of the dialog. It contains two commands: a Display command corresponding to the initial selections that you added to the lower list and an AppendToGraph command corresponding to the current selections in the Y Wave(s) and X Wave lists.

11. **Click Do It.**

    The following graph is created.



    The interior axis is called a "free" axis because it can be moved relative to the plot rectangle. We will be moving it outside of the plot area but first we must make room by adjusting the plot area margins.

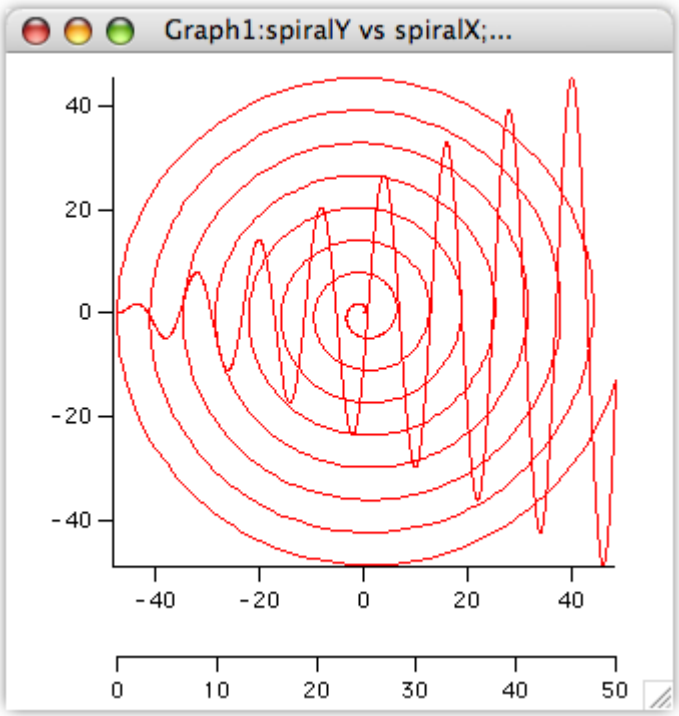
12. **Press Option (*Macintosh*) or Alt (*Windows*) and position the cursor over the bottom axis until the cursor changes to this shape: ↕.**

    This shape indicates you are over an edge of the plot area rectangle and that you can drag that edge to adjust the margin.

13. **Drag the margin up about 2 cm. Release the Option (*Macintosh*) or Alt (*Windows*).**
14. **Drag the interior axis down into the margin space you just created.**

15. **Resize the graph so the spiral is nearly circular.**

    Your graph should now look like this:



Saving Your Work

1. **Choose the File→Save Experiment As menu item.**
2. **Type "Tour #1 c.pxp" in the name box and click Save.**

   If you want to take a break, you can quit from Igor now.

Using Cursors

0. **If you are returning from a break, open your "Tour #1 c.pxp" experiment and turn off preferences.**
1. **Click in the graph and choose the Graph→Show Info menu item.**

   A cursor info panel appears below the graph.
2. **Turn on Igor Tips (*Macintosh*) or use context-sensitive help (*Windows*) to examine the info panel.**

   On Macintosh, choose Help→Show Igor Tips and let the cursor hover over an item in the info panel. When you have seen all the help, turn Igor Tips off.

   On Windows, press Shift+F1 to get the cursor and click an item in the info panel. You can see similar help information in the status bar at the bottom of the Igor Pro frame window as you let the cursor hover over an item in the info panel
3. **Control-click (*Macintosh*) or right-click (*Windows*) in the name area for graph cursor A (the round one).**
4. **Choose "spiralY" from the pop-up menu.**

   The A cursor is placed on point zero of spiralY.
5. **Repeat for cursor B but choose "spiralY#1" from the pop-up menu.**

   The wave spiralY is graphed twice. The #1 suffix is used to distinguish the second instance from the first. It is #1 rather than #2 because in Igor, indices start from zero.
6. **Position the mouse pointer over the center of the slide control bar .**

7. **Gently drag the slide bar to the right.**

   Both cursors move to increasing point numbers. They stop when one or both get to the end.

8. **Practice moving the slide bar to the left and right.**

   Notice that the cursors move with increasing speed as the bar is displaced farther from the center.

9. **Click once on the dock for cursor A (the round black circle).**

   The circle turns white.

10. **Move the slide bar to the left and right.**

    Notice that only cursor B moves.

11. **Click cursor B in the graph and drag it to another position on either trace.**

    You can also drag cursors from their docks to the graph.

12. **Click cursor A in the graph and drag it completely outside the graph.**

    The cursor is removed from the graph and returns to its dock.

13. **Choose Graph→Hide Info.**

14. **Click in the command window, type the following and press Return or Enter.**

    ```
    Print vcsr(B)
    ```

    The Y value at cursor B is printed into the history area. There are many functions available for obtaining information about cursors.

15. **Click in the graph and then drag cursor B off of the graph.**

Removing a Trace and Axis

1. **Choose the Graph→Remove from Graph menu item.**

   The Remove From Graph dialog appears with spiralY listed twice. When we created the graph we used spiralY twice, first versus spiralX to create the spiral and second versus calculated X values to show the sine wave.

2. **Click the second instance of spiralY (spiralY#1) and click Do It.**

   The sine wave and the bottom-most (free) axis are removed. An axis is removed when its last trace is removed.

3. **Drag the horizontal axis off the bottom of the window.**

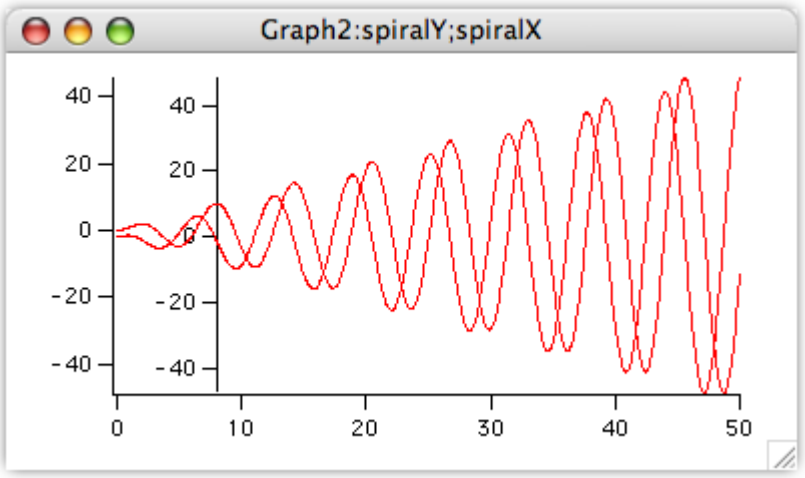
   This returns the margin setting to auto. We had set it to a fixed position when we option-dragged (*Macintosh*) or Alt-dragged (*Windows*) the margin in a previous step.

Creating a Graph with Stacked Axes

1. **Choose the Windows→New Graph menu item.**
2. **If you see a button labeled More Choices, click it.**
3. **In the Y Wave(s) list, select "spiralY".**
4. **In the X Wave list, select "_calculated_".**
5. **Click Add.**
6. **In the Y Wave(s) list, select "spiralX".**
7. **In the X Wave list, select "_calculated_".**
8. **Choose New from the Axis pop-up menu under the Y Wave(s) list.**
9. **Enter "L2" in the name box.**
10. **Click OK.**

11. **Click Do It.**

    The following graph is created.



    In the following steps we will stack the interior axis on top of the left axis.

12. **Double-click the far left axis.**

    The Modify Axis dialog appears. If any other dialog appears, cancel and try again making sure the cursor is over the axis.

13. **Click the Axis tab.**

    The Left axis should already be selected in the pop-up menu in the upper-left corner.

14. **Set the Left axis to draw between 0 and 45% of normal.**

15. **Choose L2 from the Axis pop-up menu.**

16. **Set the L2 axis to draw between 55 and 100% of normal.**

17. **In the Free Axis Position box, pop up the menu reading Distance from Margin and select Fraction of Plot Area.**

18. **Verify that the box labeled "% of Plot Area" is set to zero.**

    Steps 17 and 18 move the L2 axis so it is in line with the Left axis.

    Why don't we make this the default? Good question — positioning as percent of plot area was added in Igor Pro 6; the default maintains backward compatibility.

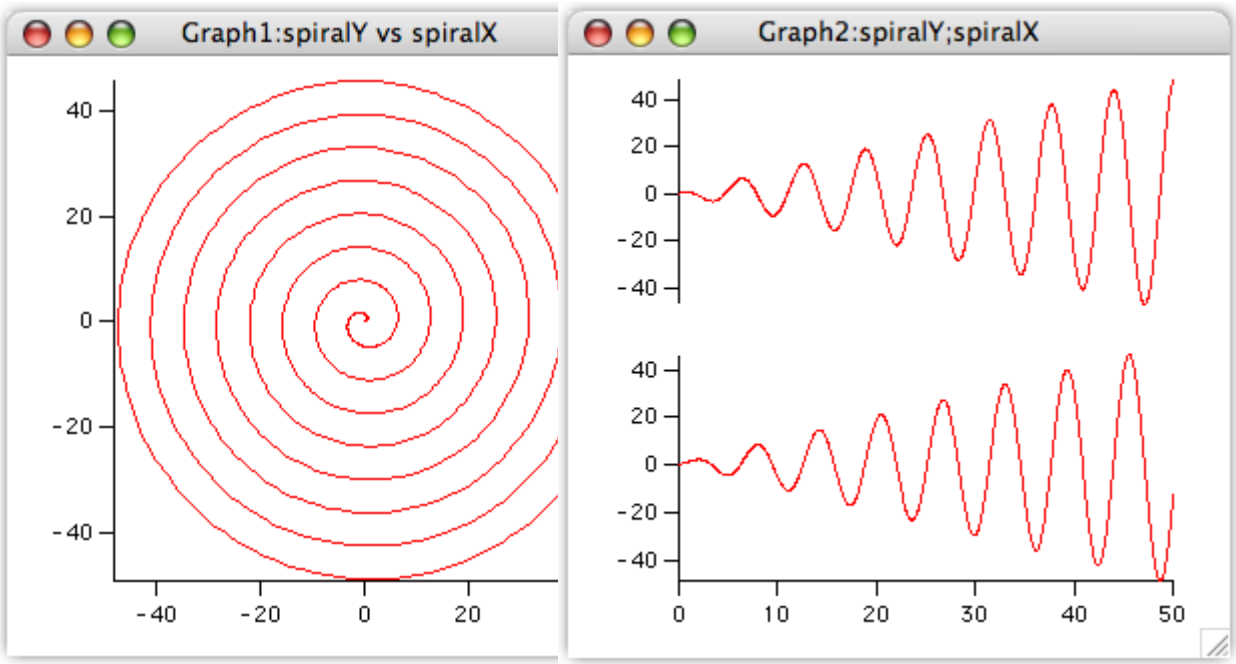
19. **Choose Bottom from the Axis pop-up menu.**

20. **Click the Axis Standoff checkbox to turn standoff off.**

21. **Click Do It.**

22. **Resize and reposition the top two graph windows so they are side-by-side and roughly square.**

23. **The graphs should look like this:**



Appending to a Layout

1. **Choose the Windows→Layouts→Layout0 menu item.**
2. **Adjust the layout window size and scrolling so you can see the blank area below the graph that is already in the layout.**
3. **Click in the graph icon, , and choose "Graph1".**

   Graph1 is added to the layout.
4. **Again, click in the graph icon and choose "Graph2".**

   Graph2 is added to the layout.
5. **Click the marquee icon .**
6. **Drag out a marquee that fills the printable space under the original graph.**
7. **Choose Layout→Arrange Objects.**

   The Arrange Objects dialog appears.
8. **Select both Graph1 and Graph2. Leave the Use Marquee checkbox checked.**
9. **Click Do It.**

   The two graphs are tiled inside the area defined by the marquee.
10. **Click in the page area outside the marquee to dismiss it.**
11. **Choose Windows→Control→Send Behind or press Control-Command-E (*Macintosh*) or Ctrl+E (*Windows*).**

Saving Your Work

1. **Choose the File→Save Experiment As menu item.**
2. **Type "Tour #1 d.pxp" in the name box and click Save.**

   If you want to take a break, you can quit Igor Pro now.

Creating Controls

This section illustrates adding controls to an Igor graph — the type of thing a programmer might want to do. If you are not interested in programming, you can skip to the **End of the General Tour** on page I-47.

0. **If you are returning from a break, open your "Tour #1 d.pxp" experiment and turn off preferences.**
1. **Click in the graph with the spiral (Graph1) to bring it to the front.**
2. **Choose the Graph→Show Tools menu item or press Command-T (*Macintosh*) or Ctrl+T (*Windows*).**

   A toolbar is displayed to the left of the graph. The second icon is selected indicating that the graph is in the drawing as opposed to normal mode.

   The selector tool (arrow) is active. It is used to create, select, move and resize controls.
3. **Choose Graph→Add Controls→Control Bar.**

   The Control Bar dialog appears.
4. **Enter a height of 30 pixels and click Do It.**

   This reserves a space at the top of the graph for controls.
5. **Click in the command line, type the following and press Return or Enter.**

   ```
   Variable ymult=1, xmult=1
   ```

   This creates two numeric variables and sets both to 1.0.
6. **Click in the graph and then choose Graph→Add Controls→Add Set Variable.**

   The SetVariable Control dialog appears.

   A SetVariable control provides a way to display and change the value of a variable.
7. **Choose ymult from the Value pop-up menu.**
8. **Enter 80 in the Width edit box.**

   This setting is back near the top of the scrolling list.
9. **Set the High Limit, Low Limit, and Increment values to 10, 0.1, and 0.1 respectively.**

   You may need to scroll down to find these settings.
10. **Click Do It.**

    A SetVariable control attached to the variable ymult appears in the upper-left of the control bar.
11. **Double-click the ymult control.**

    The SetVariable Control dialog appears.
12. **Click the Duplicate button (it's at the bottom center of the dialog).**
13. **Choose xmult as the value.**
14. **Click Do It.**

    A second SetVariable control appears in the control bar. This one is attached to the xMult variable.
15. **Choose Graph→Add Controls→Add Button.**

    The Button Control dialog appears.
16. **Enter "Update" in the Title box.**
17. **Click the New button adjacent to Procedure.**

    The Control Procedure dialog appears.
18. **Make sure the "Prefer structure-based procedures" checkbox is not selected.**
19. **Edit the procedure text so it looks like this:**

```
Function ButtonProc(ctrlName) : ButtonControl
   String ctrlName

   Wave spiralY, spiralX
   NVAR ymult, xmult
```

```
    spiralY= x*sin(ymult*x)
    spiralX= x*cos(xmult*x)
End
```

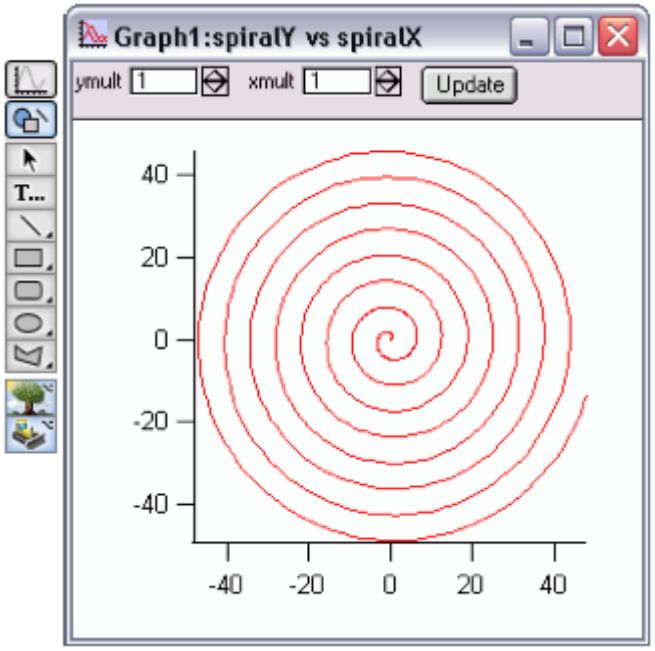
20. **Click the Save Procedure Now button.**

    The Control Procedure dialog disappears and the text you were editing is inserted into the (currently hidden) procedure window.

21. **Click Do It.**

    A Button control is added to the control bar.

    The three controls are now functional but are not esthetically arranged.



22. **Use the Arrow tool to rearrange the three controls into a more pleasing arrangement. Expand the button so it doesn't crowd the text by dragging its handles.**

23. **Click the top icon in the tool palette to enter "operate mode".**

24. **Choose Graph→Hide Tools or press Command-T (*Macintosh*) or Ctrl+T (*Windows*).**

25. **Click the up arrow in the ymult control.**

    The value changes to 1.1.

26. **Click the Update button.**

    The ButtonProc procedure that you created executes. The spiralY and spiralX waves are recalculated according to the expressions you entered in the procedure and the graphs are updated.

27. **Experiment with different ymult and xmult settings as desired.**

28. **Set both values back to 1 and click the Update button.**

    You can use Tab to select a value and then simply type "1" followed by Return or Enter.

Creating a Dependency

A dependency is a rule that relates the value of an Igor wave or variable to the values of other waves or variables. By setting up a dependency you can cause Igor to automatically update a wave when another wave or variable changes.

1. **Click in the command window to bring it to the front.**

2. **Execute the following commands in the command line:**

```
spiralY := x*sin(ymult*x)
spiralX := x*cos(xmult*x)
```

This is exactly what you entered before except here := is used in place of =. The := operator creates a dependency formula. In the first expression, the wave spiralY is made dependent on the variable ymult. If a new value is stored in ymult then the values in spiralY are automatically recalculated from the expression.

3. **Click in the graph with the spiral (Graph1) to bring it to the front.**
4. **Adjust the ymult and xmult controls but do not click the Update button.**

   When you change the value of ymult or xmult using the SetVariable control, Igor automatically executes the dependency formula. The spiralY or spiralX waves are recalculated and both graphs are updated.
5. **On the command line, execute this:**

   ```
   ymult := 3*xmult
   ```

   Note that the ymult SetVariable control as well as the graphs are updated.
6. **Adjust the xmult value.**

   Again notice that ymult as well as the graphs are updated.
7. **Choose the Misc→Object Status menu item.**

   The Object Status dialog appears. You can use this dialog to examine Igor objects that might otherwise have no visual representation such as string and numeric variables.
8. **Click the "The Current Object" pop-up menu and choose spiralY from the Dependent Objects item (*Macintosh*) or drop-down list (*Windows*).**

   The list on the right indicates that spiralY depends on the variable ymult.
9. **Double-click the ymult entry in the right hand list.**

   ymult becomes the current object. The list on the right now indicates that ymult depends on xmult.
10. **Click the Delete Formula button.**

    Now ymult no longer depends on xmult.
11. **Click Done.**
12. **Adjust the xmult setting.**

    The ymult value is no longer automatically recalculated but the spiralY and spiralX waves still are.
13. **Click the Update button.**
14. **Adjust the xmult and ymult settings.**

    The spiralY and spiralX waves are no longer automatically recalculated. This is because the ButtonProc function called by the Update button does a normal assignment using = rather than := and that action removes the dependency formulae.

    Note: In real work, you should avoid the kind of multilevel dependencies that we created here because they are too confusing.

    In fact, it is best to avoid dependencies altogether as they are hard to keep track of and debug. If a button action procedure or menu item procedure can do the job then use the procedure rather than the dependency.

Saving Your Work

1. **Choose the File→Save Experiment As menu item.**
2. **Type "Tour #1 e.pxp" in the name box and click Save.**

End of the General Tour

This is the end of the general tour of Igor Pro.

If you want to take a break, you can quit from Igor Pro now.

Guided Tour 2 - Data Analysis

In this tour we will concentrate on the data analysis features of Igor Pro. We will generate synthetic data and then manipulate it using sorting and curve fitting.

Launching Igor Pro

1. **Double-click the Igor Pro application file on your hard disk.**

   If Igor was already running, choose New Experiment from the File menu.

2. **Use the Misc menu to turn preferences off.**

Creating Synthetic Data

We need something to analyze, so we generate some random X values and create some Y data using a math function.

1. **Type the following in the command line and then press Return or Enter:**

   ```
   SetRandomSeed 0.1
   ```

   This initializes the random number generator so you will get the same results as this guided tour.

2. **Type the following in the command line and then press Return or Enter:**

   ```
   Make/N=100 fakeX=enoise(5)+5,fakeY
   ```

   This generates two 100 point waves and fills fakeX with evenly distributed random values ranging from 0 to 10.

3. **Execute this in the same way:**

   ```
   fakeY = exp(-(fakeX-4)^2)+gnoise(0.1)
   ```

   This generates a Gaussian peak centered at 4.

4. **Choose the Windows→New Graph menu item.**
5. **In the Y Wave(s) list, select "fakeY".**
6. **In the X Wave list, select "fakeX".**
7. **Click Do It.**

   The graph is a rat's nest of lines because the X values are not sorted.

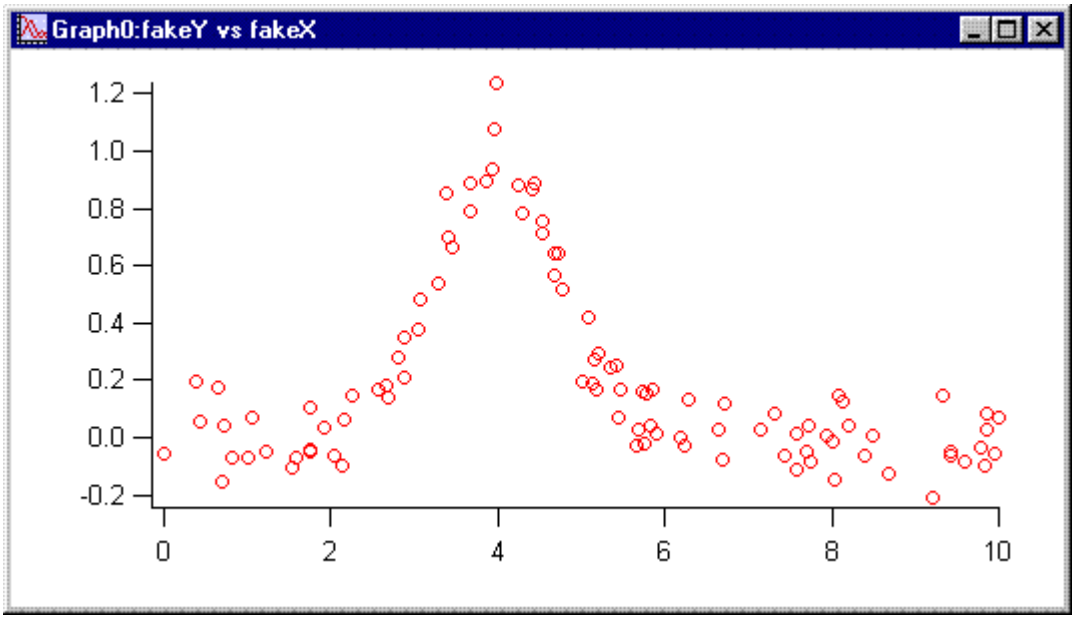
8. **Double-click the red trace.**

   The Modify Trace Appearance dialog appears.

9. **From the Mode pop-up choose Markers.**
10. **From the pop-up menu of markers choose the open circle.**

11. **Click Do It.**

    Now the graph makes sense.



Quick Curve Fit to a Gaussian

Our synthetic data was generated using a Gaussian function so let's try to extract the original parameters by fitting to a Gaussian of the form:

```
y = y0 + A*exp(-((x-x0)/width)^2)
```

Here y0, A, x0 and width are the parameters of the fit.

1. **Choose the Analysis→Quick Fit→gauss menu item.**

   Igor generated and executed a CurveFit command which you can see if you scroll up a bit in the history area of the command window. The CurveFit command performed the fit, appended a fit result trace to the graph, and reported results in the history area.

   At the bottom of the reported results we see the values found for the fit parameters. The amplitude parameter (A) should be 1.0 and the position parameter (x0) should be 4.0. We got 0.99222 ± 0.0299 for the amplitude and 3.9997 ± 0.023 for the position.

   Let's add this information to the graph.

2. **Choose Analysis→Quick Fit→Textbox Preferences.**

   The Curve Fit Textbox Preferences dialog appears.

   You can add a textbox containing curve fit results to your graph. The Curve Fit Textbox Preference dialog has a checkbox for each component of information that can be included in the textbox.

3. **Click the Display Curve Fit Info Textbox to select it and then click OK.**

   You have specified that you want an info textbox. This will affect future Quick Fit operations.

4. **Choose Analysis→Quick Fit→gauss again.**

   This time, Igor displays a textbox with the curve fit results. Once the textbox is made, it is just a textbox and you can double-click it and change it. But if you redo the fit, your changes will be lost unless you rename the textbox.

   That textbox is nice, but it's too big. Let's get rid of it.

5. **Choose Analysis→Quick Fit→Textbox Preferences again. Click the Display Curve Fit Info Textbox to deselect it. Click OK.**

6. **Choose Analysis→Quick Fit→gauss again.**

   The textbox is removed from the graph.

   You could just double-click the textbox and click Delete in the Modify Annotation dialog. The next time you do a Quick Fit you would still get the textbox unless you turn the textbox feature off.

More Curve Fitting to a Gaussian

The Quick Fit menu provides easy access to curve fitting using the built-in fit functions, with a limited set of options, to fit data displayed in a graph. You may want more options. For that you use the Curve Fitting dialog.

1. **Choose the Analysis→Curve Fitting menu item.**

   The curve fitting dialog appears.
2. **Click the Function and Data tab.**
3. **From the Function pop-up menu, choose gauss.**
4. **From the Y Data pop-up menu, choose fakeY.**
5. **From the X Data pop-up menu, choose fakeX.**
6. **Click the Data Options tab.**

   The Weighting and Data Mask pop-up menus should read "_none_".
7. **Click the Output Options tab.**

   The Destination pop-up menu should read "_auto_" and Residual should read "_none_".
8. **Click Do It.**

   During the fit a Curve Fit progress window appears. After a few passes the fit is finished and Igor waits for you to click OK in the progress window.
9. **Click OK.**

   The curve fit results are printed in the history. They are the same as in the previous section.

Sorting

In the next section we will do a curve fit to a subrange of the data. For this to work, the data must be sorted by X values.

1. **Double-click one of the open circle markers in the graph.**

   The Modify Traces Appearance dialog appears with fakeY selected. If fakeY is not selected, click it.
2. **From the Mode pop-up choose Lines between points and click Do It.**

   The fakeY trace reverts to a rat's nest of lines.
3. **Choose the Analysis→Sort menu item.**

   The Sorting dialog appears.
4. **If necessary choose Sort from the Operation pop-up menu.**
5. **Select "fakeX" in the "Key Wave" list and both "fakeX" and "fakeY" in the "Waves to Sort" list.**

   This will sort both fakeX and fakeY using fakeX as the sort key.
6. **Click Do It.**

   The rat's nest is untangled. Since we were using the lines between points mode just to show the results of the sort, we now switch back to open circles but in a new way.
7. **Press Control and click (*Macintosh*) or right-click (*Windows*) on the fakeY trace.**

   A pop-up menu appears with the name of the trace at the top. If it is not "Browse fakeY" try again.
8. **Choose Markers from the Mode item.**

Fitting to a Subrange

Here we will again fit our data to a Gaussian but using a subset of the data. We will then extrapolate the fit outside of the initial range.

1. **Choose the Graph→Show Info menu item.**

   A cursor info panel is appended to the bottom of the graph.

   Two cursors are "docked" in the info panel, Cursor A and Cursor B.

2. **Place cursor A (the round one) on the fakeY trace.**

   One way to place the cursor is to drag it to the trace. Another way is to control-click (*Macintosh*) or right-click (*Windows*) on the name area which is just to the right of the cursor icon in the cursor info panel.

   Note that the cursor A icon in the dock is now black. This indicates that cursor A is selected, meaning that it will move if you use the arrow keys on the keyboard or the slider in the cursor info panel.

3. **Move cursor A to point #14.**

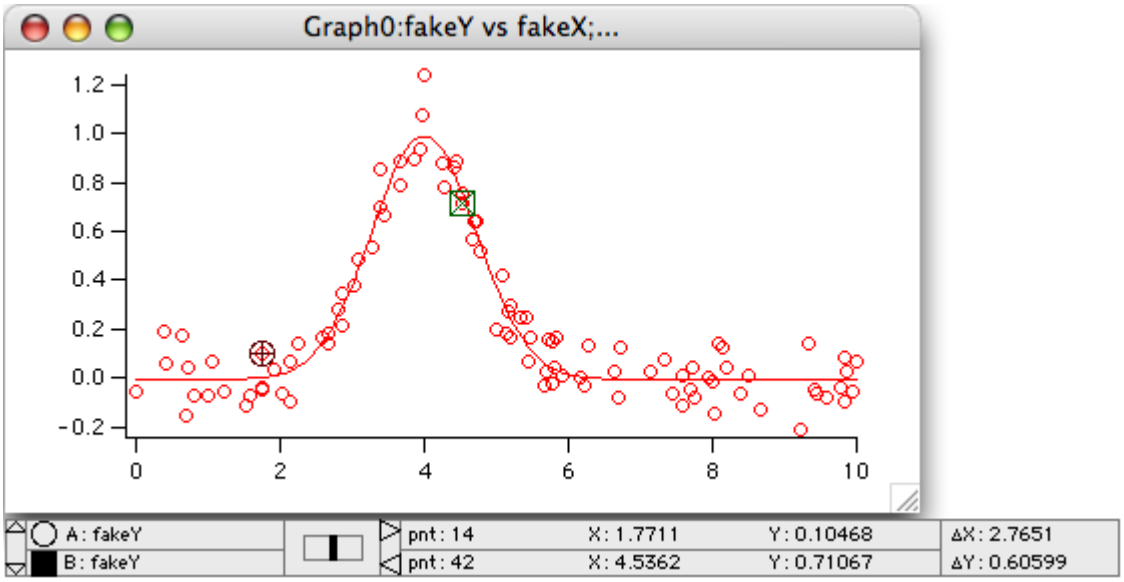
   To move the cursor one point at a time, use the arrow keys on the keyboard or click on either side of the slider in the cursor info panel.

4. **Click the dock for cursor A in the cursor info panel to deselect it.**

   This is so you can adjust cursor B without affecting the position of cursor A.

5. **Place cursor B (the square one) on the fakeY trace and move it to point #42.**

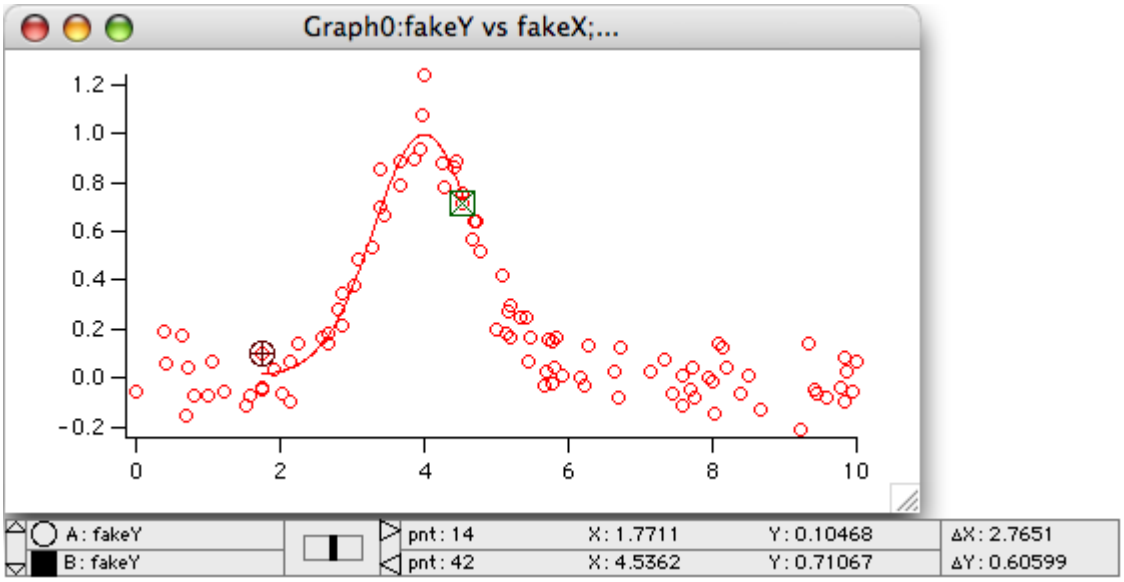
   Your graph should look like this:



6. **In the Analysis→Quick Fit menu make sure the Fit Between Cursors item is checked. If it is not, select it to check it.**

7. **Choose Analysis→Quick Fit→gauss.**

   Note that the fit curve is evaluated only over the subrange identified by the cursors.



   We would like the fit trace to extend over the entire X range, while fitting only to the data between the cursors. This is one of the options available only in the Curve Fitting dialog.

8. **Choose Analysis→Curve Fitting and then click the Function and Data tab.**

   The curve fitting dialog appears and the settings should be as you left them. Check that the function type is gauss, the X data is fakeY, the X data is fakeX.

9. **Click the Data Options tab.**

10. **Click the Cursors button in the Range area.**

    This puts the text "pcsr(A)" and "pcsr(B)" in the range entry boxes.

    pcsr is a function that returns the wave point number at the cursor position.

11. **Select the Output Options tab and click the X Range Full Width of Graph checkbox to select it.**

12. **Click Do It.**

    The curve fit starts, does a few passes and waits for you to click OK.

13. **Click OK.**

    The fit has been done using only the data between the cursors, but the fit trace extends over the entire X range.

    In the next section, we need the short version of the fit curve, so we will simply do the fit again:

14. **Choose Analysis→Quick Fit→gauss.**

Extrapolating a Fit After the Fit is Done

When you used the Quick Fit menu, and when you chose "_auto_" from the Destination pop-up menu in the curve fit dialog, Igor created a wave named fit_fakeY to show the fit results. This is called the "fit destination wave." It is just an ordinary wave whose X scaling is set to the extent of the X values used in the fit.

In the preceding sections you learned how to make the curve fit operation extrapolate the fit curve beyond the subrange. Here we show you how to do this manually to illustrate some important wave concepts.

To extrapolate, we simply change the X scaling of fit_fakeY and re-execute the fit destination wave assignment statement which the CurveFit operation put in the history area.

1. **Choose the Data→Change Wave Scaling menu item.**

2. **If you see a button labeled More Options, click it.**

3. **From the SetScale Mode pop-up menu, choose Start and End.**
4. **Double-click "fit_fakeY" in the list.**

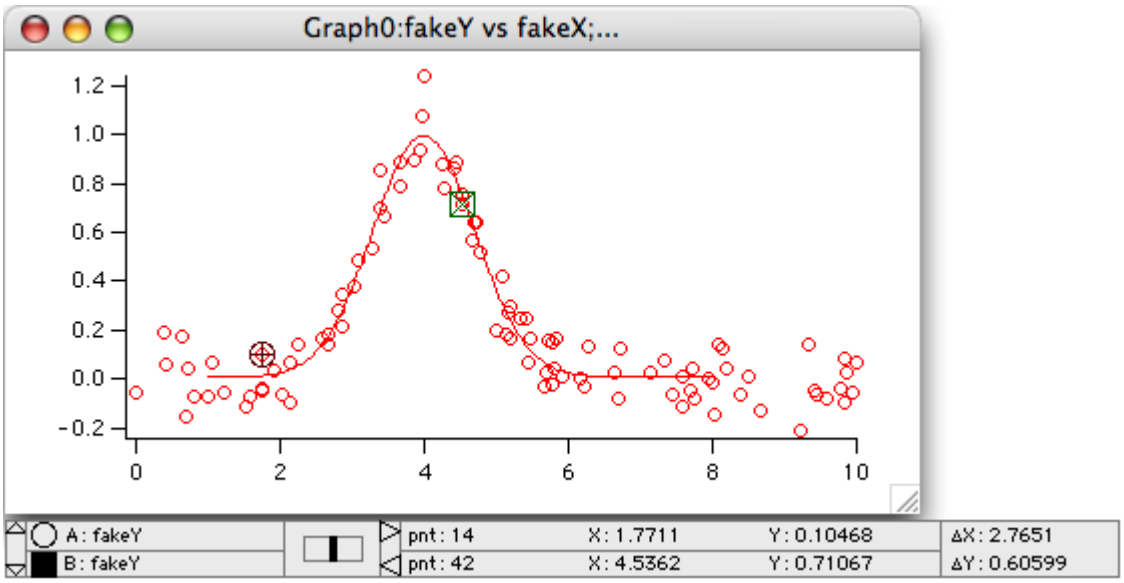
   This reads in the current X scaling values of fit_fakeY. The starting X value will be about 1.77 and the ending X will be about 4.53.
5. **Press Tab until the Start box is selected and enter 1.0.**
6. **Tab to the End box and type "8.0".**
7. **Click Do It**

   The fit_fakeY trace is stretched out and now runs between 1 and 8.

   Now we need to calculate new Y values for fit_fakeY using its new X values.
8. **In the history, find the line that starts "fit_fakeY=" and click it.**

   The entire line is selected. (The line in question is near the top of the curve fit report printed in the history.)
9. **Press Return or Enter once to copy the selection from the history to the command line and a second time to execute it.**

   The fit_fakeY wave now contains valid data between 1 and 8.



Appending a Fit

The fit trace added automatically when Igor does a curve fit uses a wave named by adding "fit_" to the start of the Y data wave's name. If you do another fit to the same Y data, that fit curve will be overwritten. If you want to show the results of several fits to the same data, you will have to somehow protect the fit destination wave from being overwritten. This is done by simply renaming it.

1. **Choose the Data→Rename menu item.**
2. **Double-click the wave named fit_fakeY to move it into the list on the right.**
3. **Edit the name in the New Name box to change the name to "gaussFit_fakeY" and click Do It.**
4. **Position the A and B cursors to point numbers 35 and 65, respectively.**

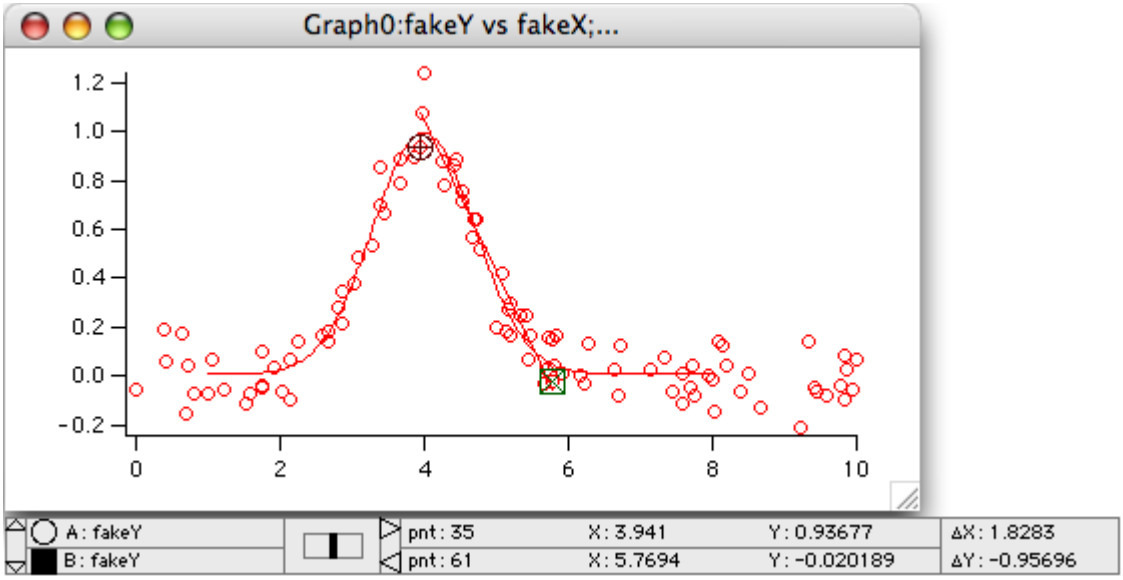
   Tip: Click in the dock for a given cursor to enable/disable its being moved by the slide control and arrow keys. Click to either side of the central slide or use the arrow keys to move the cursor one point number at a time.

5. **Choose Analysis→Quick Fit→line.**

   Because there are two traces on the graph, Quick Fit doesn't know which one to fit and puts up the Which Trace to Fit dialog.

6. **Select fakeY from the menu and click OK.**

   The curve fit is performed without displaying the fit progress window because the line fit is not iterative.



This concludes Guided Tour 2.

Guided Tour 3 - Histograms and Curve Fitting

In this tour we will explore the Histogram operation and will perform a curve fit using weighting. The optional last portion creates a residuals plot and shows you how to create a useful procedure from commands in the history.

Launching Igor Pro

1. **Double-click the Igor Pro application file on your hard disk.**

   If Igor was already running, choose New Experiment from the File menu.

2. **Use the Misc menu to turn preferences off.**

Creating Synthetic Data

We need something to analyze, so let's generate some random values.

1. **Type the following in the command line and then press Return or Enter:**

   ```
   SetRandomSeed 0.1
   ```

   This initializes the random number generator so you will get the same results as this guided tour.

2. **Type the following in the command line and then press Return or Enter:**

   ```
   Make/N=10000 fakeY= enoise(1)
   ```

   This generates a 10,000 point wave filled with evenly distributed random values from -1 to 1.

Histogram of White Noise

Here we will generate a histogram of the evenly distributed "white" noise.

1. **Choose the Analysis→Histogram menu item.**

   The Histogram dialog appears.

2. **Select fakeY from the Source Wave list.**
3. **Verify that Auto is selected in the Output Wave menu.**
4. **Select the Auto-set bin range radio button.**
5. **Set the Number of Bins box to 100.**

   Note in the command box at the bottom of the dialog there are two commands:

   ```
   Make/N=100/O fakeY_Hist;DelayUpdate
   Histogram/B=1 fakeY,fakeY_Hist
   ```

   The first command makes a wave to receive the results, the second performs the analysis. The Histogram operation in the "Auto-set bin range" mode takes the number of bins from the output wave.

6. **Click the Do It button.**

   The histogram operation is performed.

   Now we need to display the results.

7. **Choose Windows→New Graph.**
8. **Select fakeY_Hist in the Y Wave(s) list and "_calculated_" in the X list.**
9. **Click the Do It button.**

   A graph is created showing the histogram results. We need to touch it up a bit.

10. **Double-click the trace in the graph.**

    The Modify Trace Appearance dialog appears.

    "Left" is selected in the Axis pop-up menu in the top/left corner of the dialog indicating that changes made in the dialog will affect the left axis.

11. **Choose "Sticks to zero" from the Mode pop-up menu and click Do It.**

    The graph is redrawn using the new display mode.

12. **Double-click one of the tick mark labels (e.g., "100) of the left axis.**

    The Modify Axis dialog appears, showing the Axis Range tab.

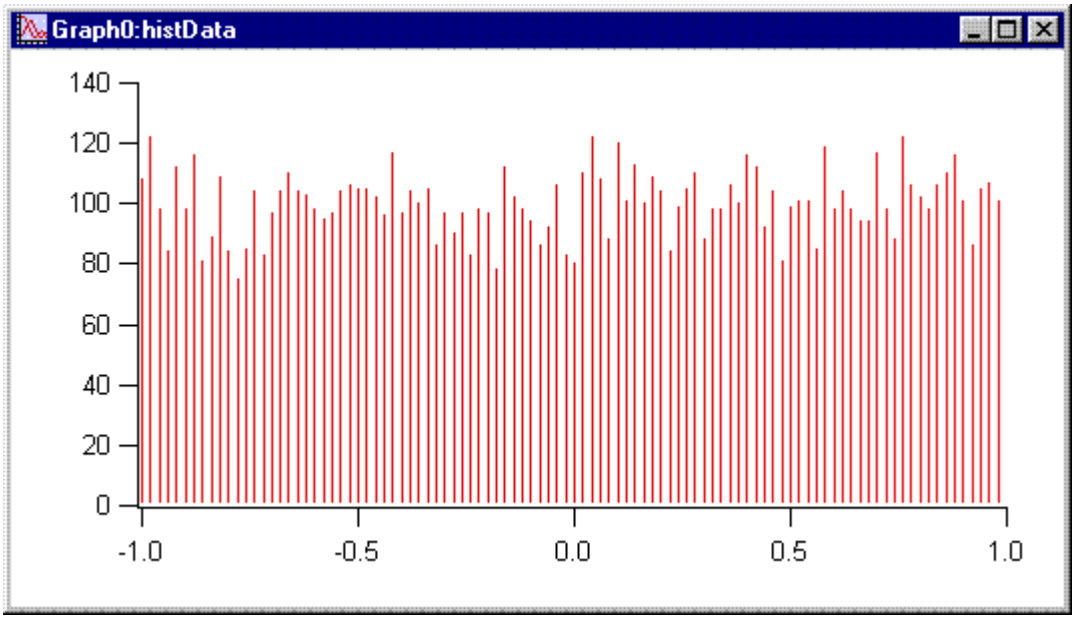
13. **From the two pop-up menus in the Autoscale Settings area, choose "Round to nice values" and "Autoscale from zero".**

14. **Choose Bottom from the Axis pop-up menu.**

15. **From the two pop-up menus in the Autoscale Settings area, choose "Round to nice values" and "Symmetric about zero".**

16. **Click the Do It button.**

    Your graph should now look like this:



Histogram of Gaussian Noise

Now we'll do another histogram, this time with Gaussian noise.

1. **Type the following in the command line and then press Return or Enter:**

   ```
   fakeY = gnoise(1)
   ```

2. **Choose the Analysis→Histogram menu item.**

   The dialog should still be correctly set up from the last time.

3. **Click the radio button labeled "Auto-set bins: 3.49*Sdev*N^-1/3".**

   The information text at the bottom of the Destination Bins box tells you that he histogram will have 48 bins.

   This is a method by Sturges for selecting a "good" number of bins for a histogram. See the **Histogram** operation on page V-245 for a reference.

4. **Select the Bin-Centered X Values checkbox.**

   By default, the Histogram operation sets the X scaling of the output wave such that the X values are at the left edge of each bin, and the right edge is given by the next X value. This makes a nice bar plot.

   In the next section you will do a curve fit to the histogram. For curve fitting you need X values that represent the center of each bin.

5. **Select the Create Square Root(N) Wave checkbox.**

   Counting data, such as a histogram, usually has Poisson-distributed values. The estimated mean of the Poisson distribution is simply the number of counts (N) and the estimated standard deviation is the square root of N.

   The curve fit will be biased if this is not taken into account. You will use this extra wave for weighting when you do the curve fit.

6. **Click the Do It button.**

   Note that the histogram output as shown in Graph0 has a Gaussian shape, as you would expect since the histogram input was noise with a Gaussian distribution.

7. **Choose Data→Data Browser.**

   The Data Browser shows you the waves and variables in your experiment. You should see three waves now: fakeY, fakeY_Hist, and W_SqrtN. FakeY_Hist contains the output of the Histogram operation and W_SqrtN is the wave created by the Histogram operation to receive the square root of N data.

8. **Click in Graph0 and then double-click the trace to bring up the Modify Trace Appearance dialog.**

9. **Select Markers from the Mode menu, then select the open circle marker.**

10. **Click the Error bars checkbox.**

    The Error Bars dialog appears.

11. **Select "+/- wave" from the Y Error Bars menu.**

12. **Pop up the Y+ menu and select W_SqrtN.**

    Note that W_SqrtN is also selected in the Y- menu. You could now select another wave from the Y- menu if you needed asymmetric error bars.

13. **Click OK, then Do It.**

Curve Fit of Histogram Data

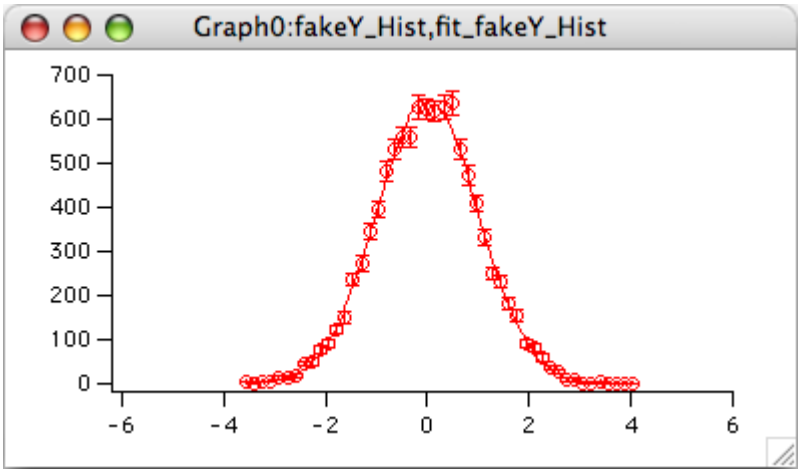
The previous section produces all the pieces required to fit a Gaussian to the histogram data, with proper weighting to account for the variance of Poisson-distributed data.

1. **Click in the graph to make sure it is the target window.**

2. **In the Analysis→Quick Fit menu make sure the Weight from Error Bar Wave item is checked. If it is not, select it to check it.**

3. **Choose Analysis→Quick Fit→gauss.**

   With all the changes you've made, by now the graph looks like this:



   As shown in the history area, the fit results are:

```
Coefficient values ± one standard deviation
   y0    =-0.35284 ± 0.513
   A     =644.85 ± 7.99
   x0    =-0.0014111 ± 0.00997
   width=1.406 ± 0.0118
```

The original data was made with a standard deviation of 1. Why is the width 1.406? The way Igor defines its gauss fit function, width is sigma*$2^{1/2}$.

4. **Enter this command in the command line:**

```
Print 1.406/sqrt(2)
```

The result, 0.994192, is pretty close to 1.0.

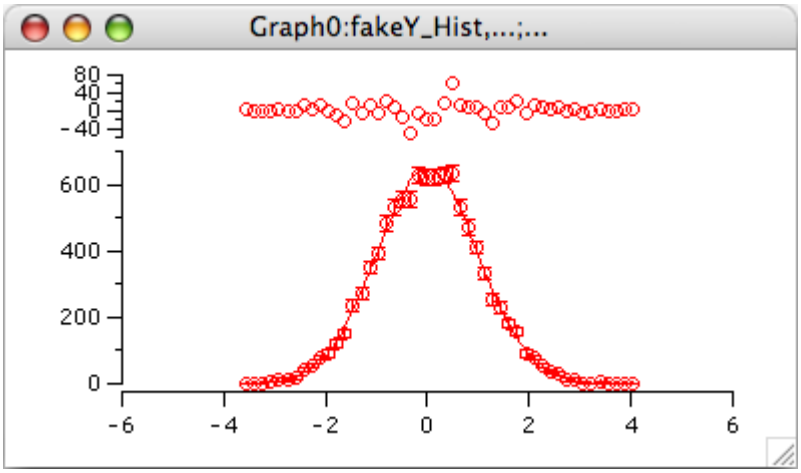
It is often useful to plot the residuals from a fit to check for various kinds of problems. For that you need to use the Curve Fit dialog.

5. **Choose Analysis→Curve Fitting.**
6. **Click the Function and Data tab and choose gauss from the Function menu.**
7. **Choose fakeY_Hist (not fakeY) from the Y Data menu.**
8. **Leave the X Data pop-up menu set to "_calculated_".**
9. **Click the Data Options tab. If there is text in the Start or End Range boxes, click the Clear button in the Range section.**
10. **Choose W_SqrtN from the Weighting pop-up menu.**
11. **Just under the Weighting pop-up menu there are two radio buttons. Click the top one which is labeled "Standard Deviation".**
12. **Click the Output Options tab and choose "_auto_" from the Destination pop-up menu.**
13. **Set the Residual pop-up menu to "_auto trace_".**

Residuals will be calculated automatically and added to the curve fit in our graph.

14. **Click Do It.**

The curve fit starts, does a few passes, and waits for you to click OK.



There is one small issue not addressed above. One of the bins contains zero; the square root of zero is, of course, zero. So the weighting wave contains a zero, which causes the curve fit to ignore that data point. It's not clear what is the best approach to fixing that problem. Some replace the zero with a one. These commands replace any zeroes in the weighting wave and re-do the fit:

```
W_SqrtN = W_SqrtN[p] == 0 ? 1 : W_SqrtN[p]
CurveFit/NTHR=0 gauss  fakeY_Hist /W=W_SqrtN /I=1 /D /R
```

This doesn't change the result very much, since there was just one zero in the histogram:

```
Coefficient values ± one standard deviation
   y0     =-0.40357 ± 0.464
   A      =644.76 ± 7.98
```

```
x0     =-0.0014186 ± 0.00996
width  =1.4065 ± 0.0115
```

Curve Fit Residuals (Optional)

This section and the next one are primarily of interest to people who want to use Igor programming to automate tasks.

In the next section, as an illustration of how the history area can be used as a source of commands to generate procedures, we will create a procedure that appends residuals to a graph. The preceding section illustrated that Igor is able to automatically display residuals from a curve fit, so the procedure that we write in the next section is not needed. Still, it demonstrates the process of creating a procedure. In preparation for writing the procedure, in this section we append the residuals manually.

If the curve fit to a Gaussian function went well and if the gnoise function truly produces noise with a Gaussian distribution, then a plot of the difference between the histogram data and the fitted function should not reveal any curvature.

0. **To remove the automatically generated residual from the Gaussian fit in the previous section, Control-click (*Macintosh*) or right-click (*Windows*) directly on the residual trace at the top of the graph and select Remove Res_fakeY_Hist from the pop-up menu.**
1. **Choose the Data→Duplicate Waves menu item.**
2. **Choose fakeY_Hist from the Template pop-up menu.**
3. **In the first Names box, enter "histResids".**
4. **Click Do It.**

   You now have a wave suitable for containing residuals.
5. **In the history area of the command window, find the line that reads (all on one line):**

   ```
   fit_fakeY_Hist= W_coef[0] +
      W_coef[1]*exp(-((x-W_coef[2])/W_coef[3])^2)
   ```

   W_coef is a wave created by the CurveFit operation to contain the fit parameters. W_coef[0] is the y0 parameter, W_coef[1] is the A parameter, W_coef[2] is the x0 parameter and W_coef[3] is the width parameter.

   This line shows conceptually what the CurveFit operation did to set the data values of the fit destination wave.
6. **Click once on the line to select it and then press Return or Enter once.**

   The line is transferred to the command line.
7. **Edit the line to match the following (all on one line):**

   ```
   histResids= fakeY_Hist -
      ( W_coef[0]+W_coef[1]*exp(-((x-W_coef[2])/W_coef[3])^2) )
   ```

   In other words, change "fit_fakeY_Hist" to "histResids", click after the equals and type "fakeY_Hist - (" and then add a ")" to the end of the line.

   The expression inside the parentheses that you added represents the model value using the parameters determined by the fit. This command computes residuals by subtracting the model values from the data values on which the fit was performed.

   Note: If the fit had used an X wave rather than calculated X values then it would have been necessary to substitute the name of the X wave for the "x" in the expression.
8. **Press Return or Enter.**

   This wave assignment statement calculates the difference between the measured data (the output of the Histogram operation) and the theoretical Gaussian (as determined by the CurveFit operation).

   Now we will append the residuals to the graph stacked above the current contents.
9. **Choose Graph→Append Traces to Graph.**
10. **Select histResids from the Y wave(s) list and "_calculated_" from the X wave list.**

11. **Choose New from the Axis pop-up menu under the Y Wave(s) list.**
12. **Enter "Lresid" in the Name box and click OK.**
13. **Click Do It.**

    The new trace and axis is added.

    Now we need to arrange the axes. We will do this by partioning the available space between the Left and Lresid axes.
14. **Double-click the far-left axis.**

    The Modify Axis dialog appears. If any other dialog appears, cancel and try again making sure the cursor is over the axis.

    If you have enough screen space you will be able to see the graph change as you change settings in the dialog. Make sure that the Live Update checkbox in the top/right corner of the dialog is selected.
15. **Click the Axis tab.**

    The Left axis should already be selected in the pop-up menu in the top-left corner of the dialog.
16. **Set the Left axis to draw between 0 and 70% of normal.**
17. **Choose Lresid from the Axis pop-up menu.**
18. **Set the Lresid axis to draw between 80 and 100% of normal.**
19. **Choose Fraction of Plot Area in the "Free axis position" menu.**

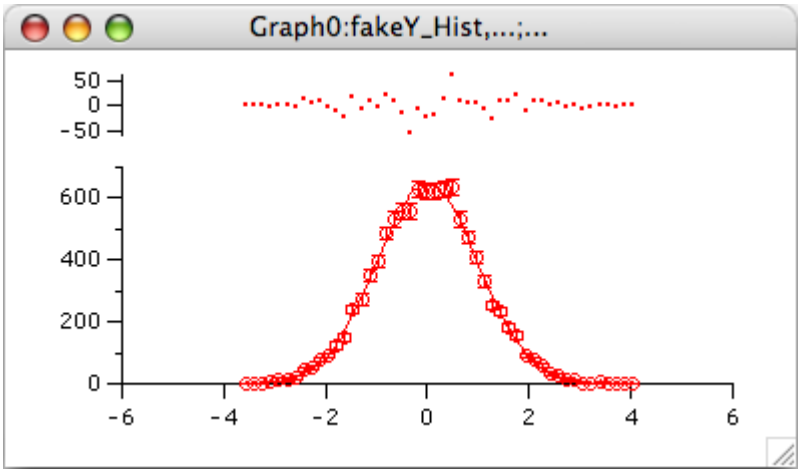
    The Lresid axis is a "free" axis. This moves it horizontally so it is in line with the Left axis.
20. **Choose Bottom from the Axis pop-up menu.**
21. **Click the Axis Standoff checkbox to turn standoff off.**

    Just a couple more touch-ups and we will be done. The ticking of the Lresid axis can be improved. The residual data should be in dots mode.
22. **Choose Lresid from the Axis pop-up menu again.**
23. **Click the Auto/Man Ticks tab.**
24. **Change the Approximately value to 2.**
25. **Click the Axis Range tab.**
26. **In the Autoscale Settings area, choose "Symmetric about zero" from the menu currently reading "Zero isn't special".**
27. **Click the Do It button.**
28. **Double-click the histResids trace.**

    The Modify Trace Appearance dialog appears with histResids already selected in the list.
29. **Choose Dots from the Mode pop-up menu**
30. **Set the line size to 2.00.**

31. **Click Do It.**

    Your graph should now look like this:



Writing a Procedure (Optional)

In this section we will collect commands that were created as we appended the residuals to the graph. We will now use them to create a procedure that will append a plot of residuals to a graph.

1. **Click the zoom button (*Macintosh*) or the maximize button (*Windows*) of the command window to enlarge it to fill the screen.**
2. **Find the fifth line from the bottom that reads:**

   `•AppendToGraph/L=Lresid histResids`
3. **Select this line and all the lines below it and press Command-C (*Macintosh*) or Ctrl+C (*Windows*) to copy them to the Clipboard.**
4. **Click the zoom button (*Macintosh*) or the restore button (*Windows*) of the command window to return it to its normal size.**
5. **Choose the Windows→New→Procedure menu item.**
6. **Type "Append Residuals" (without the quotes) in the Document Name box and click New.**

   A new procedure window appears. We could have used the always-present built-in procedure window, but we will save this procedure window as a stand-alone file.
7. **Add a blank line to the window, type "Function AppendResiduals()", and press Return or Enter.**
8. **Press Command-V (*Macintosh*) or Ctrl+V (*Windows*) to paste the commands from the history into the new window.**
9. **Type "End" and press Return or Enter.**
10. **Select the five lines that you pasted into the procedure window and then choose Edit→Adjust Indentation.**

    This removes the bullet characters from the history and prepends tabs to apply the normal indentation for procedures.

    If you are running on an Asian-language system, you will have asterisks at the start of each line and you must remove them manually.

    Your procedure should now look like this:

```
Function AppendResiduals()
    AppendToGraph/L=Lresid histResids
    ModifyGraph nticks(Lresid)=2,standoff(bottom)=0,
                            axisEnab(left)={0,0.7};DelayUpdate
    ModifyGraph axisEnab(Lresid)={0.8,1},
                            freePos(Lresid)=0;DelayUpdate
    SetAxis/A/E=2 Lresid
```

```
    ModifyGraph mode(histResids)=2,lsize(histResids)=2
End
```

(Some commands in this manual are shown on two lines to fit them on the page but in reality will be on one line. In Igor, a command must be entered on a single line of text.)

11. Delete the ";DelayUpdate" at the end of the two ModifyGraph commands.

DelayUpdate has no effect in a function.

We now have a nearly functional procedure but with a major limitation — it only works if the residuals wave is named "histResids". In the following steps, we will change the function so that it can be used with any wave and also with an XY pair, rather than just with equally-spaced waveform data.

12. Convert the first two lines of the function to match the following:

```
Function AppendResiduals(ywave,xwave)
    String ywave,xwave

    if (CmpStr("_calculated_",xwave) == 0)
        AppendToGraph/L=Lresid $ywave
    else
        AppendToGraph/L=Lresid $ywave vs $xwave
    endif
```

13. In the last ModifyGraph command in the function, change both "histResids" to "$ywave".

The "$" character converts the string expression that follows it into the name of an Igor object (see **String Substitution Using $** on page IV-15 for details).

Here is the completed procedure.

```
Function AppendResiduals(ywave,xwave)
    String ywave,xwave

    if (CmpStr("_calculated_",xwave) == 0)
        AppendToGraph/L=Lresid $ywave
    else
        AppendToGraph/L=Lresid $ywave vs $xwave
    endif
    ModifyGraph nticks(Lresid)=2,standoff(bottom)=0,
                                    axisEnab(left)={0,0.7}
    ModifyGraph axisEnab(Lresid)={0.8,1},freePos(Lresid)=0
    SetAxis/A/E=2 Lresid
    ModifyGraph mode($ywave)=2,lsize($ywave)=2
End
```

Let's try it out.

14. Click the Compile button at the bottom of the procedure window to compile the function.

If you get an error, edit the function text to match the listing above.

15. Click the close button in the Append Residuals procedure window. A dialog will ask if you want to kill or hide the window. Click Hide.

If you press Shift while clicking the close button, the window will be hidden without a dialog. (Use the Help→Shortcuts menu to learn about this and other shortcuts.)

16. Choose Windows→New Graph.

17. Choose fakeY_Hist from the Y Wave(s) list and _calculated_ from the X Wave list and click Do It.

A graph without residuals is created.

18. In the command line, execute the following command:

```
AppendResiduals("histResids", "_calculated_")
```

The AppendResiduals function runs and displays the residuals in the graph, above the original histogram data.

Next, we will add a function that displays a dialog so we don't have to type wave names into the command line.

19. Use the Windows→Procedure Windows menu to show the Append Residuals procedure window.

20. **Enter the following function after the AppendResiduals function.**

```
Function AppendResidualsDialog()
   String ywave,xwave

   Prompt ywave,"Residuals Data",popup WaveList("*",";","")
   Prompt xwave,"X Data",
               popup "_calculated_;"+WaveList("*",";","")
   DoPrompt "Append Residuals", ywave, xwave
   if (V_flag != 0)
      return -1;                    // User canceled.
   endif

   AppendResiduals(ywave,xwave)
End
```

The second Prompt statement is shown above on two lines but must be entered on one line in Igor.

This function will display a dialog to get parameters from the user and will then call the AppendResiduals function.

Let's try it out.

21. **Click the Compile button at the bottom of the procedure window to compile the function.**

If you get an error, edit the function text to match the listing above.

22. **Shift-click the close button to hide the procedure window. Then activate the graph.**

23. **Control-click (*Macintosh*) or right-click (*Windows*) on the residual trace at the top of the graph and select Remove histResids from the pop-up menu.**

The axis displaying histData will stay short because the residuals were not appended to the graph automatically.

24. **On the command line, execute the following command:**

```
AppendResidualsDialog()
```

The AppendResidualsDialog function displays a dialog to let you choose parameters.

25. **Choose histResids from the Residuals Data pop-up menu.**

26. **Leave the X Wave pop-up set to "_calculated_".**

27. **Click Continue.**

The graph should once again contain the residuals plotted on a new axis above the main data.

Next we will add a menu item to the Macros menu.

28. **Use the Windows→Procedure Windows menu to open the Append Residuals procedure window.**

29. **Enter the following code before the AppendResiduals function:**

```
Menu "Macros"
   "Append Residuals...", AppendResidualsDialog()
End
```

30. **Click the Compile button.**

Igor compiles the function and adds the menu item to the Macros menu.

31. **Press Control-Command-E (*Macintosh*) or Ctrl+E (*Windows*) to send the procedure window to the back, and then activate the graph.**

32. **Control-click (*Macintosh*) or right-click (*Windows*) on the residual trace at the top of the graph and select "Remove histResids" from the pop-up menu.**

33. **Click the Macros menu and choose the "Append Residuals" item**

The procedure displays a dialog to let you choose parameters.

34. **Choose histResids from the Residuals Data pop-up menu.**

35. **Leave the X Wave pop-up menu set to "_calculated_".**

36. **Click the Continue button.**

    The graph should once again contain the residuals plotted on a new axis above the main data.

Saving a Procedure File (Optional)

Note: *If you are using the demo version of Igor Pro beyond the 30-day trial period, you cannot save a procedure file.*

Now that we have a working procedure, let's save it so it can be used in the future. We will save the file in the "Igor Pro User Files" folder - a folder created by Igor for you to store your Igor files.

1. **Choose Help→Show Igor Pro User Files.**

   Igor opens the "Igor Pro User Files" folder on the desktop.

   By default, this folder has the Igor Pro major version number in its name, for example, "Igor Pro 6 User Files", but it is generically called the "Igor Pro User Files" folder.

   Note where in the file system hierarchy this folder is located as you will need to know this in a subsequent step. The default locations are:

   Macintosh:

   ```
   /Users/<user>/Documents/WaveMetrics/Igor Pro 6 User Files
   ```

   Windows XP:

   ```
   C:\Documents and Settings\<user>\My Documents\WaveMetrics\Igor Pro 6 User Files
   ```

   Windows VISTA and Windows 7:

   ```
   C:\Users\<user>\My Documents\WaveMetrics\Igor Pro 6 User Files
   ```

   We will save the procedure file in the "User Procedures" subfolder of the Igor Pro User Files folder. You could save the file anywhere on your hard disk, but saving in the User Procedures subfolder makes it easier to access the file as we will see in the next section.

2. **Back in Igor, activate the Append Residuals procedure window again.**

3. **Choose the File→Save Procedure As menu item.**

4. **Enter the file name "Append Residuals.ipf".**

5. **Navigate to your Shared Procedures folder and click Save.**

   The Append Residuals procedure file is now saved in a stand-alone file.

6. **Click the close button on the procedure window.**

   Igor will ask if you want to kill or hide the file. Click Kill. This removes the file from the current experiment, but it still exists on disk and you can open it as needed.

   There are several ways to open the procedure file to use it in the future. One is to double-click it. Another is to choose the File→Open File→Procedure menu item. A third is to put a `#include` statement in the built-in procedure window, which is how we will open it in the next section.

Including a Procedure File (Optional)

The preferred way to open a procedure window that you intend to use from many different experiments is to use a #include statement. This section demonstrates how to do that.

Note: If you are using the demo version of Igor Pro beyond the 30-day trial period, you did not create the Append Residuals.ipf file in the preceding section so you can't do this section. See **The Include Statement** on page IV-145 for details about including procedure files.

1. **In Igor, use the Windows→Procedure Windows menu to open the built-in procedure window.**

2. **At the top of the built-in procedure window, notice the line that says:**

   ```
   #pragma rtGlobals = 1
   ```

   This is technical stuff that you can ignore.

3. **Under the rtGlobals line, leave a blank line and then enter:**

   ```
   #include "Append Residuals"
   ```

4. **Click the Compile button at the bottom of the built-in procedure window.**

   Igor compiles the procedure window. When it sees the `#include` statement, it looks for the Append Residuals.ipf procedure file in the User Procedures folder and opens it. You don't see it because it was opened hidden.

5. **Use the Windows→Procedure Windows menu to verify that the Append Residuals procedure file is in fact open.**

   To remove the procedure file from the experiment, you would remove the `#include` statement from the built-in procedure window.

   #include is powerful because it allows procedure files to include other procedure files in a chain. Each procedure file automatically opens any other procedure files it needs.

   User Procedures is special because Igor searches it to satisfy #include statements.

   Another special folder is Igor Procedures. Any procedure file in Igor Procedures is automatically opened by Igor at launch time and left open till Igor quits. This is the place to put procedure files that you want to be open all of the time.

This concludes Guided Tour 3.

For Further Exploration

We developed the guided tours in this chapter to provide an overview of the basics of using Igor Pro and to give you some experience using features that you will likely need for your day-to-day work. Beyond these fundamentals, Igor includes a wide variety of features to facilitate much more advanced graphing and analysis of your data.

As you become more familiar with using Igor, you will want to further explore some of the additional learning and informational aids that we have included with Igor Pro.

- The Igor Pro manual is installed on your hard disk in PDF format. You can access it through the Igor Help Browser or open it directly from the Manual folder in the Igor Pro Folder.

  The material in the manual is the same as the material in the online help files but is organized in book format and is therefore better suited for linear reading. Unlike the help files, the PDF manual includes an index. You may want to print selected chapters. You can purchase hard copy of the Igor Pro manual from **<http://www.lulu.com/wavemetrics>**.

  The most important chapters at this point in your Igor learning curve are Chapter II-3, **Experiments, Files and Folders** and Chapter II-5, **Waves**. If you want to learn Igor programming, read Chapter IV-1, **Working with Commands**, Chapter IV-2, **Programming Overview**, and Chapter IV-3, **User-Defined Functions**.
- The Igor Help Browser provides online help, including reference material for all built-in operations and functions, an extensive list of shortcuts, and the ability to search Igor help files, procedure files, examples and technical notes for key phrases. See **Igor Help Browser** on page II-6 for more information.
- The Examples folder contains a wide variety of sample experiments illustrating many of Igor's advanced graphing and programming facilities. You can access these most easily through the File→Example Experiments submenus.
- The Learning Aids folder contains additional guided tours and tutorials including a tutorial on image processing. You can access these through the File→Example Experiments→Tutorials submenus.
- A tutorial on 3D visualization can be found in the Visualization help window, accessible through the Windows→Help Windows menu.
- The More Help Files folder contains several supplementary help files. Use the File→Open File→Help Files menu item to open them.
- The WaveMetrics Procedures folder contains a number of utility procedures that you may find useful for writing your own procedures and for your more advanced graphing requirements. For an overview of the WaveMetrics procedures and easy loading of the procedure files, choose Windows→Help Windows→WM Procedures Index.
- The More Extensions folder contains a number of External Operations (XOPs), which add functionality not built into the Igor Pro application. Read the included help files to find out more about the individual XOPs and how to install them, or consult the External Operations Index in the XOP Index help file, which has brief description of each XOP.
- The Technical Notes folder contains miscellaneous additional information and services. Tech Note #000 contains an index to all of the other notes.
- The Igor Pro mailing list is an Internet mailing list where Igor users share ideas and help each other. See **Igor Mailing List** on page II-16 or select the Help→Support menu and then select Igor Mailing List from the Support Options list for information about the mailing list.
- **IgorExchange** is a user-to-user support web page and a repository for user-created Igor Pro projects. Choose Help→IgorExchange to visit it.

Volume II User's Guide: Part 1

Table of Contents

Chapter
II-1

Getting Help

Overview

There are a number of sources of information on using Igor:

- The Igor Pro online manual
- The online help system
- WaveMetrics support web page
- WaveMetrics Technical Support
- The Igor mailing list
- The IgorExchange user-to-user support web page

Online Manual

The Igor Pro installer installs the entire Igor Pro manual as an Adobe PDF (portable document format) file. You need Adobe Reader or a comparable PDF viewer, such as Apple's Preview, to view the online manual.

From within Igor Pro you can launch your PDF viewer program and view the online manual by choosing Help→Manual. From the desktop you can view the manual by double-clicking the IgorMan.pdf file in "Igor Pro Folder/Manual".

The PDF manual includes a fast-search index for Acrobat Reader version 6 or later. To activate the fast search index, open the IgorMan.pdf file in Acrobat Reader 6 or later, choose Edit→Search, and select the IgorMan.pdx file.

The PDF manual is available in hard-copy form from http://www.lulu.com/wavemetrics.

WaveMetrics Support Web Page

For up-to-date information on Igor Pro, visit the WaveMetrics support Web page at:

```
http://www.wavemetrics.com/support/
```

From this web page you can search our support database, search archives of the Igor Mailing List and find links to updaters.

You can access this page by choosing Help→Support Web Page.

Online Help

Igor provides several forms of online help:

- The Igor help system
- Macintosh Igor Tips
- Windows status line help, context-sensitive help, and tool tips

The Igor help system is the same on Macintosh and Windows. Its major elements are Igor help files and the Igor Help Browser.

This table summarizes the ways to access online help in Igor.

Help Access Method	What It Is Good For
The Igor Help Browser window (Help menu)	Finding a specific topic or subtopic in the Igor help files, learning about handy shortcuts, getting help for operations, functions and programming keywords, and searching Igor files for specific phrases.
The Help button that appears in many dialogs	Getting a general idea of how to use the dialog.

Templates in procedure windows*	Getting the syntax of built-in and external functions and operations and flow-control structures.
Igor Tips (*Macintosh only*)	Clarifying the meaning of icons, menu items and dialog items. Identifying traces in graphs and columns in tables.
Status line help (*Windows only*)	Displaying a brief description of menu items and tools.
Tool tips (*Windows only*)	Displaying a short description of a button or tool.
Context-sensitive help (*Windows only*)	Getting more detail than is available from the status line help.

* Chapter III-13, **Procedure Windows** describes the use of templates in procedure windows. This chapter covers the other help access methods listed in the table.

The main sources of information for help are the files in the Igor Help Files folder. Igor automatically opens files in this folder when it starts up.

Igor also automatically opens help files in "Igor Pro User Files/Igor Help Files" (see **Igor Pro User Files** on page II-46 for details). If you want an additional help file to be automatically opened, put it or an alias/shortcut for it in that folder.

Addtional WaveMetrics help files can be found in "Igor Pro Folder/More Help Files". You can open these help files by double-clicking them or using the Open File submenu in the File menu. You can search these files (and all help files in the Igor Pro Folder and in the Igor Pro User Files folder) using the Search Igor Files tab of the Igor Help Browser.

Many Igor extensions come with help files describing their use. These help files are stored in the same folder as the extension itself — in either "Igor Pro Folder/Igor Extensions" or "Igor Pro Folder/More Extensions".

Igor's help system is extensible. You can write your own help files and add balloons help or context-sensitive help for your own menu items and controls. This is something that you might want to do if you write Igor procedures to be used by others.

Igor Tips (*Macintosh*)

We've tried to provide concise yet useful tips for nearly every menu item, dialog item and icon in Igor. There are two ways to show the Igor Tips window:

- Choose Help→Show Igor Tips.
- Press Option-Help.

If your keyboard lacks a Help key you must use the Help menu.

Once you've turned Igor Tips on, position the cursor over a menu item, dialog item or icon. In most cases, Igor will present a window that will explain what that item is good for.

Note: Pressing Option-Help toggles Igor Tips off or on.

You can also use Igor Tips to get information about traces in graphs and columns in tables. However, these Igor Tips appear *only* while you press Command-Option-Control and click a trace or column. Showing Igor Tips with the Help menu will not do it.

User-Defined Igor Tips

You can define tips for menu items and controls created by your Igor procedures. See **Help for User Menus** on page IV-108 and **Help Text for User-Defined Controls** on page III-382.

Status Line Help, Tool Tips and Context-Sensitive Help (*Windows*)

On Windows Igor provides three ways to get help for icons, menu items and dialog items. These are status line help, tool tips, and context-sensitive help.

Status Line Help

The status line area at the bottom of the main Igor Pro window shows brief descriptions of icons and menu items. This help is shown automatically; you don't have to do anything to make it appear.

Tool Tips

If you point at an icon in an Igor Pro window, a tool tip will appear after a short delay. It contains just a two or three word description of the button.

You can adjust the delay before the tool tip appears, and the duration of display in the Help page of the Miscellaneous Settings dialog, which you can choose from the Misc menu.

Context-Sensitive Help

Context-sensitive help (sometimes referred to as F1 help) is displayed in a pop-up window and provides more detail than the status line help. It is accessed in several different ways depending on the type of item you need help for.

Menus

For help on items in a menu, pull down the menu and highlight the item of interest. Then press the F1 key to display the context-sensitive help window.

Icons

For help on icons in windows such as graphs and tables, hold down Shift and press F1. This changes the mouse cursor to a question-mark; click on a button or icon to display the context-sensitive help window.

Dialogs

Help for individual items in Igor's dialogs can be summoned by clicking the question-mark button at the right end of the dialog's title bar, then clicking on the item for which you want help.

Igor Shortcuts Help

Igor supports a number of very handy shortcuts. The Shortcuts tab of the Igor Help Browser lists these shortcuts, organized in logical categories.

Help from a Procedure Window or the Command Line

There is a quick and easy way to get help for a function, operation or flow-control keyword from the command line or from a procedure, notebook or help window. Type or select the name or keyword. Control-click (*Macintosh*) or right-click (*Windows*) and choose help from the resulting menu.

Here are some keyboard shortcuts for summoning help.

Keyboard Shortcut		**What It Does**
Macintosh	***Windows***	
Press Help	Press F1	Displays Help Browser window
Press Shift-Help	Press Ctrl+F1	Inserts template for selected function, operation or flow-control keyword
Press Shift-Option-Help	Press Ctrl+Alt+F1	Shows help for selected function, operation or flow-control keyword

The Help Button in Dialogs

The Help button in Igor dialog's provides an overview of the dialog and tips for using it. Use Igor Tips (*Macintosh*) or context-sensitive help (*Windows*) for information on individual dialog items.

Igor Help Browser

The Igor Help Browser is designed to provide quick access to the most frequently-used Igor reference material and also to provide a starting point in searching for other kinds of information. You can display the Igor Help Browser by

- Choosing Igor Help Browser from the Help menu.
- Pressing the Help key (*Macintosh*) or F1 key (*Windows*).
- Clicking the Igor Help Browser icon in the command window.

The Igor Help Browser consists of six tabs.

Help Topics Tab

The Help Topics tab provides a table of contents for the open Igor help files. When you first launch Igor, Igor opens the help files in "Igor Pro Folder/Igor Help Files" and "Igor Pro User Files/Igor Help Files".

The Topics list initially presents all topics in all open help files. You can choose a specific help file from the Show Topics From pop-up menu to narrow the scope of topics. Once you locate and select the topic of interest in the Topics list, the Subtopics list displays subtopics within that topic, if any exist.

After selecting a topic and optionally a subtopic, click the Show Selected Topic button to see the help.

If you know that the information that you are looking for is in a help file that is not normally open, for example, in a help file associated with an Igor extension, click the Open Another Help File button. Most additional help files can be found in "Igor Pro Folder/More Help Files", "Igor Pro Folder/Igor Extensions" or "Igor Pro Folder/More Extensions".

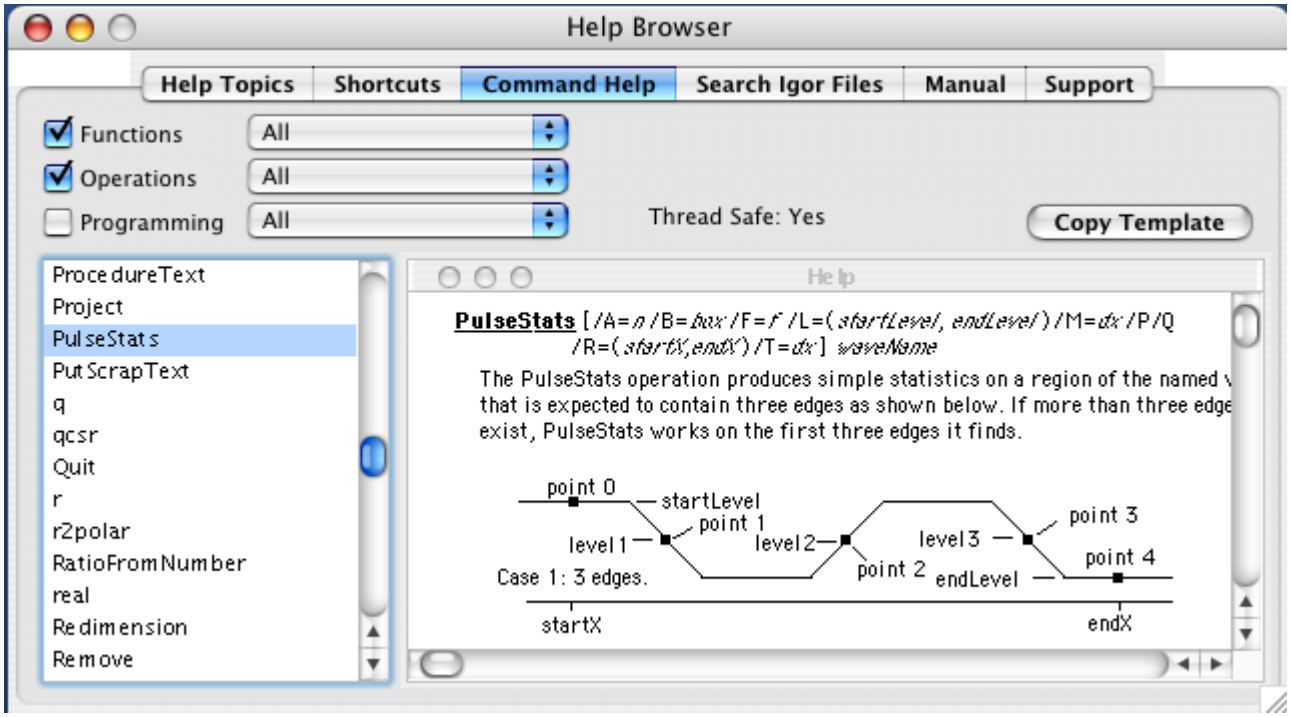
If you don't know what help file or what topic may contain the information of interest, use the Search Igor Files tab instead of the Help Topics tab.

Shortcuts Tab

The Shortcuts tab presents a list of shortcuts, organized in functional groups.

Command Help Tab

The Command Help tab provides quick access to reference information on Igor functions, operations and programming keywords. When you choose a function, operation or keyword in the list, Igor displays the associated help.



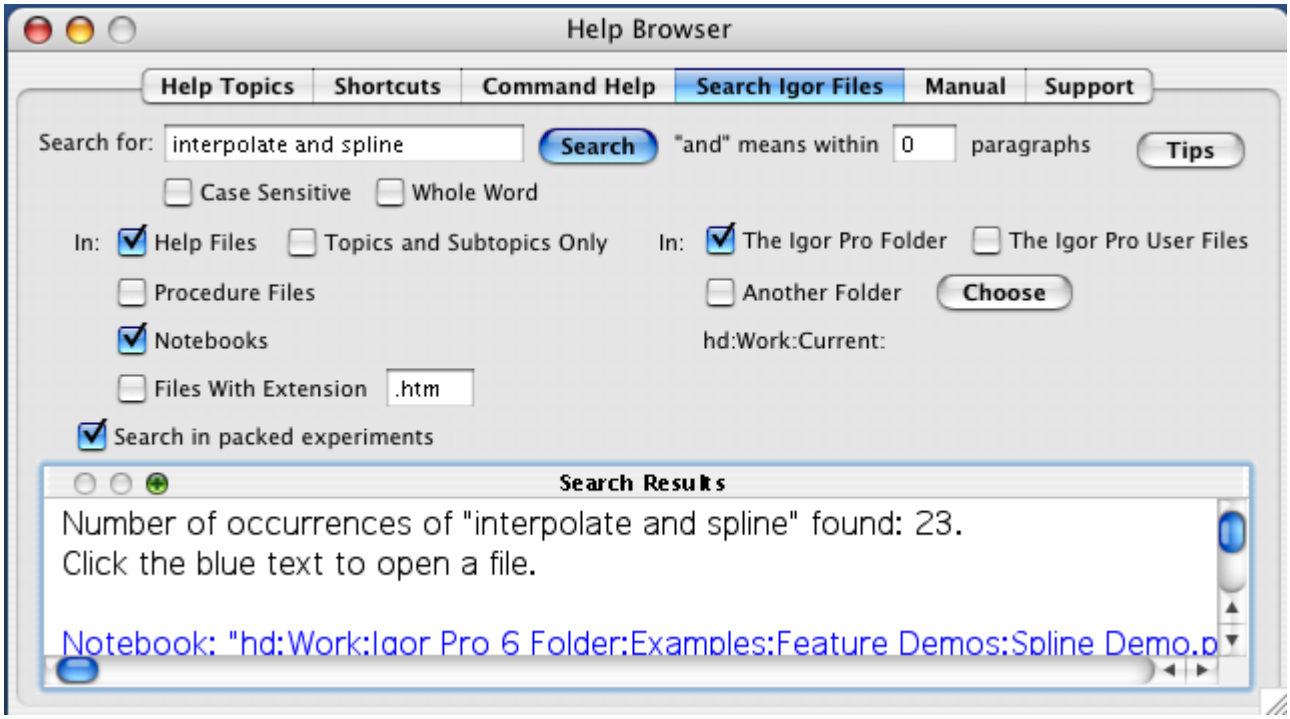
A checkbox for each of the three main categories adds or removes the associated items from the list. The pop-up menu next to each checkbox further narrows the scope of the list.

The Copy Template button copies a template to the Clipboard which you can then paste into the command line or into a procedure window. All functions and operations have templates, but only some keywords do.

Here is a tip to help you understand the distinction between an operation and a function: a function returns a direct result (e.g., sin) while an operation does not (e.g., Display).

Search Igor Files Tab

The Search Igor Files tab provides a way for you to search Igor help files, procedure files, and notebooks for information of interest.



Search Expression

The expression can consist of one or more (up to 8) terms. Terms are separated by the word "and". Here are some examples:

interpolation	*One term*
spline interpolation	*One term*
spline and interpolation	*Two terms*
spline and interpolation and smoothing	*Three terms*

The second example finds the exact phrase "spline interpolation" while the third example finds sections that contain the words "spline" and "interpolation", not necessarily one right after the other.

The only keyword supported in the search expression is "and". "Or" and "near" are not supported. Also, quotation marks in the search expression don't mean anything special and should not be used.

If your search expression includes more than one term, a text box appears in which you can enter a number that defines what "and" means. For example, if you enter 10, this means that the secondary terms must appear with 10 paragraphs of the primary term to constitute a hit. A value of 0 means that the terms must appear in the same paragraph. In a plain text file, such as a procedure file, a paragraph is a single line of text. Blank lines count as one paragraph.

Search Folders

You can search the Igor Pro Folder and all subfolders by selecting the associated checkbox. This is intended for situations in which you are looking for help on Igor features or examples of Igor programming.

You can search another folder and all subfolders by selecting the Another Folder checkbox and then by clicking the Choose button to specify the folder. This is useful when searching your own files or if you want to search a specific folder inside the Igor Pro Folder.

Types of Files

You can choose what type of files Igor should search: help files, procedure files, notebooks, or files with a specific extension. You can also look for notebooks and procedure files inside packed Igor experiment files.

When Igor searches notebooks, it searches both formatted notebooks (including Igor technical notes) and plain text notebooks. On Macintosh, any file whose file type is `'TEXT'` is considered to be a plain text notebook. On Windows, any file whose extension is ".txt" is considered to be a plain text notebook.

On Macintosh, Igor procedure files have the file type `'TEXT'`. Other kinds of plain text files, such as data files and readme files, also have the file type `'TEXT'` so Igor can not distinguish a procedure file from some other kind of plain text file. When you search procedure files, it searches all plain text files in the specified folder and subfolders.

On Windows, procedure files use the extension ".ipf" while various kinds of plain text files use other extensions, such as ".dat" and ".txt". When you search procedure files, it searches only files with the ".ipf" extension.

On both platforms, you can search files with a specific extension (".txt", ".dat") using the Files With Extension checkbox.

Search Results

Igor displays hits (occurrences of the search expression) in the Search Results windoid. At the top of the windoid, Igor displays the total number of hits.

Each hit is displayed as a file reference, in blue text, and the contents of the paragraph containing the hit, in black text. To open the file containing the hit, click the blue text.

To reduce clutter, if a single paragraph contains the search expression multiple times, this is considered to be one hit and is displayed as one file reference.

When you click a hit that refers to a stand-alone help file, procedure file or notebook, Igor opens the file and displays the paragraph containing the hit, highlighting the first term in the search expression.

If you click a hit that refers to a notebook or procedure file in a packed experiment, Igor can not open the file directly. It presents a dialog with two options:

- Open the experiment containing the file that contains the hit.
- Create and open a copy of the file that contains the hit.

Search Strategies

Usually when you are searching for information about Igor, you should include help files and notebooks and you should elect to search inside packed experiment files. This is because the examples provided in the Examples folder of the Igor Pro folder are mostly in the form of packed experiment files that include an explanatory notebook. Igor technical notes are usually in the form of stand-alone (not packed) notebook files.

If you are searching for user-defined functions in the WaveMetrics Procedures folder, you should elect to search procedure files. This gives you a handy way to find a function that does something that you need or to find an example of Igor programming.

Search Speed

Igor's searching does not use indexing. In other words, Igor opens and reads each file of the specified type or types in the specified folder or folders, and searches for the search expression. It does not store anything from one search to another. On slow computers, this may make searches annoyingly slow. In this case, you can speed things up by reducing the number of types of files to be searched or by more narrowly targeting the folder to be searched.

The online Igor Pro manual does provide indexed searches. This is described in the next section.

Manual Tab

The Manual tab provides a quick way for you to open the Igor Pro online manual.

The Open Online Manual button launches your PDF viewer program and opens the IgorMan.pdf file. Igor expects to find IgorMan.pdf in the "Igor Pro Folder:Manual" folder, where the Igor Pro installer installs it.

The Open Online Manual button displays an error message if it can not find the IgorMan.pdf file in the "Igor Pro Folder:Manual" folder or if there is no program on your hard disk configured to open PDF files. In this case, install Adobe Acrobat Reader from the Igor Pro CD or by downloading from <http://www.adobe.com/>.

Support Tab

The Support tab lists additional sources of help with Igor Pro.

Igor Help Files

The Igor installer places help files primarily in "Igor Pro Folder/Igor Help Files" and in "Igor Pro Folder/More Help Files". Help files for Igor extensions are installed in "Igor Pro Folder/Igor Extensions" and in "Igor Pro Folder/More Extensions".

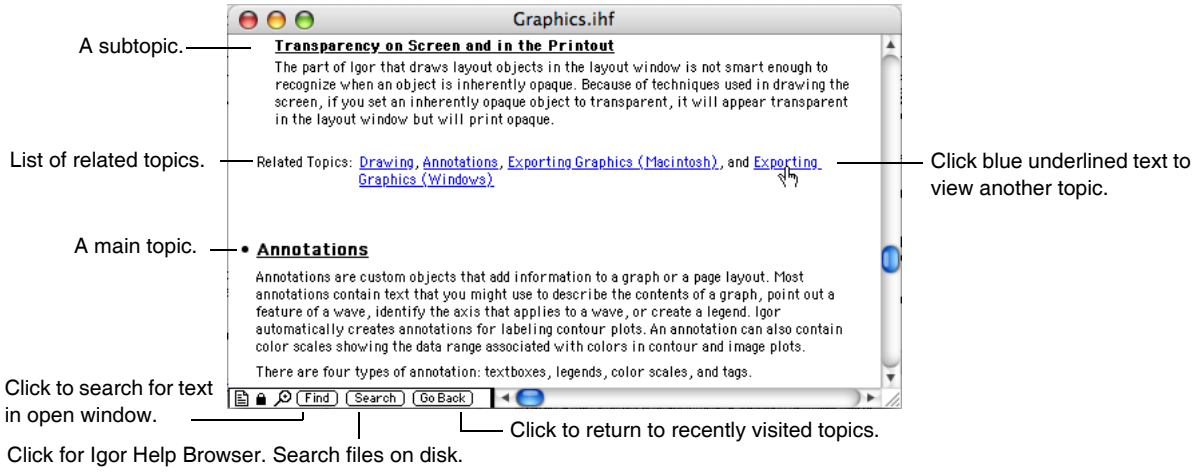
When Igor is launched it automatically opens any Igor help files in "Igor Pro Folder/Igor Help Files" and in "Igor Pro User Files/Igor Help Files". If you want Igor to automatically open another help file, create an alias (*Macintosh*) or shortcut (*Windows*) for that help file and drag it into "Igor Pro User Files/Igor Help Files" (see **Special Folders** on page II-44 for details).

Help files that you want to use occasionally can be stored anywhere on your hard disk. You can open them manually by double-clicking in the desktop, choosing File→Open File, or clicking the Open Another Help File button in the Help Topics tab of the Igor Help Browser.

Igor Help Windows

When Igor starts up, it automatically creates help windows by opening the Igor help files stored in "Igor Pro Folder/Igor Help Files" and in "Igor Pro User Files/Igor Help Files". You can display a help window by choosing it from the Help Windows submenu in the Windows menu.

Each Igor help file consists of a number of help topics, each of which may contain subtopics, and a list of related topics.



All of the topics are in the help file, one after another, like a big word-processing document. To see a list of topics, use the Igor Help Browser Help Topics tab. Blue underlined phrases are links that take you other parts of the help file or to other help files.

Click the Find button to search for words or phrases in the active help window. Click the Search button to search for words or phrases in multiple help files on disk.

Clicking the Go Back button or pressing Command-B (*Macintosh*) or Ctrl+B (*Windows*) takes you back to previous places in the help that you have visited. If you also press the Shift key, Igor will hide the active help window if you are going back to a different help window.

Hiding and Killing a Help Window

When you click the close button in a help window, Igor hides it. It also hides the help window if you choose Hide from the Windows menu or press Command-W (*Macintosh*) or Ctrl+W (*Windows*).

If you are finished using an Igor help window, you can kill it. This closes the file, removes its topics from the Help Browser and kills the window. It does not delete the file. To kill an Igor help file, you must press Option (*Macintosh*) or Alt (*Windows*) while clicking the close FilterFIR or while choosing Close from the Windows menu.

Executing Commands from a Help Window

Help windows often show example Igor commands. To execute a command or a section of commands from a help window, select the command text and press Control-Enter or Control-Return. This sends the selected text to the command line and starts execution.

Compiling Help Files

Each Igor help file contains compiled help information that Igor uses to quickly find topics and subtopics. If you open a help file that has been modified, Igor will ask if you want to "compile" it. Compiling is what creates this information. This will happen only if you intentionally or accidentally modify a help file and then open it.

Windows: Prior to Igor Pro 5, Igor stored the compiled help information in a separate file with a ".igr" extension. Now Igor stores the compiled help information in the help file itself. The ".igr" file is no longer needed.

Creating Your Own Help File (For Advanced Users)

You can create an Igor help file that extends the Igor help system. This is something you might want to do if you write a set of Igor procedures or extensions for use by your colleagues. If your procedures or extensions are generally useful, you might want to make them available to all Igor users. In either case, you can provide documentation in the form of an Igor help file.

Here are the steps for creating an Igor help file.

1. Create a formatted-text notebook.

   A good way to do this is to open the Igor Help File Template provided by WaveMetrics in the More Help Files folder. Alternatively, you can start by duplicating another WaveMetrics-supplied help file and then open it as a notebook using File→Open File→Notebook. Either way, you are starting with a notebook that contains the rulers used to format an Igor help file.
2. Choose Save Notebook As from the File menu to create a new file. Use a ".ihf" extension so that Igor will recognize it as a help file.
3. Enter your help text in the new file.
4. Save and kill the notebook.
5. Open the file as a help file using File→Open File→Help File.

When you open the file as a help file, it needs to be compiled. When Igor compiles a help file, it scans through it to find out where the topics start and end and makes a note of subtopics. When the compilation is finished, it saves the help file which now includes the help compiler information.

Once Igor has successfully compiled the help file, it will act like any other Igor help file. That is, when opened it will appear in the Help Windows submenu, its topics will appear in the Help Browser and you can click links to jump around.

Here are the steps for modifying a help file.

1. If the help file is open, kill it by pressing Option (*Macintosh*) or Alt (*Windows*) and clicking the close button.
2. Open it as a notebook, using File→Open File→Notebook.
3. Modify it using normal editing techniques.
4. Choose Save Notebook from the File menu.
5. Click the close button and kill the notebook.
6. Reopen it as a help file using File→Open File→Help File.

Syntax of a Help File

Igor needs to be able to identify topics, subtopics, related-topics declarations and links in Igor help files. To do this it looks for certain rulers, text patterns and text formats described in **Creating Links** on page II-13. You can get most of the required text formats by using the appropriate ruler from the Igor Help File Template file.

Igor considers a paragraph to be a help topic declaration if it starts with a bullet character followed by a tab and if the paragraph's ruler is named Topic. By convention, the Topic ruler's font is Geneva on Macintosh or Arial on Windows, its text size is 12 and its text style is bold-underlined. The bullet and tab characters should be plain, not bold or underlined.

The easiest way to create a new topic with the right formatting is to copy an existing topic and then modify it.

Once Igor finds a topic declaration, it scans the body of the topic. The body is all of the text until the next topic declaration, a related-topics declaration or the end of the file. While scanning, it notes any subtopics.

Igor considers a paragraph to be a subtopic declaration if the name of the ruler governing the paragraph starts with "Subtopic". Thus if the ruler is named Subtopic or Subtopic+ or Subtopic2, the paragraph is a subtopic declaration. By convention, the Subtopic ruler's font is Geneva on Macintosh or Arial on Windows, its text size is 10 and its text style is bold and underlined. Text following the subtopic name that is not bold and underlined is not part of the subtopic name.

The easiest way to create a new subtopic with the right formatting is to copy an existing subtopic and then modify it.

Igor considers a paragraph to be a related-topics declaration if the ruler governing the paragraph is named RelatedTopics and if the paragraph starts with the text pattern "Related Topics:". When Igor sees this pattern it knows that this is the end of the current topic. The related-topics declaration is optional. Prior to Igor Pro 4, Igor displayed a list of related topics in the Igor Help Browser. Igor Pro no longer displays this list. The user can still click the links in the related topics paragraph to jump to the referenced topics.

Igor knows that it has hit the end of the current topic when it finds the related-topics declaration or when it finds a new topic declaration. In either case, it proceeds to compile the next topic. It continues compiling until it hits the end of the file.

When compiling the help file, Igor may encounter syntax that it can't understand. For example, if you have a related-topics declaration paragraph, Igor will expect the next paragraph to be a topic declaration. If it is not, Igor will stop the compilation and display an error dialog. You need to open the file as a notebook, fix the error, save and kill it and then reopen it as a help file.

Another error that is easy to make is to fail to use the plain text format for syntactic elements like bullet-tab, "Related Topics:" or the comma and space between related topics. If you run into a non-obvious compile error in a topic, subtopic or related topics declaration, recreate the declaration by copying from a working help file.

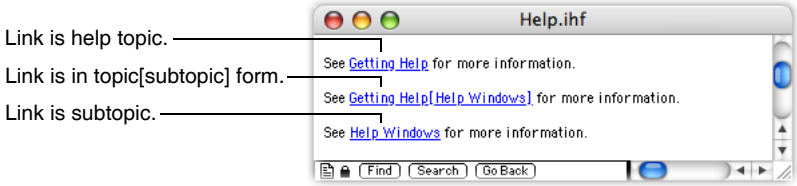
The help files supplied by WaveMetrics contain a large number of rulers to define various types of paragraphs such as topic paragraphs, subtopic paragraphs, related topic paragraphs, topic body paragraphs and so on. The Igor Help File Template contains many but not all of these rulers. If you find that you need to use a ruler that exists in a WaveMetrics help file but not in your help file then copy a paragraph governed by that ruler from the WaveMetrics help file and paste it into your file. This will transfer the ruler to your file.

Creating Links

A link is text in an Igor help file that, when clicked, takes the user to some other place in the help. Igor considers any pure blue, underlined text to be a link. Pure blue means that the RGB value is (0, 0, 65535). By convention links use the Geneva font on Macintosh and the Arial font on Windows.

To create a link, select the text in the notebook that you are preparing to be a help file. Then choose Make Help Link from the Notebook menu. This sets the text format for the selected text to pure blue and underlined.

The link text refers to another place in the help using one of the following forms:



When the user double-clicks a link, Igor performs the following search:

1. If the link is a topic name, Igor goes to that topic.
2. If the link is in topic[subtopic] form, Igor goes to that subtopic.
3. If steps 1 and 2 fail, Igor searches for a subtopic with the same name as the link. First, it searches for a subtopic in the current topic. If that fails, it searches for a subtopic in the current help file. If that fails, it searches for a subtopic in all help files.
4. If step 3 fails, Igor searches all help files in the Igor Pro folder. If it finds the topic in a closed help file, it opens and displays it.
5. If all of the above fail, Igor displays a dialog suggesting that the required help file is not available.

You can create a link in a help file that will open a Web page or FTP site in the user's Web or FTP browser. You do this by entering the Web or FTP URL in the help file while you are editing it as a notebook. The URL must appear in this format:

```
<http://www.wavemetrics.com>
<ftp://ftp.wavemetrics.com>
```

The URL must include the angle brackets and the "http://" or "ftp://" protocol specifier.

After entering the URL, select the entire URL (including the angle brackets) and choose Make Help Link from the notebook menu. Once the file is compiled and opened as a help file, clicking the link will open the user's Web or FTP browser and display the specified URL.

It is currently not possible make ordinary text into a Web or FTP link. The text must be an actual URL in the format shown above or you can insert a notebook action which brings up a web page using the **BrowseURL** operation on page V-38. See **Notebook Action Special Characters** on page III-18 for details.

Checking Links

You can get Igor to check your help links as follows:

1. Open your Igor help file and any other help files that you link to.
2. Activate your help window and click at the very start of the help text.
3. Press Command-Shift-Option-H (*Macintosh*) or Ctrl+Shift+Alt+H (*Windows*). Igor will check your links from where you clicked to the end of the file and note any problems by writing diagnostics to the history area of the command window.
4. When Igor finishes checking, if it found bad links, kill the help file and open it as a notebook.
5. Use the diagnostics that Igor has written in the history to find and fix any link errors.
6. Save the notebook and kill it.
7. Open the notebook as a help file. Igor will compile it.
8. Repeat the check by going back to Step 1 until you have no bad links.

You can abort the check by pressing Command-period (*Macintosh*) or Ctrl-Break (*Windows*) and holding it for a second.

The diagnostic that Igor writes to the history in case of a bad link is in the form:

```
Notebook $nb selection={(33,292), (33,334)} …
```

This is set up so that you can execute it to find the bad link. At this point, you have opened the help file as a notebook. Assuming that it is named Notebook0, execute

```
String/G nb = "Notebook0"
```

Now, you can execute the diagnostic commands to find the bad link and activate the notebook. Fix the bad link and then proceed to the next diagnostic. It is best to do this in reverse order, starting with the last diagnostic and cutting it from the history after fixing the problem.

When fixing a bad link, check the following:

- A link is the name of a topic or subtopic in a currently open help file. Check spelling.
- There are no extraneous blue/underlined characters, such as tabs or spaces, before or after the link. (You can not identify the text format of spaces and tabs by looking at them. Check them by selecting them and then using the Set Text Format dialog.)
- There are no duplicate topics. If you specify a link in topic[subtopic] form and there are two topics with the same topic name, Igor may not find the subtopic.

Updating Igor

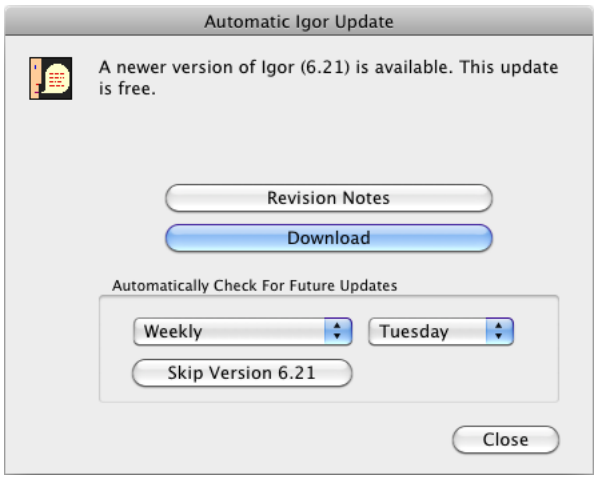
WaveMetrics periodically releases updates for Igor. Updates provide bug fixes and new features. Updates are free and are usually indicated by a .01 increment in version number (e.g., 6.20 to 6.21). By contrast, upgrades are released roughly every two years, provide major new functionality, and usually require a purchase.

Igor 6.20 or later (English version only) optionally checks during startup for available updates to the Igor application by contacting one of our web sites. This update check is performed on a separate processing thread to minimize any impact on starting Igor.

If an update is found, Igor presents a dialog in which you can choose to:

- Download the update
- Ignore the specific update
- Set the schedule for future update checks
- Disable automatic update checking

The dialog's buttons vary depending on the updates available but generally looks like this:



If scheduled, Igor checks for updates during the first startup of the scheduled day, or after the scheduled day has passed. Igor never automatically checks more than once per day.

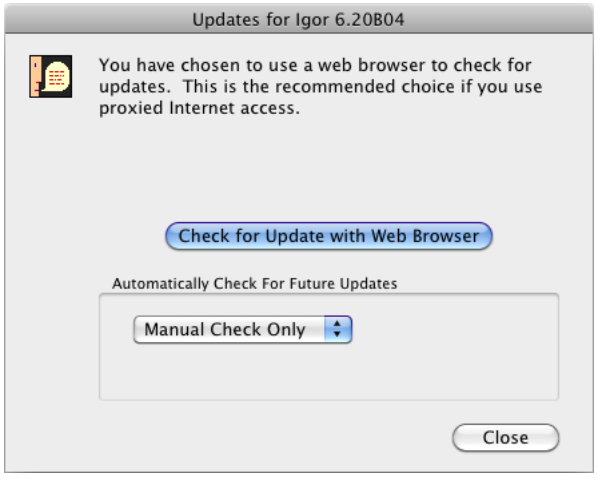
The frequency of automatic update checks can be set in the Updates for Igor dialog.

The default update check frequency is weekly. If this is too often, you might elect to automatically check once per month.

You can also completely disable the automatic update checks:

1. Choose Help→Updates For Igor to display the Updates for Igor dialog.
2. Choose Manual Check Only from the left popup menu.

You can check for updates manually by choosing Help→Updates For Igor, which presents this dialog:



Technical Support

WaveMetrics provides technical support via telephone and email.

Before contacting WaveMetrics, please gather this information so that we can help you more effectively:

- The exact version of Igor you are running. The version number is displayed in the About Igor dialog displayed when Igor is launched.
- Which operating system you are running.

In most cases, we need to reproduce your problem in order to solve it. It is best if you can provide a simplified example showing the problem.

Email Support

Send technical questions to us via email at:

```
support@wavemetrics.com
```

For information on upgrades and other nontechnical information, send queries to:

```
sales@wavemetrics.com
```

FTP Sites

Several FTP sites store the latest versions of Igor technical notes, utilities and user contributions. A list of these sites is maintained on our support web page at:

```
http://www.wavemetrics.com/support/
```

World Wide Web

You will find our Web site at:

```
http://www.wavemetrics.com/
```

You can also choose Help→WaveMetrics Home Page.

Our Web site contains a page for searching our support database, and links to Igor-related FTP sites and to Igor users' Web pages. In addition, it contains a number of cool graphs. We are always grateful for new cool graphs. Contact us at sales@wavemetrics.com if you have a cool graph to share.

WaveMetrics Support Web Page

The support Web page includes a searchable support database and archives of the Igor Mailing List. The WaveMetrics support Web address is:

```
http://www.wavemetrics.com/support/
```

You can also choose Help→Support Web Page.

Igor Mailing List

The Igor mailing list is an Internet discussion list that provides a way for Igor Pro users to help one another and to share solutions and ideas. WaveMetrics also uses the list to post information on the latest Igor developments. For information about subscribing and other details about the mailing list, please visit this web page:

```
http://www.wavemetrics.com/users/mailinglist.htm
```

IgorExchange

IgorExchange is a user-to-user support and collaboration web site sponsored by WaveMetrics but run by and for Igor users. For information about IgorExchange, please visit this web page:

```
http://www.igorexchange.com
```

Telephone Support

You can reach us at 503-620-3001 from 9 AM to 5 PM Pacific time.

It is often very helpful if you can try things on your computer while speaking to us so, if possible, call us from a phone near your computer.

FAX Support

You can reach our FAX machine any time at 503-620-6754.

Help Shortcuts

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To activate Igor Tips	Press Option-Help.	—
To get a contextual menu of commonly-used actions	Press Control and click in the body of an Igor help window.	Right-click in an Igor help window.
To activate the Igor Help Browser	Press Help or click the Igor Help Browser icon in the lower-right corner of the command window.	Press F1 or click the Igor Help Browser icon in the lower-right corner of the command window.
To jump to a topic in an Igor help window	Click a blue underlined topic link in the help window.	Click a blue underlined topic link in the help window.
To jump back to recently visited topics	Click the Go Back button at the bottom of the help window or press Command-B.	Click the Go Back button at the bottom of the help window or press Ctrl+B.
	Press Command-Shift-B to hide the current help window when going back to a different help window.	Press Shift+Ctrl+B to hide the current Igor help window when going back to a different Igor help window.
To execute commands in an Igor help window	Select the commands and press Control-Return or Control-Enter.	Select the commands and press Ctrl+Enter.
To kill a help window	Option-click the close button.	Press Alt and click the close button.
To insert a function or operation template in a procedure window or in the command line	Type or select the name of an Igor operation or function and Control-click it or press Shift-Help.	Type or select the name of an Igor operation or function and right-click it or press Ctrl+F1.
To get help for a function or operation from a procedure, notebook or help window or from the command line	Type or select the name of an Igor operation or function and Control-click it or press Shift-Option-Help.	Type or select the name of an Igor operation or function and right-click it or press Ctrl+Alt+F1.

Chapter
II-2

The Command Window

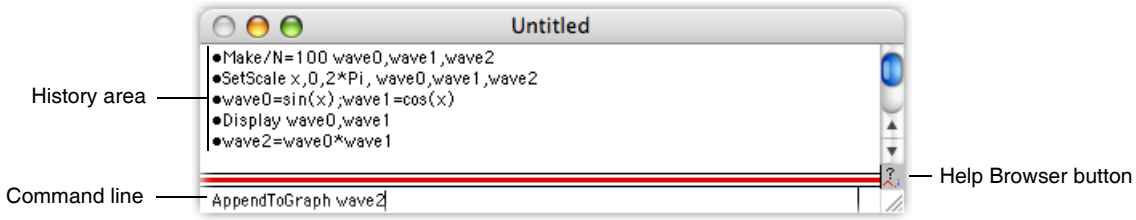
Overview

You can control Igor using menus and dialogs or using commands that you execute from the command window. Some actions, for example waveform assignments, are much easier to perform by entering a command. Commands are also convenient for trying variations on a theme — you can modify and reexecute a command very quickly. If you use Igor regularly, you may find yourself using commands more and more for those operations that you frequently perform.

In addition to executing commands in the Command window, you can also execute commands in a notebook, procedure or help window. These techniques are less commonly used than the command window. See **Notebooks as Worksheets** on page III-5 for more information.

This chapter describes the command window and general techniques and shortcuts. See Chapter IV-1, **Working with Commands**, for details on command usage and syntax.

The command window consists of a **command line** and a **history area.** When you enter commands in the command line and press Return (*Macintosh*) or Enter (*Windows and Macintosh*), the commands are executed. Then they are saved in the history area for you to review. If a command produces text output, that output is also saved in the history area. A bullet character is prepended to command lines in the history so that you can easily distinguish command lines from output lines.



The Command window includes a help button just below the History area scroll bar. Clicking the button displays the Help Browser window. See **Igor Help Browser** on page II-6 for more details about the Help Browser.

The total length of a command on the command line must not exceed 400 characters.

There is no line continuation character in Igor. However, it is nearly always possible to break a single command up into multiple lines.

Command Window Example

Here is a quick example designed to illustrate the power of commands and some of the shortcuts that make working with commands easy.

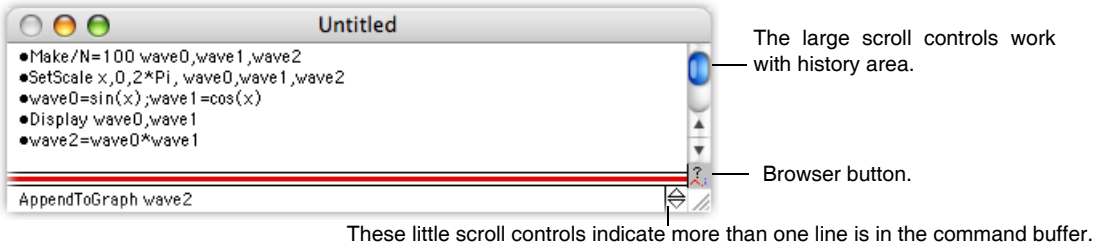
1. **Choose New Experiment from the File menu.**
2. **Execute the following command by typing in the command line and then pressing Return or Enter.**
   ```
   Make/N=100 wave0; Display wave0
   ```
   This displays a graph.
3. **Press Command-J** (*Macintosh*) **or Ctrl+J** (*Windows*)**.**
   This activates the command window.
4. **Execute**
   ```
   SetScale x, 0, 2*PI, wave0; wave0 = sin(x)
   ```
   The graph shows the sine of x from 0 to 2π.
   Now we are going to see how to quickly retrieve, modify and reexecute a command.
5. **Press the Up Arrow key.**
   This selects the command that we just executed.
6. **Press Return or Enter.**
   This transfers the selection back into the command line.

7. **Change the "2" to "4".**
   The command line should now contain:
   ```
   SetScale x, 0, 4*PI, wave0; wave0 = sin(x)
   ```
8. **Press Return or Enter to execute the modified command.**
   This shows the sine of x from 0 to 4π.
9. **While pressing Option** (*Macintosh*) **or Alt** (*Windows*)**, click the last command in the history.**
   This is another way to transfer a command from the history to the command line. The command line should now contain:
   ```
   SetScale x, 0, 4*PI, wave0; wave0 = sin(x)
   ```
10. **Press Command-K** (*Macintosh*) **or Ctrl+K** (*Windows*)**.**
    This "kills" the contents of the command line.
    Now let's see how you can quickly reexecute a previously executed command.
11. **With Command and Option** (*Macintosh*) **or Ctrl and Alt** (*Windows*) **pressed, click the second-to-last command in the history.**
    This reexecutes the clicked command (the 2*PI command).
    Repeat this step a number of times, clicking the second-to-last command each time. This will alternate between the 2*PI command and the 4*PI command.
12. **Execute**
    ```
    WaveStats wave0
    ```
    Note that the WaveStats operation has printed its results in the history where you can review them. You can also copy a number from the history to paste into a notebook or an annotation.

There is a summary of all command window shortcuts at the end of this chapter.

The Command Buffer

The command line shows a single line of the **command buffer**. Normally the command buffer is either empty or contains just one line of text. However you can copy multiple lines of text from any window and paste them in the command buffer. When more than one line is in the command buffer, little scroll controls appear at the right end of the command line.



You can clear the contents of the command buffer by choosing the Clear Command Buffer item in the Edit menu or by pressing Command-K (*Macintosh*) or Ctrl+K (*Windows*).

When you invoke an operation from a typical Igor dialog, the dialog puts a command in the command buffer and executes it. The command is then transferred to the history as if you had entered the command manually.

If an error occurs during the execution of a command, Igor leaves it in the command buffer so you can edit and reexecute it. If you don't want to fix the command, you should remove it from the command buffer by pressing Command-K (*Macintosh*) or Ctrl+K (*Windows*).

Because the command buffer usually contains nothing or one command, we usually think of it as a single line and use the term "command line".

Command Window Title

The title of the command window is the name of the experiment that is currently loaded. When you first start Igor or if you choose New from the File menu, the title of the experiment and therefore of the command window is “Untitled”.

When you save the experiment to a file, Igor sets the name of the experiment to the file name minus the file extension. If the file name is “An Experiment.pxp”, the experiment name is “An Experiment”. Igor displays “An Experiment” as the command window title.

For use in procedures, the **IgorInfo**(1) function returns the name of the current experiment.

History Area

The history area is a repository for commands and results.

Text in the history area can not be edited but can be copied to the Clipboard or to the command line. Copying text to the Clipboard is done in the normal manner. To copy a command from the history to the command buffer, select the command in the history and press Return or Enter. An alternate method is to press Option (*Macintosh*) or Alt (*Windows*) and click in the history area.

To make it easy tocopy a command from the history to the command line, clicking a line in the history area selects the entire line. You can still select just part of a line by clicking and dragging.

Up Arrow and Down Arrow move the selection range in the history up or down one line selecting an entire line at a time. Since you normally want to select a line in the history to copy a command to the command line, Up Arrow and Down Arrow skip over non-command lines. Left Arrow and Right Arrow move the insertion point in the command line.

When you save an experiment, the contents of the history area are saved. The next time you load the experiment the history will be intact. Some people have the impression that Igor recreates an experiment by reexecuting the history. This is not correct. See **How Experiments Are Loaded** on page II-39 for details.

Limiting Command History

The contents of the history area can grow to be quite large over time. You can limit the number of lines of text retained in the history using the Limit Command History feature in the Command Settings section of the Miscellaneous Settings dialog which is accessible through the Misc menu.

If you limit command history, when you save the experiment, Igor checks the number of history lines. If they exceed the limit, the oldest lines are deleted.

History Archive

When history lines are deleted through the Limit Command History feature, the History Archive feature allows you to tell Igor to write the deleted lines to a text file in the experiment's home folder.

To enable the History Archive feature for a given experiment, create a plain text file in the home folder of the experiment. The text file must be named "<Experiment Name> History Archive.txt" where <Experiment Name> is the name of the current experiment. Now, when you save the experiment, Igor writes any deleted history lines to the history archive file.

If the history archive file is open in any program, including Igor, the history archive feature will fail and no history lines will be written.

History Carbon Copy

This feature is expected to be of interest only in rare cases for advanced Igor programmers such as Bela Farago who requested it.

You can designate a notebook to be a "carbon copy" of the history area by creating a plain text or formatted notebook and setting its window name, via Windows->Window Control, to HistoryCarbonCopy. If the HistoryCarbonCopy notebook exists, Igor inserts history text in the notebook as well as in the history. However, if a command is initiated from the HistoryCarbonCopy notebook (see **Notebooks as Worksheets** on page III-5), Igor suspends sending history text to that notebook during the execution of the command.

If you rename the notebook to something other than HistoryCarbonCopy, Igor will cease sending history text to it. If you later rename it back to HistoryCarbonCopy, Igor will resume sending history text to it.

The history trimming feature accessed via the Miscellaneous Settings dialog does not apply to the HistoryCarbonCopy notebook. You must trim it yourself. Notebooks are limited to 16 million paragraphs.

When using a formatted notebook as the history carbon copy, you can control the formatting of commands and results by creating notebook rulers named Command and Result. When Igor sends text to the history carbon copy notebook, it always applies the Command ruler to commands. It applies the Result ruler to results if the current ruler is Normal, Command or Result. You must create the Command and Result rulers if you want Igor to use them when sending text to the history carbon copy.

This function creates a formatted history carbon copy notebook with the Command and Result rulers used automatically by Igor as well as an Error ruler which we will use for our custom error messages:

```
Function CreateHistoryCarbonCopy()
   NewNotebook /F=1 /N=HistoryCarbonCopy /W=(50,50,715,590)

   Notebook HistoryCarbonCopy backRGB=(0,0,0)// Set background to black

   Notebook HistoryCarbonCopy showRuler=0

   // Define ruler to govern commands.
   // Igor will automatically apply this to commands sent to history carbon copy.
   Notebook HistoryCarbonCopy newRuler=Command,
                        rulerDefaults={"Geneva",10,0,(65535,65535,0)}

   // Define ruler to govern results.
   // Igor will automatically apply this to results sent to history carbon copy.
   Notebook HistoryCarbonCopy newRuler=Result,
                        rulerDefaults={"Geneva",10,0,(0,65535,0)}

   // Define ruler to govern user-generated error messages.
   // We will apply this ruler to error messages that we send
   // to history carbon copy via Print commands.
   Notebook HistoryCarbonCopy newRuler=Error,
rulerDefaults={"Geneva",10,0,(65535,0,0)}
End
```

If the current ruler is not Normal, Command or Result, it is assumed to be a custom ruler that you want to use for special messages sent to the history using the Print operation. In this case, Igor does not apply the Result ruler but rather allows your custom ruler to remain in effect.

This function sends an error message to the history using the custom Error ruler in the history carbon copy notebook:

```
Function PrintErrorMessage(message)
   String message

   Notebook HistoryCarbonCopy, ruler=Error
   Print message

   // Set ruler back to Result so that Igor's automatic use of the Command
   // and Result rulers will take effect for subsequent commands.
```

```
   Notebook HistoryCarbonCopy, ruler=Result
End
```

XOP programmers can use the XOPNotice3 XOPSupport routine to control the color of text sent to the History Carbon Copy notebook.

Searching the Command Window

You can search the command line or the history by choosing Find from the Edit menu or by using the keyboard shortcuts as shown in the Edit menu. Searching the command line is most often used to modify a previously executed command before reexecuting it. For example, you might want to replace each instance of a particular wave name with another wave name.

If there is an active selection in the history, Find searches the history. Otherwise it searches the command line. To be sure that Find will search the area that you want, you can click in that area before starting the search.

Command Window Formats

You can change the text format used for the command line. For example, you might prefer larger type. To do this, click in the command line and then choose Set Text Format from the Command Buffer submenu of the Misc menu. To set the text format for the history area, click in the history area and then choose Set Text Format from the History Area submenu of the Misc menu. To do this, the history area must have some text in it.

You can set other properties, such as background color, by choosing Document Settings from the Command Buffer or History Area submenus. The Document Settings dialog also sets the header and footer used when printing the history.

When you change the text format or document settings, you are changing the current experiment only. You may want to capture the new format and settings as a preference for new experiments. To do this, choose Capture Prefs from the Command Buffer and History Area submenus.

Getting Help from the Command Line

When working with the command line, you might need help in formulating a command. There are shortcuts that allow you to insert a template, view help, or find the definition of a user function.

To insert a template, type the name of the operation or function and then press Shift-Help (*Macintosh*) or Ctrl+F1 (*Windows*).

To view help or to view the definition of a user function, type the name of the operation or function and then press Shift-Option-Help (*Macintosh*) or Ctrl+Alt+F1 (*Windows*).

You can also insert a template or get help by Control-clicking (*Macintosh*) or right-clicking (*Windows*).

To view text window keyboard navigation shortcuts, see **Text Window Navigation** on page II-68.

This table may help you remember what the various keyboard shortcuts do.

Keyboard Shortcut		**What It Does**
Macintosh	*Windows*	
Press Help	Press F1	Displays help browser window
Press Shift-Help	Press Ctrl+F1	Inserts template for selected operation or function
Press Shift-Option-Help	Press Ctrl+Alt+F1	Displays help for selected operation or function

Command Window Shortcuts

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To activate the command window	Press Command-J.	Press Ctrl+J.
To clear the command buffer	Press Command-K.	Press Ctrl+K.
To get a contextual menu of commonly-used actions	Press Control and click the history area or command line	Right-click the history area or command line
To copy a line from the history to the command buffer	Click the line and press Return or Enter. Press Option and click the line.	Click the line and press Enter. Press Alt and click the line.
To reexecute a line from the history	Click the line and press Return or enter twice Press Command-Option and click the line.	Click the line and press Ctrl+Enter Press Ctrl+Alt and click the line.
To find a recently executed command in the history	Press the Up or Down Arrow keys.	Press the Up or Down Arrow keys.
To find text in the history	Click in the history area and press Command-F.	Click in the history area and press Ctrl+F.
To find text in the command line	Click in the command line and press Command-F.	Click in the command line and press Ctrl+F.
To get a template	Type the name of an operation or function and press Shift-Help.	Type the name of an operation or function and press Ctrl+F1.
To get help or view the definition of a user function	Type the name of an operation or function and press Shift-Option-Help.	Type the name of an operation or function and press Ctrl+Alt+F1.

Chapter
II-3

Experiments, Files and Folders

Experiments

An **experiment** is a collection of Igor objects, including waves, variables, graphs, tables, page layouts, notebooks, control panels and procedures. When you create or modify one of these objects you are modifying the **current experiment**.

You can save the current experiment by choosing File→Save Experiment. You can open an experiment by double-clicking its icon on the desktop or choosing File→Open Experiment.

Saving Experiments

There are two formats for saving an experiment on disk:

- As a packed experiment file. A packed experiment file has the extension .pxp.
- As an experiment file and an experiment folder (unpacked format). An unpacked experiment file has the extension .uxp.

The packed format is recommended for most purposes. The unpacked format is useful for experiments that include very large numbers of waves (thousands or more).

Extensions are not strictly necessary on Macintosh but they are recommended for ease of file sharing and future compatibility.

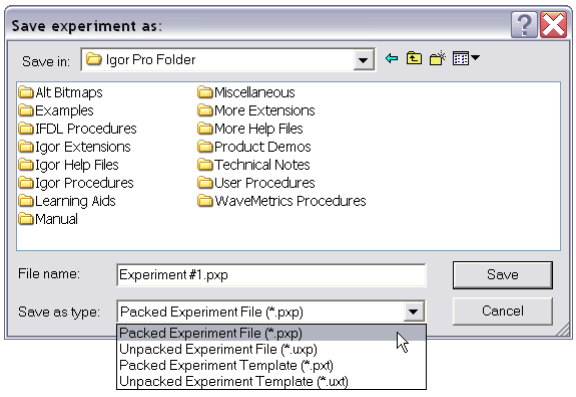
Saving as a Packed Experiment File

In the packed experiment file, all of the data for the experiment is stored in one file. This saves space on disk and makes it easier to copy experiments from one disk to another. *For most work, we recommend that you use the packed experiment file format*.

The folder containing the packed experiment file is called the **home folder**.

To save a new experiment in the packed format, choose Save Experiment from the File menu. Igor displays the following dialog:

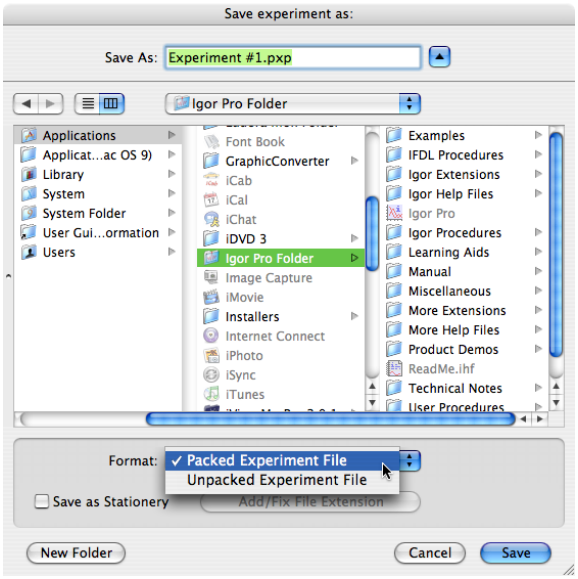
Windows



Select Packed Experiment File.

You can select the default using the Experiment section of the Miscellaneous Settings dialog in the Misc menu.

Macintosh



The next illustration shows the icon for a packed experiment file.

Contains the startup commands that Igor executes to recreate the experiment, including all experiment windows.

Also contains data for waves, variables, history, procedures, notebooks, pictures and other items.

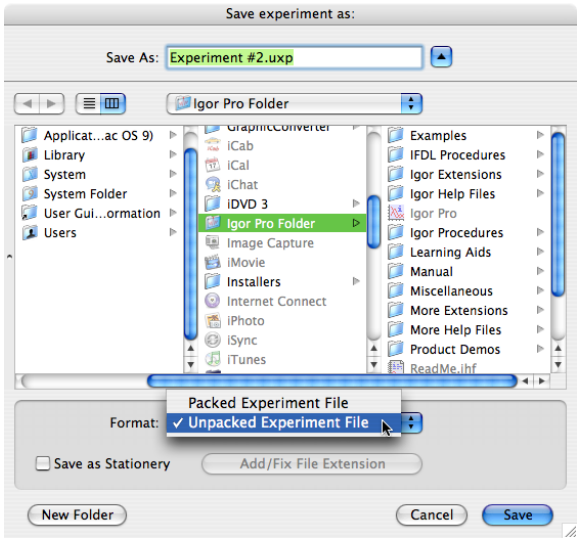
Saving as an Unpacked Experiment File

In the unpacked format, an experiment is saved as an **experiment file** and an **experiment folder**. The file contains instructions that Igor uses to recreate the experiment while the folder contains files from which Igor loads data. The experiment folder is also called the **home folder**.

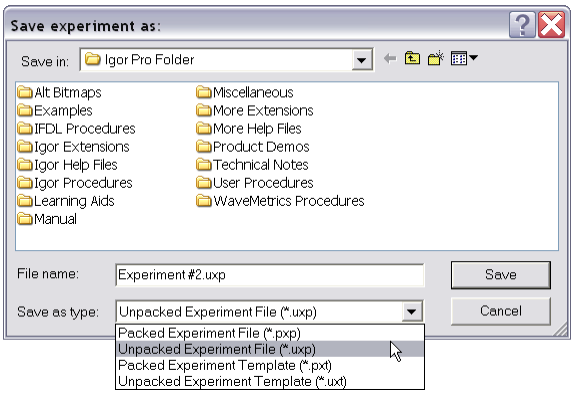
The main utility of this format is that it is faster for experiments that contain very large numbers of waves (thousands or more). However the unpacked format is more fragile and thus is not recommended for routine use.

To save a new experiment in the unpacked format, choose Save Experiment from the File menu. Igor displays the following dialog:

Macintosh



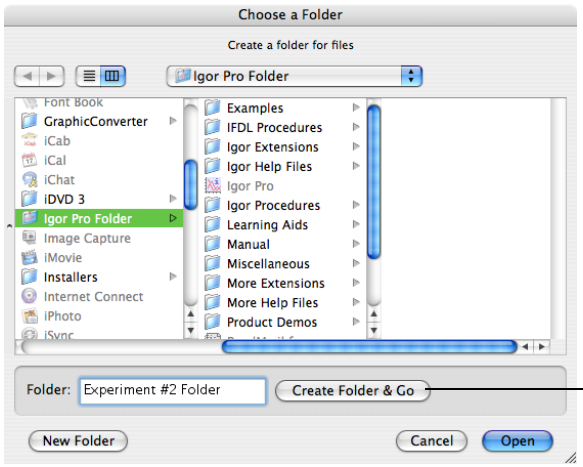
Windows



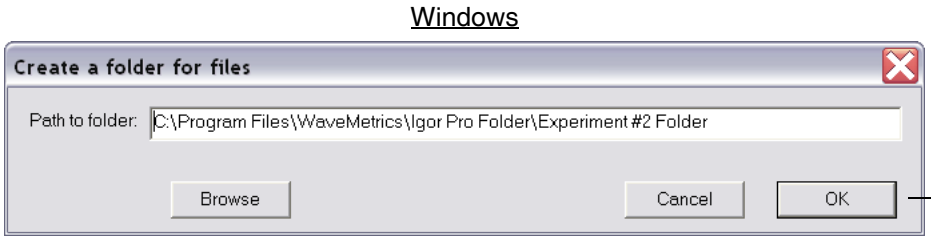
Select Unpacked Experiment File.

Once you click the Save button with Packed deselected, Igor presents a second dialog to allow you to create the experiment folder:

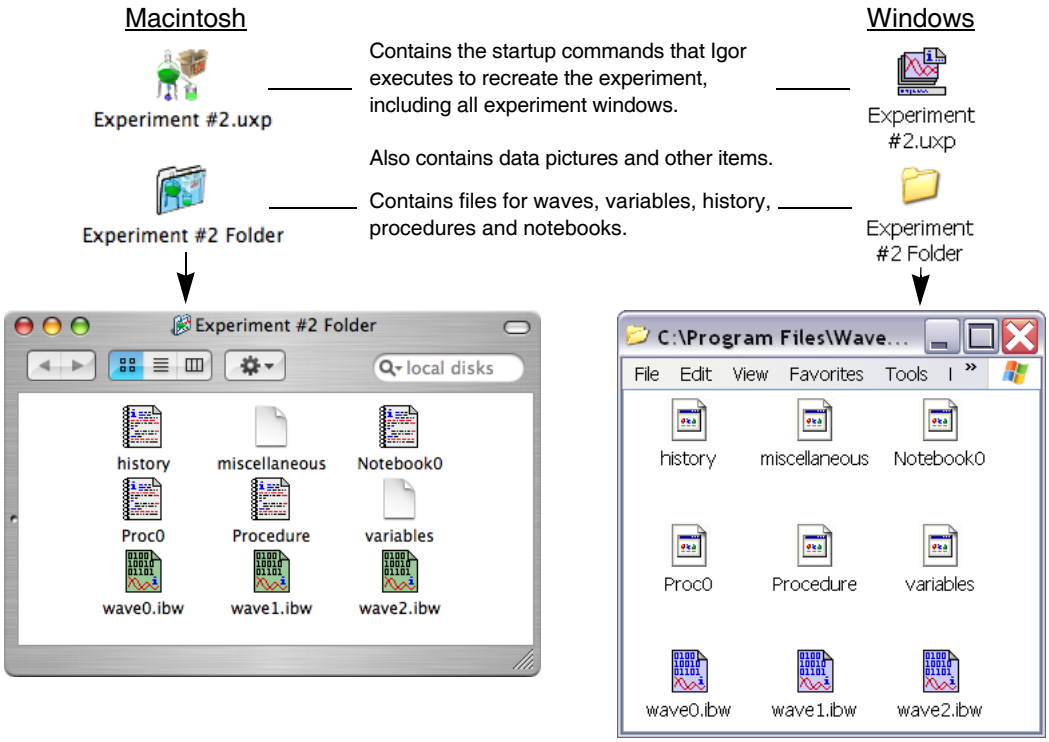
Macintosh



Click to create the experiment folder.



The next illustration shows the icons used with an unpacked experiment and explains where things are stored.



You normally have no need to deal with the files inside the experiment folder. Igor automatically writes them when you save an experiment and reads them when you open an experiment.

If the experiment includes data folders (see Chapter II-8, **Data Folders**) other than the root data folder, then Igor will create one subfolder in the experiment folder for each data folder in the experiment. The experiment shown in the illustration above contains no data folders other than root.

Note that there is one file for each wave. These are Igor Binary data files and store the wave data in a compact format. For the benefit of programmers, the Igor Binary file format is documented in Igor Technical Note #003.

The "procedure" file holds the text in the experiment's built-in procedure window. In this example, the experiment has an additional procedure window called Proc0 and a notebook.

The "variables" file stores the experiment's numeric and string variables in a binary format.

The advantages of the unpacked experiment format are:

- Igor can save the experiment faster because it does not need to update files for waves, procedures or notebooks that have not changed.
- You can share files stored in one experiment with another experiment. However, sharing files can cause problems when you move an experiment to another disk. See **References to Files and Folders** on page II-37 for an explanation.

The disadvantages of the unpacked experiment format are:

- It takes more disk space, especially for experiments that have a lot of small waves.

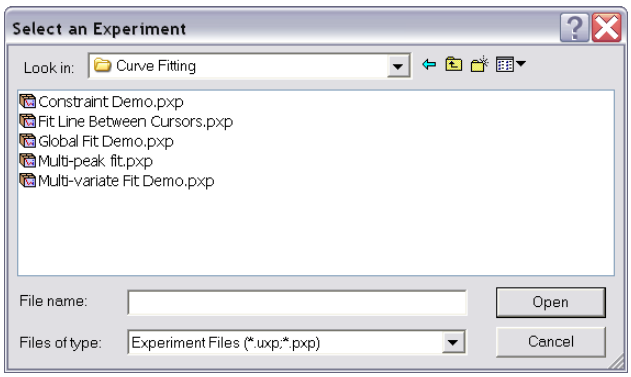
- You need to keep the experiment file and folder together when you move the experiment to another disk.

If you create an experiment with a very large number of waves, you might find it convenient to save it as an unpacked experiment while you are actively working with it and later do a Save Experiment As using the packed format for archiving.

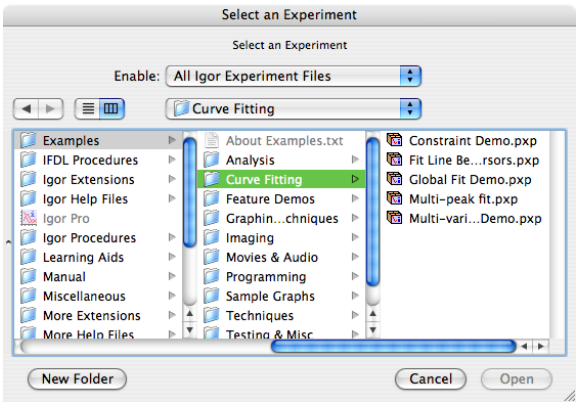
Opening Experiments

You can open an experiment stored on disk by choosing Open Experiment from the File menu. You can first save your current experiment if it has been modified. Then Igor presents the standard Open File dialog.

Windows



Macintosh



When you select an experiment and click the Open button, Igor loads the experiment including all waves, variables, graphs, tables, page layouts, notebooks, procedures and other objects that constitute the experiment.

Some people mistakenly believe that Igor recreates an experiment by reexecuting its history. See **How Experiments Are Loaded** on page II-39 for the real story.

Merging Experiments

Normally Igor closes the currently opened experiment before opening a new one. But it is possible to merge the contents of an experiment file into the current experiment. This is useful, for example, if you want to create a page layout that contains graphs from two or more experiments. To do this, press Option (*Macintosh*) or Alt (*Windows*) while choosing Open Experiment from the File menu.

Note: *Merging experiments is an advanced feature that has some inherent problems and should be used judiciously*. If you are just learning to use Igor Pro, you should avoid merging experiments until you have become proficient. You may want to skim the rest of this section or skip it entirely. It assumes a high level of familiarity with Igor.

The first problem is that the merge operation creates a copy of data and other objects (e.g., graphs, procedure files, notebooks) stored in a packed experiment file. Whenever you create a copy there is a possibility that copies will diverge, creating confusion about which is the "real" data or object. One way to avoid this problem is to discard the merged experiment after it has served its purpose.

The second problem has to do with Igor's use of names to reference all kinds of data, procedures and other objects. When you merge experiment B into experiment A, there is a possibility of name conflicts.

Igor prevents name conflicts for data (waves, numeric variables, string variables) by creating a new data folder to contain the data from experiment B. The new data folder is created inside the current data folder of the current experiment (experiment A in this case).

For other globally named objects, including graphs, tables, page layouts, control panels, notebooks, symbolic paths, page setups and pictures, Igor renames objects from experiment B if necessary to avoid a name conflict.

During the merge experiment operation, Igor looks for conflicts between target windows, between window recreation macros and between a target window and a recreation macro. If any such conflict is found, the window or window macro from experiment B is renamed.

Because page layouts reference graphs, tables and pictures by name, renaming any of these objects may affect a page layout. The merge experiment operation handles this problem for page layouts that are open in experiment B. It does not handle the problem for page layout recreation macros in experiment B that have no corresponding open window.

If there are name conflicts in procedures other than window recreation macros, Igor will flag an error when it compiles procedures after finishing the merge experiment operation. You will have to manually resolve the name conflict by removing or renaming conflicting procedures.

Procedure windows have titles but do not have standard Igor names. The merge experiment operation makes no attempt to retitle procedure windows that have the same title.

The contents of the main procedure window from experiment B are appended to the contents of the main procedure window for experiment A.

During a normal experiment open operation, Igor executes experiment initialization commands. This is not done during an experiment merge.

Each experiment contains a default font setting that affects graphs and page layouts. When you do an experiment merge, the default font setting from experiment B is ignored, leaving the default font setting for experiment A intact. This may affect the appearance of graphs and layouts in experiment B.

The history from experiment B is not merged into experiment A. Instead, a message about the experiment merge process is added to the history area.

The system variables (K0…K19) from experiment B are ignored and not merged into experiment A.

Although the merge experiment operation handles the most common name conflict problems, there are a number problems that it can not handle. For example, a procedure, dependency formula or a control from experiment B that references data using a full path may not work as expected because the data from experiment B is loaded into a new data folder during the merge. Another example is a procedure that references a window, symbolic path or picture that is renamed by the merge operation because of a name conflict. There are undoubtedly many other situations where name conflicts could cause unexpected behavior.

Reverting an Experiment

If you choose Revert Experiment from the File menu, Igor asks if you're sure that you want to discard changes to the current experiment. If you answer Yes, Igor reloads the current experiment from disk, restoring it to the state it was in when you last saved it.

New Experiments

If you choose New from the File menu, Igor first asks if you want to save the current experiment if it was modified since you last saved it. Then Igor creates a new, empty experiment. The new experiment has no experiment file until you save it.

By default, when you create a new experiment, Igor automatically creates a new, empty table. This is convenient if you generally start working by entering data manually. However, in Igor data can exist in memory without being displayed in a table. If you wish, you can turn automatic table creation off using the Experiment Settings category of the Miscellaneous Settings dialog (Misc menu).

Saving an Experiment as a Template

A template experiment provides a way to customize the initial contents of a new experiment. When you open a template experiment, Igor opens it normally but leaves it untitled and disassociates it from the template experiment file. This leaves you with a new experiment based on your prototype. When you save the untitled experiment, Igor creates a new experiment file.

Packed template experiments have ".pxt" as the file name extension instead of ".pxp". Unpacked template experiments have ".uxt" instead of ".uxp".

To make a template experiment, start by creating a prototype experiment with whatever waves, variables, procedures and other objects you would like in a new experiment.

On Macintosh, choose File→Save Experiment As, check the Save as Stationery checkbox, and save the template experiment.

On Windows, choose File→Save Experiment As, choose Packed Experiment Template or Unpacked Experiment Template from the "Save as type" menu, and save the template experiment.

You can convert an existing experiment file into a template file by changing the extension (".pxp" to ".pxt" or ".uxp" to ".uxt").

The Macintosh Finder's file info window has a Stationery Pad checkbox. Checking it turns a file into a stationery pad. When you double-click a stationery pad file, Mac OS X creates a copy of the file and opens the copy. For most uses, the template technique is more convenient.

Browsing Experiments

You can see what data exists in the current experiment as well as experiments saved on disk using the Data Browser. To open the browser, choose Data→Data Browser. The Data Browser is described in Chapter II-8, **Data Folders**.

Symbolic Paths

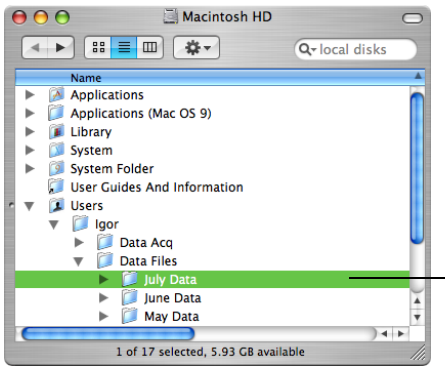
A symbolic path is an Igor object that associates a short name with a folder on a disk drive. You can use this short name instead of a full path to specify a folder when you load, open or save a file. A full path is a complete specification of the location of a folder on a disk drive, as illustrated in the next section.

Igor creates some symbolic paths automatically and you can also create symbolic paths.

Symbolic Path Example

This example is intended to illustrate why you should use symbolic paths and how to use them. We will assume that you have a folder full of text files containing data that you want to graph in Igor and that the organization of your hard disk is as follows:

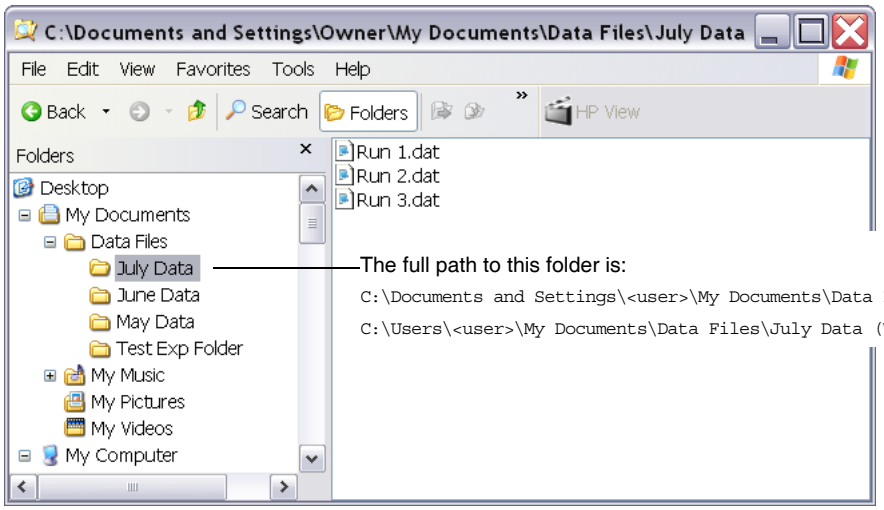
Macintosh



The full path to this folder is:

```
Macintosh HD:Users:Igor:Data Files:July Data
```

Windows



The full path to this folder is:

```
C:\Documents and Settings\<user>\My Documents\Data Files\July Data (Windows XP)
C:\Users\<user>\My Documents\Data Files\July Data (Windows VISTA, Windows 7)
```

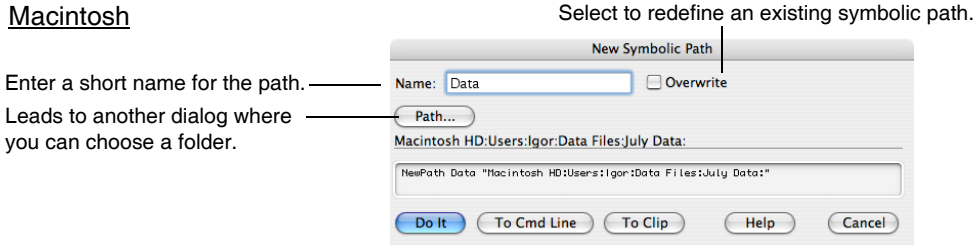
To create a symbolic path for the folder, choose New Path from the Misc menu. This leads to the New Symbolic Path dialog.

The NewPath command created by the dialog makes a symbolic path named Data which represents:

```
Macintosh HD:Users:Igor:Data Files:July Data: (Macintosh)
C:\Documents and Settings\<user>\My Documents\Data Files\July Data\ (Windows XP)
C:\Users\<user>\My Documents\Data Files\July Data\ (Windows VISTA and 7)
```

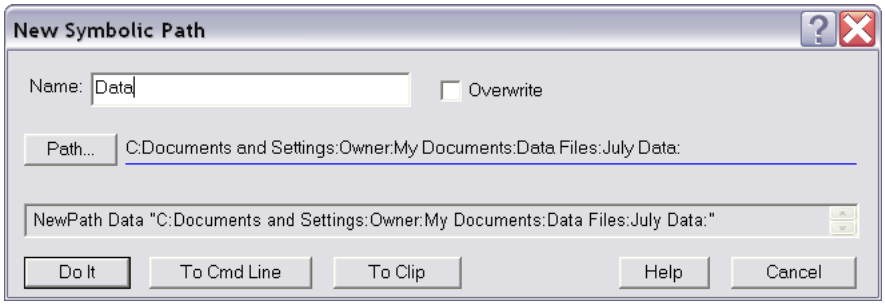
Note that on Windows, the NewPath command generated by the dialog has a Macintosh-style path using colons to separate the folder levels. The command can also accept Windows-style paths with backslash characters, but this can cause problems and is not recommended. For details, see **Path Separators** on page III-398.

Macintosh



Windows

Note that Igor generates a command using Macintosh style with colons separating folder levels.



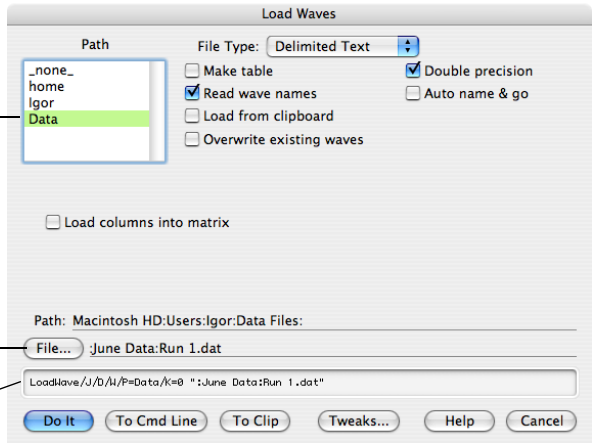
Once you've executed this, you can select the Data path in dialogs where you need to choose a folder.

For example, the Load Waves dialog would look like this:

The "Data" symbolic path appears in this list of paths. When you select it, you specify that you want to load a file from the "Hard Disk:...:July Data" folder.

When you click, Igor asks you to choose a file from the folder.

Igor uses the symbolic path name as a shorthand reference to the folder.



You can also use the symbolic path in commands that you execute from the command line or from Igor procedures. Typically this is done using a /P=<symbolic path name> flag.

Automatically Created Paths

Igor automatically creates a symbolic path named **Igor** which refers to the folder containing the Igor application. This is mainly of interest if you write Igor procedures.

Igor also automatically creates the **home** symbolic path. This path refers to the home folder for the current experiment. For unpacked experiments, this is the experiment folder. For packed experiments, this is the folder containing the experiment file. For new experiments that have never been saved, home is undefined.

Finally, Igor automatically creates a symbolic path if you do something that causes the current experiment to *reference* a file not stored as part of the experiment. This happens when you:

- Load an Igor Binary file from another experiment into the current experiment
- Open a notebook file not stored with the current experiment
- Open a procedure file not stored with the current experiment

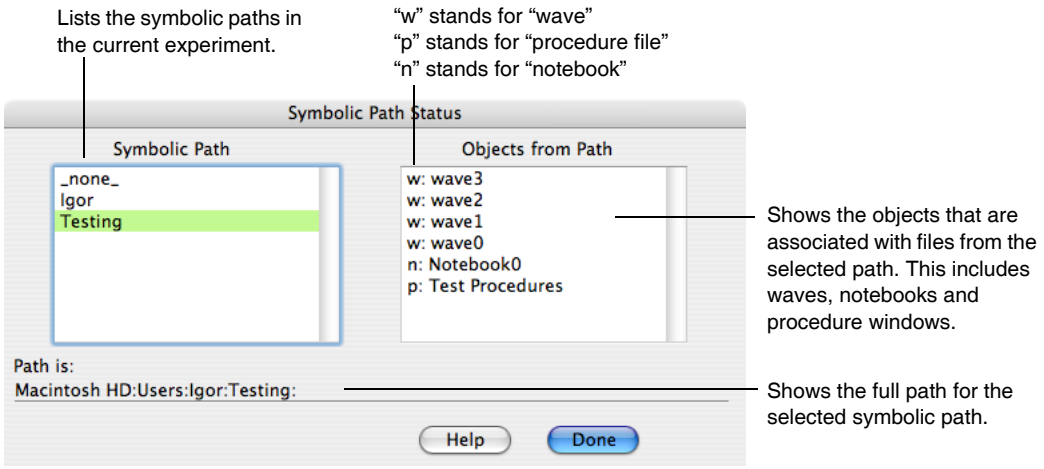
Creating these paths makes it easier for Igor to find the referenced files if they are renamed or moved. See **References to Files and Folders** on page II-37 for more information.

New Symbolic Path Dialog

To access the New Symbolic Path dialog, choose New Path from the Misc menu. The dialog is illustrated in the example **Symbolic Path Example** on page II-34.

Symbolic Path Status Dialog

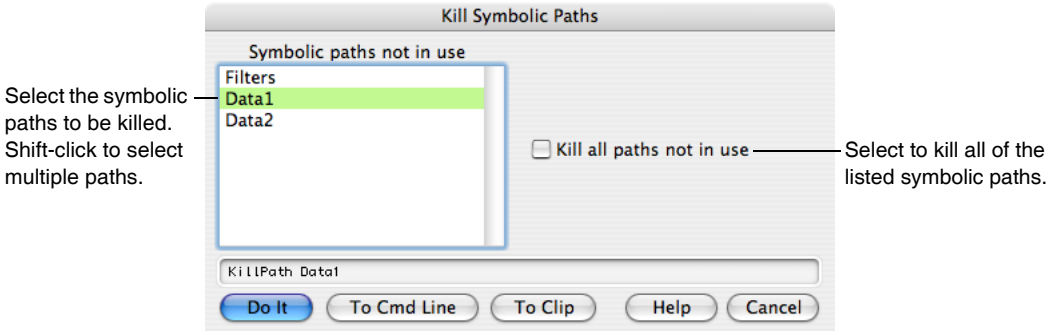
The Symbolic Path Status dialog shows you what paths exist in the current experiment. To invoke it, choose Path Status from the Misc menu.



If you click <none>, the Objects from Path list shows objects that are not associated with any of the symbolic paths. This includes waves that have not yet been saved to disk and waves, notebooks and procedure files stored in packed experiment files.

Kill Paths Dialog

The Kill Symbolic Paths dialog removes from the current experiment symbolic paths that you no longer need. Killing a path does nothing to the folder referenced by the symbolic path. It just deletes the symbolic path name from Igor's list of symbolic paths. To invoke the dialog, choose Kill Paths from the Misc menu.



A symbolic path is in use — and Igor won't let you kill it — if the experiment contains a wave, notebook window or procedure window linked to a file in the folder the symbolic path points to.

References to Files and Folders

An experiment can *reference* files that are not stored with the experiment. This happens when you load an Igor Binary data file which is stored with a different experiment or is not stored with any experiment. It also happens when you open a notebook or procedure file that is not stored with the current experiment. We say the current experiment is *sharing* the wave, notebook or procedure file.

For example, imagine that you open an existing text file as a notebook and then save the experiment. The data for this notebook is in the text file somewhere on your hard disk. It is not stored in the experiment. What *is* stored in the experiment is a *reference* to that file. Specifically, the experiment file contains a command that will reopen the notebook file when you next reopen the experiment.

Note: When an experiment refers to a file that is not stored as part of the experiment, there is a potential problem. If you copy the experiment to a CD to take it to another computer, for example, the experiment file on the CD will contain a *reference* to a file on your hard disk. If you open the experiment on the other computer, Igor will ask you to find the referenced file. If you have forgotten to also copy the referenced file to the other computer, Igor will not be able to completely recreate the experiment.

For this reason, we recommend that you use references only when necessary and that you be aware of this potential problem.

If you transfer files between platforms file references can be particularly troublesome. See **Experiments and Paths** on page III-395.

Avoiding Shared Igor Binary Files

When you load a wave from an Igor Binary file stored in another experiment, you need to decide if you want to *share* the wave with the other experiment or *copy* it to the new experiment. Sharing creates a reference from the current experiment to the wave's file and this reference can cause the problem noted above. Therefore, you should avoid sharing unless you want to access the same data from multiple experiments *and* you are willing to risk the problem noted above.

If you load the wave via the Load Igor Binary dialog or via the Browse Waves dialog, Igor will ask you if you want to share or copy. You can use the Miscellaneous Settings dialog to always share or always copy instead of asking you.

If you load the wave via the LoadWave operation, from the command line or from an Igor procedure, Igor will *not* ask what you want to do. You should normally use this operation's /H flag, which uses "copy the wave to home" and avoids sharing.

If you use the Data Browser to transfer waves from one experiment to another, Igor always copies the waves.

Adopting Notebook and Procedure Files

Adoption is a way for you to copy a notebook or procedure file into the current experiment and break the connection to its original file. The reason for doing this is to make the experiment self-contained so that, if you transfer it to another computer or send it to a colleague, all of the files needed to recreate the experiment will be stored in the experiment itself.

To adopt a file, choose Adopt Window from the File menu. This item will be available only if the active window is a notebook or procedure file that is stored separate from the current experiment *and the current experiment has been saved to disk*.

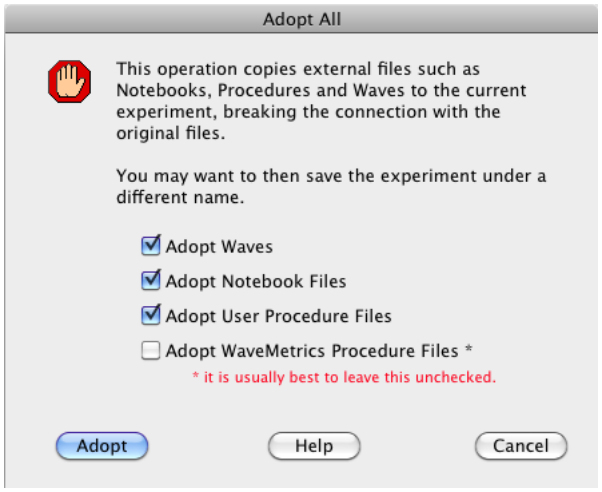
If the current experiment is stored in packed form then, when you adopt a file, Igor does a save-as to a temporary file. When you subsequently save the experiment, the contents of the temporary file are stored in the packed experiment file. Thus, the adoption is not finalized until you save the experiment.

If the current experiment is stored in unpacked form then, when you adopt a file, Igor does a save-as to the experiment's home folder. When you subsequently save the experiment, Igor updates the experiment's recreation procedures to open the new file in the home folder instead of the original file. Note that if you adopt a file in an unpacked experiment and then you do not save the experiment, the new file will still exist in the home folder but the experiment's recreation procedures will still refer to the original file. Thus, you should save the experiment after adopting a file.

To "unadopt" a procedure or notebook file, choose Save Procedure File As or Save Notebook As from the File menu.

Adopt All

You can adopt all referenced notebooks, procedure files and waves by pressing Shift and choosing File→Adopt All. This is useful when you want to create a self-contained packed experiment to send to someone else.



After clicking Adopt, choose File→Save Experiment As to save the packed experiment.

How Experiments Are Loaded

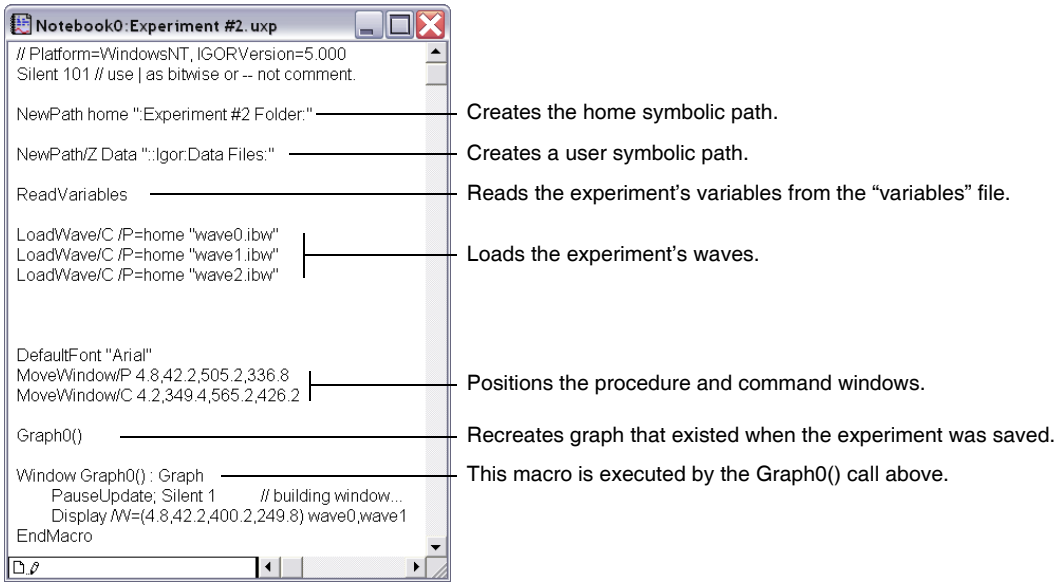
It is not essential to know how Igor stores your experiment or how Igor recreates it. However, understanding this may help you avoid some pitfalls and increase your overall understanding of Igor.

Experiment Recreation Procedures

When you save an experiment, Igor creates procedures and commands, called "experiment recreation procedures" that Igor will execute the next time you open the experiment. These procedures are normally not visible to you. They are stored in the experiment file.

The experiment file of an unpacked experiment contains plain text, but its extension is not ".txt", so you can't open it with most word processors or editors. You can open it by choosing File→Open File→Notebook and then selecting All Documents from the Show pop-up menu (*Macintosh*) or All Files from the Files Of Type pop-up menu (*Windows*). This is not something you would normally do, but it can be instructive.

As an example, let's look at the experiment recreation procedures for a very simple experiment.



When you open the experiment, Igor reads the experiment recreation procedures from the experiment file into the procedure window and executes them. The procedures recreate all of the objects and windows that con-

stitute the experiment. Then the experiment recreation procedures are removed from the procedure window and your own procedures are loaded from the experiment's procedure file into the procedure window.

For a packed experiment, the process is the same except that all of the data, including the experiment recreation procedures, is packed into the experiment file.

Experiment Initialization Commands

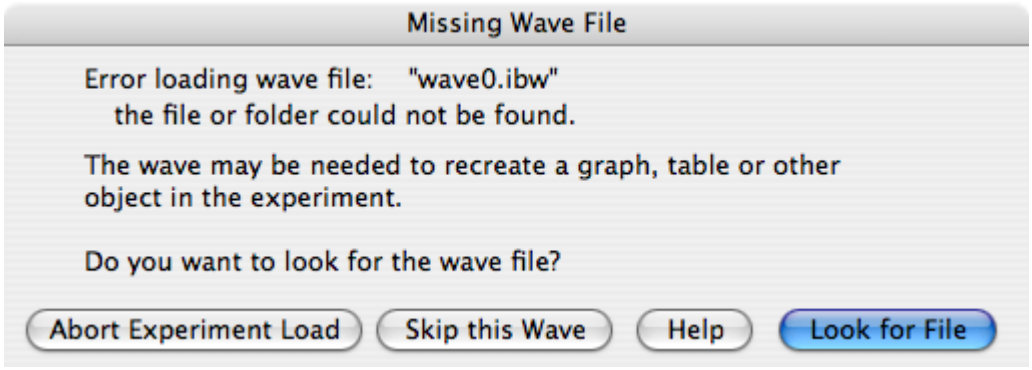
After executing the experiment recreation procedures and loading your procedures into the procedure window, Igor checks the contents of the procedure window. Any commands that precede the first macro, function or menu declaration are considered **initialization commands**. If you put any initialization commands in your procedure window then Igor executes them. This mechanism initializes an experiment when it is first loaded.

Savvy Igor programmers can also define a function that is executed whenever Igor opens any experiment. See **User-Defined Hook Functions** on page IV-251.

Errors During Experiment Load

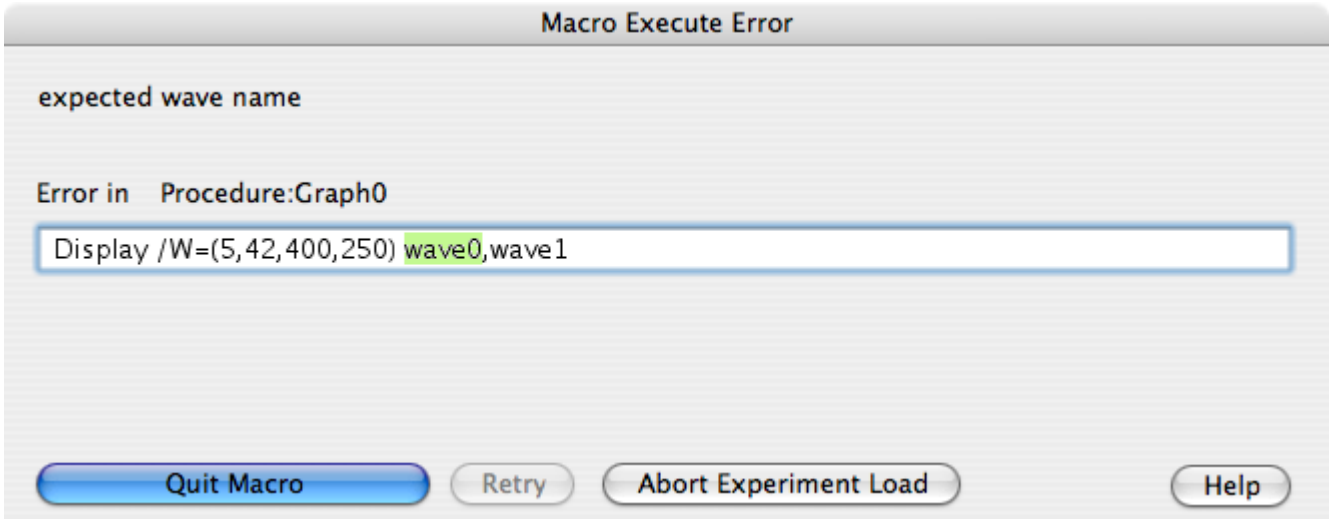
It is possible for the experiment loading process to fail to run to a normal completion. This occurs most often when you run out of memory or when you move or rename a file or folder and you can't help Igor find it. It also happens if you move an experiment to a different computer and forget to also move referenced files or folders. See **References to Files and Folders** on page II-37 for details.

These errors occur while Igor is executing the experiment recreation procedures. Igor uses several techniques to try to find the missing file or folder (see **How Igor Searches for Missing Folders** on page II-41). The techniques include asking you for help via a dialog like this:



If you elect to abort the experiment load, Igor will alert you that the experiment is in an inconsistent state. It displays some diagnostic information that might help you understand the problem and changes the experiment to Untitled. You should use the New Experiment or Open Experiment items in the File menu to clear out the partially loaded experiment.

If you elect to skip loading a wave file, you may get another error later, when Igor tries to display the wave in a graph or table. In that case, you will see a dialog like this:



In this example, Igor is executing the Graph0 macro from the experiment recreation procedures in an attempt to recreate a graph. Since you elected to skip loading wave0, Igor can't display it.

You have three options at this point, as explained in the following table.

Option	Effect
Quit Macro	Stops executing the current macro but continues experiment load. In this example, Graph0 would not be recreated. After the experiment load Igor displays diagnostic information.
Abort Experiment Load	Aborts the experiment load immediately and displays diagnostic information.
Fix Macro	In this example, you could fix the macro by deleting "wave0,". You would then click the Retry button. Igor would create Graph0 without wave0 and would continue the experiment load.

With the first two options, Igor leaves the experiment untitled so that you don't inadvertently wipe out the original experiment file by doing a save.

How Igor Searches for Missing Folders

When Igor saves an experiment, it stores commands in the experiment file that will recreate the experiment's symbolic paths when you reopen the experiment. The commands look something like this (in the fourth line, under *Windows* the path would start with "C:"):

```
NewPath home ":Test Exp Folder:"
NewPath/Z Data1 "::Data Folder #1:"
NewPath Data2 "::Data Folder #2:"
NewPath/Z Data3 "hd:Test Runs:Data Folder #3:"
```

The location of the home folder is specified relative to the experiment file. The locations of all other folders are specified relative to the experiment folder or, if they are on a different volume, using absolute paths. Using relative paths, where possible, ensures that no problems will arise if you move the experiment file and experiment folder *together* to another disk drive or another location on the same disk drive.

The /Z flags indicate that the experiment does not need to load any files from the Data1 and Data3 folders. In other words, the experiment has symbolic paths for these folders but no files need to be loaded from them to recreate the experiment.

When you reopen the experiment, Igor executes these NewPath commands. If you have moved or renamed folders or if you have moved the experiment file, the NewPath operation will be unable to find the folder. Here is what Igor does in this case.

First of all, if the symbolic path is not needed to recreate the experiment then Igor does nothing. It generates no error and just continues the load. The experiment will wind up without the missing symbolic path.

If the missing folder *is* needed to load some object then Igor will search for it using a number of techniques. The search uses additional information that Igor stores in the experiment file when the experiment is saved. This includes such things as the full path to the folder, the folder's "directory ID" and an "alias record", which are explained in the next section.

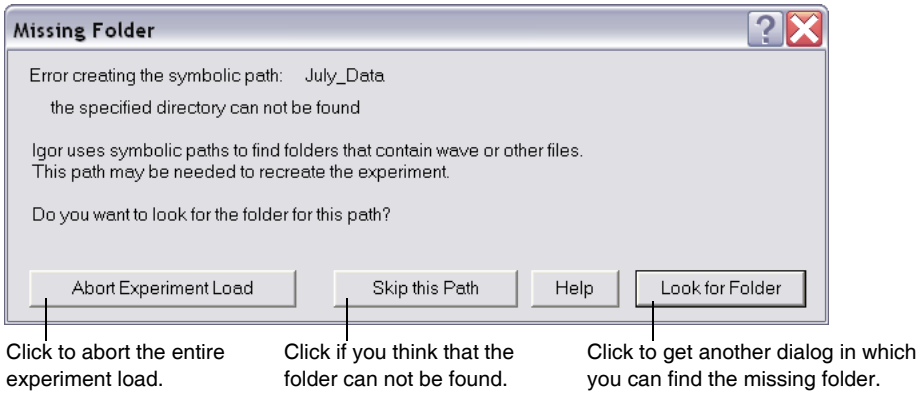
Folder Search Techniques

1. Search by full path
   A full path is one that starts from the volume that the folder is on. An example is "hd:Test Runs:Data Folder #3:" (*Macintosh*) or "C:\Test Runs\Data Folder #3\" (*Windows*).
   This technique will find the folder if the full path to the folder is the same as when the experiment was saved. This handles the case where you moved or renamed the experiment folder but did not move or rename the missing folder.
2. Search by directory ID (*Macintosh only*)
   The directory ID is a unique number that the Macintosh file system assigns to each folder on a particular volume. This technique will find the folder if the missing folder was moved or renamed but not moved to a different volume.
3. Search using Alias Manager (*Macintosh only*)
   The Alias Manager is a Macintosh feature designed to locate missing files and folders. If Igor is trying to create a symbolic path that is not needed to recreate the experiment then Igor will skip the Alias Manager search. The Alias Manager's attempt to mount network volumes is generally not desirable if the folder is not critical for the experiment load.
4. Search of the Igor Pro Folder
   If the path is a full path that points to a folder inside the Igor Pro Folder (e.g., "hd:Igor Pro Folder:User Procedures" on a Macintosh), then Igor looks for the folder inside the Igor Pro Folder, even if the Igor Pro Folder is on a differently-named root volume (e.g., "C:\Igor Pro Folder\User Procedures" on a PC).
5. Search of the volume containing Igor
   If the path is a full path (e.g., "hd:Igor Work:Data Files" on a Macintosh) then Igor searches for the folder at the same location but on the volume containing the Igor application (e.g., "C:\Igor Work\Data Files" on a PC).
6. Search of the volume containing the experiment file
   This is the same as the previous search except that Igor uses the volume containing the experiment file.

The last three techniques are designed to help find files when you move an experiment from one platform to another. They work only if the path is a full path, which will be the case if the target folder is on a different volume from the experiment file. If this is not the case, then the path will be relative to the experiment file and Igor will be able to find the target folder if it has the same relationship to the experiment file on both platforms.

Note: If you use Igor on both Macintosh and Windows, it is best if you use the same folder hierarchy for your Igor files on both computers. This will give Igor the best chance of automatically finding missing folders.

If all of these techniques fail, Igor asks if you want to look for the folder by putting up a the Missing Folder dialog.



If you click the Look for Folder button, Igor presents another dialog in which you can find the missing folder.

If you click Skip this Path in the Missing Folder dialog then Igor will not create the symbolic path and therefore you will get one or more errors later, when Igor tries to use it. For example, if the experiment loads two waves using the Data2 path then the experiment's recreation commands would contain two lines like this:

```
LoadWave/C/P=Data2 "wave0.bwav"
LoadWave/C/P=Data2 "wave1.bwav"
```

If you were unable to find the Data2 folder then each of these LoadWave commands will present the Missing Wave File dialog.

If you are unable to find the wave file and if the wave is used in a graph or table, you will get more errors later in the experiment recreation process, when Igor tries to use the missing wave to recreate the graph or table.

How Experiments Are Saved

When you save an experiment for the first time, Igor just does a straight-forward save in which it creates a new file, writes to it, and closes it. However, when you resave a pre-existing experiment, which overwrites the previous version of the experiment file, Igor uses a "safe save" technique. This technique is designed to preserve your original data in the event of an error during the save.

For purposes of illustration, we will assume that we are resaving an experiment file named "Experiment.pxp". The safe save proceeds as follows:

1. Write the new data to a temporary file named "Experiment.pxpT0". If an error occurs during this step, the save operation is stopped and Igor displays an error message.
2. Delete the original file, "Experiment.pxp".
3. Rename the temporary file with the original name. That is, rename "Experiment.pxpT0" as "Experiment.pxp".

On Windows, the temporary file name is "Experiment.pxpT0" but on Macintosh it is "Experiment.pxpT0.noindex". The ".noindex" suffix tells Apple's Spotlight program not to interfere with the save by opening the temporary file at an inopportune time.

The next three subsections are for use in troubleshooting file saving problems only. If you are not having a problem, you can skip them.

Experiment Save Errors

There are many reasons why an error may occur during the save of an experiment. For example, you may run out of disk space, the server volume you are saving to might be disconnected, or you may have a hardware failure, but these are uncommon.

The most common reason for a save error is that you cannot get write access to the file because:

1. The file is locked (Macintosh Finder) or marked read-only (Windows desktop).

2. You don't have permission to write to the folder containing the file.
3. You don't have permission to write to this specific file.
4. The file has been opened by another application. This could be a virus scanner, an anti-spyware program or an indexing program such as Apple's Spotlight.

Here are some troubleshooting techniques.

Macintosh File Troubleshooting

Open the file's Get Info window and verify that the file is not marked as locked. Also check the lock setting of the folder containing the file.

Next try doing a Save As to a folder for which you know you have write access, for example, to your home folder (e.g., "/Users/<user>" where <user> is your user name). If this works, the problem may be that you did not have sufficient permissions to write to the original folder or to the original file. This would happen, for example, if the folder was inside the Applications folder and you are not running as an administrator.

If you think you should be able to write to the original file location, look at the Ownership and Permissions section (Mac OS X 10.4) or the Sharing and Permissions section (Mac OS X 10.5) of the Get Info window for both the file and the folder containing it and make sure that you have read/write access.

If you are able to save a file to a new location but get an error when you try to resave the file, which overwrites the original file, then this may be an issue of another program opening the file at an inopportune time. This typically happens in step 3 of the safe-save technique described above. Try disabling your antivirus software. For a technical explanation of this problem, see http://developer.apple.com/qa/qa2006/qa1497.html.

Windows File Troubleshooting

Open the file's Properties window and uncheck the read-only checkbox if it is checked. Do the same for the folder containing the file.

Next try doing a Save As to a folder for which you know you have write access, for example, to your home folder (e.g., My Documents). If this works, the problem may be that you did not have sufficient permissions to write to the original folder or to the original file. This would happen, for example, if the folder was inside the Program Files folder and you are not running as an administrator.

If you think you should be able to write to the original file location, you will need to investigate permissions. By default, Windows runs in "Simple File Sharing" mode. In this mode, a file or folder's Properties window does not tell you anything about permissions. Therefore, to investigate, you must leave simple file sharing mode. You may want to enlist the help of a local expert as this can get complicated.

You turn simple file sharing off by choosing Tools→Folder Options from any folder's window. In the Folder Options dialog, click the View tab and uncheck the Use Simple File Sharing checkbox at the bottom of the tab. When you next display the Properties dialog for a file or folder, it will include a Security tab in which you can inspect permissions.

In the Security tab of the Properties window, make sure you have read/write permission and modify permission. The modify permission is required in order for Igor to delete the file in step 2 of the safe-save technique described above. Verify that you have write and modify permission for both the file and the folder containing it.

If you are able to save a file to a new location but get an error when you try to resave the file, which overwrites the original file, then this may be an issue of another program opening the file at an inopportune time. This typically happens in step 3 of the safe-save technique described above. Try disabling your antivirus software. For a technical explanation of this problem, see http://support.microsoft.com/kb/316609.

Special Folders

This section describes special folders that Igor automatically searches when looking for help files, Igor extensions (plug-ins that are also called XOPs) and procedure files.

The folder containing the Igor Pro application file is called the "Igor Pro Folder". Several subfolders in the Igor Pro Folder are treated specially, as described below.

At launch time, Igor creates another special folder, called the Igor Pro User Files folder, outside of the Igor Pro Folder. By default, this folder has the Igor Pro major version number in its name, for example, "Igor Pro 6 User Files", but it is generically called the "Igor Pro User Files" folder.

On Macintosh, the Igor Pro User Files folder is created by default in:

```
/Users/<user>/Documents/WaveMetrics
```

On Windows XP it is created by default in:

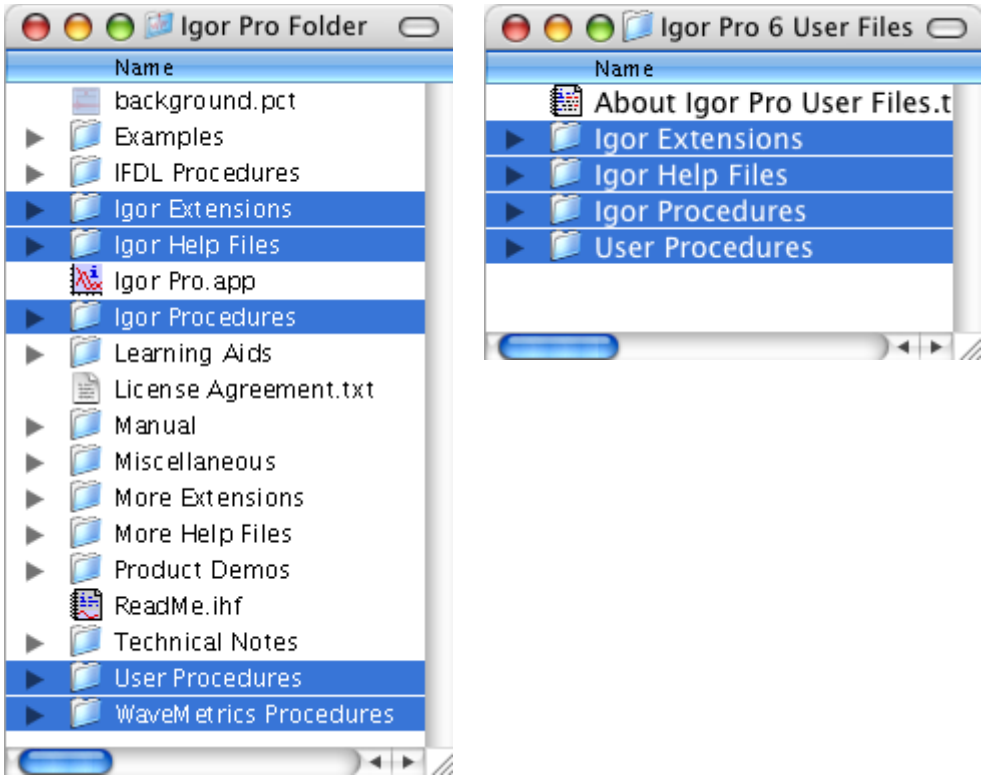
```
C:\Documents and Settings\<user>\My Documents\WaveMetrics
```

On Windows VISTA and Windows 7 it is created by default in:

```
C:\Users\<user>\My Documents\WaveMetrics
```

You can change the location of your Igor Pro User Files folder using Misc→Miscellaneous Settings but this should rarely be necessary.

Several subfolders in the Igor Pro User Files folder are also treated specially as described in the following sections. Here we see the Igor Pro Folder on the left and the Igor Pro User Files folder on the right with the special subfolders highlighted:



Help files, extensions and procedure files are active if they, or aliases or shortcuts pointing to them, are in the appropriate special subfolder. When you install Igor, special subfolders inside the Igor Pro Folder contain files that are active. Examples include the standard WaveMetrics help files, standard WaveMetrics extensions like the Excel file loader, and standard WaveMetrics procedure files that add items to Igor's built-in menus.

You may want to activate additional help files, extensions and procedure files that are part of the Igor Pro installation, that you create or that you receive from third parties. You can do this by adding files or

aliases/shortcuts pointing to files to the special subfolders within the Igor Pro Folder. However, it is better to use the special subfolders within the Igor Pro User Files folder because the Igor Pro Folder is not accessible to non-administrative users and because adding your own files to the Igor Pro Folder complicates backup and updating Igor.

Igor Pro Folder

The Igor Pro Folder is the folder containing the Igor application. Igor looks inside the Igor Pro Folder for these special subfolders: Igor Help Files, Igor Extensions, Igor Procedures, User Procedures and WaveMetrics Procedures.

The Igor installer puts files in the special folders. Igor searches them when looking for help files, extensions and procedure files. In most cases, you should not put files in these special folders - use the Igor Pro User Files folder instead.

Igor Pro User Files

Igor automatically creates the Igor Pro User Files folder at launch time if it does not already exist. Igor looks inside the Igor Pro User Files folder for these special subfolders: Igor Help Files, Igor Extensions, Igor Procedures, and User Procedures.

By default, the Igor Pro User Files folder has the Igor Pro major version number in its name, for example, "Igor Pro 6 User Files", but it is generically called the "Igor Pro User Files" folder.

The default location of the Igor Pro User Files folder is:

```
Macintosh:
   /Users/<user>/Documents/WaveMetrics/Igor Pro 6 User Files

Windows XP:
   C:\Documents and Settings\<user>\My Documents\WaveMetrics\Igor Pro 6 User Files

Windows VISTA and Windows 7:
   C:\Users\<user>\My Documents\WaveMetrics\Igor Pro 6 User Files
```

You can change the location of your Igor Pro User Files folder using Misc→Miscellaneous Settings but this should rarely be necessary.

You can display the Igor Pro User Files folder on the desktop by choosing Help→Show Igor Pro User Files. To display both the Igor Pro Folder and the Igor Pro User Files folder, press the shift key and choose Help→Show Igor Pro Folder and User Files.

You can put files or aliases/shortcuts pointing to files in these subfolders. Igor searches them when looking for help files, extensions and procedure files.

Igor Help Files Folder

When Igor starts up, it opens any Igor help files in "Igor Pro Folder/Igor Help Files" and in "Igor Pro User Files/Igor Help Files". It treats any aliases, shortcuts and subfolders in Igor Help Files in the same way.

Standard WaveMetrics help files are pre-installed in "Igor Pro Folder/Igor Help Files".

If there is an additional help file that you want Igor to automatically open at launch time, put it or an alias/shortcut for it in "Igor Pro User Files/Igor Help Files".

Igor Extensions Folder

When Igor starts up, it searches "Igor Pro Folder/Igor Extensions" and "Igor Pro User Files/Igor Extensions" for Igor extension files. These extensions are available for use in Igor. It treats any aliases, shortcuts and subfolders in Igor Extensions in the same way. See **Igor Extensions** on page III-423 for details.

Standard WaveMetrics extensions are pre-installed in "Igor Pro Folder/Igor Extensions".

If there is an additional extension that you want to use, put it or an alias/shortcut pointing to it in "Igor Pro User Files/Igor Extensions". Additional WaveMetrics extensions are described in the "XOP Index" help file and can be found in "Igor Pro Folder/More Extensions". You may also create your own Igor extensions or obtain them from third parties.

Igor Procedures Folder

When Igor starts up, it automatically opens any procedure files in "Igor Pro Folder/Igor Procedures" and in "Igor Pro User Files/Igor Procedures". It treats any aliases, shortcuts and subfolders in Igor Procedures in the same way. Such procedure files are called "global" procedure files and are available for use from all experiments. See **Global Procedure Files** on page III-345 for details.

Standard WaveMetrics global procedure files are pre-installed in "Igor Pro Folder/Igor Procedures".

If there is an additional procedure file that you want Igor to automatically open at launch time, put it or an alias/shortcut pointing to it in "Igor Pro User Files/Igor Procedures". Additional WaveMetrics procedure files are described in the "WM Procedures Index" help file and can be found in "Igor Pro Folder/WaveMetrics Procedures". You may also create your own global procedure files or obtain them from third parties.

User Procedures Folder

You can load a procedure file from another procedure file using a #include statement. This technique is used when one procedure file requires another. See **Including a Procedure File** on page III-346 for details.

When Igor encounters a #include statement, it searches for the included procedure file in "Igor Pro Folder/User Procedures" and in "Igor Pro User Files/User Procedures". Any aliases, shortcuts and subfolders in User Procedures are treated the same way.

If there is an additional procedure file that you want to include from your procedure files, put it or an alias/shortcut pointing to it in "Igor Pro User Files/User Procedures".

WaveMetrics Procedures Folder

The "Igor Pro Folder/WaveMetrics Procedures" folder contains an assortment of procedure files created by WaveMetrics that may be of use to you. These files are described in the WM Procedures Index help file which you can access through the Windows→Help Windows menu.

You can load a WaveMetrics procedure file from another procedure file using a #include statement. See **Including a Procedure File** on page III-346 for details.

There is no WaveMetrics Procedures folder in the Igor Pro User Files folder.

Activating Additional WaveMetrics Files

If you want to activate a WaveMetrics file that is stored in the Igor Pro Folder, make an alias or shortcut for the file and put it in the appropriate subfolder of the Igor Pro User Files folder.

For example, the HDF5 file loader package consists of an extension named HDF5.xop, a help file named "HDF5 Help.ihf" and a procedure file named "HDF5 Browser.ipf". Here is how you would activate these files:

1. Press the shift key and choose Help→Show Igor Pro Folder and User Files. This displays the Igor Pro Folder and the Igor Pro User Files folder on the desktop.
2. Make an alias/shortcut for "Igor Pro Folder/More Extensions/File Loaders/HDF5.xop" and put it in "Igor Pro User Files/Igor Extensions". This causes Igor to load the extension the next time Igor is launched.
3. Make an alias/shortcut for "Igor Pro Folder/More Extensions/File Loaders/HDF5 Help.ihf" and put it in "Igor Pro User Files/Igor Help Files". This causes Igor to automatically open the help file the next time Igor is launched. This step is necessary only if you want the help file to be automatically opened.
4. Make an alias/shortcut for "Igor Pro Folder/WaveMetrics Procedures/File Input Output/HDF5

Browser.ipf" and put it in "Igor Pro User Files/Igor Procedures". This causes Igor to load the procedure the next time Igor is launched and to keep it open until you quit Igor.

5. Restart Igor.

You can verify that the HDF5 extension and the HDF5 Browser procedure file were loaded by choosing Data→Load Waves→New HDF5 Browser. You can verify that the HDF5 Help file was opened by choosing Windows→Help Windows→HDF5 Help.ihf.

Activating Other Files

You may create an Igor package or receive a package from a third party. You should store each package in its own folder in the Igor Pro User Files folder or elsewhere, at your discretion. You should not store such files in the Igor Pro Folder because it complicates backup and updating.

To activate files from the package, create aliases/shortcuts for the package files and put them in the appropriate subfolder of the Igor Pro User Files folder.

If you have a single procedure file or a single Igor extension that you want to activate, you may prefer to put it directly in the appropriate subfolder of the Igor Pro User Files folder.

Activating Files in a Multi-User Scenario

Our recommendation is that you activate files using the special subfolders in the Igor Pro User Files folder, not in the Igor Pro Folder. An exception to this is the multi-user scenario where multiple users are running the same copy of Igor from a server. In this case, if you want to activate a file for all users, put the file or an alias/shortcut for it in the appropriate subfolder of the Igor Pro Folder. Users will have to restart Igor for the change to take effect.

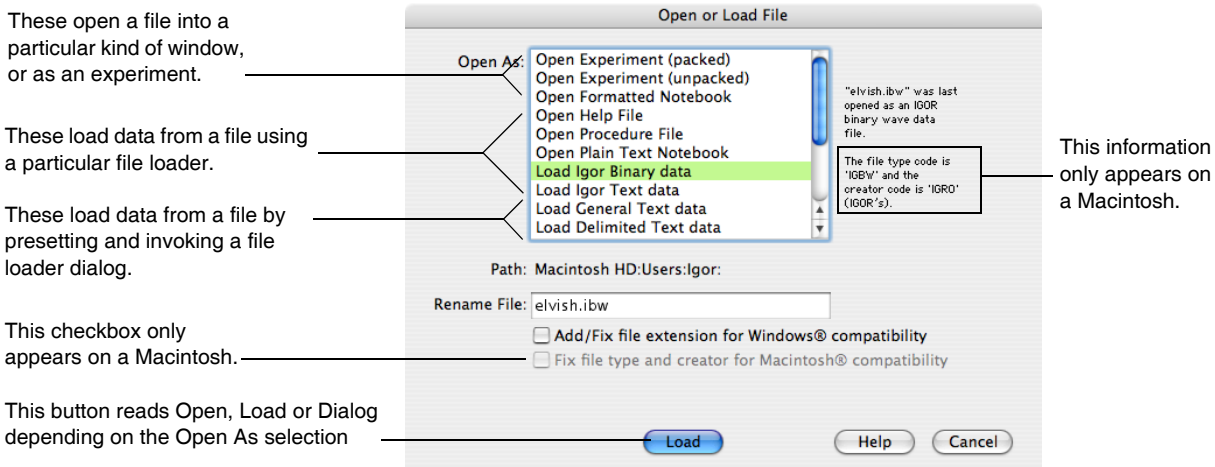
Igor File-Handling

Igor has many ways to open and load files. The following sections discuss some of the ways Igor deals with the various files it is asked to open.

Open or Load File Dialog

When you open a file using an open file dialog, there is no question of how Igor should treat the file. This is not always the case when you drop a file onto the Igor icon or double-click a file on the desktop.

Often, Igor can determine how to open or load a file, and it will simply do that without asking the you about it. Sometimes Igor recognizes that a file (such as a plain text file or a formatted Igor notebook) can be appropriately opened several ways, and will ask you what to do by bringing up the Open or Load File Dialog. The dialog presents a list of ways to open the file (usually into a window) or to load it as data. You can also change the file's name, extension, type, and creator code.



Tip: You can force this dialog to appear by holding down Shift when opening a file through the Recent Files or Recent Experiments menus, or when dropping a file onto the Igor icon.

This is especially useful for opening Igor help files as a notebook file for editing, or to open a notebook as a help file, causing Igor to compile it.

The list presents three kinds of methods for handling the file:

1. Open the file as a document window or an experiment.
2. Load the file as data without opening another file dialog.
3. Load the file as data through the Load Waves Dialog or a File Loader Extension dialog.

If you choose one of the Load <whatever> Dialog methods, Igor will open the selected dialog as if you had chosen it from the Load Waves submenu of the Data menu.

Information about the file, or about how it was most recently opened, is displayed to the right of the list. On a Macintosh, the file's type and creator codes are also shown. The complete path to the file is shown below the list.

You can rename the file, and it will be changed before the file is opened or loaded.

Use the Add/Fix file extension checkbox to conform the file's extension (one is added if necessary) to what is needed for the Windows operating system to automatically open the file with the Windows version of Igor when the file is double-clicked. Adding an extension may also helps the Mac OS X Finder determine how the file is to be handled.

While the Add/Fix file extension checkbox is selected, Igor will update the extension when you choose from the list. Igor stops updating the extension when you deselect the checkbox.

Macintosh: Mac OS X uses both the file's extension and its file type and creator codes when determining how a file should be handled. Select the Fix file type and creator checkbox to change file's type and creator codes so that the Mac OS will automatically open the file with the Macintosh version of Igor when the file is double-clicked, and so that it will be opened using the method selected in the list. (The file type depends on how you choose to open or load the file, but the creator is always changed to Igor's creator code.) The Fix file type and creator checkbox will be disabled if the type and creator are already correct.

Note: Changing the file's extension, file type, or creator does not convert the contents of the file in any way. If you open what is actually an Igor Binary data file by choosing Open Help File, WaveMetrics won't be held liable for the results!

The file renaming and the type/creator checkbox are provided to help users share files between the Macintosh and Windows versions of Igor.

Recent Files and Experiments

When you use a dialog to open or save an experiment or a file, Igor adds it to the Recent Experiments or Recent Files submenu (in the File menu). When you choose an item from these submenus, Igor opens the experiment or file the same way in which you last opened or saved it.

For example, if you last opened a text file as an unformatted notebook, selecting the file from Recent Files will again open the file as an unformatted notebook. If you loaded it as a general text data file, Igor will load it as data again.

Igor does not remember all the details of how you originally load a data file, however. If you load a text data file with all sorts of fiddly tweaks about the format, Igor won't load it using the those same tweaks. To guarantee that Igor does load the data correctly, use the appropriate Load Data dialog.

Selecting an experiment or file with Shift held down, will cause the Open or Load File Dialog to appear, in which you can choose how Igor will open or load that file.

Desktop Drag and Drop

On the desktop, you can drop one or more files of almost any type onto the Igor Pro icon. *Under Windows*, you can drag files into any Igor window. Igor will open files that it understands and ignore those that it doesn't. One use for this feature is to load multiple data files in one operation; simply select the data files and drop them on the Igor icon.

If the file has been opened or loaded recently (it is listed in the Recent Files or Recent Experiments menu), then Igor will reopen or reload it the same way. See **Recent Files and Experiments** on page II-49.

Sometimes Igor recognizes that a file (such as a plain text file or a formatted Igor notebook) can be appropriately opened several ways, and will ask you what to do by bringing up the Open or Load File Dialog. Igor also opens the dialog if it does not recognize the file.

Tip: Holding down Shift before releasing the mouse button to drop the files onto Igor forces the Open or Load File Dialog to be displayed.

This table shows how Igor attempts to handle various types of files.

File	File Extension	What Igor Does
Experiment file	.pxp or .uxp	Opens the experiment.
Igor Binary wave (.ibw or .bwav)	.ibw	Loads as a data file, creating waves.
Text file that appears to contain no data		Opens as plain-text notebook.
Text file that appears to contain data		Loads as a data file, creating waves.
Igor help file (.ihf)	.ihf	Opens as help file.
Igor procedure file (.ipf)	.ipf	Opens as procedure file.
Igor notebook file (.ifn or .ift)	.ifn or .ift	Opens as notebook.
Igor Extension	.xop	Error dialog. See **Activating Extensions** on page III-424.
Unknown file		Displays the Open or Load File Dialog.

Igor extension files or aliases (*Macintosh*) or shortcuts (*Windows*) for them must be in "Igor Pro User Files/Igor Extensions", "Igor Pro Folder/Igor Extensions" or in a subfolder when Igor is launched. Dragging an XOP onto the Igor icon will not activate an extension that wasn't in the Igor Extensions folder when Igor was launched.

Advanced programmers can customize Igor to handle specific types of files in different ways, such as automatically loading files with an XOP. See **User-Defined Hook Functions** on page IV-251.

Problems With File Names Using Non-ASCII Characters

This section explains errors that you may encounter when attempting to access files or folders containing characters that are not part of the current system encoding. For example, if you try to open a file with a Japanese name while running with English as your system language, you will get an error.

The simplest way avoid these problems is to limit file names to the standard ASCII characters. These have character codes ranging from 32 decimal to 127 decimal and include the upper- and lower-case English letters, digits, and common punctuation symbols. All encodings include these characters as a subset so file names consisting of these characters work regardless of which encoding is the system encoding.

If you experience these errors and do not want to change your file names, read the rest of this section to understand the fundamental problem and how to deal with it.

Igor internally stores file paths using the "system encoding".

On Mac OS X, the system encoding in effect for Igor is determined by the first language in the Languages list in the International Preferences panel at the time the Finder is launched. Typical system encodings are "Mac Roman", for Western European languages, and "Mac Japanese".

On Windows, the system encoding in effect for Igor is determined by the "Language for non-Unicode Programs" setting in the Advanced tab of the Regional and Language Options control panel at the time the operating system starts up.

Operating systems store file and folder information using Unicode - a character encoding system that can represent virtually all characters in all languages.

In the ideal world, Igor would be a Unicode program and dealing with a Unicode-based file system would be easy. However, converting Igor, as well as all XOPs, to Unicode, is a massive task, so we have deferred it. This means that Igor must internally convert Unicode file names and paths to the system encoding.

The conversion of Unicode file and folder names to system encoding brings some potential problems with it. The problems stem from the fact that Unicode can represent any character while the system encoding can represent only a specific subset of characters.

Imagine that you are running on Mac OS X with Mac Roman as the system encoding and you try to open a file with a Japanese file name. Igor gets the Unicode version of the Japanese file name from Mac OS X and tries to convert it to Mac Roman. This causes an error because Japanese characters can not be represented in the Mac Roman encoding. A similar error occurs on Windows.

In order to access files with Japanese names, or use paths containing Japanese names, you must run with Japanese as the system encoding. If you have booted the system with Japanese as the preferred language then the system encoding is Japanese and you can access Japanese files. If another language is preferred then you must make Japanese the preferred language and then reboot. (In Mac OS X you can also log out and back in or relaunch the Finder by pressing option key while clicking the Finder icon in the dock.) Then relaunch Igor.

A similar problem arises if you are running with Japanese as the system encoding and you try to access a file whose name contains English special characters like the bullet character. When Igor tries to convert the Unicode representation of the file name to Japanese, it gets an error because the English bullet character is not in the Japanese encoding. In a case like this, you would have to switch the preferred language to English to access the file from Igor.

Chapter
II-4

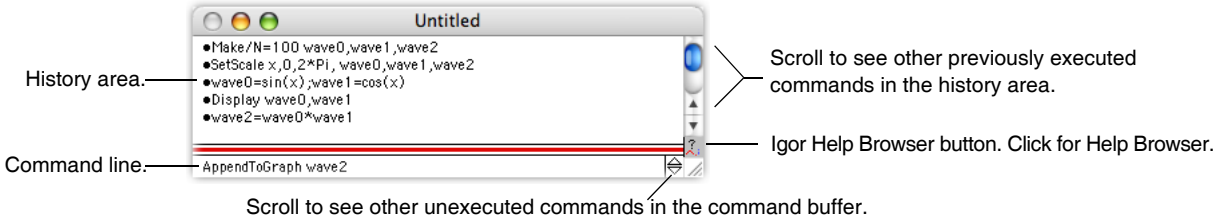
Windows

Overview

This chapter describes Igor's windows in general terms, just a bit about the File menu, and the Windows menu and window recreation macros in detail.

The Command Window

When Igor first starts, the **command window** appears at the bottom of the screen:



Commands are automatically entered and executed in the command window's **command line** when you use the mouse to "point-and-click" your way through dialogs. You may optionally type the commands directly and press Return or Enter. Igor preserves a history of executed commands in the **history area**.

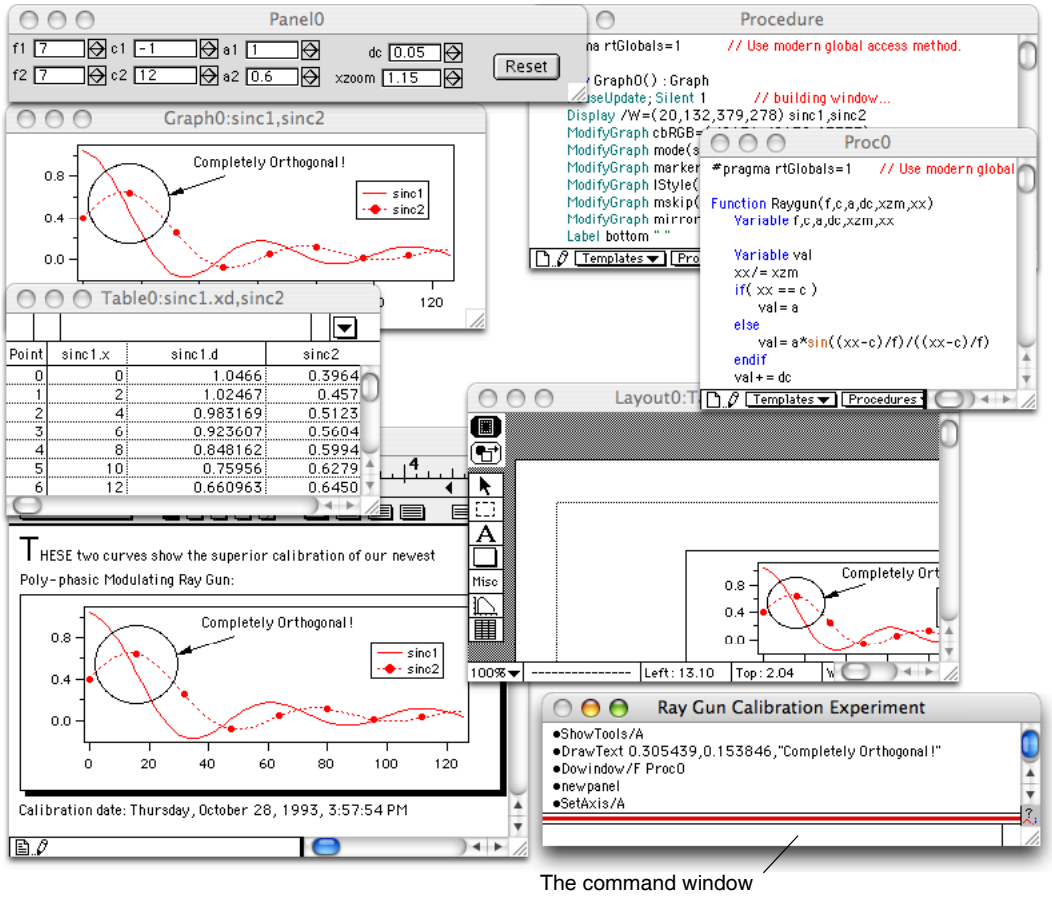
For more about the command window, see Chapter II-2, **The Command Window**, and Chapter IV-1, **Working with Commands**.

The Rest of the Windows

At startup, by default, Igor displays a table window. There are also a number of additional windows which are initially hidden:

- The main procedure window
- The Igor Help Browser
- Help windows for files in "Igor Pro Folder/Igor Help Files" and "Igor Pro User Files/Igor Help Files"

You can create additional windows for graphs, tables, page layouts, notebooks, panels and auxiliary procedure windows, as well as more help windows.



See the section **Chapters About Specific Windows** on page II-72.

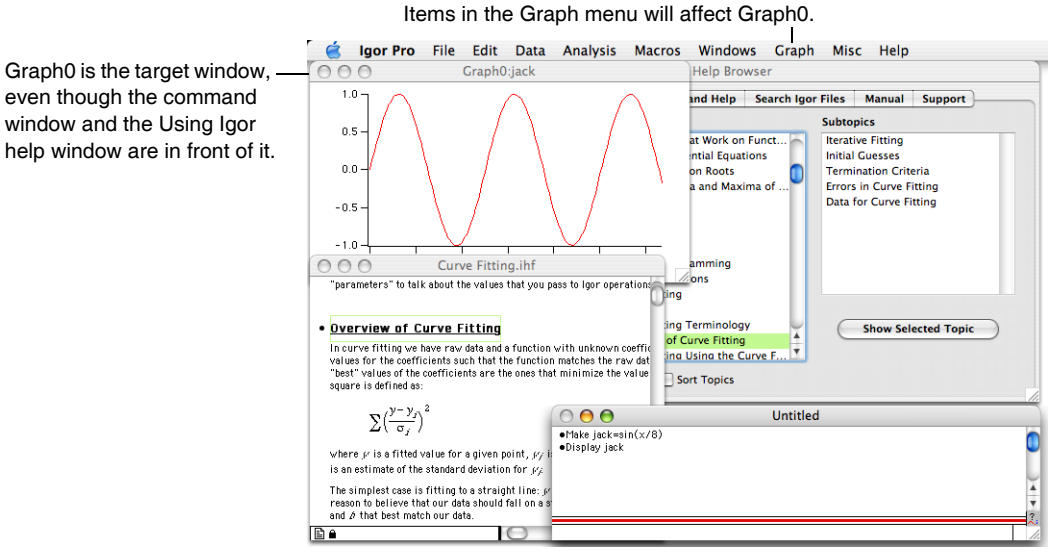
Igor extensions may add other windows to Igor. For example, the Data Browser window, which lets you see what data exists in the current experiment, is added by the Data Browser extension.

The Target Window

Igor commands and menus operate on the **target window**. The target window is the top graph, table, page layout, notebook, control panel or XOP target window. The term "target" comes from the fact that these windows can be the target of command line operations such as ModifyGraph, ModifyTable and so on. The command window, procedure windows, help windows and dialogs can not be targets of command line operations and thus are not target windows.

Prior to version 4, Igor attempted to draw a special icon to indicate which window was the target. However, this special target icon is no longer drawn because of operating system conflicts.

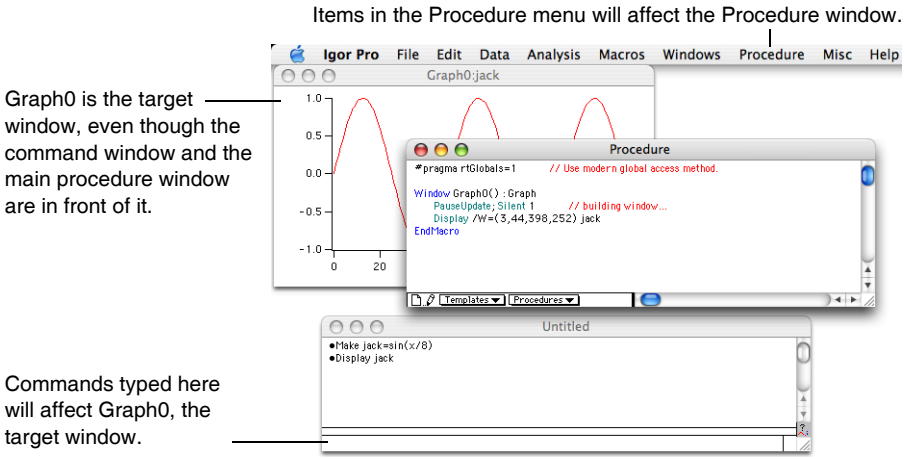
The menu bar changes depending on the top window and the target window. For instance, if a graph is the target window the menu bar contains the Graph menu:



The menu bar changes to contain menus that apply to the target window, but you may type any command into the command line, including commands that do not apply to the target window. Igor will apply the command to the top window of the correct type.

For instance, in the example above, you could type a `ModifyTable` command while Graph0 was the target window. Igor will apply the `ModifyTable` command to the top table, if there is one.

Sometimes the top window isn't a target window, but it causes the menu bar to change. To continue our example, if at this point you were to bring a procedure window to the top, the graph would still be the target window, but the Graph menu would be replaced with the Procedure menu. Menu items chosen from the Procedure menu apply to the top procedure window, but typed commands like `AppendToGraph myWave` or `DoWindow` will still affect the target window, Graph0.



Window Names and Titles

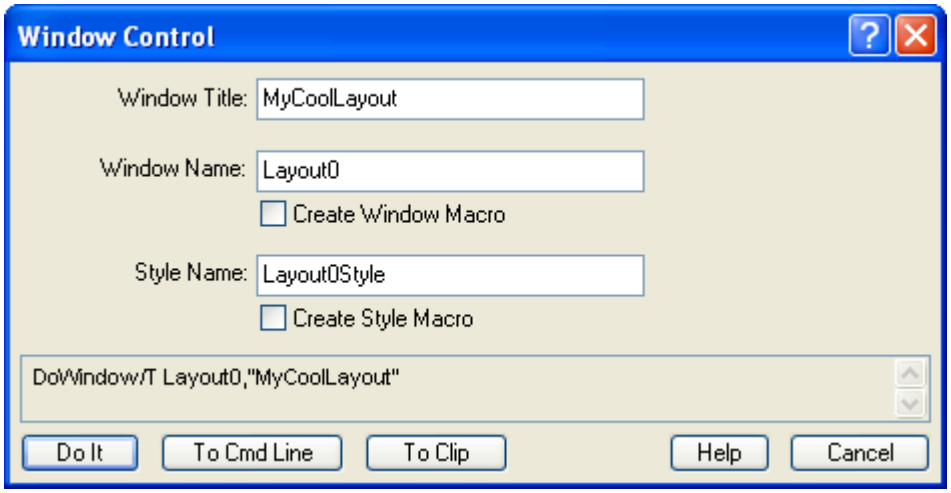
Each graph, table, page layout or panel has a **title** and a **name**.

The title is what you see at the top of the window frame and in the Windows menu. Its purpose is to help you visually identify the window, and is usually descriptive of its contents or purpose.

The window *name* is not the same as the *title*. The purpose of the name is to allow you to refer to the window from a command, such as the DoWindow or AppendToGraph operations.

When you first create one of these windows, Igor gives it a name like Graph0, Table0, Layout0 or Panel0, and a title based on the name and window contents. You can change the window's title and name to something more descriptive using the Window Control dialog (Windows→Control submenu). Among other things, it renames and retitles the target window.

Here we are about to change the title of the window named Layout0:



The Window Control dialog is also a good way to discover the name of the top window, since the window shows only the window title.

The command window, procedure windows, and help windows have *only* a title. The title is the name of the file in which they are stored. These windows do not have names because they can not be affected by command line operations.

In summary, you set the title of windows in various ways:

Window Type	How Titled	Has Window Name?
Graphs, tables, page layouts, notebooks and panels	Igor initially assigns a title based on the window name and content. You can retitle these windows with the Window Control dialog or the DoWindow/T command.	Yes
Command window	Initially "Untitled", it takes on the file name of saved experiment.	No
Built-in procedure window	Always titled "Procedure".	No
Auxiliary procedure windows	Titled when created, they take on the file name if saved as a file.	No
Igor Help windows	Same as the Igor help file.	No

Allowable Window Names

A window name is used for commands and therefore must follow the standard rules for naming Igor objects:

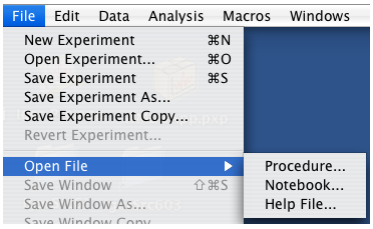
- The name must start with a letter.
- Additional characters can be alphanumeric or the underscore character.
- No other characters, including spaces, are allowed in standard Igor object names.
- No more than 31 characters are allowed.
- The name must not conflict with other object names (you see a message if it does).

For more information, see **Object Names** on page III-415.

The Open File Submenu

The File menu contains the Open File submenu for opening an existing file as a notebook, Igor help window, or procedure window.

When you choose an item from the submenu, the Open File dialog appears for you to select a file.



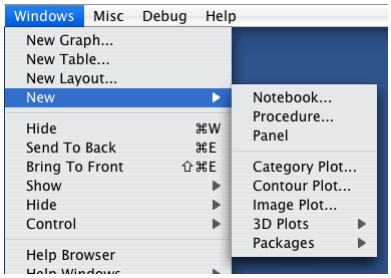
The Windows Menu

You can use the Windows menu for making new windows, and for showing, arranging and closing (either hiding or "killing") windows. You can also execute "window recreation macros" that recreate windows that have been killed and "style macros" that modify an existing window's appearance.

Making a New Window

You can use the various items in the Windows→New submenu to create new windows.

The menu items that end with "…" invoke dialogs which produce commands that Igor executes to create the windows. These dialogs are explained in the chapter about the corresponding window.



You can type these commands yourself directly in the command line. For example,

```
Display yData vs xData
```

creates a graph of the wave named yData on the Y axis, versus xData on the X axis.

You can create a new window by selecting the name of a window recreation macro from the Windows menu. See **Window Macros Submenus** on page II-62.

You can also create a window using the File→Open File submenu.

Activating Windows

To activate a window, choose an item from Windows menu or from the Help Windows, Procedure Windows, Graphs, Tables, Layouts, Other Windows, or Recent Windows submenus in the Windows menu.

The Recent Windows submenu shows windows recently activated. This information is saved when you save an experiment to disk and restored when you later reopen the experiment.

If you press Command (*Macintosh*) or Ctrl (*Windows*) while clicking the menu bar, a temporary Recent Windows menu will be accessible from the main menu bar. This shortcut is intended to save you the trouble of navigating through the Windows menu to the permanent Recent Windows submenu.

By default, just the window's title is displayed in the Windows menu. You can choose to display the title or the name for target windows using the Windows Menu Shows popup menu in the Misc Settings category of the Miscellaneous Settings dialog.

Showing and Hiding Windows

All built-in window types and some XOP window types can be hidden.

To hide a window, press Shift and choose Windows→Hide or use the keyboard shortcut Command-Shift-W (*Macintosh*) or Ctrl+Shift+W (*Windows*). You can also hide a window by pressing Shift and clicking the close button.

You can hide multiple windows at once using the Windows→Hide submenu. For example, to hide all graphs, choose Windows→Hide→All Graphs. If you press Shift while clicking the Windows menu, the sense of the menu items changes. For example, Hide→All Graphs changes to Hide→All Except Graphs.

The command window is not included in mass hides of any kind. If you want to hide it you must do so manually.

Similarly, you can show multiple windows at once using the Windows→Show submenu. For example, to show all graphs, choose Windows→Show→All Graphs. If you press Shift while clicking the Windows menu, the sense of the menu items changes. For example, Show→All Graphs changes to Show→All Except Graphs.

The Show All Except menu items do not show procedure windows and help files because there are so many of them that it would be counterproductive.

The Windows→Show→Recently Hidden Windows item shows windows recently hidden by a mass hide operation, such as Hide→All Graphs, or windows recently hidden manually (one-at-a-time using the close button or Command-Shift-W or Ctrl+Shift+W). In the case of manually hidden windows, "recently hidden" means within the last 30 seconds.

XOP windows do participate in Hide All XOP Windows and Show All XOP Windows only if XOP programmers specifically support these features.

Closing a Window

You can close a window by either choosing the Close item or by clicking in the window's close button. Depending on the top window's type, this will either kill or hide the window, possibly after a dialog asking for confirmation.

Killing Versus Hiding

"Killing" a window means the window is removed from the experiment. The memory used by the window is released and available for other purposes. The window's title is removed from the Windows menu. Killing a window that represents a file on disk does not delete the file. You can also kill a window with a `DoWindow/K` *winName* command.

"Hiding" a window simply means the window is made invisible, but is still part of the experiment and uses the same amount of memory. It can be made visible again by choosing its title from the Windows menu.

The command window and the built-in procedure window can be hidden but not killed. All other built-in windows can be hidden or killed.

When you create a window from a procedure, you can control what happens when the user clicks the close button using the `/K=<`*num*`>` flag in the command that creates the window.

You can hide a window programmatically using the `DoWindow/HIDE=1` operation. To show a hidden window without activating it, use `DoWindow/HIDE=0`. To show the window and activate it, use `DoWindow/F`.

Saving the Window Contents

Notebooks and procedure windows can be saved either in their own file, or in a packed experiment file with everything else. You can tell which is the case by choosing Notebook→Info or Procedure→Info. When you kill a notebook or a procedure window that contains unsaved information, a dialog will allow you to save it before killing the window.

Graph, table, panel and page layout windows are not saved as separate files, and are lost when you kill them unless you save a **window recreation macro** which you can execute to later recreate the window. Killing these windows and saving them as window recreation macros (stored in the built-in procedure

window) frees up memory and reduces window clutter without losing any information. You can think of window recreation macros as "freeze-dried windows".

This table shows how windows are hidden, killed, and saved:

Window Type	Hideable?	Killable?	Save Recreation Macro?	Stand-Alone File?
Graphs, Tables	Yes	Yes	Yes	Yes*
Layouts, Panels	Yes	Yes	Yes	No
Main procedure window	Yes	No	No	†
Help Browser	Yes	No	No	No
Auxiliary procedures, Notebooks, Help windows	Yes	Yes‡	No	Yes
Command window	No	No	No	No

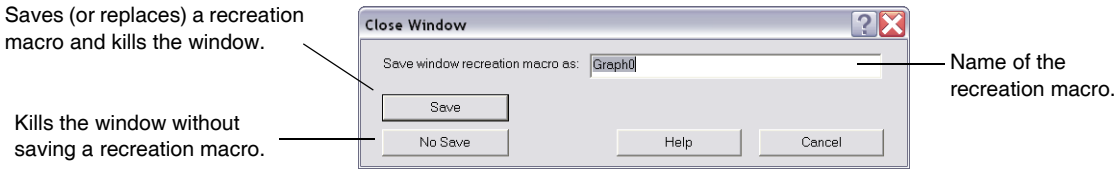
* The **SaveGraphCopy** operation (page V-539) and the **SaveTableCopy** operation (page V-546) can be used to save a graph or table, along with all associated waves, as a stand-alone experiment file.

† The main procedure window is stored as a separate file in the home folder if the experiment is saved in unpacked format, or within the experiment file if the experiment is saved in packed format.

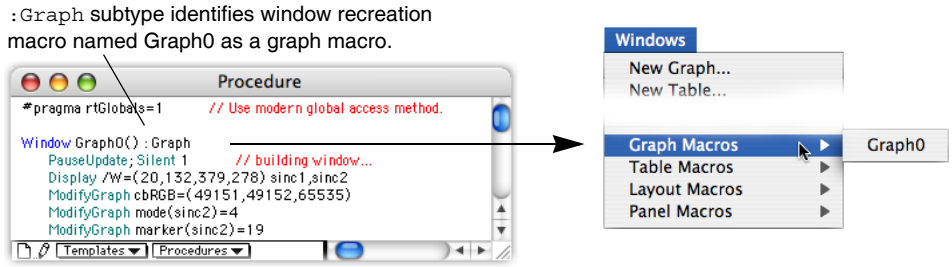
‡ The only way to kill a help window is to press Option (*Macintosh*) or Alt (*Windows*) when clicking the close button, or by pressing Command-Option-W (*Macintosh*) or Ctrl+Alt+W (*Windows*).

Close Window Dialogs

When you close a graph, table, layout or control panel, Igor presents a Close dialog.



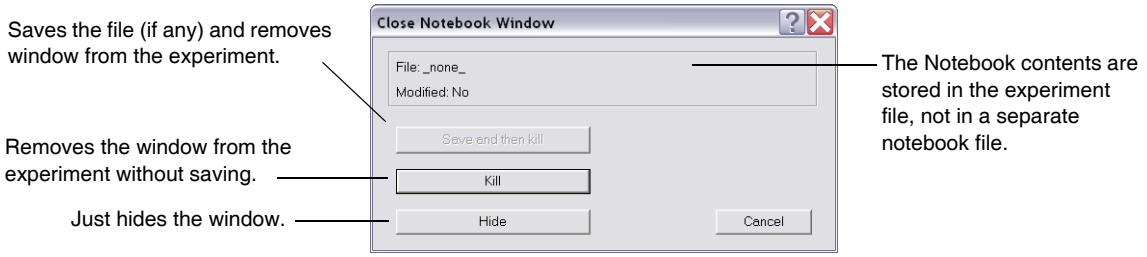
If you click the Save button Igor creates a window recreation macro in the main procedure window. It sets the macro's subtype to Graph, Table, Layout or Panel so the name of the macro appears in the appropriate Macros submenu of the Windows menu. You can recreate the window using this menu.



If you don't plan to use the window again, you should click the No Save button and no window recreation macro will be created.

If you have previously created a recreation macro for the window then the dialog will have a Replace button instead of a Save button. Clicking Replace replaces the old window recreation macro with a new one. If you know that you won't need to recreate the window, you can delete the macro (see **Saving a Window as a Recreation Macro** on page II-61).

When you close a notebook or procedure window (other than the built-in procedure window), Igor presents a "hide or kill dialog".



Macintosh: When the Close Window dialog is showing, you can press Option to make the Kill button the default. The Kill button will become highlighted while the "Save and then kill" button will become normal. You can then press Return or Enter to kill the window. Similarly, press Shift to make the Hide button the default button.

To hide a window, press Shift while clicking the close button. To kill a graph, table, layout, or control panel without the Close dialog, press Option (*Macintosh*) or Alt (*Windows*) while clicking the close button.

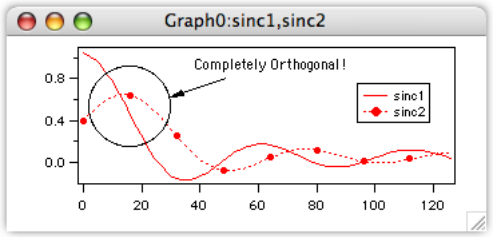
By specifying /K=<*num*> for the NewNotebook, Layout, Display, and NewPanel operations, you can modify this behavior.

Saving a Window as a Recreation Macro

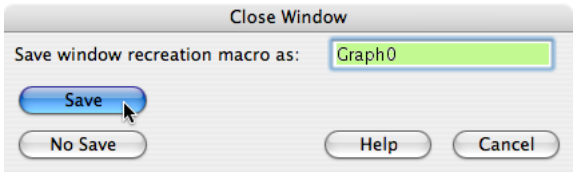
When you close a window that can be saved with a recreation macro, Igor offers to create one by displaying the Close Window dialog. Igor stores the window recreation macro in the main procedure window of the current experiment. The macro uses much less memory than the window, and reduces window clutter. You can invoke the window recreation macro later to recreate the window. You can also create or update a macro with the Window Control dialog.

The window macro contains all the necessary commands to reconstruct the window provided the underlying data is still present. For instance, a graph recreation macro contains commands to append waves to the graph, but does not contain any wave data. Similarly, a page layout recreation macro does not contain graphs or tables (nor the commands to create them). The macros refer to waves, graphs and tables in the current experiment by name.

Here is how you would use recreation macros to keep a graph handy, but out of your way:

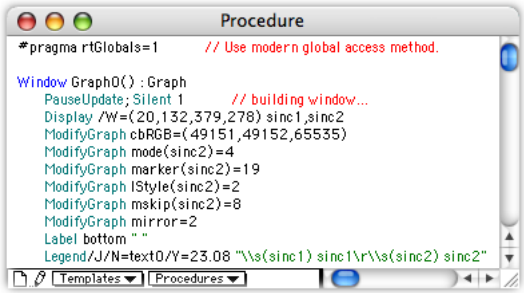


Choosing Close or clicking the close button...

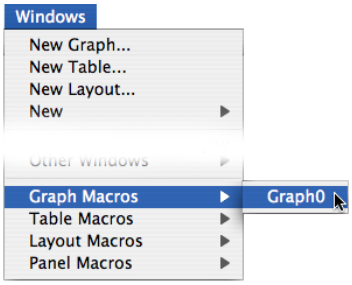


...summons the Close Window dialog.

Clicking Save...



...saves a window recreation macro for the graph in the main procedure window.



When the graph window is needed again choosing the macro from the Graph Macros menu runs the macro that recreates the graph.

The window macro is evaluated in the context of the root data folder. This detail is of consequence only to programmers. See **Data Folders and Commands** on page II-125 for more information.

You can create or replace a window macro without killing the window using **The Window Control Dialog** described on page II-64. The most common reason to replace a window macro is to keep the macro consistent with the window that it creates. This is useful if you are about to clone the window, having changed it since the recreation macro was made.

Notice that the proposed name of the window recreation macro is the same as the name of the saved window. You can save the window recreation macro under a different name, if you want, by entering the new name in the dialog. If you do this, Igor creates a new macro and leaves the original macro intact. You can run the new macro to create a new version of the window or you can run the old macro to recreate the old version. This way you can save several versions of a window, while displaying only the most recent one.

Window recreation macros stay in an experiment's procedure window indefinitely. If you know that you won't need to recreate a window for which a window recreation macro exists, you can delete the macro.

To locate a window macro quickly:

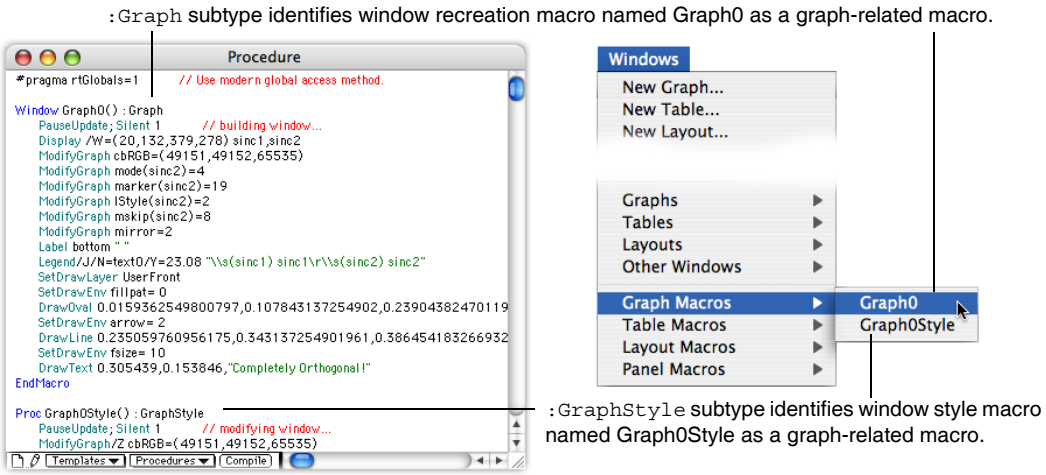
- Bring any procedure window to the top, press Option (*Macintosh*) or Alt (*Windows*) and choose the window macro name from the appropriate macro submenu in the Windows menu.

To delete the macro (if you're sure you won't want it again), simply select all the text from the Macro declaration line to the End line. Press Delete to remove the selected text.

See **Saving and Recreating Graphs** on page II-300 for details specific to graphs.

Window Macros Submenus

The Windows menu has hierarchical menus containing graph, table, page layout and panel recreation macros. These menus also include graph, table or page layout style macros.



Window recreation macros are created by the Close Window and Window Control dialogs, and by the DoWindow/R command. Style macros are created by the Window Control dialog and the DoWindow/R/S command.

Igor places macros into the appropriate macro submenu by examining the macro's subtype. The subtypes are Graph, Table, Layout, Panel, GraphStyle, TableStyle and LayoutStyle. See **Procedure Subtypes** on page IV-179 for details.

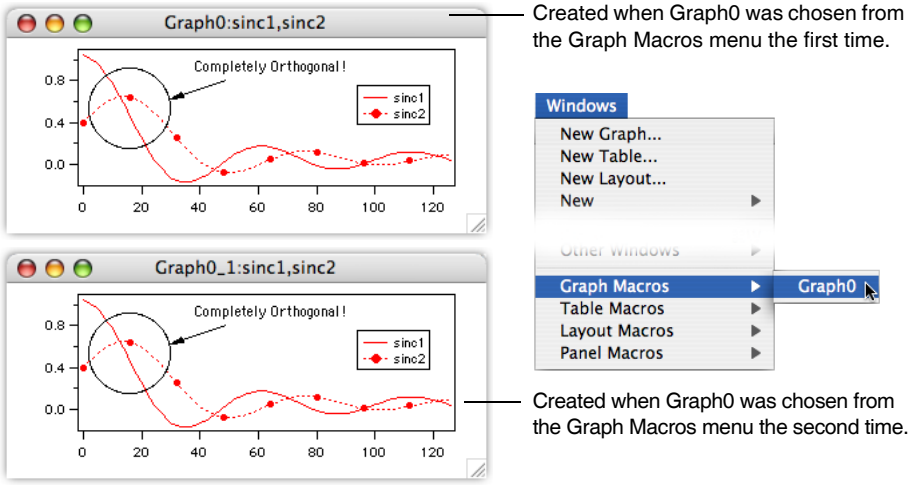
When you choose the name of a recreation macro from a macro submenu, the macro runs and recreates the window. Choosing a style macro runs the macro which changes the target window's appearance (its "style").

However, if a procedure window is the top window and you press Option (*Macintosh*) or Alt (*Windows*) and then choose the name of any macro, Igor displays that macro but does not execute it.

The Name of a Recreated Window

When you run a window recreation macro, Igor recreates the window with the same name as the macro that created it unless there is already a window by that name. In this case, Igor adds an underscore followed by a digit (e.g. _1) to the name of the newly created window to distinguish it from the preexisting window.

For example, this figure shows the result of running a graph recreation macro twice. There was no graph named Graph0 when we started:



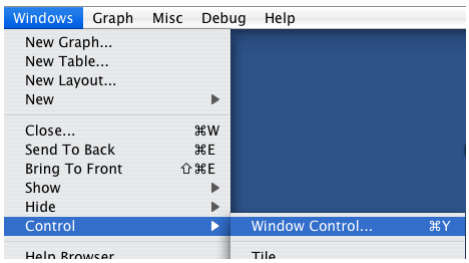
Changing a Window's Style From a Macro

When you run a style macro by invoking it from the Windows menu, from the command line or from another macro, Igor applies the commands in the macro to the top window. Usually these commands change the appearance of the window. For example, a graph style macro may change the color of graph traces or the axis tick marks.

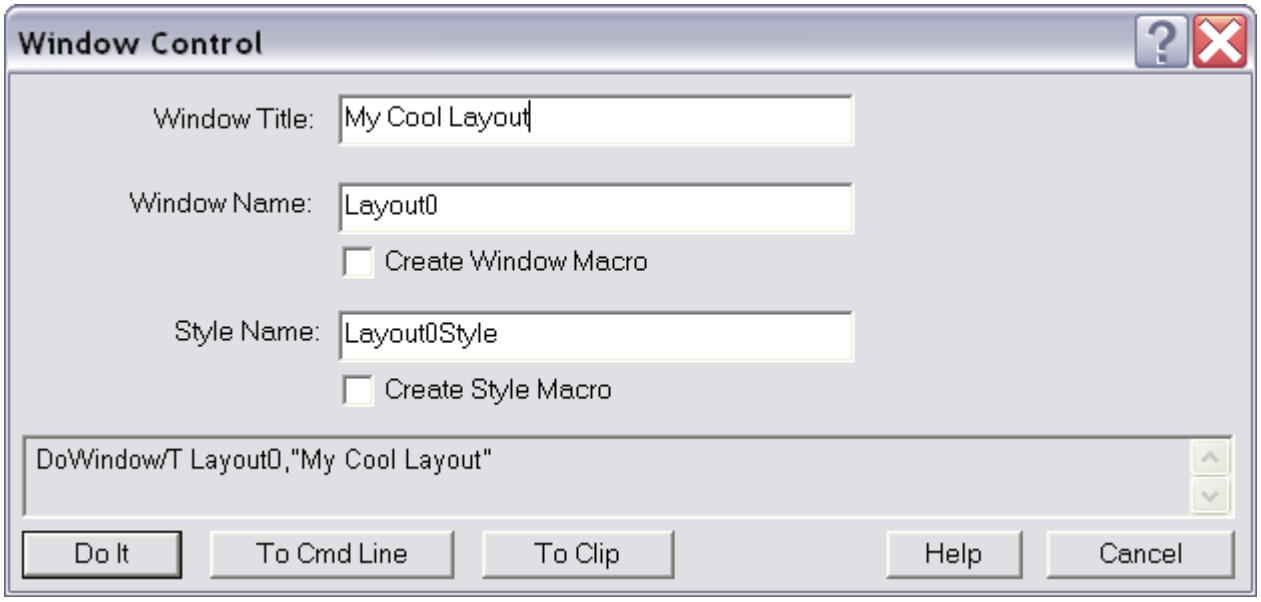
Style macros are used most effectively with graph windows. For more information, see **Saving and Recreating Graphs** on page II-300 and **Graph Style Macros** on page II-300.

The Window Control Dialog

Choosing Control→Window Control brings up a dialog you can use to change the top window's title and name, and create or update its recreation and style macros. You can access this dialog quickly by pressing Command-Y (*Macintosh*) or Ctrl+Y (*Windows*).



Here we are using the dialog to change the window named Layout0 to have a new title of "My Cool Layout".

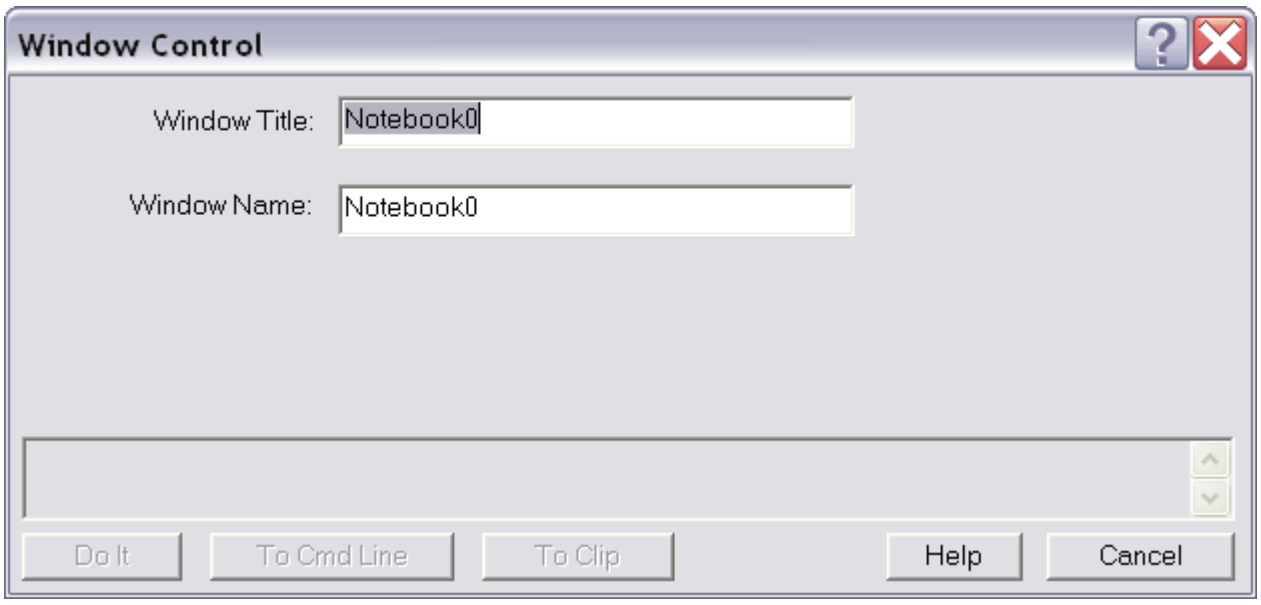


You can also change the window's name. The window name is used to address the window from command line operations such as DoWindow and also appears in the macro submenus of the Windows menu.

If the window name matches the name of an existing a window or style macro, the checkboxes will change to Update Window Macro and Update Style Macro.

The dialog may look a little different for some window types. For instance, panels don't have style macros, so for panel windows the Create Style Macro item will be missing.

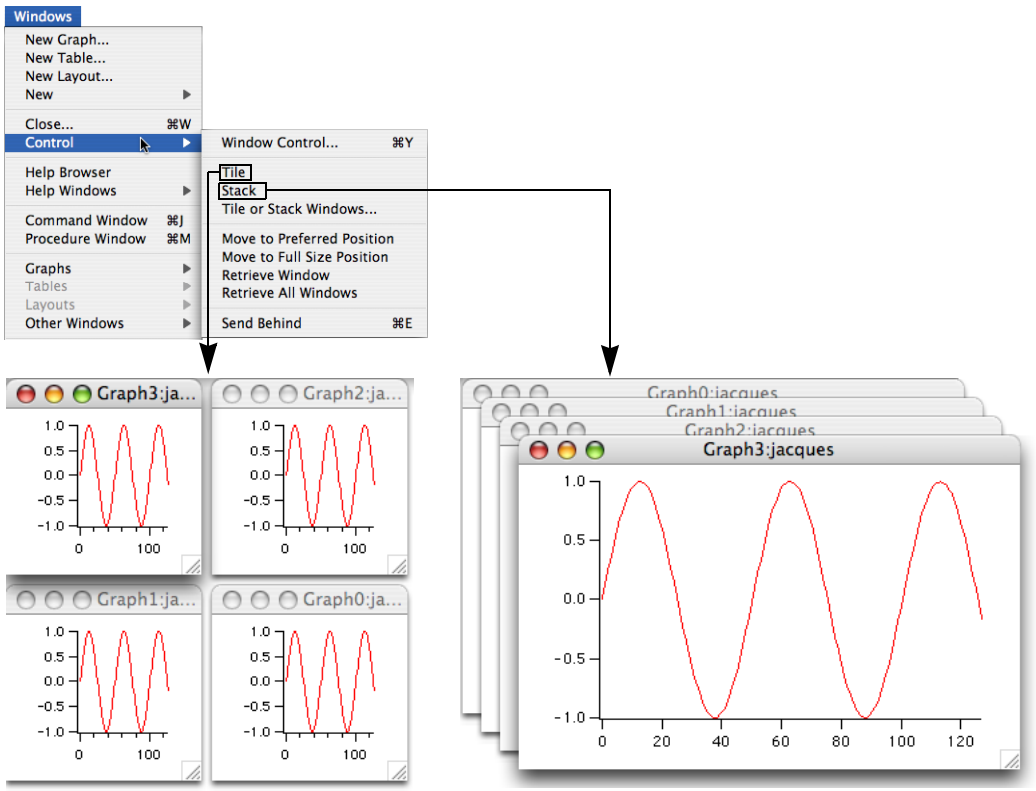
Similarly, notebooks can not be saved as macros, so both the Create Window Macro and Create Style Macro items will be missing:



For more about names and titles, see **Window Names and Titles** on page II-56. Also see **Saving a Window as a Recreation Macro** on page II-61 for a discussion of window recreation macros, and see **Graph Style Macros** on page II-300 for details on style macros.

Arranging Windows

You can tile or stack windows by choosing the appropriate items from the Control submenu in the Windows menu.

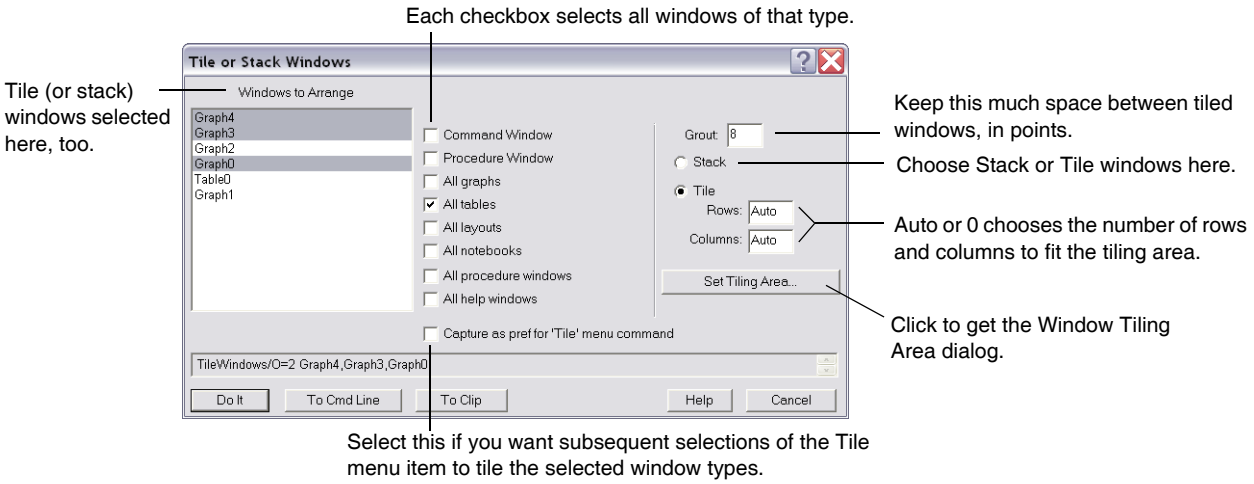


You can customize the behavior of the Tile and Stack items using the Tile or Stack Windows dialog.

You can also move windows around using the MoveWindow, StackWindows, and TileWindows commands.

The Tile or Stack Windows Dialog

The Tile or Stack Windows dialog is useful for tiling a few windows or even for setting the size and position of a single window.



Select individual windows from the Windows to Arrange list, and entire classes of windows with the checkboxes.

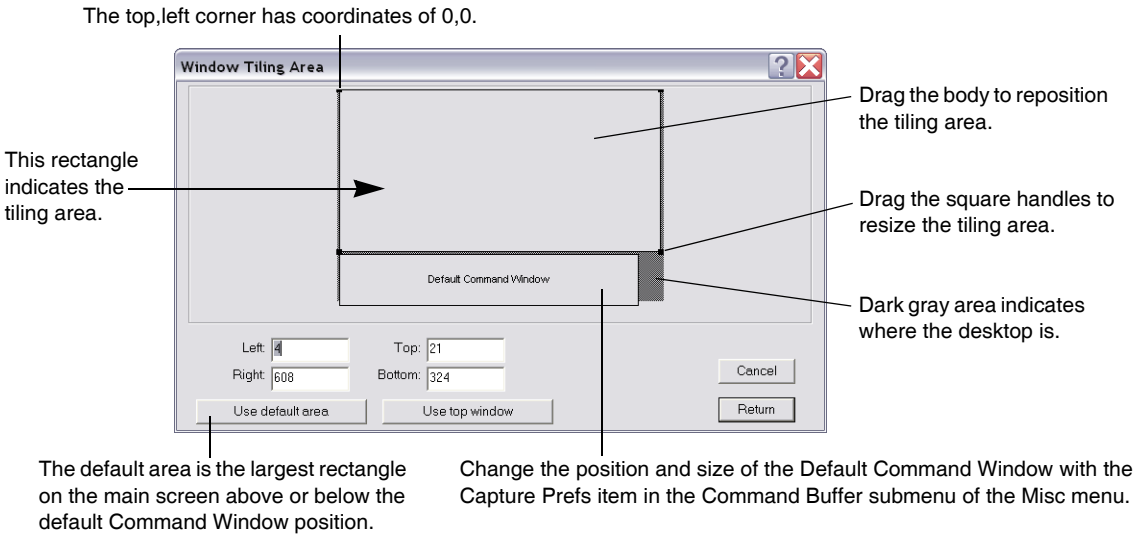
If you want subsequent selections of the Tile (or Stack) menu item to stack the same types of windows with the same rows, columns, grout, tiling area, etc., you should select the "Capture as pref" checkbox. Windows selected in the Windows to Arrange menu aren't remembered by the preferences: only the window type checkboxes. There are separate settings and preferences for Stack and for Tile.

Notice that although the TileWindows and StackWindows operations can tile and stack panels, panels don't show up here because they don't resize very well.

The Window Tiling Area subdialog specifies the area where tiling *and stacking* take place.

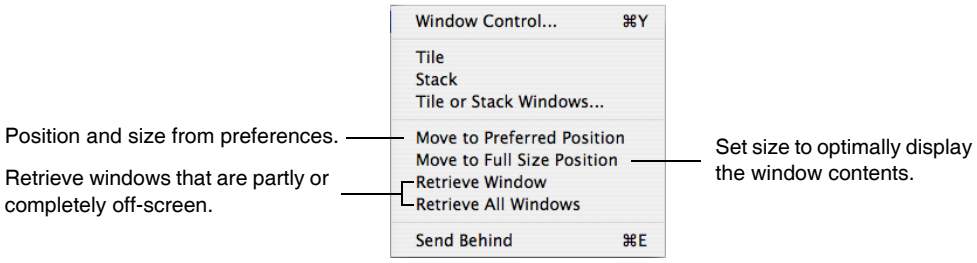
You can specify the tiling area in one of four ways:

- By entering screen positions in units of points.
- By dragging the pictorial representation of the tiling area.
- Use the default tiling area by clicking the "Use default area" button.
- By positioning any nondialog window *before* you enter the dialog, and clicking the "Use top window" button.



Window Position and Size Management

There are four items in the Control submenu of the Windows menu that help you manage the position and size of windows.



Move to Preferred Position

Moves the active window to the position and size determined by preferences. For each type of window, you can set the preferred position and size using the Capture Prefs dialog (e.g., Capture Graph Prefs for graphs).

Shortcut for *Windows*: Press Alt and click the maximize button.

Move to Full Size Position

Moves and sizes the active window to display as much of the content as practical. On Macintosh, this is the same as clicking the zoom button. On Windows, the size is limited to the size of the frame window.

Shortcut: Press Shift-Option (*Macintosh*) or Shift+Alt (*Windows*) and click the maximize button.

Retrieve Window

Moves the active window and sizes it if necessary so that all of the window is visible.

Retrieve All Windows

Moves all windows and sizes them if necessary so that all of each window is within the screen on Macintosh or within the frame on Windows. This is often useful when you open an experiment that was created on a system with a larger screen or Windows frame than yours.

Send to Back — Bring to Front

The Send to Back item in the Windows menu sends the top window to the bottom of the desktop, behind all other windows. This function can also be accessed by pressing Control-Command-E (*Macintosh*) or Ctrl+E (*Windows*). After sending a window behind, you can bring it to the front by choosing Bring to Front or by pressing Shift-Control-Command-E (*Macintosh*) or Ctrl+Shift+E (*Windows*). You can also press Control-Command-E or Ctrl+E repeatedly to cycle through all windows.

Send to Back is handy to use in conjunction with Command-J (*Macintosh*) or Ctrl+J (*Windows*) which brings the command window to the top of the desktop. You can press Command-J or Ctrl+J to bring the command window to the top, enter a command, and then press Control-Command-E or Ctrl+E to get the command window out of the way again.

Igor has a nifty feature that comes in handy if you have many windows tiled such that some are completely behind others. If you press Option (*Macintosh*) or Alt (*Windows*) and choose Send to Back or press Command-Option-E (*Macintosh*) or Ctrl+Alt+E (*Windows*), any window that is completely visible is sent to the back.

For example, imagine that you have eight graphs. You can tile them into two planes of four graphs per plane using the Tile or Stack Windows dialog, or with the command: `TileWindows/O=1/A=(2,2)`. Now, pressing Command-Option-E or Ctrl+Alt+E sends the top four graphs behind, revealing the bottom four graphs.

You can also send a window to the back with the `DoWindow/B` command and bring it to the front with the `DoWindow/F` command.

Text Windows

Igor Pro displays text in procedure, notebook, and Igor help windows as well as in the command and history areas of the command window. This section discusses behavior common to all of these windows.

Executing Commands

You can execute commands selected in a notebook, procedure or help window by pressing Control-Enter or Control-Return. You can also execute selected commands by Control-clicking (*Macintosh*) or right-clicking (*Windows*) and choosing Execute Selection.

For more on this, see **Notebooks as Worksheets** on page III-5.

Text Window Navigation

The term "keyboard navigation" refers to selection and scrolling actions that Igor performs in response to the arrow keys and to the Home, End, Page Up, and Page Down keys. Macintosh and Windows have different conventions for these actions in windows containing text. You can use either Macintosh or Windows conventions on either platform.

By default, Igor uses Macintosh conventions on Macintosh and Windows conventions on Windows. You can change this using the Keyboard Navigation menu in the Misc Settings section of the Miscellaneous Settings dialog. If you use Macintosh conventions on Windows, use Ctrl in place of the Macintosh Command key. If you use Windows conventions on Macintosh, use Command in place of the Windows Ctrl key.

Macintosh Text Window Navigation

Key	No Modifier	Option	Command
Left Arrow	Move selection left one character	Move selection left one word	Move selection to start of line
Right Arrow	Move selection right one character	Move selection right one word	Move selection to end of line
Up Arrow	Move selection up one line	Move selection up one paragraph	Move selection up one screen
Down Arrow	Move selection down one line	Move selection down one paragraph	Move selection down one screen
Home	Scroll to start of document	Scroll to start of document	Not used
End	Scroll to end of document	Scroll to end of document	Not used
Page Up	Scroll up one screen	Scroll up one screen	Not used
Page Down	Scroll down one screen	Scroll down one screen	Not used

Windows Text Window Navigation

Key	No Modifier	Ctrl
Left Arrow	Move selection left one character	Move selection left one word
Right Arrow	Move selection right one character	Move selection right one word
Up Arrow	Move selection up one line	Move selection up one paragraph
Down Arrow	Move selection down one line	Move selection down one paragraph
Home	Move selection to start of line	Move selection to start of document
End	Move selection to end of line	Move selection to end of document

Windows Text Window Navigation

Key	No Modifier	Ctrl
Page Up	Scroll up 1 screen	Scroll up 1 screen
Page Down	Scroll down 1 screen	Scroll down 1 screen

Finding Text in the Active Window

You can access the Find Text dialog via the Edit menu or by pressing Command-F (*Macintosh*) or Ctrl+F (*Windows*). The Find Text dialog is available for help, procedure, and notebook windows, for the command line and the history area, and for some XOP windows.

You can search for the next occurrence of a string (Edit→Find Selection) without using the dialog by selecting the string and pressing Command-Control-H (*Macintosh*) or Ctrl+H (*Windows*). (Unreformed old-timers can change the Macintosh key combination to its original setting of Command-H using the Miscellaneous Settings dialog.)

After doing a find, you can search for the same text again by pressing Command-G (*Macintosh*) or Ctrl+G (*Windows*) (Find Same in the Edit menu). You can search for the same text but in the reverse direction by pressing Command-Shift-G (*Macintosh*) or Ctrl+Shift+G (*Windows*).

You can abort a find by clicking the Stop button in the Find Text dialog or by pressing Command-period (*Macintosh*) or Ctrl+Break (*Windows*).

Find and Replace

To find and replace:

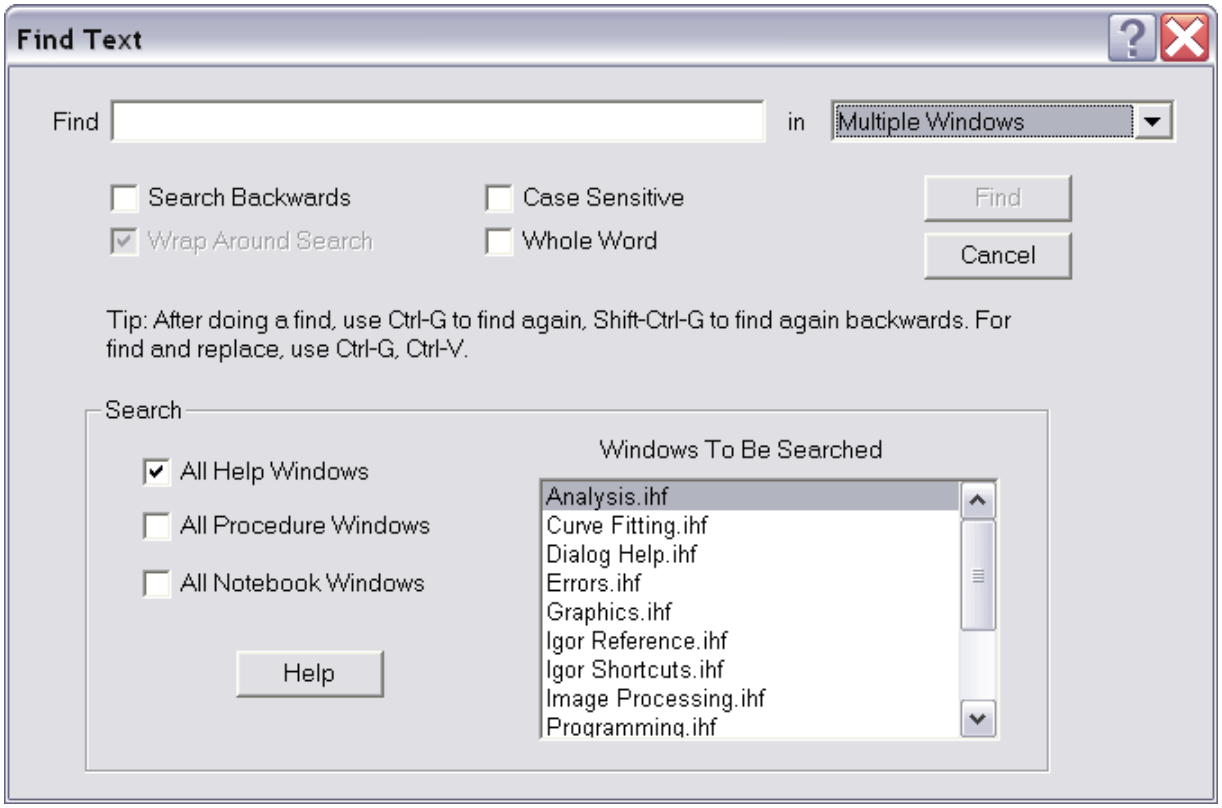
1. Move the selection to the top of the active window.
2. Use Edit→Find to find the first instance of the target string.
3. Manually change the first instance, then copy the new text to the Clipboard.
4. Press Command-G (*Macintosh*) or Ctrl-G (*Windows*) to find the next occurrence.
5. Press Command-V (*Macintosh*) or Ctrl-V (*Windows*) to paste.
6. Repeat steps 4 and 5 until done.

Finding Text in Multiple Windows

You can perform a Find on multiple help, procedure and notebook windows at one time using the following procedure.

1. Open a help, procedure or notebook window.
2. Press Command-F (*Macintosh*) or Ctrl+F (*Windows*) or choose Find from the Edit menu.
3. Choose Multiple Windows from the pop-up menu in the Find dialog.
4. Enter the text to find and click Find.

When you follow this procedure, the Find Text dialog looks something like this:



The windows that will be searched appear in the list, in the order in which they will be searched. You add windows to the list by selecting one of the checkboxes to the left of the list. The search is done from the top to the bottom of the list, or from the bottom to top if you have selected Search Backwards.

The selection in the list indicates the file to be searched next. You can change the selection by clicking the list or using the arrow keys. However, this is usually not necessary.

You can abort a find by clicking the Stop button in the Find Text dialog or by pressing Command-period (*Macintosh*) or Ctrl+Break (*Windows*).

When you turn multiple window find on, it stays on until you turn it off, by choosing Active Window Only from the pop-up menu. The setting of this pop-up menu affects not only Find but also Find Same and Find Selection. These last two operations do not display the Find Text dialog. To avoid confusion, use Find if you are unsure whether multiple window find is off or on.

The multiple window find may sometimes cause surprising behavior. For example, you may expect that the search will start with the active window. However, when doing a multiple window find, this is not the case. The search starts with the item highlighted in the list. Also, the search does not start from the selection in that window but rather from the top of the window, or from the bottom if you have selected Search Backwards.

When you click Find, Igor searches the windows in the list. When it finds the first occurrence of the target string, it closes the dialog and displays the found text. At this point, you can do a Command-G (*Macintosh*) or Ctrl+G (*Windows*) or choose Find Same from the Edit menu, to find the next occurrence of the target string. When you do a Find Same, the search starts from the selection in the window in which the target string was last found. This will normally be the active window but not if you activated another window after the last find.

While you are doing a multiple-window find you can, as in a single window find, press Command-Shift-G (*Macintosh*) or Ctrl+Shift+G (*Windows*) to search in the opposite direction (e.g., backwards if you were searching forwards). This provides a handy way to move quickly back and forth between two occurrences of the search string.

If you do a find after finding the last occurrence of the search string, Igor will beep, indicating that there are no more occurrences. At this point, if you want to go back to the preceding occurrence, you may need to press Command-Shift-G or Ctrl+Shift+G twice. The reason for this is that, after hitting the end of the list of windows to be searched, when you search backwards, Igor starts from the end of the list and finds the last occurrence again.

If you open or kill a window of a type that is to be searched, Igor rebuilds the list of files to be searched and resets the multiple-window find to the top of the list (or bottom if you are searching backwards). Igor also resets the multiple-window find when you change any of the settings in the Find Text dialog.

If you choose Find Selection from the Edit menu or press Command-H or Ctrl+H, Igor resets the find mode to active window only, enters the selected text as the search string, and then does the find. If what you really wanted was to do a multiple-window find, then after doing the Find Selection, press Command-F or Ctrl+F to active the Find Text dialog, choose Multiple Windows from the pop-up menu, and then click the Find button. This will start the multiple window find from the start of the list.

You can use the Igor Help Browser to search in multiple files, including files that are not open in your current experiment. See **Igor Help Browser** on page II-6 for further details.

Text Magnification

You can magnify the text in any window to make it bigger or smaller to suit your taste.

In help windows, procedure windows, plain text notebooks, and formatted text notebooks, you can use the magnifying glass icon in the bottom-left corner of the window. You can also use the Magnification submenu in the contextual menu for the window. To display the contextual menu, Control-click (*Macintosh*) or right-click (*Windows*) in the body of the window.

You can also set the magnification for the command line, history area, and text areas in dialogs such as Browse Waves and Add Annotation. These areas do not display the magnifying glass icon so you must use the contextual menu.

You may notice some anomalies when you use text magnification. For example, in a formatted text notebook, text may wrap at a different point in the paragraph and may change in relation to tab stops. This happens because fonts are not available in fractional sizes and because the actual width of text does not scale linearly with font size.

The Fit Width and Fit Page modes are inaccurate because of the availability of integer font sizes only. However they still may be useful. These modes are based on the available space for printing the document, which depends on the paper size chosen in the Page Setup dialog and the page margins as set in the Document Settings dialog. Because the actual content of the document may be much narrower or much wider than the available space for printing, these modes may sometimes give unexpected results.

You can set the default magnification for each type of text area by choosing a magnification from the Magnification popup menu and then choosing Set As Default from the same popup menu. Any text areas whose magnification is set to Default will use the newly specified default magnification. For example, if you want text in all help files to appear larger, open any help file, choose a larger magnification, 125% for example, and then choose Set As Default For Help Files. All help files whose current magnification is set to Default will be updated to use the new default.

The default magnification for the command line and history area controls the magnification that will be used the next time you launch Igor Pro.

The magnification setting is saved in formatted notebooks and help files only. If you change the magnification setting for one of these files and then save and close the file, the magnification setting will be restored when you reopen the file. For all other types of text areas, including procedure windows and plain text notebooks, the magnification setting is not stored in the file. If you close and reopen such a file, it will reopen using the default magnification for that type of text area.

Window User Data

You can store arbitrary data with a window using the userdata keyword with the **SetWindow** operation (page V-569). This is a topic of interest to advanced Igor programmers.

Each window has a primary, unnamed user data that is used by default. You can also store an unlimited number of different user data strings by specifying a name for each different user data string. The name can be anything you desire as long as it is a legal Igor name.

You can retrieve information from the default user data with the **GetWindow** operation (page V-225). To retrieve any named user data, you must use the **GetUserData** operation (page V-224).

Following is a simple example of user data using the top window:

```
SetWindow kwTopWin,userdata= "window data"
Print GetUserData("","","")
```

Although there is no size limit to how much user data you can store, it does have to be generated as part of the recreation macro for the window when experiments are saved. Consequently, huge user data can slow down experiment saving and loading

User data is intended to replace or reduce the usage of global variables. Named user data is intended for authors of packages that add features to existing windows. The primary author of a window should use the default, unnamed user data.

Chapters About Specific Windows

Detailed information about each type of window can be found in these chapters:

Window Type	Chapter
Command window	Chapter II-2, **The Command Window** Chapter IV-1, **Working with Commands**
Procedure windows	Chapter III-13, **Procedure Windows**
Help Browser Help windows	Chapter II-1, **Getting Help**
Graphs	Chapter II-12, **Graphs**
Tables	Chapter II-11, **Tables**
Layouts	Chapter II-16, **Page Layouts**
Notebooks	Chapter III-1, **Notebooks**
Control panels	Chapter III-14, **Controls and Control Panels**

Window Shortcuts

Action	Shortcut (*Macintosh*)	Shortcut (*Windows*)
To close a window:	Click the close button or press Command-W.	Click the close button or press Ctrl+W.
To kill a window with no dialog:	Press Option and click the close button or press Command-Option-W.	Press Alt and click the close button or press Ctrl+Alt+W.
To hide a window:	Press Shift and click the close button or press Command-Shift-W.	Press Shift and click the close button or press Ctrl+Shift+W.
To invoke the Window Control dialog:	Press Command-Y.	Press Ctrl+Y.
To send the top window behind all others:	Press Control-Command-E.	Press Ctrl+E.
To bring the bottom window on top of all others:	Press Shift-Control-Command-E.	Press Ctrl+Shift+E.
To send all windows that are completely visible behind all others:	Press Command-Option-E.	Press Ctrl+Alt+E.
To activate a recently activated window:	Press Command, click the main menu bar, and select from the Recent menu.	Press Ctrl, click the main menu bar, and select from the Recent menu.
To move a window to its preferred size and position:	Click the zoom button.	Press Alt and click the maximize button.
To move a window to its full-size position:	Press Shift-Option and click the zoom button.	Press Shift +Alt and click the maximize button.
To activate the command window:	Press Command-J.	Press Ctrl+J.
To clear the command buffer:	Press Command-K.	Press Ctrl+K.
To open the built-in procedure window:	Press Command-M.	Press Ctrl+M.
To cycle through all procedure windows:	Press Command-Option-M.	Press Ctrl+Alt+M.
To open the Help Browser:	Press Help or Command-?.	Press F1.
To find a phrase in a text window:	Press Command-F.	Press Ctrl+F.
To find the same phrase again:	Press Command-G. Press Command-Shift-G to search backward.	Press Ctrl+G. Press Ctrl+Shift+G to search backward.
To find the selected phrase:	Press Command-Control-H. Press Command-Shift-Control-H to search backward. This key combination can be changed through the Miscellaneous Settings dialog.	Press Ctrl+H. Press Ctrl+Shift+H to search backward.

Chapter
II-5

Waves

Overview

We use the term "wave" to describe the Igor object that contains an array of numbers. Wave is short for "waveform". The main purpose of Igor is to store, analyze, transform, and display waves.

Chapter I-1, **Introduction to Igor Pro**, presents some fundamental ideas about waves. Chapter I-2, **Guided Tour of Igor Pro**, is designed to make you comfortable with these ideas. In this chapter, we assume that you have been introduced to them.

This chapter focuses on one-dimensional numeric waves. Waves can have up to four dimensions and can store text data. Multidimensional waves are covered in Chapter II-6, **Multidimensional Waves**. Text waves are discussed in this chapter.

The primary tools for dealing with waves are Igor's built-in operations and functions and its waveform assignment capability. The built-in operations and functions are described in detail in Chapter V-1, **Igor Reference**.

This chapter covers:

- waves in general
- operations for making, killing and managing waves
- setting and examining wave properties
- waveform assignment

and other topics.

Waveform Model of Data

A wave consists of a number of components and properties. The most important are:

- the wave name
- the X scaling property
- X units
- an array of data values
- data units

The waveform model of data is based on the premise that there is a straight-line mapping from a point number index to an X value or, stated another way, that the data is uniformly spaced in the X dimension. This is the case for data acquired from many types of scientific and engineering instruments and for mathematically synthesized data. If your data is *not* uniformly spaced, you can use two waves to form an XY pair. See **XY Model of Data** on page II-78.

A wave is similar to an array in a standard programming language like BASIC, FORTRAN, Pascal, or C.

An array

	Index	Value
array0	0	3.74
	1	4.59
	2	4.78
	3	5.89
	4	5.66

A wave

	Point Number	X value (s)	data value (V)
wave0	0	0	3.74
	1	.001	4.59
	2	.002	4.78
	3	.003	5.89
	4	.004	5.66

An array in a standard language has a **name** (array0 in this case) and a number of **values**. We can reference a particular value using an **index**.

A wave also has a **name** (wave0 in this case) and **data values**. It differs from the array in that it has *two* indices. The first is called the **point number** and is identical to an array index or row number. The second is called the **X value** and is in the natural X units of the data (e.g., seconds, meters). Like point numbers, X values are not stored in memory but rather are *computed*.

The X value is related to the point number by the wave's X scaling, which is a property of the wave that you can set. The X scaling of a wave specifies how to compute an X value for a given point number using the formula:

```
x[p] = x0 + dx·p
```

where x[p] is the X value for point p. The two numbers x0 and dx constitute the wave's X scaling property. x0 is the starting X value. dx is the difference in X value from one point to the next. X values are uniformly spaced along the data's X dimension.

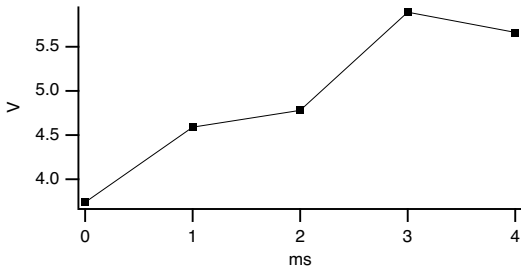
The **SetScale** operation (see page V-564) sets a wave's X scaling. You can use the Change Wave Scaling dialog to generate SetScale commands.

Why does Igor use this model for representing data? We chose this model because it provides all of the information that needed to properly display, analyze and transform waveform data.

By setting your data's X scaling, and X and data units in addition to its data values, you can make a proper graph in one step. You can execute the command

```
Display wave0
```

to produce a graph like this:



If your data is uniformly spaced on the X axis, it is *critical* that you understand and use X scaling.

The X scaling information is essential for operations such as integration, differentiation and Fourier transforms and for functions such as the **area** function (see page V-28). It also simplifies waveform assignment by allowing you to reference a single value or range of values using natural units.

In Igor Pro 3.0, we extended Igor from one dimension to multiple (up to 4) dimensions. To the X dimension, we added the Y, Z and T dimensions. X scaling extends to dimension scaling. For each dimension, there is a starting index value (x0, y0, z0, t0) and a delta index value (dx, dy, dz, dt). See Chapter II-6, **Multidimensional Waves**, for more about multidimensional waves.

XY Model of Data

If your data is not uniformly spaced along its X dimension then it can not be represented with a single wave. You need to use two waves as an XY pair.

In an XY pair, the data values of one wave provide X values and the data values of the other wave provide Y values. The X scaling of both waves is irrelevant so we leave it in its default state in which the x0 and dx components are 0 and 1. This gives us

```
x[p] = 0 + 1·p
```

This says that a given point's X value is the same as its point number. We call this "point scaling". Here is some sample data that has point scaling.

	X wave			Y wave		
	Point Number	X value ()	data value (V)	Point Number	X value ()	data value (V)
xWave / yWave	0	0	0.0	0	0	3.74
	1	1	.0013	1	1	4.59
	2	2	.0021	2	2	4.78
	3	3	.0029	3	3	5.89
	4	4	.0042	4	4	5.66

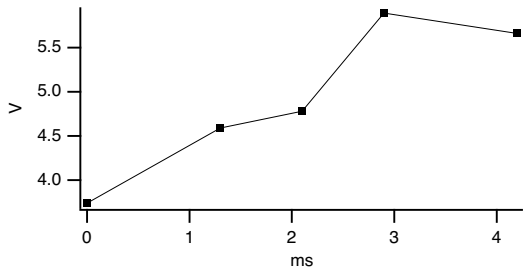
The X values serve no purpose in the XY model. Therefore, we change our thinking and look at an XY pair this way.

	X wave		Y wave	
	Point Number	Value (s)	Point Number	Value (V)
xWave / yWave	0	0.0	0	3.74
	1	.0013	1	4.59
	2	.0021	2	4.78
	3	.0029	3	5.89
	4	.0042	4	5.66

We can execute

```
Display yWave vs xWave
```

and it produces a graph like this.



Some operations, such as Fast Fourier Transforms and convolution, require equally spaced data. In these cases, it may be desirable for you to create a uniformly spaced version of your data by interpolation. See **Converting XY Data to a Waveform** on page III-116.

Some people who have uniformly spaced data still use the XY model because it is what they are accustomed to. **This is a mistake**. If your data is uniformly spaced, it will be well worth your while to learn and use the waveform model. It greatly simplifies graphing and analysis and makes it easier to write Igor procedures.

Making Waves

You can make waves by:

- Loading data from a file
- Typing or pasting in a table
- Using the **Make** operation (via a dialog or directly from the command line)
- Using the **Duplicate** operation (via a dialog or directly from the command line)

Most people start by loading data from a file. Igor can load data from text files. In this case, Igor makes a wave for each column of text in the file. Using external operations, Igor can also load data from binary files or application-specific files created by other programs. For information on loading data from files, see Chapter II-9, **Importing and Exporting Data**.

You can enter data manually into a table. This is recommended only if you have a small amount of data. See **Using a Table to Create New Waves** on page II-195.

To synthesize data with a mathematical expression, you would start by making a wave using the **Make** operation (see page V-366). This operation is also often used inside an Igor procedure to make waves for temporary use.

The **Duplicate** operation (see page V-133) is an important and handy tool. Many built-in operations transform data in place. Thus, if you want to keep your original data as well as the transformed copy of it, use Duplicate to make a clone of the original.

Wave Names

All waves in Igor have names so that you can reference them from commands. You also use a wave's name to select it from a list or pop-up menu in Igor dialogs or to reference it in a waveform assignment statement.

You need to choose wave names when you use the **Make**, **Duplicate** or **Rename** operations via dialogs, directly from the command line, and when you use the Data Browser.

All names in Igor are case insensitive; wave0 and WAVE0 refer to the same wave.

The rules for the kind of characters that you can use to make a wave name fall into two categories: standard and liberal. Both standard and liberal names are limited to 31 characters in length.

Standard names must start with an alphabetic character (A - Z or a-z) and may contain alphabetic and numeric characters and the underscore character only. Other characters, including spaces, dashes and periods, are not allowed. We put this restriction on standard names so that Igor can identify them unambiguously in commands, including waveform assignment statements.

Liberal names, on the other hand, can contain any character except control characters (such as tab or carriage return) and the following four characters:

```
"  '  :  ;
```

Standard names can be used without quotation in commands and expressions but liberal names must be quoted. For example:

```
Make wave0; wave0 = p          // wave0 is a standard name
Make 'wave 0'; 'wave 0' = p    // 'wave 0' is a liberal name
```

Igor can not unambiguously identify liberal names in commands unless they are quoted. For example, in

```
wave0 = miles/hour
```

miles/hour could be a single wave or it could be the quotient of two waves.

To make them unambiguous, you must enclose liberal names in single straight quotes whenever they are used in commands or waveform arithmetic expressions. For example:

```
wave0 = 'miles/hour'
Display 'run 98', 'run 99'
```

Warning: Some Igor procedures and extensions written prior to Igor Pro 3.0 will not work on objects with liberal names. Providing for liberal names requires extra effort and testing on the part of Igor programmers (See **Programming with Liberal Names** on page IV-147). We recommend that you avoid using liberal names until you understand the potential problems and how to solve them.

See **Object Names** on page III-415 for a discussion of object names in general.

Number of Dimensions

Waves can consist of one to four dimensions. You determine this when you make a wave. You can change it using the **Redimension** operation (see page V-513). See Chapter II-6, **Multidimensional Waves** for details.

Number Type and Precision

Each numeric wave has a numeric type and a numeric precision. You can set a wave's type and precision when you make it. You can change it using the **Redimension** operation (see page V-513) or the Redimension dialog. The numeric type can be real or complex.

Not all operations and functions work on complex waves. Also, when you use a complex wave in a real number expression you will get an error message. See **Example: Complex Wave Calculations** on page II-98 for more information.

This table shows the numeric precisions available in Igor.

Precision	Type	Range	Bytes per Point
double	floating point	10^{-324} to 10^{+307} (~15 decimal digits)	8
single	floating point	10^{-45} to 10^{+38} (~7 decimal digits)	4
signed integer	integer	-2,147,483,647 to 2,147,483,648	4
signed word	integer	-32,768 to 32,767	2
signed byte	integer	-128 to 127	1
unsigned integer	integer	0 to 4,294,967,295	4
unsigned word	integer	0 to 65,535	2
unsigned byte	integer	0 to 255	1

For most work, single precision waves are appropriate.

Single precision waves take up half the memory and disk space of double precision. With the exception of the FFT, Igor uses double precision for all calculations regardless of the numeric precision of the source wave. However, the narrower dynamic range and smaller precision of single precision is not appropriate for all data. If you are not familiar with numeric errors due to limited range and precision, it is safer to use double precision for analysis.

Integer waves are intended for data acquisition purposes and are not intended for use in analysis. See **Integer Waves** on page II-101 for details.

Default Wave Properties

When you create a wave using the **Make** operation (see page V-366) operation with no optional flags, it has the following default properties.

Property	Default
Number of points	128
Precision	single
X scaling	x0=0, dx=1 (point scaling)
X units	blank
Data units	blank

These are the key wave properties. For a comprehensive list of properties, see **Complete List of Wave Properties** on page II-104.

If you make a wave by loading it from a file or by typing in a table, it has the same default properties except for the number of points.

However you make waves, you should use the Change Wave Scaling dialog to set their X scaling and units.

It is possible to change the default wave properties using the **SetScale** operation (see page V-564).

Make Operation

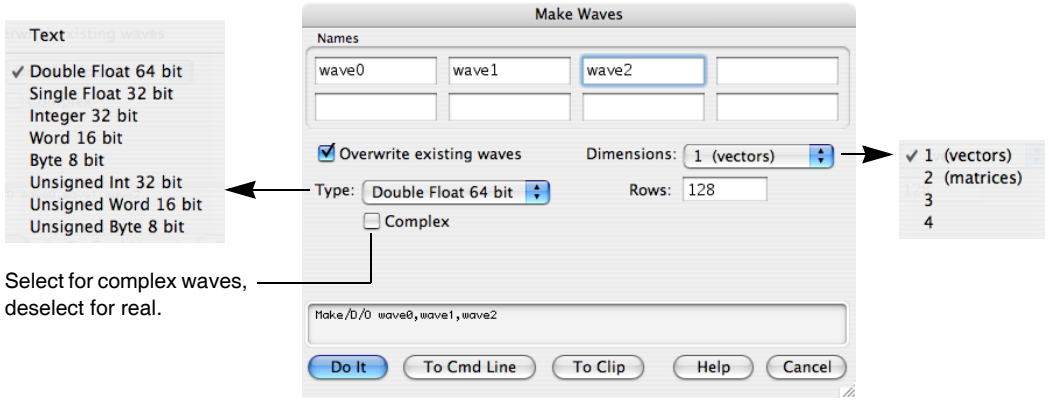
Most of the time you will probably make waves by loading data from a file (see Chapter II-9, **Importing and Exporting Data**), by entering it in a table (see **Using a Table to Create New Waves** on page II-195), or by duplicating existing waves (see **Duplicate Operation** on page II-86).

The **Make** operation is used for making new waves. See the **Make** operation (see page V-366) for additional details.

Here are some reasons to use Make:

- To make waves to play around with.
- For plotting mathematical functions.
- To hold the output of analysis operations.
- To hold miscellaneous data, such as the parameters used in a curve fit or temporary results within an Igor procedure.

The Make Waves dialog provides an interface to the **Make** operation. To use it, choose Make Waves from the Data menu.



You can make 1 to 8 waves with this dialog. You can use it any number of times to create as many waves as your memory will hold. It is most often used to create 1D numeric waves but can also create multidimensional waves and text waves.

Waves have a definite number of points. Unlike a spreadsheet program which automatically ignores blank cells at the end of a column, there is no such thing as an "unused point" in Igor. You can change the number of points in a wave using the Redimension Waves dialog or the **Redimension** operation (see page V-513).

The "Overwrite existing waves" option is useful when you don't know or care if there is a wave with the same name as the one you are about to make.

Make Operation Examples

Make coefs for use in curve fitting:

```
Make/O coefs = {1.5, 2e-3, .01}
```

Make a wave for plotting a math function:

```
Make/O/N=200 test; SetScale x 0, 2*PI, test; test = sin(x)
```

Make a 2D wave for image or contour plotting:

```
Make/O/N=(20,20) w2D; w2D = (p-10)*(q-10)
```

Make a text wave for a category plot:

```
Make/O/T quarters = {"Q1", "Q2", "Q3", "Q4"}
```

It is often useful to make a clone of an existing wave. Don't use Make for this. Instead use the **Duplicate** operation (see page V-133).

Make/O does not preserve the contents of a wave and in fact will leave garbage in the wave if you change the number of points, numeric precision or numeric type. Therefore, after doing a Make/O you should not assume anything about the wave's contents. If you know that a wave exists, you can use the **Redimension** operation instead of Make. Redimension does preserve the wave's contents (however, see the **Redimension** operation (see page V-513)).

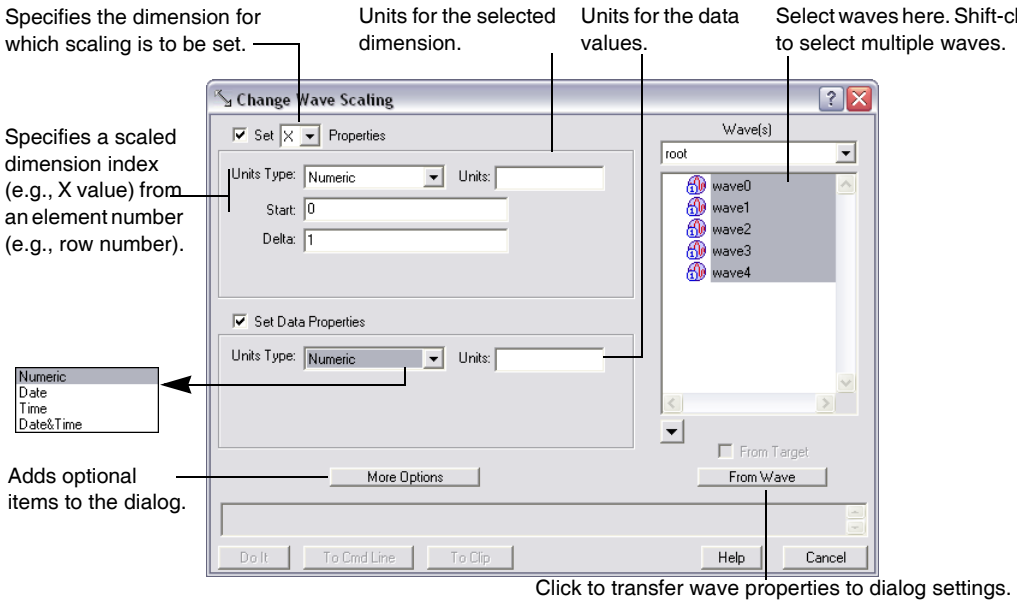
Waves and the Miscellaneous Settings Dialog

The state of the precision items in the Make Waves and Load Waves dialogs, and the way Igor Binary waves are loaded (whether they are copied or shared) are preset with the Miscellaneous Settings dialog using the Data Loading Settings category; see **Miscellaneous Settings** on page III-411.

Changing Dimension and Data Scaling

When you make a 1D wave, it has default X scaling, X units and data units. You should use the **SetScale** operation (see page V-564) to change these properties.

The Change Wave Scaling dialog provides an interface to the **SetScale** operation. To use it, choose Change Wave Scaling from the Data menu.



Scaled dimension indices can represent ordinary numbers, dates, times or date&time values. In the most common case, they represent ordinary numbers and you can leave the Units Type pop-up menu in the Set X Properties section of the dialog on its default value: Numeric.

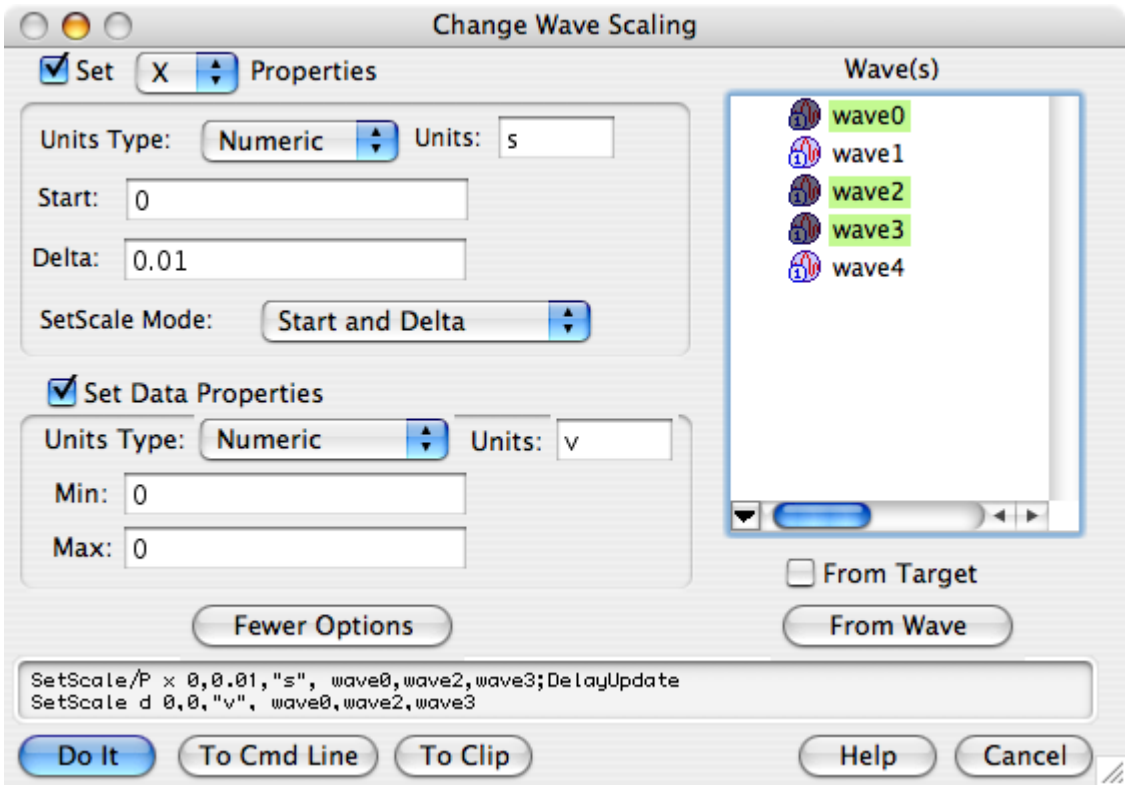
If your data is waveform data, you should enter the appropriate Start and Delta X values. If your data is XY data, you should enter 0 for Start and 1 for Delta. This results in the default "point scaling" in which the X value for a point is the same as the point number.

Normally you should leave the Set X Properties and Set Data Properties checkboxes selected. Deselect one of them if you want the dialog to generate commands to set only X or only Data properties. When working with multidimensional data, the X of Set X Properties can be changed to Y, Z or T via the pop-up menu. See Chapter II-6, **Multidimensional Waves**.

If you want to observe the properties of a particular wave, double-click it in the list or select the wave and then click the From Wave button. This sets all of the dialog items according to that wave's properties.

Igor uses the dimension and data Units to automatically label axes in graphs. Igor can handle units consisting of 49 characters or less. Typically, units should be short, standard abbreviations such as "m", "s", or "g". If your data has more complex units, you can enter the complex units or you may prefer to leave the units blank.

If you click More Options, Igor will display some additional items in the dialog. These items add some convenience but also tend to obscure the critical purpose of the dialog. With the additional options, the dialog looks like this:



In spite of the fact that there is only one way of calculating X values, there are three ways you can specify the x0 and dx values. The SetScale Mode pop-up menu changes the meaning of the scaling entries above. The simplest way is to simply specify x0 and dx directly. This is the Start and Delta mode in the dialog and is the only way of setting the scaling unless you click the More Options button. As an example, if you have data that was acquired by a digitizer that was set to sample at 1 MHz starting 150 µs after t=0, you would enter 150E-6 for Start and 1E-6 for Delta.

The other two ways of specifying X scaling are to set the starting and ending X values are and to calculate dx from the number of points. In the Start and End mode you specify the X value of the last data point. Using the Start and Right mode you specify the X at the end of the last interval. For example, assuming our digitizer (above) created a 100 point wave, we would enter 150E-6 as Start for either mode. If we selected the Start and End mode we would enter 249E-6 for End (150E-6 + 99*1E-6). If we selected Start and Right we would enter 250E-6 for Right.

The min and max entries allow you to set a property of a wave called its "data full scale". This property doesn't serve a critical purpose. Igor does not use it for any computation or graphing purposes. It is merely a way for you to document the conditions under which the wave data was acquired. For example, if your data comes from a digital oscilloscope and was acquired on the ±10v range, you could enter -10 for min and +10 for max. When you make waves, both of these will initially be set to zero. If your data has a meaningful data full scale, you can set them appropriately. Otherwise, leave them zero.

The data units, on the other hand *are* used for graphing purposes, just like the dimension units.

Date, Time, and Date&Time Units

The units "dat" are special, specifying that the scaled dimension indices or data values of a wave contain date, time, or date&time information. In these cases, waves must be double-precision floating point in order to have enough precision to represent dates accurately.

For example, if you have a waveform that contains some quantity measured once per day, you would set the X units for the wave to "dat", set the starting X value to the date on which the first measurement was done, and set the Delta X value to one day. Choosing Date from the Units Type pop-up menu sets the X

units to "dat". You can enter the starting value as a date rather than as a number of seconds since 1/1/1904, which is how Igor represents dates internally. When Igor graphs the waveform, it will notice that the X units are "dat" and will display dates on the X axis.

If instead of a waveform, you have an XY pair, you would set the data units of the X wave to "dat", by choosing Date from the Units Type pop-up menu in the Set Data Properties section of the dialog. When you graph the XY pair, Igor will notice that the X wave contains dates and will display dates on the X axis.

The Units Type pop-up menus do not correspond directly to any property of a wave. That is, a wave doesn't have a units type property. Instead, these menus merely identify what kind of values you are dealing with so that the dialog can display the values in the appropriate format.

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-102.

For information on dates and times in tables, see **Date/Time Formats** on page II-216.

For information on dates and times in graphs, see **Date/Time Axes** on page II-276.

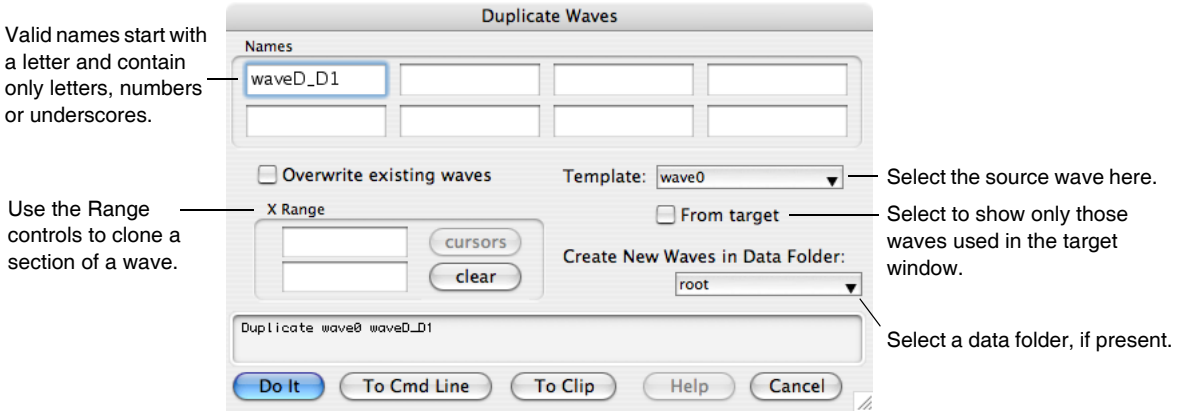
Duplicate Operation

Duplicate is a handy and frequently-used operation. It can make new waves that are exact clones of existing waves. It can also clone a section of a wave and thus provides an easy way to break a big wave up into smaller waves.

Here are some reasons to use Duplicate:

- To hold the results of a transformation (e.g. integration, differentiation, FFT) while preserving the original data.
- To hold the "destination" of a curve fit.
- For holding temporary results within an Igor procedure.
- To extract a section of a wave.

The Duplicate Waves dialog provides an interface to the **Duplicate** operation (see page V-133). To use it, choose Duplicate Waves from the Data menu.



The cursors button is used in conjunction with a graph. You can make a graph of your template wave. Then put the cursors on the section of the template that you want to extract. Choose Duplicate Waves from the Data menu and click the cursors button. Then click Do It. This clones the section of the template wave identified by the cursors.

People sometimes make the mistake of using the **Make** operation when they should be using Duplicate. For example, the destination wave in a curve fit must have the same number of points, numeric type and numeric precision as the source wave. Duplicating the source wave insures that this will be true.

Duplicate Operation Examples

Clone a wave and then transform the clone:

```
Duplicate/O wave0, wave0_d1; Differentiate wave0_d1
```

Use Duplicate to inherit the properties of the template wave:

```
Make/N=200 wave0; SetScale x 0, 2*PI, wave0; wave0 = sin(x)
Duplicate wave0, wave1; wave1 = cos(x)
```

Make a destination wave for a curve fit:

```
Duplicate/O data1, data1_fit
CurveFit gauss data1 /D=data1_fit
```

Compare the first half of a wave to the second:

```
Duplicate/O/R=[0,99] data1, data1_1
Duplicate/O/R=[100,199] data1, data1_2
Display data1_1, data1_2
```

We often use the /O flag (overwrite) with Duplicate because we don't know or care if a wave already exists with the new wave name.

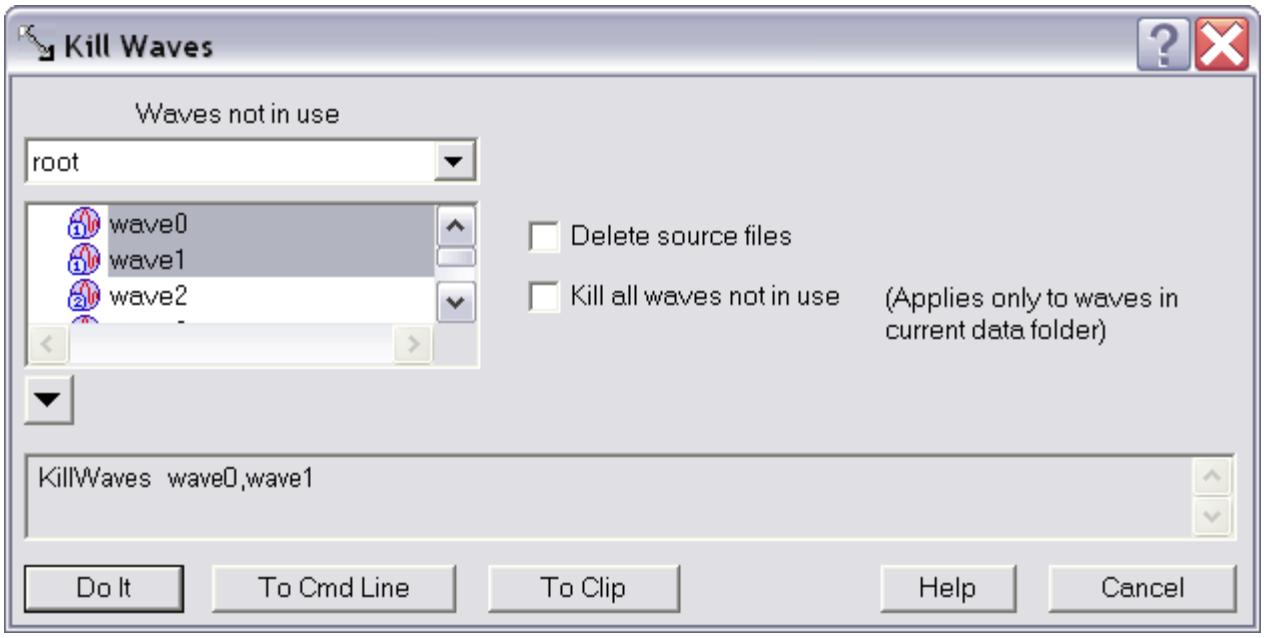
Killing Waves

The **KillWaves** operation (see page V-323) removes waves from the current experiment. This releases the memory used by the waves. Waves that you no longer need clutter up lists and pop-up menus in dialogs. By killing them, you reduce this clutter.

Here are some situations in which you would use KillWaves:

- You are finished examining data that you loaded from a file.
- You are finished using a wave that you created for experimentation.
- You no longer need a wave that you created for temporary use in an Igor procedure.

The Kill Waves dialog provides an interface to the **KillWaves** operation. To use it, choose Kill Waves from the Data menu.



Igor will not let you kill waves that are used in graphs, tables or user defined functions so they do not appear in the list.

Note: Igor can not tell if a wave is referenced from a macro. Thus, Igor will let you kill a wave that is referenced from a macro but not used in any other way. The most common case of this is when you close a graph and save it as a recreation macro. Waves that were used in the graph are now used only in the macro and Igor will let you kill them. If you execute the graph recreation macro, it will be unable to recreate the graph.

KillWaves can delete the Igor Binary file from which a wave was loaded, called the "source file". This is normally not necessary because the wave you are killing either has never been saved to disk or was saved as part of a packed experiment file and therefore was not loaded from a standalone file.

The "Kill all waves not in use" option is intended for those situations where you have created an Igor experiment that contains procedures which load, graph and process a batch of waves. After you have processed one batch of waves, you can kill all graphs and tables and then kill all waves in the experiment in preparation for loading the next batch. This affects only those waves in the current data folder; waves in any other data folders will not be killed.

KillWaves Operation Examples

Here are some simple examples using KillWaves. See also the "Kill Waves" procedure file in the "WaveMetrics Procedures" folder.

```
// Kills all target windows and all waves.
// Does not kill nontarget windows (procedure and help windows).
Function KillEverything()
   String windowName

   do
      windowName = WinName(0, 1+2+4+16+64)// Get next target window
      if (CmpStr(windowName, "") == 0) // If name is ""
         break                         // we are done so break loop
      endif
      DoWindow/K $windowName           // Kill this target window
   while (1)

   KillWaves/A                         // Kill all waves
End

// This illustrates killing a wave used temporarily in a procedure.
Function/D Median(w)              // Returns median value of wave w
   Wave w

   Variable result

   Duplicate/O w, temp            // Make a clone of wave
   Sort temp, temp                // Sort clone
   result = temp[numpnts(temp)/2]

   KillWaves temp                 // Kill clone

   return result
End
```

Browsing Waves

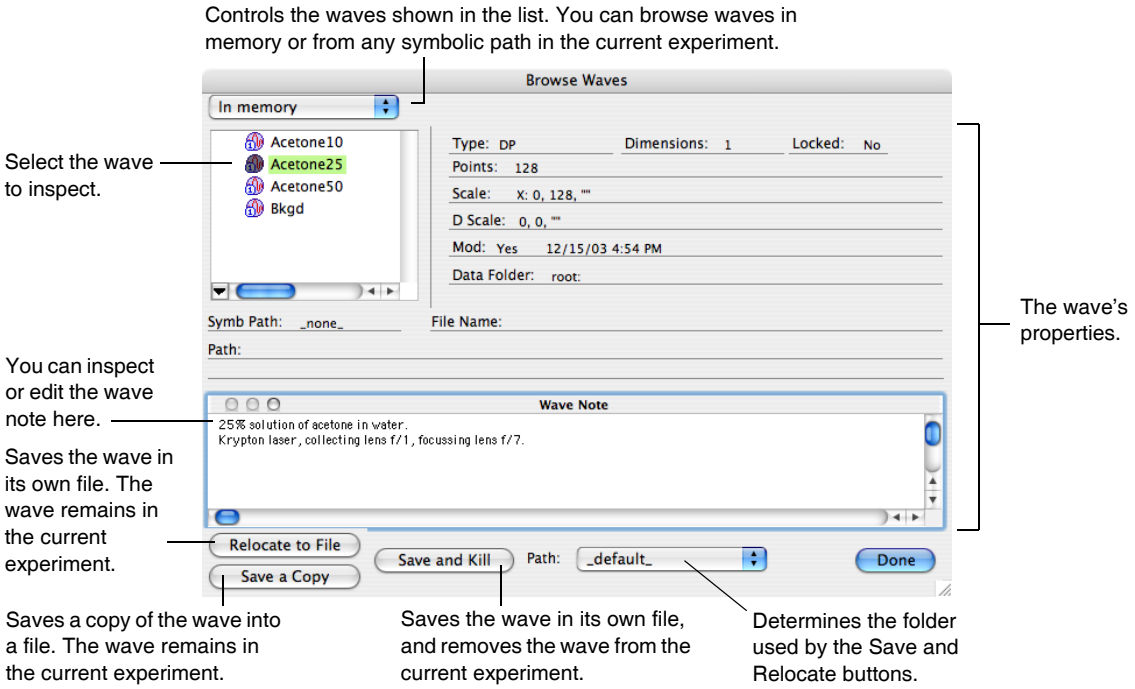
The Data Browser (Data menu) lets you see what waves (as well as strings and variables) exist at any given time. It also lets you see what data folders exist and set the current data folder. The Data Browser is described in detail in Chapter II-8, **Data Folders**. Note that the Data Browser is an external code module (XOP) and will not be available if you have removed it from the Igor Extensions folder.

The Browse Waves dialog (also in the Data menu) lets you examine wave properties, such as the number of points, numeric type, X scaling and wave note.

You can use the Browse Waves dialog to:

- Inspect the properties of your waves.
- Edit the wave note (arbitrary text that Igor stores with each wave).
- Move waves between memory and disk files.

Note: Use the Relocate to File button with care. This button can create a situation in which your experiment depends on files stored separately from the experiment. This causes problems if you move the experiment to another computer because you must also move the separate files. See Chapter II-3, **Experiments, Files and Folders** for further explanation.



The wave type shown in the Type field is abbreviated as follows:

Abbreviation	Description
SP	single precision floating point
DP	double precision floating point
INT32	32 bit signed integer
INT16	16 bit signed integer
INT8	8 bit signed integer
UINT32	32 bit unsigned integer
UINT16	16 bit unsigned integer
UINT8	8 bit unsigned integer
CMPLX	complex

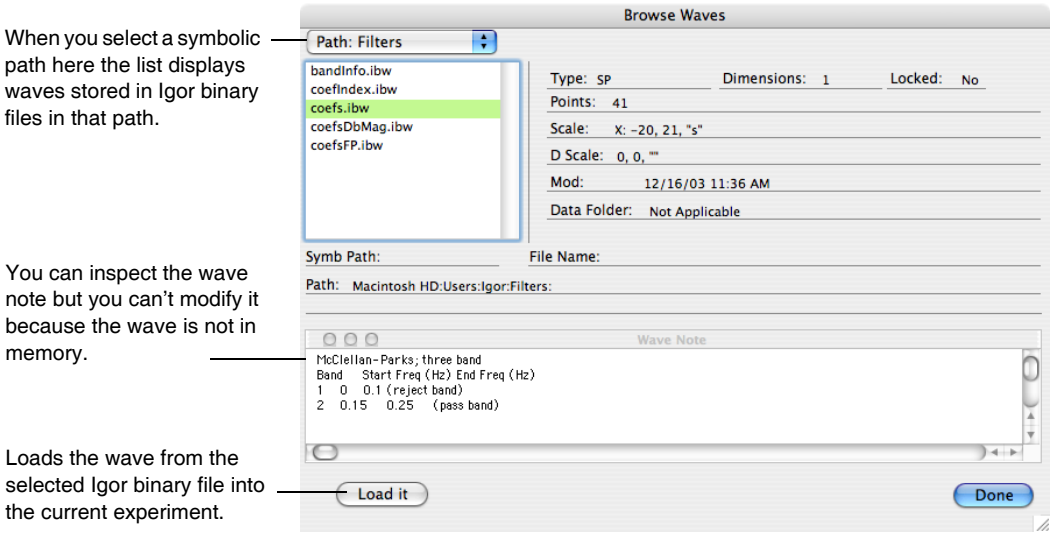
Most often this dialog is used to check the X scaling of or number of points in a wave. It is also often used to inspect or edit the wave note. See **Using the Wave Note** on page II-101 for details.

The Save and Kill button is useful in certain situations involving data acquisition. For example, when you are repeating the same experiment many times under different conditions, you can document the conditions in the wave note and then archive the wave in a data folder for later analysis.

The Path pop-up menu includes a "_default_" item in addition to all the symbolic paths that are defined in the current experiment. The "_default_" item represents the path containing the file the wave was originally loaded from (if any) or the experiment's "home folder".

The pop-up menu above the wave list determines what waves appear in the list. Normally, you will leave this set to In Memory and the list will display only those waves that are in the current data folder of the current experiment. The From Target item displays only those waves in the current data folder and in the target graph or table.

The other items in the pop-up are the names of symbolic paths that exist in the current experiment. If you select one of these items, the list displays unpacked Igor Binary wave files stored in the path and the buttons at the bottom of the dialog change.



Note: The "Load it" button has the same potential pitfall as described above for the Relocate to File button. If you use this, your experiment will depend on the standalone Igor Binary file that you loaded.

Renaming Waves

You can rename a wave using:

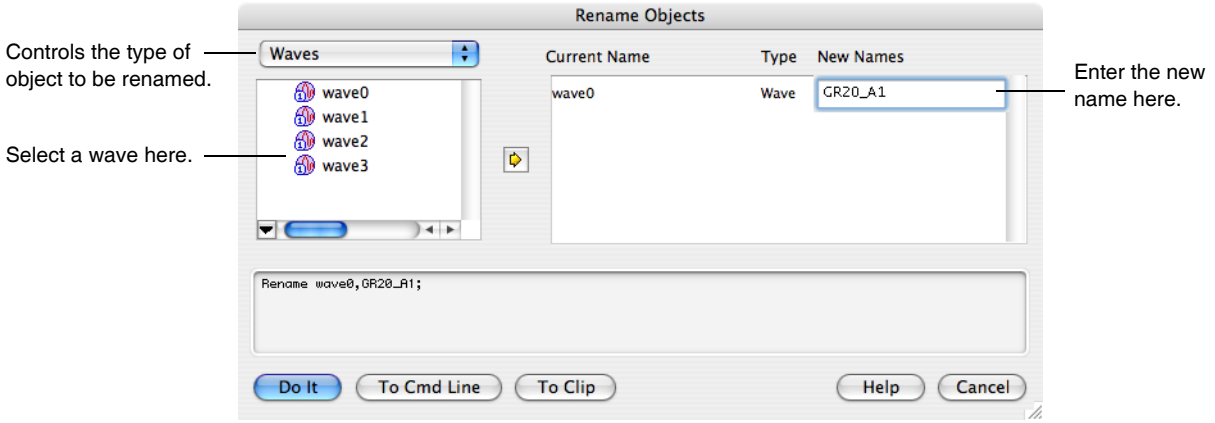
- The Data Browser
- The Rename dialog (Data menu)
- The **Rename** operation from the command line

The **Rename** operation (see page V-519) renames waves as well as other objects.

Here are some reasons for renaming waves:

- You have loaded a bunch of waves from a file and Igor auto-named the waves.
- You have decided on a naming convention for waves and you want to make existing waves follow the convention.
- You are about to load a set of waves whose names will be the same as existing waves and you want to get the existing waves out of the way but still keep them in memory. (You could also achieve this by moving them to a new data folder.)

To use the **Rename** operation, choose Rename from the Data menu. This brings up the Rename Objects dialog.

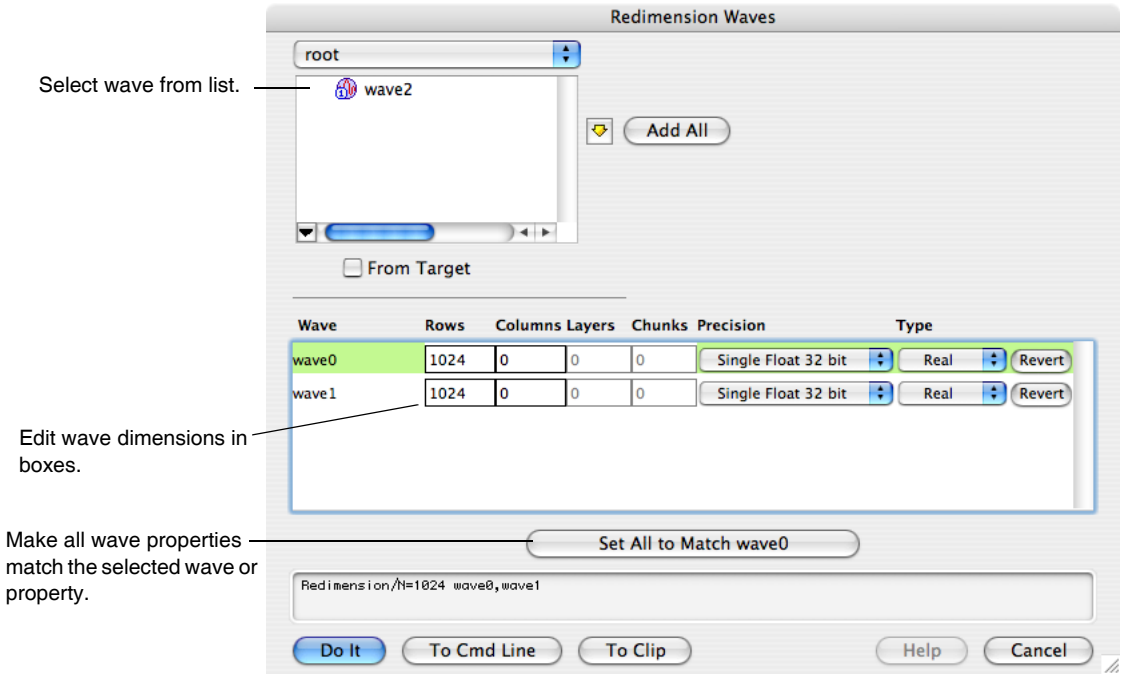


Redimensioning Waves

Redimension can change the following properties of a wave:

- The number of dimensions in the wave.
- The number of elements in each dimension.
- The numeric precision (e.g., single to double).
- The numeric type (e.g., real to complex).

The Redimension Waves dialog provides an interface to the **Redimension** operation (see page V-513). To use it, choose Redimension Waves from the Data menu.



When Redimension adds new elements to a wave, it sets them to zero for a numeric wave and to blank for a text wave.

The following commands illustrate two ways of changing the numeric precision of a wave. Redimension preserves the contents of the wave whereas Make does not.

```
Make/N=5 wave0=x
Edit wave0
```

```
Redimension/D wave0         // This preserves the contents of wave0.
Make/O/D/N=5 wave0          // This does not.
```

See **Vector (Waveform) to Matrix Conversion** on page II-113 for information on converting a 1D wave into a 2D wave while retaining the data (i.e., reshaping).

You cannot change a wave from numeric to text or vice versa. The following examples illustrate how you can make a text copy of a numeric wave and a numeric copy of a text wave:

```
Make/N=10 numWave = p
Make/T/N=(numpnts(numWave)) textWave = num2str(numWave)
Make/N=(numpnts(textWave)) numWave2 = str2num(textWave)
```

However, you can lose precision because num2str prints with only 6 digits of precision.

Inserting Points

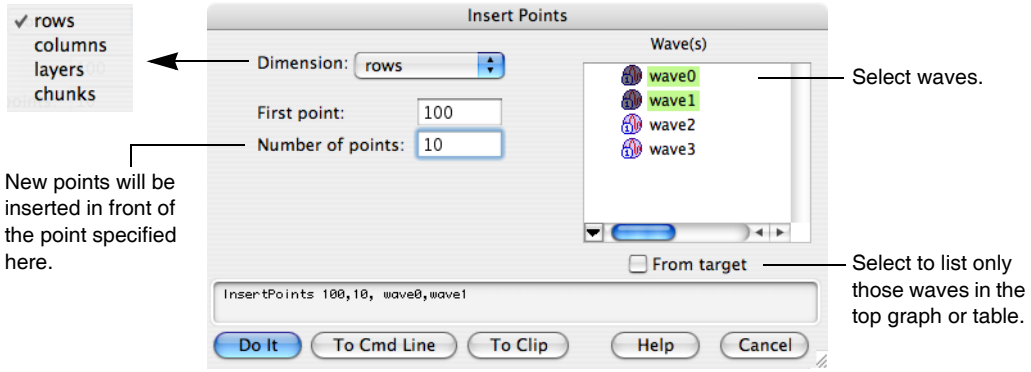
There are two ways to insert new points in a wave. You can do this by:

- Using the **InsertPoints** operation.
- Typing or pasting in a table.

This section deals with the **InsertPoints** operation (see page V-309). For information on typing or pasting in a table, see Chapter II-11, **Tables**.

Using the **InsertPoints** operation, you can insert new data points at the start, in the middle or at the end of a 1D wave. You can also insert new elements in multidimensional waves. For example, you can insert new columns in a 2D matrix wave. The inserted values will be 0 for a numeric wave and "" for a text wave.

The Insert Points dialog provides an interface to the **InsertPoints** operation. To use it, choose Insert Points from the Data menu.



If the value that you enter for first point is greater than the number of elements in the selected dimension of a selected wave, the new points are added at the end of the dimension. InsertPoints can change the dimensionality of a wave. For example, if you insert a column in a 1D wave, you end up with at 2D wave.

If the top window is a table at the time that you select Insert Points, Igor will preset the dialog items based on the selection in the table.

Deleting Points

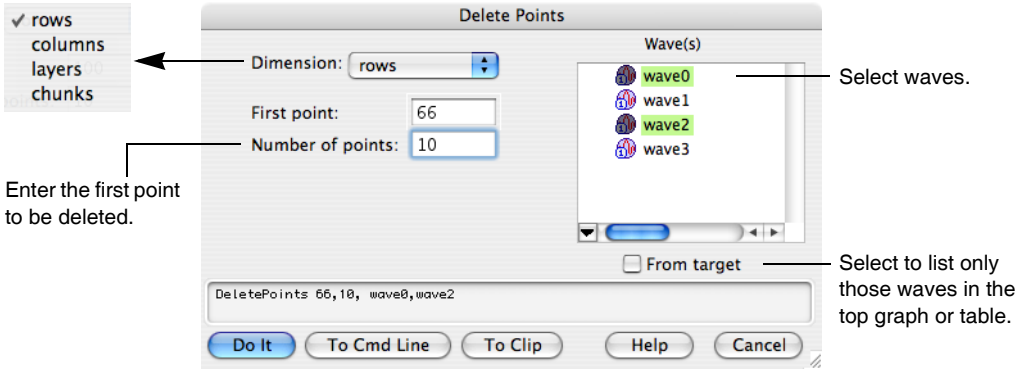
There are two ways to delete points from a wave. You can do this by:

- Using the **DeletePoints** operation.
- Doing a cut in a table.

This section deals with the **DeletePoints** operation (see page V-113). For information on cutting in a table, see Chapter II-11, **Tables**.

Using the **DeletePoints** operation, you can delete data points from the start, middle or end of a 1D wave. You can also delete elements from multidimensional waves. For example, you can delete columns from a 2D matrix wave.

The Delete Points dialog provides an interface to the **DeletePoints** operation. To use it, choose Delete Points from the Data menu.



If the value that you enter for first point is greater than the number of elements in the selected dimension of a selected wave, DeletePoints will do nothing to that wave. If the number of elements is too large, DeletePoints will delete from the specified first element to the end of the dimension. For multidimensional waves, if you delete all but one element of a given dimension, the wave is changed to a lower dimensionality. For example, if you have a 3D wave and delete all but one layer, you are left with a 2D wave.

If the top window is a table at the time that you choose Delete Points, Igor will preset the dialog items based on the selection in the table.

Waveform Arithmetic and Assignments

Waveform arithmetic is the most flexible and powerful part of Igor's analysis capability. You can write assignment statements that work on an entire wave or on a subset of a wave much as you would write an assignment to a single variable in a standard programming language.

This section deals with waveform arithmetic on 1D waves. See also **Multidimensional Wave Assignment** on page II-111.

In a wave assignment, a wave appears on the left side and a mathematical expression appears on the right side. Here are some examples.

```
wave0 = sin(x)
wave0 = log(wave1/wave2)
wave0[0,99] = wave1[100 + p]
```

A wave on the left side is called the **destination wave**. A wave on the right side is called a **source wave**.

Usually, source waves have the same number of points and X scaling as the destination wave. In rare cases, it is useful to write a wave assignment where this is not true. See **Mismatched Waves** on page II-99 for a discussion of this.

When Igor executes a wave assignment, it evaluates the expression on the right-hand side one time for each point in the destination wave. The result of each evaluation is stored in the corresponding point in the destination wave.

During execution, the symbol *p* has a value equal to the number of the point in the destination wave which is being evaluated and the symbol *x* has a value equal to the X value at that point. The X value for a given point is determined by the number of the point and the X scaling for the wave. To see this, try the following:

```
Make/N=5 wave0; SetScale/P x 0, .1, wave0; Edit wave0.xy
wave0 = p
wave0 = x
```

The first assignment sets the value of each point of wave0 to the point number. The second assignment sets the value of each point of wave0 to the X value for that point.

In Igor Pro 3.0, we extended Igor from one dimension to multiple (up to 4) dimensions. Just as the symbol p returns the current element number in the rows dimension, the symbols q, r and s return the current element number in the columns, layers and chunks dimensions. The symbol x in the rows dimension has analogs y, z and t in the columns, layers and chunks dimensions. See Chapter II-6, **Multidimensional Waves**, for details.

A source wave returns its data value at the point being evaluated. In the example

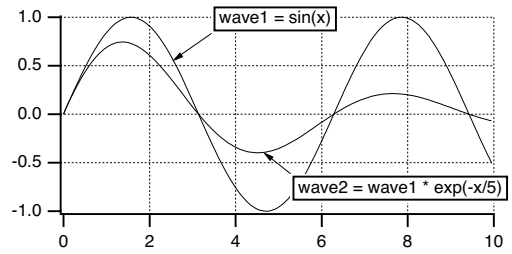
```
wave0 = log(wave1/wave2)
```

Igor evaluates the right-hand expression once for each point in wave0. During each evaluation of the expression, wave1 and wave2 return their data values at the point being evaluated.

The right-hand expression is evaluated in the context of the data folder containing the destination wave. See **Data Folders and Assignment Statements** on page II-126 for details.

This command sequence illustrates some of these ideas.

```
Make/N=200 wave1, wave2                 // 2 waves, 200 points each
SetScale/P x, 0, .05, wave1, wave2      // set X values from 0 to 10
Display wave1, wave2                    // create a graph of waves
wave1 = sin(x)                          // assign values to wave1
wave2 = wave1 * exp(-x/5)               // assign values to wave2
```



Since wave1 has 200 points, the wave assignment `wave1=sin(x)` evaluates `sin(x)` 200 times, once for each point in wave1. The first point of wave1 is point number 0 and the last point of wave1 is point number 199. The symbol p, not used in this example, goes from 0 to 199. The symbol x steps through the 200 X values for wave1 which start from 0 and step by .05, as specified by the SetScale command. The result of each evaluation is stored in the corresponding point in wave1, making wave1 about 1.5 cycles of a sine wave.

Since wave2 also has 200 points, the wave assignment `wave2=wave1*exp(-x/5)` evaluates `wave1*exp(-x/5)` 200 times, once for each point in wave2. In this assignment, the right-hand expression contains a wave, wave1. As Igor executes the assignment, p goes from 0 to 199. Each of the 200 times the right side is evaluated, wave1 returns its data value for the corresponding point. The result of each evaluation is stored in the corresponding point in wave2 making wave2 about 1.5 cycles of a damped sine wave.

The effect of a wave assignment is to set the data values of the destination wave. Igor does not remember the functional relationship implied by the assignment. In this example, if you changed wave1, wave2 would not change automatically. If you wanted wave2 to have the same functional relationship to wave1 as it had before you changed wave1, you would have to reexecute the `wave2=wave1*exp(-x/5)` assignment.

There is a special kind of wave assignment that *does* establish a functional relationship. See **Wave Dependency Formulas** on page II-101 for details.

In Igor Pro 6.1 or later, you can use multiple processors to execute a waveform assignment statement that takes a long time. See **Automatic Parallel Processing with MultiThread** on page IV-283 for details.

Indexing and Subranges

Igor provides two ways to refer to a specific point or range of points in a 1D wave: X value indexing and point number indexing. Consider the following examples.

```
wave0(54) = 92          // sets wave0 at X=54 to 92
wave0[54] = 92          // sets wave0 at point 54 to 92
wave0(1,10) = 92        // sets wave0 from X=1 to X=10 to 92
wave0[1,10] = 92        // sets wave0 from point 1 to point 10 to 92
```

Parentheses specify the range start and end values in terms of X. It is the equivalent of using brackets with the **x2pnt** function to translate X values to point numbers. Brackets index the wave in terms of point number — the number or numbers inside the parentheses are in terms of point numbers of the indexed wave. If the wave has point scaling then these two methods have identical effects. However, if you set the X scaling of the wave to other than point scaling then these commands behave differently. In both cases the range is *inclusive*.

You can specify not only a range but also a point number increment. For example:

```
wave0[0,98;2] = 1       // sets even numbered points in wave0 to 1
wave0[1,99;2] = -1      // sets odd numbered points in wave0 to -1
```

The number after the semicolon is the increment. Igor begins at the starting point number and goes up to and including the ending point number, skipping by the increment. At each resulting point number, it evaluates the right-hand side of the wave assignment and sets the destination point accordingly. Increments can also be used when you specify a range in terms of X value but the increment is always in terms of point number. For example:

```
wave0(0,100;5) = PI     // sets wave0 at specified X values to PI
```

Here, Igor starts from the point number corresponding to x = 0 and goes up to and including the point number that corresponds to x = 100. The point number is incremented by 5 at each iteration.

You can take some shortcuts in specifying the range of a destination wave. The subrange start and end values can both be missing. When the start is missing, point number zero is used and when the end is missing, the last point of the wave is assumed. You can also use a * character to specify the last point. A missing increment value defaults to a single point.

Here are some examples that illustrate these shortcuts:

```
wave0[ ,50] = 13            // sets wave0 from point 0 to point 50
wave0[51,] = 27             // sets wave0 from point 51 to last point
wave0[ , ;2] = 18.7         // sets every even point of wave0
wave0[1,*;2] = 100          // sets every odd point of wave0
```

A subrange of a destination wave may consist of a single point or a range of points but a subrange of a source wave must consist of a single point. In other words the wave assignment:

```
wave1(4,5) = wave2(5,6)     // Illegal!
```

is not legal. In this assignment, x ranges from 4 to 5. You can get the desired effect using:

```
wave1(4,5) = wave2(x+1)     // OK!
```

By virtue of the range specified on the left hand side, x goes from 4 to 5. Therefore, x+1 goes from 5 to 6 and the right-hand expression returns the values of wave2 from 5 to 6.

If, in specifying a subrange, you use an X value that is out of range, Igor clips it to the closest valid X value. For example, the smallest X value of our sample waves is zero because of the X scaling that we assigned to them. If you use `wave1(-0.5)` Igor clips the `-0.5` to `0` and therefore returns `wave1(0)`. Future versions of Igor may regard this as an error so you should avoid using invalid subranges.

Interpolation in Wave Assignments

If you specify a fractional point number or an X value that falls between two data points, Igor will return a linearly interpolated data value. For example, wave1[1.75] returns the value of wave1 3/4 of the way from the data value of point 1 to the data value of point 2. This interpolation is done only for one-dimensional waves. See **Multidimensional Wave Assignment** on page II-111, for information on assignments with multidimensional data.

This is a very powerful feature. Imagine that you have an evenly spaced calibration curve, called calibration, and you want to find the calibration values at a specific set of X coordinates as stored in a wave called xData. If you have set the X scaling of the calibration wave, you can do the following:

```
Duplicate xData, yData
yData = calibration(xData)
```

This uses the interpolation feature of Igor's wave assignment to find a linearly-interpolated value in the calibration wave for each X coordinate in the xData wave.

Lists of Values

You can assign values to a wave or to a subrange of a wave using a list of values. For example:

```
wave0 = {1, 3, 5}          // sets length of wave0 to three
                           // and sets Y values to 1, 3, 5
wave0[10]= {1, 3, 5}       // sets points 10 through 12 to 1, 3, 5
```

In these examples, {1, 3, 5} is a list of values.

If, in the second example, wave0 had less than 10 points, it would have been automatically extended with zeros before setting points 10 through 12.

Wave Initialization

From Igor's command line, you can make a wave and initialize it with a single command, as illustrated in the following examples:

```
Make wave0=sin(p/8)        // wave0 has default number of points
Make coeffs={1,2,3}        // coeffs has just three points
```

Example: Normalizing Waves

When comparing the shape of multiple waves you may want to normalize them so that the share a common range. For example:

```
// Create some sample data
Make waveA = 3*sin(x/8)
Make waveB = 2*sin(pi/16 + x/8)

// Display the waves
Display waveA, waveB
ModifyGraph rgb(waveB)=(0,0,65535)

// Normalize the waves
Variable aMin = WaveMin(waveA)
Variable bMin = WaveMin(waveB)
waveA -= aMin
waveB -= bMin
Variable aMax = WaveMax(waveA)
Variable bMax = WaveMax(waveB)
waveA /= aMax
waveB /= bMax
```

Note the use of the temporary variables aMin and bMin. They are needed for two reasons. First, if we wrote `waveA -= WaveMin(waveA)`, then **WaveMin** would be called once for each point in waveA, which would

be a waste of time. Worse than that, the minimum value in waveA would change during the course of the waveform assignment statement, giving incorrect results.

There are sometimes faster ways to do waveform arithmetic. For large waves, the **FastOp** and **MatrixOp** operations provide increased speed:

```
waveA -= aMin                       // FastOp does not support wave-variable
FastOp waveA = (1/aMax) * waveA

MatrixOp/O waveA = waveA - aMin
MatrixOp/O waveA = waveA / aMax
```

Example: Converting XY Data to Waveform Data

There are some times when it is desirable to convert XY data to uniformly spaced waveform data. For example, the Fast Fourier Transform requires uniformly spaced data. If you have measured XY data in the time domain, you would need to do this conversion before doing an FFT on it.

We can make some sample XY data as follows:

```
Make/N=1024 xWave, yWave
xWave = 2*PI*x/1024 + gnoise(.001)
yWave = sin(xwave)
```

xWave has values from 0 to 2π with a bit of noise in them. Our data is not uniformly spaced in the x dimension but it is monotonic — always increasing, in this case. If it were not monotonic we could sort the XY pair.

We can create a waveform representing our XY data as follows:

```
Duplicate ywave, wave0
SetScale x 0, 2*PI, wave0
wave0 = interp(x, xwave, ywave)
```

The SetScale command sets the scaling of wave0 so that its X values run from 0 to 2π. Its data values are generated by picking a value off the curve represented by ywave versus xwave at each of these X values using linear interpolation.

See **Converting XY Data to a Waveform** on page III-116 for more information. It illustrates how to use cubic spline instead of linear interpolation. Also, see the WM Procedures Index help file, which you will find under the Windows→Help Windows menu. This help file provides an index of standard easy-to-use procedures that deal with XY data. These procedures can be accessed by simply copying a single line and pasting it into the procedure window.

Example: Concatenating Waves

Concatenating waves can be done much more easily using the **Concatenate** operation (see page V-58). This simple example serves mainly to illustrate a use of wave assignments.

Suppose we have three waves of 100 points each: wave1, wave2 and wave3. We want to create a fourth wave, wave4, which is the concatenation of the three original waves. Here is the sequence of commands to do this.

```
Make/N=300 wave4
wave4[0,99] = wave1[p]              // set first third of wave4
wave4[100,199] = wave2[p-100]       // set second third of wave4
wave4[200,299] = wave3[p-200]       // set last third of wave4
```

In this example, we use a subrange of wave4 as the destination of our wave assignments. The right-hand expressions index the appropriate values of wave1, wave2 and wave3. Remember that p ranges over the points being evaluated in the destination. So, p ranges from 0 to 99 in the first assignment, from 100 to 199 in the second assignment and from 200 to 299 in the third assignment. In each of the assignments, the wave on the right-hand side has only 100 points, from point 0 to point 99. Therefore, we index the wave on the right-hand side to pick out the 100 values of that wave.

Example: Decomposing Waves

Here is a another example that illustrates a use of wave assignments. Suppose we have a 300 point wave, wave4, that we want to decompose into three waves of 100 points each: wave1, wave2 and wave3. Here is the sequence of commands to do this.

```
Make/N=100 wave1,wave2,wave3
wave1 = wave4[p]              // get first third of wave4
wave2 = wave4[p+100]          // get second third of wave4
wave3 = wave4[p+200]          // get last third of wave4
```

In this example, we use a subrange of wave4 as the source of our data. We index the desired segment of wave4 using point number indexing. Since wave1, wave2 and wave3 each have 100 points, p ranges from 0 to 99. In the first assignment, we access points 0 to 99 of wave4. In the second assignment, we access points 100 to 199 of wave4. In the third assignment, we access points 200 to 299 of wave4.

You could also use the **Duplicate** operation (see page V-133) to make a wave from a section of another wave.

Note that the wave assignment `wave1=wave4` does *not* copy the first 100 points of wave4 to wave1 because wave4 has more points than wave1. This is described in the section **Mismatched Waves** on page II-99.

Example: Complex Wave Calculations

Igor includes a number of built-in functions for manipulating complex numbers and complex waves. These are illustrated in the following examples.

Here, we make a time domain waveform and do an FFT on it to generate a complex wave. The examples show how to pick out the real and imaginary part of the complex wave, how to find the sum of squares and how to convert from rectangular to polar representation. For more information on frequency domain processing, see Chapter III-9, **Signal Processing**.

```
// first, make a time domain waveform
Make/O/N=1024 wave0
SetScale x 0, 1, "s", wave0            // goes from 0 to 1 second
wave0=sin(2*PI*x)+sin(6*PI*x)/3+sin(10*PI*x)/5+sin(14*PI*x)/7
Display wave0 as "Time Domain"

// now, do FFT
Duplicate/O wave0, cwave0              // get copy to do FFT on
FFT cwave0                             // cwave0 is now complex
cwave0 /= 512;cwave0[0] /= 2           // normalize amplitude
Display cwave0 as "Frequency Domain";SetAxis bottom, 0, 25

// calculate magnitude and phase
Make/O/N=513 mag0, phase0, power0      // these are real waves
CopyScales cwave0, mag0, phase0, power0
mag0 = real(r2polar(cwave0))
phase0 = imag(r2polar(cwave0))
phase0 *= 180/PI                       // convert to degrees
Display mag0 as "Magnitude and Phase";AppendToGraph/R phase0
SetAxis bottom, 0, 25
Label left, "Magnitude";Label right, "Phase"

// calculate power spectrum
power0 = magsqr(cwave0)
Display power0 as "Power Spectrum";SetAxis bottom, 0, 25
```

Example: Comparison Operators and Wave Synthesis

The comparison operators ==, >=, >, <= and < can be useful in synthesizing waves. Imagine that you want to set a wave so that its data values all equal -π for x<0 and +π for x>=0. The following wave assignment accomplishes this:

```
wave1 = -pi*(x<0) + pi*(x>=0)
```

This works because the conditional statements return 1 when the condition is TRUE and 0 when it is FALSE, and then the multiplication proceeds.

You can also make such assignments using the conditional operator (see **Operators** on page IV-5):

```
wave0 = (x>0) ? pi : -pi
```

A series of impulses can be made using the **mod** function and ==. This wave equation will assign 5 to every tenth point starting with point 0, and 0 to all the other points:

```
wave1 = (mod(p,10)==0)*5
```

Example: Wave Assignment and Indexing Using Labels

A useful, and almost entirely overlooked, feature of dimension labels is that such labels can be used to refer to wave values by a meaningful name. Thus, for example, you can create a wave to store coefficient values and directly refer to these values by the name of the coefficient (e.g., coef[%Friction]) instead of a potentially confusing and less meaningful numeric index (e.g., coef[1]). You can also view the wave values and labels in a table.

You create wave labels using the **SetDimLabel** operation (see page V-553); more details can be found under **Dimension Labels** on page II-109. Label names may be up to 31 characters in length; if you use liberal names, such as those containing spaces, make certain to enclose these names within single quotation marks.

In this example we create a wave and use the **FindPeak** operation (see page V-173) to get peak parameters of the wave. Next we create an output parameter wave with appropriate labels and then assign the FindPeak results to the output wave using the labels.

```
// Make a wave and get peak parameters
Make test=sin(x/30)
FindPeak/Q test

// Create a wave with appropriate row labels
Make/N=6 PeakResult
SetDimLabel 0,0,'Peak Found', PeakResult
SetDimLabel 0,1,PeakLoc, PeakResult
SetDimLabel 0,2,PeakVal, PeakResult
SetDimLabel 0,3,LeadingEdgePos, PeakResult
SetDimLabel 0,4,TrailingEdgePos, PeakResult
SetDimLabel 0,5,'Peak Width', PeakResult

// Fill PeakResult wave with FindPeak output variables
PeakResult[%'Peak Found']   =V_flag
PeakResult[%PeakLoc]        =V_PeakLoc
PeakResult[%PeakVal]        =V_PeakVal
PeakResult[%LeadingEdgePos] =V_LeadingEdgeLoc
PeakResult[%TrailingEdgePos]=V_TrailingEdgeLoc
PeakResult[%'Peak Width']   =V_PeakWidth

// Display the PeakResult values and labels in a table
Edit PeakResult.ld
```

In addition to the method illustrated above, you can also create and edit dimension labels by displaying the wave in a table and showing the dimension labels with the data. See **Showing Dimension Labels** on page II-191 for further details on using tables with labels.

Mismatched Waves

For most applications you will not need to mix waves of different lengths. In fact, doing this is more often the result of a mistake than it is intentional. However, if your application requires mixing you will need to know how Igor handles this.

Let's consider the case of assigning the value of one wave to another with a command such as

```
wave1 = wave2
```

In this assignment, there is no explicit indexing, so Igor evaluates the expression as if you had written:

```
wave1 = wave2[p]
```

If wave2 has more points than wave1, the extra points have no effect on the assignment since p ranges from 0 to n-1, where n is the number of points in wave1.

If wave2 has fewer points than wave1 then Igor will try to evaluate wave2[p] for values of p greater than the length of wave2. In this case, it simply returns the value of the last point in wave2.

It may be that you actually want the values in wave1 to span the values in wave2 by interpolating between values in wave2. To get Igor to do this, you must explicitly index the appropriate X values on the right side. For instance, if you have two waves of different lengths, you can do this:

```
big = small[p*(numpnts(small)-1)/(numpnts(big)-1)]
```

Of course, if you know how many points are in each wave, you can simply type the correct number rather than typing out "`numpnts(small)-1`" and "`numpnts(big)-1`".

NaNs, INFs and Missing Values

The data value of a point in a floating point numeric wave is normally a finite number but can also be a NaN or an INF. NaN means "not a number". An expression returns the value NaN when it makes no sense mathematically. For example, `log(-1)` returns the value NaN. You can also set a point to NaN, using a table or a wave assignment, to represent a missing value. An expression returns the value INF when it makes sense mathematically but has no finite value. `log(0)` returns the value -INF.

The IEEE floating point standard defines the representation and behavior of NaN values. There is no way to represent a NaN in an integer wave. If you attempt to store NaN in an integer wave, you will store a garbage value.

Igor ignores NaNs and INFs in curve fit and wave statistics operations. NaNs and INFs have no effect on the scaling of a graph. When plotting, Igor handles NaNs and INFs properly, as missing and infinite values respectively.

Igor does *not* ignore NaNs and INFs in many other operations, especially those that are DSP related such as FFT. In general, any operation that numerically combines all or most of the data points from a wave will give meaningless results if one or more points is a NaN or INF. Notable examples include the **area** and **mean** functions and the **Integrate** and **FFT** operations. Some operations that only mix a few points such as Smooth and Differentiate will "contaminate" only those points in the vicinity of the NaN or INF. You can use the Interpolate operation (Analysis menu) to create a NaN-free version of a wave.

The "Remove Points" procedure file provides a user- function for removing NaNs from a wave. See the WM Procedures Index help file, which you will find under the Windows→Help Windows menu, for information on how to access it.

If you get NaNs from functions such as **area** or **mean** or operations such as **Convolve** or any other functions or operations that sum points in waves, it indicates that some of the points in the wave are NaN. If you get NaNs from curve-fitting results, it indicates that Igor's curve fitting has failed. See **Curve Fitting Troubleshooting** on page III-231 for troubleshooting tips.

See **Dealing with Missing Values** on page III-119 for techniques for dealing with NaNs.

Strange Cases

You may get unexpected results if the destination of a wave assignment also appears in the right-hand expression. Consider these examples:

```
wave1 -= wave1(5)
wave1 -= vcsr(A)            // where cursor A is on wave1
```

Each of these examples is an attempt to subtract the value of wave1 at a particular point from every point in wave1. This will not work as expected because the value of wave1 at that particular point is altered

during the assignment. At some point in the assignment, wave1(5) or vcsr(A) will return 0 since the value at that point in wave1 will have been subtracted from itself.

You can get the desired result by using a variable to store the value of wave1 at the particular point.

```
K0 = wave1(5); wave1 -= K0
K0 = vcsr(A); wave1 -= K0
```

Wave Dependency Formulas

You can cause a wave assignment to "stick" to the wave by substituting ":=" for "=" in the statement. This causes the wave to become dependent upon the objects referenced in the expression. For example:

```
K0 = 5
wave1 := sin(x/K0)      // Note ":="
Display wave1
```

If you now execute "K0 = 8" you will see the wave automatically update. Similarly if you change the wave's X scaling using the **SetScale** operation (see page V-564), the wave will be automatically recalculated for the new range of X values.

See Chapter IV-9, **Dependencies**, for further discussion.

Using the Wave Note

One of the properties of a wave is the **wave note**. This is just some plain text that Igor stores with each wave. The note is empty when you create a wave. There is no limit on its length.

You can inspect and edit a wave note using the Browse Waves dialog. You can set or get the contents of a wave note from an Igor procedure using the **Note** operation (see page V-445) or the **note** function (see page V-445).

You can see part of a wave note for a wave displayed in a graph or table by pressing the Command-Option-Control (*Macintosh*) or Shift+F1 (*Windows*) and then clicking the wave.

Originally we thought of the wave note as a place for an experimenter to store informal comments about a wave and it is fine for that purpose. However, over time both we and many Igor users have found that the wave note is also a handy place to store additional, user-defined properties of a wave in a structured way. These additional properties are editable using the Browse Waves dialog but they can also be used and manipulated by procedures.

To do this, you store keyword-value pairs in the wave note. For example, a note might look like this:

```
CELLTYPE:rat hippocampal neuron;
PATTERN:1VN21;
TREATMENT:PLACEBO;
```

You could then write Igor functions to set or retrieve the CELLTYPE, PATTERN and TREATMENT properties of a wave. Using these functions you can write other procedures to, for example, display all waves whose TREATMENT property is PLACEBO in a graph.

You can use functions in the "Keyword-Value" procedure file to manipulate keyword-value strings. See the WM Procedures Index help file, which you will find under the Windows→Help Windows menu, for information on how to access the functions.

Integer Waves

Igor provides support for integer waves primarily to aid in data acquisition projects. They allow people who are interfacing with hardware to write/read directly into integer waves. This allows for slightly quicker live display and also saves the XOP writer from having to convert integers to floating point. Integer waves are also appropriate for storing images. Aside from memory considerations there is no other reason to use

integer waves. You might expect that wave assignment statements would evaluate more quickly when an integer wave is the destination. This is not the case, however, because Igor still uses floating point for the assignment and only converts to integer for storage.

Note: Behavior on under/over-flow is undefined.

Date/Time Waves

Dates are represented in Igor date format - as the number of seconds since midnight, January 1, 1904. Dates before that are represented by negative values. There is no practical limit to the range of dates that can be represented except that on Windows dates must be greater than January 1, 1601.

A date can not be accurately stored in the data values of a single precision or integer wave. Make sure to use double precision to store dates and times.

You must set the data units of a wave containing date or date/time data to "dat". This tells Igor that the wave contains date/time data and affects the default display of axes in graphs and columns in tables.

The following example illustrates the use of date/time data. First we create some date/time data and view it in a table:

```
Make/D/N=10/O testDate = date2secs(2011,4,p+1)
SetScale d 0, 0, "dat", testDate // Tell Igor this wave stores date/time data
```

Note that we used `SetScale d` to set the data units of the wave to "dat".

Next we view the wave in a table:

```
Edit testDate
ModifyTable width(testDate)=120  // Make column wider
```

The data is displayed as dates but it is stored as numbers - specifically the number of seconds since January 1, 1904. We can see this by changing the column format:

```
ModifyTable format=1             // Display as integer
```

Now we return to date format:

```
ModifyTable format(testDate)=6   // Display as date again
```

Next we create some time data. This wave will not store dates and therefore does not need to be double-precision:

```
Make/N=10/O testTime = 3600*p    // Data is stored in seconds
AppendToTable testTime
```

Now we create a date/time wave by adding the time data to the date data. Since this wave will store dates it must be double-precision and must have "dat" data units. We accomplish this by using the Duplicate operation to duplicate the original date wave:

```
Duplicate/O testDate, testDateTime
AppendToTable testDateTime
ModifyTable width(testDateTime)=120    // Make wider
```

Igor displays the date/time wave in date format because it has "dat" units. We now change the column format to date/time:

```
ModifyTable format(testDateTime)=8     // Set column to date/time format
```

Finally, we add the time:

```
testDateTime = testDateTime + testTime // Add time to date
```

To check the data type of your waves, choose Data→Browse Waves. The data type shown for date/time waves should be DP (double-precision). If not, use Data→Redimension Waves to redimension as DP.

So far we have looked at storing dates in the data of a wave. Typically such a date wave is used to supply the X wave of an XY pair. More rarely, you might want to store date data in the X values of a wave treated as a waveform. For example:

```
Make/N=100 wave0 = sin(p/8)
SetScale/P x date2secs(2011,4,1), 60*60*24, "dat", wave0
Display wave0
Edit wave0.id; ModifyTable format(wave0.x)=6
```

Here the SetScale command is used to set the X scaling and units of the wave, not the data units as before. In this case, the wave does not need to be double-precision because Igor always calculates X values using double-precision regardless of the wave's data type.

Text Waves

Text waves are just like numeric waves except they contain bits of text rather than numbers. Like numeric waves, text waves can have one to four dimensions.

To create a text wave:

- Type anything but a number into the first unused cell of a table.
- Import data from a delimited text file that contains nonnumeric columns.
- Use the **Make** operation with the /T flag.

You can use the Make Waves dialog to generate text waves by choosing Text from the Type pop-up menu. Most often you will create text waves by entering text in a table. See **Using a Table to Create New Waves** on page II-195 for more information.

You can store anything in an element of a text wave. There is no length limit and there are no illegal characters. You can edit text waves in a table or assign values to the elements of a text wave using a command-line assignment statement.

You can use text waves in category plots, to automatically label individual data points in a graph (use markers mode and choose a text wave via the marker pop-up menu) and for storing notes in a table. Programmers may find that text waves are handy for storing a collection of diverse data, such as inputs to or outputs from a complex Igor procedure.

Here is how you can create and initialize text waves on the command line:

```
Make/T textwave= {"first element","2nd and last element"}
```

To see the text wave, create a table:

```
Edit textWave
```

Now you can try some wave assignments and see the result in the table:

```
textWave[2] = {"third element"}
textWave += "*"
textWave = "*" + textwave
```

Programmer Notes

Appending to a text wave is much faster than inserting or changing existing text. If you are going to replace all the text in an existing text wave it may be faster to kill the existing text by setting the number of points to zero using the command:

```
Redimension/N=0 textwave
```

You can then use the Redimension command again to set the number of points back to the desired value before storing new data.

In user-defined functions you can let the compiler know a wave will be text by using the /T flag in conjunction with the Wave keyword.

Complete List of Wave Properties

Here is a complete list of the properties that Igor stores for each wave.

Property	Comment
Name	Used to reference the wave from commands and dialogs. 1 to 31 characters. Standard names start with a letter. May contain letters, numbers or underscores. Liberal names may contain almost any character but must be enclosed in single quotes. See **Wave Names** on page II-80. The name is assigned when you create a wave. You can use the **Rename** operation (see page V-519) to change it.
Numeric type	Real or complex. Set when you create a wave. Use the **Redimension** operation (see page V-513) to change it.
Numeric precision	Defines the range of numbers that the wave can hold. Set when you create a wave. Use the **Redimension** operation (see page V-513) to change it.
Length	Number of data points in the wave. Also, size of each dimension for multidimensional waves. Set when you create a wave. Use the **Redimension** operation (see page V-513) to change it.
X scaling (x0 and dx)	Used to compute X values from point numbers. Also Y, Z and T scaling for multidimensional waves. The X value for point p is computed as X = x0 + p*dx. Set by **SetScale** operation (see page V-564).
X units	Used to auto-label axes. Also Y, Z and T units for multidimensional waves. Set by **SetScale** operation (see page V-564).
Data units	Used to auto-label axes. Set by **SetScale** operation (see page V-564).
Data full scale	For documentation purposes only. Not used. Set by **SetScale** operation (see page V-564).
Note	Holds arbitrary text related to wave. Set by **Note** operation (see page V-445) or via Browse Waves dialog. Readable via **note** function (see page V-445).
Dimension labels	Holds short (31 character) label for each dimension index and for each dimension. See **Dimension Labels** on page II-109.
Dependency Formula	Holds right-hand expression if wave is dependent. Set when you execute a dependency assignment using := or the **SetFormula** operation (see page V-559). Cleared when you do an assignment using plain =.
Creation date/time	Date & time when wave was created.
Modification date/time	Date & time when wave was last modified.

Property	Comment
Source folder	Identifies folder containing wave's source file, if any.
File name	Name of wave's source file, if any.

Chapter
II-6

Multidimensional Waves

Overview

Chapter II-5, **Waves**, concentrated on one-dimensional waves consisting of a number of rows. In Chapter II-5, **Waves**, the rows were referred to as "points" and the symbol p stood for row number, which was called "point number". Scaled row numbers were called X values and were represented by the symbol x.

This chapter now extends the concepts from Chapter II-5, **Waves**, to allow waves having up to four dimensions by adding the column, layer and chunk dimensions. The symbols q, r and s stand for column, layer and chunk numbers. Scaled column, layer and chunk numbers are called Y, Z and T values and are represented by the symbols y, z and t.

We call a two-dimensional wave a "matrix"; it consists of rows (the first dimension) and columns (the second dimension). After two dimensions the terminology becomes a bit arbitrary. We call the next two dimensions "layers" and "chunks".

Here is a summary of the terminology:

Dimension Number	0	1	2	3
Dimension Name	row	column	layer	chunk
Dimension Index	p	q	r	s
Scaled Dimension Index	x	y	z	t

Historical Note: Prior to Igor Pro 3.0, we used the term Y values to signify the values stored in a (one-dimensional) wave. We now call these "D values" or "data values" and use the term "Y" for the columns dimension. You may find that in places we still refer to "Y values" when we really mean data values.

Creating Multidimensional Waves

Multidimensional waves can be created using the following extension to the Make operation:

```
Make/N=(nrows,ncols,nlayers,nchunks) waveName
```

The Redimension operation has been extended in the same way.

Examples:

```
Make/N=20 wave1
```

Makes a conventional (1D) wave with 20 points (rows).

```
Make/N=(20,3) wave2d
```

Makes a matrix (2D) wave with 20 rows and 3 columns for a total of 60 points.

```
Redimension/N=(10,4) wave1,wave2d
```

Changes both wave1 and wave2d so they have 10 rows and 4 columns.

The operations InsertPoints and DeletePoints take a flag (/M=*dimensionNumber*) to specify the dimension into which points are inserted. For example:

```
InsertPoints/M=1 2,5,wave2d   //M=1 means column dimension
```

inserts 5 new columns in front of column number 2. If the "/M=1" had been omitted or if /M=0 had been used then 5 new rows would have been inserted in front of row number 2.

You can also create multidimensional waves using the Make operation with a list of data values. For example, while

```
Make wave1= {1,2,3}
```

creates a conventional 1D wave containing a single column of 3 rows,

```
Make wave2= {{1,2,3},{4,5,6}}
```

creates a 2D wave containing 2 columns by 3 rows of data.

The Duplicate operation can create an exact copy of a multidimensional wave or, using the /R flag, extract a subrange. Here is the syntax of the /R flag:

```
Duplicate/R=[startRow,endRow][startCol,endCol] and so on...
```

You can use the character * for any end field to specify the last element in the given dimension or you can just omit the end field. You can also specify just [] to include all of a given dimension. If the source wave has more dimensions than you specify in the /R flag, then all of those dimensions are copied. Examples:

```
Make/N=(5,4,3) wave3d= p+10*q+100*r
Duplicate/R=[1,2][2,*] wave3d,wave3d1
```

duplicates rows 1 through 2, columns 2 through the end and all layers.

```
Duplicate/R=[][2,2][0,0] wave3d,wave3d2
```

creates a 3D wave consisting of all rows of column 2 layer 0 and containing 1 column and 1 layer. Igor considers wave3d2 to be a 3 dimensional wave and not a 1 dimensional column vector because the column and layer dimension numbers are not zero. This is a subtle distinction and can cause confusion. For example, you may think you have extracted a 1D wave from a 3D object but you will find that wave3d2 will not show up in the new graph dialog or other places where 1D vectors are required.

You can turn the 3D wave wave3d2 into a 1D wave using the following command:

```
Redimension/N=(-1,0) wave3d2
```

The -1 value does not to change the number of rows whereas the 0 value for the number of columns indicates that there are no dimensions past rows (in other words, no columns, layers or chunks).

Programmer Notes

For historical reasons, you can treat the symbols x and p like global variables, meaning that you can store into them as well as retrieve their values by referencing them. Unlike x and p, y, z, t, q, r and s act like functions and you can't store into them.

The command "SetScale d" sets the data full scale and data units for a wave. Prior to Igor Pro 3.0, we used "SetScale y" for this purpose. With the extension of Igor to multiple dimensions, "SetScale y" was needed for setting the column dimension scaling and units. For backward compatibility "SetScale y" acts like "SetScale d" when used on a 1D wave.

Here are some functions and operations that are useful in programming with multidimensional waves:

```
DimOffset, DimDelta, DimSize
FindDimLabel, SetDimLabel, GetDimLabel
```

Dimension Labels

A dimension label is a name associated with a dimension or with an index into an element of a dimension. Dimension labels are primarily an aid to the Igor procedure programmer when dealing with waves in which certain elements have distinct purposes. Dimension labels can be set when loading from a file, and can be displayed, created or edited in a table (see **Showing Dimension Labels** on page II-191).

You can give names to individual dimension indices in multidimensional or 1D waves. For example, if you have a 3 column wave, you can give column 0 the name "red", column 1 the name "green" and column 2 the name "blue". You can use the names in wave assignments in place of literal numbers. To do so, you use the % symbol in front of the name like so:

```
my3dwave[][%red]=my3dwave[p][%green]   //Set red col equal to green col
```

To create a label for a given index of a given dimension, use the SetDimLabel operation.

For example:

```
SetDimLabel 1,0,red,my3dwave
```

The 1 is the dimension number (columns), 0 is the dimension index (column #0) and red is the label.

The function GetDimLabel returns a string containing the name associated with a given dimension and index. For example:

```
Print GetDimLabel(my3dwave,1,0)
```

prints “red” into the history area.

The FindDimLabel function returns the index value associated with the given label. It returns the special value -2 if the label is not found. This function is useful in user-defined functions so that you can use a numeric index instead of a dimension label when accessing a wave in a loop. Accessing wave data using a numeric index is faster than using a dimension label.

In addition to setting the name for individual dimension index values, you can set the name for an entire dimension by using an index value of -1. For example:

```
SetDimLabel 1,-1,colors,my3dwave
```

Dimension names can contain up to 31 characters and may contain spaces and other normally illegal characters if you surround the name in single quotes or if you use the $ operator to convert a string expression to a name. For example:

```
wavename[%'a name with spaces']
wavename[%$"a name with spaces"]
```

Dimension names have the same characteristics as object names. See **Object Names** on page III-415 for a discussion of object names in general.

Graphing Multidimensional Waves

You can easily view two-dimensional waves as images and as contour plots using Igor’s built-in operations. See Chapter II-14, **Contour Plots**, and Chapter II-15, **Image Plots**, for further information about these types of graphs. You can also create waterfall plots where each column in the matrix wave corresponds to a separate trace in the waterfall plot. For more details, see **Waterfall Plots** on page II-296.

Additional facilities for displaying multi-dimensional waves in Igor Pro are provided by the Gizmo extension, which create surface plots, slices through volumes and many other 3D plots. To get started with Gizmo, choose Windows→New→3D Plots→3D Help.

It is possible to graph a subset of a wave, including graphing rows or columns from a multidimensional wave. The New Graph dialog supports graphing subsets, and allows selection of 2D waves if the More Choices button is clicked. See **Subrange Display** on page II-288 for more information.

Analysis on Multidimensional Waves

Igor Pro includes the following capabilities for analysis of multidimensional data:

- Multidimensional waveform arithmetic
- Matrix math operations
- Image processing
- Multidimensional Fast Fourier Transform

There are many analysis operations for 1D data that we have not yet extended to support multiple dimensions. Multidimensional waves will not appear in dialogs for these operations. If you invoke them on multidimensional waves from the command line or from an Igor procedure, Igor will treat the multidimensional waves as if they were 1D. For example, the Smooth operation will treat a 2D wave consisting of n rows and m columns as if it were a 1D wave with n*m rows. In some cases the operation will be useful. In other cases, it will make no sense.

Multidimensional Wave Indexing

You can use multidimensional waves in wave expressions and assignment statements just as you do with 1D waves (see **Indexing and Subranges** on page II-95). To specify a particular point in a wave, use the syntax:

```
wavename[rowIndex][columnIndex][layerIndex][chunkIndex]
```

Similarly, to specify a point using *scaled* dimension indices, use the syntax:

```
wavename(xIndex)(yIndex)(zIndex)(tIndex)
```

rowIndex is the number, starting from zero, of the row of interest. It is an unscaled index. *xIndex* is simply the row index, offset and scaled by the wave's X scaling property, which you set using the SetScale operation (Change Wave Scaling in Data menu). Using scaled indices you can access the wave's data using its natural units. You can use unscaled or scaled indices, whichever is more convenient. column/y, layer/z and chunk/t indices are analogous to row/x indices.

Using [] notation specifies that the index that you are supplying is an unscaled dimension index. Using () notation specifies that you are supplying a scaled dimension index. You can even mix the [] notation with () notation.

Here are some examples:

```
Make/N=(5,4,3) wave3d= p+10*q+100*r
SetScale/I x,0,1,"" wave3d
SetScale/I y,-1,1,"" wave3d
SetScale/I z,10,20,"" wave3d
Print wave3d[0][1][2]
Print wave3d(0.5)[2](15)
```

The first Print command prints 210, the value in row 0, column 1 and layer 2. The second Print command prints 122, the value in row 2 (where x=0.5), column 2 and layer 1 (where z=15).

Since wave3D has three dimensions, we do not (and must not) specify a chunk (4th dimension) index.

There is one important difference between wave access using 1D waves versus multidimensional waves. For 1D waves alone, Igor performs linear interpolation when the specified index value (scaled or unscaled) falls between two points. For multidimensional waves, Igor returns the value of the element whose indices are closest to the specified indices.

When a multidimensional wave is the destination of a wave assignment statement, you can specify a subrange for each dimension. You can specify an entire dimension by using []. For example:

```
wave3d[2][][1,2]= 3
```

sets row 2 of all columns and layers 1 and 2 to the value 3.

Note that indexing of the form [] (entire dimension) or [1,2] (range of a dimension) can be used on the left hand side only. This is because the indexing on the left side determines which elements of the destination are to be set whereas indexing on the right side identifies a particular element in the source which is to contribute to a particular value in the destination.

Multidimensional Wave Assignment

As with one-dimensional waves (waveform data), you can assign a value to a multidimensional wave using a wave assignment statement.

```
Make/O/N=(3,3) wave0_2D, wave1_2D, wave2_2D
wave1_2D = 1.0; wave2_2D = 2.0
wave0_2D = wave1_2D / wave2_2D
```

The last command sets all elements of wave0_2D equal to the quotient of the corresponding elements of wave1_2D and wave2_2D.

Important: Wave assignments as shown in the above example where waves on the right hand side do not include explicit indexing are defined only when all waves involved have the same dimensionality. The result of the following assignment is undefined and may produce surprising results.

```
Make/O/N=(3,3) wave33
Make/O/N=(2,2) wave22
…
wave33= wave22
```

Whenever waves of mismatched dimensionality are used you should specify explicit indexing as described here.

In a wave assignment, Igor evaluates the right hand side one time for each element specified by the left hand side. During this evaluation, the symbols p, q, r and s take on the value of the row, column, layer and chunk, respectively, of the element in the destination for which a value is being calculated. So,

```
Make/O/N=(5,4,3) wave3D = 0
Make/O/N=(5,4) wave2d = 999
wave3D[][][0] = wave2D[p][q]
```

stores the contents of wave2D in layer 0 of wave3D. In this case, the destination (wave3D) has three dimensions, so p, q and r are defined and s is undefined. The following discussion explains this assignment and presents a way of thinking about wave assignments in general.

The left hand side of the assignment specifies that Igor is to store a value into all rows (the first []) and all columns (the second []) of layer zero (the [0]) of wave3D. For each of these elements, Igor will evaluate the right hand side. During the evaluation, the symbol p will return the row number of the element in wave3D that Igor is about to set and the symbol q will return the column number. The symbol r will have the value 0 during the entire process. Thus, the expression wave2D[p][q] will return a value from wave2D at the corresponding row and column in wave3D.

As the preceding example shows, wave assignments provide a way of transferring data between waves. With the proper indexing, you can build a 2D wave from multiple 1D waves or a 3D wave from multiple 2D waves. Conversely, you can extract a layer of a 3D wave into a 2D wave or extract a column from a 2D wave into a 1D wave. Here are some examples that illustrate these operations.

```
// Build a 2D wave from multiple 1D waves (waveforms)
Make/O/N=5, wave0=p, wave1=p+1, wave2=p+2    // 1D waveforms
Make/O/N=(5,3) wave0_2D
wave0_2D[][0] = wave0[p]      // Store into all rows, column 0
wave0_2D[][1] = wave1[p]      // Store into all rows, column 1
wave0_2D[][2] = wave2[p]      // Store into all rows, column 2

// Build a 3D wave from multiple 2D waves
Duplicate/O wave0_2D, wave1_2D; wave1_2D *= -1
Make/O/N=(5,3,2) wave0_3D
wave0_3D[][][0]= wave0_2D[p][q]  // Store into all rows/cols, layer 0
wave0_3D[][][1]= wave1_2D[p][q]  // Store into all rows/cols, layer 1

// Extract a layer of a 3D wave into a 2D wave
wave0_2D = wave0_3D[p][q][0]     // Extract layer 0 into 2D wave

// Extract a column of a 2D wave into a 1D wave
wave0 = wave0_2D[p][0]           // Extract column 0 into 1D wave
```

To understand assignments like these, first figure out, by looking at the indexing on the left hand side, which elements of the destination wave are going to be set. (If there is no indexing on the left then all elements are going to be set.) Then think about the range of values that p, q, r and s will take on as Igor evaluates the right hand side to get a value for each destination element. Finally, think about how these values, used as indices on the right hand side, select the desired source element.

To create such an assignment, first determine the indexing needed on the left hand side to set the elements of the destination that you want to set. Then think about the values that p, q, r and s will take on. Then use p, q, r and s as indices to select a source element to be used when computing a particular destination element.

Here are some more examples:

```
// Extract a ROW of a 2D wave into a 1D wave
Make/O/N=3 row1
row1 = wave0_2D[1][p]          // Extract row 1 of the 2D wave
```

In this example, the *row* index (p) for the destination is used to select the source *column* while the source row is always 1.

```
// Extract a horizontal slice of a 3D wave into a 2D wave
Make/O/N=(2,3) slice_R2        // Slice consisting of all of row 2
slice_R2 = wave0_3D[2][q][p]   // Extract row 2, all columns/layers
```

In this example, the row data for slice_R2 comes from the layers of wave0_3D because the p symbol (row index) is used to select the layer in the source. The column data for slice_R2 comes from the columns of wave0_3D because the q symbol (column index) is used to select the column in the source. All data comes from row 2 in the source because the row index is fixed at 2.

You can store into a range of elements in a particular dimension by using a range index on the left hand side. As an example, here are some commands that shift the horizontal slices of wave0_3D.

```
Duplicate/O wave0_3D, tmp_wave0_3D
wave0_3D[0][][] = tmp_wave0_3D[4][q][r]
wave0_3D[1,4][][] = tmp_wave0_3D[p-1][q][r]
KillWaves tmp_wave0_3D
```

The first assignment transfers the slice consisting of all elements in row 4 to row zero. The second assignment transfers slice n-1 to slice n. To understand this, realize that as p goes from 1 to 4, p-1 indexes into the preceding row of the source.

Vector (Waveform) to Matrix Conversion

Occasionally you will may need to convert between a vector form of data and a matrix form of the same data values. For example, you may have a vector of 16 data values stored in a waveform named sixteenVals that you want to treat as a matrix of 8 rows and 2 columns.

Though the Redimension operation normally doesn't move data from one dimension to another, in the special case of converting to or from a 1D wave Redimension will leave the data in place while changing the dimensionality of the wave. You can use the command:

```
Make/O/N=16 sixteenVals          // 1D
Redimension/N=(8,2) sixteenVals  // now 2D, no data lost
```

to accomplish the conversion. When redimensioning from a 1D wave, columns are filled first, then layers, followed by chunks. Redimensioning from a multidimensional wave to a 1D wave doesn't lose data, either.

Matrix to Matrix Conversion

To convert a matrix from one matrix form to another, don't directly redimension it to the desired form. For instance, if you have a 6x6 matrix wave, and you would like it to be 3x12, you might try:

```
Make/O/N=(6,6) thirtySixVals        //2D
Redimension/N=(3,12) thirtySixVals  //this loses the last three rows
```

but Igor will first shrink the number of rows to 3, discarding the data for the last three rows, and then add 6 columns of zeroes.

The simplest way to work around this is to convert the matrix to a 1D vector, and then convert it to the new matrix form:

```
Make/O/N=(6,6) thirtySixVals        // 2D
Redimension/N=36 thirtySixVals      // 1D vector preserves the data
Redimension/N=(3,12) thirtySixVals  // data preserved
```

Multidimensional Fourier Transform

Igor's FFT and IFFT routines are mixed-radix and multidimensional. Mixed-radix means you do not need a power of two number of data points (or dimension size). There is only one restriction on the dimensions of a wave: when performing a forward FFT on real data, the number of rows must be even. Note, however, that if a given dimension size is a prime number or contains a large prime in its factorization, the speed will be reduced to that of a normal Discrete Fourier Transform (i.e., the number of operations will be on the order of N^2 rather than N•log(N)). For more information about the FFT, see **Fourier Transforms** on page III-235 and the **FFT** operation on page V-156.

Chapter
II-7

Numeric and String Variables

Overview

This chapter discusses the properties and uses of global numeric and string variables. For the fine points of programming with global variables, see **Accessing Global Variables and Waves** on page IV-50.

Numeric variables are double precision floating point and can be real or complex. String variables can hold an arbitrary number of characters. Igor stores all global variables when you save an experiment and restores them when you reopen the experiment.

Numeric variables or numeric expressions containing numeric variables can be used in any place where literal numbers are appropriate including as operands in assignment statements and as parameters to operations, functions or macros (but require parentheses in operation flags, see **Reference Syntax Guide** on page V-13).

String variables or string expressions can be used in any place where strings are appropriate. String variables can also be used as parameters where Igor expects to find the name of an object such as a wave, variable, graph, table or page layout. For details on this see **Converting a String into a Reference Using $** on page IV-47.

Creating Global Variables

There are 20 built-in numeric variables (K0 … K19), called system variables, that exist all the time. Igor uses these mainly to return results from the CurveFit operation. All other variables are user variables. User variables can be created in one of two ways:

- Automatically in the course of certain operations.
- Explicitly by the user, via the Variable/G and String/G operations.

When you create a variable directly from the command line using the Variable or String operation, it is always global and you can omit the /G flag. You need /G in Igor procedures to make variables global. The /G flag has a secondary effect — it permits you to overwrite existing global variables.

Uses For Global Variables

Global variables have two properties that make them useful: globalness and persistence. Since they are global, they can be accessed from any procedure. This provides an easy way to communicate values from one procedure to another. Since they are persistent, you can use them to store settings over time.

Variable Names

Variable names consist of 1 to 31 characters. The first character must be alphabetic. The remaining characters can be alphabetic, numeric or the underscore character. Variable names must not conflict with the names of other Igor objects, functions or operations. Names in Igor are case insensitive. You can rename a variable using the Rename operation, or the Rename Objects dialog in the Misc menu. See **Object Names** on page III-415 for more information.

System Variables

System variables are built in to Igor. They are mainly provided for compatibility with older versions of Igor but are sometimes useful as “scratch” variables. You can see a list of system variables and their values by choosing the Object Status item in the Misc menu.

There are 20 system variables named K0,K1...K19 and one named veclen. The K variables are used by the curve fitting operations but are otherwise free for your use.

The `veclen` variable is present for compatibility reasons. In previous versions of Igor, it contained the default number of points for waves created by the Make operation. This is no longer the case. Make will always create waves with 128 points unless you explicitly specify otherwise using the /N=(<number of points>) flag.

Although the CurveFit operation stores results in the K variables, it does so only for compatibility reasons and it also creates user variables and waves to store the same results.

However, the CurveFit operation does use system variables for the purpose of setting up initial parameter guesses if you specify manual guess mode. You can also use a wave for this purpose if you use the kwCWave keyword. See the **CurveFit** operation on page V-85.

It is best to not rely on system variables unless necessary. Since Igor writes to them at various times, they may change when you don't expect it.

The Data Browser does not display system variables since this tends to obscure the (usually more interesting) user variables.

Note: System variables are stored on disk as single precision values so that they can be read by older versions of Igor. Thus, you should store values that you want to keep indefinitely in your own global variables.

User Variables

You can create your own global variables by using the `Variable/G` (see **Numeric Variables** on page II-117) and `String/G` operations (see **String Variables** on page II-118). Variables that you create are called "user variables" whether they be numeric or string. You can browse the global user variables by choosing the Object Status item in the Misc menu. You can also use the Data Browser window (Data menu) to view your variables.

Global user variables are mainly used to contain persistent settings used by your procedures. They are also sometimes used to pass results from a macro to the macro that called it.

Special User Variables

In the course of some operations, Igor automatically creates special user variables. For example, the curve fitting operation creates the user variable `V_chisq` and others to store various results generated by the curve fit. The names of these variables always start with the characters "`V_`" for numeric variables or "S_" for string variables. The meaning of these variables is documented along with the operations that generate them in Chapter V-1, **Igor Reference**.

In addition, Igor sometimes checks for `V_` variables that you can create to modify the default operation of certain routines. For example, if you create a variable with the name V_FitOptions, Igor will use that to control the CurveFit, FuncFit and FuncFitMD operations. The use of these variables is documented along with the operations that they affect.

When used inside interpreted procedures (defined using Proc or Macro), V_ and S_ variables are created as local variables. When used inside compiled procedures (defined using Function), such variables *can* be local (but might be global under certain circumstances). See **Accessing Variables Used by Igor Operations** on page IV-103 for details.

Numeric Variables

You create numeric user variables by using the `Variable` command from the command line or in a procedure. The syntax for the Variable command is:

`Variable [`*`flags`*`] `*`varName`*` [=`*`numExpr`*`] [,`*`varName`*` [=`*`numExpr`*`]]...`

There are three optional *flags* parameters:

/C specifies complex variable.
/D obsolete. Used in previous versions to specify double precision (now all variables are double precision).
/G specifies variable is to be global and overwrites any existing variable.

The variable is initialized when it is created if you supply the initial value with a numeric expression using `=`*`numExpr`*. If you create a numeric variable and specify no initializer, it is initialized to zero.

You can create more than one variable at a time by separating the names and optional initializers for multiple variables with a comma.

If used in a procedure, the new variable is local to that procedure unless the /G (global) flag is used. If used on the command line, the new variable is always global.

Here is an example of a variable creation with initialization:

```
Variable v1=1.1, v2=2.2, v3=3.3*sin(v2)/exp(v1)
```

Since the /C flag was not specified, the data type of v1, v2 and v3 is double precision real.

Since the /G flag was not specified, these variables would be global if you invoked the `Variable` operation directly from the command line or local if you invoked it in a procedure.

`Variable/G varname` can be invoked whether or not a variable of the specified name already exists. If it does exist as a variable, its contents are not altered by the operation unless the operation includes an initial value for the variable.

To assign a value to a complex variable, use the `cmplx()` function:

```
Variable/C cv1 = cmplx(1,2)
```

You can kill (delete) a global user variable using the Data Browser or the `KillVariables` operation. The syntax is:

```
KillVariables [flags] [variableName [,variableName]...]
```

There are two optional *flags*:

/A kills all global variables in the current data folder. If you use /A, omit *variableName*.

/Z doesn't generate an error if a global variable to be killed does not exist. To kill all global variables in the current data folder, use KillVariables/A/Z.

For example, to kill global variable `cv1` without worrying about whether it was previously defined, use the command:

```
KillVariables/Z cv1
```

Killing a variable reduces clutter and saves a bit of memory. You can not kill a system variable or local variable.

String Variables

You create user string variables by using a `String` declaration on the command line or in a procedure. The syntax is:

```
String [/G] strName [=strExpr] [,strName [=strExpr]... ]
```

The optional /G flag specifies that the string is to be global, and it overwrites any existing string variable.

The string variable is initialized when it is created if you supply the initial value with a string expression using =*strExpr*. If you create a string variable and specify no initializer it is initialized to the empty string ("").

You can create more than one string variable at a time by separating the names and optional initializers for multiple string variables with a comma.

If used in a procedure, the new string is local to that procedure unless the /G (global) flag is used. If used on the command line, the new string is always global.

Here is an example of a variable creation with initialization:

```
String str1 = "This is string 1", str2 = "This is string 2"
```

Since /G was not used, these strings would be global if you invoked String directly from the command line or local if you invoked it in a procedure.

`String/G strName` can be invoked whether or not a variable of the given name already exists. If it does exist as a string, its contents are not altered by the operation unless the operation includes an initial value for the string.

You can kill (delete) a global string using the Data Browser or the `KillStrings` operation. The syntax is:

```
KillStrings [flags] [stringName [,stringName ]...]
```

There are two optional *flags*:

/A kill all global strings in the current data folder. If you use /A, omit *stringName*.

/Z doesn't generate an error if a global string to be killed does not exist. To kill all global strings in the current data folder, use KillStrings/A/Z.

For example, to kill global string `myGlobalString` without worrying about whether it was previously defined, use the command:

```
KillStrings/Z myGlobalString
```

Killing a string reduces clutter and saves a bit of memory. You can not kill a local string.

There are a number of functions that return or operate on string expressions. See **Strings** on page V-10 for a list. There are also a number of ways to manipulate string variables. See **Strings** on page IV-12.

Local and Parameter Variables in Procedures

You can create variables in macros and user defined functions as parameters or local variables. These variables exist only while the macro or function is running. They can not be accessed from outside the macro or function and do not retain their values from one invocation of the macro or function to the next. See **Local Versus Global Variables** on page IV-46 for more information.

Chapter
II-8

Data Folders

Overview

Using data folders, you can store your data within an experiment in a hierarchical manner. Hierarchical storage is useful when you have multiple sets of similar data. By storing each set in its own data folder, you can organize your data in a meaningful way and also avoid name conflicts.

Data folders contain four kinds of **data objects**:

- Waves
- Numeric variables
- String variables
- Other data folders

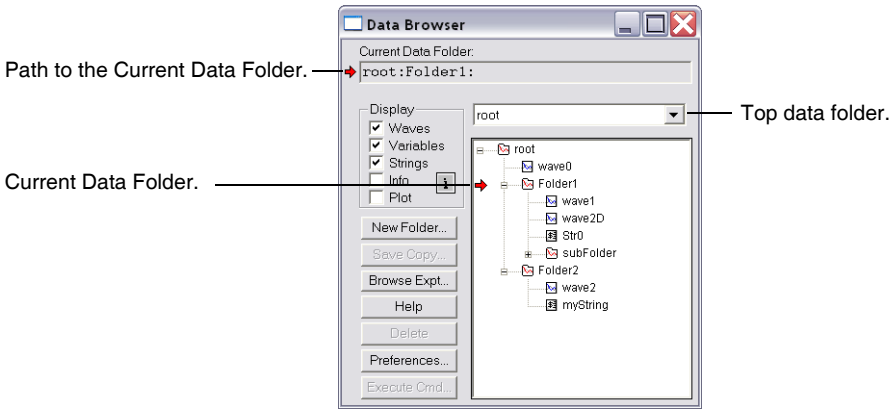
Igor's data folders are very similar to a computer's hierarchical disk file system except they reside wholly in memory and not on disk. This similarity can help you understand the concept of data folders but you should take care not to confuse them with the computer's folders and files.

Data folders are particularly useful when you conduct several runs of an experiment. You can store the data for each run in a separate data folder. The data folders can have names like "run1", "run2", etc., but the names of the waves and variables in each data folder can be the same as in the others. In other words, the information about which run the data objects belong to is encoded in the data folder name, allowing the data objects themselves to have the same names for all runs. You can write procedures that use the same wave and variable names regardless of which run they are working on.

Data folders are very handy for programmers who need to create temporary waves during a procedure. You can create a temporary data folder with a name designed not to conflict with any other Igor object names and then create waves without having to worry about conflict. When done, you can kill the data folder and everything it contains with a single command rather than having to kill waves, variables, and strings. You can also use a data folder to store persistent waves and global variables not intended to be seen by the end user. As a programmer, you can use nested data folders as a sort of data structure.

All operations in Igor Pro are data-folder aware. On the command line you can specify waves from several different data folders within one command.

You can use the Data Browser window (Data menu) to see the data folder hierarchy and to set the current data folder:



You can use the Data Browser not only to see the hierarchy and set the current data folder but also to:

- Create new data folders.
- Move, duplicate, rename and delete objects.
- Browse other Igor experiment files and load data from them into memory.
- Save a copy of data in the current experiment to an experiment file or folder on disk.
- See and edit the contents of variables, strings or waves in the information pane by selecting an object

- See a simple plot of 1D or 2D waves by selecting one wave at a time in the main list while the Plot pane is visible.
- See a simple plot of a wave while browsing other Igor experiments.
- See variable, string and wave contents by double-clicking their icons.
- See a simple histogram or wave statistics for one wave at a time.

Before using data folders, be sure to read **Using Data Folders** on page II-127, and **Problems with Data Folders** on page II-130.

Programmers should read **Programming with Data Folders** on page IV-148.

A similar browser is used for wave selection in dialogs. For details see **Dialog Wave Browser** on page II-181.

Data Folder Syntax

Data folders are named objects like other Igor objects such as waves and variables. Data folder names follow the same rules as wave names. See **Liberal Object Names** on page III-415.

Like the Macintosh file system, Igor Pro's data folders use the colon character (:) to separate components of a path to an object. This is analogous to Unix which uses / and Windows which uses \. (**Reminder**: Igor's data folders exist wholly in memory while an experiment is open. It is not a disk file system!)

A data folder named "root" always exists and contains all other data folders.

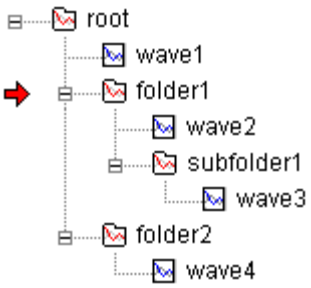
A given object can be specified in a command using:

- A full path
- A partial path
- Just the object name

The object name alone can only be used when the current data folder contains the object.

A full path starts with "root" and does not depend on the current data folder. A partial path starts with ":" and is relative to the current data folder.

Assume the data folder structure shown below, where the arrow indicates that folder1 is the current data folder.



Each of the following commands creates a graph of one of the waves in this hierarchy:

```
Display wave2
Display :subfolder1:wave3
Display root:folder1:subfolder1:wave3
Display ::folder2:wave4
```

The last example illustrates the rule that you can use multiple colons to walk back up the hierarchy: from folder1 (the current data folder), up one level, then down to folder2 and wave4. Here is another valid (but silly) example:

```
Display root:folder1:subfolder1:::folder2:wave4
```

Occasionally you will need to specify a data folder itself rather than an object in a data folder. In that case, just leave off the object name. The path specification should therefore have a trailing colon. However, Igor will generally understand what you mean if you forget the trailing colon.

If you need to specify the current data folder, you can use just a single colon. For example:

```
KillDataFolder :
```

kills the current data folder (and all its contents) and then sets the current data folder to the parent of the current. Nonprogrammers might prefer to use the Data Browser to delete data folders.

Recall that the $ operator converts a string expression into a single name. Since data folders are named, the following is valid:

```
String df1 = "folder1", df2="subfolder1"
Display root:$(df1):$(df2):wave3
```

This is a silly example but the technique would be useful if df1 and df2 were parameters to a procedure.

Note that parentheses must be used in this type of statement. That is a result of the precedence of $ relative to :.

When used at the beginning of a path, the $ operator works in a special way and can (and must) be used on the entire path:

```
String path1 = "root:folder1:subfolder1:wave3"
Display $path1
```

When liberal names are used within a path, they must be in single quotes. For example:

```
Display root:folder1:'subfolder 1':'wave 3'
String path1 = "root:folder1:'subfolder 1':'wave 3'"
Display $path1
```

However, when a simple name is passed in a string, single quotes must not be used:

```
Make 'wave 1'
String name
name = "'wave 1'"       // Wrong.
name = "wave 1"         // Correct.
Display $name
```

Data Folder Operations and Functions

Most people will use the Data Browser (Data menu) to create, view and manipulate data folders. The following operations will be mainly used by programmers, who should read **Programming with Data Folders** on page IV-148.

NewDataFolder *path*
SetDataFolder *path*
KillDataFolder *path*
DuplicateDataFolder *srcPath*, *destPath*
MoveDataFolder *srcPath*, *destPath*
MoveString *srcPath*, *destPath*
MoveVariable *srcPath*, *destPath*
MoveWave *wave*, *destPath* [*newname*]
RenameDataFolder *path*, *newName*
Dir

The following are functions that are used with data folders.

GetDataFolder(*mode*)
CountObjects(*pathStr*,*type*)
GetIndexedObjName(*pathStr*,*type*,*index*)
GetWavesDataFolder(*wave*,*mode*)

DataFolderExists(*pathStr*)
DataFolderDir(*mode*)

Data Folders Reference Functions

As of Igor Pro 6.1, function programmers can utilize data folder references in place of paths. Data folder references are lightweight objects that refer directly to a data folder whereas a path, consisting of a sequence of names, has to be looked up in order to find the actual target folder.

Here are functions that work with data folder references:

GetDataFolderDFR()
GetIndexedObjNameDFR(*dfr*, *type*, *index*)
GetWavesDataFolderDFR(*wave*)
CountObjectsDFR(*dfr*, *type*)
DataFolderRefStatus(*dfr*)
NewFreeDataFolder()
DataFolderRefsEqual(*dfr1*, *dfr2*)

For information on programming with data folder references, see **Data Folder References** on page IV-61.

Data Folders and Commands

Igor normally evaluates commands in the context of the current data folder. This means that, unless qualified with a path to a particular data folder, object names refer to objects in the *current* data folder. For example:

```
Macro MyMacro()
   Make wave1
   Variable/G myGlobalVariable
EndMacro
```

creates wave1 and myGlobalVariable in the current data folder. Likewise executing:

```
WaveStats wave1
```

creates WaveStats output variables (V_avg, etc.) in the current data folder.

Data Folders and User-Defined Functions

You must exercise some care when accessing global variables from "data-folder ignorant" user-defined functions. See **Accessing Global Variables and Waves** on page IV-50 for details.

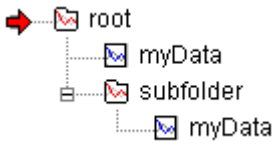
Data Folders and Window Macros

Window macros are evaluated in the context of the *root* data folder. Window macros begin with the Window keyword, as in the example below. Macros that begin with the "Macro" or Proc keywords evaluate their commands in the context of the *current* data folder.

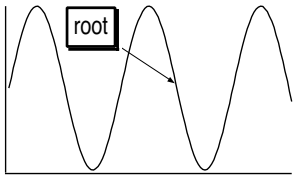
Evaluating window macros this way ensures that a window is recreated correctly regardless of the current data folder, and provides some compatibility with window macros created with prior versions of Igor Pro which didn't have data folders.

This means that object names within window macros or functions that don't explicitly contain a data folder path refer to objects in the root data folder. This is important when the current data folder is not the root data folder.

For example, given identically named waves organized as follows:

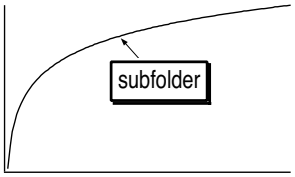


A window recreation macro for a graph of root:myData (whose tag shows the wave's data folder) will resemble the following:




```
Window Graph0() : Graph
   PauseUpdate; Silent 1   // building window...
   Display myData          // note: no data folder specified
   Tag/N=text0/X=21.15/Y=35.00 myData, 50
   AppendText/N=text0 "\\{\"%s\",GetWavesDataFolder(TagWaveRef(),0)}"
EndMacro
```

Observe that myData is referred to in the Display command without its data folder (root:myData would be the fully qualified name of the wave object). If you change the current data folder to the subfolder and run the window macro, the resulting graph will be identical because the myData wave in the root data folder would be graphed.




```
Window Graph1() : Graph
   PauseUpdate; Silent 1   // building window...
   String fldrSav= GetDataFolder(1)
   SetDataFolder root:subfolder: // note: data folder is specified
   Display myData
   SetDataFolder fldrSav
   Tag/N=text0/X=21.15/Y=35.00 myData, 50
   AppendText/N=text0 "\\{\"%s\",GetWavesDataFolder(TagWaveRef(),0)}"
EndMacro
```

Data Folders and Assignment Statements

Wave and variable assignment statements are evaluated in the context of the data folder containing the wave or variable on the left-hand side of the statement:

```
root:subfolder:wave0 = wave1 + var1
```

is a shorter way of writing the equivalent:

```
root:subfolder:wave0 = root:subfolder:wave1 + root:subfolder:var1
```

This rule also applies to dependency formulae which use := instead of = as the assignment operator.

Data Folders and Controls

ValDisplay controls evaluate their value expression in the context of the root data folder.

SetVariable controls remember the data folder in which the controlled global variable exists, and continue to function properly when the current data folder is different than the controlled variable.

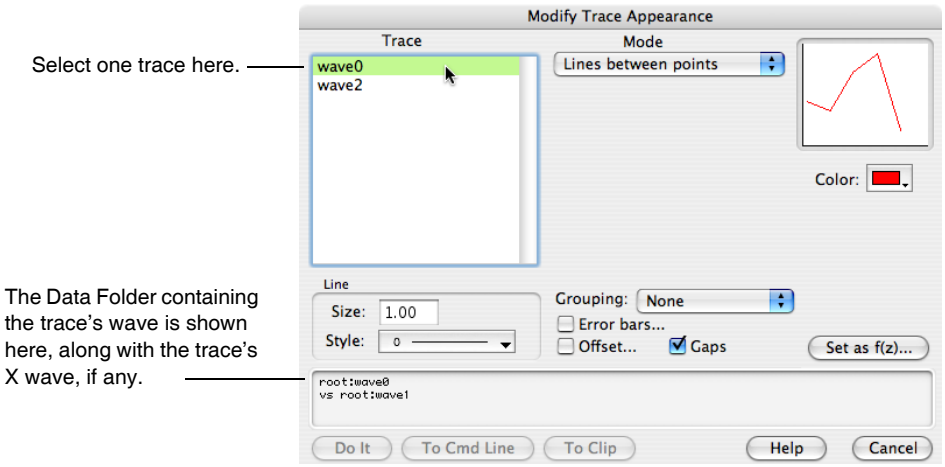
Note: The system variables (K0 through K19) belong to no particular data folder (they are available from any data folder), and there is only *one* copy of these variables. If you create a SetVariable controlling K0 while the current data folder is "aFolder", and another SetVariable controlling K0 while the current data folder is "bFolder", *they are actually controlling the same K0*.

See Chapter III-14, **Controls and Control Panels**, for details about controls.

Data Folders and Traces

You cannot tell by looking at a trace in a graph which data folder it resides in. You could save and examine the graph window recreation macro. The easiest way to find out what data folder a trace's wave resides in is to use the trace info help. On Macintosh, press Command-Option-Control and click on the trace in the graph window. On Windows, press Shift+F1 to summon context-sensitive help and then click on the trace to get trace info.

Another method is to use the Modify Trace Appearance dialog. When you press and hold down the mouse button on a trace in the dialog's Trace list, Igor displays data folder (and X wave) information where the commands are usually shown:



Using Data Folders

You can use data folders for many purposes, just like you use the folders on your hard disk for organizing files in many different ways. Here are two common uses of data folders.

Hiding Waves, Strings, and Variables

Sophisticated Igor procedures may need a large number of global variables, strings and waves that aren't intended to be directly accessed by the user. The programmer who creates these procedures should keep all such items within data folders they create with unique names designed not to conflict with other data folder names.

Users of these procedures should leave the current data folder set to the data folder where their raw data and final results are kept, so that the procedure's globals and waves won't clutter up the dialog lists.

Programmers creating procedures should read **Programming with Data Folders** on page IV-148.

Separating Similar Data

One situation that arises during repeated testing is needing to keep the data from each test "run" separate from the others. Often the data from each run is very similar to the other runs, and may even have the same name. Without data folders you would need to choose new names after the first run.

By making one data folder for each test run, you can put all of the related data for one run into each folder. The data can use identical names, because other identically named data is in different data folders.

Using data folders also keeps the data from various runs from being accidently combined, since only the data in the current data folder shows up in the various dialogs or can be used in a command without a data folder name.

The Wavemetrics-supplied "Multi-peak Fitting" example experiment's procedures work this way: they create data folders to hold separate peak curve fit runs and global state information.

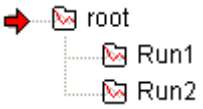
Using Data Folders Example

This example will use data folders to:

- load data from two test runs into separate data folders
- create graphs showing each test run by itself
- create a graph comparing the two test runs

First we'll use the Data Browser to create a data folder for each test run.

Open the Data Browser (in the Data menu), and set the Current Data Folder to root. Click the root data folder, and click the New Folder button. Enter "Run1" for the new data folder's name. Click New Folder again and enter "Run2". The Data Browser window should resemble the one shown here.

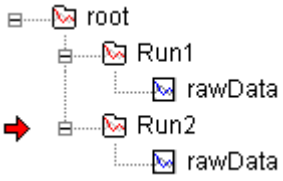


Now let's load sample data into each data folder, starting with Run1.

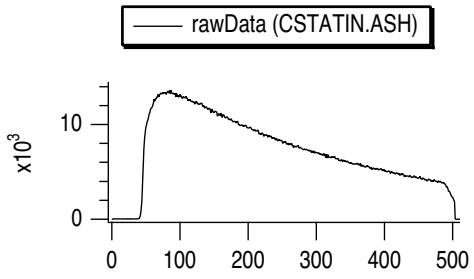
Set the Current Data Folder to Run1, then choose Load Delimited Text from the Data menu's Load Data submenu. Select the CSTATIN.ASH file from the Sample Data subfolder of the Learning Aids folder, and click Open. In the resulting Load Waves dialog, name the loaded wave "rawData". We will pretend this data is the result of Run 1. Type "Display rawData" on the command line to graph the data.

Set the Current Data Folder to Run2, and repeat the wave loading steps, selecting the CSTATIN.ASV file instead. In the resulting Load Waves dialog, name the loaded wave "rawData". We will pretend this data is the result of Run 2. Repeat the "Display rawData" command to make a graph of this data.

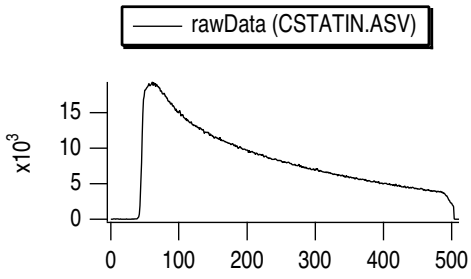
Notice that we used the same name for the loaded data. No conflict exists because the other rawData wave is in another data folder. At this point, the Data Browser should look something like this example (we've deselected Display Variables and Display Strings).



The graphs of our loaded waves look like this:

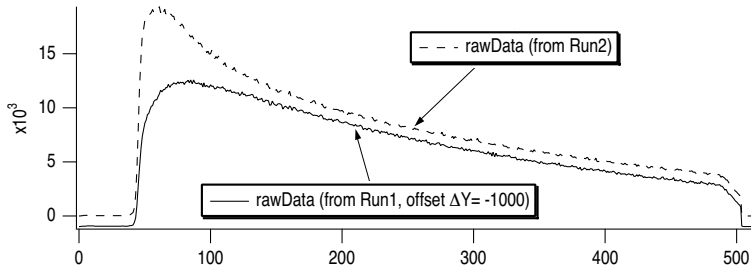


Graph of data in data folder "Run1"



Graph of data in data folder "Run2"

You can easily make a graph displaying both rawData waves to compare them better. Using the New Graph dialog, make sure Show Data Folders is selected in the Wave Browsers (see **Dialog Wave Browser** on page II-181). You can then select both waves to display in a graph. Alternatively, you can execute two commands on the Command Line: first execute `Display rawData`, change the current data folder to Run1, and then execute `AppendToGraph rawData` (or use the Append To Graph dialog)



You can change the current data folder to anything you want and the graphs will continue to display the same data; graphs remember which data folder the waves belong to, and so do graph recreation macros. This is often what you want, but not always.

Suppose you have many test runs in your experiment, each safely tucked away in its own data folder, and you want to "visit" each test run by looking at the data using a single graph which displays data from the test run's data folder only. When you visit another test run, you want the graph to display data from that other data folder only.

Additionally, suppose you want the graph characteristics to be the same (the same axis labels, annotations, line styles and colors, etc.). You could:

- Create a graph for the first test run
- Kill the window, and save the graph window macro.
- Edit the recreation macro to reference data in another data folder.
- Run the edited recreation macro.

The recreated graph will have the same appearance, but use the data from the other data folder. The editing usually involves changing a command like:

```
SetDataFolder root:Run1:
```

to:

```
SetDataFolder root:Run2:
```

If the graph displays waves from more than one data folder, you may need to edit commands like:

```
Display rawData,::Run1:rawData
```

as well.

However, there is another way that doesn't require you to edit recreation macros: use the ReplaceWave operation to replace waves (traces) in the graph with waves from the other folder.

- Switch to the other data folder.
- Select the desired graph
- Type in the command line:

```
ReplaceWave allinCDF
```

This replaces all the waves in the graph with identically named waves from the Current Data Folder, if they exist. There is no dialog for this command; see the **ReplaceWave** operation on page V-524 for more details.

Though we have only one wave, we can try it out:

- Set the Current Data Folder to Run1.
- Select the graph showing data from Run2 only (CSTATIN.ASV).
- Type in the command line:

```
ReplaceWave allinCDF
```

The graph will be updated to show the rawData wave from Run1.

You could create a Button control in the graph (see **Button** on page III-365) that executes a macro containing the ReplaceWave allinCDF command. Then you would use the Data Browser to change the Current Data Folder, and click the button to update the graph with waves from that data folder. You could also execute the same macro directly from the Data Browser in response to the user dragging the current folder indicator. To do so, use the command:

```
ModifyBrowser command3="ReplaceWave allinCDF"
```

For another Data Folder example, see the Data Folder Tutorial in "Igor Pro Folder:Learning Aids:Tutorials".

Problems with Data Folders

If you are a nonprogrammer and do not use procedures written by others then you can probably use data folders without problems. Just be aware that you need to set the current data folder (using the Data Browser) to the data folder of interest and Igor will behave as if the other data folders do not exist.

If you are a programmer and have written your own procedures, you can use data folders *after* you have made your procedures data-folder aware. However, rewriting legacy code to be data folder aware can be a big job and you should make sure the benefits will outweigh the costs before undertaking such a project.

Nonprogrammers who use procedures written by others should avoid data folders until the procedures are updated.

Igor procedures written for versions of Igor prior to Igor Pro 3.1 may not work properly if the current data folder is not root and yet will not be able to access data in other data folders if the current data folder is set to root. If you rely on procedures (and even some XOPs and XFUNCs) that are not data-folder aware, you should do some testing to verify that they work properly before committing to data folder use.

Procedures that rely on global variables and waves are likely to fail when the current data folder is not root. Unfortunately this is a very common occurrence. The reason for this is that non data-folder-aware procedures refer to waves and variables with simple object names (no data folder paths). Igor will look in the current data folder for objects that are actually in the root data folder. If the current data folder is not root, Igor will not find the named objects and will generate an error.

Also with the introduction of data folders, the name of a wave is no longer sufficient to uniquely identify it because the name does not tell you in which data folder the wave can be found (and waves with the same name can exist in different data folders). For a discussion of how to deal with this problem, see **Wave Reference Functions** on page IV-173.

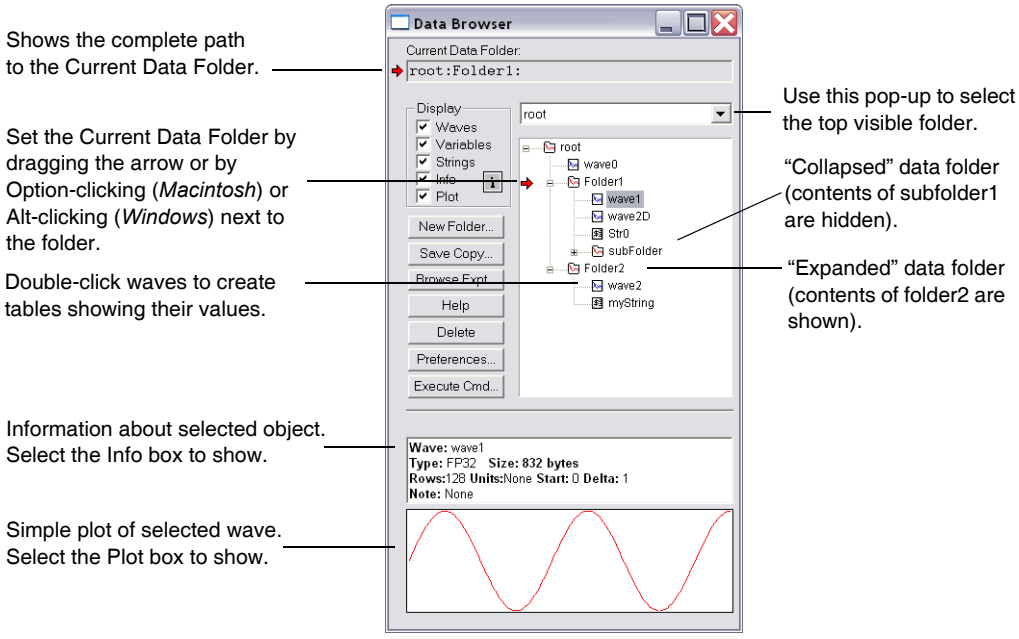
Note that the window recreation macros generated by Igor itself when you click the close button of a graph or table window are data-folder aware and will work properly regardless of the current data folder setting.

Data Browser

The Data Browser is an extension that lets you navigate through the different levels of data folders, examine values of variables, strings and waves, load data objects from other Igor experiments, and save a copy of data from the current experiment to an experiment file or folder on disk.

To open the browser choose Data Browser from the Data menu.

The user interface of the browser is similar to that of the computer desktop. The basic Igor data objects (variables, strings, waves and data folders) are represented by unique icons and arranged in the main list based on their hierarchy in the current experiment. The browser also sports several buttons that provide you with additional functionality:

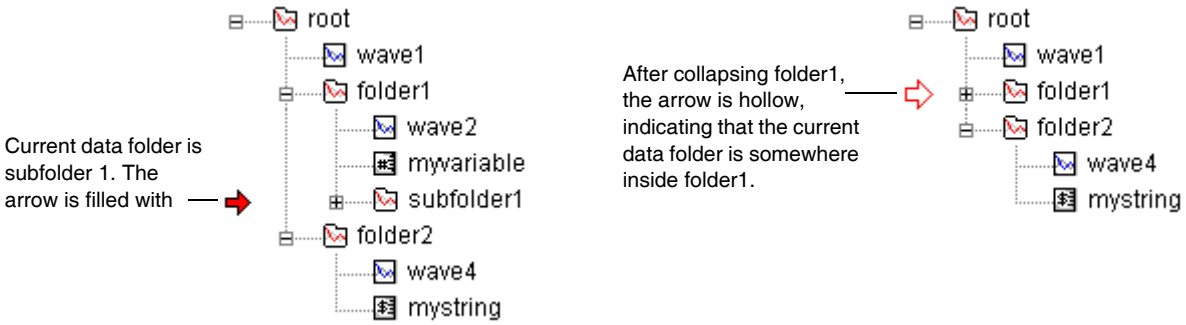


The main list occupies most of the browser when it is first invoked. At the top of the data tree is the root data folder which by default appears expanded. By double-clicking a data folder icon you can change the display so that the tree is displayed with your selection as the top data folder instead of root. You can use the pop-up menu above the main list to replace the current top data folder with another folder in the hierarchy. Following the top folder are all the data objects that it contains. Objects are grouped by type and by default they are listed in the order that they were created.

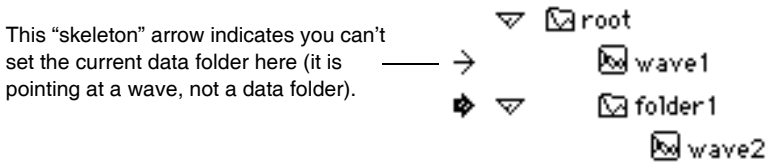
Current Data Folder

The "current data folder" is the folder that Igor uses by default for storing newly-created variables, strings, waves and other data folders. There are two indicators for the current data folder. First, above the main list there is a text box that contains the full path to the current data folder. Second, the main list has a painted red arrow to the left of the icon for the current data folder.

When the current data folder is contained inside a collapsed data folder, an unpainted (empty) arrow indicator points to the icon of the data folder containing the current data folder.



To set the current data folder, drag the current-folder indicator (red arrow) until it points to the desired data folder. You can also set the current data folder directly by clicking next to the desired data folder while pressing Option (*Macintosh*) or Alt (*Windows*).



Display Checkboxes

The Display checkboxes group lets you determine which object types are shown in the main list. Data folders are always shown.

Info Checkbox

Click in the Info Checkbox to display the Information pane of the Data Browser. The Information pane is situated below the main list. When you select a data object in the main list, its properties or contents appear in the information pane. For example, when you select a variable, its name is displayed in bold face and its value is displayed below the name. You can edit the numerical value by selecting it and typing in a new numerical value. If you modify the value of the variable, Accept and Cancel buttons will appear above the Information pane. You must either accept the change or cancel it before doing anything else with Igor.

When you select a string in the main list, the contents of the string (up to 32000 characters) will be displayed in the Information pane. Longer strings will be clipped. You can then select and edit any part of the string.

If you select a wave in the main list, the Information pane displays the wave type, size, dimensions, units, start, delta and note. Each one of these fields is displayed as a bold face name followed by plane text value. You can select and modify each one of the plane text fields by typing the new values. The only exception here is the wave type field, where you need to Control-click (*Macintosh)* or right-click (*Windows*) to select a new wave type from a pop-up menu. Note that when you change the wave type or any one of its dimensions, you might irreversibly change your data.

Another option offered by the Information pane is to display WaveStats for any selected wave. The **WaveStats** operation on page V-729 provides several statistical properties of a wave. To show WaveStats, click the sigma icon next to the Info checkbox. Note that WaveStats calculations are performed in the background and should not affect your interaction with or the performance of Igor. When you click the sigma icon, it changes to an *i* icon which you can click to return to normal mode.

Plot Checkbox

Click the Plot Checkbox to display the Plot pane of the Data Browser. The plot pane is situated below the main list and the optional Information pane. It displays a small graph or image of a wave selected in the main list above it.

Simple 1D real waves are drawn in red on a white background. Complex 1D waves are drawn as two traces with the real part drawn in red and the imaginary in blue. 2D waves are drawn as an image that by default is scaled to the size of the Plot pane and uses the Rainbow color table. To display the image using the aspect ratio implied by the number of samples in each direction or to change the color table, Control-click in the Plot pane (*Macintosh*) or right-click (*Windows*) and make the appropriate choice in the pop-up menu.

When you select 3D or 4D wave in the main list, the Plot pane displays animated images of one slice at a time. The slices represent layers relative to the data cube (3D) or the selected chunk (4D). You can stop the animation at any time by selecting from a pop-up menu in the Plot pane. You can also select the Plot pane by clicking in it and then toggling the animation by pressing Enter. When the animation is stopped you can use the cursor keys to navigate through layers and chunks.

New Folder Button

The New Folder button is used to create a new data folder inside the current data folder. The browser provides a simple dialog for specifying the name of the new data folder and tests that the name provided is valid. When entering liberal object names, you should not use single quotes around the name.

Browse Expt. Button

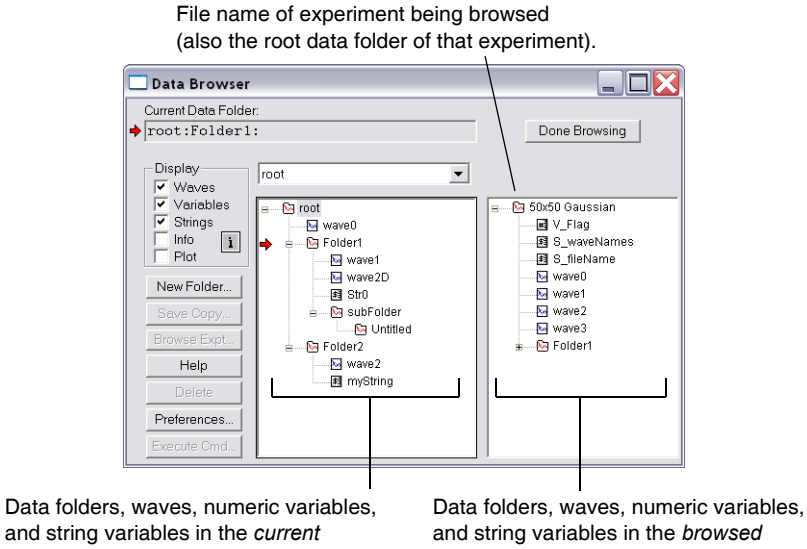
The Browse Expt. button loads data objects from Igor packed or unpacked experiments into the current experiment (in memory). When you click Browse Expt., the browser presents the standard Open dialog. You can choose to browse a packed Igor experiment file or to browse a folder on your hard disk and any subfolders.

To browse a disk folder, select the folder and click the Folder button.

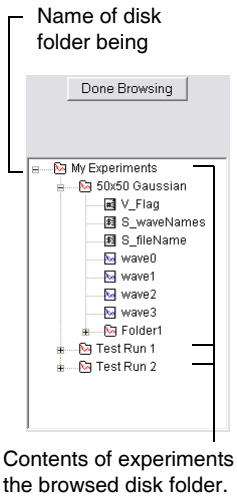
When you browse a disk folder, the browser shows you all packed Igor experiment files or unpacked Igor data files in the selected folder as well as in any subfolders.

To browse a packed experiment file, select the file in the dialog and click the Choose button.

At this stage, the browser will display, on the right-hand side, a new list containing icons representing the data in the file or folder that you selected for browsing.



Browsing an Experiment

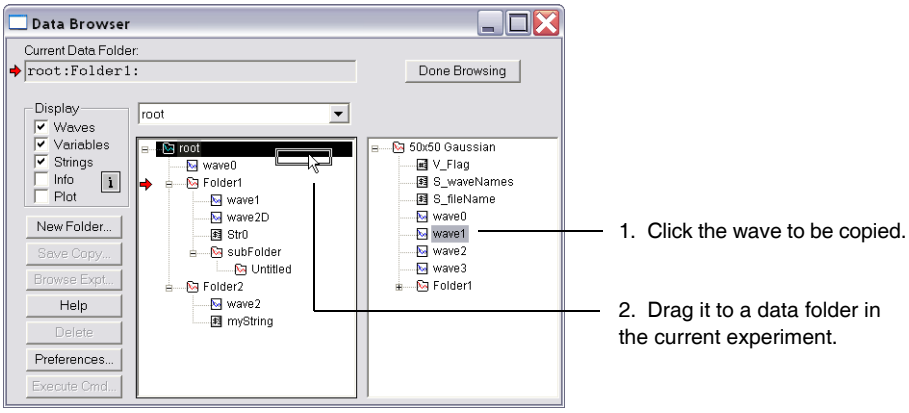


Browsing a Disk Folder

Each experiment in the new list is represented by a data folder which may contain any number of data folders and data objects.

Note: Although data folders exist wholly in memory while an experiment is active, *unpacked* experiments create a disk folder hierarchy that mirrors the data folder hierarchy. *Packed* experiments do not create a disk folder hierarchy at all. The Data Browser displays a saved experiment's data folders by examining either the packed experiment's file contents or the unpacked experiment's disk hierarchy.

You may select one or more data objects and drag them to the main list. When the cursor appears on top of a valid drop target (a data folder in the current experiment), the target is highlighted. When you release the mouse button on a valid drop target, the browser loads the corresponding data objects into the specified data folder. There is no change to the experiment from which the data is loaded.



Copying a wave from a browsed experiment into the current experiment

Clicking in the Done Browsin' button removes the additional list and resets the browser window to its size prior to the load operation.

Save Copy Button

The Save Copy button copies data objects from the current experiment to an Igor packed experiment file or to an unpacked folder on disk. Most users will not need to do this because the data will be saved when the current experiment is saved.

Before clicking Save Copy, select the data that you want to save. When you click Save Copy the browser presents a dialog in which you specify the name and location of the packed Igor experiment file which will contain a copy of the saved data.

If you press Option (*Macintosh*) or Alt (*Windows*) while clicking Save Copy, the browser presents a dialog in which you specify a folder on disk in which the data is to be saved in unpacked format. The unpacked format is intended for advanced users with specialized applications.

By default, objects are written to the output without regard to the state of the Waves, Variables and Strings checkboxes in the Display section of the Data Browser. However, there is a preference that you use change this behavior. If you enable the appropriate checkbox in the Data Browser preferences dialog, then Save Copy writes a particular type of object only if the corresponding Display checkbox is selected.

The Data Browser does not provide a method for adding or deleting data to or from a packed experiment file on disk. It can only overwrite an existing file. To add or delete, you need to open the experiment (Open Experiment in the File menu), make additions and deletions and then save the experiment. Advanced users can add data to an unpacked folder using the **SaveData** operation on page V-536.

Delete Button

The Delete button is enabled whenever data objects are selected in the main list. The browser provides a warning message listing the number of items that will be deleted. Note that clicking this button when the root folder is selected deletes all data objects in the current experiment.

Note that if you try to delete a wave that's displayed in a graph or table, it will not be deleted and you will not get an error message.

To skip the warnings, press Option (*Macintosh*) or Alt (*Windows*) when clicking the Delete button.

Warning: If you mistakenly delete something, you cannot undo it except by reverting the entire experiment to its last saved state.

The Preferences Button

The Preferences button sets the following:

- The font and font size used in the browser's directory window.

- Whether the browser will remember its window size and position when you relaunch Igor.
- The order of objects in data folders. You can choose to sort them by creation date, by name or by name and type. If you choose Creation date, objects are ordered according to the time they were created, but they are grouped according to type with waves appearing first followed by variables and strings. If you choose to order objects by name only, objects are arranged in alphabetical order within each data folder. Data folders always appear based on the tree structure and the order in which they were created.
- Whether the Save Copy button affects only the currently visible objects.

The Execute Cmd Button

The Execute Cmd button provides you with a shortcut for executing a command on selected objects in the Data Browser window. When you click in the button you get a dialog where you can specify the command, the execution mode and a secondary command for an overflow. If you set the commands once, you can skip the dialog by pressing Option when you click in the button.

The format of commands is exactly the same as any Igor commands except that you use %s where the selection is to be inserted, e.g.,

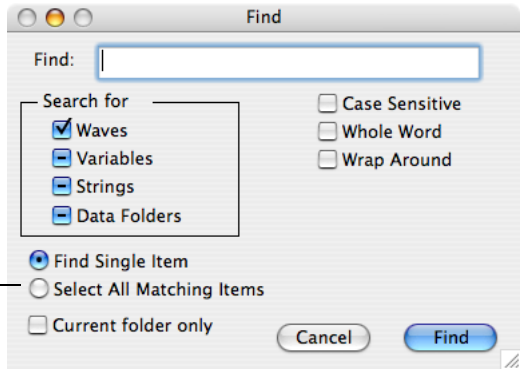
```
Display %s
Print "%s"
```

When "Execute once for all selected items" is chosen, the Data Browser enters the full path for each selection in place of %s. This may cause the command to exceed the maximum of 400 characters. Before that limit is reached, the Data Browser executes the first command on the selection followed by executing the overflow command on the remaining objects in the selection. For example, if you want to display many waves use "`Display %s`" for your first command and "`AppendToGraph %s`" as the overflow command. The command syntax should not include printf, sprintf or sscanf because of conflict between the formatting string and the %s used here.

Using the Data Browser Find Dialog

If you choose the Find item in the Edit menu, the Data Browser displays a Find dialog. This dialog finds waves, variables and data folders that might be buried in subdata folders. It also provides a convenient way to select a number of objects at one time, based on a search string. Any object with a name containing your search string will be found and selected. You can then use the Execute Cmd button to operate on the selection.

To find and select every item whose name contains the search string, click this button.



The Data Browser's Find dialog specifies the objects that you would like to find or select. You may use the "*" wildcard to specify object names of the form "abc*ef" where * represents zero or more arbitrary characters. Note that "abc*", "*abc" and "abc" are completely equivalent.

Choosing Find Same in the Edit menu or pressing Command-G (*Macintosh*) or Ctrl+G (*Windows*) performs a search in the forward direction for an item matching the same search string as was used in the previous Find. When the search reaches the end of the data objects list, it will wrap around only if the wrap around box is selected in the find dialog.

Choosing Find Selection in the Edit menu or pressing Command-H (*Macintosh*) or Ctrl+H (*Windows*) searches for an item matching the first item that is currently selected in the main list. All other search settings are those specified in the Find dialog.

Programming the Browser

The Data Browser can be controlled from the command line or from Igor procedures. Detailed reference information about the CreateBrowser, ModifyBrowser, and GetBrowserSelection commands can be found in the Command Help tab of the Igor Help Browser.

Advanced Igor programmers can use the browser as an input device via the GetBrowserSelection function or by modifying stored command strings. For an example, see the Data Folder Tutorial in "Igor Pro Folder:Learning Aids:Tutorials".

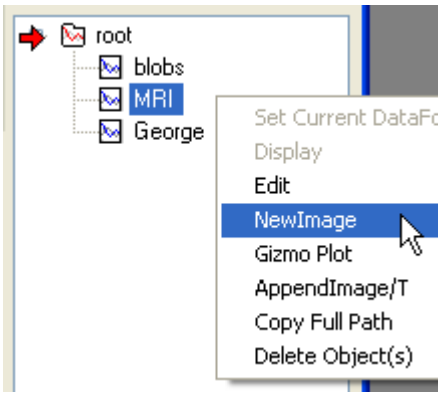
You can use the Data Browser as a modal dialog permitting a user to select one or more waves from multiple data folders. For details, see the Data Browser help file.

Browser Pop-Up Menu

You can apply various Igor operations to objects by selecting the objects in the Data Browser and choosing the operation from a pop-up menu you obtain by Control-clicking (*Macintosh*) or right-clicking (*Windows*).

Using the Display and New Image pop-up items, you can create a new graph or image plot of the selected wave. You can also select multiple waves, in the same or different data folders, to display together in the same graph.

The Copy Full Path item copies the complete data folder paths of the selected objects to the clipboard.



Other Browser Operations

You can rename data objects by clicking the name of the object and editing the name.

The browser also supports icon dragging as means of moving or copying data objects from one data folder to another. You can select multiple data objects by Shift-clicking (*Macintosh*) or Ctrl-clicking (*Windows*) on them.

You can move data objects from one data folder to another by dragging them.

You can copy data objects from one data folder to another by holding down Option (*Macintosh*) or Alt (*Windows*) while dragging.

You can duplicate data objects within a folder by choosing Duplicate from the Edit menu or by pressing Command-D (*Macintosh*) or Ctrl+D (*Windows*).

Note: Objects remain selected even when they are hidden inside collapsed data folders. If you select a wave, collapse its data folder, Shift-select another wave, and drag it to another data folder, both waves will be moved there.

However, when a selected object is hidden by deselecting the relevant Display checkbox, no action (e.g., delete or duplicate) is taken upon it except if you use Save Copy and your preference setting is to save nonvisible objects.

Data Browser Shortcuts

Action	Shortcut
To set the current data folder	Drag the red arrow until it points to the desired data folder, or Option-click (*Macintosh*) or Alt-click (*Windows*) next to the desired data folder.
To display a graph or an image of a wave	Control-click (*Macintosh*) or right-click (*Windows*) and select an option from the pop-up menu.
To view the contents of a "collapsed" data folder	Click the triangle (*Macintosh*) or the plus button (*Windows*) next to the data folder.
To "collapse" a data folder	Click the triangle (*Macintosh*) or the minus button (*Windows*) next to the data folder.
To move the selection up or down by one object	Press Up Arrow or Down Arrow.
To move an object from one data folder to another	Drag the object onto the destination data folder.
To move several objects from one data folder to another	Select the objects by Shift-clicking them (*Macintosh*) or Ctrl-clicking them (*Windows*). Drag the selected objects onto the desired data folder.
To copy an object from one data folder to another	Drag the object while holding down Option (*Macintosh*) or Alt (*Windows*).
To duplicate an object	Select the object and press Command-D (*Macintosh*) or Ctrl+D (*Windows*).
To rename an object	Click the object's name and type a new name. To finish, press Return, Enter, Tab, or click outside the name.
To view a wave's values in a table	Double-click the wave's icon.
To print the value of a variable or string in the history area	Double-click the variable or string icon.
To delete an object without the confirmation dialog	Press Option (*Macintosh*) or Alt (*Windows*) while clicking the Delete button.
To find objects in the browser list	Choose Find from the Edit menu, or press Command-F (*Macintosh*) or Ctrl+F (*Windows*).
To find the same thing again	Choose Find Same from the Edit menu, or press Command-G (*Macintosh*) or Ctrl+G (*Windows*).
To find names containing selected text	Choose Find Selection from the Edit menu, or press Command-H (*Macintosh*) or Ctrl+H (*Windows*).
To execute a command on a set of waves	Select the icon for each wave that the command is to act on and click the Execute Cmd button.
To execute a command on a set of waves without going through the Execute Cmd dialog	Select the icon for each wave that the command is to act on, press Option (*Macintosh*) or Alt (*Windows*), and click the Execute Cmd button. This reexecutes the command entered previously in the Execute Cmd dialog.
To see WaveStats for a selected wave	Control-click (*Macintosh*) or right-click (*Windows*) in the information pane and select the WaveStats mode from the pop-up menu.
To change colormap, activate animation in plot pane	Control-click (*Macintosh*) or right-click (*Windows*) in the plot pane and select the appropriate option from the pop-up menu.

Action	Shortcut
To stop the animation in the plot pane	Click in the plot pane and then press Return or Enter.
To navigate between displayed layers or chunks	Stop the animation in the plot pane and use the cursor keys, Page Down, Page Up, Home, and End.
To close the information pane	Double click the horizontal separator bar.
To save a copy of selected data in unpacked format	Press Option (*Macintosh*) or Alt (*Windows*) while clicking the Save Copy button.

Chapter
II-9

Importing and Exporting Data

Loading Waves

Most Igor users create waves by loading data from a file created by another program. The process of loading a file creates new waves and then stores data from the file in them. Optionally, you can overwrite existing waves instead of creating new ones. The waves can be numeric or text and of dimension 1 through 4.

Igor provides a number of different routines for loading data files. There is no single file format for numeric or text data that all programs can read and write.

There are two broad classes of files used for data interchange: text files and binary files. Text files are usually used to exchange data between programs. Although they are called text files, they may contain numeric data, text data or both. In any case, the data is encoded as plain text that you can read in a text editor. Binary files usually contain data that is efficiently encoded in a way that is unique to a single program and can not be viewed in a text editor.

The closest thing to a universally accepted format for data interchange is the "delimited text" format. This consists of rows and columns of numeric or text data with the rows separated by carriage return characters (*Macintosh*), linefeed return characters (*Unix*), or carriage return/linefeed (*Windows*) and the columns separated by tabs or commas. The tab or comma is called the "delimiter character". Igor can read delimited text files written by most programs.

FORTRAN programs usually create fixed field text files in which a fixed number of characters is used for each column of data with spaces as padding between columns. The Load Fixed Field Text routine is designed to read these files.

Text files are convenient because you can create, inspect or edit them with any text editor. In Igor, you can use a notebook window for this purpose. If you have data in a text file that has an unusual format, you may need to manually edit it before Igor can load it.

Text files generated by scientific instruments or custom programs often have "header" information, usually at the start of the file. The header is not part of the block of data but contains information associated with it. Igor's text loading routines are designed to load the block of data, not the header. The Load General Text routine can usually automatically skip the header. The Load Delimited Text and Load Fixed Field Text routines needs to be told where the block of data starts if it is not at the start of the file.

An advanced user could write an Igor procedure to read and parse information in the header using the Open, FReadLine, StrSearch, sscanf and Close operations as well as Igor's string manipulation capabilities. Igor includes an example experiment named Load File Demo which illustrates this.

If you will be working on a Macintosh, and loading data from files on a PC, or vice-versa, you should look at **File System Issues** on page III-398.

The following table lists the six types of built-in data loading routines in Igor and their salient features.

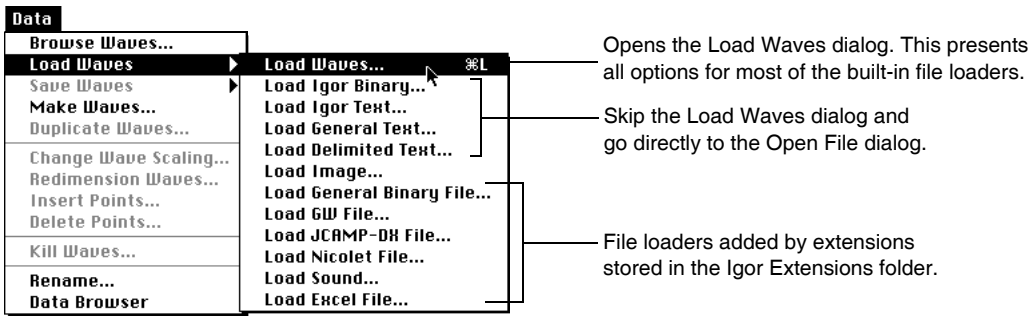
File Type	Description
Delimited text	Created by spreadsheets, database programs, data acquisition programs, text editors, custom programs. This is the most commonly used format for exchanging data between programs.
	Row Format: `<data><delimiter><data><CR>`
	Contains one block of data with any number of rows and columns. A row of column labels is optional.
	Can load numeric, text, date, time, and date/time columns.
	Can load columns into 1D waves or blocks into 2D waves.
	Columns may be equal or unequal in length.

File Type	Description
Fixed field text	Created by FORTRAN programs.
	Row Format: <data><padding><data><padding><CR>
	Contains one block of data with any number of rows and columns.
	Each column consists of a fixed number of characters including any space characters which are used for padding.
	Can load numeric, text, date, time and date/time columns.
	Can load columns into 1D waves or blocks into 2D waves.
	Columns are usually equal in length but do not have to be.
General text	Created by spreadsheets, database programs, data acquisition programs, text editors, custom programs.
	Row Format: `<number><white space><number><CR>`
	Contains one or more blocks of numbers with any number of rows and columns. A row of column labels is optional.
	Can not handle columns containing nonnumeric text, dates and times.
	Can load columns into 1D waves or blocks into 2D waves.
	Columns must be equal in length.
	Igor's Load General Text routine has the ability to automatically skip nonnumeric header text.
Image	Created by a wide variety of programs.
	Format: Always binary. Varies according to file type.
	Can load GIF, JPEG, PNG, PICT, TIFF, BMP, PhotoShop, Silicon Graphics, Sun raster, and Targa graphics files.
	Can load data into matrix waves, including TIFF image stacks.
Igor Text	Created by Igor, custom programs. Used mostly as a means to feed data and commands from custom programs into Igor.
	Format: See **Igor Text File Format** on page II-159.
	Can load numeric and text data.
	Can load data into waves of dimension 1 through 4.
	Contains one or more wave blocks with any number of waves and rows.
	Consists of special Igor keywords, numbers and Igor commands.
Igor Binary	Created by Igor, custom programs. Used by Igor to store wave data.
	Each file contains data for one Igor wave of dimension 1 through 4.
	Format: See Igor Technical Note #003, "Igor Binary Format".

In addition, extensions to Igor are available to load data from other types of files, including Excel, Matlab, HDF, HDF5, JCAMP, DEM, DLG, Nicolet, various sound formats and general binary files. See **Loading Other Files** on page II-167 for details.

Load Waves Submenu

You access all of these routines via the Load Waves submenu of the Data menu.



The Load Waves item in this submenu leads to the Load Waves dialog. This dialog invokes all of the built-in loading routines except for the image loader and accesses all available options.

The Load Igor Binary, Load Igor Text, Load General Text, and Load Delimited Text items in the Load Waves submenu are shortcuts that access the respective file loading routines with default options. We recommend that you start with the Load Waves item so that you can see what options are available. There are no shortcut items for loading fixed field text or image data because these formats require that you specify certain parameters.

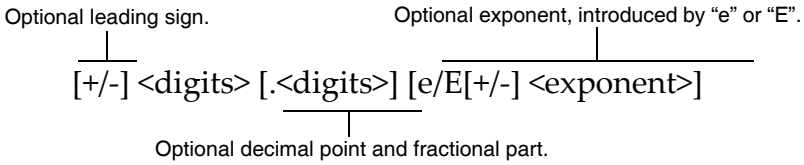
The Load Image item leads to the Load Image dialog which provides the means to load various kinds of image files.

The remaining items are provided by Igor File-Loader Extensions. These are plug-in software modules that can be installed or removed easily as described under **Loading Other Files** on page II-167.

All of the built-in file loaders can load numeric data. The delimited text and fixed field text loaders can also load string text, date, time and date/time data.

Number Formats

A number has the following form:



An example is “-17.394e+3”. Some FORTRAN programs write “d” or “D” instead of “e” or “E” to introduce the exponent. Igor recognizes this.

The End of the Line

Different computer systems use different characters to mark the end of a line in a text file. The Macintosh uses the carriage-return character (CR). Unix uses linefeed (LF). Windows uses a carriage-return and linefeed (CRLF) combination. When loading waves, Igor treats a single CR, a single LF, or a CRLF as the end of a line. This allows Igor to load text data from file servers on a variety of computers without translation.

Loading Delimited Text Files

A delimited text file consists of rows of values separated by tabs or commas with a carriage return, linefeed or carriage return/linefeed combination at the end of the row. There may optionally be a row of column labels. Igor can load each column in the file into a separate 1D wave or it can load all of the columns into a single 2D wave. There is no limit to the number of rows or columns except that all of the data must fit in available memory.

In addition to numbers and text, the delimited text file may contain dates, times or date/times. The Load Delimited Text routine attempts to automatically determine which of these formats is appropriate for each column in the file. You can override this automatic determination if necessary.

A numeric column can contain, in addition to numbers, NaN and [±]INF. NaN means "Not a Number" and is the way Igor represents a blank or missing value in a numeric column. INF means "infinity". If Igor finds text in a numeric or date/time column that it can't interpret according to the format for that column, it treats it as a NaN.

If Igor encounters, in any column, a delimiter with no data characters preceding it (i.e., two tabs in a row) it takes this as a missing value and stores a blank in the wave. In a numeric wave, a blank is represented by a NaN. In a text wave, it is represented by an element with zero characters in it.

Date/Time Formats

The Load Delimited Text routine can handle dates in many formats. A few "standard" formats are supported and in addition, you can specify a "custom" format (see **Custom Date Formats** on page II-144).

The standard date formats are:

mm/dd/yy	(month/day/year)
mm/yy	(month/year)
dd/mm/yy	(day/month/year)

To use the dd/mm/yy format instead of mm/dd/yy, you must set a tweak. See **Delimited Text Tweaks** on page II-151.

You can also use a dash or a dot as a separator instead of a slash.

Igor can also handle times in the following forms:

[+][-]hh:mm:ss [AM PM]	(hours, minutes, seconds)
[+][-]hh:mm:ss.ff [AM PM]	(hours, minutes, seconds, fractions of seconds)
[+][-]hh:mm [AM PM]	(hours, minutes)
[+][-]hhhh:mm:ss.ff	(hours, minutes, seconds, fractions of seconds)

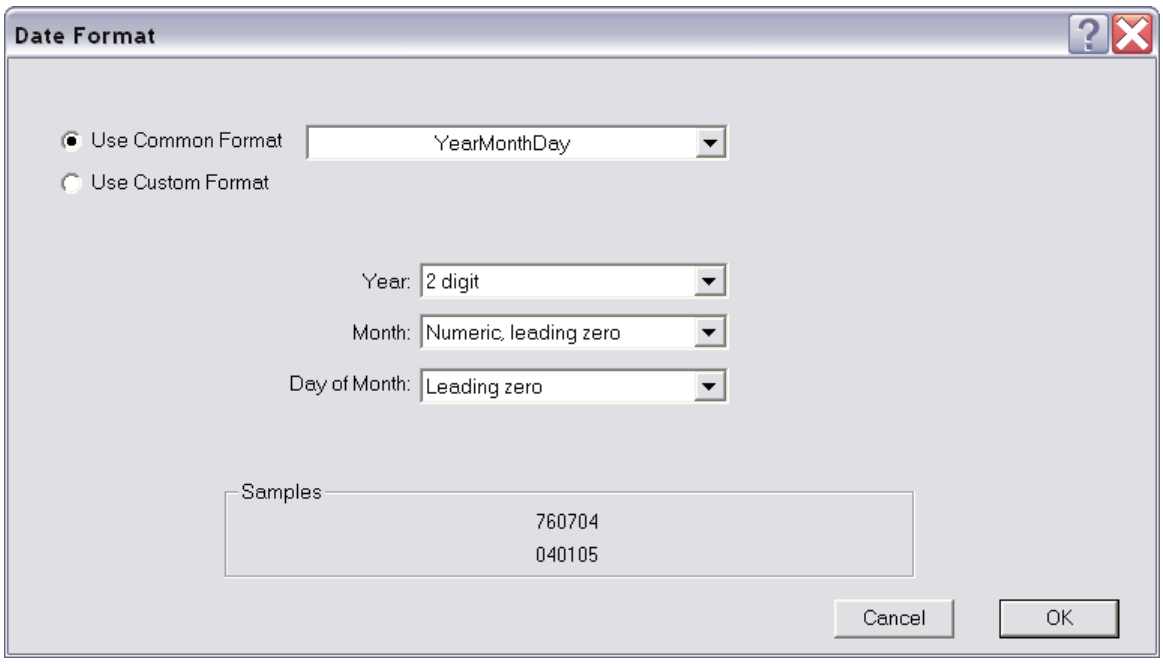
The first three forms are time-of-day forms. The last one is the elapsed time. In an elapsed time, the hour is in the range 0 to 9999.

The year can be specified using two digits (99) or four digits (1999). If a two digit year is in the range 00 … 39, Igor treats this as 2000 … 2039. If a two digit year is in the range 40 … 99, Igor treats this as 1940 … 1999.

The Load Delimited Text routine can also handle date/times which consist of one of these date formats, a single space or the letter T, and then one of the time formats.

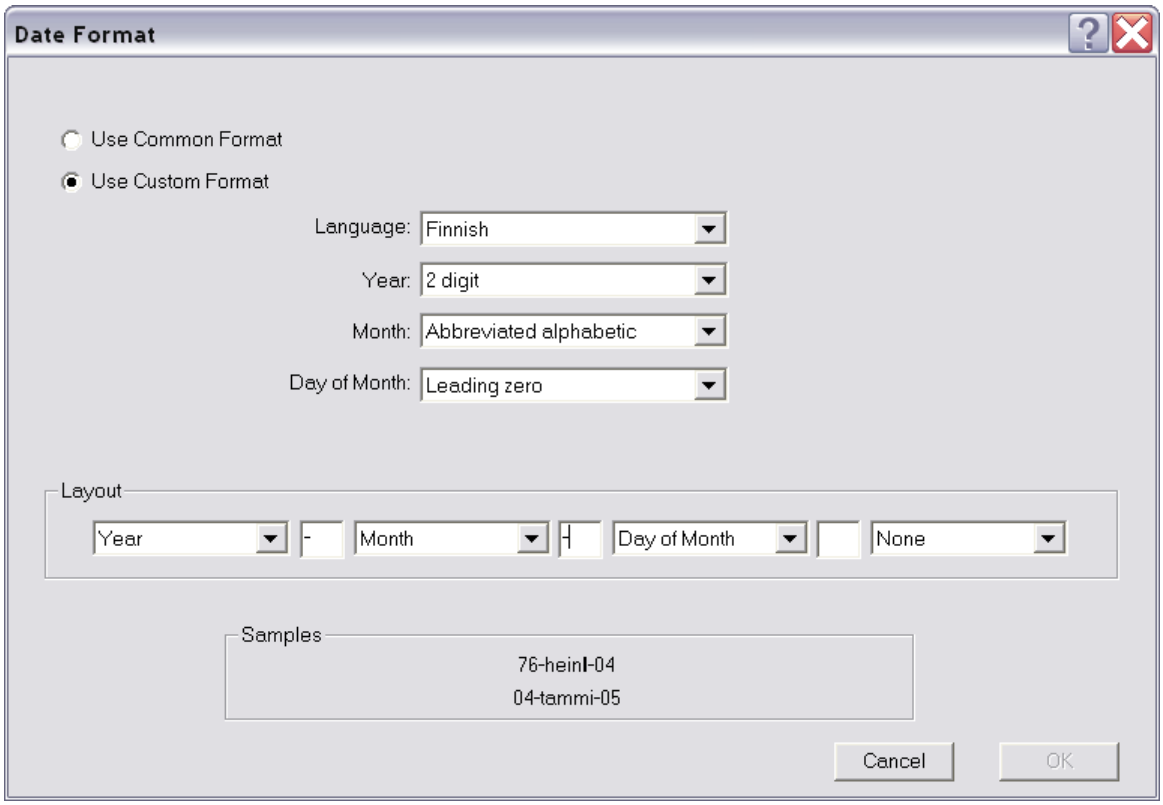
Custom Date Formats

If your data file contains dates in a format other than the "standard" format, you can use Load Delimited Text to specify exactly what date format to use. You do this using the Delimited Text Tweaks dialog which you access through the Tweaks button in the Load Waves dialog. Choose Other from the Date Format pop-up menu. This leads to the Date Format dialog.



By clicking the Use Common Format radio button, you can choose from a pop-up menu of common formats. After choosing a common format, you can still control minor properties of the format, such as whether to use 2 or 4 digits years and whether to use leading zeros or not.

In the rare case that your file's date format does not match one of the common formats, you can use a full custom format by clicking the Use Custom Format radio button. It is best to first choose the common format that is closest to your format and then click the Use Custom Format button. Then you can make minor changes to arrive at your final format.



When you use either a common format or a full custom format, the format that you specify must match the date in your file exactly.

When loading data as delimited text, if you use a date format containing a comma, such as "October 11, 1999", you must make sure that LoadWave operation will not treat the comma as a delimiter. You can do this using the Delimited Text Tweaks dialog.

When loading a date format that consists entirely of digits, such as 991011, you should use the LoadWave/B flag to specify that the data is a date. Otherwise, LoadWave will treat it as a regular number. The /B flag can not be generated from the dialog — you need to use the LoadWave operation from the command line. Another approach is to use the dialog to generate a LoadWave command without the /B flag and then specify that the column is a date column in the Loading Delimited Text dialog that appears when the LoadWave operation executes.

Column Labels

Each column may optionally have a column label. When loading 1D waves, if you read wave names and if the file has column labels, Igor will use the column labels for wave names. Otherwise, Igor will automatically generate wave names of the form wave0, wave1 and so on.

Igor considers text in the label line to be a column label if that text can not be interpreted as a data value (number, date, time, or datetime) or if the text is quoted using single or double quotes.

When loading a 2D wave, Igor optionally uses the column labels to set the wave's column dimension labels. The wave name does not come from column labels but is automatically assigned by Igor. You can rename the wave after loading if you wish.

Igor expects column labels to appear in a row of the form:

```
<label><delimiter><label><delimiter>…<label><CR> (or CRLF or LF)
```

where <column label> may be in one of the following forms:

<label>	(label with no quotes)
"<label>"	(label with double quotes)
'<label>'	(label with single quotes)

The default delimiter characters are tab and comma. There is a tweak (see **Delimited Text Tweaks** on page II-151) for using other delimiters.

Igor expects that the row of column labels, if any, will appear at the beginning of the file. There is a tweak (see **Delimited Text Tweaks** on page II-151) that you can use to specify if this is not the case.

Igor will clean up column labels found in the file, if necessary, so that they are legal wave names using standard name rules. The cleanup consists of converting illegal characters into underscores and truncating long names to the maximum of 31 characters.

Examples of Delimited Text

Here are some examples of text that you might find in a delimited text file. These examples are tab-delimited.

Simple delimited text

```
ch0         ch1         ch2         ch3         (optional row of labels)
2.97055     1.95692     1.00871     8.10685
3.09921     4.08008     1.00016     7.53136
3.18934     5.91134     1.04205     6.90194
```

Loading this text would create four waves with three points each or, if you specify loading it as a matrix, a single 3 row by 4 column wave.

Delimited text with missing values

```
ch0         ch1         ch2         ch3         (optional row of labels)
2.97055     1.95692                 8.10685
3.09921     4.08008     1.00016     7.53136
            5.91134     1.04205
```

Loading this text as 1D waves would create four waves. Normally each wave would contain three points but there is an option to ignore blanks at the end of a column. With this option, ch0 and ch3 would have two points. Loading as a matrix would give you a single 3 row by 4 column wave with blanks in columns 0, 2 and 3.

Delimited text with a date column

```
Date        ch0         ch1         ch2         (optional row of labels)
2/22/93     2.97055     1.95692     1.00871
2/24/93     3.09921     4.08008     1.00016
2/25/93     3.18934     5.91134     1.04205
```

Loading this text as 1D waves would create four waves with three points each. Igor would convert the dates in the first column into the appropriate number using the Igor system for storing dates (number of seconds since 1/1/1904). Loading as a matrix would give you a single 3 row by 4 column wave with column 0 containing dates encoded as numbers.

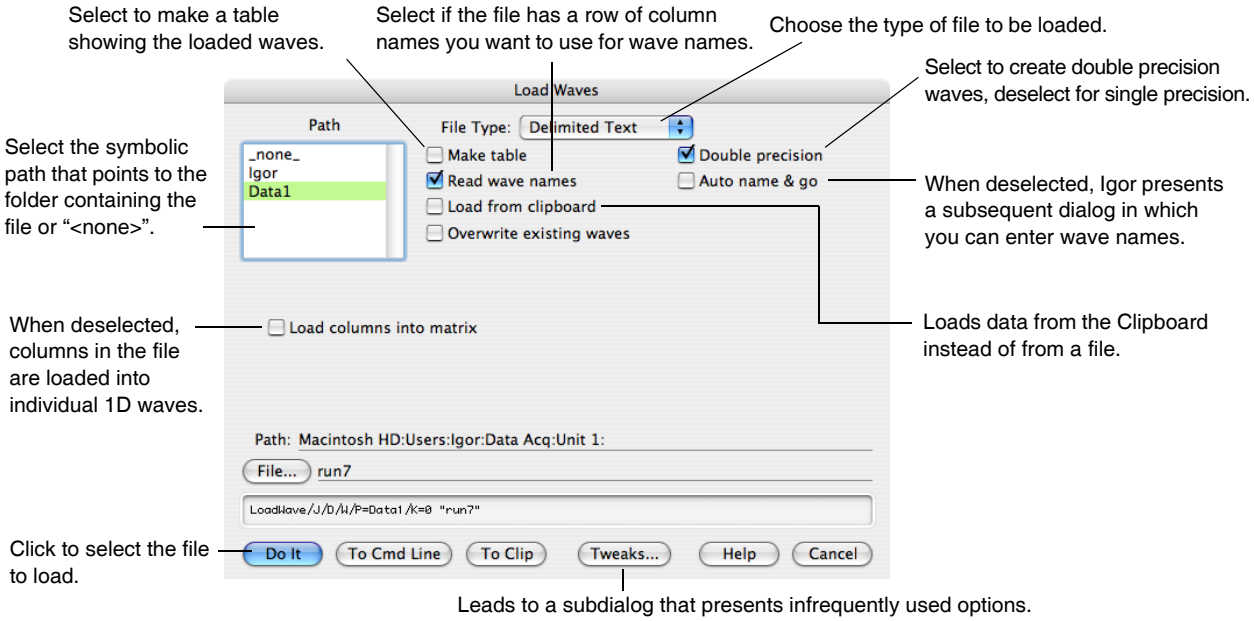
Delimited text with a nonnumeric column

```
Sample      ch0         ch1         ch2         (optional row of labels)
Ge          2.97055     1.95692     1.00871
Si          3.09921     4.08008     1.00016
GaAs        3.18934     5.91134     1.04205
```

Loading this text as 1D waves would normally create four waves with three points each. The first wave would be a text wave and the remaining would be numeric. You could also load this as a single 3x3 matrix, treating the first row as column labels and the first column as row labels for the matrix. If you loaded it as a matrix but did not treat the first column as labels, it would create a 3 row by 4 column text wave, not a numeric wave.

The Load Waves Dialog for Delimited Text — 1D

To load a delimited text file as 1D waves, invoke the Load Waves dialog by choosing the Load Waves menu item.



The basic process of loading 1D data from a delimited text file is as follows:

1. Bring up the Load Waves dialog.

2. Choose Delimited Text from the File Type pop-up menu.
3. Click the File button to select the file containing the data.
4. Click Do It.

When you click Do It, the LoadWave operation runs. It executes the Load Delimited Text routine which goes through the following steps:

1. Optionally, determine if there is a row of column labels.
2. Determine the number of columns.
3. Determine the format of each column (number, text, date, time or date/time).
4. Optionally, present another dialog allowing you to confirm or change wave names.
5. Create waves.
6. Load the data into the waves.

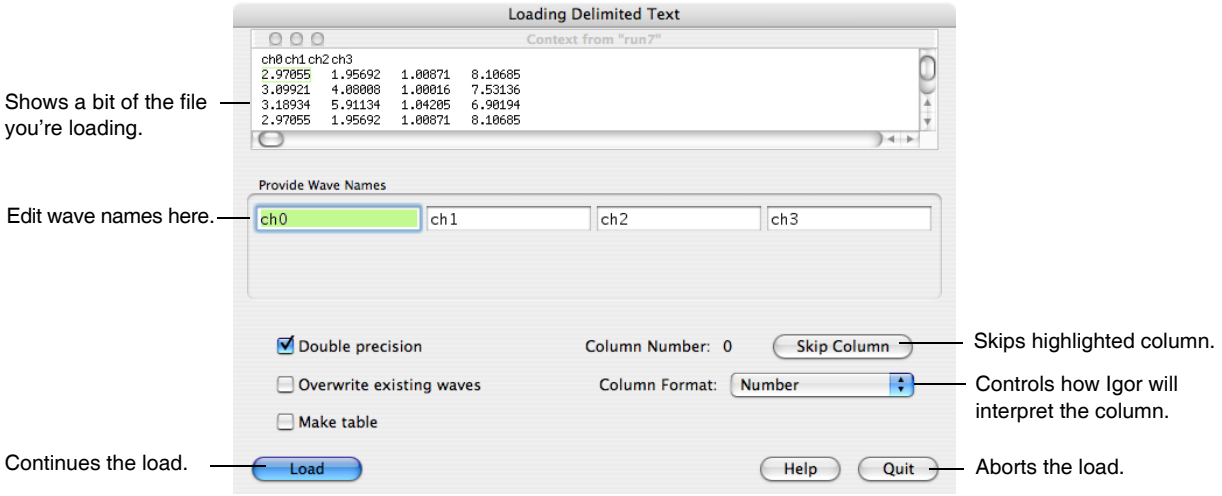
Igor looks for a row of labels only if you enable the "Read wave names" option. If you enable this option and if Igor finds a row of labels then this determines the number of columns that Igor expects in the file. Otherwise, Igor counts the number of data items in the first row in the file and expects that the rest of the rows have the same number of columns.

In step 3 above, Igor determines the format of each column by examining the first data item in the column. Igor will try to interpret all of the remaining items in a given column using the format that it determines from the first item in the column.

If you choose Load Delimited Text from the Load Waves submenu instead of choosing Load Waves, Igor will display a dialog from which you can select the delimited text file to load directly. This is a shortcut that skips the Load Waves dialog and uses default options for the load. This will always load 1D waves, not a matrix. Before you use this shortcut, take a look at the Load Waves dialog so you can see what options are available.

Editing Wave Names

The "Auto name & go" option is used mostly when you're loading 1D data under control of an Igor procedure and you want everything to be automatic. When loading 1D data manually, you normally leave the "Auto name & go" option deselected. Then Igor presents an additional dialog in which you can confirm or change wave names.



The context area gives you feedback on what Igor is about to load. You can't edit the file here. If you want to edit the file, abort the load and open the file as an Igor notebook or open it in a word processor.

Set Scaling After Loading Delimited Text Data

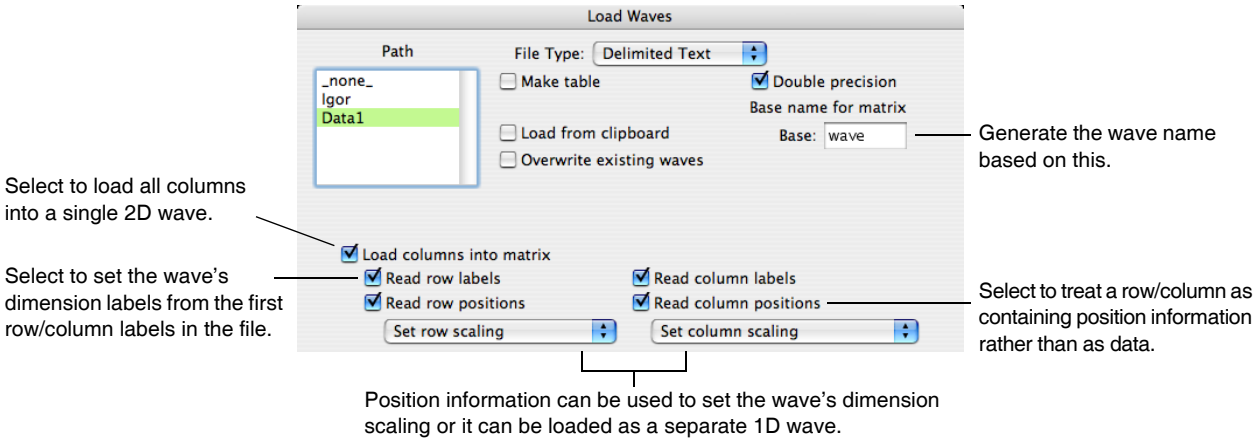
If your 1D numeric data is uniformly spaced in the X dimension then you will be able to use the many operations and functions in Igor designed for waveform data. You will need to set the X scaling for your waves after you load them, using the Change Wave Scaling dialog.

Note: If your 1D data is uniformly spaced it is *very important* that you set the X scaling of your waves. Many Igor operations depend on the X scaling information to give you correct results.

If your 1D data is not uniformly spaced then you will use XY pairs and you do not need to change X scaling. You may want to use Change Wave Scaling to set the data units.

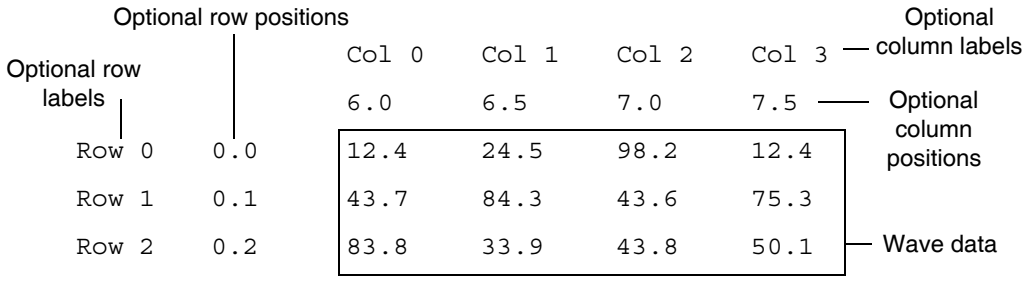
The Load Waves Dialog for Delimited Text — 2D

To load a delimited text file as a 2D wave, choose the Load Waves menu item. Then, select the "Load columns into matrix" checkbox.



When you load a matrix (2D wave) from a text file, Igor creates a single wave. Therefore, there is no need for a second dialog to enter wave names. Instead, Igor automatically names the wave based on the base name that you specify. After loading, you can then rename the wave if you want.

To understand the row/column label/position controls, you need to understand Igor's view of a 2D delimited text file:



In the simplest case, your file has just the wave data — no labels or positions. You would indicate this by deselecting all four label/position checkboxes.

2D Label and Position Details

If your file does have labels or positions, you would indicate this by selecting the appropriate checkbox. Igor expects that row labels appear in the first column of the file and that column labels appear in the first line of the file unless you instruct it differently using the Tweaks subdialog (see **Delimited Text Tweaks** on page II-151). Igor loads row/column labels into the wave's dimension labels (described in Chapter II-6, **Multidimensional Waves**).

Igor can treat column positions in one of two ways. It can use them to set the dimension scaling of the wave (appropriate if the positions are uniformly-spaced) or it can create separate 1D waves for the positions. Igor expects row positions to appear in the column immediately after the row labels or in the first column of the file if the file contains no row labels. It expects column positions to appear immediately after the column labels or in the first line of the file if the file contains no column labels unless you instruct it differently using the Tweaks subdialog.

A row position wave is a 1D wave that contains the numbers in the row position column of the file. Igor names a row position wave "RP_ " followed by the name of the matrix wave being loaded. A column position wave is a 1D wave that contains the numbers in the column position line of the file. Igor names a column position wave "CP_" followed by the name of the matrix wave being loaded. Once loaded (into separate 1D waves or into the matrix wave's dimension scaling), you can use row and column position information when displaying a matrix as an image or when displaying a contour of a matrix.

If your file contains header information before the data, column labels and column positions, you need to use the Tweaks subdialog to specify where to find the data of interest. The "Line containing column labels" tweak specifies the line on which to find column labels. The "First line containing data" tweak specifies the first line of data to be stored in the wave itself. The first line in the file is considered to be line zero.

If you instruct LoadWave to read column positions, it determines which line contains them in one of two ways, depending on whether or not you also instructed it to read column labels. If you do ask LoadWave to read column labels, then LoadWave assumes that the column positions line immediately follows the column labels line. If you do not ask LoadWave to read column labels, then LoadWave assumes that the column positions line immediately precedes the first data line.

Loading Text Waves from Delimited Text Files

With regard to text columns, the Load Delimited Text operation can work in one of three ways: auto-identify column type, treat all columns as numeric, treat all columns as text. You can specify which method you want to use using the Tweaks subdialog of the Load Delimited Text dialog.

In the "auto-identify column type" method, Igor attempts to determine whether a column is numeric or text by examining the file. This is the default method when you choose Data→Load Waves→Load Delimited Text. Igor looks for the first nonblank value in each column and determines if the value is numeric or not. If it is numeric, Igor loads the column into a numeric wave which could be plain numeric, date, time or date/time as appropriate. If it is not numeric, Igor loads the column into a text wave.

In the "treat all columns as numeric" method, Igor loads all columns into numeric waves. This is the default method when you use the LoadWave/J operation from the command line or from an Igor procedure. We made LoadWave/J behave this way by default for backward-compatibility reasons. In ensures that Igor procedures will work the same in Igor Pro 3.0 and later as they did before. To use the "auto-identify column type" method, you need to use LoadWave/J/K=0.

In the "treat all columns as text" method, Igor loads all columns into text waves. This method may have use in rare cases in which you want to do text-processing on a file by loading it into a text wave and then using Igor's string manipulation capabilities to massage it.

There are a few issues relating to special characters that you may need to deal with when loading data into text waves.

By default, the Load Delimited Text operation considers comma and tab characters to be delimiters which separate one column from the next. If the text that you are loading may contain commas or tabs as values rather than as delimiters, you will need to change the delimiter characters. You can do this using the Tweaks subdialog of the Load Delimited Text dialog.

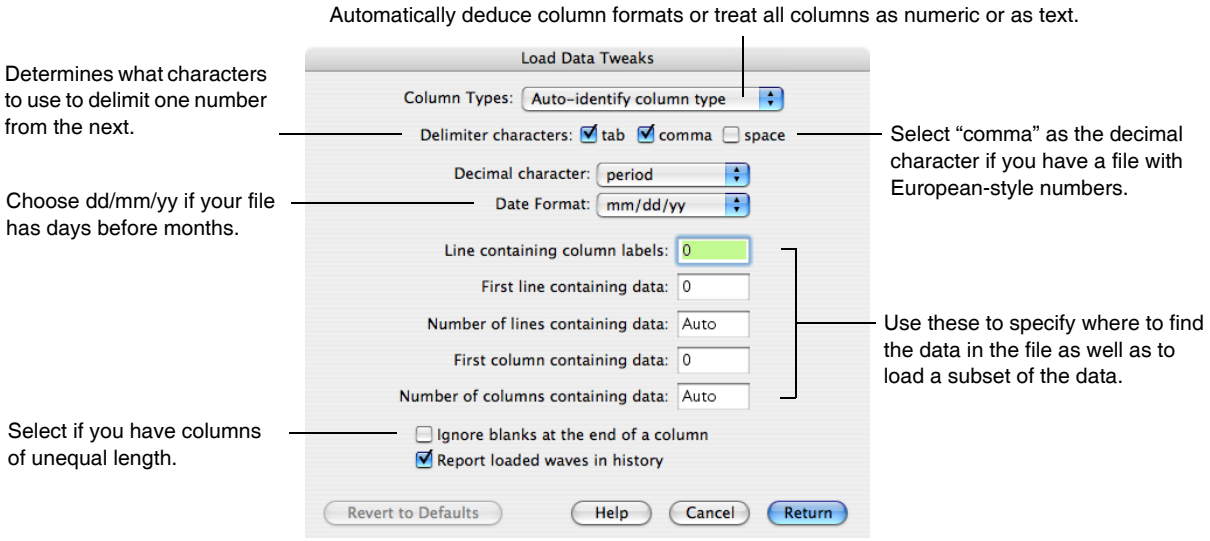
The Load Delimited Text operation always considers carriage return and linefeed characters to mark the end of a line of text. It would be quite unusual to find a data file that uses these characters as values. In the extremely rare case that you need to load a carriage return or linefeed as a value, you can use an escape sequence. Replace the carriage return value with "\r" (without the quotes) and the linefeed value with "\n". Igor will convert these to carriage return and linefeed and store the appropriate character in the text wave.

In addition to "\r" and "\n", Igor will also convert "\t" into a tab value and do other escape sequence conversions (see **Escape Characters in Strings** on page IV-13). These conversions create a possible problem which should be quite rare. You may want to load text that contains "\r", "\n" or "\t" sequences which you do not want to be treated as escape sequences. To prevent Igor from converting them into carriage return and tab, you will need to replace them with "\\r", "\\n" and "\\t".

Igor does not remove quotation marks when loading data from delimited text files into text waves. If necessary, you can do this by opening the file as a notebook and doing a mass replace before loading or by displaying the loaded waves in a table and using Edit→Replace.

Delimited Text Tweaks

There are many variations on the basic form of a delimited text file. We've tried to provide tweaks that allow you to guide Igor when you need to load a file that uses one of the more common variations. To do this, use the Tweaks button in the Load Waves dialog. Most people will not need to use the tweaks.



The Tweaks dialog can specify the space character as a delimiter. Use the LoadWave operation to specify other delimiters as well.

The main reason for allowing space as a delimiter is so that we can load files that use spaces to align columns. This is a common format for files generated by FORTRAN programs. Normally, you should use the fixed field text loader to load these files, not the delimited text loader. If you do use the delimited text loader and if space is allowed as a delimiter then Igor treats any number of consecutive spaces as a single delimiter. This means that two consecutive spaces do not indicate a missing value as two consecutive tabs would.

When loading a delimited file, Igor normally expects the first line in the file to contain either column labels or the first row of data. There are several tweaks that you can use for a file that doesn't fit this expectation.

Lines and columns in the tweaks dialog are numbered starting from zero.

Using the "Line containing column labels" tweak, you can specify on what line column labels are to be found if not on line zero. Using this and the "First line containing data" tweak, you can instruct Igor to skip garbage, if any, at the beginning of the file.

The "First line containing data", "Number of lines containing data", "First column containing data", and "Number of columns containing data" tweaks are designed to allow you to load any block of data from anywhere within a file. This might come in handy if you have a file with hundreds of columns but you are only interested in a few of them.

If "Number of lines containing data" is set to "auto" or 0, Igor will load all lines until it hits the end of the file. If "Number of columns containing data" is set to "auto" or 0, Igor will load all columns until it hits the last column in the file.

The proper setting for the "Ignore blanks at the end of a column" tweak depends on the kind of 1D data stored in the file. If a file contains some number of similar columns, for example four channels of data from a digital oscilloscope, you probably want all of the columns in the file to be loaded into waves of the same length. Thus, if a particular column has one or more missing values at the end, the corresponding points in the wave should contain NaNs to represent the missing value. On the other hand, if the file contains a number of dissimilar columns, then you might want to ignore any blank points at the end of a column so that the resulting waves will not necessarily be of equal length. If you enable the "Ignore blanks at the end of a column" tweak then LoadWave will not load blanks at the end of a column into the 1D wave. If this option is enabled and a particular column has nothing but blanks then the corresponding wave is not loaded at all.

Troubleshooting Delimited Text Files

You can examine the waves created by the Load Delimited Text routine using a table. If you don't get the results that you expected, you will need to try other LoadWave options or inspect and edit the text file until it is in a form that Igor can handle. Remember the following points:

- Igor expects the file to consist of numeric values, text values, dates, times or date/times separated by tabs or commas unless you set tweaks to the contrary.
- Igor expects a row of column labels, if any, to appear in the first line of the file unless you set tweaks to the contrary. It expects that the column labels are also delimited by tabs or commas unless you set tweaks to the contrary. Igor will not look for a line of column labels unless you enable the Read Wave Names option for 1D waves or the Read Column Labels options for 2D waves.
- Igor determines the number of columns in the file by inspecting the column label row or the first row of data if there is no column label row.

If merely inspecting the file does not identify the problem then you should try the following troubleshooting technique.

- Copy just the first few lines of the file into a test file.
- Load the test file and inspect the resulting waves in a table.
- Open the test file as a notebook.
- Edit the file to eliminate any irregularities, save it and load it again. Note that you can load a file as delimited text even if it is open as a notebook. Make sure that you have saved changes to the notebook before loading it.
- Inspect the loaded waves again.

This process usually sheds some light on what aspect of the file is irregular. Working on a small subset of your file makes it easier to quickly do some trial and error investigation.

If you are unable to get to the bottom of the problem, email a small segment of the file to support@wavemetrics.com along with a description of the problem. Do not send the segment as plain text because email programs may strip out or replace unusual control characters in the file. Instead, send a compressed version of the file.

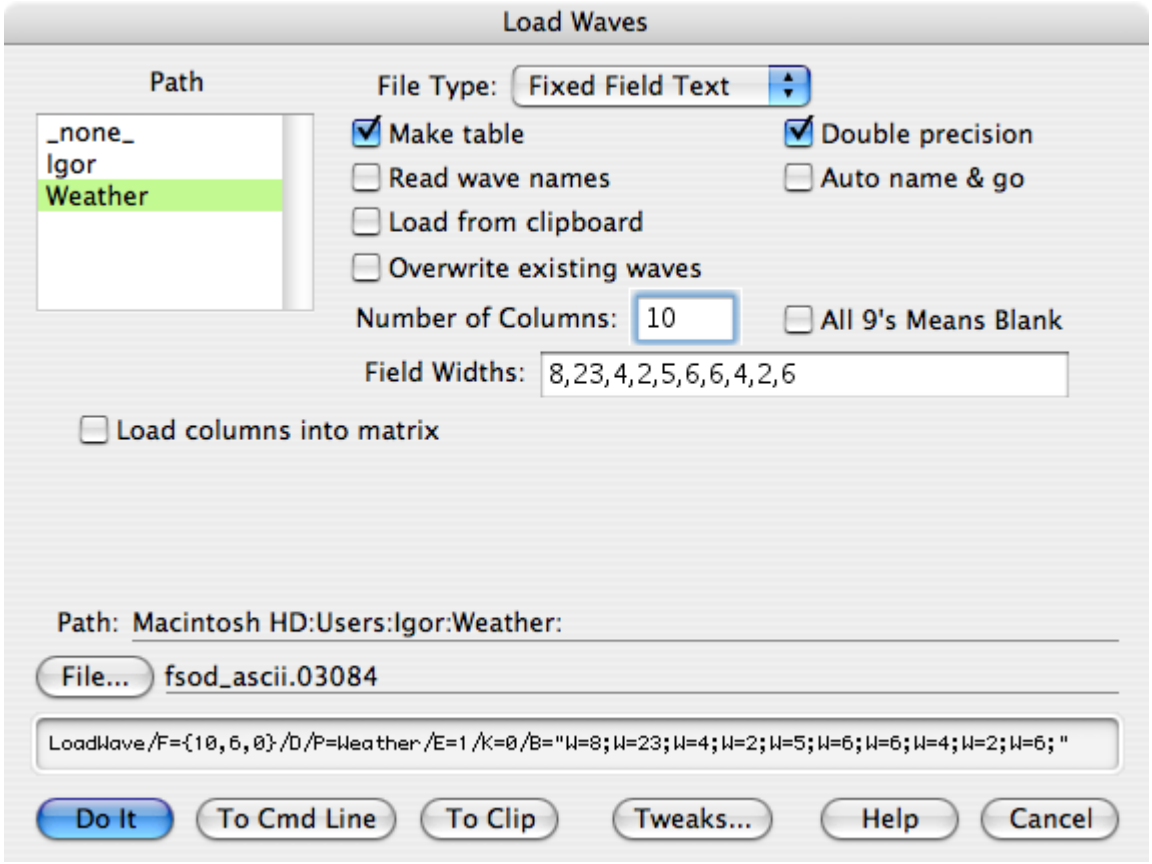
Loading Fixed Field Text Files

A fixed field text file consists of rows of values, organized into columns, that are a fixed number of characters wide with a carriage return, linefeed, or carriage return/linefeed combination at the end of the row. Space characters are used as padding to ensure that each column has the appropriate number of characters. In some cases, a value will fill the entire column and there will be no spaces after it. FORTRAN programs typically generate fixed field text files.

Igor's Load Fixed Field Text routine works just like the Load Delimited Text routine except that, instead of looking for a delimiter character to determine where a column ends, it counts the number of characters in the column. All of the features described in the section **Loading Delimited Text Files** on page II-143 apply also to loading fixed field text.

The Load Waves Dialog for Fixed Field Text

To load a fixed field text file, invoke the Load Waves dialog by choosing the Load Waves menu item.



The dialog is the same as for loading delimited text except for three additional items.

In the Number of Columns item, you must enter the total number of columns in the file. In the Field Widths item, you must enter the number of characters in each column of the file, separated by commas. The last value that you enter is used for any subsequent columns in the file. If all columns in the file have the same number of characters, just enter one number.

If you select the All 9's Means Blank checkbox then Igor will treat any column that consists entirely of the digit 9 as a blank. If the column is being loaded into a numeric wave, Igor sets the corresponding wave value to NaN. If the column is being loaded into a text wave, Igor sets the corresponding wave value to "" (empty string).

Loading General Text Files

We use the term "general text" to describe a text file that consists of one or more blocks of numeric data. A block is a set of rows and columns of numbers. Numbers in a row are separated by one or more tabs or spaces. One or more consecutive commas are also treated as white space. A row is terminated by a carriage return character, a linefeed character, or a carriage return/linefeed combination.

The Load General Text routine handles numeric data only, not date, time, date/time or text. Use Load Delimited Text or Load Fixed Field Text for these formats. Load General Text can handle 2D numeric data as well as 1D.

The first block of data may be preceded by header information which the Load General Text routine will automatically skip.

If there is a second block, it is usually separated from the first with one or more blank lines. There may also be header information preceding the second block which Igor will also skip.

When loading 1D data, the Load General Text routine loads each column of each block into a separate wave. It treats column labels as described above for the Load Delimited Text routine, except that spaces as well as tabs and commas are accepted as delimiters. When loading 2D data, it loads all columns into a single 2D wave.

The Load General Text routine determines where a block starts and ends by counting the number of numbers in a row. When it finds two rows with the same number of numbers, it considers this the start of a block. The block continues until a row which has a different number of numbers.

Examples of General Text

Here are some examples of text that you might find in a general text file.

Simple general text

```
ch0         ch1         ch2         ch3         (optional row of labels)
2.97055     1.95692     1.00871     8.10685
3.09921     4.08008     1.00016     7.53136
3.18934     5.91134     1.04205     6.90194
```

The Load General Text routine would create four waves with three points each or, if you specify loading as a matrix, a single 3 row by 4 column wave.

General text with header

```
Date: 3/2/93
Sample: P21-3A
ch0         ch1         ch2         ch3         (optional row of labels)
2.97055     1.95692     1.00871     8.10685
3.09921     4.08008     1.00016     7.53136
3.18934     5.91134     1.04205     6.90194
```

The Load General Text routine would automatically skip the header lines (Date: and Sample:) and would create four waves with three points each or, if you specify loading as a matrix, a single 3 row by 4 column wave.

General text with header and multiple blocks

```
Date: 3/2/93
Sample: P21-3A
ch0_1       ch1_1       ch2_1       ch3_1       (optional row of labels)
2.97055     1.95692     1.00871     8.10685
3.09921     4.08008     1.00016     7.53136
3.18934     5.91134     1.04205     6.90194

Date: 3/2/93
Sample: P98-2C
ch0_2       ch1_2       ch2_2       ch3_2       (optional row of labels)
2.97055     1.95692     1.00871     8.10685
3.09921     4.08008     1.00016     7.53136
3.18934     5.91134     1.04205     6.90194
```

The Load General Text routine would automatically skip the header lines and would create eight waves with three points each or, if you specify loading as a matrix, two 3 row by 4 column waves.

Comparison of General Text, Fixed Field and Delimited Text

You may wonder whether you should use the Load General Text routine, Load Fixed Field routine or the Load Delimited Text routine. Most commercial programs create simple tab-delimited files which these routines can handle. Files created by scientific instruments, mainframe programs, custom programs, or exported from spreadsheets are more diverse. You may need to try these routines to see which works better. To help you decide which to try first, here is a comparison.

Advantages of the Load General Text compared to Load Fixed Field and to Load Delimited Text:

- It can automatically skip header text.

- It can load multiple blocks from a single file.
- It can tolerate multiple tabs or spaces between columns.

Disadvantages of the Load General Text compared to Load Fixed Field and to Load Delimited Text:

- It can not handle blanks (missing values).
- It can not tolerate columns of nonnumeric text or nonnumeric values in a numeric column.
- It can not load text values, dates, times or date/times.
- It can not handle comma as the decimal point (European number style).

The Load General Text routine *can* load missing values if they are represented in the file explicitly as "NaN" (Not-a-Number). It can not handle files that represent missing values as blanks because this confounds the technique for determining where a block of numbers starts and ends.

The Load Waves Dialog for General Text — 1D

To load a general text file as 1D waves, invoke the Load Waves dialog by choosing the Load Waves menu item. The dialog appears as shown above for delimited text.

The basic process of loading data from a general text file is as follows:

1. Bring up the Load Waves dialog.
2. Choose General Text from the File Type pop-up menu.
3. Click the File button to select the file containing the data.
4. Click Do It.

When you click Do It, Igor's LoadWave operation runs. It executes the Load General Text routine which goes through the following steps:

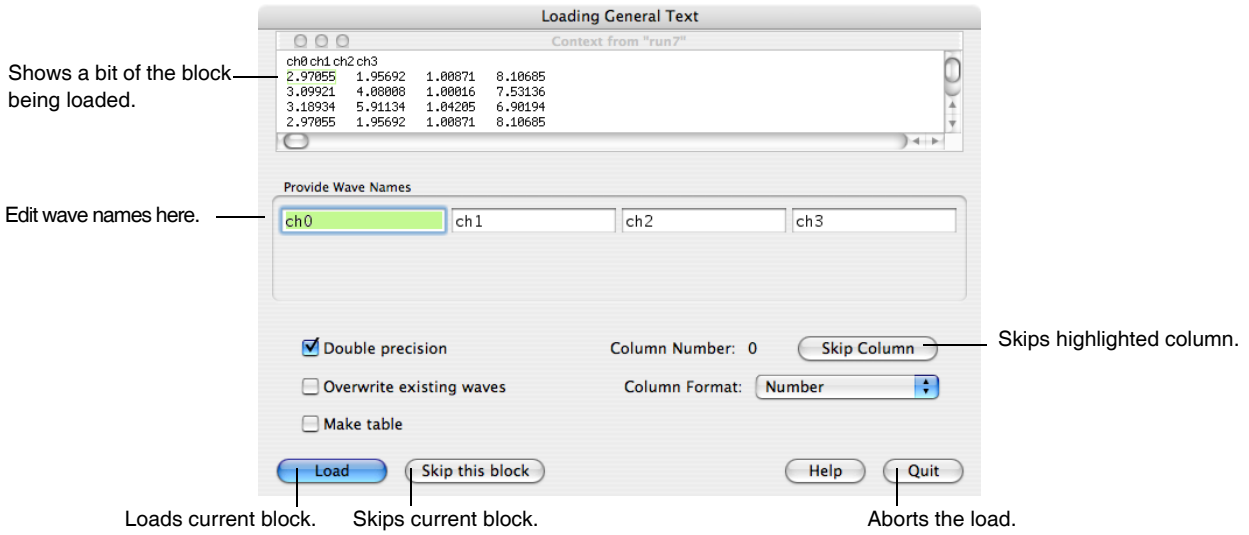
1. Locate the start of the block of data using the technique of counting numbers in successive lines. This step also skips the header, if any, and determines the number of columns in the block.
2. Optionally, determine if there is a row of column labels immediately before the block of numbers.
3. Optionally, present another dialog allowing you to confirm or change wave names.
4. Create waves.
5. Load data into the waves until the end of the file or a until a row that contains a different number of numbers.
6. If not at the end of the file, go back to step one to look for another block of data.

Igor looks for a row of column labels only if you enable the "Read wave names" option. It looks in the line immediately preceding the block of data. If it finds labels and if the number of labels matches the number of columns in the block, it uses these labels as wave names. Otherwise, Igor will automatically generate wave names of the form wave0, wave1 and so on.

If you choose the Load General Text item from the Load Waves submenu instead of the Load Waves item, Igor will display a dialog from which you can select the general text file to load directly. This is a shortcut that skips the Load Waves dialog and uses default options for the load. This will always load 1D waves, not a matrix. Before you use this shortcut, take a look at the Load Waves dialog so you can see what options are available.

Editing Wave Names for a Block

In step 3 above, the Load General Text routine presents a dialog in which you can change wave names. This works exactly as described above for the Load Delimited Text routine except that it has one extra button: "Skip this block".



Use "Skip this block" to skip one or more blocks of a multiple block general text file.

Click the Skip Column button to skip loading of the column corresponding to the selected name box. Shift-click the button to skip all columns except the selected one.

The Load Waves Dialog for General Text — 2D

Igor can load a 2D wave using the Load General Text routine. However, Load General Text does not support the loading of row/column labels and positions. If the file has such rows and columns, you must load it as a delimited text file.

The main reason to use the Load General Text routine rather than the Load Delimited Text routine for loading a matrix is that the Load General Text routine can automatically skip nonnumeric header information. Also, Load General Text treats any number of spaces and tabs, as well as one comma, as a single delimiter and thus is tolerant of less rigid formatting.

Set Scaling After Loading General Text Data

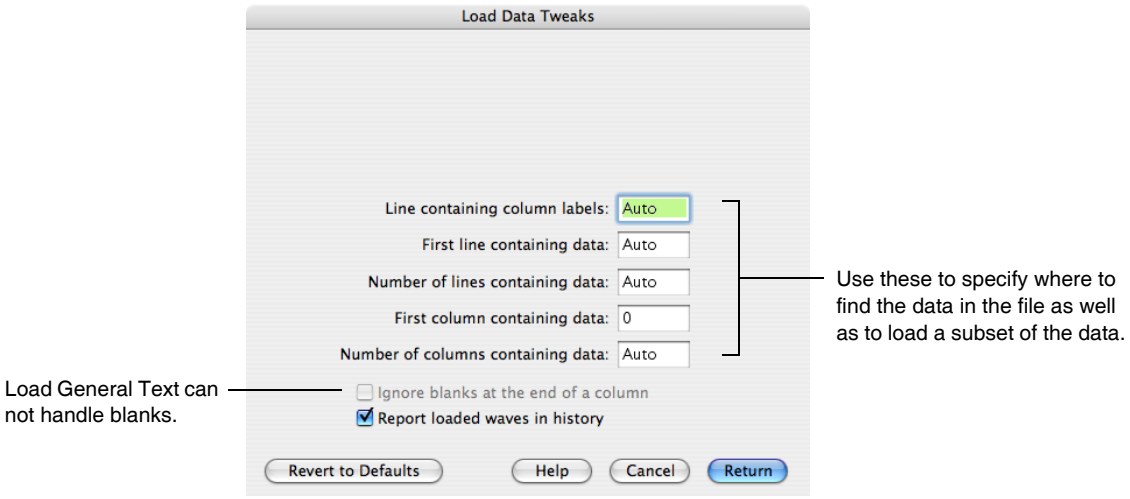
If your 1D data is uniformly spaced in the X dimension then you will be able to use the many operations and functions in Igor designed for waveform data. You will need to set the X scaling for your waves after you load them, using the Change Wave Scaling dialog.

Note: If your data is uniformly spaced it is *very important* that you set the X scaling of your waves. Many Igor operations depend on the X scaling information to give you correct results.

If your 1D data is not uniformly spaced then you will use XY pairs and you do not need to change X scaling. You may want to use Change Wave Scaling to set the waves' data units.

General Text Tweaks

The Load General Text routines provides some tweaks that allow you to guide Igor as it loads the file. To do this, use the Tweaks button in the Load Waves dialog. Most people will not need to use these tweaks.



The items at the top of the dialog are hidden because they apply to the Load Delimited Text routine only. Load General Text always skips any tabs and spaces between numbers and will also skip a single comma. The "decimal point" character is always period and it can not handle dates.

The items relating to column labels, data lines and data columns have two potential uses. You can use them to load just a part of a file or to guide Igor if the automatic method of finding a block of data produces incorrect results.

Lines and columns in the tweaks dialog are numbered starting from zero.

Igor interprets the "Line containing column labels" and "First line containing data" tweaks differently for general text files than it does for delimited text files. For delimited text, zero means "the first line". For general text, zero for these parameters means "auto".

Here is what "auto" means for general text. If "First line containing data" is auto, Igor starts the search for data from the beginning of the file without skipping any lines. If it is not "auto", then Igor skips to the specified line and starts its search for data there. This way you can skip a block of data at the beginning of the file. If "Line containing column labels" is auto then Igor looks for column labels in the line immediately preceding the line found by the search for data. If it is not auto then Igor looks for column labels in the specified line.

If the "Number of lines containing data" is not "auto" then Igor will stop loading after the specified number of lines or when it hits the end of the first block, whichever comes first. This behavior is necessary so that it is possible to pick out a single block or subset of a block from a file containing more than one block.

If a general text file contains more than one block of data and if "Number of lines containing data" is "auto" then, for blocks after the first one, Igor maintains the relationship between the line containing column labels and first line containing data. Thus, if the column labels in the first block were one line before the first line containing data then Igor will expect the same to be true of subsequent blocks.

You can use the "First column containing data" and "Number of columns containing data" tweaks to load a subset of the columns in a block. If "Number of columns containing data" is set to "auto" or 0, Igor will load all columns until it hits the last column in the block.

Troubleshooting General Text Files

You can examine the waves created by the Load General Text routine using a table. If you don't get the results that you expected, you will need to inspect and edit the text file until it is in a form that Igor can handle. Remember the following points:

- Load General Text can not handle dates, times, date/times, commas used as decimal points, or blocks of data with nonnumeric columns. Try Load Delimited Text for this.
- It skips any tabs or spaces between numbers and will also skip a single comma.

- It expects a line of column labels, if any, to appear in the first line before the numeric data unless you set tweaks to the contrary. It expects that the labels are also delimited by tabs, commas or spaces. It will not look for labels unless you enable the Read Wave Names option.
- It works by counting the number of numbers in consecutive lines. Some unusual formats (e.g., 1,234.56 instead of 1234.56) can throw this count off, causing it to start a new block prematurely.
- It can not handle blanks or nonnumeric values in a column. Each of these will start a new block of data.
- If it detects a change in the number of columns, it starts loading a new block into a new set of waves.

If merely inspecting the file does not identify the problem then you should try the technique of loading a subset of your data. This is described under **Troubleshooting Delimited Text Files** on page II-152 and often sheds light on the problem. In the same section, you will find instructions for sending the problem file to WaveMetrics for analysis, if necessary.

Loading Igor Text Files

An Igor Text file consists of keywords, data and Igor commands. The data can be numeric, text or both and can be of dimension 1 to 4. Many Igor users have found this to be an easy and powerful way to import data from their own custom programs into Igor.

The file name extension for an Igor Text file is ".itx". Old versions of Igor used ".awav" and this is still accepted.

Examples of Igor Text

Here are some examples of text that you might find in an Igor Text file.

Simple Igor Text

```
IGOR
WAVES/D unit1, unit2
BEGIN
      19.7  23.9
      19.8  23.7
      20.1  22.9
END
X SetScale x 0,1, "V", unit1; SetScale d 0,0, "A", unit1
X SetScale x 0,1, "V", unit2; SetScale d 0,0, "A", unit2
```

Loading this would create two double-precision waves named unit1 and unit2 and set their X scaling, X units and data units.

Igor Text with extra commands

```
IGOR
WAVES/D/O xdata, ydata
BEGIN
      98.822      486.528
      109.968     541.144
      119.573     588.21
      133.178     654.874
      142.906     702.539
END
X SetScale d 0,0, "V", xdata
X SetScale d 0,0, "A", ydata
X Display ydata vs xdata; DoWindow/C TempGraph
X ModifyGraph mode=2,lsize=5
X CurveFit line ydata /X=xdata /D
X Textbox/A=LT/X=0/Y=0 "ydata= \\{W_coef[0]}+\\{W_coef[1]}*xdata"
X PrintGraphs TempGraph
X DoWindow/K TempGraph              // kill the graph
X KillWaves xdata, ydata, fit_ydata // kill the waves
```

Loading this would create two double-precision waves and set their data units. It would then make a graph, do a curve fit, annotate the graph and print the graph. The last two lines do housekeeping.

Igor Text File Format

An Igor Text file starts with the keyword **IGOR**. The rest of the file may contain blocks of data to be loaded into waves or Igor commands to be executed and it must end with a blank line.

A block of data in an Igor Text file must be preceded by a declaration of the waves to be loaded. This declaration consists of the keyword **WAVES** followed by optional flags and the names of the waves to be loaded. Next the keyword **BEGIN** indicates the start of the block of data. The keyword **END** marks the end of the block of data.

A file can contain any number of blocks of data, each preceded by a declaration. If the waves are 1D, the block can contain any number of waves but waves in a given block must all be of the same data type. Multidimensional waves must appear one wave per block.

A line of data in a block consists of one or more numeric or text items with tabs separating the numbers and a carriage return at the end of the line. Each line should have the same number of items.

You can't use blanks, dates, times or date/times in an Igor Text file. To represent a missing value in a numeric column, use "NaN" (not-a-number). To represent dates or times, use the standard Igor date format (number of seconds since 1/1/1904).

There is no limit to the number of waves or number of points except that all of the data must fit in available memory.

The WAVES keyword accepts the following optional flags:

Flag	Effect
/N=(…)	Specifies size of each dimension for multidimensional waves.
/O	Overwrites existing waves.
/R	Makes waves real (default).
/C	Makes waves complex.
/S	Makes waves single precision floating point (default).
/D	Makes waves double precision floating point.
/I	Makes waves 32 bit integer.
/W	Makes waves 16 bit integer.
/B	Makes waves 8 bit integer.
/U	Makes integer waves unsigned.
/T	Specifies text data type.

Normally you should make single or double precision floating point waves. Integer waves are normally used only to contain raw data acquired via external operations. They are also appropriate for storing image data.

The /N flag is needed only if the data is multidimensional but the flag is allowed for one-dimensional data, too. Regardless of the dimensionality, the dimension size list must always be inside parentheses. Examples:

```
WAVES/N=(5) wave1D
WAVES/N=(3,3) wave2D
WAVES/N=(3,3,3) wave3D
```

Integer waves are signed unless you use the /U flag to make them unsigned.

If you use the /C flag then a pair of numbers in a line supplies the real and imaginary value for a single point in the resulting wave.

If you specify a wave name that is already in use and you don't use the overwrite option, Igor will display a dialog so that you can resolve the conflict.

The /T flag makes text rather than numeric waves. See **Loading Text Waves from Igor Text Files** on page II-161.

A command in an Igor Text file is introduced by the keyword **X** followed by a space. The command follows the X on the same line. When Igor encounters this while loading an Igor Text file it executes the command.

Anything that you can execute from Igor's command line is acceptable after the X. Introduce comments with "X //". There is no way to do conditional branching or looping. However, you can call an Igor procedure defined in a built-in or auxiliary procedure window.

Commands, introduced by X, are executed as if they were entered on the command line or executed via the Execute operation. Such command execution is not thread-safe. Therefore, you can not load an Igor text file containing a command from an Igor thread.

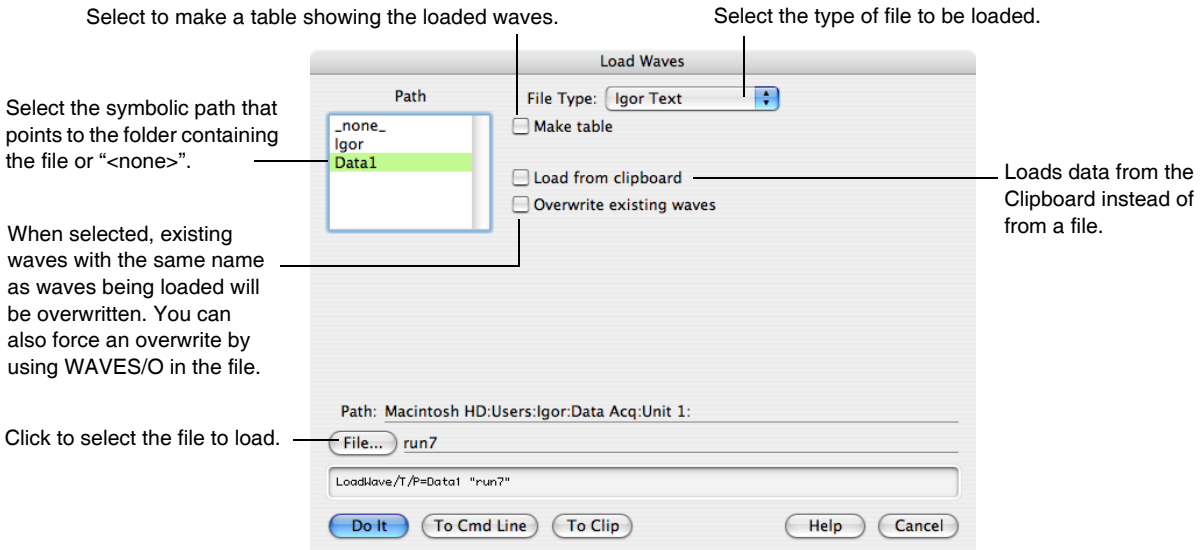
Setting Scaling in an Igor Text File

When Igor writes an Igor Text file, it always includes commands to set each wave's scaling, units and dimension labels. It also sets each wave's note.

If you write a program that generates Igor Text files, you should set at least the scaling and units. If your 1D data is uniformly spaced in the X dimension, you should use the SetScale operation to set your waves X scaling, X units and data units. If your data is not uniformly spaced, you should set the data units only. For multidimensional waves, use SetScale to set Y, Z and T units if needed.

The Load Waves Dialog for Igor Text

To load an Igor Text file, invoke the Load Waves dialog by choosing the Load Waves menu item.



The basic process of loading data from an Igor Text file is as follows:

1. Bring up the Load Waves dialog.
2. Choose Igor Text from the File Type pop-up menu.
3. Click the File button to select the file containing the data.
4. Click Do It.

When you click Do It, Igor's LoadWave operation runs. It executes the Load Igor Text routine which loads the file.

If you choose the Load Igor Text item from the Load Waves submenu instead of the Load Waves item, Igor will display a dialog from which you can select the Igor Text file to load directly. This is a shortcut that skips the Load Waves dialog.

Loading MultiDimensional Waves from Igor Text Files

In an Igor Text file, a block of wave data is preceded by a WAVES declaration. For multidimensional data, you must use a separate block for each wave. Here is an example of an Igor Text file that defines a 2D wave:

```
IGOR
WAVES/D/N=(3,2) wave0
BEGIN
      1     2
      3     4
      5     6
END
```

The "/N=(3,2)" flag specifies that the wave has three rows and two columns. The first line of data (1 and 2) contains data for the first row of the wave. This layout of data is recommended for clarity but is not required. You could create the same wave with:

```
IGOR
WAVES/D/N=(3,2) wave0
BEGIN
      1     2      3      4      5      6
END
```

Igor merely reads successive values and stores them in the wave, storing a value in each column of the first row before moving to the second row. All white space (spaces, tabs, return and linefeed characters) are treated the same.

When loading a 3D wave, Igor expects the data to be in column/row/layer order. You can leave a blank line between layers for readability but this is not required.

Here is an example of a 3 rows by 2 columns by 2 layers wave:

```
IGOR
WAVES/D/N=(3,2,2) wave0
BEGIN
      1     2
      3     4
      5     6

      11    12
      13    14
      15    16
END
```

The first 6 numbers define the values of the first layer of the 3D wave. The second 6 numbers define the values of the second layer.

When loading a 4D wave, Igor expects the data to be in column/row/layer/chunk order. You can leave a blank line between layers and two blank lines between chunks for readability but this is not required.

If loading a multidimensional wave, Igor expects that the dimension sizes specified by the /N flag are accurate. If there is more data in the file than expected, Igor ignores the extra data. If there is less data than expected, some of the values in the resulting waves will be undefined. In either of these cases, Igor will print a message in the history area to alert you to the discrepancy.

Loading Text Waves from Igor Text Files

Loading text waves from Igor Text files is similar to loading them from delimited text files except that in an Igor Text file you declare a wave's name and type. Also, text strings are quoted in Igor Text files as they are in Igor's command line. Here is an example of Igor Text that defines a text wave:

```
IGOR
WAVES/T textWave0, textWave1
BEGIN
      "This"      "Hello"
      "is"        "out"
      "a test"    "there"
END
```

All of the waves in a block of an Igor Text file must have the same number of points and data type. Thus, you can not mix numeric and text waves in the same block. You can have any number of blocks in one Igor Text file.

As this example illustrates, you must use double quotes around each string in a block of text data. If you want to embed a quote, tab, carriage return or linefeed within a single text value, use the escape sequences \", \t, \r or \n. Use \\ to embed a backslash. For less common escape sequences, see **Escape Characters in Strings** on page IV-13.

The Igor Text File Type Code and File Extension

On *Macintosh*, Igor recognizes files of type IGTX as Igor Text. The file type can also be TEXT. If you are writing a program that generates Igor text files, use file type IGTX, creator code IGR0 (last character is zero) and the file name extension ".itx".

On *Windows*, just use the file name extension ".itx".

Loading UTF-16 Files

The LoadWave operation can load data from UTF-16 (two-byte Unicode) text files. It does not recognize non-ASCII characters, but does ignore the byte-order mark at the start of the file (BOM) and null bytes contained in UTF-16 text files. Consequently it can load data from UTF-16 files containing just numeric data and ASCII text.

Loading Igor Binary Data

This section discusses loading Igor Binary data into memory. Igor stores Igor Binary data in two ways: one wave per Igor Binary file in unpacked experiments and multiple waves within a packed experiment file.

When you open an experiment, Igor *automatically* loads the Igor Binary data to recreate the experiment's waves. The main reason to *explicitly* load an Igor Binary file is if you want to load data from another program that knows how to create an Igor Binary file. The easiest way to load data from another experiment is to use the Data Browser (see **Data Browser** on page II-130).

Warning: You can get into trouble if two Igor experiments load data from the same Igor Binary file. See **Sharing Versus Copying Igor Binary Files** on page II-165 for details.

There are a number of ways to load Igor Binary data into the current experiment in memory. Here is a summary. For most users, the first and second methods — which are simple and easy to use — are sufficient.

Method	Loads	Action	Purpose
Open Experiment (Chapter II-3)	Packed and unpacked files	Restores experiment to the state in which it was last saved.	To restore experiment.
Data Browser (Chapter II-8)	Packed and unpacked files	Copies data from one experiment to another.	To collect data from different sources for comparison.
Browse Waves Dialog (Chapter II-5)	Unpacked files only	Copies data from one experiment to another or shares between experiments.	To collect data from different sources for comparison.

Method	Loads	Action	Purpose
Desktop Drag and Drop (Chapter II-3)	Unpacked files only	Copies data from one experiment to another or shares between experiments.	To collect data from different sources for comparison.
Load Waves Dialog	Unpacked files only	Copies data from one experiment to another or shares between experiments.	To create a LoadWave command that can be used in an Igor procedure.
LoadWaves Operation	Unpacked files only	Copies data from one experiment to another or shares between experiments.	To automatically load data using an Igor Procedure.
LoadData Operation	Packed and unpacked files	Copies data from one experiment to another.	To automatically load data using an Igor Procedure.

The Igor Binary File

The Igor Binary file format is Igor's native format for storing waves. This format stores one wave per file very efficiently. The file includes the numeric contents of the wave (or text contents if it is a text wave) as well as all of the auxiliary information such as the dimension scaling, dimension and data units and the wave note. In an Igor packed experiment file, any number of Igor Binary wave files can be packed into a single file.

The file name extension for an Igor Binary file is ".ibw". Old versions of Igor used ".bwav" and this is still accepted. The Macintosh file type code is IGBW and the creator code is IGR0 (last character is zero).

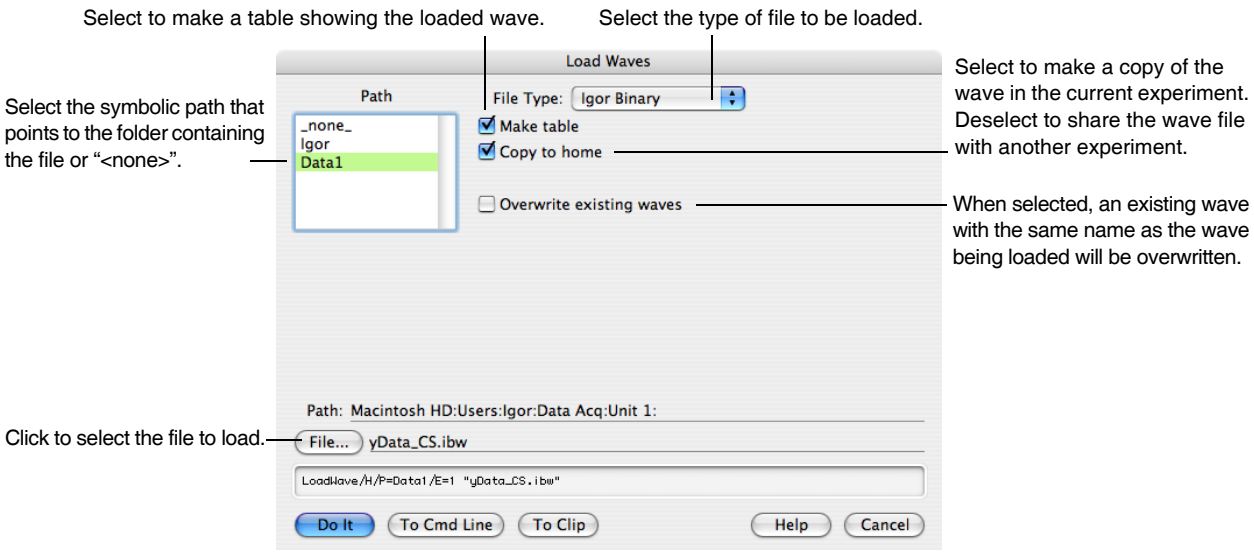
The name of the wave is stored *inside* the Igor Binary file. It does not come from the name of the file. For example, wave0 might be stored in a file called "wave0.ibw". You could change the name of the file to anything you want. This does not change the name of the wave stored in the file.

The Igor Binary file format was designed to save waves that are part of an Igor experiment. In the case of an unpacked experiment, the Igor Binary files for the waves are stored in the experiment folder and can be loaded using the LoadWave operation. In the case of a packed experiment, data in Igor Binary format is packed into the experiment file and can be loaded using the LoadData operation.

Some Igor users have written custom programs that write Igor Binary files which they load into an experiment. Igor Technical Note #003, "Igor Binary Format", provides the details that a programmer needs to do this. See also Igor Pro Technical Note PTN003.

The Load Waves Dialog for Igor Binary

To load an Igor Binary file, invoke the Load Waves dialog by choosing the Load Waves menu item.



The basic process of loading data from an Igor Binary file is as follows:

1. Bring up the Load Waves dialog.
2. Choose Igor Binary from the File Type pop-up menu.
3. Click the File button to select the file containing the data.
4. Set the "Copy to home" checkbox.
5. Click Do It.

When you click Do It, Igor's LoadWave operation runs. It executes the Load Igor Binary routine which loads the file. If the wave that you are loading has the same name as an existing wave or other Igor object, Igor will present a dialog in which you can resolve the conflict.

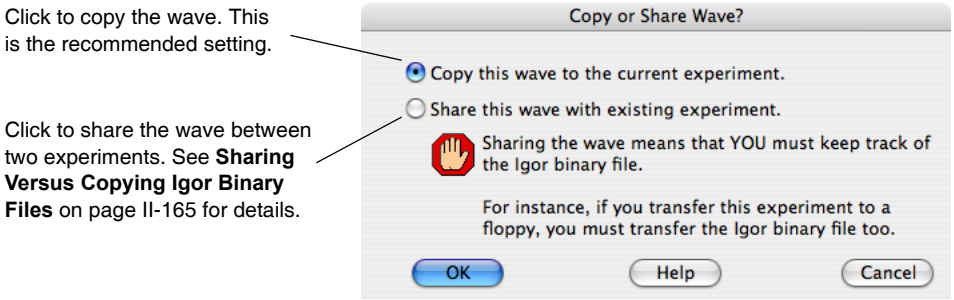
Notice the "Copy to home" checkbox. It is very important.

If it is selected, Igor will disassociate the wave from its source file after loading it into the current experiment. When you next save the experiment, Igor will store a new copy of the wave with the current experiment. The experiment will not reference the original source file. We call this "copying" the wave to the current experiment.

If "Copy to home" is not selected, Igor will keep the connection between the wave and the file from which it was loaded. When you save the experiment, it will contain a *reference* to the source file. We call this "sharing" the wave between experiments.

We strongly recommend that you copy waves rather than share them. See **Sharing Versus Copying Igor Binary Files** on page II-165 for details.

If you choose the Load Igor Binary item from the Load Waves submenu instead of the Load Waves item, Igor will display a dialog from which you can select the Igor Binary file to load directly. This is a shortcut that skips the Load Waves dialog. When you take this shortcut, you lose the opportunity to set the "Copy to home" checkbox. Thus, during the load operation, Igor will present a dialog from which you can choose to copy or share the wave.



The LoadData Operation

The LoadData operation provides a way for Igor programmers to automatically load data from packed Igor experiment files or from a file-system folder containing unpacked Igor Binary files. It can load not only waves but also numeric and string variables and a hierarchy of data folders that contains waves and variables.

The Data Browser's Browse Expt button provides interactive access to the LoadData operation and permits you to drag a hierarchy of data from one Igor experiment into the current experiment in memory. To achieve the same functionality in an Igor procedure, you need to use the LoadData operation directly. See the **LoadData** operation (see page V-344).

LoadData, accessed from the command line or via the Data Browser, has the ability to overwrite existing waves, variables and data folders. Igor automatically updates any graphs and tables displaying the overwritten waves. This provides a very powerful and easy way to view sets of identically structured data, as would be produced by successive runs of an experiment. You start by loading the first set and create graphs and tables to display it. Then, you load successive sets of identically named waves. They overwrite the preceding set and all graphs and tables are automatically updated.

Sharing Versus Copying Igor Binary Files

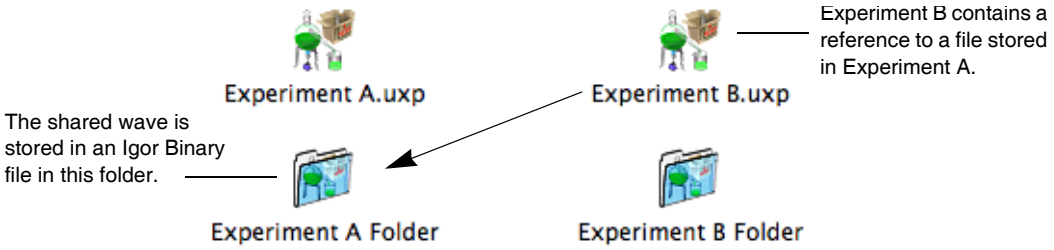
There are two reasons for loading a binary file that was created as part of another Igor experiment: you may want your current experiment to *share* data with the other experiment or, you may want to *copy* data to the current experiment from the other experiment.

There is a potentially serious problem that occurs if two experiments share a file. The file can not be in two places at one time. Thus, it will be stored *with* the experiment that created it but *separate from* the other. The problem is that, if you move or rename files or folders, the second experiment will be unable to find the binary file.

Here is an example of how this problem can bite you.

Imagine that you create an experiment at work and save it as an unpacked experiment file on your hard disk. Let's call this "experiment A". The waves for experiment A are stored in individual Igor Binary files in the experiment folder.

Now you create a new experiment. Let's call this "experiment B". You use the Load Igor Binary routine to load a wave from experiment A into experiment B. You elect to share the wave. You save experiment B on your hard disk. Experiment B now contains a *reference* to a file in experiment A's home folder.



Now you decide to take experiment B to another computer. You copy it to a CD and go to the other computer. When you try to open experiment B, Igor can't find the file it needs to load the shared wave. This file is back on the hard disk of the original computer.

A similar problem occurs if, instead of moving experiment B to another computer, you change the name or location of experiment A's folder. Experiment B will still be looking for the shared file under its old name or in its old location and Igor will not be able to load the file when you open experiment B.

Because of this problem, we recommend that you *avoid file sharing* as much as possible. If it is necessary to share a binary file, you will need to be very careful to avoid the situation described above.

The Data Browser always copies when transferring data from disk into memory.

For more information on the problem of sharing files, see **References to Files and Folders** on page II-37.

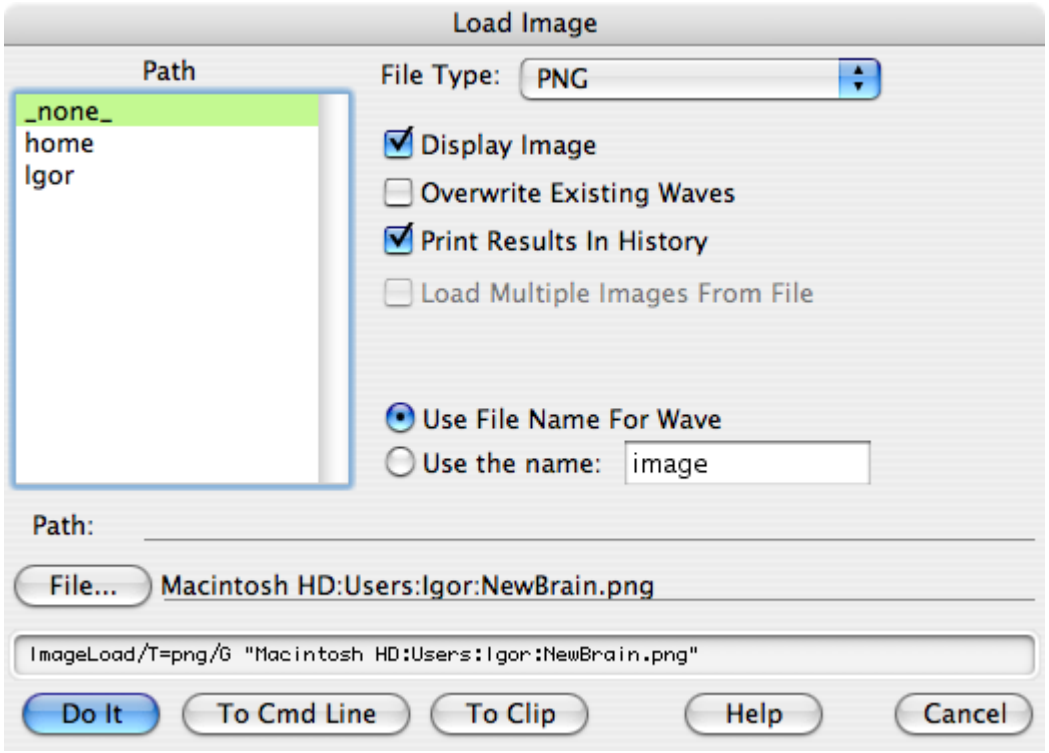
Loading Image Files

You can load PICT, TIFF, JPEG, PNG, GIF, Photoshop, SGI, Sun Raster, BMP, and Targa image files into Igor Pro using the Load Image dialog. The same file types are supported both on the Macintosh and on Windows.

Loading the following types requires that you have Apple's QuickTime software installed on your computer: PICT (on Windows only), JPEG, GIF, PhotoShop, SGI and Targa. All Mac OS X machines have QuickTime installed. Windows users who want to load these types of files can download QuickTime from <http://www.apple.com/quicktime/>.

The Load Image Dialog

To load an image file into an Igor matrix wave, invoke the Load Image dialog by choosing the Load Image menu item in the Load Waves submenu.



This dialog looks and works much the same as the other Igor file loading dialogs.

When you choose a particular type of image file from the File Type pop-up menu, you are setting a file filter that is used when displaying the image file selection dialog. If you are not sure that your image file has the correct file type or file name extension, choose "Any" from the File pop-up menu so that the filter does not restrict your selection. Note that when you choose "Any" QuickTime will be used to load the file and therefore you can only load images from file formats supported by QuickTime.

Names for the loaded matrix waves can be the name of the file or a name that you specify. If you enter a matrix wave name in the dialog that conflicts with an existing wave name and you have not selected the Overwrite Existing Waves checkbox, Igor will append a numeric suffix to the new wave names.

Image Loading Details

Except for certain kinds of TIFF and Sun Raster files, images are loaded into a 3D RGB, RGBA, or CMYK wave. See the **ImageLoad** operation (see page V-269) for further details.

The wave is of type unsigned byte with layer 0 containing the red channel, layer 1 the green channel and layer 2 the blue channel. The wave may contain four layers if you load a CMYK image or if you load an image that has an alpha channel in addition to the RGB information. Grayscale TIFF and Sun Raster images are loaded as 2D waves. If you load a TIFF or Sun Raster image that contains a colormap, Igor creates (in addition to the image wave) a colormap wave (usually with the suffix "_CMap"). You can display images using the NewImage command or convert image waves into other forms using the ImageTransform operation.

There are two menu choices for the PNG format: Raw PNG and PNG. When Raw PNG is selected, the data is read directly from the file into the wave. When PNG is selected, the file is loaded into memory, and offscreen image is created, and the wave data is set by reading the offscreen image. In nearly all cases, you should choose Raw PNG.

When you choose TIFF from the File Type pop-up menu, an additional checkbox appears: Load Multiple Images From File. If your TIFF file contains a stack of images, select this checkbox. You can then set the number of the first image to load (zero-based) and the number of images to load from the TIFF stack.

If just one image is loaded from the TIFF file then Igor creates a single 2D wave. If more than one image is loaded, Igor creates a single 3D wave, each layer of which contains the data from one of the images in the stacked TIFF file. Reading a TIFF image stack into a single 3D wave is supported only for images that are 8, 16 or 32- bits/pixel deep.

You can convert a number of 2D image waves into a 3D stack using the ImageTransform operation (stackImages keyword).

HDF images can be loaded only by the HDF or HDF5 XOPs, see **Loading HDF Data** on page II-169 for further details.

Loading Other Files

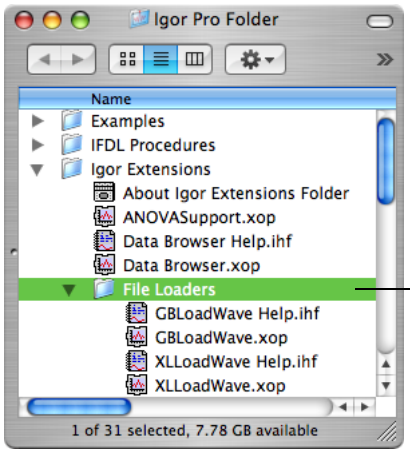
WaveMetrics provides a number of extensions that add additional file-loading capabilities to Igor. Most of these file loaders add a menu item to the Load Waves submenu and an entry in the Open or Load File Dialog's list so you can use it interactively. They also usually add a command line operation so you can use them from an Igor procedure.

The following table lists many of the file loaders included with Igor Pro. Some more obscure file loaders are also available..

File Loader/Writer	Description
GBLoadWave	Loads numeric data from "general binary" files. The XOP can load 8, 16 and 32 bit integer data and 32 and 64 bit IEEE data from a binary file. It can also load a subset of the file. It can handle numerous kinds of files including interleaved and byte-swapped files. You must know the format of the binary file precisely.
GISLoadWave	Loads Digital Elevation Model (DEM) and Digital Line Graph (DLG) data for standard U.S. Geological Survey (USGS) format quadrangles. Such geographic data are the basic elements of digital mapping.
GWLoadWave	Loads an old Macintosh-only file format from GW Instruments.
HDF Loader	Loads HDF (Hierarchical Data Format) version 4 and earlier files. See **Loading HDF Data** on page II-169.
HDF5 XOP	Loads HDF version 5 files. See **Loading HDF Data** on page II-169.
JCAMPLoadWave	Loads JCAMP files, used in spectroscopy.
LoadWAVfile	Windows only. Adds operations to load and save WAV sound files.
MLLoadWave	Loads data from Matlab binary files. WaveMetrics thanks Yves Peysson and Bernard Saoutic for this file loader.
NILoadWave	Loads numeric data from files produced by a number of scientific instruments from Nicolet Instruments.
SndLoadSaveWave	Loads a variety of sound files on Macintosh and Windows.
TDM XOP	Loads data from National Instruments TDM files.
XLLoadWave	Loads numeric and text data from an Excel spreadsheet file. You need to know the cells containing the numeric data, for example, B10 - D25.

If you are a C programmer, you can write your own extension to load data into Igor. To do this you need the Igor External Operations Toolkit, available from WaveMetrics.

The Igor installer puts file loaders and other extensions in "Igor Pro Folder/Igor Extensions" and "Igor Pro Folder/More Extensions". To use an extension, put an alias (*Macintosh*) or shortcut (*Windows*) for it in "Igor Pro User Files/Igor Extensions" (see **Igor Pro User Files** on page II-46 for details) and then relaunch Igor.



Put file loader extensions, or aliases for them, anywhere inside the Igor Extensions folder. If you'd like, you can make a File Loaders subfolder.

Each file loader has an associated Igor help file. The help file provides all the information you need to use the file loader.

Loading Non-TEXT Files as TEXT Files

The plain text file is the most common type of file used for transferring data from one program into another. On the *Macintosh*, files have a file type property. A plain text file is supposed to have the file type TEXT. Under Windows, text files often have the file name extensions ".txt" or ".dat".

Macintosh Files

On a Macintosh, choosing Load Delimited Text, Load General Text or Load Igor Text leads to a dialog initially showing TEXT files as well as files with certain filename extensions. You can choose to show all files by choosing All Documents from the Show pop-up menu. Choosing All Documents will also allow you to navigate into package folders.

Windows Files

Under Windows, choosing Load Delimited Text, Load General Text or Load Igor Text leads to a dialog initially showing files with the extension .txt. You can choose to show files with other common extensions, such as .dat, or you can show all files.

Loading Row-Oriented Text Data

All of the built-in text file loaders are column-oriented — they load the columns of data in the file into 1D waves. There is a row-oriented format that is fairly common. In this format, the file represents data for one wave but is written in multiple columns. Here is an example:

```
350         2.97        1.95        1.00        8.10        2.42
351         3.09        4.08        1.90        7.53        4.87
352         3.18        5.91        1.04        6.90        1.77
```

In this example, the first column contains X values and the remaining columns contain data values, written in row/column order.

Igor Pro does not have a file-loader extension to handle this format, but there is a WaveMetrics procedure file for it. To use it, use the Load Row Data procedure file in the "WaveMetrics Procedures:File Input Output" folder. It adds a Load Row Data item to the Macros menu. When you choose this item, Igor will present a dialog that presents several options. One of the options treats the first column as X values or as data. If you specify treating the column as X values, Igor will use it to determine the X scaling of the output wave, assuming that the values in the first column are evenly spaced. This is usually the case.

Loading HDF Data

HDF stands for "Hierarchical Data Format". HDF is a complex and powerful format and you will need to understand it as well as the structure of your HDF files to conveniently use it. Information on HDF is available via the World Wide Web from:

<http://www.hdfgroup.org/>

The current version of HDF is HDF5. Igor Pro includes an HDF5XOP that can read and write HDF5 files. HDF5XOP is documented in the "HDF5 Help.ihf" file in "Igor Pro Folder:More Extensions:File Loaders". An HDF5 browser based on HDFXOP is also provided and documented in the help file.

Igor Pro also includes an older XOP that supports HDF version 3 and version 4 files. This HDF Loader XOP is documented in "HDF Loader Help.ihf" file in the same folder.

Loading Very Big Binary Files

Binary data files can be loaded using the GBLoadWave operation or the FBinRead operation.

Most binary data files are not so large as to present issues for Igor. However, if your data file approaches hundreds of millions or billions of bytes, size and memory issues arise.

GBLoadWave and FBinRead can handle very large files, up to hundreds of trillions of bytes, in theory. However, other constraints put a limit on the amount of data you can load into Igor.

First there is the maximum amount of virtual memory that Igor can handle on your machine - between 2 and 4 GB. For details, see **Memory Management** on page III-425.

Even if you maximize the amount of virtual memory accessible by Igor, you still need as much physical memory as possible to avoid slowing your computer to a crawl. 2 GB is good, 4 GB is better, if your computer supports it.

Even if Igor can theoretically address 4 GB, this does not mean that you can create a 4 GB wave. You are further limited by memory fragmentation, also discussed under **Memory Management** on page III-425.

If you want GBLoadWave or FBinRead to convert the type of the data, for example from 16-bit signed to 32-bit floating point, this requires an extra buffer during the load process which takes more memory.

Furthermore, Igor itself currently can not create a wave with more than 2 billion points because of the use of signed longs throughout the program.

Finally, there is very little that you can do in Igor with a 1 billion point wave that won't take forever. Consequently you need to load your data a piece at a time using the GBLoadWave or FBinRead.

Some experimentation will be necessary to determine how to deal with very large files. It is a good idea to start with a reasonably-sized chunk of data, say 100 million bytes.

Loading Waves Using Igor Procedures

One of Igor's strong points is that it you can write procedures to automatically load, process and graph data. This is useful if you have accumulated a large number of data files with identical or similar structures or if your work generates such files on a regular basis.

The input to the procedures is one or more data files. The output might be a printout of a graph or page layout or a text file of computed results.

Each person will need procedures customized to his or her situation. In this section, we present some examples that might serve as a starting point.

Variables Set by the LoadWave Operation

The LoadWave operation uses the numeric variable V_flag and the string variables S_fileName, S_path, and S_waveNames to provide information that is useful for procedures that automatically load waves. When used in a function, the LoadWave operation creates these as local variables.

LoadWave sets the string variable S_fileName to the name of the file being loaded. This is useful for annotating graphs or page layouts.

LoadWave sets the string variable S_path to the full path to the folder containing the file that was loaded. This is useful if you need to load a second file from the same folder as the first.

LoadWave sets the variable V_flag to the number of waves loaded. This allows a procedure to process the waves without knowing in advance how many waves are in a file.

LoadWave also sets the string variable S_waveNames to a semicolon-separated list of the names of the loaded waves. From a procedure, you can use the names in this list for subsequent processing.

Loading and Graphing Waveform Data

Here is a very simple example designed to show the basic form of an Igor function for automatically loading and graphing the contents of a data file. It loads a delimited text file containing waveform data and then makes a graph of the waves.

In this function, we make the assumption that the files that we are loading contain three columns of waveform data. Tailoring the function for a specific type of data file allows us to keep it very simple.

```
Function LoadAndGraph(fileName, pathName)
   String fileName      // Name of file to load or "" to get dialog
   String pathName      // Name of path or "" to get dialog

   // Load the waves and set the local variables.
   LoadWave/J/D/O/P=$pathName fileName
   if (V_flag==0)       // No waves loaded. Perhaps user canceled.
      return -1
   endif

   // Put the names of the three waves into string variables
   String s0, s1, s2
   s0 = StringFromList(0, S_waveNames)
   s1 = StringFromList(1, S_waveNames)
   s2 = StringFromList(2, S_waveNames)

   Wave w0 = $s0                // Create wave references.
   Wave w1 = $s1
   Wave w2 = $s2

   // Set waves' X scaling, X units and data units
   SetScale/P x, 0, 1, "s", w0, w1, w2
   SetScale d 0, 0, "V", w0, w1, w2

   Display w0, w1, w2           // Create a new graph

   // Annotate graph
   Textbox/N=TBFileName/A=LT "Waves loaded from " + S_fileName

   return 0                     // Signifies success.
End
```

s0, s1 and s2 are local string variables into which we place the names of the loaded waves. We then use the $ operator to create a reference to each wave, which we can use in subsequent commands.

Once the function is entered in the procedure window, you can execute it from the command line or call it from another function. If you execute

```
LoadAndGraph("", "")
```

the LoadWave operation will display a dialog allowing you to choose a file. If you call LoadAndGraph with the appropriate parameters, LoadWave will load the file without presenting a dialog.

You can add a "Load And Graph" menu item by putting the following menu declaration in the procedure window:

```
Menu "Macros"
   "Load And Graph...", LoadAndGraph("", "")
End
```

Because we have not used the "Auto name & go" option for the LoadWave operation, LoadWave will put up another dialog in which you can enter names for the new waves. If you want the macro to be more automatic, use /A or /N to turn "Auto name & go" on. If you want the procedure to specify the names of the loaded waves, use the /B flag. See the description of the **LoadWave** operation (see page V-349) for details.

To keep the function simple, we have hard-coded the X scaling, X units and data units for the new waves. You would need to change the parameters to the SetScale operation to suit your data. For more flexibility, you would add additional parameters to the function.

It is possible to write LoadAndGraph so that it can handle files with any number of columns. This makes the function more complex but more general.

For more advanced programmers, here is the more general version of LoadAndGraph.

```
Function LoadAndGraph(fileName, pathName)
   String fileName      // Name of file to load or "" to get dialog
   String pathName      // Name of path or "" to get dialog

   // Load the waves and set the variables.
   LoadWave/J/D/O/P=$pathName fileName
   if (V_flag==0)       // No waves loaded. Perhaps user canceled.
      return -1
   endif

   Display              // Create a new graph

   String theWave
   Variable index=0
   do                   // Now append waves to graph
      theWave = StringFromList(index, S_waveNames) // Next wave
      if (strlen(theWave) == 0)              // No more waves?
            break                            // Break out of loop
      endif
      Wave w = $theWave
      SetScale/P x, 0, 1, "s", w             // Set X scaling
      SetScale d 0, 0, "V", w                // Set data units
      AppendToGraph w
      index += 1
   while (1)         // Unconditionally loop back up to "do"

   // Annotate graph
   Textbox/A=LT "Waves loaded from " + S_fileName

   return 0          // Signifies success.
End
```

The do-loop picks each successive name out of the list of names in S_waveNames and adds the corresponding wave to the graph. S_waveNames will contain one name for each column loaded from the file.

There is one serious shortcoming to the LoadAndGraph function. It creates a very plain, default graph. There are three approaches to overcoming this problem:

- Use preferences.
- Use a style macro.
- Overwrite data in an existing graph.

Normally, Igor does not use preferences when a procedure is executing. To get preferences to take effect during the LoadAndGraph function, you would need to put the statement "Preferences 1" near the beginning of the function. This turns preferences on just for the duration of the function. This will cause the Display and AppendToGraph operations to use your graph preferences.

Using preferences in a function means that the output of the function will change if you change your preferences. It also means that if you give your function to a colleague, it will produce different results. This dependence on preferences can be seen as a feature or as a problem, depending on what you are trying to achieve. We normally prefer to keep procedures independent of preferences.

Using a style macro is a more robust technique. To do this, you would first create a prototype graph and create a style macro for the graph (see **Graph Style Macros** on page II-300). Then, you would put a call to the style macro at the end of the LoadAndGraph macro. The style macro would apply its styles to the new graph.

The last approach is to overwrite data in an existing graph rather than creating a new one. The simplest way to do this is to always use the same names for your waves. For example, imagine that you load a file with three waves and you name them wave0, wave1, wave2. Now you make a graph of the waves and set everything in the graph to your taste. You now load another file, use the same names and use LoadWave's overwrite option. The data from the new file will replace the data in your existing waves and Igor will automatically update the existing graph. Using this approach, the function simplifies to this:

```
Function LoadAndGraph(fileName, pathName)
   String fileName      // Name of file to load or "" to get dialog
   String pathName      // Name of path or "" to get dialog

   // load the waves, overwriting existing waves
   LoadWave/J/D/O/N/P=$pathName fileName
   if (V_flag==0)       // No waves loaded. Perhaps user canceled.
      return -1
   endif

   Textbox/C/N=TBFileName/A=LT "Waves loaded from " + S_fileName

   return 0             // Signifies success.
End
```

There is one subtle change here. We have used the /N option with the LoadWave operation, which autonames the incoming waves using the names wave0, wave1, and wave2.

You can see that this approach is about as simple as it can get. The downside is that you wind up with uninformative names like wave0. You can use the LoadWave /B flag to provide better names.

If you are loading data from Igor Binary files or from packed Igor experiments, you can use the LoadData operation instead of LoadWave. This is a powerful operation, especially if you have multiple sets of identically structured data, as would be produced by multiple runs of an experiment. See **The LoadData Operation** on page II-164 above.

Loading and Graphing XY Data

In the preceding example, we treated all of the columns in the file the same: as waveforms. If you have XY data then things change a bit. We need to make some more assumptions about the columns in the file. For example, we might have a collection of files with four columns which represent two XY pairs. The first two columns are the first XY pair and the second two columns are the second XY pair.

Here is a modified version of our function to handle this case.

```
Function LoadAndGraphXY(fileName, pathName)
   String fileName   // Name of file to load or "" to get dialog
   String pathName   // Name of path or "" to get dialog

   // load the waves and set the globals
   LoadWave/J/D/O/P=$pathName fileName
   if (V_flag==0)    // No waves loaded. Perhaps user canceled.
      return -1
   endif

   // Put the names of the waves into string variables.
   String sx0, sy0, sx1, sy1
   sx0 = StringFromList(0, S_waveNames)
   sy0 = StringFromList(1, S_waveNames)
   sx1 = StringFromList(2, S_waveNames)
   sy1 = StringFromList(3, S_waveNames)

   Wave x0 = $sx0                   // Create wave references.
   Wave y0 = $sy0
   Wave x1 = $sx1
   Wave y1 = $sy1

   SetScale d 0, 0, "s", x0, x1     // Set wave data units
   SetScale d 0, 0, "V", y0, y1

   Display y0 vs x0                 // Create a new graph
   AppendToGraph y1 vs x1

   Textbox/A=LT "Waves loaded from " + S_fileName  // Annotate graph

   return 0          // Signifies success.
End
```

The main difference between this and the waveform-based LoadAndGraph function is that here we append waves to the graph as XY pairs. Also, we don't set the X scaling of the waves because we are treating them as XY pairs, not as waveforms.

It is possible to write a more general function that can handle any number of XY pairs. Once again, adding generality adds complexity. Here is the more general version of the function.

```
Function LoadAndGraphXY(fileName, pathName)
   String fileName      // Name of file to load or "" to get dialog
   String pathName      // Name of path or "" to get dialog

   // Load the waves and set the globals
   LoadWave/J/D/O/P=$pathName fileName
   if (V_flag==0)       // No waves loaded. Perhaps user canceled.
      return -1
   endif

   Display              // Create a new graph

   String sxw, syw
   Variable index=0
   do       // Now append waves to graph
      sxw=StringFromList(index, S_waveNames) // Next name
      if (strlen(sxw) == 0)                  // No more?
            break       // break out of loop
      endif
      syw=StringFromList(index+1, S_waveNames)// Next name
```

```
      Wave xw = $sxw                    // Create wave references.
      Wave yw = $syw

      SetScale d 0, 0, "s", xw          // Set x wave's units
      SetScale d 0, 0, "V", yw          // Set y wave's units
      AppendToGraph yw vs xw

      index += 2
   while (1)          // Unconditionally loop back up to "do"

   // Annotate graph
   Textbox/A=LT "Waves loaded from " + S_fileName

   return 0           // Signifies success.
End
```

Loading All of the Files in a Folder

In the next example, we assume that we have a folder containing a number of files. Each file contains three columns of waveform data. We want to load each file in the folder, make a graph and print it. This example uses the LoadAndGraph function as a subroutine.

```
Function LoadAndGraphAll(pathName)
   String pathName      // Name of symbolic path or "" to get dialog

   String fileName
   String graphName
   Variable index=0

   if (strlen(pathName)==0)       // If no path specified, create one
      NewPath/O temporaryPath     // This will put up a dialog
      if (V_flag != 0)
         return -1                // User cancelled
      endif
      pathName = "temporaryPath"
   endif

   Variable result
   do       // Loop through each file in folder
      fileName = IndexedFile($pathName, index, ".dat")
      if (strlen(fileName) == 0)     // No more files?
         break                       // Break out of loop
      endif
      result = LoadAndGraph(fileName, pathName)
      if (result == 0)               // Did LoadAndGraph succeed?
                                     // Print the graph.
         graphName = WinName(0, 1)   // Get the name of the top graph
         String cmd
         sprintf cmd, "PrintGraphs %s", graphName
         Execute cmd    // Explained below.

         DoWindow/K $graphName       // Kill the graph
         KillWaves/A/Z               // Kill all unused waves
      endif
      index += 1
   while (1)

   if (Exists("temporaryPath"))      // Kill temp path if it exists
      KillPath temporaryPath
   endif
   return 0           // Signifies success.
End
```

This function relies on the IndexedFile function to find the name of successive files of a particular type in a particular folder. The last parameter to IndexedFile says that we are looking for files with a ".dat" extension. On Macintosh, if we changed the last parameter to "TEXT", IndexedFile would return all files of type TEXT, regardless of their extension.

Once we get the file name, we pass it to the LoadAndGraph function. After printing the graph, we kill it and then kill all the waves in the current data folder so that we can start fresh with the next file. A more sophisticated version would kill only those waves in the graph.

To print the graphs, we use the PrintGraphs operation. PrintGraphs is one of a few built-in operations that can not be directly used in a function. Therefore, we put the PrintGraphs command in a string variable and call Execute to execute it.

If you are loading data from Igor Binary files or from packed Igor experiments, you can use the LoadData operation. See **The LoadData Operation** on page II-164 above.

Saving Waves

Igor automatically saves the waves in the current experiment on disk when you save the experiment. Many Igor users load data from files into Igor and then make and print graphs or layouts. This is the end of the process. They have no need to explicitly save waves.

You can save waves in an Igor packed experiment file for archiving using the SaveData operation or using the Save Copy button in the Data Browser. The data in the packed experiment can then be reloaded into Igor using the LoadData operation or the Load Expt button in Data Browser. Or you can load the file as an experiment using File→Open Experiment. See the **SaveData** operation on page V-536 for details.

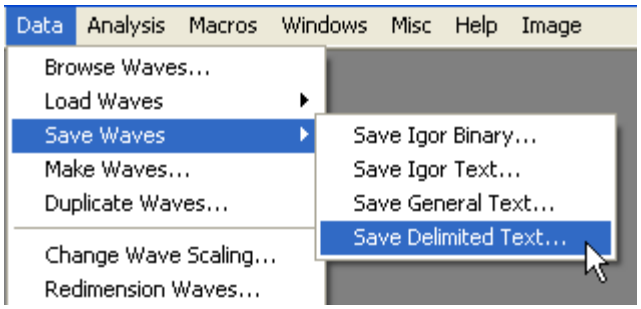
The main reason for saving a wave separate from its experiment is to export data from Igor to another program. To explicitly save waves to disk, you would use Igor's Save operation.

The following table lists the four types of built-in data saving routines in Igor and their salient features.

File type	Description
Delimited text	Used for archiving results or for exporting to another program.
	Row Format: `<data><tab><data><terminator>`*
	Contains one block of data with any number of rows and columns. A row of column labels is optional.
	Columns may be equal or unequal in length.
	Can export 1D or 2D waves.
General text	Used for archiving results or for exporting to another program.
	Row Format: `<number><tab><number><terminator>`*
	Contains one or more blocks of numbers with any number of rows and columns. A row of column labels is optional.
	Columns in a block must be equal in length.
	Can export 1D or 2D waves.
Igor Text	Used for archiving waves or for exporting waves from one Igor experiment to another.
	Format: See **Igor Text File Format** on page II-159 above.
	Contains one or more wave blocks with any number of waves and rows. A given block can contain either numeric or text data.
	Consists of special Igor keywords, numbers and Igor commands.
	Can export waves of dimension 1 through 4.
Igor Binary	Used for exporting waves from one Igor experiment to another.
	Contains data for one Igor wave.
	Format: See Igor Technical Note #003, "Igor Binary Format".

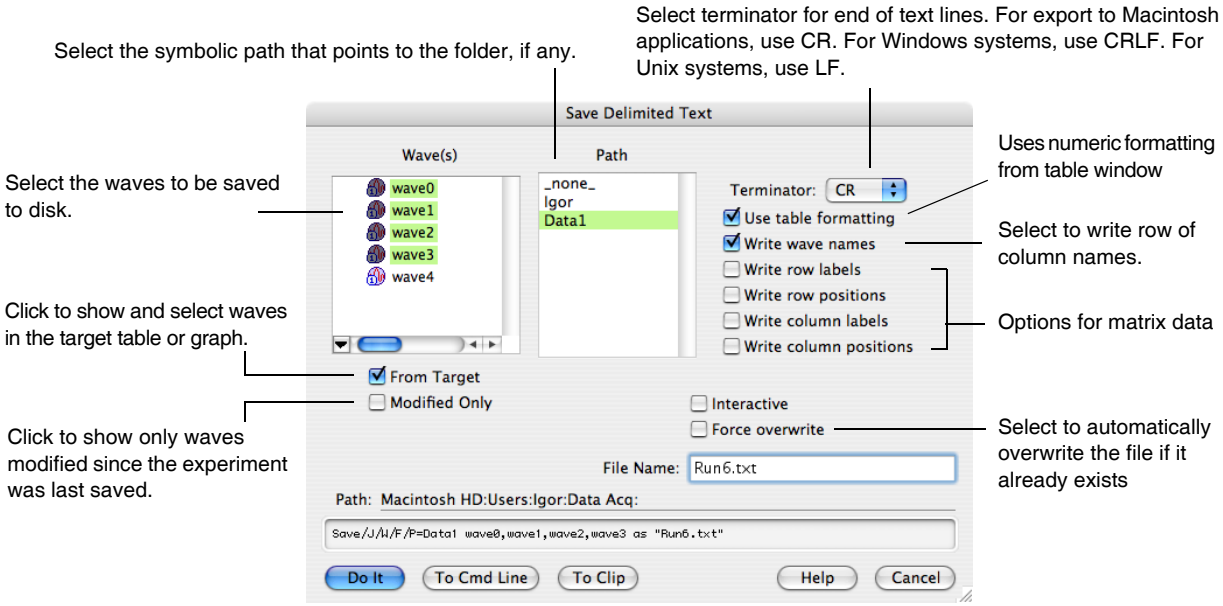
* `<terminator>` can be carriage return, linefeed or carriage return/linefeed. You would use carriage return for exporting to a Macintosh program, carriage return/linefeed for Windows systems, and linefeed for Unix systems.

You can access all of the built-in routines via the Save Waves submenu of the Data menu.



Saving Waves in a Delimited Text File

To save a delimited text file, invoke the Save Delimit ed Text dialog via the Save Waves submenu of the Data menu.



The Save Delimited Text routine writes a file consisting of numbers separated by tabs with a selectable line terminator at the end of each line of text. When writing 1D waves, it can optionally include a row of column labels. When writing a matrix, it can optionally write row labels as well as column labels plus row and column position information.

Save Delimited Text can save waves of any dimensionality. Multidimensional waves are saved one wave per block. Data is written in row/column/layer/chunk order. Multidimensional waves saved as delimited text can not be loaded back into Igor as delimited text because the Load Delimited Text routine does not support multiple blocks. They can be loaded back in as general text. However, for data that is intended to be loaded back into Igor later, the Igor Text, Igor Binary or Igor Packed Experiment formats are preferable.

The order of the columns in the file depends on the order in which the wave names appear in the Save command. This dialog generates the wave names based on the order of the waves in the dialog list which in turn depends on the order in which the waves were created. If you want to change the order then you should click the To Cmd Line button instead of the Do It button and edit the command in Igor's command line.

By default, the Save operation writes numeric data using the "%.15g" format for double-precision data and "%.7g" format for data with less precision. These formats give you up to 15 or 7 digits of precision in the file.

To use different numeric formatting, create a table of the data that you want to export. Set the numeric formatting of the table columns as desired. Be sure to display enough digits in the table because the data will

be written to the file as it appears in the table. In the Save Delimited Text dialog, select the "Use table formatting" checkbox. When saving a multi-column wave (1D complex wave or multi-dimensional wave), all columns of the wave are saved using the table format for the first table column from the wave.

The wfPrintf command line operation can also be used to save waves to text files using a specific numeric format.

The Save operation is capable of appending to an existing file, rather than overwriting the file. This is useful for accumulating results of a analysis that you perform regularly in a single file. You can also use this to append a block of numbers to a file containing header information that you generated with the fPrintf operation. The append option is not available through the dialog. If you want to do this, see the discussion of the **Save** operation (see page V-533).

Saving Waves in a General Text File

Saving waves in a general text file is very similar to saving a delimited text file. The Save General Text dialog is identical to the Save Delimited Text dialog.

All of the columns in a single block of a general text file must have the same length. The Save General Text routine writes as many blocks as necessary to save all of the specified waves. For example, if you ask it to save two 1D waves with 100 points and two 1D waves with 50 points, it will write two blocks of data. Multidimensional waves are written one wave per block.

Saving Waves in an Igor Text File

Saving waves in an Igor Text file is also very similar to saving a delimited text file. The Save Igor Text dialog is identical to the Save Delimited Text dialog.

The Igor Text format is capable of saving not only the numeric contents of a wave but its other properties as well. It saves each wave's dimension scaling, units and labels, data full scale and units and the wave's note, if any. All of this data is saved more efficiently as binary data when you save as an Igor packed experiment using the SaveData operation.

As in the general text format, all of the columns in a single block of an Igor Text file must have the same length. The Save Igor Text routine handles this requirement by writing as many blocks as necessary.

Save Igor Text can save waves of any dimensionality. Multidimensional waves are saved one wave per block. The /N flag at the start of the block identifies the dimensionality of the wave. Data is written in row/column/layer/chunk order.

Saving Waves in Igor Binary Files

Igor's Save Igor Binary routine saves waves in Igor Binary files, one wave per file. Most users will not need to do this since Igor automatically saves waves when you save an Igor experiment. You might want to save a wave in an Igor Binary file to send it to a colleague.

The Save Igor Binary dialog is similar to the Save Delimited Text dialog. There is a difference in file naming since, in the case of Igor Binary, each wave is saved in a separate file. If you select a single wave from the dialog's list, you can enter a name for the file. However, if you select multiple waves, you can not enter a file name. Igor will use default file names of the form "wave0.ibw".

When you save an experiment in a packed experiment file, all of the waves are saved in Igor Binary format. The waves can then be loaded into another Igor experiment using the **Data Browser** (see page II-130) or **The LoadData Operation** (see page II-164).

Saving Waves in Image Files

You can save some types of multidimensional waves as image files. The two main limitations of image files are that they usually support only 8 bit depth and that some formats (e.g., JPEG) rely on lossy compression. To avoid compression loss you should choose either TIFF or PNG file formats. At present, the extended

TIFF file format is a bit more flexible in that you can save in 8, 16, or 32 bits per sample and you can use image stacks to support 3D and 4D waves. See the **ImageSave** operation on page V-281 for more details.

Saving Sound Files

You can save waves as sound files using the SndLoadSaveWave XOP. See the corresponding help file in the More Extensions:File Loaders folder.

Exporting Text Waves

Igor does not quote text when exporting text waves as a delimited or general text file. It does quote text when exporting it as an Igor Text file.

Certain special characters, such as tabs, carriage returns and linefeeds, cause problems during exchange of data between programs because most programs consider them to separate one value from the next or one line of text from the next. Igor Text waves can contain any character, including special characters. In most cases, this will not be a problem because you will have no need to store special characters in text waves or, if you do, you will have no need to export them to other programs.

When Igor writes a text file containing text waves, it replaces the following characters, when they occur within a wave, with their associated escape codes:

Character	Name	ASCII Code	Escape Sequence
CR	carriage return	13	\r
LF	linefeed	10	\n
tab	tab	9	\t
\	backslash	92	\\

Igor does this because these would be misinterpreted if not changed to escape sequences. When Igor loads a text file into text waves, it reverses the process, converting escape sequences into the associated ASCII code.

This use of escape codes can be suppressed using the /E flag of the **Save** operation (see page V-533). This is necessary to export text containing backslashes to a program that does not interpret escape codes.

Exporting MultiDimensional Waves

When exporting a multidimensional wave as a delimited or general text file, you have the option of writing row labels, row positions, column labels and column positions to the file. Each of these options is controlled by a checkbox in the Save Waves dialog. There is a discussion of row/column labels and positions under **2D Label and Position Details** on page II-149.

Igor writes multidimensional waves in column/row/layer/chunk order.

Accessing SQL Databases

Igor Pro includes an XOP, called SQL XOP, which provides access to relational databases from IGOR procedures. It uses ODBC (Open Database Connectivity) libraries and drivers on Mac OS X and Windows to provide this access.

For details on configuring and using SQL XOP, open the SQL Help file in “Igor Pro Folder:More Extensions:Utilities”.

Chapter
II-10

Dialog Features

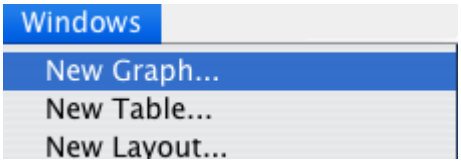
Overview

Most Igor Pro dialogs are designed with common features. This chapter describes some of those common features.

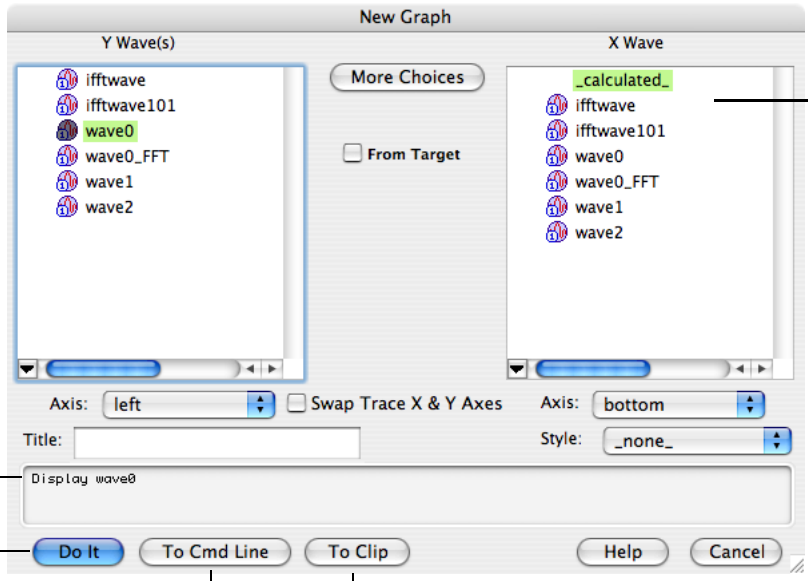
Operation Dialogs

Menus and dialogs provide easy access to many of Igor Pro's operations.

When you choose a menu item:



You will see a dialog:



Read about Wave Browsers below.

The dialog generates a command suitable for execution in the command line.

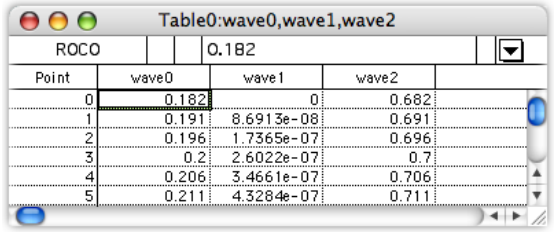
Transfers the command to the command line and executes it.

Copies the command to the command line where you can edit it and then execute it.

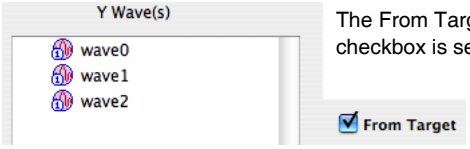
Copies the command to the Clipboard. Useful when you are writing Igor procedures.

As you click and type in the items in the dialog, Igor generates an appropriate command. The command being generated is displayed in the command box near the bottom of the dialog. As you become more proficient, you will find that some commands are easier to invoke from a dialog and others are easier to enter directly in the command line. There are some menus and dialogs that bypass the command line, usually because they perform functions that have no command line equivalents.

Many dialogs include a From Target checkbox. If it is selected, the lists of waves available for you to choose are restricted to waves that appear in the topmost graph or table window:



The table is showing wave0, wave1, and wave2.



The From Target checkbox is selected…

… only wave0, wave1, and wave2 are available.

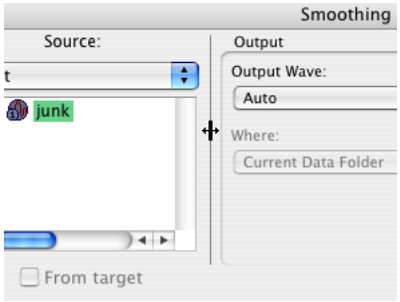
Sometimes, if your top graph or table contains waves that are not in the current data folder, or if you are not viewing the current data folder in the wave browser, you may not see any waves. In that case, you may need to hunt for them. If you are unfamiliar with data folders, this probably won't be a problem.

Resizable Dialogs

Many dialogs are resizable. The dialog at the beginning of this chapter is one. On Macintosh, you can identify a resizable dialog by the resizing handle in the lower-right corner. On Windows, resizable dialogs are marked with a special icon in the upper-left corner.

When you resize a dialog, parts of the dialog that may need more space will be made larger. In the New Graph dialog pictured above, the Wave Browsers are given more room when you increase the size of the dialog.

Movable Dividers

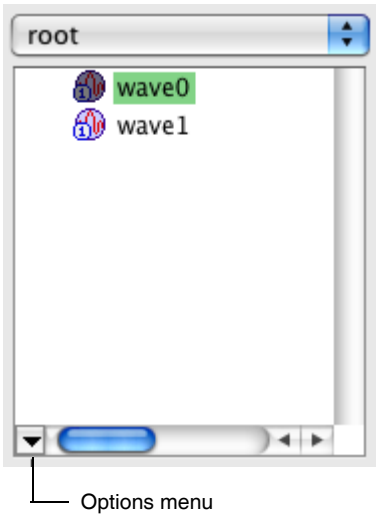


Many dialogs include movable dividers. These allow you to change the proportion of dialog real estate allocated to different parts of the dialog. The dividers may be hard to find- they look just like plain divider lines. Moveable dividers can be recognized by the drag arrow cursor when the mouse cursor is over a moveable divider.

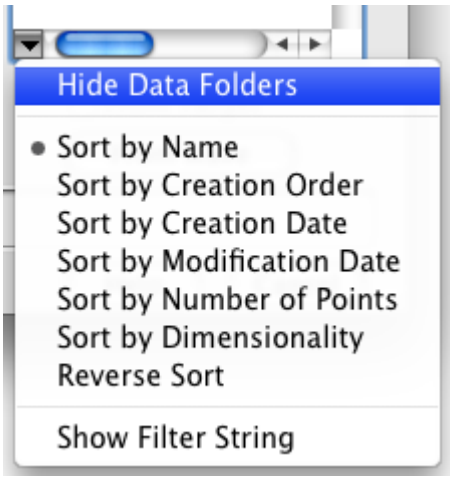
In this case a vertical divider can be dragged left or right to give more or less room to the Wave Browser in the left portion of the dialog.

Dialog Wave Browser

In dialogs in which a wave must be selected, Igor presents a list of suitable waves in a Dialog Wave Browser. As shipped from the factory, Igor shows only waves in the current data folder. As shipped from the factory, Igor shows a hierarchical list of data folders and waves and/or numeric or string variables. Here is a picture of a typical dialog wave browser, after two waves have been made, and no data folders have been created:



If you don't know what data folders are, or you prefer not to deal with them, you may prefer to hide the data folders view. To do so, pop up the Options menu and select Hide Data Folders:

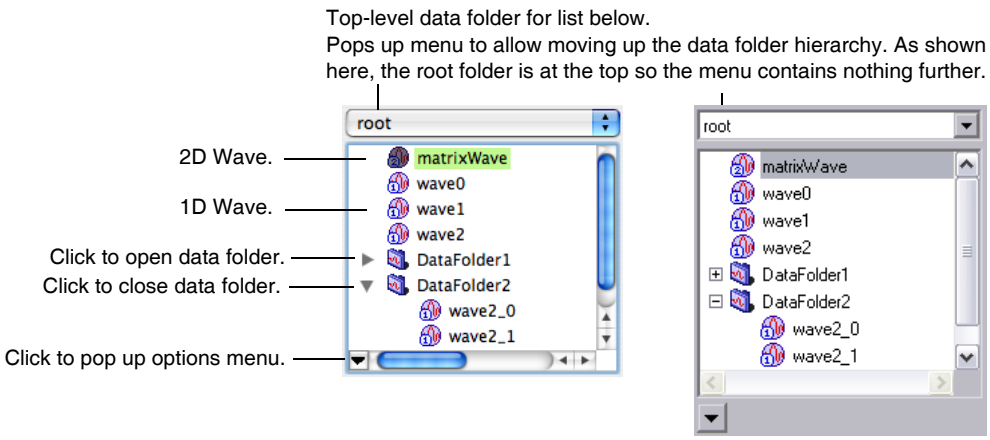


You can restore the data folders view by again popping up the Options menu and selecting Show Data Folders. When you change from Show to Hide or from Hide to Show, all dialog wave browsers in all the dialogs are changed, and the change is stored in preferences so that you won't have to make the selection again.

To learn more about data folders, see **Data Folders** on page II-121.

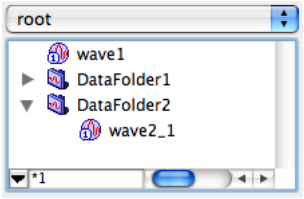
Dialog Wave Browser Details

With data folders displayed, if you have created a couple of data folders, and some waves, the dialog wave browser looks like this:

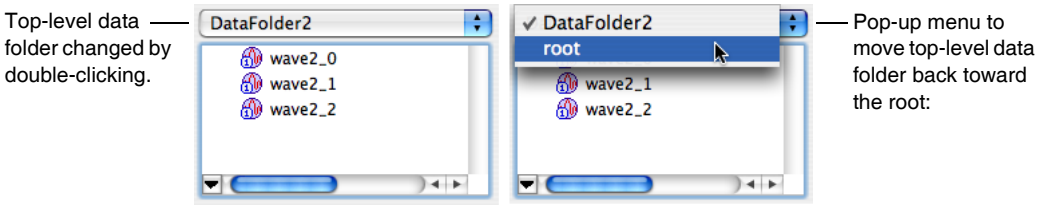


The options menu turns display of data folders on or off, sets criteria for sorting the waves, and displays an edit box for editing a filter string to select a subset of the waves.

Here is a view of the Wave Browser with the filter string displayed. Unless you change it, the filter string is * which simply allows any names. The * is the "wild card" character, which matches anything. In the picture below, the filter string has been changed to *1, which will show only waves whose names end with the character 1.



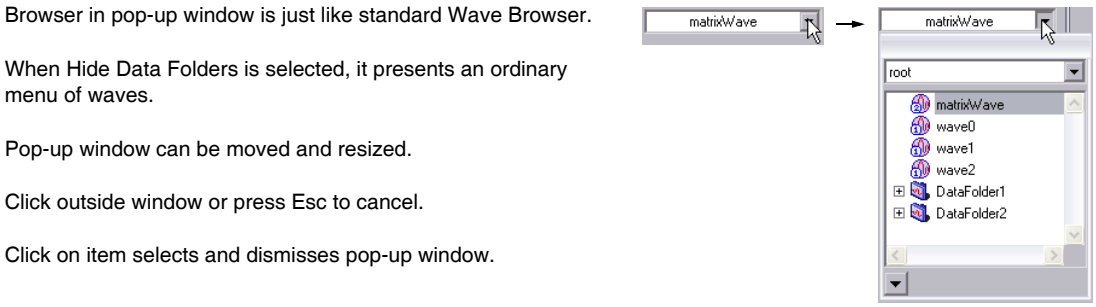
Double-click a data folder icon to make that data folder the top level for the hierarchical display. To return to levels closer to the root, use the menu at the top of the browser:



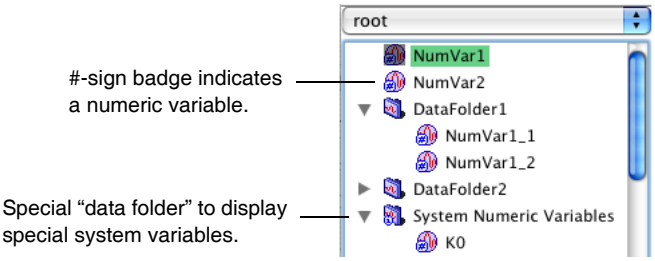
Depending on how the Wave Browser is used in a given dialog, it may support selection of multiple items or it may allow selection of only one item. In multiple-selection browsers, you can hold down the mouse button and drag over multiple items to select more than one item. To add additional items to a selection:

- Hold down Shift and click an item to extend the selection over all items between the current selection and the item clicked.
- Hold down Command (*Macintosh*) or Ctrl (*Windows*) and click an item to add just that item to the current selection or to remove it if it is already selected.

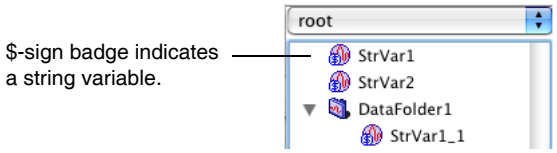
In some dialogs, a pop-up version of the Wave Browser is used. It attempts (not entirely successfully) to mimic a pop-up menu:



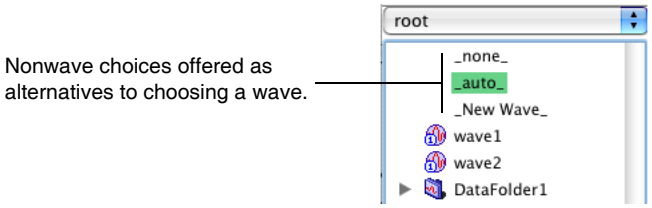
Throughout this discussion, we have talked only about selecting waves. In a few cases, a Wave Browser may be presented to select global variables, strings or even data folders. Here is a Wave Browser for selection of numeric variables:



or string variables:

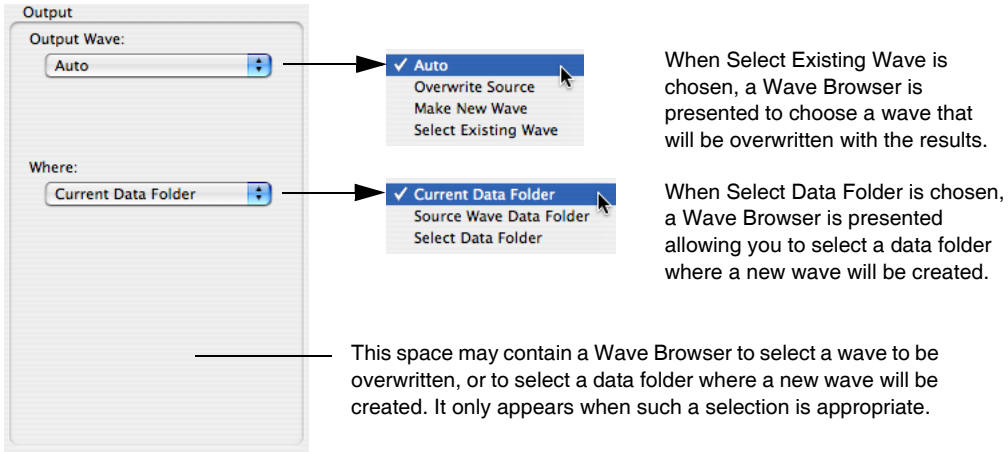


Sometimes a dialog will offer some alternative choice that is not a wave. For instance, several operations allow you to choose X values that come from either the Y wave's X scaling or from an X wave. In that case, the X wave browser will offer the choice "_calculated_" in addition to waves. Other dialogs may offer other nonwave choices. These nonwave choices will be visible, usually at the top of the Wave Browser window, regardless of the top-level data folder. Here is a picture of a Wave Browser from the Curve Fitting dialog with several nonwave choices:



Operation Result Chooser

In most Igor dialogs that perform numeric operations (Analysis menu: Integrate, Smooth, FFT, etc.) there is a group of controls allowing you to choose what to do with the result. Here is what the Result Chooser looks like in the Integrate dialog:



The Result Chooser is not always laid out vertically as in this picture, but it generally offers all the choices shown here.

Note: Users of older versions of Igor will recall that in almost all cases, an operation dialog would replace the original data in the wave with the result, thereby destroying the input data. The Result Chooser eliminates having to cancel the dialog in order to make a duplicate of the input data.

The Output Wave menu offers choices of a wave to receive the result of the operation:

Auto	Igor will create a new wave to receive the results. The source wave is not changed. The new wave will have a name derived from the source wave by adding a suffix that depends on the operation. choosing Auto makes the Where menu available.
Overwrite Source	The source wave (the wave that contains the input data) will be overwritten with the results of the operation. This will destroy the original data. This is how most operations worked prior to Igor Pro 5. The Where menu will not be available.
Make New Wave	This is like the Auto choice, but an edit box is presented that you use to type a name of your own choosing. Igor will make a new wave with this name to receive the results of the operation. This selection makes the Where menu available.
Select Existing Wave	A Wave Browser will be presented allowing you to choose any existing wave to be overwritten with the results. This choice preserves the contents of the source wave, but destroys the contents of the wave chosen to receive the results.

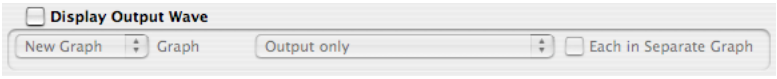
The Where menu offers choices for the location of a new wave created when you choose Auto or Make New Wave. Usually you will want to choose Current Data Folder. If you don't understand what this means, it is almost certain that you should choose Current Data Folder.

Current Data Folder	The new wave is created in the current data folder. If you don't know about data folders, this is probably the best choice.
Source Wave Data Folder	The new wave is created in the same data folder as the source wave. It is quite likely that the source wave will be in the current data folder, in which case this choice is the same as choosing Current Data Folder.
Select Data Folder	This choice presents a Wave Browser in which you can choose a data folder where the new wave will be created.

Operation Result Displayer

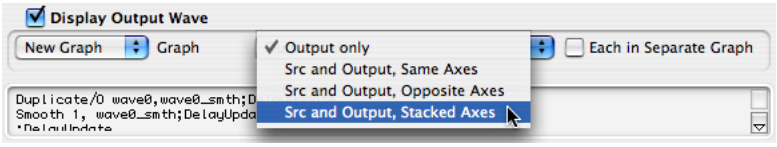
In some Igor dialogs that perform numeric operations (Analysis menu: Integrate, Smooth, FFT, etc.) there is a group of controls allowing you to choose how to display the result. Choices are offered to put the result into the top graph, a new graph, the top table, or a new table. For two-dimensional results, New Image and New Contour are also offered. If the result is complex, as is the case for an FFT, New Contour is not available.

Here is what the Result Displayer looks like in the Smooth dialog:



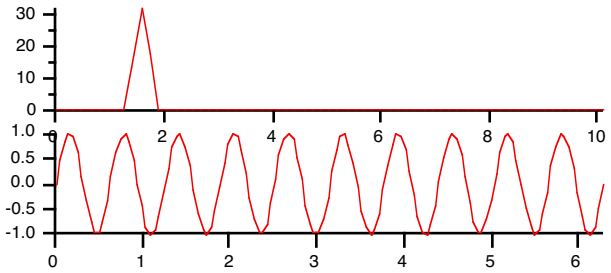
The contents of the displayer are not available here because the Display Output Wave checkbox is not selected. This is the default state.

When you choose New Graph, there are four choices in the Graph menu for the contents and layout of the new graph. In this menu, *Src* stands for Source. It is the wave containing the input data; Output is the wave containing the result of the operation.



In many cases, the second choice, Src and Output, Same Axes, will not be appropriate because the operation changes the magnitude of data values or the range of the X values.

This picture shows the result of an FFT operation when Src and Output, Stacked Axes is chosen:



When you choose New Image or New Contour to display matrix results, the Graph Layout menu allows only Output Only or Src and Output, Stacked Axes. The axes aren't really stacked- it makes side-by-side graphs. It makes little sense to put two images or two contours on one set of axes.

The Result Displayer doesn't give you many options for formatting the graph, and doesn't allow any control over trace style, placement of axes, etc. It is intended to be a convenient way to get started with a graph. You can then modify the graph in any way you choose.

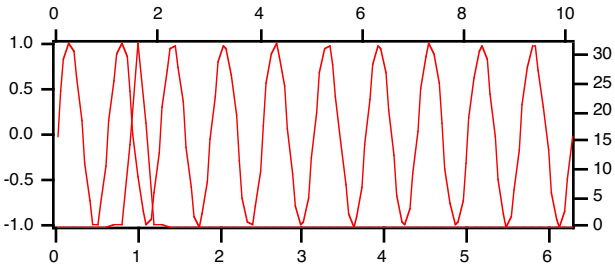
If you want a more complex graph, you may need to use the New Graph dialog (choose New Graph from the Windows menu) after you have clicked Do It in an operation dialog.

If you choose Top Graph instead of New Graph, the output wave will be appended to the top graph. It is assumed that this graph will already contain the source wave, so there is no option to append the source wave to the top graph. The Graph layout menu disappears, and two menus are presented to let you choose axes for the new wave:

The menus allow you to choose the standard axes: left and right in the V Axis menu; top and bottom in the H Axis menu. If the top graph includes any free axes (axes you defined yourself) they will be listed in the appropriate menu as well.

In most cases the source wave will be plotted on the left and bottom axes. You will usually want to select the right axis because of the differing magnitude of data values that result from most operations. You may also want to select the top axis if the operation (like the FFT) changes the X range as well.

Here is the result of choosing right and top when doing an FFT (this is the same input data as in the graph above):



Note that the format of the graph is poor. We leave it to you to format it as you wish. If you want a stacked graph, it may be better to choose the New Graph option.

Chapter
II-11

Tables

Overview

Tables are useful for entering, modifying or inspecting waves. You can also use a table for presentation purposes by exporting it to another program as a picture or by including it in a page layout. However, it is not optimized for this purpose.

If your data has a small number of points you will probably find it most convenient to enter it in a table. In this case, creating a new empty table will be your first step.

If your data has a large number of points you will most likely load it into Igor from a file. In this case it is not necessary to make a table. However, you may want to display the waves in a table to inspect them. Igor Pro tables can handle virtually any number of rows and columns provided you have sufficient memory.

A table in Igor is similar to but not identical to a spreadsheet in other graphing programs. The main difference is that in Igor data exists independent of the table. You can create new waves in Igor's memory by entering data in a table. Once you have entered your data, you may, if you wish, kill the table. The waves exist independently in memory so killing the table does not kill the waves. You can still display them in a graph or in a new table.

In a spreadsheet, you can create a formula that makes one cell dependent on another. You can not create cell-based dependencies in Igor. You can create dependencies that control entire waves using Analysis→Compose Expression, Misc→Object Status, or the **SetFormula** operation (see page V-559).

To make a table, use the New Table item in the Windows menu. When the active window is a table, the Table menu appears in Igor's menu bar. This menu appends and removes columns, changes the appearance of columns, and sets table preferences.

Waves in tables are updated dynamically. Whenever the values in a wave change, Igor automatically updates any tables containing that wave. Because of this, tables are often useful for troubleshooting number-crunching procedures.

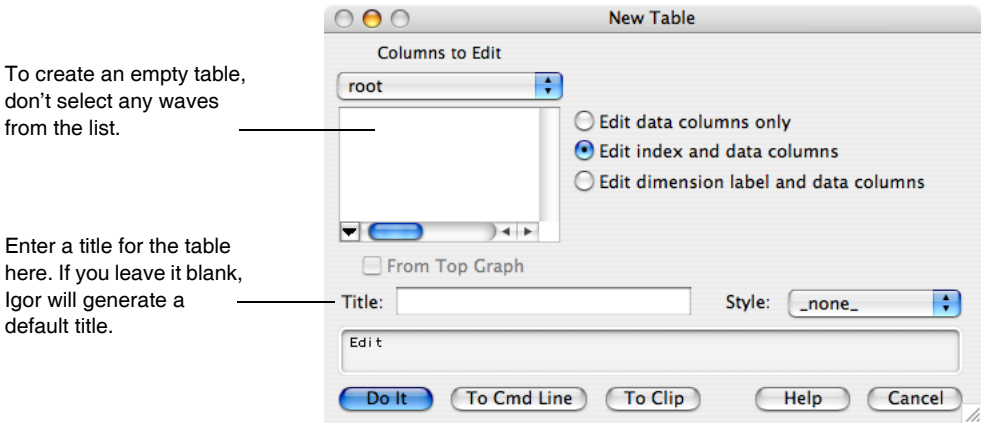
Creating Tables

Table Creation with New Experiment

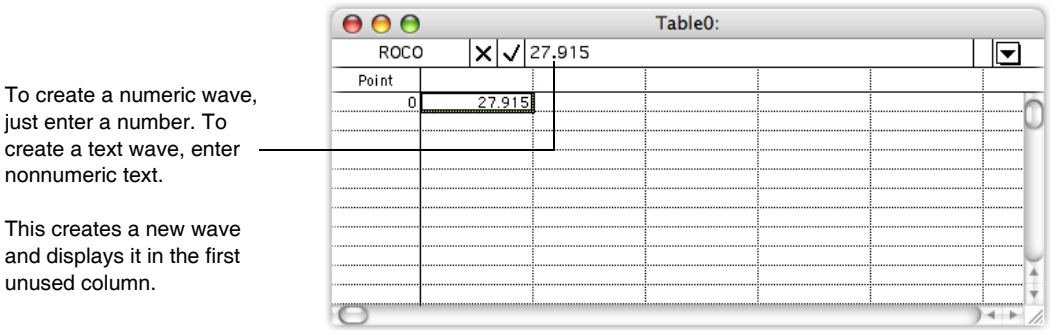
By default, when you create a new experiment, Igor automatically creates a new, empty table. This is convenient if you generally start working by entering data manually. However, in Igor data can exist in memory without being displayed in a table. If you wish, you can turn automatic table creation off using the Experiment Settings category of the Miscellaneous Settings dialog (Misc menu).

Creating an Empty Table for Entering New Waves

Choose New Table from the Windows menu.



After clicking the Do It button, you create an empty table in which you can enter data.



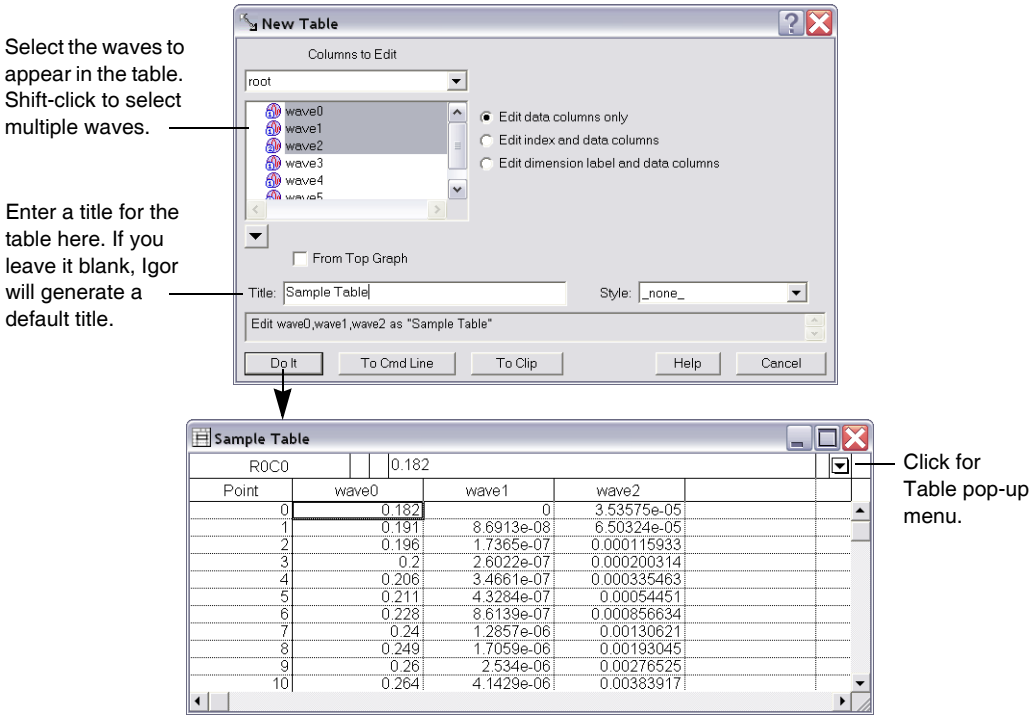
If you enter a numeric value, Igor will create a numeric wave. If you enter a nonnumeric value, Igor will create a text wave.

To create multidimensional waves you must use the Make Waves dialog (Data menu).

After creating the wave, you may want to rename it. Choose Rename from the Table pop-up menu or from the Data menu in the main menu bar.

Creating a Table to Edit Existing Waves

Choose New Table from the Windows menu.



Showing Index Values

As described in Chapter II-5, **Waves**, waves have built-in scaled index values. The New Table and Append to Table dialogs allow you to display just the data in the wave or the index values and the data.

A 1D wave's X index values are determined by its X scaling which is a property that you set using the Change Wave Scaling dialog or SetScale operation. A 2D wave has X and Y scaling, controlling X and Y scaled index values. Higher dimension waves have additional scaling properties and scaled index values. Displaying index values in a table is of use mostly if you are not sure what a wave's scaling is or if you want to see the effect of a SetScale operation.

Showing Dimension Labels

As described in Chapter II-6, **Multidimensional Waves**, waves have dimension labels. The New Table and Append Columns to Table dialogs allow you to display just the data in the wave or the dimension labels and the data. If you click the Edit Dimension Label And Data Columns radio button in either of these dialogs, Igor will display the wave's dimension labels.

Dimension labels are of use only when individual rows or columns of data have distinct meanings. In an image, for example, this is not the case because the significance of one row or column is the same as any other row or column. It is the case when a multidimensional wave is really a collection of related but disparate data.

Here is an example of a table showing a 2D wave with dimension labels.

The table shows the yardage at various golf courses from the red (front), white (middle), and blue (back) tees.

The golf course names are dimension labels for particular rows of the wave. "Course" is the overall dimension label for the rows dimension. The colors are dimension labels for particular columns of the wave. "Tees" is the overall dimension label for the columns dimension.

When you choose to display dimension labels, a table that contains multidimensional waves has one or more columns of dimension labels and one row of dimension labels. If the table contains 1D waves only, Igor does not display the row of dimension labels.

You can display dimension labels or dimension indices in a table, but you can not display both at the same time for the same wave.

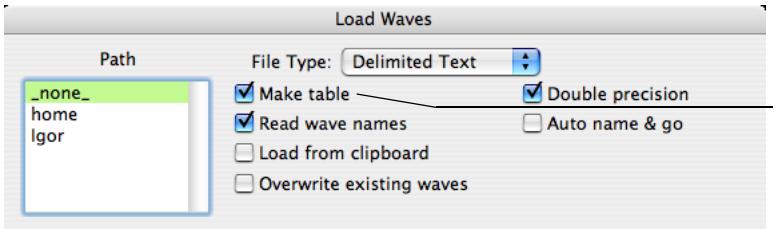
The Horizontal Index Row

When a multidimensional wave is displayed in a table, Igor adds the horizontal index row, which appears below the column names and above the data cells. This row can display numeric dimension indices or textual dimension labels.

By default, the horizontal index row displays dimension labels if the wave's dimension label column is displayed in the table. Otherwise it displays numeric dimension indices. You can override this default using the Table→Horizontal Index submenu.

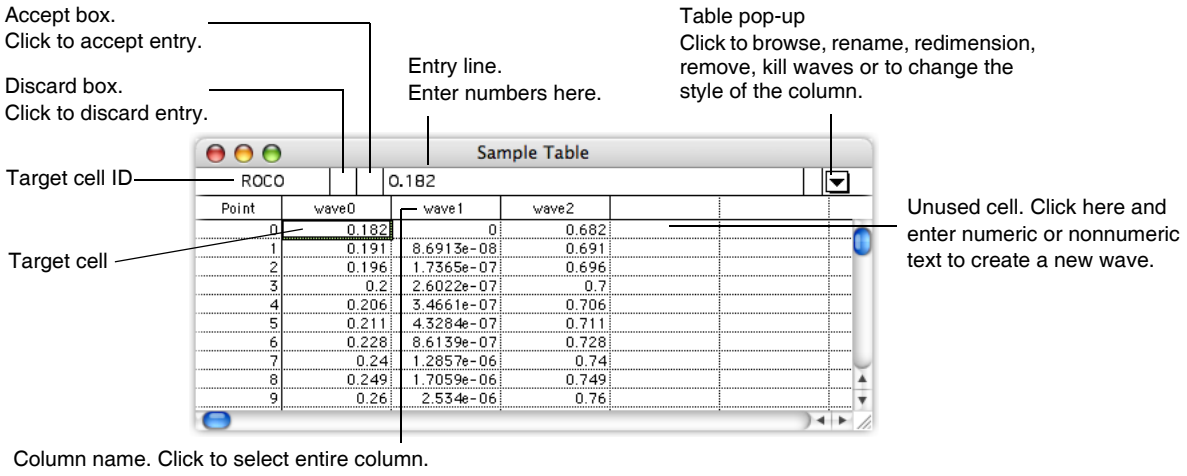
Creating a Table While Loading Waves From a File

The Load Waves dialog (Data menu) has an option to create a table to show the newly loaded waves.



Parts of a Table

This diagram shows the parts of a table displaying 1D waves. If you display multidimensional waves, Igor adds some additional items to the table, described under **Editing Multidimensional Waves** on page II-218.



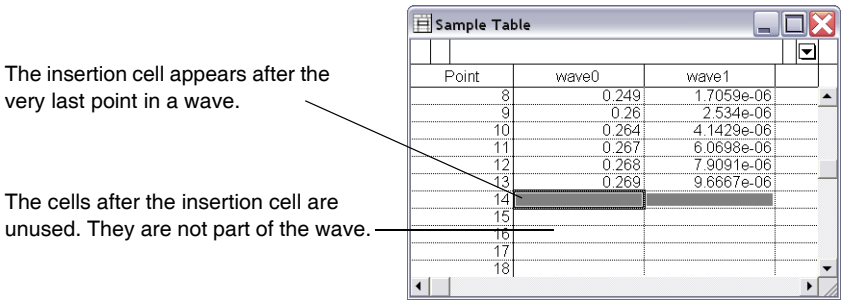
The bulk of a table is the **cell area**. The cell area contains columns of numeric or text data values as well as the column of point numbers on the left. If you wish, it can also display index columns or dimension label columns. To the right are unused columns into which you can type or paste new data.

If the table displays multidimensional waves then it will include a row of column indices or dimension labels below the row of names. Use the Append Columns to Table dialog to switch between the indices and labels.

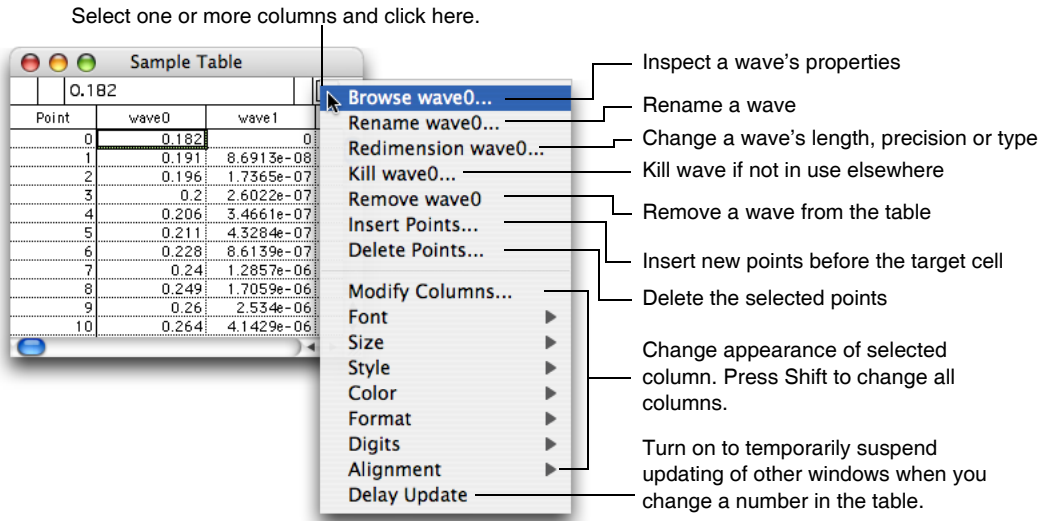
In the top left corner is the **target cell ID** area. This identifies a wave element corresponding to the target cell. For example, if a table displays a 2D wave, the ID area might show "R13 C22", meaning that the target cell is on row 13, column 22 of the 2D wave. For 3D waves the target cell ID includes the layer ("L") and for a 4D wave it includes the chunk ("Ch").

If you scroll the target cell out of view you can quickly bring it back into view by clicking in the target cell ID

There is a special cell, called the **insertion cell**, at the bottom of each column of data values. You can add points to a wave by entering a value or pasting in the insertion cell.



The **Table pop-up menu** provides a quick way to inspect or change a wave, remove or kill a wave and change the appearance of one or more columns. You can get the Table pop-up menu by clicking the ▼ icon or right-clicking (*Windows*) or Control-clicking (*Macintosh*) a column.



For waves displayed in multiple columns (complex waves and multidimensional waves), if you change the display format of any data column from the wave, Igor changes the format for all data columns from that wave.

One of the items in the pop-up menu is Delay Update. Normally, when you change the value of a cell in the table, Igor immediately updates any other tables or graphs to reflect the new value. Enabling Delay Update forces this updating of other tables and graphs to be postponed until you click in another window or disable Delay Update. When Delay Update has been enabled, there is a checkmark next to the menu item. You can do this if you have a list of values to enter into a table and you don't want other tables or graphs to be updated until you are finished.

Delay Update does not delay updates when you remove or add cells to a wave. It only delays updates when you change the value of a cell.

Showing and Hiding Parts of a Table

The Table menu has a Show submenu that shows or hides various parts of a table. This is of use only in specialized situations such as when you are using a table subwindow in a control panel to display data but don't want the user to enter data. For normal use you should leave all of the items in the Show submenu checked so that all parts of the table will be visible.

When the entry line is hidden, the user can not change values in the table.

Arrow Keys in Tables

By default, if you are in the process of entering data, the arrow keys accept the entry as if you pressed Enter and then move the selected cell.

Some users prefer to use the arrow keys to move the selection in the entry line when an entry is in progress. You can specify your preference via the Table Settings category in the Miscellaneous Settings dialog (Misc menu).

Keyboard Navigation in Tables

The term "keyboard navigation" refers to selection and scrolling actions in response to the arrow keys and to the Home, End, Page Up, and Page Down keys. Macintosh and Windows have different conventions for these actions in windows containing text. You can use either Macintosh or Windows conventions on either platform.

By default, Macintosh conventions apply on Macintosh and Windows conventions apply on Windows. You can change this using the Keyboard Navigation menu in the Misc Settings section of the Miscellaneous Set-

tings Dialog. If you use Macintosh conventions on Windows, use Ctrl in place of Command. If you use Windows conventions on Macintosh, use Command in place of Ctrl.

Macintosh Table Navigation

Key	No Modifier	Option	Command
Left Arrow	Move selection left one cell	Not used	Move selection to first column
Right Arrow	Move selection right one cell	Not used	Move selection to last column
Up Arrow	Move selection up one cell	Show previous layer	Scroll and move selection to first row
Down Arrow	Move selection down one cell	Show next layer	Scroll and move selection to last row
Home	Scroll to start of document	Scroll to start of document	Scroll to start of document
End	Scroll to end of document	Scroll to end of document	Scroll to end of document
Page Up	Scroll up one screen	Scroll left one screen	Scroll up one screen
Page Down	Scroll down one screen	Scroll right one screen	Scroll down one screen

When viewing a 3D or 4D wave, Option-Up Arrow and Option-Down Arrow change the currently viewed layer.

When viewing a 4D wave, Command-Option-Up Arrow and Command-Option-Down Arrow change the currently viewed chunk.

Pressing shift-arrow-key extends the selection in the direction of the arrow key. Pressing cmd-shift-arrow-key extends selection as far as possible in the direction of the arrow key.

Windows Table Navigation

Key	No Modifier	Alt	Ctrl
Left Arrow	Move selection left one cell	Not used	Move selection to first column
Right Arrow	Move selection right one cell	Not used	Move selection to last column
Up Arrow	Move selection up one cell	Show previous layer	Scroll and move selection to first row
Down Arrow	Move selection down one cell	Show next layer	Scroll and move selection to last row
Home	Move selection to first visible cell	Not used	Scroll and move selection to first cell
End	Move selection to last visible cell	Not used	Scroll and move selection to last cell
Page Up	Scroll up one screen	Scroll left one screen	Scroll up one screen
Page Down	Scroll down one screen	Scroll right one screen	Scroll down one screen

When viewing a 3D or 4D wave, Alt+Up Arrow and Alt+Down Arrow change the currently viewed layer.

When viewing a 4D wave, Ctrl+Alt+Up Arrow and Ctrl+Alt+Down Arrow change the currently viewed chunk.

Pressing shift-arrow-key extends the selection in the direction of the arrow key. Pressing cmd-shift-arrow-key extends selection as far as possible in the direction of the arrow key.

Decimal Symbol and Thousands Separator in Tables

By default, the decimal symbol for entering a number in a table is period. You can change this to comma or Per System Setting using the Table→Table Misc Settings menu. The selected decimal symbol is used for entering, copying, and pasting data in tables.

If comma is selected as the decimal symbol then it is not supported as a column separator when creating new waves by pasting text into a table.

When you choose Per System Setting, the decimal symbol is determined by the Formats tab of the International Control panel (*Macintosh*) and by the Regional Options tab of the Regional and Language Options control panel (*Windows*). Only period and comma are supported as the decimal symbol. If you choose any decimal symbol other than comma, period will be used as the decimal symbol for tables.

When creating a new wave by entering data into an unused table cell, there are some rare situations when what you are trying to enter cannot be properly interpreted unless you first choose the appropriate column numeric format from the Table menu. For example, if the decimal symbol is comma and you want to enter a time or date/time value with fractional seconds, you must choose Time or Date/Time from the Table→Formats menu before entering the data.

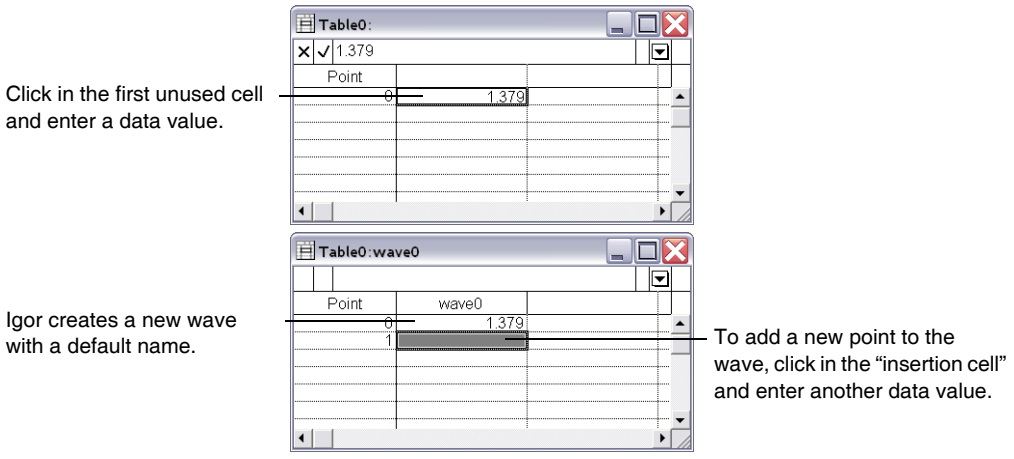
If the decimal symbol is period then the thousands separator is comma. If the decimal symbol is comma then the thousands separator is period. The thousands separator is permitted when entering data in a table. You can also choose a column numeric format that displays thousands separators. However thousands separators are not permitted when creating new waves by pasting text into a table.

Using a Table to Create New Waves

If you click in any unused column, Igor will select the first cell in the first unused column. You can then create new waves by entering a value or pasting data that you have copied to the Clipboard.

Creating a New Wave by Entering a Value

When you enter a data value in the first unused cell, Igor creates a single new 1D wave and displays it in the table. This is handy for entering a small list of numbers or text items. If you enter a numeric value, including date/time values, Igor creates a numeric wave. If you enter a nonnumeric value, Igor creates a text wave.



Igor gives the wave a default name, such as wave0 or wave1. You can rename the wave using the Rename item in the Data menu or the Rename item in the Table pop-up menu. You can also rename the wave from the command line by simply executing:

```
Rename oldName, newName
```

When you create a new wave, the wave has one data point — point 0. The cell in point number 1 will appear gray. This is the **insertion cell**. It indicates that the preceding cell is the last point of the wave. You can click in the insertion cell and enter a value or do a paste. This adds one or more points to the wave.

If the new wave is numeric, it will be single or double precision, depending on the Default Data Precision setting in the Miscellaneous Settings dialog. The number of digits displayed, however, depends on the numeric format. See **Numeric Formats** on page II-215.

If you enter a date (e.g., 1/26/93), time (e.g., 10:23:30) or date/time (e.g., 1/26/93 10:23:30) value, Igor will notice this. It will set the column's numeric format to display the value properly. It will also force the new wave to be double precision, regardless the Default Data Precision setting in the Miscellaneous Settings dialog. This is necessary because single precision does not have enough range to store date and time values.

Creating New Waves by Pasting Data from Another Program

If you have data in a spreadsheet program or other graphing program, you may be able to import that data into Igor using the copy and paste technique.

This will work if the other program can copy its data to the Clipboard as tab-delimited text. Most programs that handle data in columns can do this. Tab-delimited data consists of a number of lines of text with following format:

```
value <tab> value <tab> value <terminator>
```

It may start with a line containing column names. The end of a line is marked by a terminator which may be a carriage return, a linefeed, or a carriage return/linefeed combination. If pasted into a word processor, tab delimited text would look something like this:

```
column1     column2     column3     (this line is optional)
27.95       -13.738     12.74e3
31.37       -12.89      13.97e3
.           .           .
.           .           .
.           .           .
```

In the other program, select the cells containing the data of interest and copy them to the Clipboard. In Igor, select the first cell in the first unused column in a table and then select Paste from Igor's Edit menu.

Igor scans the contents of the Clipboard to determine the number of rows and columns of numeric text data. It also checks the first line of text in the Clipboard to see if it contains column names. It creates waves and displays them in the table using the names found in the Clipboard or default names. If the text contains names which conflict with existing names, Igor presents a dialog in which you can correct the problem.

If you attempt to paste nonnumeric data that does not start with a line of column names, Igor will use the first line of data as column names. This, of course, is not what you want. There are three solutions. First, you can paste the text into an Igor notebook, add a line of column names at the top, recopy all of the text and then paste it into the table. Second, you can create some one point text waves by entering text values in the table, select the newly created text values and overwrite them by doing a paste. Third, you can use the Load Delimited Text routine to load data from the Clipboard. Load Delimited Text has an option not to look for column names. See **Loading Delimited Text Files** on page II-143 for details.

Troubleshooting

If the waves that are created when you paste don't contain the values you expect, chances are that the Clipboard does not contain tab-delimited text. In this case you will need to undo the paste. To examine the contents of the Clipboard, paste it into an Igor plain text notebook or into the word processor of your choice. After editing the text, copy it to the Clipboard again and repaste it into the table.

Creating New Waves by Pasting Data from Igor

You can also create new waves by copying data from existing waves. When you copy wave data in a table, Igor stores not only the raw data but also the following properties of the wave or waves:

- Data units and dimension units
- Data full scale and dimension scaling
- Dimension labels
- The wave note

This feature duplicates a wave by copying it in a table and pasting into the unused area of the same table or a different table. You can also copy from a table in one experiment and paste in a table in another experiment.

You can copy and paste the wave note only if you copy the entire wave. If you copy part of the wave, it does not copy the wave note.

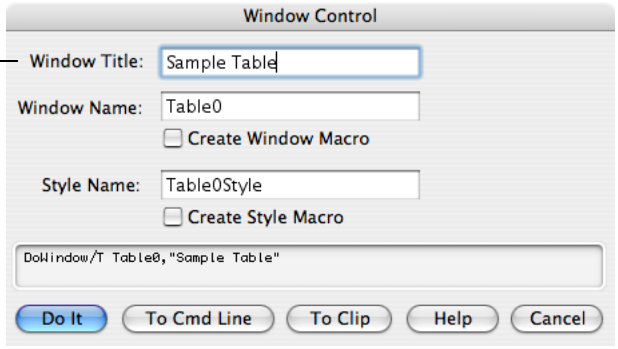
Table Names and Titles

Every table that you create has a name. The name is a short Igor object name that you or Igor can use to reference the table from a command or procedure. When you create a new table, Igor assigns it a name of the form Table0, Table1 and so on. You will most often use a table's name when you kill and recreate the table, as described in the next section.

A table also has a title. The title is the text that appears at the top of the table window. Its purpose is to identify the table visually. It is not used to identify the table from a command or procedure. The title can consist of any text, up to 255 characters.

You can change the name and title of a table using the Window Control dialog. This dialog is a collection of assorted window-related things. Choose Window Control from the Control submenu of the Windows menu.

The Window Control dialog changes a table's title and name.



Hiding and Showing a Table

You can hide a table by Shift-clicking the close button.

If the Minimize Is Hide checkbox is selected in the Miscellaneous Settings dialog (Misc menu), you can hide a table by clicking the minimize icon. You can hide tables by clicking the minimize button while pressing Option (*Macintosh*) or Alt (*Windows*).

You can show a table by choosing its name from the Windows→Tables submenu.

Killing and Recreating a Table

Igor provides a way for you to kill a table and then later to recreate it. Use this to temporarily get rid of a table that you expect to be of use later.

You kill a table by clicking the table window's close button or by using the Close item in the Windows menu. When you kill a table, Igor offers to create a **window recreation macro**. Igor stores the window recreation macro in the procedure window of the current experiment. You can invoke the window recreation macro later to recreate the table. The name of the window recreation macro is the same as the name of the table.

A table does not contain waves but is just a way of viewing them. Killing a table does not kill the waves displayed in a table. If you want to kill the waves in a table, select all of them (Select All in Edit menu) and then choose Kill All Selected Waves from the Table pop-up menu.

For further details, see **Closing a Window** on page II-59 and **Saving a Window as a Recreation Macro** on page II-61.

Index Columns

There are two kinds of numeric values associated with a numeric wave: the stored data values and the computed index values. For example, each point in a real 1D wave has two values: a data value and an X index value. The data value is stored in memory. The X value is computed based on the point number and the wave's X scaling property. The correspondence between point numbers and X values is discussed in detail under **Waveform Model of Data** on page II-77.

Because the index values for a wave are computed, *a value in an index column in a table can not be altered by editing the wave*. Only values in data columns of a table can be edited. To alter the index values of a wave, use the Change Wave Scaling dialog.

Column Names

Column names are related to but not identical to wave names. You need to use column names to append, remove or modify table columns from the command line or from an Igor procedure.

A column name consists of a wave name and a suffix that identifies which part of the wave the column displays. For each real 1D wave there can be two columns: one for the X index values or dimension labels of the wave and one for the data values of the wave. For complex waves there can be three columns: one for the X index values or dimension labels of the wave, one for the real data values of the wave and one for the imaginary data values of the wave.

If we have a real 1D wave named "test" then there are three column names associated with that wave: test.i ("i" for "index"), test.l ("l" for "label") and test.d ("d" for "data"). If we have a complex 1D wave named "ctest" then there are four column names associated with that wave:ctest.i, ctest.l, ctest.d.real and ctest.d.imag.

Wave Name	Column Name	Column Contents
test	test.i	Index values of test
test	test.l	Dimension labels of test
test	test.d	Data values of test
ctest	ctest.i	Index values of ctest
ctest	ctest.d.real	Real part of data values of ctest
ctest	ctest.d.imag	Imaginary part of data values of ctest

For multidimensional waves, the ".i" and ".l" suffixes still specify a single column of index values or dimension labels while the ".d" suffix specifies all of the data columns.

In the table-related commands, you can abbreviate column names as follows:

Full Column Specification	Abbreviated Column Specification
test.d	test
test.i, test.d	test.id
test.l, test.d	test.ld
ctest.d.real, ctest.d.imag	ctest.d or ctest
ctest.i, ctest.d.real	ctest.id.real
ctest.l, ctest.d.real	ctest.ld.real
ctest.i, ctest.d.imag	ctest.id.imag
ctest.l, ctest.d.imag	ctest.ld.imag
ctest.i,ctest.d.real,ctest.d.imag	ctest.id
ctest.l,ctest.d.real,ctest.d.imag	ctest.ld

A 2D wave has X and Y index values. A 3D wave has X, Y and Z index values. A 4D wave has X, Y, Z and T index values. Regardless of the dimensionality of the wave, however, it has only one index column in a table. The index column for a 2D wave, for example, may show the X values or the Y values, depending on how you are viewing the data. The index column will be labeled "wave.x" or "wave.y", depending on the view. However, when referring to the column from an Igor command, you must always use the generic "wave.i". A dimension label column is always called "wave.l", regardless of which dimension is showing in the table.

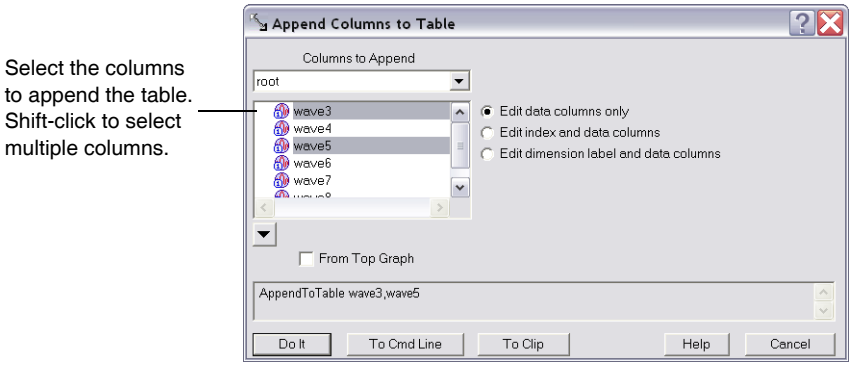
Prior to Igor Pro 3.0, Igor supported only 1D waves. The suffix used for the index column was ".x" and the suffix used for the data column (previously called the Y column) was ".y". For compatibility with earlier versions of Igor, Igor Pro still accepts the old suffixes. Furthermore, when Igor generates a table recreation macro, it uses the old suffixes for 1D waves to make it easier for you to open a new experiment in an older version of Igor.

The ability to display dimension labels was added in Igor Pro 4.0. Earlier versions of Igor will generate an error if they encounter column names that end with ".l".

See the section on **Edit** on page V-138 for some examples of commands using column names.

Appending Columns

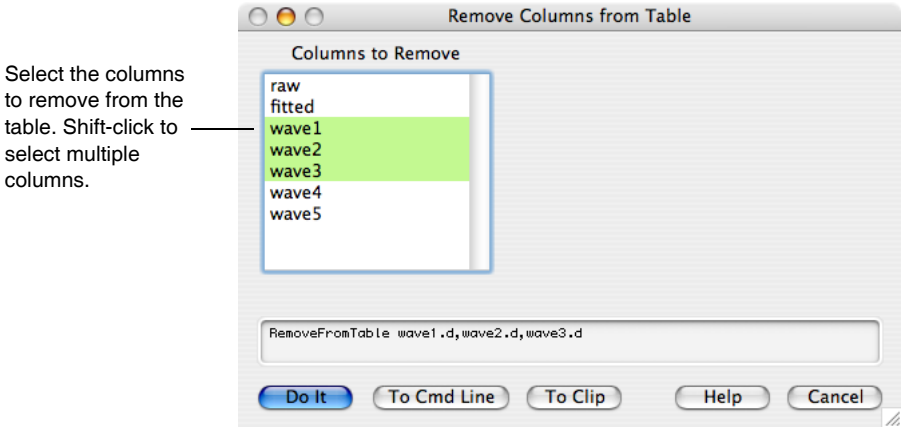
To append columns to a table, choose Append Columns to Table from the Table menu.



Igor appends columns to the right end of the table. You can drag a column to a new position by pressing Option (*Macintosh*) or Alt (*Windows*) and dragging the column name.

Removing Columns

To remove columns from a table, choose Remove Columns from Table from the Table menu.



You can also select the columns in the table, and use the Table pop-up menu to remove the selected columns.

Note that removing a column from a table does not kill the underlying wave. The column is not the wave but just a *view* of the wave. Use the Kill Waves item in the Table pop-up menu to remove waves from the table and kill them. Use the Kill Waves item in the Data menu to kill waves you have already removed from a table.

Selecting Cells

If you click in a cell, it becomes the target cell. The old target cell is deselected and the cell you clicked on is highlighted. The target cell ID changes to reflect the row and column of the new target cell and the value of the target cell is shown in the entry line. You click in a cell and make it the target when you want to enter a new value for that cell or because you want to select a range of cells starting with that cell.

Here are the selections that you can make:

Click	Action
Click	Selects a single cell and makes it the target cell
Shift-click	Extends or reduces the selection range
Click in the point column	Selects the entire row
Click in a column name	Selects the entire column
Click in an unused column	Selects the first unused cell
Choose Select All (Edit menu)	Selects all cells (if possible)

The selection in a table must be rectangular. Igor will not let you select a range that is not rectangular. If you choose Select All, Igor will attempt to select all of the cells in the table. However, if you have columns of different length, Igor will be limited to selecting a rectangular array of cells.

If, after clicking in a cell to make it the target cell, you drag the mouse, the cells over which you drag are selected and highlighted to indicate that they are selected. You select a range of cells in preparation for copying, cutting, pasting or clearing those cells. While you drag, the cell ID area shows the number of rows and columns that you have currently selected. If you drag beyond the edges of the table, the cells will scroll so that you can select as many cells as you want.

Moving the target cell accepts any data entry in progress.

You can change which cell is the target cell using Return, Enter, Tab, or arrow keys. If you are entering a value, these keys also accept the entry.

Key	Action (When a Single Cell is Selected)
Return, Enter, Down Arrow	Moves target cell down
Shift-Return, Shift-Enter, Up Arrow	Moves target cell up
Tab, Right Arrow	Moves target cell right
Shift-Tab, Left Arrow	Move target cell left

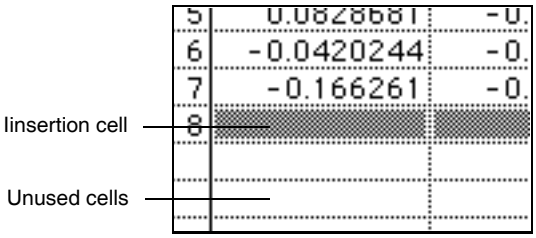
If you have a range of cells selected, these keys keep the target cell within that selected range. If it is at one extreme of the selected range it will wrap around to the other extreme.

Since the arrow keys move the target cell, you cannot use them to move the insertion point in the entry line.

The used columns in a table are always contiguous. If you click in any unused column, Igor selects the first unused cell. There are just two things you can do when the first unused cell is selected: create a new wave by entering a value or create new waves by pasting data from the Clipboard. Igor will not allow you to select any unused cell other than the first cell in the first unused column.

The Insertion Cell

At the bottom of every column of data values is a special cell called the **insertion cell**. It appears after the very last point in a wave.



Sometimes you know the number of points that you want a wave to contain and don't need to insert additional points into the wave. However, if you want to enter a short list of values into a table or to add new data to an existing wave, you can do this by entering data in the insertion cell.

When you enter a value in an insertion cell, Igor extends the wave by one point. Then the insertion cell moves down one position and you can insert another point.

The insertion cell can also be used to a extend a wave or waves by more than one point at a time. This is described under **Pasting Values** on page II-205.

You can also insert points in waves using the Insert Points item which appears in both the Table pop-up menu and the Data menu or using the InsertPoints operation from the command line.

Entering Values

You can alter the data value of a point in a wave by making the cell corresponding to that value the target cell, typing the new value in the entry line, and then confirming the entry.

You can also accept the entry by clicking in any cell or by pressing any of the keys that move the target cell: Return, Enter, Tab, or arrow keys. You can discard the entry by pressing Escape or by clicking the X icon.

If a range of cells is selected when you confirm an entry, the target cell will move within the range of selected cells unless you click in a cell outside this range.

While you are in the process of entering a value, the Clear, Copy, Cut and Paste items in the Edit menu as well as their corresponding command or accelerator key shortcuts affect the entry line. If you are not in the process of entering, these operations affect the cells.

Entering a value in an insertion cell is identical to entering a value in any other cell except that when the entry is confirmed the wave is extended by one point.

Igor will not let you enter a value in an index column since index values are computed based on a waves dimension scaling.

Dimension labels are limited to 31 characters. If you paste into a dimension label cell, Igor will clip the pasted data to 31 characters.

When entering a value in a numeric column, if what you have entered in the entry line is not a valid numeric entry, Igor will not let you confirm it. The check icon will be dimmed to indicate that the value can not be entered. What is valid depends on a column's numeric format. If a column is formatted to display data as dates then 1.234 is not valid. If a column is formatted to display data as numbers then 1/30/93 is not valid. In a numeric column, text like "Blank" or "January" is never valid. In a text column, anything is valid.

Date Values

Dates and times are represented in Igor date format — as a number of seconds since midnight, January 1, 1904. Dates before that are represented by negative values. There is no practical limit to the range of dates that can be represented except that on Windows dates must be greater than January 1, 1601.

A date can not be accurately stored in the data values of a single precision wave. Make sure to use double precision to store dates and times.

The way you enter dates and the way that Igor displays them in tables is controlled by the Table Date Format dialog which you invoke through the Table menu. This dialog sets a global preference that determines the date format for all tables in all experiments.

If in the Table Date Format dialog you choose to use the system date format, which is the factory default setting, Igor always displays dates in a table using the short date format as set by the Date & Time control panel (*Macintosh*) or by the Regional Settings control panel (*Windows*). However, when parsing dates that you enter in the table, Igor accepts only the following formats:

```
mm/dd/yy    mm-dd-yy    mm.dd.yy    // Month-before-day format
dd/mm/yy    dd-mm-yy    dd.mm.yy    // Day-before-month format
```

You can omit the day and just enter:

```
mm/yy mm-yy mm.yy             // Month-before-day format
mm/yy mm-yy mm.yy             // Day-before-month format
```

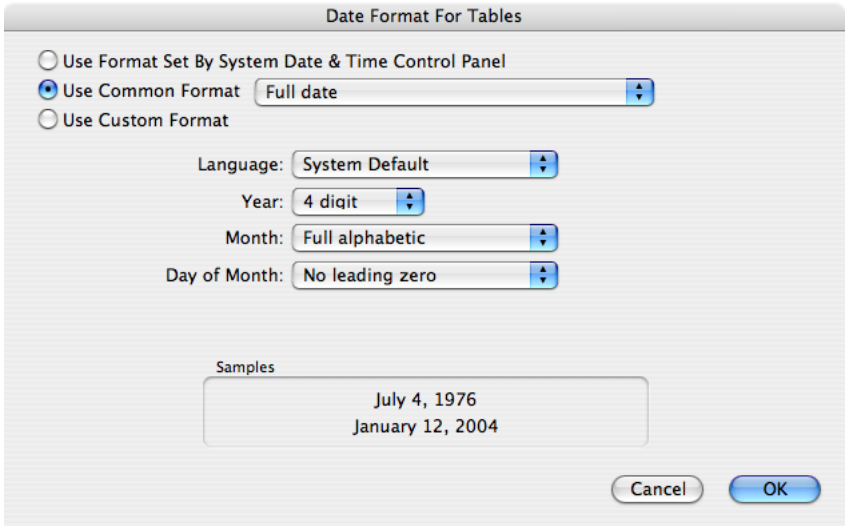
In this case, the day is assumed to be 1.

If the system short date shows the day before the month then Igor expects you to use the day-before-month format. Otherwise, it expects you to use the month-before-day format.

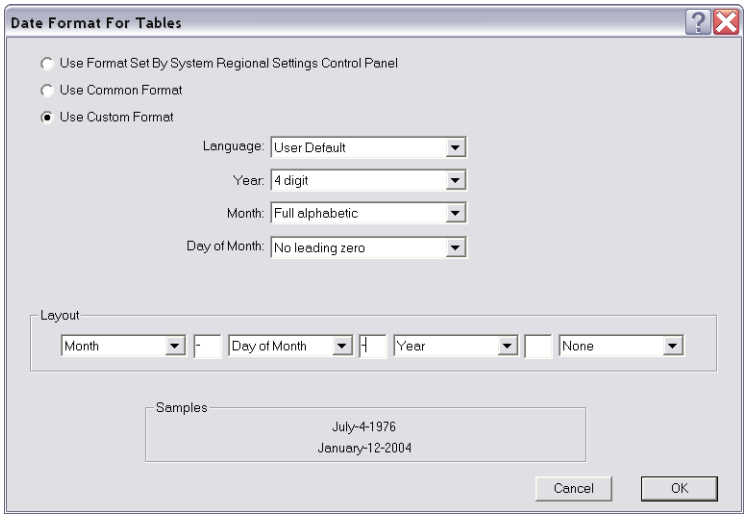
The year can be entered using two digits (99) or four digits (1999). If a two digit year is in the range 00 … 39, Igor treats this as 2000 … 2039. If a two digit year is in the range 40 … 99, Igor treats this as 1940 … 1999.

Prior to Igor Pro 4.0, these were the only supported date formats. They are sufficient for most purposes.

On the other hand, if in the Table Date Format dialog you choose to use a common date format or a custom date format, then Igor uses the same format for display and entry. When parsing entered dates, Igor expects you to enter the exact format that you specify. These methods were added in Igor Pro 4.0. Here is what the dialog looks like if the Use Common Format radio button is selected:



You can access even more flexibility in those rare cases where it's needed by clicking the Use Custom Format radio button:



If you specify no separator in a common or custom date format, things get ambiguous unless leading zeros are used. For example, 11399 could mean 1/13/99 or 11/3/99. For this reason, if you specify no separator, you must also specify leading zeroes for the month and day of month.

When using a common or custom date format that includes separators (e.g., 10/05/99 or 10.05.99), Igor is lenient about the number of digits in the year and whether or not leading zeros are used. Igor will accept two or four digit years and leading zeros or no leading zeros for the year, month, and day of month. However, when using a format with no separators (e.g., 991005 or 19991005), Igor requires that you enter the date exactly as the format specifies.

When you enter a value in the first unused column in a table, Igor must deduce what kind of value you are entering (number, date, time, date/time, or text). It then sets the column format appropriately and interprets what you have entered accordingly. An ambiguity occurs if you use date formats with no separators. For example, if you enter 991005, are you trying to enter a date or a number? Igor has no way to know. Therefore, if you want to create a new column consisting of dates with no separators, you must choose Date from the Table Format submenu before you enter the value. This is not necessary for dates that include separators because Igor can distinguish them from numbers.

If you choose a date format that includes alphabetic characters, such as "October 11, 1999", you must enter dates exactly as the format indicates, including spaces.

There is a problem relating to pasting dates like "October 11, 1999" into a table. This problem does not affect entering a date in the table entry line or pasting a date into an existing column. It affects only pasting into an unused column in the table, an action which creates one or more new waves. During the paste, Igor treats the text in the Clipboard as delimited text and accepts tabs or commas as delimiters. Since "October 11, 1999" contains a comma, Igor sees this as two values and will create two waves. The first wave will contain the entire date and the second will contain just the year. There are two workarounds for this problem. One is to do the paste and then kill the column containing just the year. The other is to use the LoadWave operation, via the Load Waves dialog, to load the contents of the Clipboard as delimited text, and to instruct LoadWave to accept just tab, rather than tab and comma, as a delimiter. The choice of delimiters available in the Tweaks subdialog of the Load Waves dialog.

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-102.

Special Values

There are two special values that can be entered in any numeric data column. They are NaN and INF.

Missing Values (NaNs)

NaN stand for "Not a Number" and is the value Igor uses for missing or blank data. Igor displays NaNs in a table as a blank cell. NaN is a legal number in a text data file, in text pasted from the Clipboard and in a numeric expression in Igor's command line or in a procedure.

A point will have the value NaN when a computation has produced a meaningless result, for example if you take the log of a negative number. You can enter a missing value in a cell of a table by entering NaN or by deleting all of the text in the entry line and confirming the entry.

You can also get NaNs in a wave if you load a delimited text data file or paste delimited text which contains two delimiters with no number in between.

Infinities (INFs)

INF stands for "infinity". Igor displays infinities in a table as "INF". INF is a legal number in a text data file, in text pasted from the Clipboard and in a numeric expression in Igor's command line or in a procedure.

A point will have the value INF or -INF when a computation has produced an infinity, for example if you divide by zero. You can enter an infinity in a cell of a table by entering INF or -INF.

Clearing Values

You invoke the clear operation by choosing Clear from the Edit menu. Clear sets all selected cells in text and dimension label columns to zero. It sets all selected cells in text columns to "" (empty string). It has no effect on selected cells in index columns.

To set a block of numeric values to NaN (or any other numeric value), select the block and then choose Analysis→Compose Expression. In the resulting dialog, choose "_table selection_" from the Wave Destination pop-up menu. Enter "NaN" as the expression and click Do It.

Copying Values

You invoke the copy operation by choosing Copy from the Edit menu. Copy copies all selected cells to the Clipboard as text and as Igor binary. It is useful for copying ranges of points from one wave to another, from one part of a wave to another part of that wave, and for exporting data to another application or to another Igor experiment (see **Exporting Data from Tables** on page II-210).

You can also create new waves by copying data from existing waves. This is described earlier in this chapter under **Creating New Waves by Pasting Data from Igor** on page II-197.

See also **Multidimensional Copy/Cut/Paste/Clear** on page II-223.

Cutting Values

You invoke the cut operation by choosing Cut from the Edit menu. Cut starts by copying all selected cells to the Clipboard as text and as Igor binary. Then it deletes the selected points from their respective waves, thereby shortening the waves.

You cannot cut sections of an index column since index values are computed based on point numbers, not stored. However, if you cut a section of a data or dimension label column, the index column corresponding to the data column will also be shortened.

Pasting Values

You invoke the paste operation by choosing Paste from the Edit menu. There are three kinds of paste operations: a replace-paste, an insert-paste and a create-paste.

Paste Type	What You Do	What Igor Does
Replace-paste	Choose Paste.	Replaces the selected cells with data from the Clipboard.
Insert-paste	Press Shift and choose Paste.	Inserts data from Clipboard as new cells.
Create-paste	Click in the first cell in the first unused column and then choose Paste.	Creates new waves containing Clipboard data.

When dealing with multidimensional waves, there are other options. See **Multidimensional Copy/Cut/Paste/Clear** on page II-223 for details.

When you do a paste, Igor starts by figuring out how many rows and columns of values are in the Clipboard. The Clipboard may contain binary data that you just copied from an Igor table or it may contain plain text data from another application such as a spreadsheet or a text editor.

If the data in the Clipboard is plain text, Igor expects that rows of values be separated by carriage return characters, linefeed characters, or carriage return/linefeed pairs and that individual values in a row be separated by tabs or commas. This is normally no problem since most applications export data as tab-delimited text. If you have trouble with a paste and are not sure about the format of the data in the Clipboard, you can paste it into an Igor notebook to inspect or edit it.

Once Igor has figured out how many rows and columns are in the Clipboard, it proceeds to paste those values into the table and therefore into the waves that the table displays.

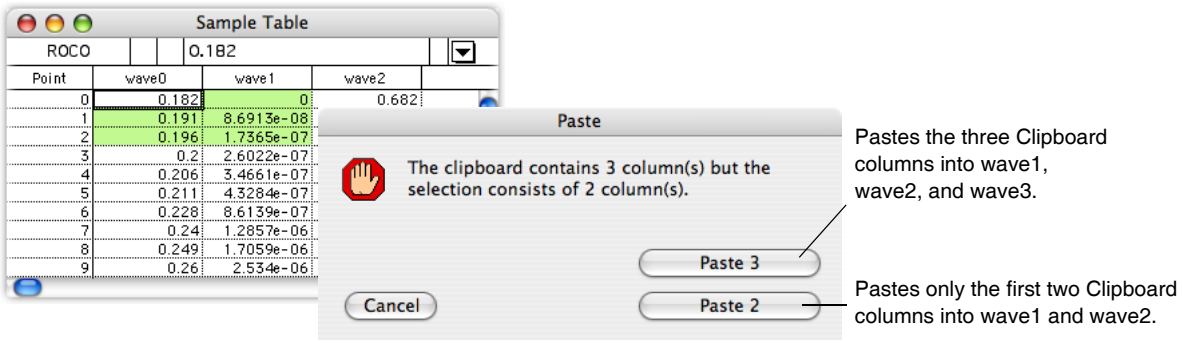
If you select the first cell in the first unused column, the paste will be a create-paste. In this case, Igor makes new waves, appends them to the table and then stores the data in the Clipboard in the new waves. It makes one new wave for each column of text in the Clipboard. If the text starts with a row of column names, Igor will use this row as the basis for the names of the new waves. Otherwise, Igor uses default wave names.

If you are attempting to paste text values, rather than numeric values and if the text in the Clipboard does not include column names, Igor will mistake the first row of values for column names. See **Creating New Waves by Pasting Data from Another Program** on page II-196 for solutions to this problem.

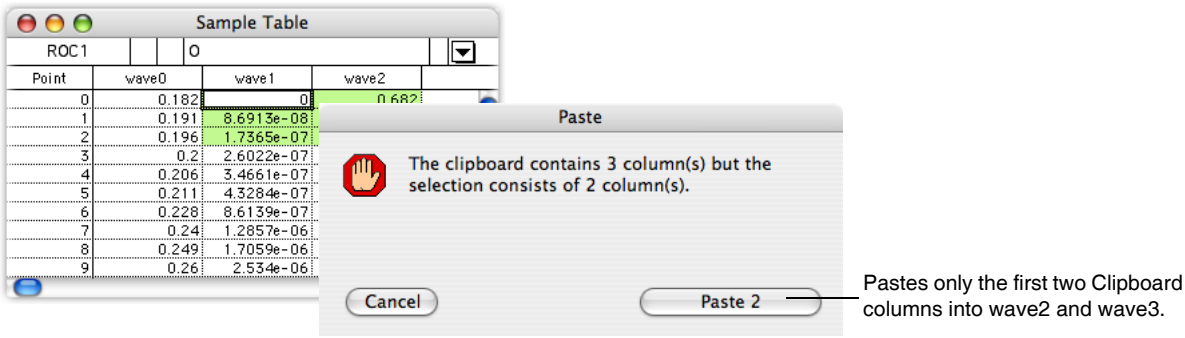
Mismatched Number of Columns

If the number of columns in the Clipboard is not the same as the number of columns selected in the table then Igor will ask you how many columns to paste. This applies to the replace-paste and the insert-paste but not to the create-paste.

For example, assume you have three columns of text in the Clipboard but you select two columns and then do a paste, Igor presents the following dialog:



If you click Paste 3, Igor extends the selection to include the third column before doing the paste. However, Igor can not extend the selection beyond the last *used* column, as illustrated by the following case:



Here Igor does not offer to paste three columns because it would have to extend the selection to include two used columns and one unused column. The resulting paste would be a combination of a replace-paste and an insert-paste. Igor is not able to do this.

Pasting and Index Columns

Since the values of an index column are computed based on point numbers they can not be altered by pasting. However if index columns and data columns are adjacent in a range of selected cells a paste can still be done. The data values will be altered by the paste but the index values will not be altered.

Pasting and Column Formats

When you paste plain text data into existing numeric columns, Igor tries to interpret the text in the Clipboard based on the numeric format of the columns. For example, if a column is formatted as dates then Igor would look for text such as 2/1/93. If the column is formatted as time then Igor would look for text such as 10:00:00. If the column has a regular number format, Igor will look for regular numbers.

When you paste plain text data into unused columns, Igor does a create-paste. In this case, Igor inspects the text in the Clipboard to determine if the data is in date format, time format, date and time format or regular number format. When it appends new columns to the table, it applies the appropriate numeric format.

When pasting octal or hexadecimal text in a table, you must first set the column format to octal or hexadecimal so that Igor will correctly interpret the text.

If the column does not appear to be in any of these formats, Igor creates a text wave rather than a numeric wave.

See **Date Values** on page II-202 for details on entering dates.

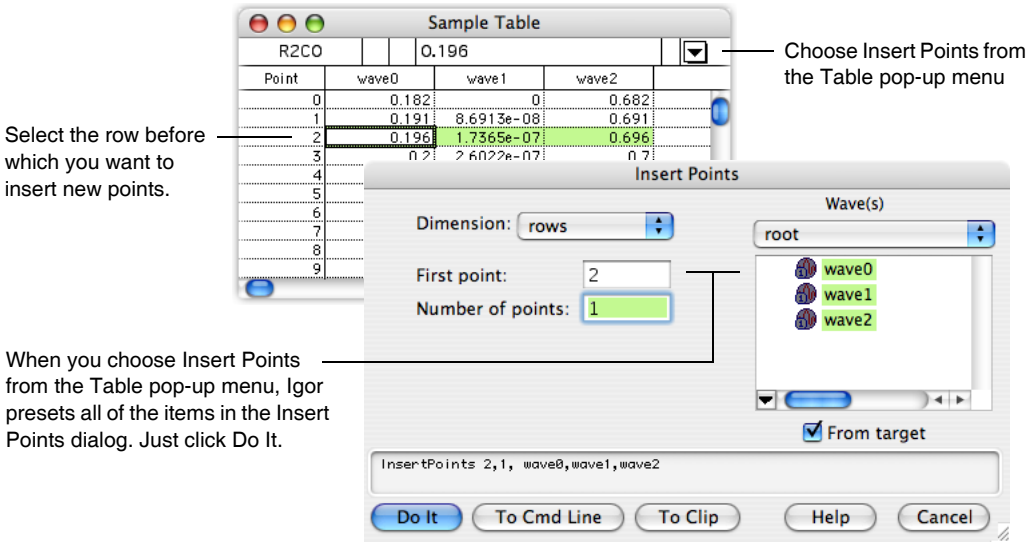
Copy-Paste Waves

You can copy and paste entire waves within Igor. This is described under **Creating New Waves by Pasting Data from Igor** on page II-197.

Inserting and Deleting Points

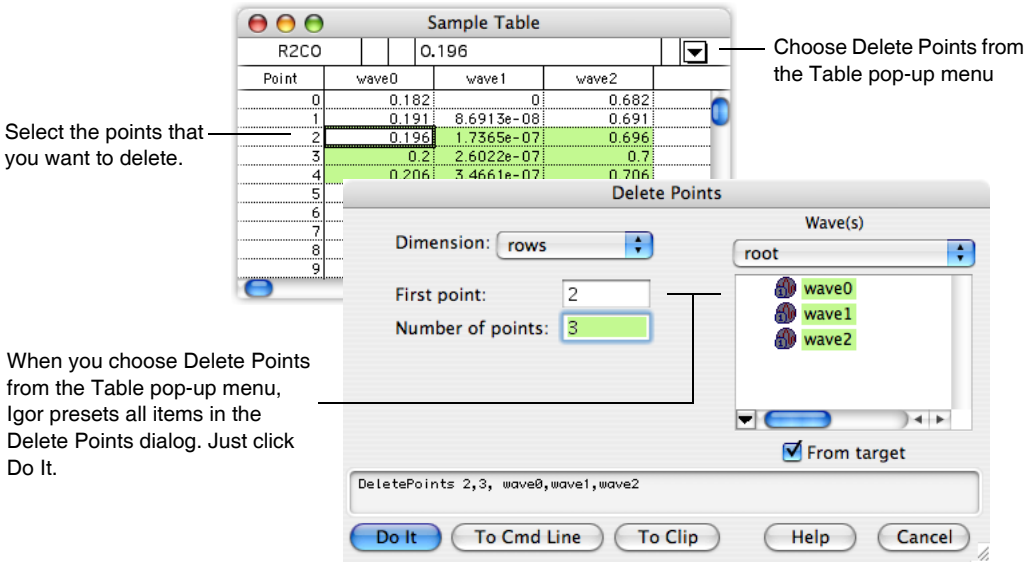
In addition to pasting and cutting, you can also insert and delete points from waves using the Insert Points and Delete Points dialogs via the Data menu or via the Table pop-up menu. You can use these dialogs to modify waves without using a table but they do work intelligently when a table is the top window.

Insert Points Dialog and Tables



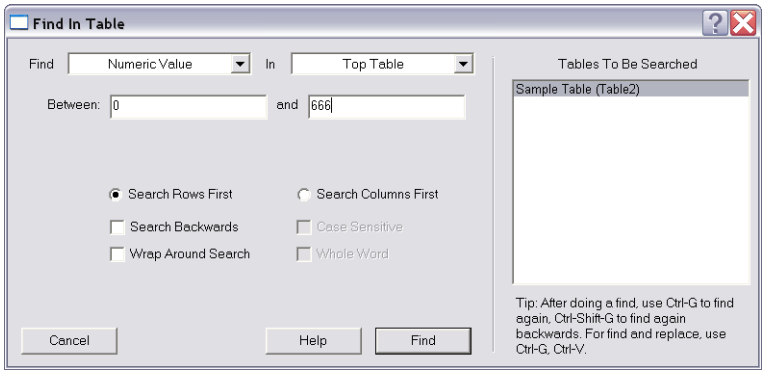
New points are inserted before the row specified by the "First point" parameter. Igor sets the value of the new points to zero.

Delete Points Dialog and Tables



Finding Table Values

You can search tables for specific contents using the Find In Table dialog (Edit→Find).



Find In Table can search the current selection in the active table, the entire active table or all table windows. You control this using the right-hand pop-up menu at the top of the dialog.

The All Table Windows mode searches standalone table windows only. It does not search embedded tables. It is possible to search an embedded table in a control panel using the Top Table mode. Searching in tables embedded in graphs and page layouts is not supported.

Find In Table can search for the following types of values which you control using the left-hand pop-up menu.

Find Type	Description
Row	Displays the specified row but does not select it.
Text String	Finds the specified text string in any type of column: text, numeric, date, time, date/time, and dimension labels.
	For example, searching for "-1" would find numeric cells containing -1.234 and -1e6. A given cell is found only once, even if the search string occurs more than once in that cell.
	The target string is limited to 254 characters.
Blank Cell	Finds blank cells in any type of column: text, numeric, date, time, date/time, and dimension labels. Finds blank cells in numeric columns (NaNs) and text columns (text elements containing zero characters).
Numeric Value	Finds numeric values within the specified range in numeric columns only. Does not search the following types of columns: text, date, time, date/time, and dimension labels.
Date	Finds date values within the specified range in date and date/time columns only. Does not search the following types of columns: text, numeric, time and dimension labels.
	Accepts input of dates in the format specified by Table Date Format dialog (Table menu).
Time of Day	Finds a time of day within the specified range in time and date/time columns only. Does not search the following types of columns: text, numeric, date and dimension labels.
	A time of day is a time between 00:00:00 and 24:00:00. Times are entered as hh:mm:ss.ff with the seconds part and fractional part optional.
Elapsed Time	Finds an elapsed time within the specified range in time columns only. Does not search the following types of columns: text, numeric, date, date/time and dimension labels.
	Unlike a time of day, an elapsed time can be negative and can be greater than 24:00:00. Times are entered as hh:mm:ss.ff with the seconds part and fractional part optional.
Date/Time	Finds a date/time within the specified range in date/time columns only. Does not search the following types of columns: text, numeric, date, time and dimension labels.
	Date/time values consist of a date, a space and a time.

Find In Table does not search the point column.

The search starts from the "anchor" cell. If you are searching the top table or the current selection, the anchor cell is the target cell. If you are searching all tables, the anchor cell is the first cell in the first-opened table (or the last cell in the last-opened table if you are doing a backward search).

When you do an initial search via the Find dialog, the search includes the anchor cell. When you do a subsequent search using Find Again, the search starts from the cell after the anchor cell (or before it if you are doing a backward search).

A find in the top table starts from the target cell and proceeds forward or backward, depending on the state of the Search Backwards checkbox. The search stops when it hits the end or beginning of the table, unless Wrap Around Search is enabled, in which case the whole table is searched.

A find in the current selection also starts from the target cell and proceeds forward or backward, depending on the state of the Search Backwards checkbox. The search stops when it hits the end or beginning of the selection, unless Wrap Around Search is enabled, in which case the whole selection is searched.

If Search Rows First is selected, all rows of a given column are searched, then all rows of the next column. If Search Columns First is selected, all columns of a given row are searched, then all columns of the next row.

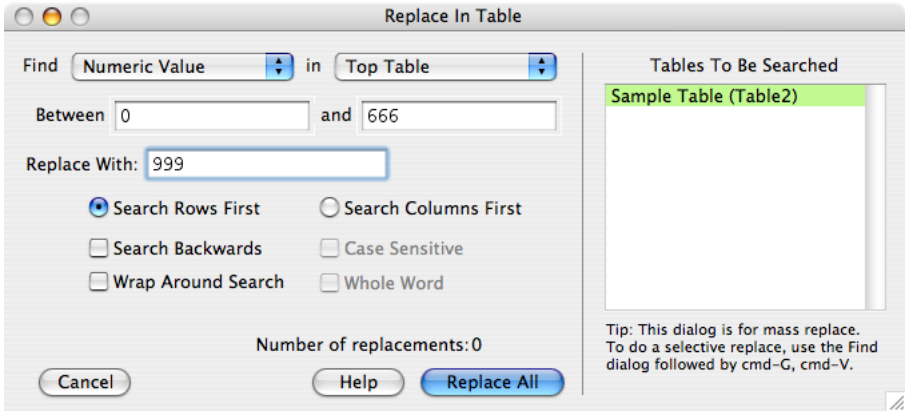
To do a search of a 3D or 4D wave, you must create a table containing just that wave. Then the Table Find will search the entirety of the wave. If the table contains more than one wave, the Table Find will not search the parts (e.g. other layers of a 3D wave) of a 3D or 4D wave that are not shown in the table.

Choosing the Edit→Find Selection menu sets the Find mode to Find Text String, Find Blank Cells, Find Numeric Value, Find Date, Find Time Of Day, Find Elapsed Time, or Find Date/Time based on the format of the target cell except that, if the target cell is blank, the mode is set to Find Blank Cells regardless of the cell's format.

You may find it convenient to use Find Again (Command-G on Macintosh, Ctrl+G on Windows) after doing an initial find to find subsequent cells with the specified contents. Pressing Shift (Comand-Shift-G on Macintosh, Ctrl+Shift+G on Windows) does a Find Again in the opposite direction.

Replacing Table Values

You can perform a mass replace in a table using the Replace In Table dialog (Edit→Replace).



Unlike Find In Table, which can search all tables, Replace In Table is limited to the top table or the current selection, as set by the right-hand pop-up menu.

When you click Replace All, Replace In Table first finds the specified cell contents using the same rules as Find In Table. It then replaces the contents with the specified replace value. It then continues searching and replacing until it hits the end of the table or selection, unless Wrap Around Search is enabled, in which case the whole table or selection is searched.

You can undo all of the replacements by choosing Edit→Undo Replace.

Replace In Table does not affect X columns. You must use the Change Wave Scaling dialog (Data menu) for that.

Replace In Table goes through each candidate cell looking for the specified search value. If it finds the value, it extracts the text from the cell and does the replacement on the extracted text. If the resulting text is legal given the format of the cell, the replacement is done. If it is not legal, Replace In Table stops and displays an error dialog showing where the error occurred.

Here are some additional considerations regarding Replace In Table.

Find Type	Description
Text String	In each cell, all occurrences of the find string are replaced with the replace string. For example, if you replacing "22" with "33" and a cell contains the value 122.223, the resulting value will be 133.333. The replace string is limited to 254 characters. Using text string replace, it is possible to come up with a value that is not legal given a cell's formatting. For example, if you replace "22" with "9.9" in the example above, you get "19.9.9.93". This is not a legal numeric value so Replace In Table displays an error dialog.
Date	You can replace a date in a date column or a date/time column. When replacing a date in a date/time column, the time component is not changed.
Time of Day	You can replace a time of day in a time column or a date/time column. When replacing a time of day in a date/time column, the date component is not changed.

Selectively Replacing Table Values

The Replace In Table dialog is designed to do a mass replace. You can do a selective replace using the Find In Table dialog followed by a series of Find Again and Paste operations. Here is the process:

1. Choose Edit→Find and find the first cell containing the value you want to replace.
2. Edit that cell so it contains the desired value.
3. Copy that cell's contents to the Clipboard.
4. Do Find Again (Command-G on Macintosh, Ctrl+G on Windows) to find the next cell you might want to replace.
5. If you want to replace the found cell, do Paste (Command-V on Macintosh, Ctrl+V on Windows).
6. If not done, go back to step 4.

Exporting Data from Tables

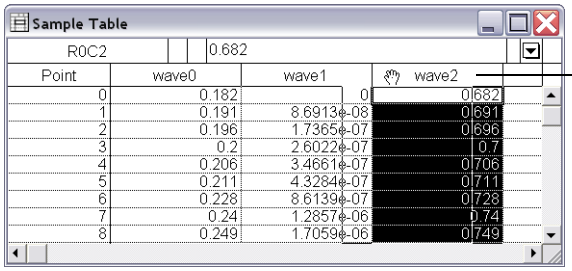
You can use the Clipboard to export data from an Igor table to another application. If you do this you must be careful to preserve the precision of the exported data.

When you copy data from an Igor table, Igor puts the data into the Clipboard in two formats: tab-delimited text and Igor binary. If you later paste that data into an Igor table, Igor uses the Igor binary data so that you retain all precision through the copy-paste operation. However if you paste the data into another application, the other application uses the plain text data in the Clipboard. Therefore the precision of the data in the other application is limited by the numeric output format that you use in the Igor table.

If you intend to export data from an Igor table to another application using the Clipboard, make sure that the numeric format of the data in the table is appropriate for the precision of the data you are exporting. You can set the numeric format for the columns of a table using the Table pop-up menu. See **Changing Column Styles** on page II-212 for details. An alternative method is to use the Save operation, which writes double-precision wave data using 15 digit precision, via Data→Save Waves→Save Delimited Text.

Changing Column Positions

You can rearrange the order of columns in the table. To do this, position the cursor over the name of the column that you want to move. Press Option (*Macintosh*) or Alt (*Windows*) and the cursor changes to a hand. If you now click the mouse you can drag an outline of the column to its new position.

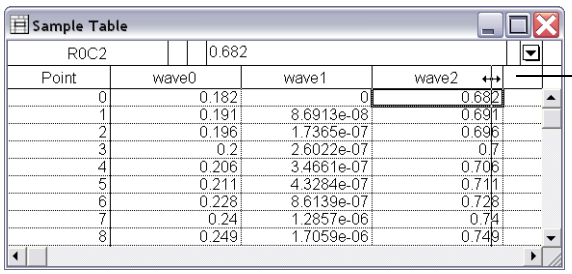


Press Option (*Macintosh*) or Alt (*Windows*), click in the column name and drag.

When you release the mouse the column will be redrawn in its new position. Igor always keeps all of the columns for a particular wave together so if you drag a column, you will move all of the columns for that wave.

The point column can not be moved and is always at the extreme left of the cell area.

Changing Column Widths



Position the cursor over the column divider line and drag.

To set the width of all columns, press Shift.

To set the width of all columns except the Point column, press Shift-Option (*Macintosh*) or Shift+Alt (*Windows*).

You can change the width of a column by dragging the vertical boundary to the right of the column name.

You can influence the manner in which column widths are changed by pressing certain modifier keys.

If Shift is pressed, all table columns except the Point column are changed to the same width.

The Command (*Macintosh*) or Ctrl (*Windows*) key determines what happens when you drag the boundary of a data column of a multidimensional wave. If that key is not pressed, all data columns of the wave are set to the same width. If that key is pressed then just the dragged column is changed.

Autosizing Columns By Double-Clicking

You can autosize a column by double-clicking the vertical boundary to the right of the column name.

You can influence the manner in which column widths are changed by pressing certain modifier keys.

If the no modifier keys are pressed and you double-click the boundary of a data column of a multidimensional wave then the width of each data column is set individually.

When pressing Option (*Macintosh*) or Alt (*Windows*) and you double-click the boundary of a data column of a multidimensional wave then the width of all data columns are set the same.

If Shift is pressed, all table columns except the Point column are autosized. Shift pressed with Option (*Macintosh*) or Alt (*Windows*) will autosize all data columns of a given wave to the same width.

When pressing Command (*Macintosh*) or Ctrl (*Windows*), only the double-clicked column is autosized, not all data columns of a multidimensional wave.

Autosizing Columns Using Menus

You can autosize columns by selecting them and choosing Autosize Columns from the Table menu or from the table popup menu in the top-right corner of the table. You can also choose Autosize Columns from the contextual menu that you get when you Control-click (*Macintosh*) or right-click (*Windows*) on a column.

You can influence the manner in which column widths are changed by pressing certain modifier keys.

If the no modifier keys are pressed and a data column of a multidimensional wave is selected, all data columns of that wave are set individually.

If Option (*Macintosh*) or Alt (*Windows*) is pressed and a data column of a multidimensional wave is selected, all data columns of that wave are set the same.

If Shift is pressed, all table columns except the Point column are autosized. Shift pressed with Option (*Macintosh*) or Alt (*Windows*) will autosize all data columns of a given wave to the same width.

When pressing Command (*Macintosh*) or Ctrl (*Windows*), only the selected columns are autosized, not all data columns of a multidimensional wave.

Autosizing Limitations

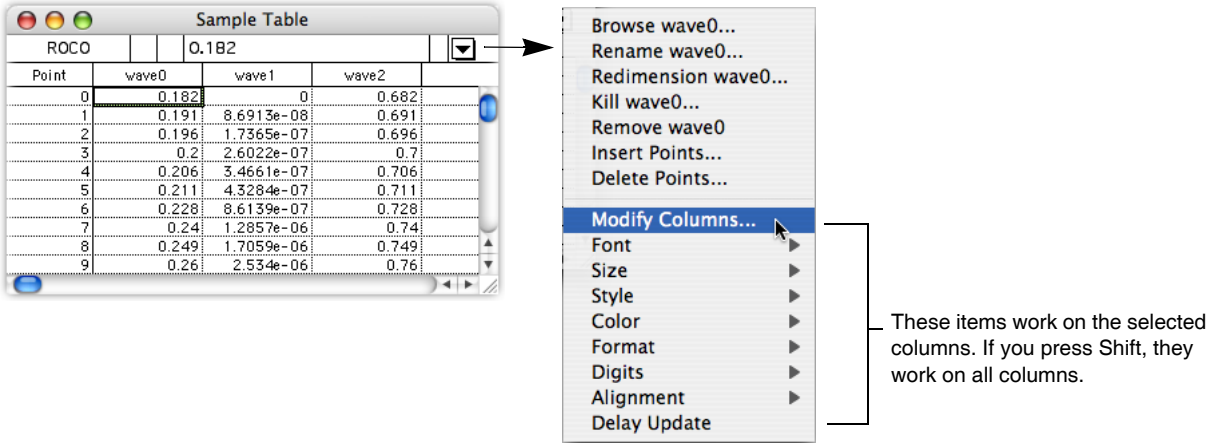
When you autosize a column, the width of every cell in that column must determined. For very long columns (100,000 points or more), this may take a very long time. When this happens, cell checking stops and the autosize is based only on the checked cells.

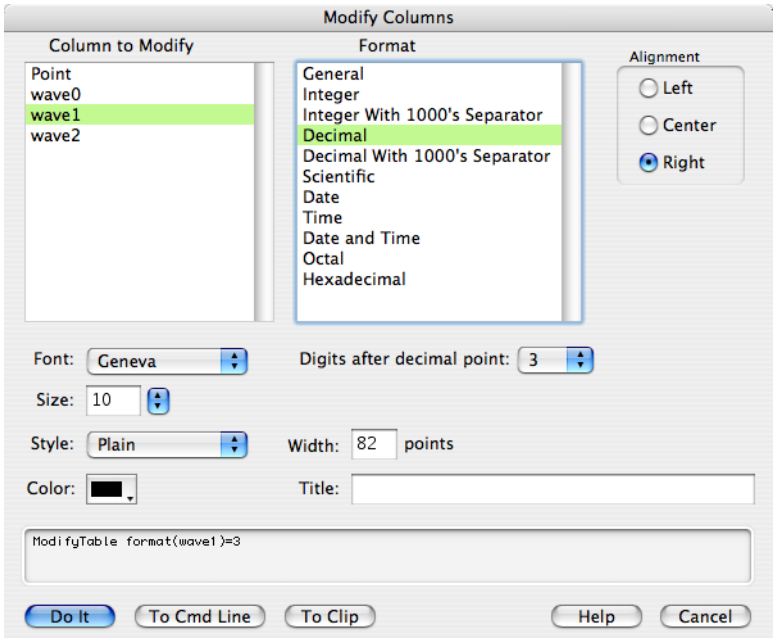
Similarly, if you autosize the data columns of a multidimensional wave with a very large number of columns (10,000 or more columns), this could take a very long time. When this happens, columns checking stops and the autosize is based only on the checked columns.

If the default time limits are not suitable, use the ModifyTable autosize keyword to set the time limits.

Changing Column Styles

You can change the presentation style of columns in a table using the Modify Columns dialog, the Table menu in the main menu bar or the Table pop-up menu.





You can invoke the Modify Columns dialog from the Table pop-up menu or from the Table menu or by double-clicking a column name.

You can select one or more columns in the Columns to Modify list. If you select more than one column, the items in the dialog reflect the settings of the first selected column.

Once you have made your selection, you can change settings for the selected columns. After doing this, you can then select a different column or set of columns and make more changes. Igor remembers all of the changes you make, allowing you to do everything in one trip to the dialog.

There is a shortcut for changing a setting for all columns at once without using the dialog: Press Shift while choosing an item from the Table pop-up menu or from the Table menu in the main menu bar.

Tables are primarily intended for on-screen data editing. You can use a table for presentation purposes by exporting it to another program as a picture or by including it in a page layout. However, it is not ideal for this purpose. For example, there no way to change the background color or the appearance of gridlines.

You can capture your favorite styles as preferences. See **Table Preferences** on page II-228.

Modifying Column Properties

You can independently set properties, such as color, column width, font, etc., for the index or dimension label column and the data column of 1D waves. Except in rare cases all of the data columns of a multidimensional wave should have the same properties. When you set the properties of one data column of a multidimensional wave using Igor's menus or using the Modify Columns dialog, Igor sets the properties of all data columns the same.

For example, if you are editing a 3 x 3 2D wave and you set the first data column to red, Igor will make the second and third data columns will red too.

Despite Igor's inclination to set all of the data columns of a multidimensional wave the same, it is possible to set them differently. Select the columns to be modified. Press Command (*Macintosh*) or Ctrl (*Windows*) before clicking the Table menu or Table pop-up menu and make a selection. Your selection will be applied to the selected columns only instead of to all data columns from the selected waves.

The ModifyTable operation supports a column number syntax that designates a column by its position in the table. Using this operation, you can set any column to any setting. For example:

```
Make/O/N=(3,3) mat
Edit mat
ModifyTable rgb[1]=(50000,0,0), rgb[2]=(0,50000,0)
```

This sets mat's first data column to red and its second to blue.

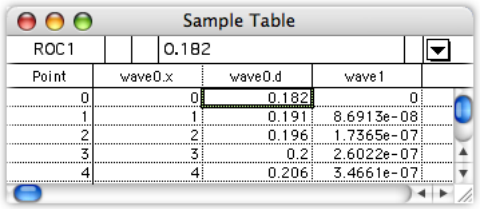
In Igor Pro 6 or later you can specify a range of columns using column number syntax:

```
ModifyTable rgb[1,3]=(50000,0,0)
```

The Modify Columns dialog sets the properties for both the real and imaginary columns of complex waves at the same time. If you really want to set the properties of the real and imaginary columns differently, you must use the column number syntax shown above.

Column Titles

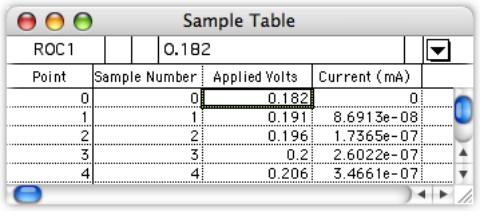
The column title appears at the top of each column. By default, Igor automatically derives the column title from the name of the wave displayed in the column. For example, if we display the X index and data values of wave1 and data values of wave2, the table looks like this:



Point	wave0.x	wave0.d	wave1
0	0	0.182	0
1	1	0.191	8.6913e-08
2	2	0.196	1.7365e-07
3	3	0.2	2.6022e-07
4	4	0.206	3.4661e-07

Note that Igor uses the suffixes ".x" and ".d" only if this is necessary to distinguish columns. If there is no index column displayed, Igor omits the suffix.

Using the Modify Columns dialog, you can replace the automatically derived title with a title of your own. For example:



Point	Sample Number	Applied Volts	Current (mA)
0	0	0.182	0
1	1	0.191	8.6913e-08
2	2	0.196	1.7365e-07
3	3	0.2	2.6022e-07
4	4	0.206	3.4661e-07

Note: This changes *only* the title of the column in this table. It does not change wave names. We provided the column title setting solely to make tables look better in presentations (in layouts and exported pictures). If you are not using a table for a presentation, it is better to let Igor automatically generate column titles since this has less potential for confusion. If you want to rename a wave, use the Rename items in the Data and Table pop-up menus.

Use the Title item in the Modify Columns dialog to set the column title. Enter your own text to turn the automatic title off. Remove all text from this item to turn the automatic title back on.

When you change a display property of a wave in a table using the Table menu or the Modify Columns dialog, Igor sets all columns of multidimensional waves the same. This behavior is reasonable for most of the settings (e.g., font, numeric format) but makes the column title setting useless for multidimensional waves. In most cases, you can achieve what you want using dimension labels instead of column titles. If you really need to use column titles for multidimensional waves, use the techniques described in **Modifying Column Properties** on page II-213 to set the title for individual columns.

Numeric Formats

Columns in tables display either text or numeric waves. For numeric waves, the column format determines how the data values in the wave are entered and displayed. The column format has no effect on data columns of text waves.

In addition to regular number formats, tables support date, time and date&time formats. The format is merely a way of displaying a number. Even dates and times are stored internally in Igor as numbers. You can enter a value in a numeric column of a table as a number, date, time or date&time if you set the format for the column appropriately.

The following table lists all of the numeric formats. The symbol column shows the symbol that appears in the Table pop-up menu and in the Modify Columns dialog.

Numeric Format	Description
General	Displays numbers in a format appropriate to the number itself. Very large or small numbers are displayed in scientific notation. Other numbers are displayed in decimal form (e.g. 1234.567). The Digits setting controls the number of significant digits. Integers are displayed with no fractional digits.
Integer	Numbers are displayed as the nearest integer number. For example, 1234.567 is displayed as 1235.
Integer with comma	Numbers are displayed as the nearest integer number. In addition, commas are used to separate groups of three digits. For example, 1234.567 is displayed as 1,235.
Decimal	As many digits to the left of the decimal point as are required are used to display the number. The Digits setting controls the number of digits to the right of the decimal point. For example, if the number of digits is specified as two, 1234.567 is displayed as 1234.57.
Decimal with comma	Identical to the decimal format except that commas are used to separate groups of three digits to the left of the decimal point.
Scientific	Numbers are displayed in scientific notation. The Digits setting controls the number of digits to the right of the decimal point.
Date	mm/dd/yy or dd/mm/yy or a custom format as set by the Table Date Format dialog. See **Date/Time Formats** on page II-216.
Time	[+][-]hhhh:mm:ss[.ff] [AM/PM]. See **Date/Time Formats** on page II-216.
Date & Time	Date format plus space plus time format. See **Date/Time Formats** on page II-216.
Octal	Numbers are displayed in octal (base 8) notation. The fractional part of the number, if any, is not displayed. The Digits setting controls the number of octal digits displayed.
Hexadecimal	Numbers are displayed in hexadecimal (base 16) notation. The fractional part of the number, if any, is not displayed. The Digits setting controls the number of octal digits displayed.

When entering a number in a table, Igor normally interprets a dot character as the decimal separator. To enter the first four digits of π, you must enter 3.141. However, if the decimal separator in the format section of the International control panel (*Mac OS X*) or Regional Settings control panel (*Windows*) is set to comma, Igor will accept comma as well as dot. You can enter either 3,141 or 3.141 for π. This feature applies *only* to entering numbers in tables. It is intended to allow European users to conveniently use the numeric keypad. This feature does not apply to entering the fractional part of a time value with fractional seconds. In this case, you must enter a dot.

For each numeric column, you can control the number of digits displayed. You can set this using the Modify Columns dialog or using the Table→Digits menu. The meaning of the number that you choose from the Digits submenu depends on the numeric format.

Numeric Format	You Specify
General	Number of displayed digits
Decimal (0.0...0)	Number of digits after the decimal point
Decimal with comma (0.0...0)	Number of digits after the decimal point
Time and Date&Time	Number of digits after the decimal point when displaying fractional seconds
Scientific (0.0...0E+00)	Number of digits after the decimal point
Octal	Total number of octal digits to display.
Hexadecimal	Total number of hexadecimal digits to display.

The Digits submenu has no effect on columns displayed using the integer, date formats and also has no effect on columns displaying text waves.

With the General format, you can choose to display trailing zeros or not.

With the time format, Igor accepts and displays times from -9999:59:59 to +9999:59:59. This is the supported range of elapsed times. If you are entering a time-of-day rather than an elapsed time, you should restrict yourself to the range 00:00:00 to 23:59:59.

With the Time and Date&Time formats, you can choose to display fractional seconds. Most people dealing with time data use whole numbers of seconds. Therefore, by default, a table does not show fractional seconds. If you want to see fractional seconds in a table, you must choose Show Fractional Seconds from the Table→Format menu. Once you do this, the Table→Digits menu controls the number of digits that will appear in the fractional part of the time.

If you always want to see fractional seconds, use the Capture Table Prefs dialog to capture columns whose Show Fractional Seconds setting is on. This applies to tables created after you capture the preference.

When displaying fractional seconds, Igor always displays trailing zeros and the Show Trailing Zeros menu item in the Table→Format menu has no effect.

When choosing a format, remember that single precision floating point data stores about 7 decimal digits and double-precision floating point data stores about 15 decimal digits. If you want to inspect your data down to the last decimal place, you need to select a format with enough digits.

The format affects the precision of data that you export via the Clipboard from a table to another application. See **Exporting Data from Tables** on page II-210.

Date/Time Formats

The manner in which Igor displays dates and interprets dates that you enter is described under **Date Values** on page II-202. The factory default date format is one of the following, depending on the default system date format:

```
mm/dd/yy    mm-dd-yy    mm.dd.yy    // Month-before-day format
dd/mm/yy    dd-mm-yy    dd.mm.yy    // Day-before-month format
```

Other date formats are supported as described in **Date/Time Formats** on page II-144.

If you set the column format to time, then Igor displays time in elapsed time format. You can enter elapsed times from -9999:59:59 to +9999:59:59. You can precede an elapsed time with a minus sign to enter a negative elapsed time. You can also enter a fractional seconds value, for example 31:35:20.19. To view fractional seconds, choose Show Fractional Seconds from the Format submenu of the Table menu.

You can also enter times in time-of-day format, for example 1:45 PM or 1:45:00 PM or 13:45 or 13:45:00. A space before the AM or PM is allowed but not required. AM and am have no effect on the resulting time. PM adds 12 hours to the time. PM has no effect if the hour is already 12 or greater.

Igor stores times the same way whether they are time-of-day times or elapsed times. The difference is in how you think of the value and how you choose to display it in a graph.

Here are some valid time-of-day times:

```
00:00:00 (midnight)           12:00:00 (noon)
06:30:00 (breakfast time)     18:30:00 (dinner time)
06:30 (breakfast time)        18:30 (dinner time)
06:30 PM (dinner time)        18:30 PM (dinner time)
```

Here are some valid elapsed times:

```
-00:00:10   (T minus 10 and counting)
72:00:00    (an elapsed time covering three days)
4:17:33.25  (an elapsed time with fractional seconds)
```

Octal and Hexadecimal Formats

You can edit data values using octal (base 8) or hexadecimal (base 16) notation by choosing Octal or Hexadecimal from the Format submenu of the Table menu. These formats are usually used to display integer data acquired from data acquisition hardware.

You can display any numeric wave as octal or hexadecimal. For display purposes, Igor treats the data as an unsigned 32 bit number. Any fractional part is not displayed in the table.

You must set the number of digits to display using the Digits submenu of the Table menu or the Digits control in the Modify Columns dialog or the sigDigits keyword in a ModifyTable command. The appropriate setting depends on the source of your data. Typically it will be 3 or 6 digits for octal and 2, 4 or 8 digits for hexadecimal.

An 8 digit hexadecimal number represents 32 bits of data. Single precision floating point waves have only about 24 bits of precision. Therefore, if you want to store 32 bit values you must use either the integer, unsigned integer, or double-precision floating point data types for your waves.

When pasting octal or hexadecimal text in a table, make sure that the column format is set to octal or hexadecimal so that Igor will correctly interpret the text.

Editing Text Waves

You can create a text wave in a table by clicking in the first unused cell and entering nonnumeric text. For the most-part, editing a text wave is self-evident. However, there are some issues, mostly relating to special characters or large amounts of text, that you may run into.

Large Amounts of Text in a Single Cell

A text wave is handy for storing short descriptions of corresponding data in numeric waves. In some cases, you may find it useful to store larger amounts of text. There is no limit on the number of characters in a point of a text wave. However, the entry area of a table can display no more than 32,000 characters. You cannot edit in a table a cell containing more than 32,000 characters.

The entry area of the table is capable of scrolling so that you can see and edit long text wave contents. The scroll controls appear only if the active cell contains text that will not fit in one line.

Tabs, CRs and Invisible Characters

In rare cases, you may want to store text containing special characters such as tabs and carriage-returns in text waves. Normally, the Tab and Return keys move the active cell. On a *Macintosh*, if instead you want to enter a tab or return character into the active cell, press Option while pressing Tab or Return.

On *Windows*, it is not possible to enter a carriage return or tab character into a text wave cell via a table. The only way to do this is via the command line. For example, to enter "Hello<tab>Goodbye" on the Macintosh, you can press Option-Tab to enter the tab character. On Windows, you would need to execute the following from the command line:

```
textWave0[<point number] = "Hello\tGoodbye"
```

The reason for this difference is that all key combinations such as Alt+Tab, Shift+Tab and Ctrl+Tab are used for other purposes either by Igor or by Windows.

On the *Macintosh*, "Hello\rGoodbye" appears in the table entry area as two lines of text. On *Windows*, it appears as "Hello<box>Goodbye", where <box> is a character that looks like a box. This difference arises from a difference in the text display routines supplied by the two operating systems.

Macintosh: A return character is invisible when viewed in the body of a table. This is because a return character has no visible representation in a font. There are other special characters that are invisible and this varies from font to font. In general, this is not a problem because you have to go to some lengths to get invisible characters into a text wave.

If you suspect that there may be invisible characters in your text wave, display it in a table and set the column font to Chicago. This font has few invisible characters. The following commands create and display a text wave containing all of the character codes (0 to 255).

```
Make/T/N=256 characters = num2char(p)
Edit characters; ModifyTable font(characters)=Chicago
```

Windows: Tabs, linefeeds, carriage returns, and other special characters appear in the table body as boxes.

Copying and pasting cells containing tabs and return characters between tables will work as expected. However, if you paste text from other types of windows or from other programs, the tabs and CRs will be considered to delimit columns or rows and will not be pasted in the cells themselves.

Treatment of Names When Pasting Text

When pasting data into the unused area of a table, Igor creates a new wave or waves. In this case, Igor looks for wave names in the first row of the pasted text. If you are pasting text data that does not start with wave names, Igor will mistakenly take your first row of data as wave names. See **Creating New Waves by Pasting Data from Another Program** on page II-196 for solutions to this problem.

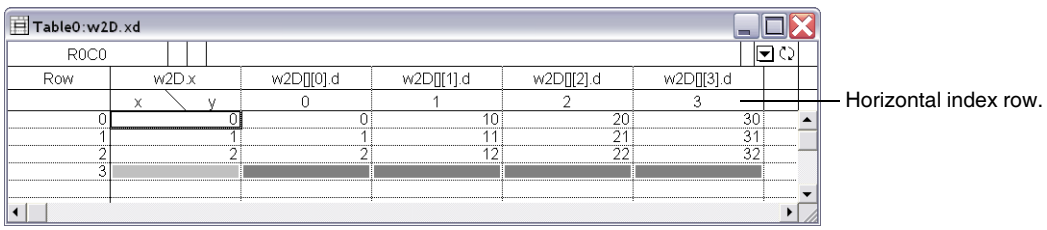
Tab Separators in Text

When pasting data into existing waves in a table, Igor expects columns to be separated by tabs. Previous versions of Igor accepted tabs or commas. This was changed to conform to the behavior of other graphing and spreadsheet programs and so that you can paste text containing a comma into a single cell of a text wave.

Editing Multidimensional Waves

If you view a multidimensional wave in a table, Igor adds some items to the table that are not present for 1D waves. To see this, execute the following commands which create and display a 2D wave:

```
Make/O/N=(3,4) w2D = p + 10*q; Edit w2D.id
```



The first column in the table is labeled Row, indicating that it shows row numbers. The second column contains the scaled row indices, which in this case are the same as the wave row numbers. The remaining columns show the wave data. Notice the name at the top of the first column of wave data: "w2D[][0].d".

The "w2D" identifies the wave. The ".d" specifies that the column shows wave data rather than wave indices. The "[][0]" identifies the part of the wave shown by this column. The "[]" means "all rows" and the "[0]" means column 0. This is derived from the syntax that you would use from Igor's command line to store values into all rows of column 0 of the wave:

```
w2D[][0] = 123        // Set all rows of column 0 to 123
```

When displaying a multidimensional wave in a table, Igor adds a row to the table below the row of names. This row is called the horizontal index row. It can display either the scaled indices or the dimension labels for the wave elements shown in the columns below.

By default, if you view a 2D wave in a table and append the wave's index column, Igor displays the wave's row indices in a column to the left of the wave data and displays the wave's column indices in the horizontal index row, above the wave data.

If you append the wave's dimension label column, Igor displays the wave's row labels in a column to the left of the wave data and displays the wave's column labels in the horizontal index row, above the wave data.

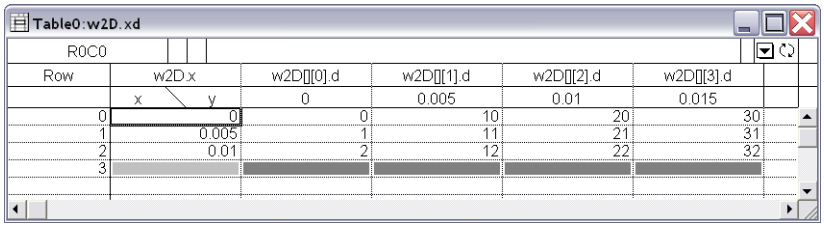
If you display neither index columns nor dimension labels, Igor still displays the wave's column indices in the horizontal index row.

If you want to show numeric indices horizontally and dimension labels vertically or vice versa, you can use the Table→Horizontal Index submenu to override the default behavior. For example, if you want dimension labels vertically and numeric indices horizontally, append the wave's dimension label column to the table and then choose Table→Horizontal Index→Numeric Indices.

In the example above, the row and column indices are equal to the row and column element numbers. You can set the row and column scaling of your 2D data to reflect the nature of your data. For example, if the data is an image with 5 mm resolution in both dimensions, you should use the following commands:

```
SetScale/P x 0, .005, "m", w2D; SetScale/P y 0, .005, "m", w2D
```

When you do this, you will notice that the row and column indices in the table reflect the scaling that you have specified.



Row	w2D.x	w2D[][0].d	w2D[][1].d	w2D[][2].d	w2D[][3].d
	x \ y	0	0.005	0.01	0.015
0	0	0	10	20	30
1	0.005	1	11	21	31
2	0.01	2	12	22	32
3					

1D waves have no column indices. Therefore, if you have no multidimensional waves in the table, Igor does not display the horizontal index row. If you have a mix of 1D and multidimensional waves in the table, Igor does display the horizontal index row but displays nothing in that row for the 1D waves.

When showing 1D data, Igor displays a column of point numbers at the left side of the table. This is called the Point column. When showing a 2D wave, Igor will title the column Row or Column depending on how you are viewing the 2D wave. If you have 3D or 4D waves in the table, Igor titles the column Row, Column, Layer or Chunk, depending on how you are viewing the waves.

It is possible to display a mix of 1D waves and multidimensional waves such that none of these titles is appropriate. For example, you could display two 2D waves, one with the wave rows shown vertically (the normal case) and one with the wave rows shown horizontally. In this case, Igor will title the Point column "Element".

You can edit dimension labels in the main body of the table by merely clicking in the cell and typing. However, you can't edit dimension labels in the horizontal index row this way. Instead you must double-click a label in this row. Igor will then display a dialog into which you can enter a dimension label. You can also set dimension labels using the SetDimLabel operation from the command line.

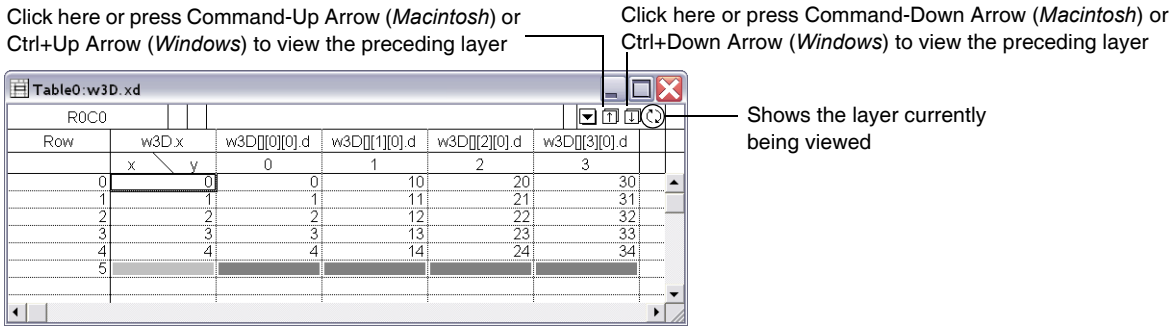
Changing the View of the Data

A table can display waves of dimension 1 through 4. A one dimensional wave appears as a simple column of numbers, or, if the wave is complex, as two columns, one real and one imaginary. A two dimensional wave appears as a matrix. In the one and two dimensional cases, you can see all of the wave data at once.

If you display a three dimensional wave in a table, you can view and edit only one slice at a time. To see this, execute the following commands:

```
Make/O/N=(5,4,3) w3D = p + 10*q + 100*r; Edit w3D.id
```

Initially you will see the slice of the wave whose layer index is 0 — layer zero of the wave. You can change which layer you are viewing using icons that appear at the top of the table or using keyboard shortcuts.



When clicking the icons, pressing Shift reverses the direction.

If you display a four dimensional wave in a table, you can still view and edit only one layer at a time. You can change which layer you are viewing by using the icons or keyboard shortcuts described above. To view the next chunk in the 4D wave, press Option (*Macintosh*) or Alt (*Windows*) while clicking the down icon or press Command-Option-Down Arrow (*Macintosh*) or Ctrl+Alt+Down Arrow (*Windows*). To view the previous chunk, press Option or Alt while clicking the up icon or press Command-Option-Up Arrow or Ctrl+Alt+Up Arrow.

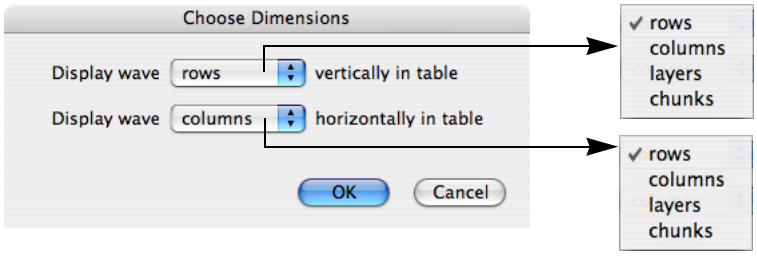
In addition to using the keyboard shortcuts, you can specify the layer and chunk that you want to use using the ModifyTable elements operation.

Changing the Viewed Dimensions

When you initially view a 3D wave in a table, Igor shows a slice of the wave in the rows-columns plane. The wave's rows dimension is mapped to the table's vertical dimension and the wave's columns dimension is mapped to the table's horizontal dimension.

Using the Choose Dimensions icon, you can instruct Igor to map any wave dimension to the vertical table dimension and any other wave dimension to the horizontal table dimension. This is primarily of interest if you work with waves of dimension three or higher because you can view and edit any orthogonal plane in the wave. You can, for example, create a new wave that contains a slice of data from the 3D wave.

When you click the Choose Dimensions icon, Igor displays the Choose Dimensions dialog. This dialog specifies how the dimensions in the wave are to be displayed in the table.



ModifyTable Elements Command

This section discusses choosing viewed dimensions using the ModifyTable "elements" keyword. You do not need to know about this unless you want a thorough understand of editing 3D and 4D waves and viewing them from different perspectives.

The best way to understand this section is to execute all of the commands shown.

Igor needs to know which wave dimension to map to the vertical table dimension and which wave dimension to map to the horizontal table dimension. In addition, for waves of dimension three or higher, Igor needs to know which element of the remaining dimensions to display.

The form of the command is:

```
ModifyTable elements(<wave name>) = (<row>,<column>,<layer>,<chunk>)
```

The parameters specify which element of the wave's rows, columns, layers and chunks dimensions you want to view. The value of each parameter may be an element number (0 or greater) or it may be a special value. There are three special values that you can use for any of the parameters:

-1 Means no change from the current value.

-2 Means map this dimension to the table's vertical dimension.

-3 Means map this dimension to the table's horizontal dimension.

In reading the following discussion, remember that the first parameter specifies how you want to view the wave's rows, the second parameter specifies how you want to view the wave's columns, the third parameter specifies how you want to view the wave's layers and the fourth parameter specifies how you want to view the wave's chunks.

If you omit a parameter, it takes the default value of -1 (no change). Thus, if you are dealing with a 2D wave, you can supply only the first two parameters and omit the last two.

To get a feel for this command, let's start with the example of a simple matrix, which is a 2D wave.

```
Make/O/N=(3,3) wave0 = p + 10*q
Edit wave0.id
```

As you look down in the table, you see the rows of the matrix and as you look across the table, you see its columns. Thus, initially, the rows dimension is mapped to the vertical table dimension in the and the columns dimension is mapped to the horizontal table dimension. This is the default mapping. You can change this with the following command:

```
ModifyTable elements(wave0) = (-3, -2)
```

The first parameter specifies how you want to view the wave's rows and the second parameter specifies how you want to view the wave's columns. Since the wave has only two dimensions, the third and fourth parameters can be omitted.

The -3 in this example maps the wave's rows to the table's horizontal dimension. The -2 maps the wave's columns to the table's vertical dimension.

You can return the wave to its default view using:

```
ModifyTable elements(wave0) = (-2, -3)
```

When you consider a 3D wave, things get a bit more complex. In addition to the rows and columns dimensions, there is a third dimension — the layers dimension. When you initially create a table containing a 3D wave, it shows all of the rows and columns of layer 0 of the wave. Thus, as with the 2D wave, the rows dimension is mapped to the vertical table dimension and the columns dimension is mapped to the horizontal table dimension. You can control which layer of the 3D wave is displayed in the table using the icons and keyboard shortcuts described above, or using the ModifyTable elements keyword.

For example:

```
Make/O/N=(5,4,3) wave0 = p + 10*q + 100*r
ModifyTable elements(wave0)=(-2, -3, 1)   //Shows layer 1 of 3D wave
ModifyTable elements(wave0)=(-2, -3, 2)   //Shows layer 2 of 3D wave
```

In these examples, the wave's layers dimension is fixed to a specific value whereas the wave's rows and columns dimensions change as you look down or across the table. The term "free dimension" refers to a wave dimension that is mapped to either of the table's dimensions. The term "fixed dimension" refers to a wave dimension for which you have chosen a fixed value.

In the preceding example, we viewed a slice of the 3D wave in the rows-columns plane. We can view any orthogonal plane. For example, this command shows us the data in the layers-rows plane:

```
ModifyTable elements=(-3, 0, -2)    // Shows column 0 of 3D wave
```

The first parameter says that we want to map the wave's rows dimension to the table's horizontal dimension. The second parameter says that we want to see column 0 of the wave. The third parameter says that we want to map the wave's layers dimension to the table's vertical dimension.

Dealing with a 4D wave is similar to the 3D case, except that, in addition to the two free dimensions, you have two fixed dimension.

```
Make/O/N=(5,4,3,2) wave0 = p + 10*q + 100*r + 1000*s
ModifyTable elements(wave0)=(-2, -3, 1, 0)   //Shows layer 1/chunk 0
ModifyTable elements(wave0)=(-2, -3, 2, 1)   //Shows layer 2/chunk 1
```

If you change a wave (using Make/O or Redimension) such that one or both of the free dimensions has zero points, Igor automatically resets the view to the default — the wave's rows dimension mapped to the table's vertical dimension and the wave's column dimension mapped to the table's horizontal dimension. Here is an example:

```
Make/O/N=(5,4,3) wave0 = p + 10*q + 100*r
Edit wave0.id
Modify elements(wave0) = (0, -2, -3)// Map layers to horizontal dim
Redimension/N=(5,4) wave0           // Eliminate layers dimension!
```

This last command has eliminated the wave dimension that is mapped to the table horizontal dimension. Thus, Igor will automatically reset the table view.

If you use a dimension with zero points as a free dimension, Igor will also reset the view to the default:

```
Make/O/N=(3,3) wave0 = p + 10*q
Edit wave0.id
Modify elements(wave0) = (0, -2, -3)   // Map layers to horizontal dim
```

This last command maps the wave's layers dimension to the table's horizontal dimension. However, the wave has no layers dimension, so Igor will reset the view to the default.

The initial discussion of changing the view using keyboard shortcuts was incomplete for the sake of simplicity. It said that Option-Down Arrow (*Macintosh*) or Alt+Down Arrow (*Windows*) displayed the next layer and that Command-Option-Down Arrow or Ctrl+Alt+Down Arrow displayed the next chunk. This is true if rows and columns are the free dimensions. A more general statement is that Option-Down Arrow or Alt+Down Arrow changes the viewed element of the first fixed dimension and Command-Option-Down Arrow or Ctrl+Alt+Down Arrow changes the viewed element of the second fixed dimension. Here is an example using a 4D wave:

```
Make/O/N=(5,4,3,2) wave0 = p + 10*q + 100*r + 1000*s
Edit wave0.id
ModifyTable elements(wave0)=(0, -2, 0, -3)
```

The ModifyTable command specifies that the columns and chunks dimensions are the free dimensions. The rows and layers dimensions are fixed at 0. If you now press Option-Down Arrow or Alt+Down Arrow, you change the element of the first fixed dimension — the rows dimension in this case. If you press Command-

Down Arrow or Ctrl+Alt+Down Arrow, you change the element of the second fixed dimension — the layers dimension in this case.

Multidimensional Copy/Cut/Paste/Clear

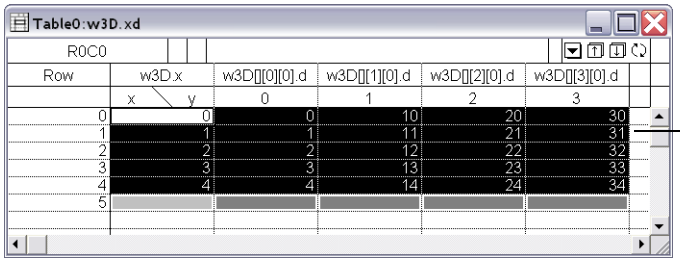
The material in this section is primarily of interest if you intend to edit 3D and 4D data in a table. There are also a few items that deal with 2D data.

For 1D and 2D waves, the subset that you copy and paste is obvious from the table selection. In the case of 3 and higher dimension waves, you can only see two dimensions in the table at one time and you can select a subset of those two dimensions. Normally a copy copies just what you see to the Clipboard. However, if you press Option (*Macintosh*) or Alt (*Windows*) while choosing Copy from the Edit menu or if you press Command-Option-C (*Macintosh*) or the Ctrl+Alt+C (*Windows*), Igor copies data from all dimensions.

Consider the following example. We make a wave with 5 rows, 4 columns and 3 layers and display it in a table:

```
Make/O/N=(5,4,3) w3D = p + 10*q + 100*r; Edit w3D.id
```

The table now displays all rows and all columns of layer 0 of the wave. If you select all of the visible data cells and do a copy, you will copy all of layer 0 to the Clipboard. However, if you do an Option-copy or Alt-copy, you will copy all three layers to the Clipboard — layer 0 plus the two layers that are currently not visible.



The Option or Alt key changes the operation from acting on just the visible selected cells to acting on all selected cells. The Shift key affects the paste operation only and does an insert-paste instead of a replace-paste.

This table shows the effect of the Option (*Macintosh*) or Alt (*Windows*) key.

	Copy	Paste	Clear
No modifiers	Copies visible	Replace-pastes visible	Clears visible
Option or Alt	Copies all	Replace-pastes all	Clears all

The middle column of the preceding table mentions "replace-pasting". When you do a paste, Igor normally replaces the selection in the table with the data in the Clipboard. However, if you press Shift while pasting, Igor inserts the data as new cells in the table. This is called an "insert-paste". This table shows the effect of the Shift and Option (*Macintosh*) or Alt (*Windows*) keys on a paste.

	Paste
No modifiers	Replace-pastes visible
Option	Replace-pastes all
Shift	Insert-pastes visible
Shift and Option or Alt	Insert-pastes all

Replace-Paste of Multidimensional Data

When you copy data in a table and then select another section of the table or a section of another table and do a paste, you are doing a replace-paste. The copied data replaces the selection when you do the paste.

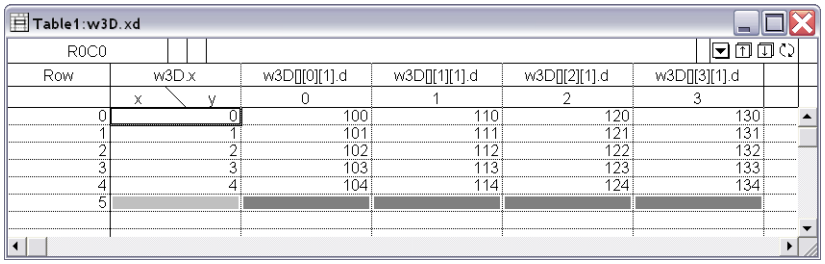
For 1D and 2D waves, the subset that you copy and paste is obvious from the table selection. In the case of 3 and higher dimension waves, you can only see two dimensions in the table at one time and you can select a subset of those two dimensions.

When you do a replace-paste involving waves of dimension 3 or higher, the data in the Clipboard replaces the data in the currently visible slice of the selected wave. Here is an example that illustrates this.

We make a wave with 5 rows, 4 columns and 3 layers and display it in a table:

```
Make/O/N=(5,4,3) w3D = p + 10*q + 100*r; Edit w3D.id
```

The table now displays all rows and all columns of layer 0 of the wave. Let's look at layer 1 of the wave. To do this, press Option-Down Arrow (*Macintosh*) or Alt+Down Arrow (*Windows*) while the table is active. This changes the view to show layer 1 instead of layer 0.



Now, select all of the visible cells and do a copy (Command-C on *Macintosh* or Ctrl+C on *Windows*). This copies all of layer 1 to the Clipboard. Now, press Option-Down Arrow or Alt+Down Arrow again to view layer 2 of the wave. With all of the cells still selected, do a paste (Command-V on *Macintosh* or Ctrl+V on *Windows*). The data copied from layer 1 replaces the data in layer 2 of the wave. Do an undo. Layer 2 is restored to its original state. Now press Option-Up Arrow or Alt+Up Arrow two times. We are now looking at layer 0. Do a paste. The data that we copied from layer 1 replaces the data in layer 0. Do an undo to return layer 0 to the original state.

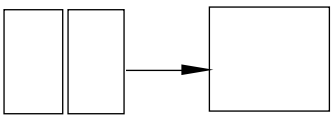
Now let's consider an example in which we copy and paste all of the wave data, not just one layer. To do the copy, press Command-Option-C (*Macintosh*) or Ctrl+Alt+C (*Windows*) or press Option (*Macintosh*) or Alt (*Windows*) and choose Copy from the Edit menu. We have now copied the entire wave, all three layers, to the Clipboard. Next, press Option or Alt and choose Clear from the Edit menu. Use Option-Down Arrow or Alt+Down Arrow and Option-Up Arrow or Alt+Up Arrow to verify that all three layers were cleared. Now do the paste by pressing Command-Option-V (*Macintosh*) or Ctrl+Alt+V (*Windows*) or pressing Option or Alt while choosing Paste from the Edit menu. This pastes all three layers from the Clipboard into the selected wave. Use Option-Down Arrow and Option-Up Arrow or Alt+Down Arrow and Alt+Up Arrow to verify that all three layers were pasted.

In this example, we make a 2D wave from two 1D waves. Execute:

```
Make/O/N=5 w1DA=p, w1DB=100+p; Edit w1DA, w1DB
```

Click the column name of w1DA to select all of it and do a copy. Select the first unused cell to the right of w1DB and do a paste. We now have a clone of w1DA named w1DA1. Choose Redimension from the Table pop-up menu. This brings up the Redimension Waves dialog with w1DA1 preselected. Find the New Columns text box which currently has the number 0 in it. Enter 2 in place of the 0 and click Do It. We have now made w1DA1 a 2D wave. Click the column name of the w1DB column to select all of it and do a copy. Now click the column name of the second column of w1DA1 and do a paste. We now have a 2D wave generated from two 1D waves.

Making a 2D Wave from Two 1D Waves



1. Select all of the first 1D wave and choose Copy.
2. Click in the first unused cell in the table and choose Paste.
3. Choose Redimension from the Table pop-up menu.
4. In the Redimension Waves dialog, enter 2 for the number of columns and click Do It.
5. Select all of the second 1D wave and choose Copy.

6. Select all of the second column of the 2D wave and choose Paste.

Insert-Paste of Multidimensional Data

When you copy data in a table and then select another section of the table or a section of another table and do a paste while pressing Shift, you are doing an insert-paste. The copied data is inserted into the selected wave or waves before the selected data. As in the case of the replace-paste, the insert-paste works on just the visible layer of data if Option (*Macintosh*) or Alt (*Windows*) is not pressed or on all layers if the Option or Alt is pressed. Here is an example that illustrates this.

We make a wave with 5 rows, 4 columns and 3 layers and display it in a table:

```
Make/O/N=(5,4,3) w3D = p + 10*q + 100*r; Edit w3D.id
```

Select all of the cells in rows 1 and 2 of the table. An easy way to do this is to click the "1" in the Row column and drag down to the "2". Now copy the selected cells by pressing Command-C (*Macintosh*) or Ctrl+C (*Windows*). Since you did not press Option or Alt, this copies just the visible layer. Next, press shift and choose Paste from the Edit menu to insert-paste the copied data. Notice that two new rows were inserted. Press Command-Down Arrow or Ctrl+Down Arrow to see what was inserted in layer 1 of the 3D wave. Notice that zeros were inserted. This is because the paste stored data only in the visible layer.

Press Option-Up Arrow or Alt+Up Arrow to view layer 0 again. Now choose Undo from the Edit menu to undo the paste. Use Option-Down Arrow or Alt+Down Arrow to check the other layers of the wave and then use Option-Up Arrow or Alt+Up Arrow to come back to layer zero.

Now we will do an insert-paste in all layers. Again, select rows 1 and 2 of the wave. Press Option or Alt and choose Copy from the Edit menu. This copies data from all three layers to the Clipboard. Now press Shift-Option or Shift+Alt and choose Paste from the Edit menu. This pastes data from the Clipboard into all three layers of the wave. By pressing Shift, we did an insert-paste rather than a replace-paste and by pressing Option or Alt, we pasted into all layers, not just the visible layer. Use Option-Down Arrow or Alt+Down Arrow to verify that we have pasted data into all layers.

Cutting and Pasting Rows Versus Columns

Normally, you cut and paste rows of data. However, there may be cases where you want to cut and paste columns. For example, if you have a 2D wave with 5 rows and 3 columns, you may want to cut the middle column from the wave. Here is how Igor determines if you want to cut rows or columns.

If the selected wave is 2D or higher, *and* if one or more *entire* columns is selected, Igor cuts the selected column or columns. In all other cases Igor cuts rows.

After copying or cutting wave data, you have data in the Clipboard. Normally, a paste overwrites the selected rows or inserts new rows (if you press Shift).

To insert columns, you need to do the following:

1. Select exactly one entire column. You can do this by clicking in the column name.
2. Press Shift and choose Paste from the Edit menu or press Command-Shift-V (*Macintosh*) or Ctrl+Shift+V (*Windows*).

If the wave data is real (not complex), Igor normally pastes the new column or columns before the selected column. This behavior would provide you with no way to paste columns after the last column of a wave. Therefore, if the selected column is the last column, Igor presents a dialog to ask you if you want to paste the new columns before or after the last column.

If the wave data is complex, Igor pastes the new columns before the selected column if the selected column is real or after the selected column if it is imaginary.

If you select more than one column or if you do not select all of the column, Igor will insert rows instead of columns.

Create-Paste of Multidimensional Data

When you copy data in a table and then select the first unused cell in the table and then do a paste, Igor creates one or more new waves.

The number of waves created and the number of dimensions in each wave are the same as for the copied data. Igor also copies and pastes the following wave properties:

- Data units and dimension units
- Data full scale and dimension scaling
- Dimension labels
- The wave note

Igor copies and pastes the wave note only if you copy the entire wave. If you copy part of the wave, it does not copy the wave note.

You can use a create-paste to create a copy of a subset of the data displayed in the table. For 1D and 2D waves, the subset that you copy and paste is obvious from the table selection. In the case of 3 and higher dimension waves, you can only see two dimensions in the table at one time and you can choose a subset of those two dimensions. If you do a copy and then a create-paste, Igor creates a new 2D wave containing just the data that was visible when you did the copy. If you do an Option-copy (*Macintosh*) or Alt -copy (*Windows*) to copy all and then a create-paste, Igor creates a new wave with the same number of dimensions and same data as what you copied.

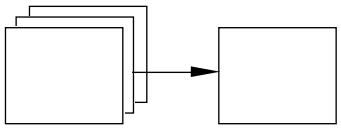
Here is an example.

We make a wave with 5 rows, 4 columns and 3 layers and display it in a table:

```
Make/O/N=(5,4,3) w3D = p + 10*q + 100*r; Edit w3D.id
```

Select all of the visible cells and do a copy. Click in the first unused column, to the right of the last column of w3D. Now do a paste. Igor creates a 2D wave consisting of the visible data that you copied.

Making a 2D Wave from a Slice of a 3D Wave

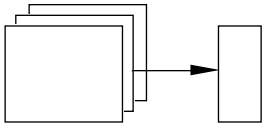


1. Select the slice of the 3D wave and choose Copy.
2. Click in the first unused cell and choose Paste.

Get rid of the new 2D wave by doing an undo. Select all of the w3D cells again and do an Option-copy or Alt-copy (press Command-Option-C on *Macintosh* or Ctrl+Alt+C on *Windows*). Select the unused cell again and do a paste. Igor creates a 3D wave consisting of the visible data that you copied *and* the data in other layers of w3D, not currently showing in the table. To demonstrate this, view the other layers of the new wave by pressing Option-Down Arrow or Alt+Down Arrow (to go to the next layer) and Option-Up Arrow or Alt+Up Arrow (to go to the preceding layer). Get rid of the new 3D wave by doing an undo.

Here is how to extract a 1D wave from a 2D or 3D wave. Select all of the first column of w3D by clicking in the column name. Now select the unused cell again and do a paste. Note the column name for the new wave: "w3D1[][0]". This indicates that the new wave is a 2D wave and that the table is showing all rows of column zero, the only column in the wave. But we want to make a 1D wave, not a 2D wave. Click in any cell in the new wave and choose Redimension w3D1 from the Table pop-up menu. This brings up the Redimension Waves dialog with w3D1 preselected. Find the New Columns text box which currently has the number 1 in it. Enter 0 in place of the 1 and click Do It. We have now made w3D1 a 1D wave. Notice that the column name is now "w3D1".

Making a 1D Wave from a Column of a Multidimensional Wave



1. Select the column of the multidimensional wave and choose Copy.
2. Click in the first unused cell in the table and choose Paste.
3. Choose Redimension from the Table pop-up menu.
4. In the Redimension Waves dialog, enter 0 for the number of columns and click Do It.

Printing Tables

Before printing a table you should bring the table to the top of the desktop and set the page size and orientation using the Page Setup dialog. Choose the Page Setup for All Tables item from the Files menu.

In each experiment, Igor stores one page setup for all tables. Thus, changing the page setup while a table is active changes the page setup for all tables in the current experiment.

When you invoke the Page Setup dialog you must make sure that the table that you want to print is the top window. Changing the page setup for graphs, page layouts or other windows does not affect the page setup for tables.

You can print all or part of a table. To do this choose the Print Table or Print Table Selection item from the File menu while the table is the active window.

On *Macintosh*, if the selection range consists of just the target cell, the entire table will be printed. If the selection range is more than just the target cell, only the selected cells will be printed. On *Windows*, use the Print Range controls in the print dialog to print all of the table or just the selection.

Save Table Copy

You can save the active table as an Igor packed experiment file or as a tab or comma-delimited text file by choosing File→Save Table Copy.

The main uses for saving as a packed experiment are to save an archival copy of data or to prepare to merge data from multiple experiments (see **Merging Experiments** on page II-32). The resulting experiment file preserves the data folder hierarchy of the waves displayed in the table starting from the "top" data folder, which is the data folder that encloses all waves displayed in the table. The top data folder becomes the root data folder of the resulting experiment file. Only the table and its waves are saved in the packed experiment file, not variables or strings or any other objects in the experiment.

Save Table Copy does not know about dependencies. If a table contains a wave, wave0, that is dependent on another wave, wave1 which is not in the table, Save Table Copy will save wave0 but not wave1. When the saved experiment is open, there will be a broken dependency.

The main use for saving as a tab or comma-delimited text file is for exporting data to another program. When saving as text, the data format matches the format shown in the table. Keep in mind that this will cause truncation if the underlying data has more precision than shown in the table. The point column is never saved.

To save data as text with full precision, choose Data→Save Waves→Save Delimited Text.

When saving 3D and 4D waves as text, only the visible layer is saved. To save the entirety of a 3D or 4D wave, choose Data→Save Waves→Save Delimited Text.

The **SaveTableCopy** operation on page V-546 provides options that are not available using the Save Table Copy menu command.

Exporting Tables as Graphics

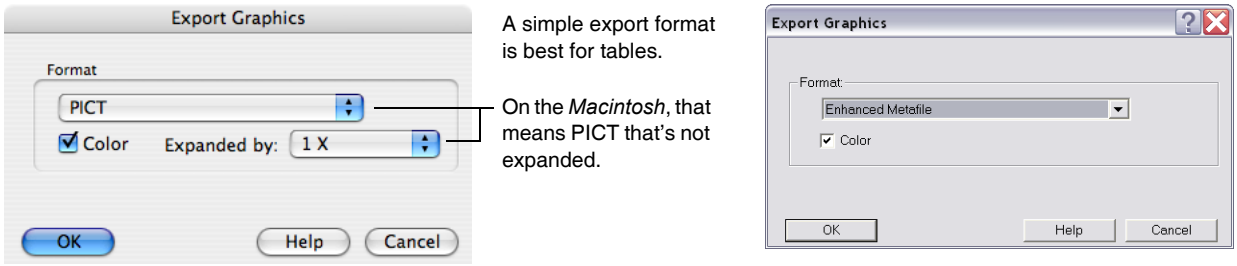
Although Igor tables are intended primarily for editing data, you can also use them for presentation purposes. You can put a table into an Igor page layout, as discussed in Chapter II-16, **Page Layouts**. This section deals with exporting a table to other applications as a picture or as an Encapsulated PostScript (EPS) file.

Typically you would do this if you are preparing a report in a word processor or page layout program or making an illustration in a drawing program. If you are exporting to a program that has strong text formatting features, it may be better to copy the data from the table as text, using the Copy item in the Edit menu. You can paste the text into the other program and then format it as you wish.

Exporting a Table as a Picture

To export a table as a Macintosh picture via the Clipboard, choose Export Graphics from the Edit menu. This copies the table to the Clipboard as a picture.

The Export Graphics dialog puts a picture of the table into the Clipboard.



The picture that Igor puts into the Clipboard contains just the visible cells in the table window. You can scroll, expand or shrink the window to control which cells will appear in the picture.

This dialog presents a lot of options that are more useful when exporting graphs or page layouts than when exporting a table. For exporting a table, you should use the simplest option. On the Macintosh this is the standard PICT with an expansion factor of 1, as shown above. Under Windows, use the Enhanced Metafile. The meaning and use of the other options are explained in Chapter III-5, **Exporting Graphics (Macintosh)**.

Igor can write a picture out to a file instead of copying it to the Clipboard. To do this, use the Save Graphics submenu of the File menu.

Most word processing, drawing and page layout programs permit you to import a picture via the Clipboard or via a file.

If you plan to print your report or illustration on a PostScript printer, you should make sure to use PostScript or TrueType fonts in the table. If you use a screen font then the text in the printed result may be somewhat distorted. Helvetica, Courier, Times and Palatino are common PostScript fonts.

Although we have not optimized tables for presentation purposes, we did put two features into tables specifically for presentation. First, you can hide the point column by setting its width to zero. Second, you can replace the automatic column titles with column titles of your own. Use the Modify Columns dialog for both of these.

There are some features lacking that would be nice for presentation. You can't change the background color of a table. You can't change or remove the grid lines. If you want to do these things, export the table to a drawing program for sprucing up.

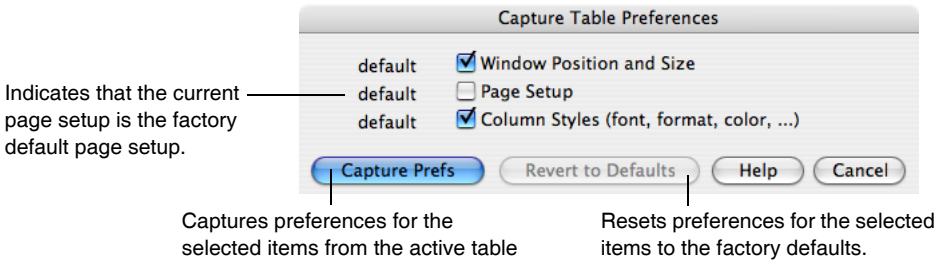
Exporting a Table as an EPS file

Many word processing, drawing and page layout programs permit you to import a picture via an Encapsulated PostScript (EPS) file. To export a table as EPS, choose EPS File from the pop-up menu in the Save Graphics File dialog.

When exporting a graph with smooth curves, arrows, dashed lines and other fancy graphic items, EPS sometimes produces better results than simpler methods. This is not the case with tables. So, the only reason to export a table as an EPS file is if the program to which you are exporting can't handle a simpler method or handles EPS better. This is true of some PostScript drawing programs.

Table Preferences

Table preferences allow you to control what happens when you create a new table or add new columns to an existing table. To set preferences, create a table and set it up to your taste. We call this your *prototype* table. Then choose Capture Table Prefs from the Table menu.



Preferences are normally in effect only for *manual* operations, not for automatic operations from Igor procedures. This is discussed in more detail in **When Preferences Are Applied** on page III-432.

When you initially install Igor, all preferences are set to the factory defaults. The dialog indicates which preferences you have changed, and which are factory defaults.

The Window Position and Size preference affects the creation of new tables only.

The Column Styles preference affects the style of newly created tables and of columns added to an existing table. This preference can store column settings for one or more columns, in addition to the point column, depending on the number of columns in your prototype table.

The simplest and recommended way to use this is to have just one column (in addition to the point column) in your prototype table. Igor will store two preferred column styles, one for the point column and one for all other columns. Be sure to select the Repeat Column Style Prefs in Tables checkbox in the Miscellaneous Settings dialog.

If you have more than one data column in your prototype table then Igor will store additional preferred column styles. When the number of columns in a table exceeds the number of captured column styles, Igor assigns a style one of two ways:

- the factory default style is used,
- or the styles repeat with the first column (not the Point column) style, and then the second style, etc.

You make this choice in the Miscellaneous Settings dialog, with the Repeat Column Style Prefs in Tables checkbox.

For example, assume that your prototype column has, in addition to the point column, a column with red text and a column with blue text. With Repeat Column Style Prefs in Tables selected, any columns in a new table and any columns you add to an existing table will alternate between red and blue. If it is not selected, the first column will be red, the second column will be blue, and all successive added columns will be black (the factory default).

The page setup preference affects what happens when you create a new *experiment*, not when you create a new *table*. Here is why.

Each experiment stores *one* page setup for *all* tables in that experiment. The preferences also store one page setup for tables. When you set the preferred page setup for tables, Igor stores a copy of the current experiment's page setup for tables in the preferences file. When you create a new experiment, Igor stores a copy of the preferred page setup for tables in the experiment.

Table Style Macros

The purpose of a table style macro is to allow you to create a number of tables with the same stylistic properties. Using the Window Control dialog, you can instruct Igor to automatically generate a style macro from a prototype table. You can then apply the macro to other tables.

Igor can generate style macros for graphs, tables and page layouts. However, their usefulness is mainly for graphs. See **Graph Style Macros** on page II-300. The principles explained there apply to table style macros also.

Table Shortcuts

To view table keyboard navigation shortcuts, see **Keyboard Navigation in Tables** on page II-193.

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To add columns for existing waves	Choose Append to Table from the Table menu.	Choose Append to Table from the Table menu.
To create new numeric waves	Click in the first unused column and enter a number or paste numeric data from the Clipboard.	Click in the first unused column and enter a number or paste numeric data from the Clipboard.
	Copy all or part of an existing numeric wave and paste in the first unused column.	Copy all or part of an existing numeric wave and paste in the first unused column.
To create new text waves	Click in the first unused column and enter nonnumeric text or paste text data from the Clipboard.	Click in the first unused column and enter nonnumeric text or paste text data from the Clipboard.
	Copy all or part of an existing text wave and paste in the first unused column.	Copy all or part of an existing text wave and paste in the first unused column.
To kill one or more waves	Select the waves in the table and choose Kill Waves from the Table pop-up menu.	Select the waves in the table and choose Kill Waves from the Table pop-up menu.
To rename a wave	Select one or more cells from the wave and choose Rename from the Table pop-up menu.	Select one or more cells from the wave and choose Rename from the Table pop-up menu.
To redimension a wave	Select one or more cells from the wave and choose Redimension from the Table pop-up menu.	Select one or more cells from the wave and choose Redimension from the Table pop-up menu.
To add cells to a column	Click in a row and choose Insert Points from the Table pop-up.	Click in a row and choose Insert Points from the Table pop-up.
	Press Command-Shift-V. This does an insert-paste.	Press Ctrl+Shift+V. This does an insert-paste.
	Click in the insertion cell at the end of the column and enter a value or paste.	Click in the insertion cell at the end of the column and enter a value or paste.
To remove cells from a column	Select the cells to be removed and choose Delete Points from the Table pop-up.	Select the cells to be removed and choose Delete Points from the Table pop-up.
	Press Command-X (Cut).	Press Ctrl+X (Cut).
To identify a particular column	Press Command-Option-Control and click in the column.	Press Shift-F1 to summon context-sensitive help and then click in the column.
To modify column styles	Double-click a column title (goes to Modify Columns dialog).	Double-click a column title (goes to Modify Columns dialog).
	Select one or more columns and click the Table pop-up menu.	Select one or more columns and click the Table pop-up menu.
	Press Control and click the column title.	Right-click the column title.
To modify the style of all columns at once	Press Shift while using the Table pop-up menu.	Press Shift while using the Table pop-up menu.
	Press Shift-Option to modify all but the Point column.	Press Alt+Shift to modify all but the Point column.

Action	Shortcut (*Macintosh*)	Shortcut (*Windows*)
To modify the style of a subset of the data columns of multidimensional wave	Select the columns to modify, press Command, then click the Table menu or the Table pop-up menu, then make a selection.	Select the columns to modify, press Ctrl, then click the Table menu or the Table pop-up menu, then make a selection.
To select entire rows	Drag in the point number column.	Drag in the point number column.
To select entire columns	Drag on the column titles.	Drag on the column titles.
To change a wave's position	Press Option and drag the column title to the new position.	Press Alt and drag the column title to the new position.
To adjust the width of a column	Drag the border at the right of the column title.	Drag the border at the right of the column title.
	Press Shift to change all widths. Press Shift-Option to change all except the point column.	Press Shift to change all widths. Press Alt+Shift to change all except the point column.
	Press Command while clicking to set the width of a single data column of a multidimensional wave.	Press Ctrl while clicking to set the width of a single data column of a multidimensional wave.
To bring the target cell into view	Click in the cell ID in the top left corner of the table.	Click in the cell ID in the top left corner of the table.

Chapter
II-12

Graphs

Overview

Igor graphs are simultaneously:

- Publication quality presentations of data.
- Dynamic windows for exploratory data analysis.

A single graph can contain one or more of the following:

Waveform plots	Wave data versus scaled point number
XY plots	Y wave data versus X wave data
Category plots	Numeric wave data versus text wave data
Image plots	Display of a matrix of data
Contour plots	Contour of a matrix or an XYZ triple
Axes	Any number of axes positioned anywhere
Annotations	Textboxes, legends and dynamic tags
Cursors	To read out XY coordinates
Drawing elements	Arrows, lines, boxes, polygons, pictures …
Controls	Buttons, pop-up menus, readouts …

The various kinds of plots can be overlaid in the same plot area or displayed in separate regions of the graph. Igor also provides extensive control over stylistic factors such as font, color, line thickness, dash pattern, etc.

This chapter describes how to create and modify graphs, how to adjust graph features precisely to your liking and how to use graphs for data exploration. Although some of the techniques are applicable to graphs of 2D data (contours and images), this chapter focuses on graphs of 1D numeric waves. Other Igor Pro plot types are described in Chapter II-13, **Category Plots**, in Chapter II-14, **Contour Plots**, and in Chapter II-15, **Image Plots**. Surface plots, isosurface plots, 3D scatter plots and other complex types of three dimensional plots can be made using the Gizmo extension. To get started with Gizmo, choose Windows→New→3D Plots→3D Help.

The use of drawing tools along with techniques for graphically editing data can be found in Chapter III-3, **Drawing**. User-defined buttons and other controls are described in Chapter III-14, **Controls and Control Panels**. Fancy textboxes and other annotations are covered in Chapter III-2, **Annotations**, whereas the in's and out's of exporting publication-quality graphics are located in Chapter III-5, **Exporting Graphics (Macintosh)**, or Chapter III-6, **Exporting Graphics (Windows)**.

Graph Features

Igor graphs are smart. If you expand a graph to fill a large monitor screen, Igor will adjust all aspects of the graph to optimize the presentation for the larger graph size. The font sizes will be scaled to sizes that look good for the large format and the graph margins will be optimized to maximize the data area without fouling up the axis labeling. If you shrink a graph down to a small size, Igor will automatically adjust axis ticking to prevent tick mark labels from running into one another. If Igor's automatic adjustment of parameters does not give the desired effect, you can override the default behavior by providing explicit parameters.

Igor graphs are dynamic. When you zoom in on a detail in your data, or when your data changes, perhaps due to data transformation operations, Igor will automatically adjust both the tick mark labels and the axis labels. For example, before zooming in, an axis might be labeled in milli-Hertz and later in micro-Hertz. No matter what the axis range you select, Igor always maintains intelligent tick mark and axis labels.

If you change the values in a wave, any and all graphs containing that wave will automatically change to reflect the new values.

You can zoom in on a region of interest, expand or shrink horizontally or vertically and you can pan through your data with a hand tool. You can offset graph traces by simply dragging them around on the screen. You can attach cursors to your traces and view data readouts as you glide the cursors through your data. You can edit your data graphically.

Igor graphs are fast. They are updated almost instantly when you make a change to your data or to the graph. In fact, Igor graphs can be made to update in a nearly continuous fashion to provide a real-time oscilloscope-like display during data acquisition.

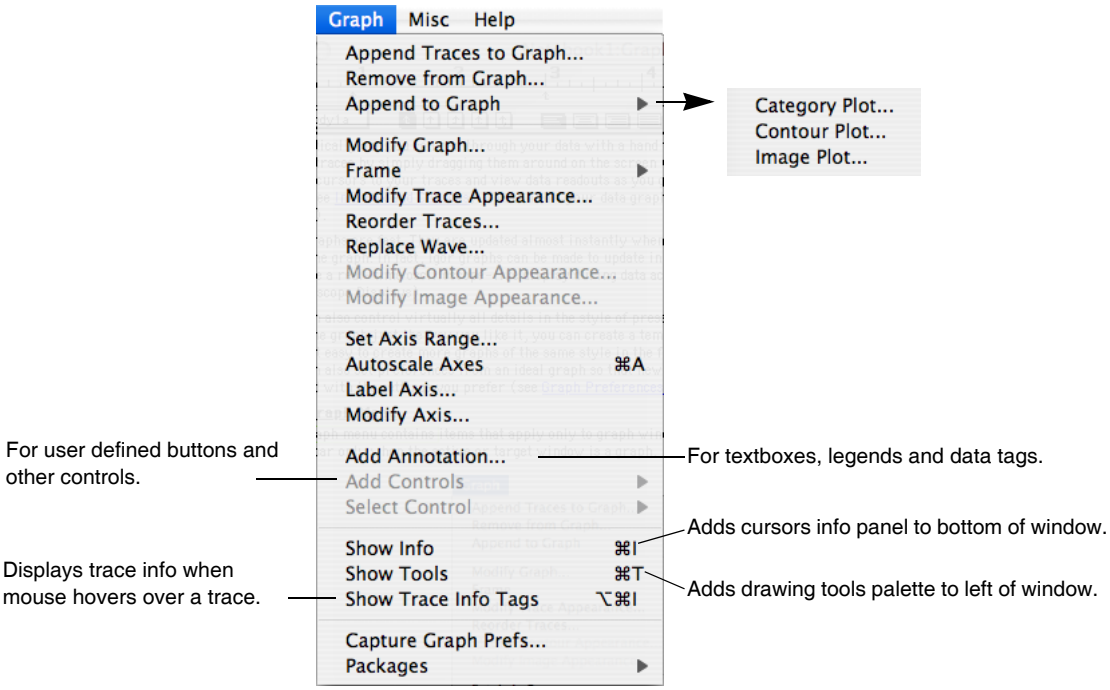
You can also control virtually every detail of a graph. When you have the graph just the way you like it, you can create a template called a "style macro" to make it easy to create more graphs of the same style in the future. You can also set preferences from a reference graph so that new graphs will automatically be created with the settings you prefer.

You can print or export graphs directly, or you can combine several graphs in a page layout window prior to printing or export. You can export graphs and page layouts in a wide variety of formats suitable for editing in a drawing program or for inclusion in a word processor or page layout program with publication-quality results. See Chapter III-5, **Exporting Graphics (Macintosh)**, or Chapter III-6, **Exporting Graphics (Windows)**, and **Printing Graphs** on page II-289 for information about maximizing the quality of printed graphs.

You can have as many graph windows open as memory allows. The amount of memory required for a graph is proportional to the size of the graph and is also related to the number of colors displayed on your monitor. In this age of cheap memory it is rare that the memory required for graphs will be a limiting factor.

The Graph Menu

The Graph menu contains items that apply only to graph windows. The menu appears in the menu bar only when the active or target window is a graph.



When you choose an item from the Graph menu it affects the top-most graph; that graph will necessarily be the "target window".

The Target Window

Operations that apply *only to graphs* (such as ModifyGraph and ShowInfo) affect the top-most graph window. Operations that apply to *any window* (such as Modify) affect the **target window**.

Typing in Graphs (*Macintosh*)

If you type on the keyboard while a graph is the top window, Igor brings the command window to the front and your typing goes into the command line. (The only exception to this is when a graph contains a selected SetVariable control.)

Graph Names

Every graph that you create has a graph name which you can use to manipulate the graph from the command line or from a macro. When you create a new graph, Igor assigns it a name of the form "Graph0", "Graph1" and so on. When you close a graph, Igor offers to create a window recreation macro which you can invoke later to recreate the graph.

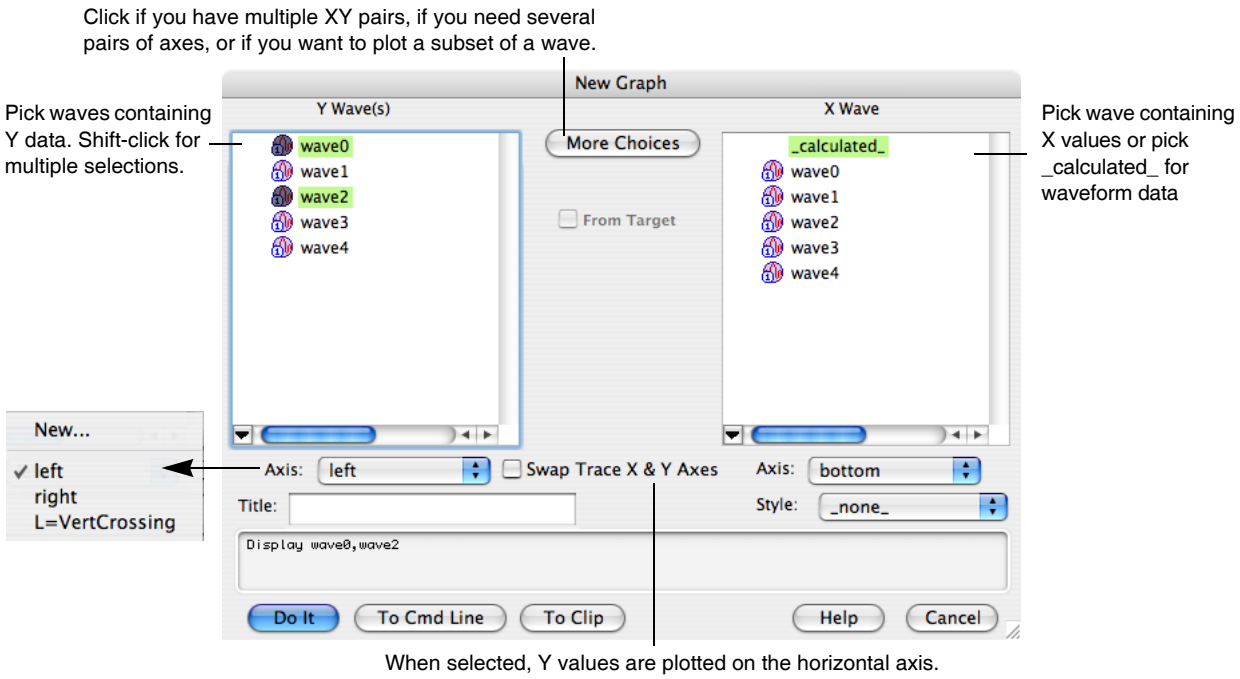
The name of the window recreation macro is the same as the name of the graph. The graph *name* is not the same as the graph *title* which is the text that appears in the graph's window frame. You can change a graph's name and title with the Window Control dialog, accessed by pressing Command-Y (*Macintosh*) or Ctrl+Y (*Windows*).

Creating Graphs

You create a graph by choosing New Graph from the Windows menu.

You can also create a graph by choosing New Category Plot, New Contour Plot or New Image Plot from the New submenu in the Windows menu.

Here is the simple version of the New Graph dialog:



You select the wave(s) to be displayed in the graph from the Y Wave(s) list. The wave is displayed as a **trace** in the graph; by default the trace is a segmented line joining the Y values of the wave. Sometimes we refer to traces as waves, though a trace is actually just a *representation* of the wave.

In fact, you can display a wave more than once in a graph by using the More Choices button or the Append Traces to Graph dialog. Igor distinguishes among the traces by a **trace instance name** such as "myWave#1", which is a kind of trace name that you will see Igor use when modifying graphs containing waves displayed more than once. See **Instance Notation** on page IV-16 and the **ModifyGraph** operation on page V-398.

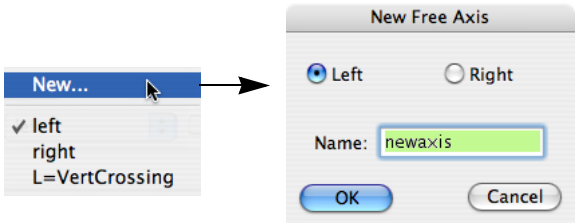
Normally the data values of the waves that you select in the Y Wave(s) list are plotted versus their calculated X values. This is called the **waveform plotting mode**. The calculated X values are derived from the wave's X scaling; see **Waveform Model of Data** on page II-77.

If you want to plot the data values of the Y waves versus the data values of another wave, select the other wave in the X Wave list. This is called the **XY plotting mode**. In this mode, X scaling is ignored; see **XY Model of Data** on page II-78.

If the lengths of the X and Y waves are not equal, then the number of points plotted is determined by the shorter of the waves. You can pick only a single X wave to service all the Y data waves. If you have multiple XY pairs, you will need to use the alternate form of this dialog as described later in this section.

If you want you can specify a **title** for the new window. The title is not used except to form the title bar of the window. It is *not* used to identify windows and does not appear *in* the graph. If you specify no title, Igor will choose an appropriate title based on the traces in the graph and the graph name. Igor automatically assigns graph names of the form "Graph0". The name of a window is important because it is used to identify windows in command line operations. The title is not important to Igor.

If you have created style macros for the current experiment they will appear in the Style pop-up menu. See **Graph Style Macros** on page II-300 for details.



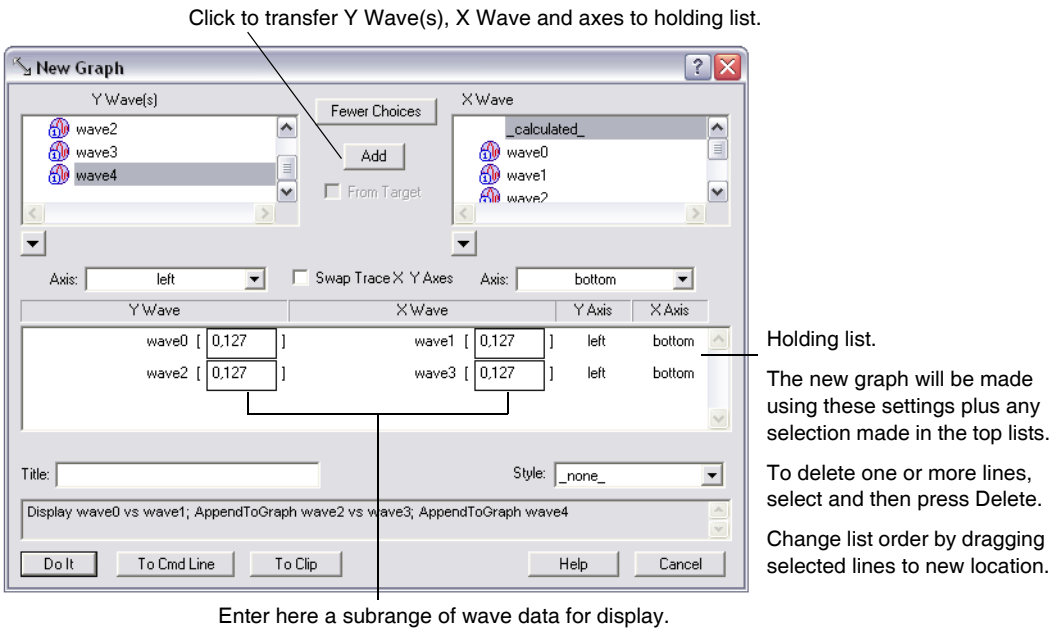
Normally, the new graph will be created using left and bottom axes. You can select other axes using the pop-up menus under the X and Y wave lists. Picking L=VertCrossing will automatically select B=HorizCrossing and vice versa. These **free axes** are used when you want to create a Cartesian type plot where the axes cross at (0,0). You can create additional free axes by choosing New from the pop-up menu. Additional axes can be added to these pop-ups by capturing axes preferences; see **Graph Preferences** on page II-298.

Axes created with the New Free Axis dialog are called "free axes" because they can be freely positioned nearly anywhere in the graph window. The standard left, bottom, right, and top axes are always at the edge of the plot area.

You should give the new axis a name that describes its intended use. The name must be unique within the current graph and can't contain spaces or other nonalphanumeric characters. The Left and Right radio buttons determine the side of the axis on which tick mark labels will be placed. They also define the edge of the graph from which axis positioning is based. See **Modifying Axes** on page II-262 for further details.

You can create a blank graph window containing no traces or axes by clicking the Do It button without selecting any Y waves to plot. The New Contour Plot and New Image Plot dialogs create a blank graph window and then append a contour or image plot to it. Blank graph windows are sometimes useful for programmers and, in conjunction with drawing tools, for the creation of pure drawings, perhaps for inclusion in a page layout.

The New Graph dialog comes in two versions. The simpler version shown above is suitable for most purposes. If, however, you have multiple pairs of XY data or when you will be using more than one pair of axes, you can click the More Choices button to get a more complex version of the dialog.



Using this dialog you can create complex graphs in one step. The graph will be created based on the selections in the Y and X Wave lists as before and also on contents of the holding list. The above example was created by first selecting wave0 from the Y Wave list, wave1 from the X Wave list and then clicking the Add button. Next wave2 was selected from the Y Wave list, wave3 from the X Wave list and the left axis was selected from the Axis pop-up menu under the Y Wave list and then the Add button was clicked. Finally wave4 was chosen from the Y Wave list and _calculated_ was chosen from the X Wave list.

The more complex version of the dialog includes two-dimensional waves in the main list. You can edit the range values for waves in the holding pen to specify individual rows or columns of a matrix or to specify other subsets of data. See **Subrange Display** on page II-288 for details.

Waves and Axes

Axes are dependent upon waves for their existence. If you remove from a graph the last wave that uses a particular axis then that axis will also be removed.

In addition, the first wave plotted against a given axis is called the **controlling wave** for the axis. There is only one thing special about the controlling wave: its units define the units that will be used in creating the axis label and occasionally the tick mark labels. This is normally not a problem since all waves plotted against a given axis will likely have the same units. You can determine which wave controls an axis with the AxisInfo function.

Types of Axes

The four axes named left, right, bottom and top are termed **standard axes**. They are the only axes that many people will ever need.

Each of the four standard axes is always attached to the corresponding edge of the **plot area**. The plot area is the central rectangle in a graph window where traces are plotted. Axis and tick mark labels are plotted outside of this rectangle. See **Graph Dimensions** on page II-246.

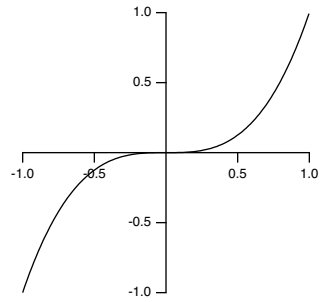
You can also add unlimited numbers of additional user-named axes termed **free axes**. Free axes are so named because you can position them nearly anywhere within the graph window. In particular, vertical free axes can be positioned anywhere in the horizontal dimension while horizontal axes can be positioned anywhere in the vertical dimension. Axes can not be offset laterally but the same effect can be obtained by shrinking the range over which they are drawn. See **Modifying Axes** on page II-262 for details.

The Axis pop-up menu entries L=VertCrossing and B=HorizCrossing you saw in the New Graph dialog above are actually free axes that are each preset to cross at the numerical zero location of the other. They are also set to suppress the tick mark and label at zero. For example, given this data:

```
Make ywave; SetScale/I x,-1,1,ywave; ywave= x^3
```

Then, using the New Graph dialog, we select ywave from the Y list and then L=VertCrossing from the Y axis pop-up menu. This generates the following command and the resulting graph:

```
Display/L=VertCrossing/B=HorizCrossing ywave
```



If desired, you could remove the tick mark and label at -0.5. To do this you would double-click the axis to reach the Modify Axis dialog, choose the Tick Options tab, and finally type -0.5 in one of the unused Inhibit Ticks boxes.

The free axis types described above all require that there be at least one trace that uses the free axis. For special purposes Igor programmers can also create a free axis that does not rely on any traces by using the **NewFreeAxis** operation (page V-435). Such an axis will not use any scaling or units information from any associated waves if they exist. You can specify the properties of a free axis using the **SetAxis** operation (page V-551) or the **ModifyFreeAxis** operation (page V-397), and you can remove them using the **KillFreeAxis** operation (page V-322).

Appending Traces

You can append waves to a graph as a waveform or XY plot by choosing Append Traces to Graph from the Graph menu. This presents a dialog identical to the New Graph dialog except that the title and style macro items are not present. Like the New Graph dialog, this dialog provides a way to create new axes and pairs of XY data. The Append to Graph submenu in the Graph menu provides the means to append category plots, contour plots and image plots.

Igor's curve fitting routines will append traces to your graph automatically if you request it (see **Auto-Trace** on page III-176 for details).

Removing Traces

You can remove traces from a graph by choosing Remove from Graph from the Graph menu. Don't overlook the pop-up above the list that selects among traces, image plots and contour plots; set it to "Traces".

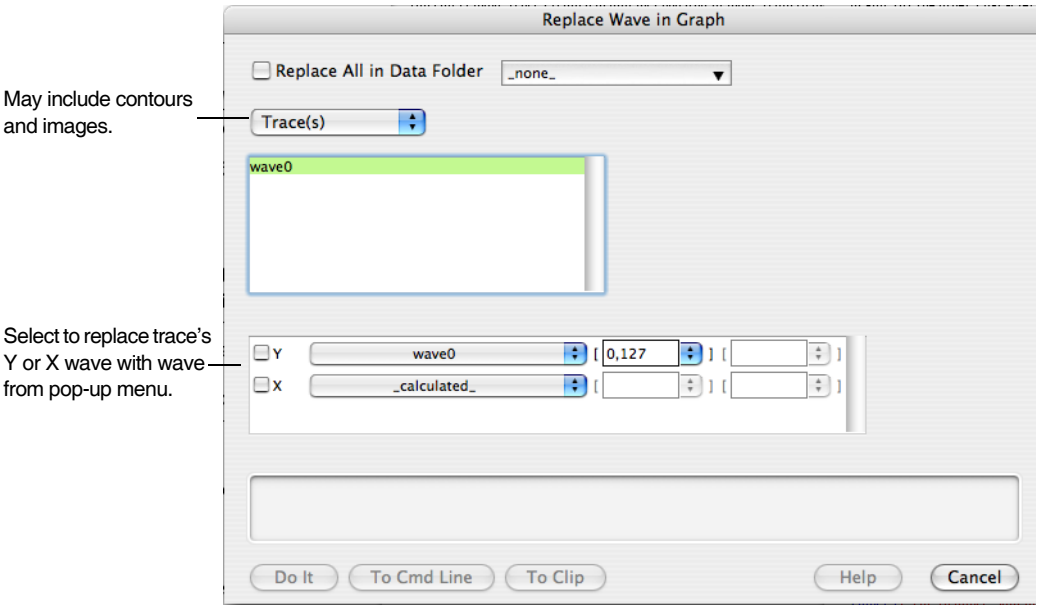
Note that a contour plot has traces that you can remove, but they will come back when the contour plot is updated. Rather than removing the contour traces, use the pop-up to select Contours, and remove the contour plot itself, which automatically removes all of the contour-related traces. See **Removing Contour Traces from a Graph** on page II-332.

If you remove the last wave associated with a given axis then that axis will also be removed. If you remove all the waves you will be left with a blank graph with no traces or axes.

Replacing Traces

You can "replace" a trace in the sense of changing the wave that the trace is displaying in a graph. All the other characteristics of the trace — such as mode, line size, color, and style — are unchanged. You can use this to update a graph with data from a wave other than the one originally used to create the trace.

To replace a trace, use the Replace Wave item in the Graph menu to display the Replace Wave in Graph dialog:



A special mode allows you to browse through groups of data sets composed of identically-named waves residing in different data folders (for a discussion of data folders, see Chapter II-8, **Data Folders**). For instance, you might take the same type of data during multiple runs on different experimental subjects. If you store the data for each run in a separate data folder, and you give the same names to the waves that result from each run, you can select the Replace All in Data Folder checkbox and then select one of the data folders containing data from a single run. All the waves in the chosen data folder whose names match the names of waves displayed in the graph will replace the same-named waves in the graph.

You can also replace waves one at a time with any other wave. With the Replace All in Data Folder checkbox unselected, choose a trace from the list below the menu. To replace the Y wave used by the trace, select the Y checkbox; to replace the X wave select the X checkbox. You can replace both if you wish. Select the waves to use as replacements from the menus to the right of the checkboxes. You can select _calculated_ from the X wave menu to remove the X wave of an XY pair, converting it to a waveform display.

The menus allow you to select waves having lengths that don't match the length of the corresponding X or Y wave. In that case, use the edit boxes to the right to select a sub-range of the wave's points. You can also use these boxes to select a single row or column from a two-dimensional wave.

The dialog creates command lines using the **ReplaceWave** operation (page V-524).

Plotting NaNs and INFs

The data value of a wave is normally a finite number but can also be a NaN or an INF. NaN means "Not a Number", and INF means "infinity". An expression returns the value NaN when it makes no sense mathematically. For example, `log(-1)` returns the value NaN. You can also set a point to NaN, using a table or a wave assignment, to represent a missing value. An expression returns the value INF when it makes sense mathematically but has no finite value. `log(0)` returns the value -INF.

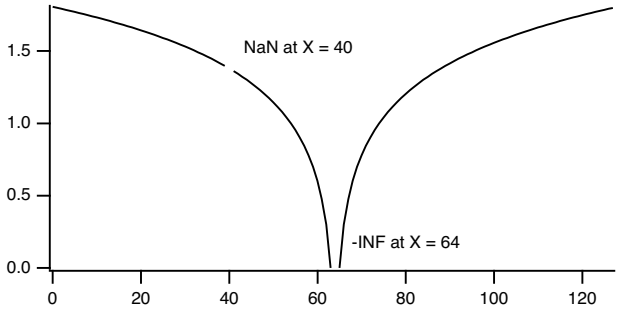
Igor ignores NaNs and INFs when scaling a graph. If a wave in a graph is set to lines between points mode then Igor draws lines toward an INF. By default, it draws no line to or from a NaN so a NaN acts like a

missing value. You can override the default, instructing Igor to draw lines through NaNs using the Gaps checkbox in the Modify Trace Appearance dialog.

The following graph, generated by the wave assignments

```
wave1= log(abs(x-64)); wave1(40)=log(-1)
```

illustrate these points.



You can override the default, instructing Igor to draw lines through NaNs. See **Gaps** on page II-260 for details.

Scaling Graphs

Igor provides several ways of scaling waves in graphs. All of them allow you to control what sections of your waves are displayed by setting the range of the graph's axes. Each axis is either autoscaled or manually scaled.

Autoscaling

When you first create a graph all of its axes are in the **autoscaling** mode. This means that Igor automatically adjusts the extent of the axes of the graph so that all of each wave in the graph is fully in view. If the data in the waves changes, the axes are automatically rescaled so that all waves remain fully in view.

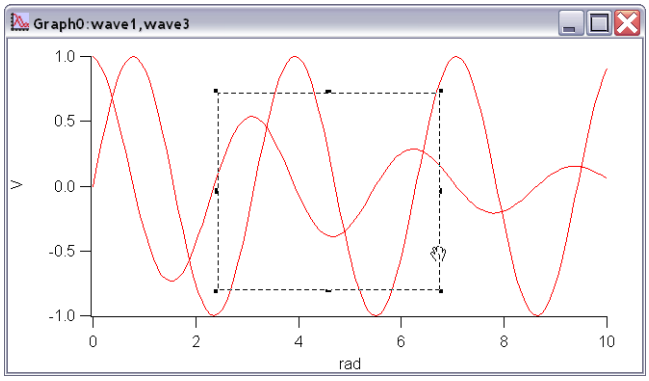
If you manually scale any axis, that axis changes to the **manual scaling** mode. The methods of manual scaling are described in the next section. Axes in manual scaling mode are never automatically scaled.

If you choose Autoscale Axes from the Graph menu all of the axes in the graph are autoscaled and returned to the autoscaling mode. You can set individual axes to autoscale mode and can control properties of the autoscale mode using the Axis Range tab of the Modify Axis dialog described in **Setting the Range of an Axis** on page II-244.

Manual Scaling

To manually scale one or more axes of a graph with the mouse, start by selecting the region of the graph that you want to examine. Then select the scaling operation that you want from a pop-up menu that appears when you click inside the region.

Click the mouse and drag it diagonally to frame the region of interest. Igor displays a dashed outline around the region. This outline is called a marquee. A marquee has handles and edges that allow you to refine its size and position.

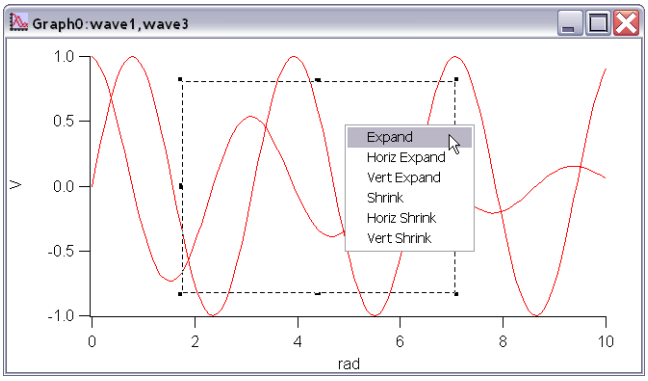


When you position the cursor over various parts of the marquee, the cursor shape indicates what clicking and dragging will do.

To refine the size of the marquee move the cursor over one of the handles. The cursor changes to a double arrow which shows you the direction in which the handle moves the edge of the marquee. To move the edge click the mouse and drag.

To refine the position of the marquee move the cursor over one of the edges away from the handles. The cursor changes to a hand. To move the marquee click the mouse and drag.

When you click inside the region of interest Igor presents a pop-up menu from which you can choose the scaling operation.



Choose the operation you want and release the mouse. These operations can be undone and redone; just press Command-Z (*Macintosh*) or Ctrl+Z (*Windows*).

The **expand** operation scales all axes so that the region inside the marquee fills the graph (zoom in). It sets the scaling mode for all axes to manual.

The **horiz expand** operation scales only the horizontal axes so that the region inside the marquee fills the graph horizontally. It has no effect on the vertical axes. It sets the scaling mode for the horizontal axes to manual.

The **vert expand** operation scales only the vertical axes so that the region inside the marquee fills the graph vertically. It has no effect on the horizontal axes. It sets the scaling mode for the vertical axes to manual.

The **shrink** operation scales all axes so that the waves in the graph appear smaller (zoom out). The factor by which the waves shrink is equal to the ratio of the size of the marquee to the size of the entire graph. For example, if the marquee is one half the size of the graph then the waves shrink to one half their former size. The point at the center of the marquee becomes the new center of the graph. The shrink operation sets the scaling mode for all axes to manual.

The **horiz shrink** operation is like the shrink operation but affects the horizontal axes only. It sets the scaling mode for the horizontal axes to manual.

The **vert shrink** operation is like the shrink operation but affects the vertical axes only. It sets the scaling mode for the vertical axes to manual.

Igor programmers can add their own items to this pop-up menu. Also, the coordinates of the marquee can be read under program control and can even be used to set up automatic (dependency) formulas. See the **GetMarquee** operation on page V-218.

Another way to manually scale axes is to use the Axis Range tab of the Modify Axis dialog (see **Manual Axis Ranges** on page II-244), or the **SetAxis** operation (page V-551).

Panning

After zooming in on a region of interest, you may want to view data that is just off screen. To do this, press Option (*Macintosh*) or Alt (*Windows*) and move the mouse to the graph interior where the cursor changes to a hand. Now drag the body of the graph. Pressing Shift will constrain movement to the horizontal or vertical directions.

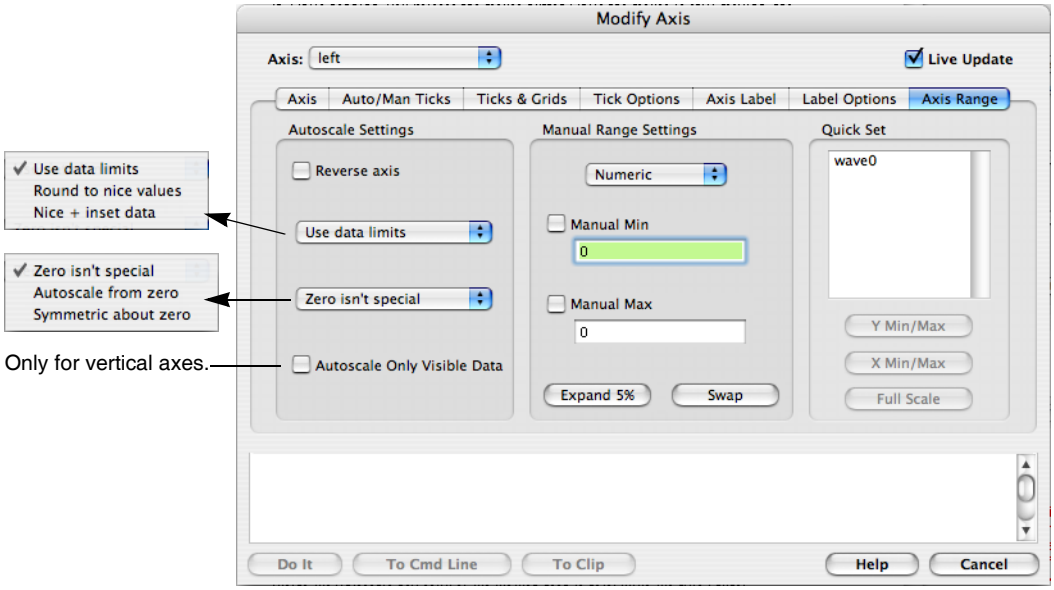
This operation is undoable.

Fling Mode

If, while panning, you release the mouse button while the mouse is still moving, the panning will automatically continue. While panning, release the Option or Alt key and change the force or direction of the mouse-click gesture to change the panning speed or direction. Click the mouse button once to stop.

Setting the Range of an Axis

You can set the range and other scaling parameters for individual axes using the Axis Range tab in the Modify Axis dialog. You can display the dialog with this tab selected by choosing Set Axis Range from the Graph menu or by double-clicking a tick mark label of the axis you wish to modify. Information on the other tabs in this dialog is available in **Modifying Axes** on page II-262.



Start by choosing the axis that you want to adjust from the Axis pop-up menu. You can adjust each axis in turn, or a selection of axes, without leaving this dialog.

Manual Axis Ranges

When a graph is first created, it is in autoscaling mode. In this mode, the axis limits automatically adjust to just include all the data. You can set the axis limits to fixed values by selecting the Manual Min and Manual Max checkboxes. When you select one of these checkboxes, both are automatically selected. You can elect to fix just one end of an axis range by de-selecting one of these checkboxes. The other end will then be adjusted by Igor to include the minimum or maximum value of the data.

The fixed limits are set by editing the values in the boxes below the Manual Min and Manual Max checkboxes. Initially these boxes are set to the automatic values of the limits.

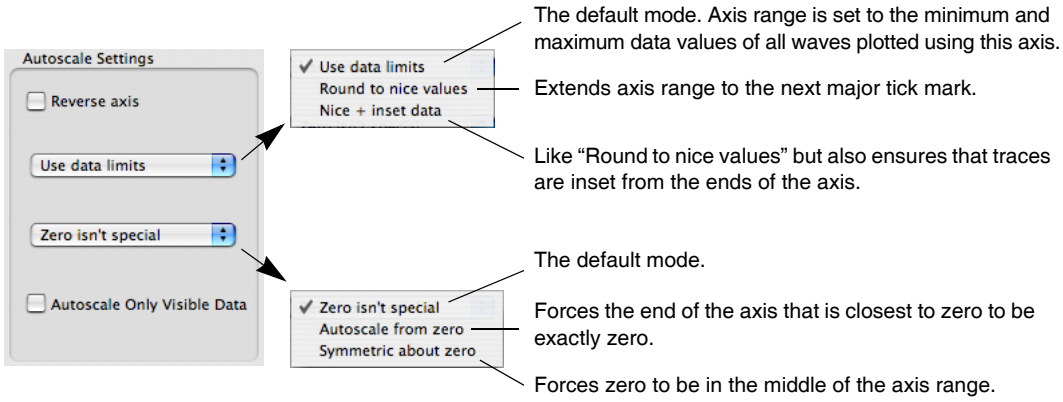
There are a number of other ways to set the Min and Max limits. Clicking the Expand 5% button expands the range between the min and the max by 5 percent. This has the effect of shrinking the graph traces plotted on the axis by 5%. Clicking the Swap button exchanges the Min and Max parameters. This has the effect of reversing the ends of the axis. You can plot waves upside-down or backwards. This works for linear, log

and date/time axes. If you don't want the axis limits to be fixed, use the Reverse Axis checkbox in the Autoscale Settings area.

An additional way to set the Min and Max parameters is to select a wave from the list and use the Quick Set buttons. If you click the X Min/Max quick set button then the minimum and maximum X values of the selected wave are transferred to the parameter boxes. If you click the Y Min/Max quick set button then the minimum and maximum Y values of the selected wave are transferred to the parameter boxes. If you specified the full scale Y values for the wave then you can click the Full Scale quick set button. This transfers the wave's Y full scale values to the parameter boxes. The full scale Y values can be set using the Change Wave Scaling item in the Data menu.

Automatic Axis Ranges

When the Manual Min and Manual Max checkboxes are not selected the given axis is in autoscaling mode. In this mode the axis limits are determined by the data values in the waves displayed using the selected axis. The items inside the Autoscale box control the method used to determine the axis range:

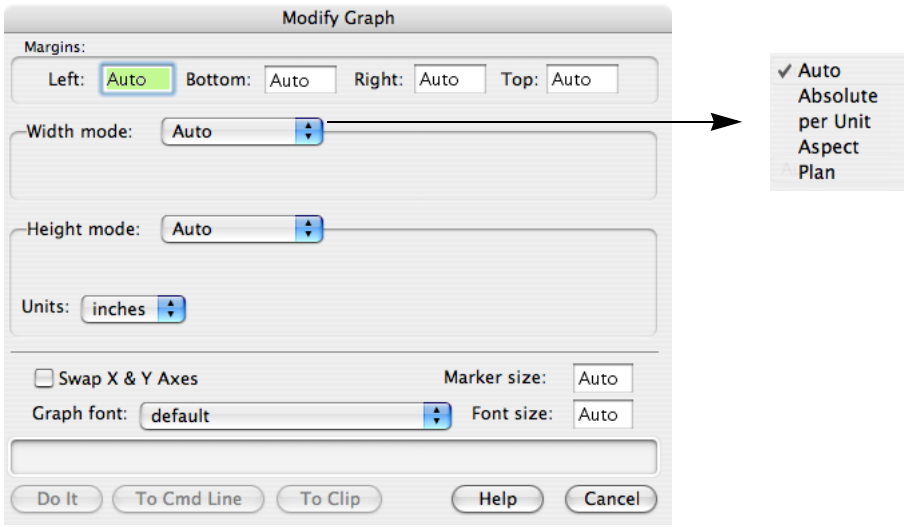


You can cause an autoscaled axis to be reversed by selecting the "Reverse axis" checkbox.

Autoscale mode usually sets the axis limits using all the data in waves associated with the traces that use the axis. This can be undesirable if the associate horizontal axis is set to display only a portion of the total X range. Select the Autoscale Only Visible Data checkbox to have Igor use only the data included within the horizontal range for autoscaling.

Overall Graph Properties

You can specify certain overall properties of a graph by choosing Modify Graph from the Graph menu. This brings up the Modify Graph dialog. You can also get to this dialog by double-clicking in any blank area outside the plot rectangle.



Normally, X axes are plotted horizontally and Y axes vertically. You can reverse this behavior by selecting the "Swap X & Y Axes" checkbox. This is commonly used when the independent variable is depth or height. This method swaps X and Y for all traces in the graph. You can cause individual traces to be plotted vertically by selecting the "Swap X & Y Axes" checkbox in the New Graph and Append Traces dialogs as you are creating your graph.

Initially, the graph font is determined by the default font which you can set using the Default Font item in the Misc menu. The graph font size is initially automatically calculated based on the size of the graph. You can override these initial settings using the "Graph font" and "Font size" settings. Igor uses the font and size you specify in annotations and axis labels unless you explicitly set the font or size for an individual annotation or label.

Initially, the graph marker size is automatically calculated based on the size of the graph. You can override this using the "Marker size" setting. You can set it to "auto" (or 0 which is equivalent) or to a number from 1 to 9. Igor uses the marker size you specify unless you explicitly set the marker size for an individual wave in the graph.

The margin is the distance from an outside edge of the graph to the edge of the plot area of the graph. The plot area, roughly speaking, is the area inside the axes. See **Graph Dimensions** on page II-246 for a definition. Initially, Igor automatically sets each margin to accommodate axis and tick mark labels and exterior textboxes, if any. You can override the automatic setting of the margin using the Margins settings. You would do this, for example, to force the left margins of two graphs to be identical so that they align properly when stacked vertically in a page layout. The Units pop-up menu determines the units in which you enter the margin values.

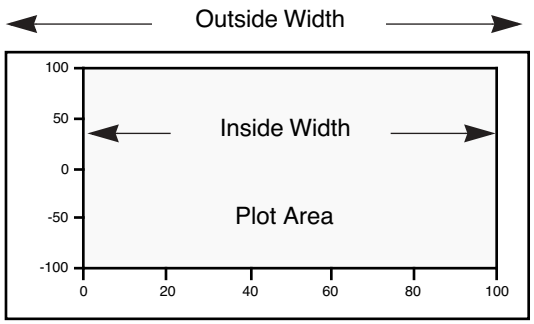
You can also set graph margins interactively. If you press Option (*Macintosh*) or Alt (*Windows*) and position the cursor over one of the four edges of the plot area rectangle, you will see the cursor change to this shape: ✣. Use this cursor to drag the margin. You can cause a margin to revert to automatic mode by dragging the margin all the way to the edge of the graph window (or beyond). If you drag to within a few pixels of the edge, the margin will be eliminated entirely (this requires good accuracy with the mouse). If you double click with this cursor showing you will get the Modify Graph dialog with the corresponding margin setting selected.

If you specify a margin for a given axis, the value you specify solely determines where the axis appears. Normally, dragging an axis will adjust its offset relative to the nominal automatic location. If, however, a fixed margin has been specified then dragging the axis will drag the margin.

Graph Dimensions

The Modify Graph dialog provides several ways of controlling the width and height of a graph. Usually you don't need to use these. They are intended for certain specialized applications.

These techniques are powerful but can be confusing unless you understand the algorithms, described below, that Igor uses to determine graph dimensions.



The graph can be in one of five modes with respect to each dimension: auto, absolute, per unit, aspect, or plan. These modes control the width and height of the **plot area** of the graph. The plot area is the shaded area in the illustration. The width mode and height mode are independent.

In this graph, the axis standoff feature, described in the **Modifying Axes** section on page II-262, is off so the plot area extends to the center of the axis lines. If it were on, the plot area would extend only to the inside edge of the axis lines.

Auto mode automatically determines the width or height of the plot area based on the outside dimensions of the graph and other factors that you specify using Igor's dialogs. This is the normal default mode which is appropriate for most graphing jobs. The remaining modes are useful for special purposes such as matching the axis lengths of two or more graphs or replicating a standard graph or a graph from a journal.

If you select any mode other than auto, you are putting a constraint on the width or height of the plot area which also affects the outside dimensions of the graph. If you adjust the outside size of the graph, by dragging the window's size box, by tiling, by stacking or by using the MoveWindow operation, Igor first determines the outside dimensions as you have specified them and then applies the constraints implied by the width/height mode that you have selected.

With **Absolute** mode, you specify the width or height of the plot area in absolute units; in inches, centimeters or points. For example, if you know that you want your *plot area* to be exactly 5 inches wide and 3.5 inches high, you should use those numbers with an absolute mode for both the width and height.

If you want the *outside* width and height to be an exact size, you must also specify a fixed value for all four margins. For instance, setting all margins to 0.5 inches in conjunction with an absolute width of 5 inches and a height of 3.5 inches yields a graph whose outside dimensions will be 6 inches wide by 4.5 inches high.

The **Aspect** mode maintains a constant aspect ratio for the plot area. For example, if you want the width to be 1.5 times longer than the height, you would set the width mode to aspect and specify an aspect ratio of 1.5.

The remaining modes, per unit and plan, are quite powerful and convenient for certain specialized types of graphs, but are more difficult to understand. You should expect that some experimentation will be required to get the desired results.

In **Per unit** mode, you specify the width or height of the plot area in units of length per axis unit. For example, suppose you want the plot width to be one inch per 20 axis units. You would specify 1/20 = 0.05 inches per unit of the bottom axis. If your axis spanned 60 units, the plot width would be three inches.

In **Plan** mode, you specify the length of a unit in the horizontal dimension as a scaling factor times the length of a unit in the vertical dimension, or vice versa. The simplest use of plan scaling is to force a unit in one dimension to be the same as in the other. To do this, you select plan scaling for one dimension and set the scaling factor to 1.

Until you learn how to use the per unit and plan modes, it is easy to create a graph that is ridiculously small or large. Since the size of the graph is tied to the range of the axes, expanding, shrinking or autoscaling the graph makes its size change.

You can also get confusing results if you over-constrain Igor. For example, it is possible to specify that the width should be 1.5 times the height and that the height should be 1.5 times the width. You should avoid this.

Sometimes you can end up with a graph whose size makes it difficult to move or resize the window. Use the Graph menu's Modify Graph dialog to reset the size of the graph to something more manageable.

You may get surprising results when these modes are used in combination with the Fill Page, Custom Size and Same Aspect radio buttons in the Print Graphs dialog. This is because of interactions between the effects of the radio buttons and the modes. The Same Size radio button does not cause interaction and therefore is the simplest to use.

If you want to fully understand how Igor arrives at the final size of a graph when the width or height is constrained, you need to understand the algorithm Igor uses:

1. The initial width and height are calculated. When you adjust a window by dragging, the initial width and height are based on the width and height to which you drag the window. When you print a graph, the initial width and height are the width and height of the graph window.
2. If you are printing, the width and height are modified by the effects of the printing mode that you have selected (full page, custom size, same size or same aspect). Usually, when using a width or height mode, you should print the graph using the same size radio button.
3. The width modes absolute and per unit are applied which may generate a new width.
4. The height mode is applied which may generate a new height.
5. The width modes aspect and plan are applied which may generate a new width.

Because there are many interactions, it is possible to get a graph so big that you can't adjust it manually. If this occurs, use the Modify Graph dialog to set the width and height to a manageable size, using absolute mode.

When you are about to print, you can get a feel for the outcome of this algorithm by adjusting the graph window to the shape of a page and then observing how the algorithm alters the graph size.

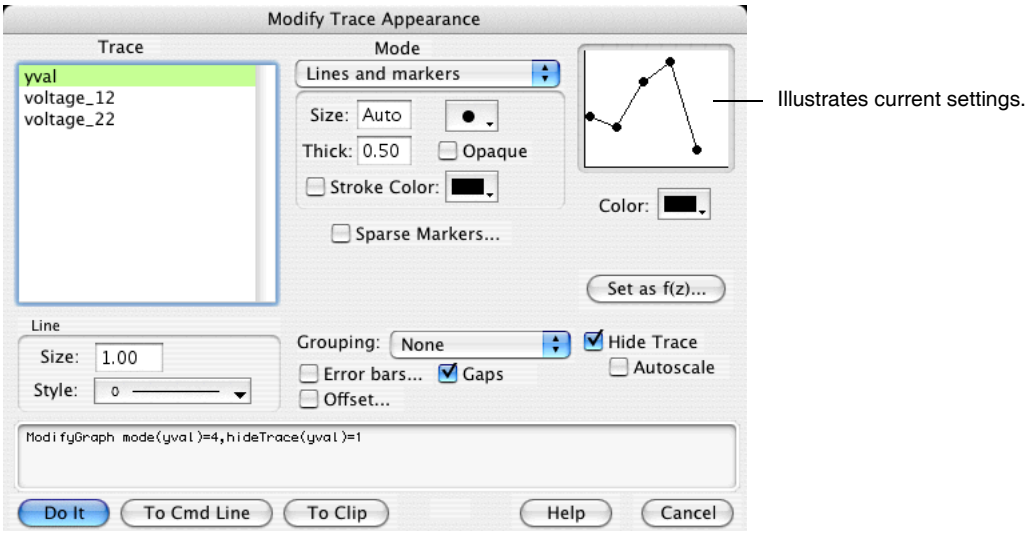
There is an additional overall graph property that you can access only via the command line. If you live in a country that uses "," rather than "." as the decimal separator in numbers then you can use the following command to cause tick mark labels to use comma as the separator:

```
ModifyGraph useComma=1
```

This displays tick mark labels such as "1,000.0" as "1.000,0".

Modifying Styles

You can specify each trace's appearance (or style) in a graph by choosing Modify Trace Appearance from the Graph menu or by double-clicking a trace in the graph. This brings up the following dialog:



For image plots, choose Modify Image Appearance from the Graph menu, rather than Modify Trace Appearance.

For contour plots, you normally should choose Modify Contour Appearance. Use this to control the appearance of all of the contour lines in one step. However, if you want to make one contour line stand out, use the Modify Trace Appearance dialog. For example, you could make the contour line at z=0 a dashed line to call special attention to it.

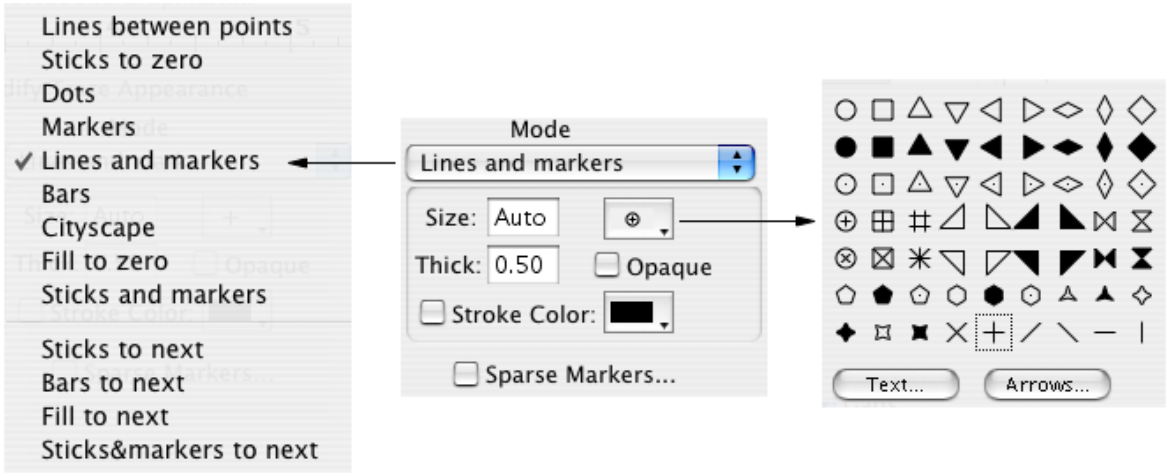
Selecting Traces to be Modified

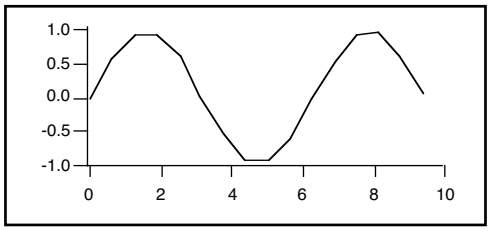
Select the trace or traces whose appearance you want to modify from the Trace list. If you got to this dialog by double-clicking a trace in the graph then that trace will automatically be selected. If you select more than one trace, the items in the dialog will show the settings for the *first* selected trace.

Once you have made your selection, you can change settings for the selected traces. After doing this, you can then select a different trace or set of traces and make more changes. Igor remembers all of the changes you make, allowing you to do everything in one trip to the dialog.

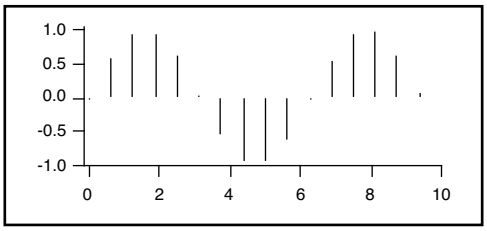
Display Modes

Choose the **mode** of presentation for the selected trace from the Mode pop-up menu.

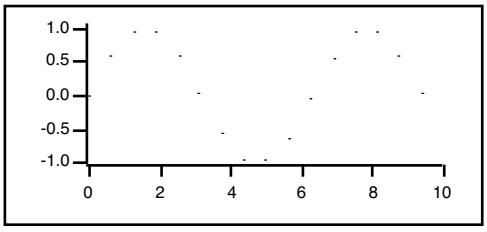




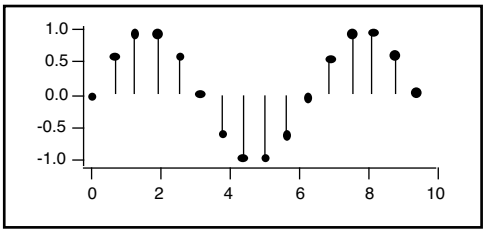
Lines between points mode



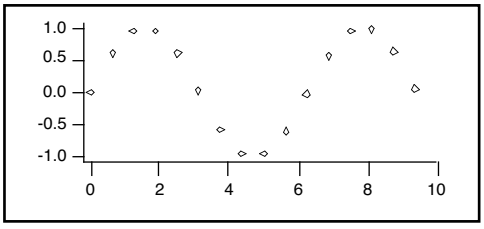
Sticks to zero mode



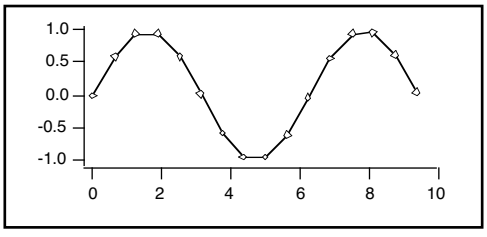
Dots mode



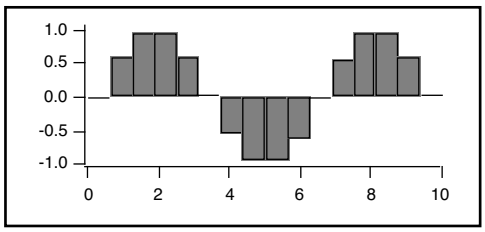
Sticks and markers mode



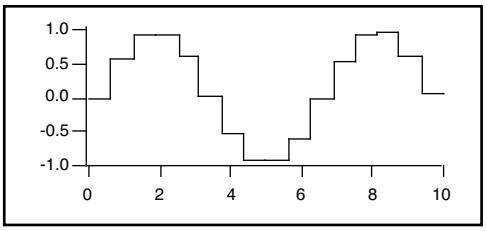
Markers mode



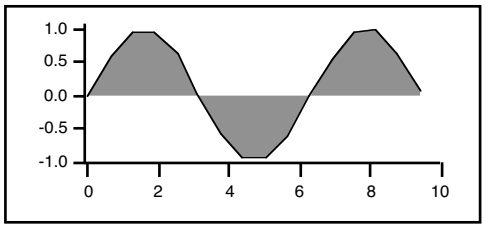
Lines and markers mode



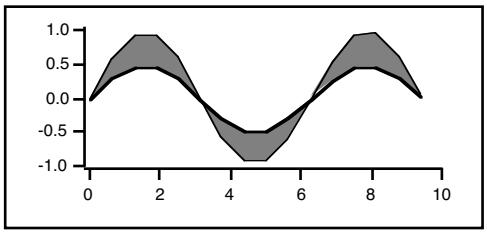
Bars mode



Cityscape mode



Fill to zero mode



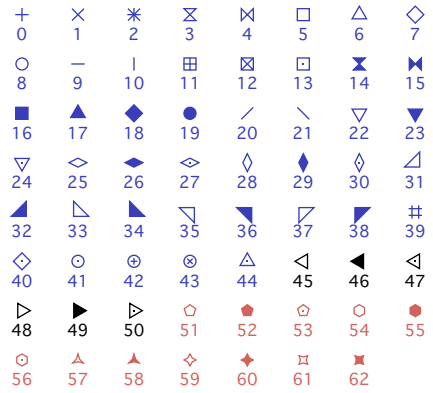
Fill to next mode

Markers

If you choose the Markers or Lines and Markers mode you also get to choose the marker, marker size, marker thickness, and whether the marker is opaque or not. The marker size is a fractional number from 1.0 to 9.0. It can also be set to 0 or "auto", which chooses a marker size appropriate to the size of the graph. The marker thickness is in points and can also be fractional. Setting the marker thickness to 0 makes the markers disappear. The screen does not have enough resolution to show fractional points but they are evident when you print the graph.

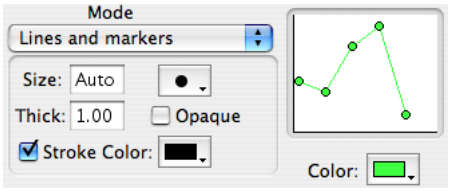
There is an interaction between marker size and marker thickness: Igor will adjust the marker size if this is needed to make the marker symmetrical. The unadjusted width and height of the marker is 2*s+1 points where s is the marker size setting.

Here is a table of the markers and the corresponding marker codes:



Markers 0-44 are compatible with Igor Pro 4. Markers 45 through 50 require Igor Pro 5.0 or later. Markers 51 through 62 require Igor Pro 6.1 or later and are available in new graphics only (see **Graphics Technology** on page III-421). You can also create custom markers. See the **SetWindow** operation's markerHook keyword.

Stroke Color



In the Markers or Lines and Markers modes you can specify a color for marker objects that is different from the fill color for the markers. To use this select the Stroke Color checkbox and select a color from the adjacent pop-up menu.

Text Markers

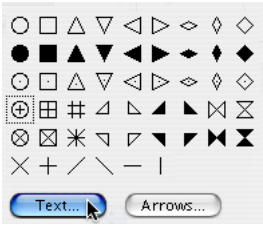
In addition to the built-in drawn markers, you can also instruct Igor to use one of the following as text markers:

- A single character from a font
- The contents of a text wave
- The contents of a numeric wave

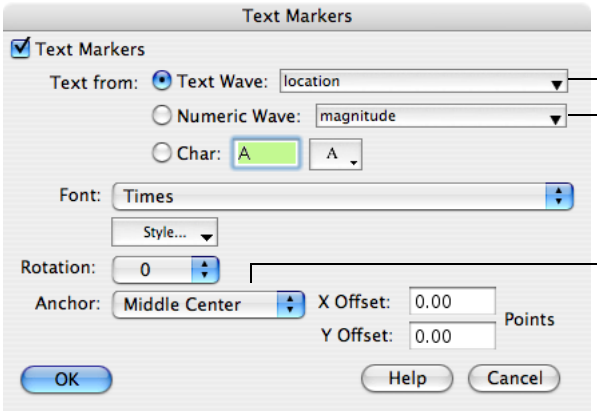
A single character from a font is mainly of interest if you want to use a special symbol that is available in a font but is not included among Igor's built-in markers. The specified character is used for all data points.

The remaining options provide a way to display a third value in an XY plot. For example, a plot of earthquake magnitude versus date could also show the location of the earthquake using a text wave to supply text markers. Or, a plot of earthquake location versus date could also show the magnitude using a numeric wave to supply text markers. For each data point in the XY plot, the corresponding point of the text or numeric wave supplies the text for the marker. The marker wave must have the same number of points as the X and Y waves.

To create a text marker, choose the Markers or Lines and Markers display mode. Then click the Markers pop-up menu and choose the Text button.



This leads to a subdialog in which you can specify the source of the text as well as the font, style, rotation and other properties of the markers.



These pop-up menus show only waves with the same number of points as the main XY waves.

Determines the reference point on the rectangle enclosing the text. Igor places this reference point at the coordinates of the data point, plus the X and Y offset.

You can offset and rotate all the text markers by the same amount but you can not set the offset and rotation for individual data points — use tags for that. You may find it necessary to experimentally adjust the X and Y offsets to get character markers exactly centered on the data points. For example, to center the text just above each data point, choose Middle bottom from the Anchor pop-up menu and set the Y offset to 5-10 points. If you need to offset some items differently from others, you will have to use tags (see **Tags** on page III-54).

Igor determines the font size to use for text markers from the marker size, which you set in the Modify Trace Appearance dialog. The font size used is 3 times the marker size.

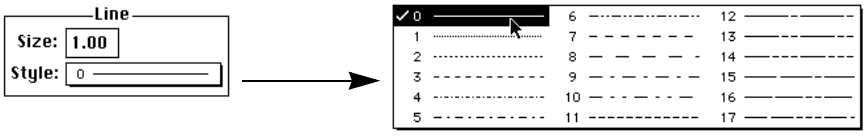
You may want to show a text marker *and* a regular drawn marker. For this, you will need to display the wave twice in the graph. After creating the graph and setting the trace to use a drawn marker, use Append Traces to Graph (Graph menu) to append a second copy of the wave. Set this copy to use text markers.

Arrow Markers

Arrow markers can be used to create vector plots illustrating flow and gradient fields, for example. Arrow markers are fairly special purpose and require quite a bit of advance preparation. See the reference for a description of the arrowMarker keyword under the **ModifyGraph (traces)** operation on page V-400 for further details.

Line Styles and Sizes

If you choose the "Lines between points", "Lines and markers", or Cityscape mode you also get to choose the line style. You can change the dash patterns using the Dashed Lines item in the Misc main menu.

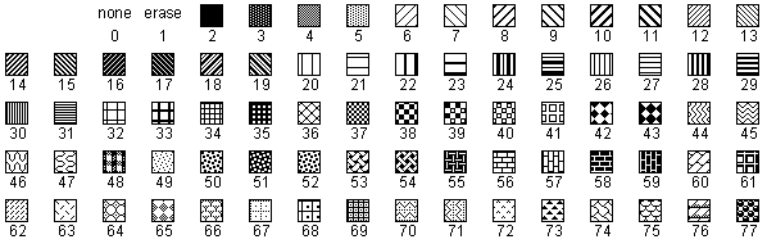


For any mode except the Markers mode you can set the line size. The line size is in points and can be fractional. If the line size is zero, the line disappears.

Fills

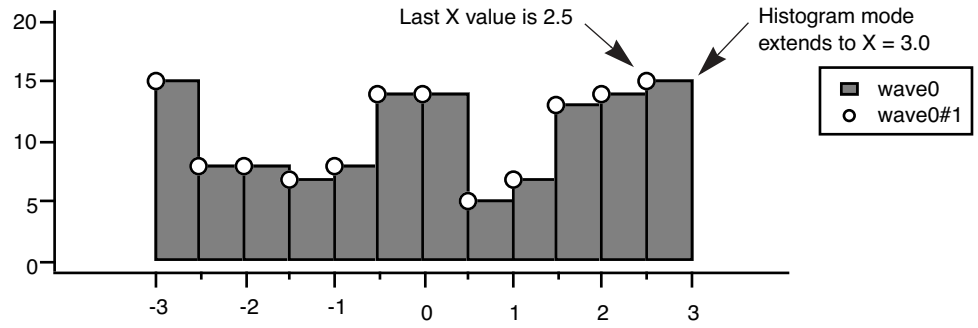
For traces in the Bars and "Fill to zero" modes, Igor presents a choice of fill type. The fill type can be None, which means the fill is transparent, Erase, which means the fill is white and opaque, Solid, or three patterns of gray. You can also choose a pattern from a palette and can choose the fill types and colors for positive going regions and negative going regions independently.

Here is a table of fill patterns with the corresponding numeric codes:



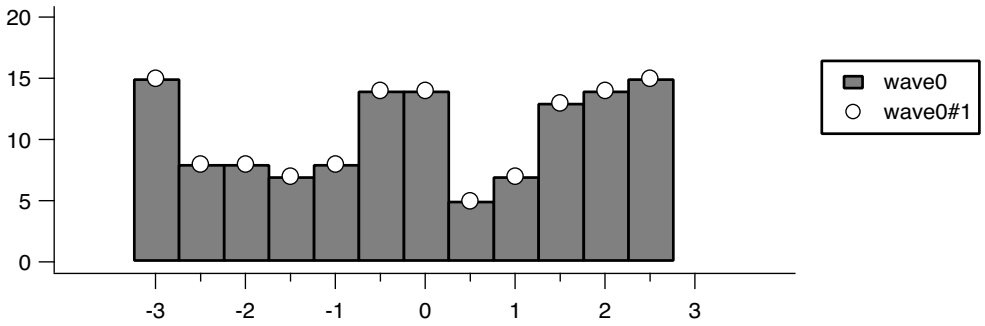
Bars

When Bars mode is used for a wave plotted on a normal continuous X axis (rather than a category axis, see Chapter II-13, **Category Plots**), the bars are drawn from the X value for a given point up to *but not including* the X value for the next point. Such bars are commonly called "histogram bars" because they are usually used to show the number of counts in a histogram that fall between the two X values.



If you want your bars centered on their X values, then you should create a Category Plot, which is more suited for traditional bar charts (see Chapter II-13, **Category Plots**). You can, however, adjust the X values for the wave so that the flat areas appear centered about its original X value as for a traditional bar chart. One way to do this without actually modifying any data is to offset the trace in the graph by one half the bar width. You can just drag it, or use the Modify Trace Appearance dialog to generate a more precise offset command. In our example, the bars are 0.5 X units wide:

```
ModifyGraph offset(wave0)={-0.25,0}
```

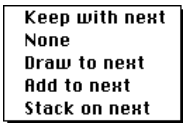


Grouping, Stacking and Adding Modes

For the four modes that normally draw to y=0 ("Sticks to zero", "Bars", "Fill to zero", and "Sticks and markers") you can choose variants that draw to the Y values of the next trace. The four variant modes are: "Sticks to next", "Bars to next", "Fill to next" and "Sticks&markers to next". *Next* in this context refers to the trace listed after (below) the selected trace in the list of traces in the Modify Trace Appearance and the Reorder Traces dialogs.

If you choose one of these four modes, Igor automatically selects "Draw to next" from the Grouping pop-up menu. You can also choose "Add to next" and "Stack on next" modes.

The Grouping pop-up menu is used to create special effects such as layer graphs and stacked bar charts. "Keep with next" is used only with category plots and is described in Chapter II-13, **Category Plots**.

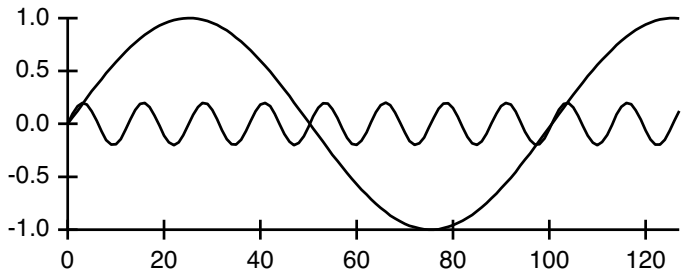


"Draw to next" modifies the action of those modes that normally draw to y=0 so that they draw to the Y values of the next trace that is plotted against the same pair of axes as the current trace. The X values for the next trace should be the same as the X values for the current trace. If not, the next trace will not line up with the bottom of the current trace.

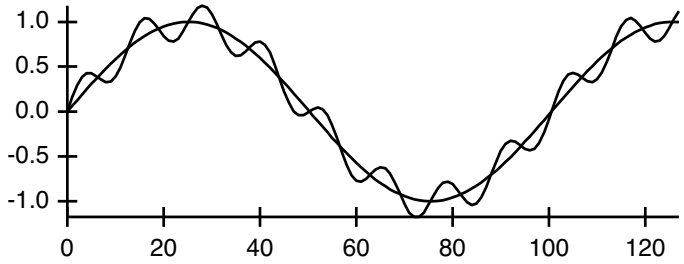
"Add to next" adds the Y values of the current trace to the Y values of the next trace before plotting. If the next trace is also using "Add to next" then that addition is performed first and so on. When used with one of the four modes that normally draw to y=0, this mode also acts like "Draw to next".

"Stack on next" works just like "Add to next" except Y values are not allowed to backtrack. On other words, negative values act like zero when the Y value of the next trace is positive and positive values act like zero with a negative next trace.

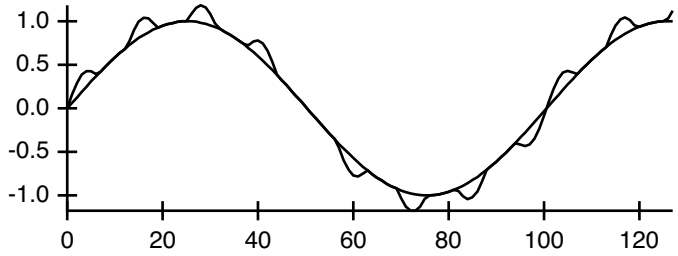
Here is a normal plot of a small sine wave and a bigger sine wave:



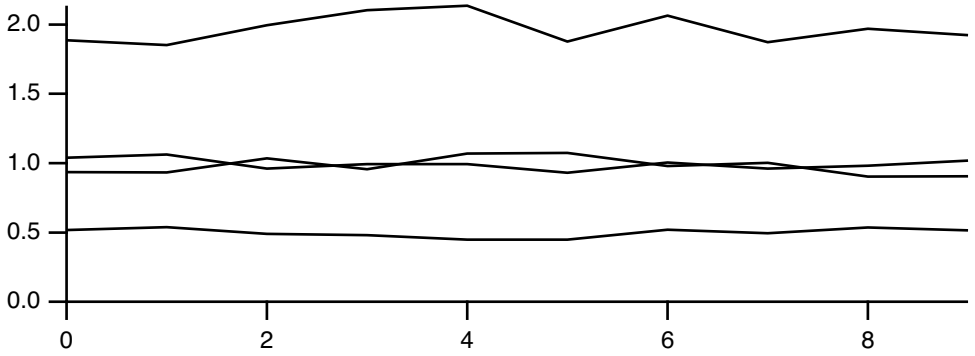
In this version, the small sine wave is set to "Add to next" mode:



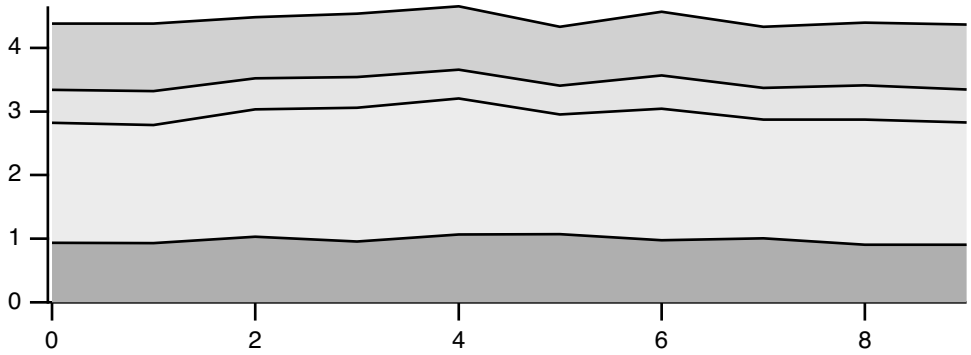
And here we use "Stack on next":



You can create layer graphs by plotting a number of waves in a graph using the fill to next mode. Depending on your data you may also want to use the add to next grouping mode. For example, in the following normal graph, each trace might represent the thickness of a geologic layer:



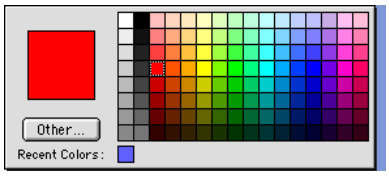
We can show the layers in a more understandable way by using fill to next and add to next modes:



Because all the Grouping modes depend on the identity of the next trace, you may need to adjust the ordering of traces in a graph. You can do this using the Reorder Traces dialog. Choose Reorder Traces from the Graph menu. Select traces you want to move. Adjust the ordering by dragging the selected traces up and down in the list, dropping them in the appropriate spot.

Note: All of the waves you use for the various grouping, adding, and stacking modes should have the same numbers of points, X scaling, and all be displayed using the same axes. Otherwise, if there is not a point-to-point correspondence between the traces in your graph, you will get rather unusual and confusing results when using any of these various graphing modes.

Color



You can choose a color for the selected trace from the Color pop-up palette of colors.

If you don't see a color you like, drag to the Other button and release. You can then choose a color from the standard color picker dialog.

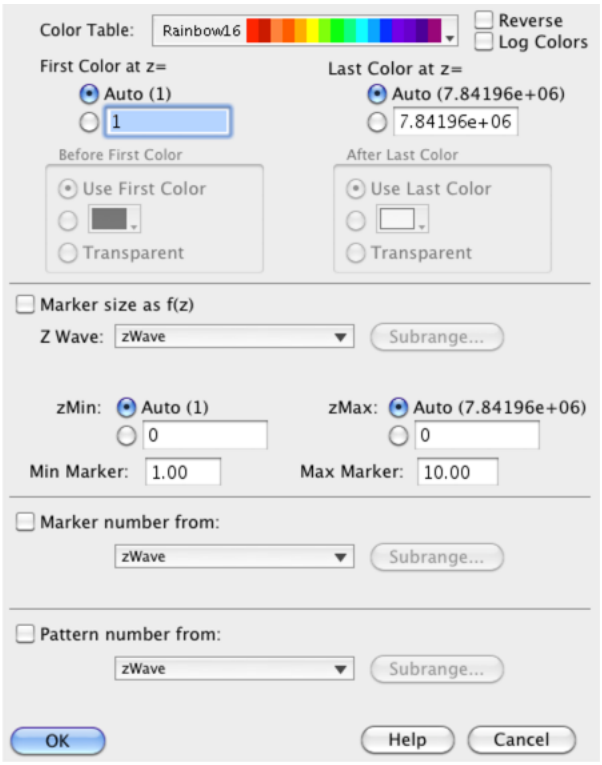
For more about the color pop-up palette, see **The Color Environment** on page III-410.

Setting Trace Properties from an Auxiliary (Z) Wave

You can set the color of a trace on a point-by-point basis as a function of the values in an auxiliary wave. You can also have the size of markers be a function of an auxiliary wave and you can set the marker number directly from an auxiliary wave. The auxiliary wave is called the "Z wave" because other waves control the X and Y position of a particular point on a trace while the Z wave controls a third property.

Setting the color or the marker size as a function of the values in an auxiliary wave can show three-dimensional data (X,Y, and Z) on a two-dimensional plot. For example, you could position markers at the location of earthquakes and vary their size to show the magnitude of each quake. You could show the depth of the quake using marker color and show different types of quakes as different marker shapes.

If you click the "Set as f(z)" button, you will see the following dialog:



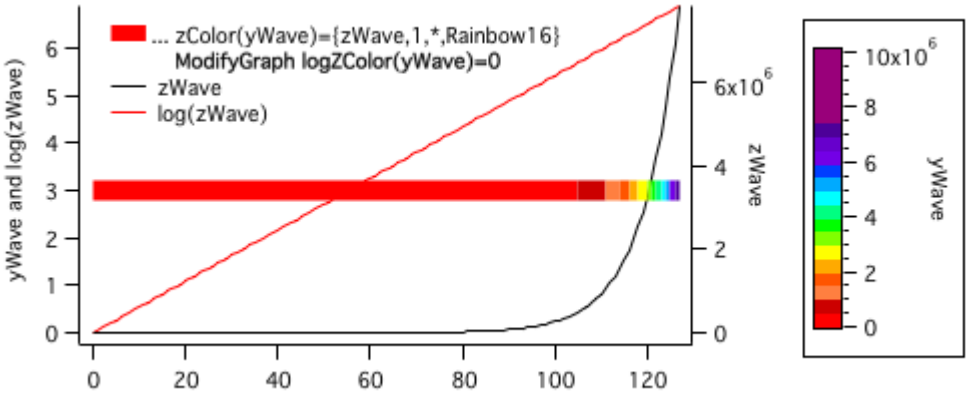
The Z Wave pop-up browser lists all waves that are the same length or greater than the trace to be colored. It also shows multicolumn waves. For a longer f(z) wave, or a multicolumn wave, the Subrange button becomes available so that you can select a specific point range or column from the wave.

Color as f(z) has three modes: Color Table mode, Color Index Wave mode, and Three-column Color Wave mode. These are selected in from the Color Mode menu.

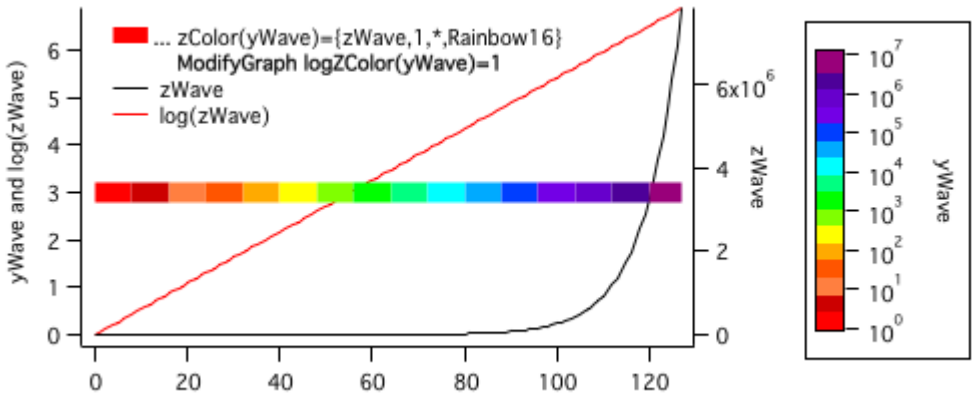
If you select “Color as f(z)” and Color Table mode, the color of data points on the trace will be derived from the Z wave you choose by linearly mapping its values into a built-in color table.

If you select 'Color as f(z)' and Color Table mode, the color of data points on the trace is derived from the Z wave you choose by mapping its values into a built-in color table either linearly or logarithmically if the Log Colors checkbox is checked.

You may wish to use the Log Colors option when the zWave spans many decades and you want to show more detailed changes of the smaller values. With the normal linear colors, this exponential zWave (shown in black and the log of zWave is shown in red) when applied to the thick yWave trace results in a trace that is mostly red:



Using `ModifyGraph logZColor(yWave)=1` spreads the colors out (the ColorScale has also been set to use a log axis):



The zMin and zMax settings define the range of values in your Z wave to map onto the color table. Values outside the range will take on the color at the end of the range. If you choose Auto for zMin or zMax, Igor will use the smallest or largest value it finds in your Z wave. If any of your Z values are NaN, Igor will treat those data points in the same way it does if your X or Y data is NaN. This depends on the Gaps setting in the main dialog.

If you select Color Index Wave mode, the color of data points on the trace will be derived from the Z wave you choose by mapping its values into the X scaling of the selected 3-column Color Index Wave. This is similar to the way **ModifyImage** cindex maps image values (in place of the Z wave values) to a color in a 3-column color index matrix. See **Indexed Color Details** on page II-356.

If you select Three-column Color Wave mode, data points are colored according to Red, Green and Blue values in the three columns of the selected wave. Each row of the three-column wave corresponds to a data point on the trace. This mode gives absolute control over the colors of each data point on a trace.

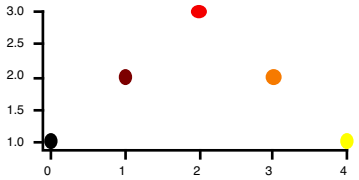
Create a graph:

```
make/N=5 Ywave={1,2,3,2,1}
display Ywave
ModifyGraph mode=3,marker=19,msize=5
```

Then make a Z wave, select Color Table mode and the YellowHot color table:

```
Make/N=5 zWave = {1,2,3,4,5}
ModifyGraph zColor(Ywave)={zWave,*,*,YellowHot}
```

These commands generate this graph:



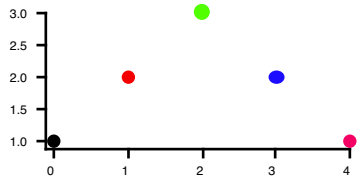
If instead you create a three-column wave and edit it to enter RGB values:

```
Make/N=(5,3) directColorWave
```

Row	directColorWav	directColorWav	directColorWav	
	0	1	2	
0	0	0	0	Black
1	65535	0	0	Red
2	0	65535	0	Green
3	0	0	65535	Blue
4	65535	0	26214	Hot pink

You can use this wave to directly control the marker colors:

```
ModifyGraph zColor(Ywave)={directColorWave,*,*,directRGB}
```



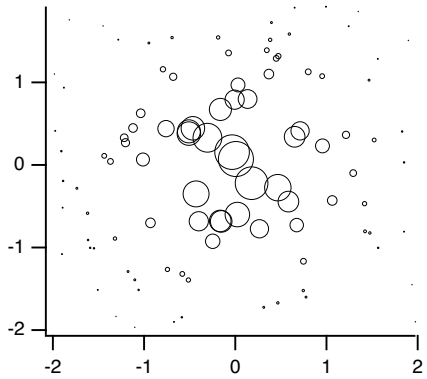
"Marker size as f(z)" works just like "Color as f(z)" in Color Table mode except the Z values map into the range of marker sizes that you define using the min and max marker settings.

The "Marker number from Z wave" mode does not do any mapping. You must create a Z wave that contains the actual marker numbers for each data point. See **Markers** on page II-250 for the marker number codes.

When you choose one or more of the property-as-f(z) modes, the corresponding item in the parent dialog is replaced with an "f(z)" button. This indicates that the given property is being set from an auxiliary wave. Clicking the button presents the same dialog shown above.

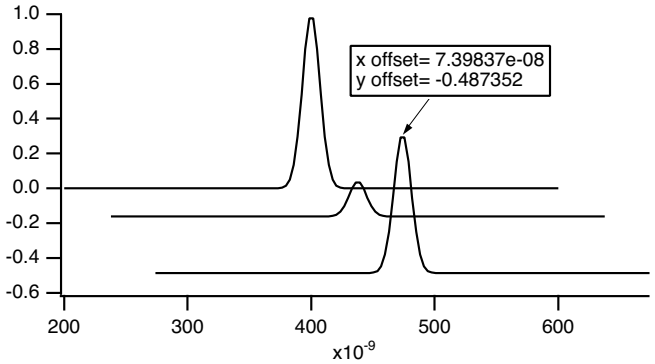
Here is an example that presents a third value as a function of marker size:

```
Make/N=100 datax,datay,dataz
datax=enoise(2); datay=enoise(2); dataz=exp(-(datax^2+datay^2))
Display datay vs datax; ModifyGraph mode=3,marker=8
ModifyGraph zmrkSize(datay)={dataz,*,*,1,10}
```



Trace Offsets

You can offset a trace in a graph in the horizontal or vertical direction without changing the data in the associated wave. This is primarily of use to offset traces so that you can compare their shape even though they have different baseline values, to offset traces which have the same baseline value so that you can spread them out, or to create a "poor man's waterfall plot" (i.e., a waterfall plot without any hidden line removal). See **Waterfall Plots** on page II-296 for more details about using Igor's built-in capabilities for creating waterfall plots.



Each trace has an X and a Y offset, both of which are initially zero. If you select the Offset checkbox in the Modify Trace Appearance dialog, you can use the Trace Offset subdialog to enter an X and Y offset for the trace selected in the main dialog.

You can also set the offsets by clicking and dragging in the graph. To do this, click the trace you want to offset. Hold the mouse down for about a second. You will see a readout box appear in the lower left of the graph. The readout shows the X and Y offsets as you drag the trace. If it doesn't take too long to display the given trace, you will be able to view the trace as you drag it around on the screen. If the cursor changes to a four pointed arrow then Igor has calculated that live update will be too slow. Drag the arrow to the spot on the graph where you want the point that you clicked on to be moved. In either case, when you release the mouse, Igor will set the wave's X and Y offsets appropriately.

If you press Shift while offsetting a wave, Igor will constrain the offset to the horizontal or vertical dimension.

You can disable trace dragging by pressing Caps Lock, which may be useful for trackball users.

Offsetting is undoable, so if you accidently drag a trace where you don't want it, choose Edit →Undo.

It is possible to attach a tag to a trace that will show its current offset values. See **Dynamic Escape Codes for Tags** on page III-46, for details.

If autoscaling is in effect for the graph, Igor tries to take trace offsets into account. If you want to set a trace's offset without affecting axis scaling, use the Set Axis Range item in the Graphs menu to disable autoscaling.

When offsetting a trace that uses log axes, the trace offsets by the same distance it does when the axis is not log. The shape of the trace is not changed — it is simply moved. If you were to try to offset a trace by adding a constant to the wave's data, it would distort the trace.

Trace Multipliers

In addition to offsetting a trace, as of Igor Pro 6, you can also provide a multiplier to scale a trace. The effective value used for plotting is then *multiplier*data+offset*. The Trace Offset subdialog also contains entries for the multiplier. Note that the default value of zero means that no multiplier is provided — not that the data should be multiplied by zero.

You can interactively scale a trace using the same click and hold technique described for trace offsets. But first, you must place Cursor A somewhere on the trace to act as a reference point. Then, after entering offset mode, you can press Option (*Macintosh*) or Alt (*Windows*) to adjust the scaling. You can press and release the key as desired to alternate between scaling and offsetting.

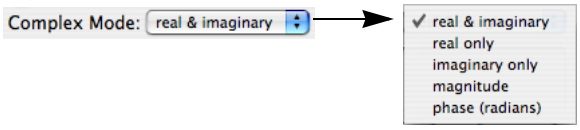
The trace multiplier feature is an alternative method of offsetting a trace on a log axis (remember: log(*a***b*)=log(*a*)+log(*b*)). For compatibility reasons and because the trace offsets method better handles switching between log and linear axis modes, the multiplier method applies when interactively dragging a trace only if the offset is zero and the multiplier is not zero (the default meaning "not set"). Consequently, to use the new technique, you must use the command line or the Trace Offset subdialog to set a nonzero multiplier (1 can be used).

Hiding Traces

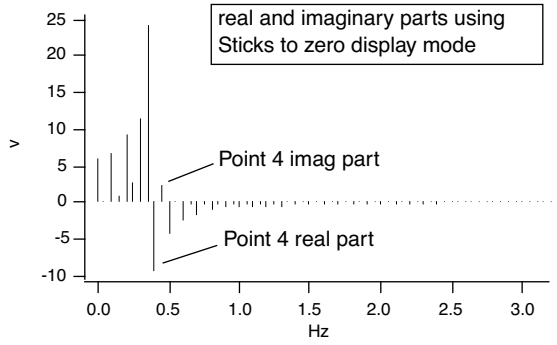
You can hide a trace in a graph, without removing the trace from the graph or by changing any other properties of the trace, by selecting the Hide Trace checkbox in the Modify Trace Appearance dialog. When you hide a trace, you can use the Autoscale checkbox to control whether or not the data of the hidden trace should be used when autoscaling the graph.

Complex Display Modes

When displaying traces for complex data you can use the Complex Mode pop-up menu to control how the data are displayed. You can display the complex and real components together or individually, and you can also display the magnitude or phase.



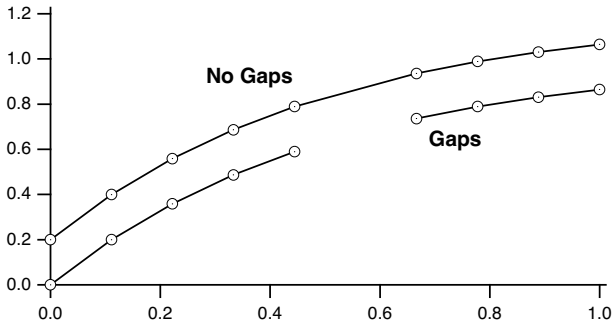
The default display mode is Lines between points. To display a wave's real and imaginary parts side-by-side on a point-for-point basis, use the Sticks to zero mode.



Gaps

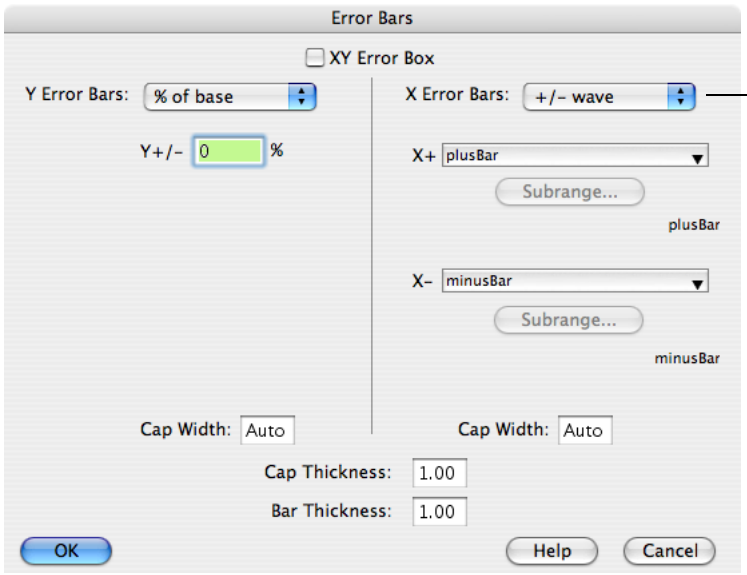
In Igor, a missing or undefined value in a wave is stored as the floating point value NaN ("Not a Number"). Normally, Igor shows a NaN in a graph as a gap, indicating that there is no data at that point. In some circumstances, it is preferable to treat a missing value by connecting the points on either side of it.

You can control this using the Gaps checkbox in the Modify Trace Appearance dialog. If this checkbox is selected (the default), Igor shows missing values as gaps in the data. If you deselect this checkbox, Igor ignores missing values, connecting the available data points on either side of the missing value.



Error Bars

The Error Bars checkbox adds error bars to the selected trace. When you select this checkbox, Igor presents the Error Bars subdialog.



Select error waves that are the same length or greater than the base. For a longer error wave, the Subrange button becomes available so that you can select a specific point range from the wave.

Error bars are a style that you can add to a trace in a graph. Error values can be a constant number, a fixed percent of the value of the wave at the given point, the square root of the value of the wave at the given point, or they can be arbitrary values taken from other waves. In this last case, the error values can be set independently for the up, down, left and right directions. See the **ErrorBars** operation on page V-143 for an illustration of the names for the various parts of error bars.

Choose the desired mode from the "Y Error bars" and "X Error bars" pop-up menus.

The dialog changes depending on the selected mode. For the "% of base" mode, you enter the percent of the base wave. For the "sqrt of base" mode, you don't need to enter any further values. This mode is meaningful only when your data is in counts. For the "constant" mode, you enter the constant error value for the X or Y direction. For the "+/- wave" mode, you select the waves to supply the positive and negative error values.

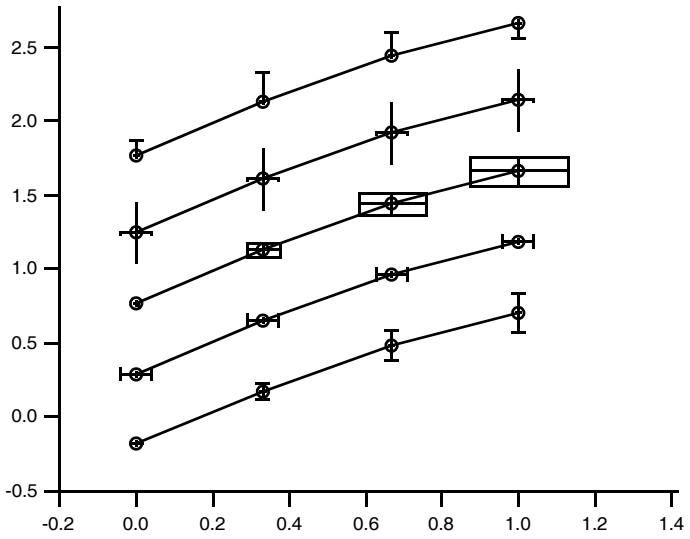
If you select "+/- wave", pop-up menus appear from which you can choose the waves to supply the upper and lower or left and right error values, These waves are called **error waves**. The values in error waves should normally all be positive since they specify the length of the line from each point to its error bar. This is the only mode that supports single-sided error bars. Error waves do not have to have the same numeric type and length as the base wave. If the value of a point in an error wave is NaN then the error bar corresponding to that point is not drawn.

The "Cap width" setting sets the width of the cap on the end of an error bar as an integral number of points. You can also set the "Cap width" to "auto" (or to zero) in which case Igor picks a cap width appropriate for the size of the graph. In this case the cap width is set to twice the size of the marker plus one. For best results the cap width should be an odd number.

For any mode you can set the thickness of the cap and the thickness of the error bar. The units for these settings are points. These can be fractional numbers. Although only integral thicknesses can be displayed on the screen, nonintegral thicknesses are properly produced on high resolution hard-copy devices. If you set "Cap thickness" to zero no caps are drawn. If you set "Bar thickness" to zero no error bars are drawn.

If you enable the "XY Error box" checkbox then a box is drawn rather than an error bar to indicate the region of uncertainty. No box is drawn for points for which one or more of the error values is NaN.

Here is a simple example of a graph with error bars.



The top trace used the "+/- Wave" mode with only a +wave. The last value of the error wave was made negative to reverse the direction of the error bar.

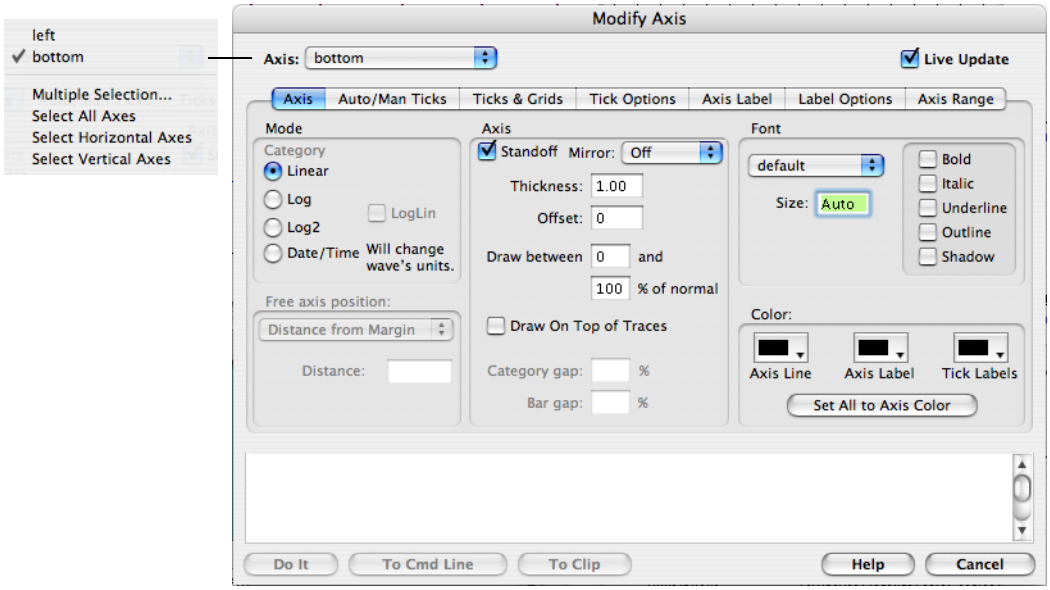
Customize at Point

You can customize the appearance of individual points on a trace in a graph displayed in bar, marker, dot and lines to zero modes. To do this interactively, right click on the desired point on a trace and choose Customize at Point from the contextual menu. The Modify Trace dialog will appear with an entry in the trace list shown with the point number in square brackets. When such an entry is selected, only those properties that can be customized will be available in the dialog. This feature was added in Igor Pro 6.20.

Modifying Axes

You can modify the style of presentation of each axis in a graph by choosing Modify Axis from the Graph menu or by double-clicking directly on an axis. This brings up the Modify Axis dialog.

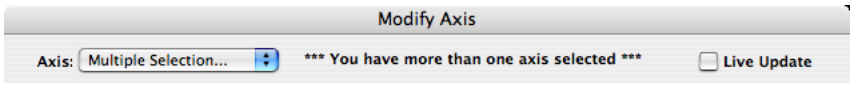
The precise appearance of the Modify Axis dialog depends on current axis settings such as whether the axis is linear, log, or a category axis and also on whether you specify tick mark positions manually or allow Igor to automatically choose them. This is what it looks like the first time you display the dialog, with a linear numeric axis and automatic ticking. The dialog remembers which tab you used last and displays that tab, so it may not look like the picture.



The dialog has tabs for various aspects of axis appearance, plus a few controls outside the tabs. These global controls include the standard Igor dialog controls: Do It, To Cmd Line, To Clip, Help and Cancel buttons, plus a box to display the commands generated by the dialog. At the top are the Axis menu and the Live Update checkbox.

The Axis pop-up menu shows all axes that are in use in the top graph. Choose the axis that you want to change from the Axis menu, or choose Multiple Selection if you want to affect more than one axis. Below the Multiple Selection item are items that provide shortcuts for selecting certain classes of axes.

A multiple selection can be convenient if you want to set something like the color or line width for all the axes in a graph the same way. When a multiple selection is active, the initial control settings reflect the settings of the first selected axis in the list. Commands are generated to change some aspect of *all* the selected axes any time a control is touched. This makes it quite easy to make sweeping changes, and rather difficult to return to the unchanged state. Consequently, when multiple axes are selected, the dialog puts a warning message at the top of the dialog:



Sometimes, with all the different things you can do to an axis, it is confusing to figure out exactly what you want. In that case, you can select the Live Update checkbox and watch the graph change is you change settings. In most cases graphs re-draw fast enough that this causes no problems. Consequently the Live Update checkbox is turned on by default. A graph with a very large number of points, or a contour plot can take quite a while to re-draw, causing annoying delays while you change settings. If it becomes annoying, simply turn off the Live Update checkbox.

You select the appropriate tab for the types of changes you want to make:

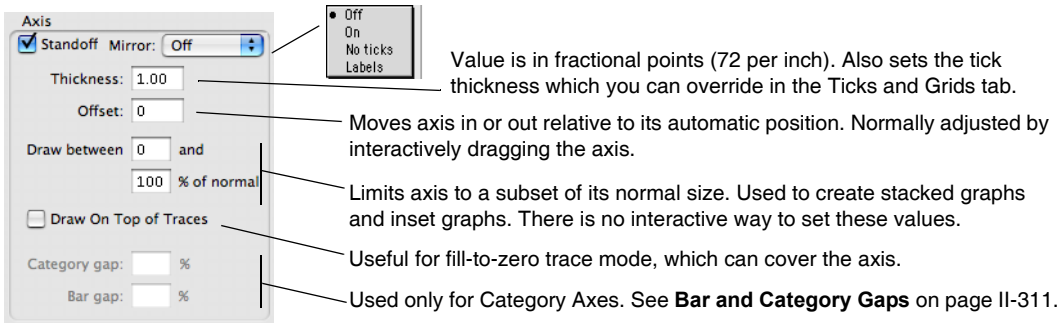
Tab	What It Does
Axis	Settings that affect the appearance of the axis itself. Color, line type, log or linear, etc. See **Axis Tab** on page II-264.
Auto/Man Ticks	Settings that affect how many ticks are drawn; select automatic or user-defined tick modes. See **Auto/Man Ticks Tab** on page II-266.
Ticks and Grids	Settings that affect the appearance of tick marks. Select a grid and its style. See **Ticks and Grids Tab** on page II-267.

Tab	What It Does
Tick Options	More obscure settings for tick appearance.
Axis Label	Create or change the axis label. See **Axis Labels** on page II-280.
Label Options	Settings that affect appearance of axis and tick mark labels, such as rotation and position relative to the axis. See **Label Options Tab** on page II-270.
Axis Range	Settings that affect the range of the axis. See **Setting the Range of an Axis** on page II-244.

Axis Tab

You can set the axis Mode for the selected axis to linear, log base 10, log base 2, or Date/Time. The Date/Time mode is special — when drawing an axis, Igor looks at the controlling wave's units to decide if it should be a date/time axis. Consequently, if you select Date/Time axis, the dialog immediately changes the units of the controlling wave. Because the dialog changes the wave's units, there is a small warning message next to the Date/Time radio button. See **Date/Time Axes** on page II-276 for details on how date/time axes work.

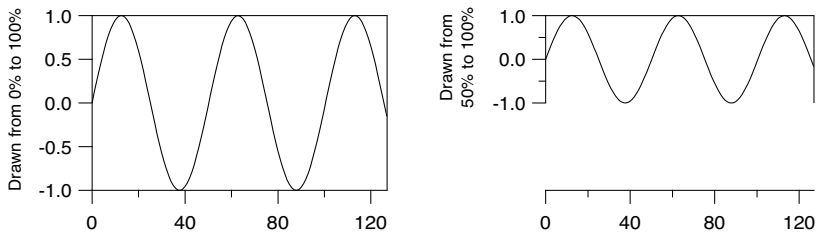
The Axis area of the Axis tab controls certain aspects of the axis layout:



Use the Mirror Axis pop-up menu to enable the mirror axis feature. A mirror axis is an axis that is the mirror image of the opposite axis. You can mirror the left axis to the right or the bottom axis to the top. The normal state is Off in which case there is no mirror axis. If you choose On from the pop-up, you get a mirror axis with tick marks but no tick mark labels. If you choose "No ticks", you get a mirror axis with no tick marks. If you choose Labels you get a mirror axis with tick marks and tick mark labels. Mirror axes may not do exactly what you want when using free axes, or when you shorten an axis using Draw Between. An embedded graph may be a better solution if free axes don't do what you need; see Chapter III-4, **Embedding and Subwindows**.

Free axes can also have mirror axes. Unlike the free axis itself, the mirror for a given free axis can not be moved — it is always attached to the opposite side of the plot area. This feature can create stacked plots; see **Creating Stacked Plots** on page II-293.

The "Draw between" items are used to create stacked graphs. You will usually leave these at 0 and 100%, which draws the axis along the entire length or width of the plot area. You could use 50% and 100% to draw the left axis over only the top half of the plot area (mirror axes are on in this example to indicate the plot area):



For additional examples of using "Draw between", see **Creating Stacked Plots** on page II-293 and **Creating Split Axes** on page II-297.

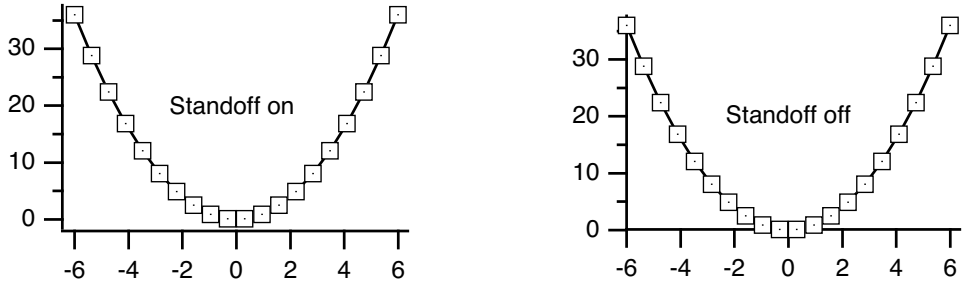
The Offset item is a way to control the distance between the edge of the graph and the axis. It specifies the distance from the default axis position to the actual axis position. This setting is in units of the size of a zero character in a tick mark label. Because of this, the axis offset adjusts reasonably well when you change the size of a graph window. The default axis offset is zero. You can restore the axis offset to zero by dragging the axis to or beyond the edge of the graph. If you enter a graph margin (see **Overall Graph Properties** on page II-245), the margin overrides the axis offset.

Normally you will adjust the axis offset by dragging the axis in the graph. If the mouse is over an axis, the cursor changes to a double-ended arrow indicating that you can drag the axis. If the axis is a mirror axis you will not be able to drag it and the cursor will not change to the double-ended arrow.

The Offset item does not affect a free axis. To adjust the position of a free axis, use the settings in the Free Axis Position area in the lower-left corner of the **Axis Tab** (see page II-264).

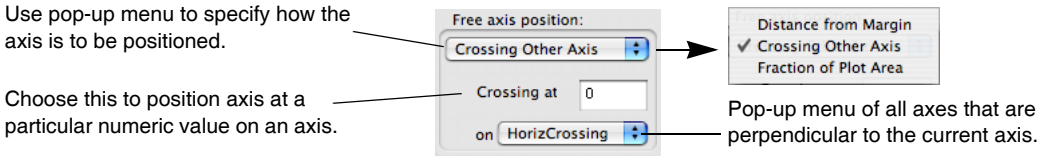
The Thickness item sets the thickness of an axis and associated tick marks in points. The thickness can be fractional and if you set it to zero the axis and ticks disappear.

The Standoff checkbox disables normal behavior with respect to offsetting axes. Normally Igor offsets axes so that waves do not cover them. For example, imagine that your Y axis goes from -1 to 0 and your waves have Y values equal to -1. These Y values would cover up the X axis if not for Igor's normal axis offset. This is especially noticeable when you use markers. If you don't like your axes to stand off, you can disable the standoff using the "Axis standoff" checkbox.



If a free axis is attached to the same edge of the plot rectangle as a normal axis then the standoff setting for the normal axis will be ignored. This is to make it easy to create stacked plots.

The Offset item applies only to the four standard axes (Bottom, Left, Top and Right); it does not apply to free axes. To change the position of free axes, use the controls in the Free axis position area:



The free position can be adjusted by dragging the axis interactively. This is the recommended way to adjust the position when using the absolute distance mode but it is not recommended when using the "crossing at" mode. This is because the crossing value as set interactively will not be exact. You should use this dialog to specify an exact crossing value.

Colors of axis components are controlled by items in the Color area. The axis label, tick mark labels and the axis line (including the tick marks) can be colored independently, but usually you will want them all to be the same color. The Set All to Axis Color button is a convenient way to do this — just choose a color in the Axis Line palette, then click the button. The color of text in the axis label can also be controlled by escape codes in the axis label text. See **Axis Labels** on page II-280. Tick marks can be assigned a color that is different than the axis line, but this is not supported by the dialog.

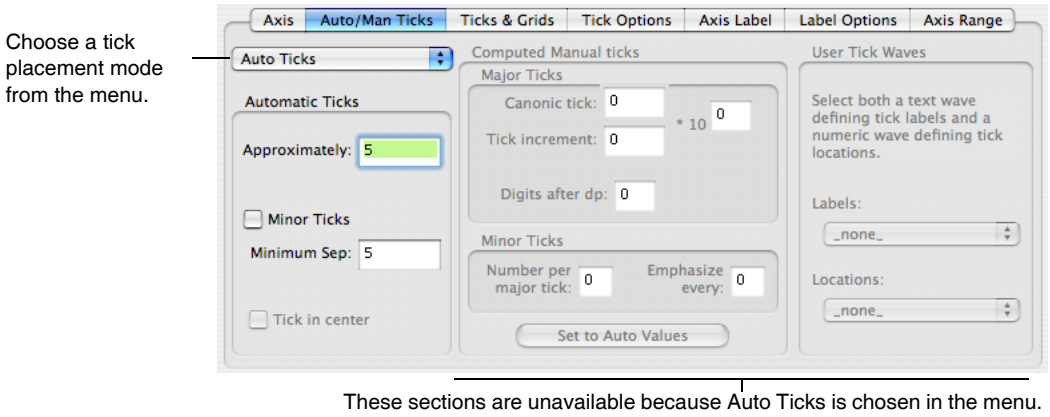
The Font section of the Axis tab specifies the font, font size, and typeface used for the tick labels and the axis label. You should leave this setting at "default" unless you want this particular axis to use a font different from

the rest of the graph. You can set the default font for all graphs using the Default Font item in the Misc menu. You can set the default font for this particular graph using the Modify Graph item in the Graph menu. The axis label font can be controlled by escape codes within the axis label text. See **Axis Labels** on page II-280.

Auto/Man Ticks Tab

The items in the Auto/Man Ticks tab control the placement of tick marks along the axis. You can choose one of three methods for controlling tick mark placement from the pop-up menu at the top of the tab. Choose Auto Ticks to compute nice tick mark intervals using some hints from you; choose Computed Manual Ticks to have complete control over the origin and interval for placing tick marks; use User Ticks from Waves to choose waves that give you complete control over tick mark placement and labelling.

Here is what the tab looks like with Auto Ticks chosen:



With Auto Ticks chosen, you can specify a *suggested* number of major ticks for the selected axis by entering that number in the Approximately parameter box. The actual number of ticks on the axis may vary from the suggested number because Igor juggles several factors, including the approximate ticks parameter, to get round number tick labels with reasonable spacing in a common numeric sequence (e.g., 1, 2, 5). In most cases, this automatically produces a correct and attractive graph. This item is not available if the selected axis is a log axis.

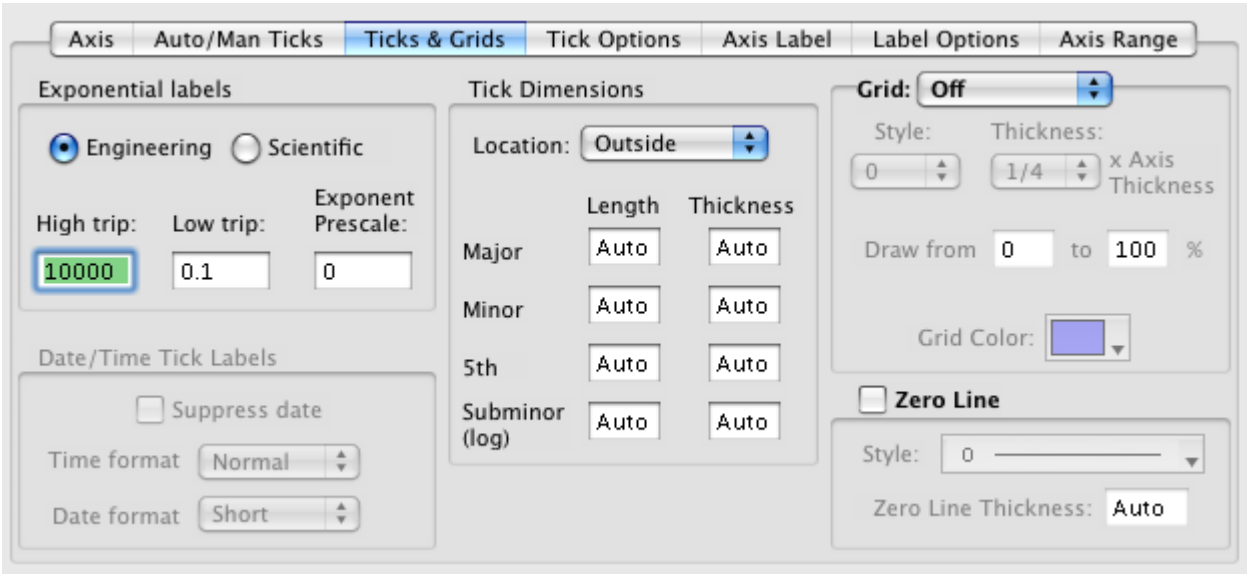
You can turn minor ticks on or off for the selected axis using the Minor Ticks checkbox.

The Minimum Sep setting controls the display of minor ticks if minor ticks are enabled. If the distance between minor ticks would be less than the specified minimum tick separation (measured in points) then Igor picks a less dense ticking scheme. For log axes Minor Ticks and Tick Separation affect the drawing of *subminor* ticks.

The **Manual Ticks** section on page 273 describes how you can completely override Igor's intelligent algorithms for tick placement.

Ticks and Grids Tab

The Ticks and Grids tab has items to control the appearance of ticks and to select and control graph grids:



Exponential Labels

When numbers that would be used to label tick marks become very large or very small, Igor switches to exponential notation. For example if the tick values for an axis are 0, 1000000000, 2000000000, 3000000000, and 4000000000 then Igor will break the values into a small number and an exponent. In this case Igor would choose 0,1,2,3 and 4 with the exponent being 10^9. Igor uses the small numbers to label the tick marks and leaves the exponent for your use in the axis label. The use of the exponent in the axis label is covered in the **Axis Labels** section on page 280. In the case of log axes, the tick marks include the exponent part.

With the Low trip and High trip settings, you can control the point at which tick mark labels switch from normal notation to exponential notation. If the absolute value of the larger end of the axis is between the low trip and the high trip, then normal notation is used. Otherwise, exponential is used. However, if the exponent would be zero, normal notation is always used.

There are actually two independent sets of low trip and high trip parameters: one for normal axes and one for log axes. The low trip point can be from 1e-38 to 1 and defaults to 0.1 for normal axes and to 1e-4 for log axes. The high trip point can be from 1 to 1e38 and defaults to 1e4.

Under some circumstances, Igor may not honor your setting of these trip points. If there is no room for normal tick mark labels, Igor will use exponential notation, even if you have requested normal notation.

The Engineering and Scientific radio buttons allow you to specify whether tick mark labels should use engineering or scientific notation when exponential notation is used. It does not affect log axes. Engineering mode is just exponential notation where the exponent is always a multiple of three.

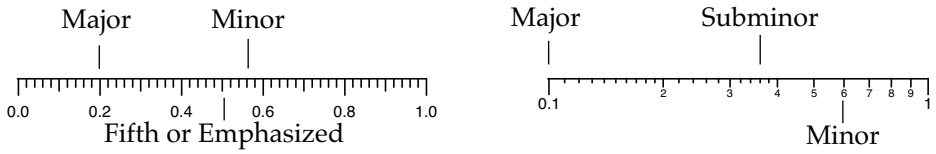
With the Exponent Prescale item, you can force the tick and axis label scaling to values different from what Igor would pick. For example, if you have data whose x scaling ranges from, say, 9pA to 120pA and you display this on a log axis, Igor will label the tick marks with 10pA and 100pA. But if you really want the tick marks labeled 10 and 100 with pA in the axis label, you can set the prescaleExp to 12. For details, see **Axis Labels** on page II-280.

Date/Time Tick Labels

The Date/Time Tick Labels area of the tab is explained in the **Date/Time Axes** section on page 276.

Tick Dimensions

You can control the length and thickness of each type of tick mark that Igor makes and the location of tick marks relative to the axis line using items in the Tick Dimensions area. Igor distinguishes four types of tick marks: major, minor, "fifth", and subminor:



Normal Axis **Log Axis**

The tick mark thicknesses normally follow the axis thickness. You can override the thickness of individual tick types by replacing the word "Auto" with your desired thickness specified in fractional points. A value of zero is equivalent to "Auto".

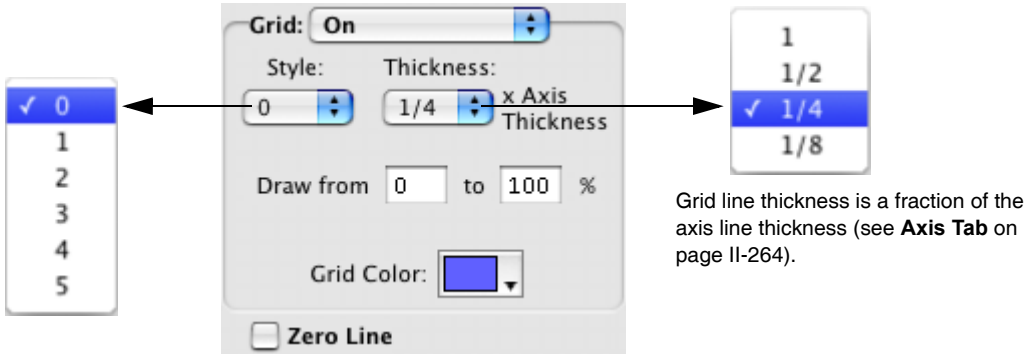
The tick length is normally calculated based on the font and font size that will be used to label the tick marks. You can enter your own values in fractional points. For example you might enter a value of 6 for the major tick mark, 3 for the minor tick mark and 4.5 for the 5th or emphasized minor tick marks. The subminor tick mark only applies to log axes.

Use the Location pop-up menu to specify that tick marks for the selected axis be outside the axis, crossing the axis or inside the axis or you can specify no tick marks for the axis at all.

Grid

Grid lines can be added to the graph using the Grid pop-up menu. choose Off if you do not want a grid, On if you want grid lines on major and minor tick marks or Major Only if you want grid lines on major tick marks only.

The default appearance of the major, minor, and subminor grid lines varies. Normally the default grids are sufficient, however Igor provides the ability to customize their appearance:



Grid line thickness is a fraction of the axis line thickness (see **Axis Tab** on page II-264).

Igor provides five grid styles identified with numbers 1 through 5. Different grid styles have major and minor grid lines that are light, heavy, dotted or solid. If the style is set to zero (the default) and the graph background is white then grid style 2 is used. If the graph background is not white then grid style 5 is used.

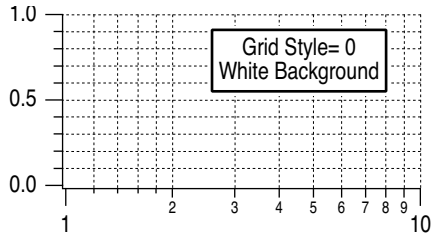
Use the Grid Color palette to set the color of the grid lines. They are by default light blue.

The grid line thickness is set to a fraction of the axis line thickness. Since the axis line thickness is usually one point, and computer monitors usually have a resolution of about a point, it generally is not possible to see the differences in thickness on your screen. To see the difference, print the graph — printer resolution is usually higher than screen resolution.

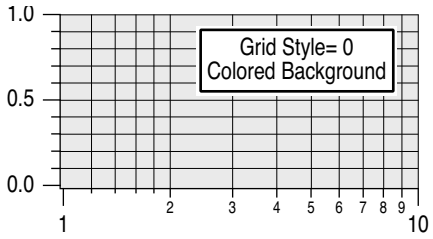
Sometimes the dotted lines in the grid disappear on color printers. This is because the colors are "dithered", that is, color shades are composed by printing several dots of varying colors, which are mixed in the eye.

The grid line dots may disappear if the dots are smaller than the dithering cell. The solution is to increase the grid line thickness, or to choose a color that can be printed without dithering, like black.

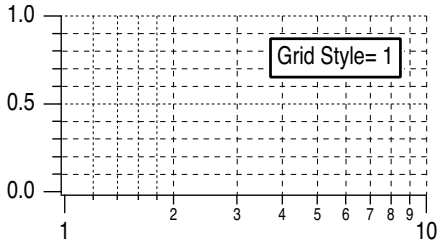
The examples here show graphs with thicker than normal axis lines and the thickest grid lines (thickness of 1 instead of the default 1/4).

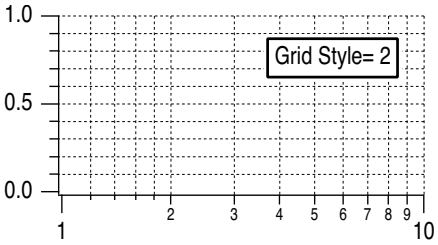


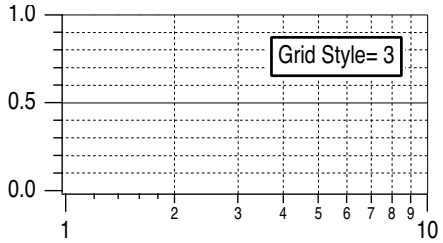
(same as gridStyle=2)

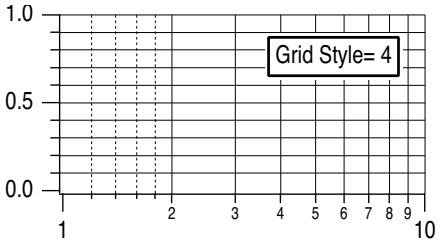


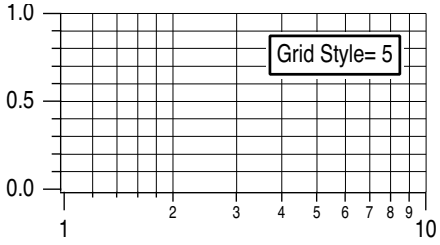
(same as gridStyle=5)







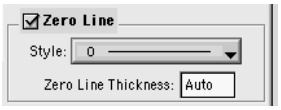




By default, the grid color is light blue, but you can change it using the Grid Color palette. You can also use the Axis pop-up menu or the Graph Background pop-up menu to avoid a trip to the Modify Axis dialog.

Using the settings "Draw from" and "to" you can restrict the length of the grid lines. This is useful if you have used the similar settings on the Axis tab to shorten one of your axes, and you want grid lines to match.

Zero Line



You can turn the Zero Line for the selected axis on or off by selecting or deselecting the Zero Line checkbox.

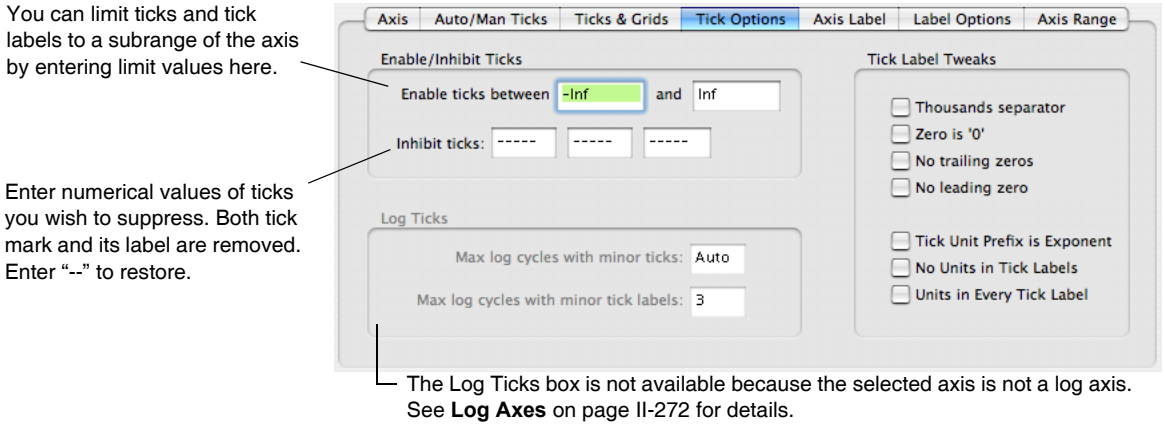
The zero line is a line perpendicular to the axis extending across the graph at the point where the value of the axis is zero. The Zero Line checkbox is not available for log axes.

If you turn the zero line on then you will be able to choose the line style from the Style pop-up menu. The thickness of the line can be set in fractional points from 0 to 5. The zero line has the same color as the axis, see the section **Axis Tab** on page II-264 for information on setting the axis line color.

The dashed line styles can be altered to your liking. See **Dashed Lines** on page III-410.

Tick Options Tab

The Tick Options tab has items to touch up certain aspects of the axes. There is a good chance that you will never need to visit this tab. Here is what it looks like for a normal axis:



Tick Label Tweaks Checkboxes

Checkbox	Result
Thousands separator	Tick labels like 10000 are drawn as 10,000.
Zero is '0'	Select this to force the zero tick mark to be drawn as 0 where it would ordinarily be drawn as 0.0 or 0.00.
No trailing zeroes	Tick labels that would normally be drawn as 1.50 or 2.00 are drawn as 1.5 or 2.
No leading zero	Select if you want tick labels such as 0.5 to be drawn as .5
Tick Unit Prefix is Exponent	If tick mark would have prefix and units (µTorr), force to exponential notation (10^{-6} Torr).
No Units in Tick Labels	If tick mark would have units, suppress them.
Units in Every Tick Label	If normal axis, force exponent or prefix and units into each label.

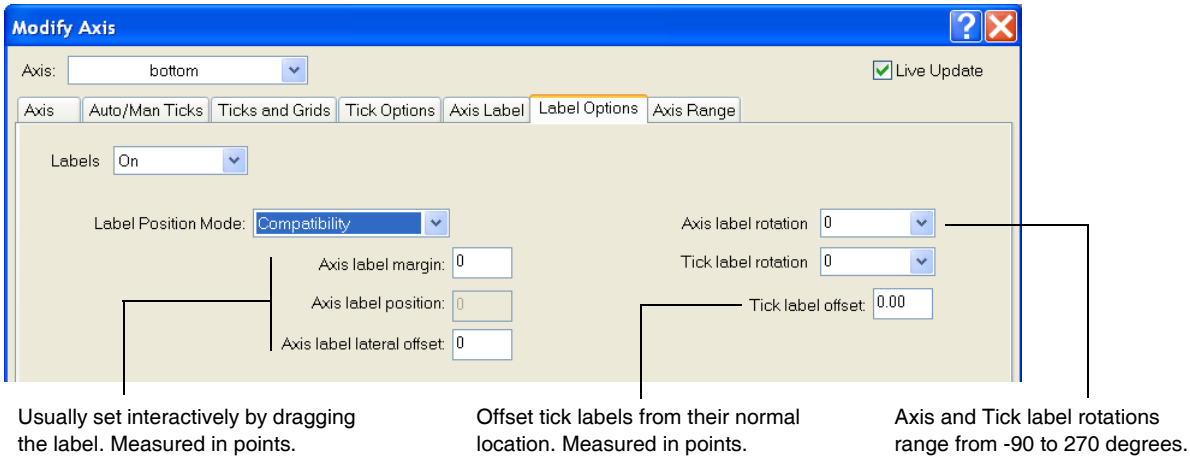
The last three of these checkboxes can be tricky — this is a situation where using Live Update can be very helpful, allowing you to see what happens as you select and deselect items.

Axis Label Tab

This is discussed in detail in **Axis Labels** on page II-280.

Label Options Tab

The Label Options tab has items to control the placement and orientation of axis and tick mark labels. You can also hide these labels completely. The tab looks like this:



Normally, you will adjust the position of the axis label by simply dragging it around on the graph. The "Axis label position" or "Axis label margin" and the "Axis label lateral offset" settings are only used when you want precise numerical control over the position.

The calculations used to position the axis label depend on the setting in the Label Position Mode menu. By default this is set to Compatibility, which will work with older versions of Igor. The other modes may allow you to line up labels on multiple axes more accurately. The choice of positioning mode affects the meaning of the three settings below the menu.

In Compatibility mode, the method Igor uses to position the axis label depends on whether or not a free axis is attached to the given plot rectangle edge. If no free axis is attached then the label position is measured from the corresponding window edge; we call this the axis label margin. Thus if you reposition an axis the axis label will not move. On the other hand, if a free axis is attached to the given plot rectangle edge then the label position is measured from the axis and when you move the axis, the label will move with it.

Because the method used to set the axis label varies depending on circumstances, one or the other of the Axis label margin or Axis label position boxes may be unavailable. If you have selected an axis on the same edge as a free axis, the Axis label position box is made available. If you have selected an axis that does not share an edge with a free axis, the Axis label margin box is made available. If you have selected multiple axes it is possible for both items to be available.

The Axis label position is the distance from the axis to the label and is measured in points.

The Axis label margin is the distance from the edge of the graph to the label and is measured in points. The default label margin is zero which butts the axis label up against the edge of the graph.

The Margin modes measure relative to an edge of the graph while the axis modes measure relative to the position of the axis. Using an Axis mode will cause the label to follow a free axis when you move the axis. The Margin modes are useful for aligning labels on stacked graphs. The Axis label margin setting applies to Margin modes; the Axis label position setting applies to Axis modes.

The Absolute modes measure distance in points. Scaled modes have similar numerical values but are scaled to respond to changes in the font size.

The Labels pop-up contains On, Axis Only and Off items. On gives normal axis labeling. Axis Only leaves the axis label in place but removes the tick mark labels. Off removes the axis labels and tick mark labels. This can be useful when you print multiple graphs on a page and want to give the impression that they share an axis.

Axis and Tick label rotations can be set to any value between -360 and 360 degrees.

Note: (*Windows* only) Sometimes you may find that a rotated axis label contains characters that fail to rotate. This is usually caused by the use of a bitmapped font. On Windows, only TrueType fonts can be

rotated. Sometimes there are multiple versions of a given font and one of them is a bitmapped font. The solution to this problem is to remove the bitmapped font from the Windows Fonts folder.

This problem frequently afflicts the Symbol font. This is due to the fact that Microsoft ships a screen font version of Symbol font. You can fix this by removing the screen font. If you find something like this in your Fonts folder:

Symbol

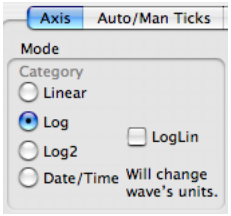
Symbol 8,10,12,14,18,24

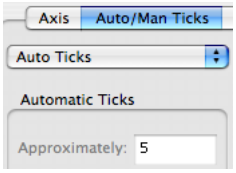
The first is a TrueType font. The second is a screen font. Removing the second fixes the problem. Another workaround is to use a font size other than 8, 10, 12, 14, 18, 24, such as 9 points.

Log Axes

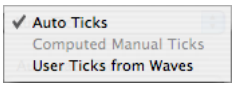
Certain items in the Modify Axis dialog are not available when a log axis is selected:

When a log axis is selected...

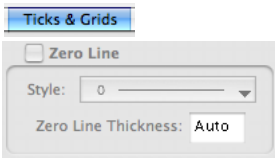




...Igor sets the number of ticks...



... you can't use computed manual ticks...



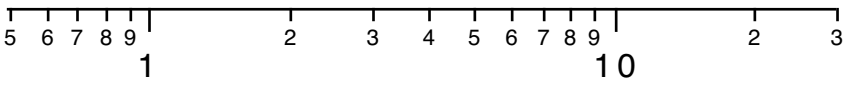
... and the zero line is not available.

Igor has three ways of ticking a log axis that are used depending on the range (number of decades) of the axis: normal, small range and large range. The normal mode is used when the number of decades lies between about one third to about ten (the exact upper limit depends on the physical size of the axis and the font size).

If the number of decades of range is less than two or greater than five, you can force Igor to use the small/large range methods by selecting the LogLin checkbox, which may give better results for log axes with small or very large range.

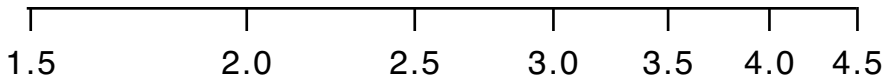
When you do this, all of the settings of a linear axis are enabled including manual ticking.

Here is a normal log axis with a range of 0.5 to 30:

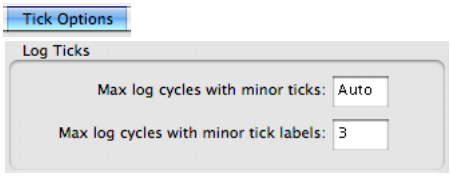


If we zoom into a range of 1.5 to 4.5 we get this:

But if we then select the LogLin checkbox, we get better results:



Selecting a log axis makes the Log Ticks box on the Tick Options tab available:

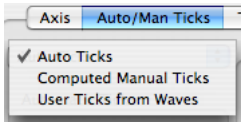


The "Max log cycles with minor ticks" setting controls whether minor ticks appear on a log axis. This setting can range from 0 to 20 and defaults to 0. If it is 0 or "auto", Igor automatically determines if minor ticks are appropriate. Otherwise, if the axis has more cycles (decades) than this number then the minor ticks are not displayed. Minor ticks are also not displayed if there is not enough room for them.

Similarly, you can control when Igor puts labels on the minor ticks of a log axis using the Max log cycles with minor tick labels item. This is a number from 0 to 8; 0 disables the minor tick labels. As long as the axis has fewer decades than this setting, the minor ticks are labeled.

Manual Ticks

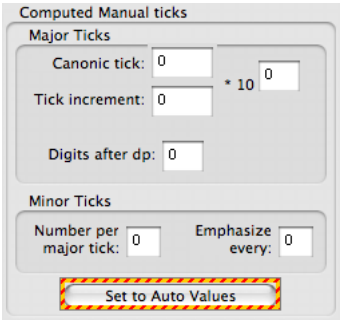
If Igor's automatic selection of ticks does not suit you, and you can't find any adjustments that make the tick marks just the way you want them, Igor provides two methods for specifying the tick marks yourself. On the Auto/Man Ticks tab of the Modify Axis dialog, you can choose either Computed Manual Ticks or User Ticks from Waves.



Computed Manual Ticks

Use Computed Manual Ticks to enter a numeric specification of the increment between tick marks and the starting point for calculating where the tick marks fall. This style of manual ticking is available for normal axes (including log axes when the loglin checkbox is selected on the Axis tab) and date/time axes only. User Ticks from Waves can be used to select waves that you have created that completely specify the ticking. This option is available for normal axes or log axes.

When you choose Computed Manual Ticks the corresponding area of the Auto/Man Ticks tab becomes available.

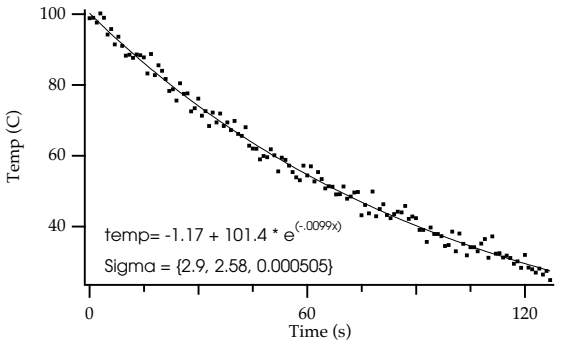


If you click the "Set to auto values" button, Igor sets all of the items in the Compute Manual Ticks box to the values they would have if you let Igor automatically determine the ticking. This is usually the starting point.

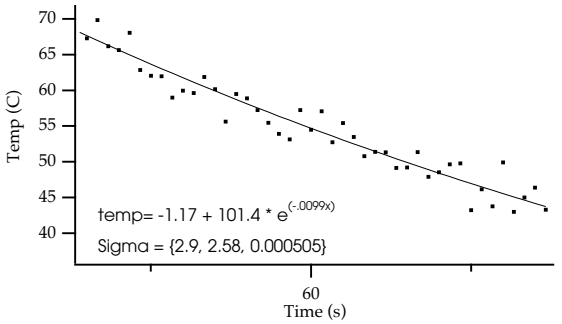
Using the "Canonic tick" setting, you specify the value of any major tick mark on the axis. Using the "Tick increment" setting, you specify the number of axis units per major tick mark. Both of these numbers are specified as a mantissa and an exponent. The canonic tick is not necessarily the first major tick on the axis. Rather, it is a major tick on an infinitely long axis of which the axis in the graph is a subset. That is, it can be *any* major tick. whether it shows on the graph or not.

Note: When you use computed manual ticks on a large range logarithmic axis, the values in the dialog refer to the exponent of the tick value.)

For example, imagine that you want to show the temperature of an object as it cools off. You want to show time in seconds but you want it to be clear where the integral minutes fall on the axis. You would turn on manual ticking for the bottom axis and set the canonic tick to zero and the tick increment to 60. You could show the half and quarter minute points by specifying three minor ticks per major tick (Number per major tick ins the Minor Ticks box) with every second minor tick emphasized (Emphasize every). This produces the following graph:



Now, imagine that you want to zoom in on t = 60 seconds.



The canonic tick, at t = 0, does not appear on the graph but it still controls major tick locations.

User Ticks from Waves

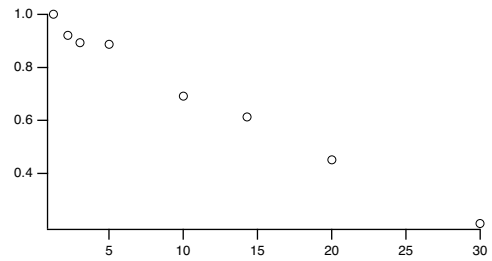
With Computed Manual Ticks you have complete control over ticking as long as you want equally-spaced ticks. If you want to specify your own ticking on a log axis, or you want ticks that are not equally spaced, you need User Ticks from Waves.

The first step in setting up User Ticks from Waves is to create two waves: a 1D numeric wave and a text wave. Numbers entered in the numeric wave specify the positions of the tick marks in axis units. The corresponding rows of the text wave give the labels for the tick marks.

Perhaps you want to plot data as a function of Tm/T (melting temperature over temperature, but you want the tick labels to be at nice values of temperature. Starting with this data:

Point	InverseTemp	Mobility
0	30	0.211521
1	20	0.451599
2	14.2857	0.612956
3	10	0.691259
4	5	0.886406
5	3.0303	0.893136
6	2.22222	0.921083
7	1.25	1

you might have this graph:



Create the waves for labelling the axes:

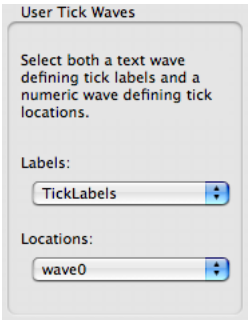
```
Make/N=5 TickPositions
Make/N=5/T TickLabels
```

Assuming that Tm is 450 degrees and that you have determined that tick marks at 20, 30, 50, 100, and 400 degrees would look good, you would enter these numbers in the text wave, TickLabels. At this point, a convenient way to enter the tick positions in the numeric wave, TickPositions is a wave assignment that embodies the relationship you think is appropriate:

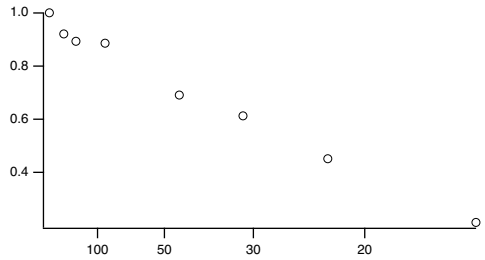
```
TickPositions = 450/str2num(TickLabels)
```

Note that the str2num function was used to interpret the text in the label wave as numeric data. This only works, of course, if the text includes only numbers.

Finally, double-click the bottom axis to bring up the Modify Axis dialog, select the Auto/Man Ticks tab and select User Ticks from Waves. Choose the TickPositions and TickLabels waves:

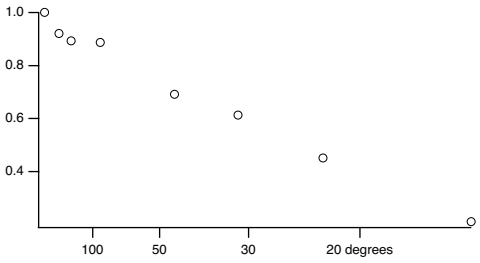


The result is this graph:



Note that you can add other text to the labels, including special marks. For instance:

TickLabels.d	TickPositions
20 degrees	22.5
30	15
50	9
100	4.5
400	1.125



Finally, you can add a column to the text wave and add minor, subminor and emphasized ticks by entering appropriate keywords in the other column. To add a column to a wave, select Redimension Waves from the Data menu, select your text wave in the list and click the yellow arrow. Then change the number of columns from 0 to 2 (or more).

This extra column must have the column label “Tick Types”. For instance:

TickLabels[][0].d	TickLabels[][1].d	TickPositions
	Tick Type	
20 degrees	Major	22.5
30	Major	15
50	Major	9
100	Major	4.5
400	Major	1.125
	Minor	21.4286
	Minor	20.4545
	Minor	19.5652
	Minor	18.75
	Emphasized	18
	Minor	17.3077
	Minor	16.6667
	Minor	16.0714
	Minor	15.5172

Dimension label "Tick Type" has keywords to set tick types

Blank entries make ticks with no labels.

Use keyword "Subminor" for subminor ticks such as Igor uses on log axes.

Dimension labels allow you (or Igor) to refer to a row or column of a wave using a name rather than a number. Thus, the Tick Type column doesn't have to be the second column (that is, column 1). For instructions on showing dimension labels in a table, see **Showing Dimension Labels** on page II-191.

Date/Time Axes

In addition to numeric axes, Igor supports axes labeled with dates, times or dates and times.

Dates and date/times are represented in Igor as the number of seconds since midnight, January 1, 1904. There is no practical limit to the range of dates that can be represented except on Windows where dates must be greater than January 1, 1601. Prior to Igor Pro 6.1 Igor supported dates in the range 1904 to 2040 only.

In Igor, a date can not be accurately represented in a single precision wave. Make sure you use double precision waves to store dates and date/times. (A single precision wave can provide dates and date/times calculated from its X scaling, but not from its data values.)

Times without dates can be thought of in two ways: as time-of-day times and as elapsed times.

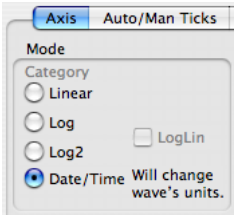
Time-of-day times are represented in Igor as the number of seconds since midnight.

Elapsed times are represented as a number of seconds in the range -9999:59:59 to +9999:59:59. For integral numbers of seconds, this range of elapsed times can be precisely represented in a signed 32-bit integer wave. A single-precision floating point wave can precisely represent integral-second elapsed times up to about +/-4600 hours.

Igor displays dates or times on an axis if the appropriate units for the wave controlling the axis is "dat". This is case-sensitive — "Dat" won't work. You can set the wave's units using the Change Wave Scaling item in the Data menu, or the SetScale operation.

To make a horizontal axis be a date or time axis for a waveform graph, you must set the X units of the wave controlling the axis to "dat". For an XY graph you must set the data units of the wave supplying the X coordinates for the curve to "dat". To make the vertical axis a date or time axis in either type of graph, you must set the data units of the wave controlling the axis to "dat". (If you're not sure what a "waveform graph" is, see **Creating Graphs** on page II-237. If you are mystified by "wave controlling the axis", see **Waves and Axes** on page II-239.)

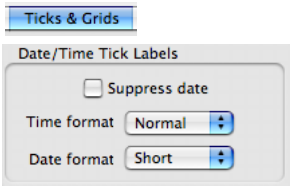
It is much easier to let the Modify Axis dialog change the wave scaling for you.



When you click the Date/Time radio button, Igor tracks down the controlling wave for the axis and sets the appropriate units to "dat".

For Igor to use a date or time axis, the following additional restrictions must be met: the axis must span at least 2 seconds and both ends must be within the legal range for a date/time value. If any of these restrictions is not met, Igor displays a single tick mark.

When an axis is in the date/time mode, the Date/Time Tick Labels box in the Ticks&Grids tab of the Modify Axis dialog is available.



From the Time Format pop-up menu, you can choose Normal, Military, or Elapsed. Use Normal or Military for time-of-day times and Elapsed for elapsed times. In normal mode, the minute before midnight is displayed as 11:59:00 PM and midnight is displayed as 12:00:00 AM. In military mode, they are displayed as 23:59:00 and 00:00:00.

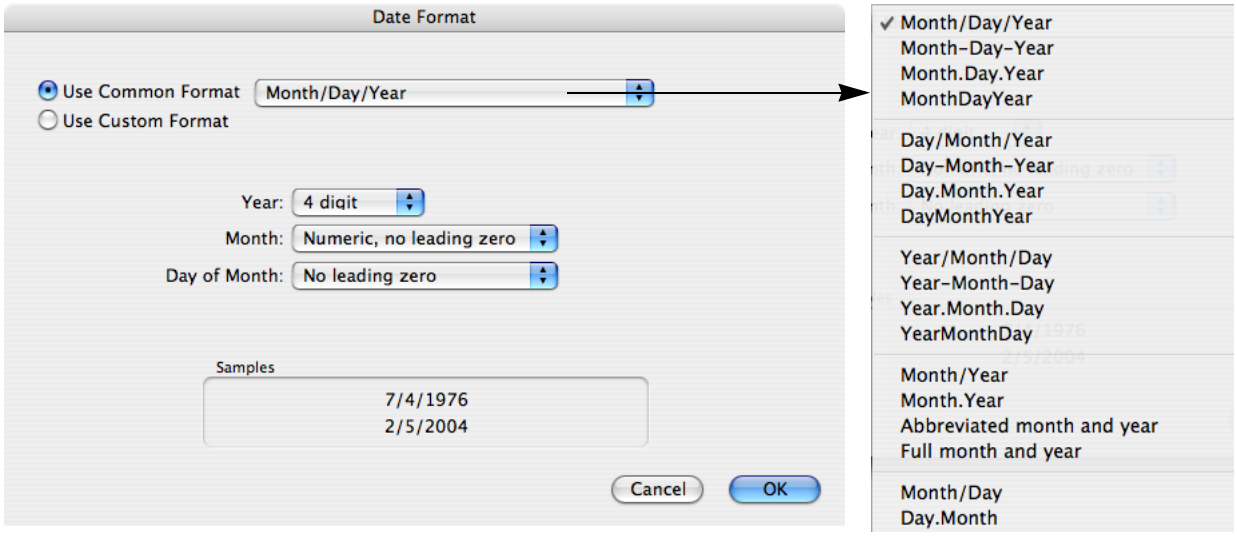
Elapsed mode can display times from -9999:59:59 to +9999:59:59. This mode makes sense if the values displayed on the axis are actually elapsed times (e.g., 23:59:00). It makes no sense and will display no tick labels if the values are actually date/times (e.g., 7/28/93 23:59:00).

The Date Format pop-up menu is relevant when you are displaying dates or date/times. It has no significance for times. From the Date Format pop-up, you can choose Short, Long, Abbrev, or Other. In the short mode, today's date is displayed in month/day/year format, such as 7/28/93. In the long mode, it is displayed as Wednesday, July 29, 1993. In the abbrev mode, it is displayed as Wed, Jul 28, 1993.

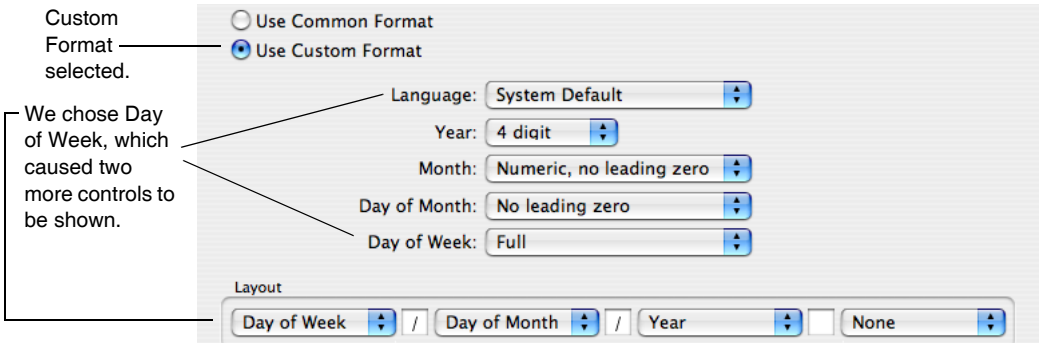
For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-102.

Custom Date Formats

If you choose Other from the Date Format pop-up, a dialog is displayed giving you almost complete control over the format of the tick labels. The dialog allows a broad range of automatic formats:



It also allows extensive control over custom formats (this is not quite the default appearance; we have changed some selections to show a larger range of possibilities):



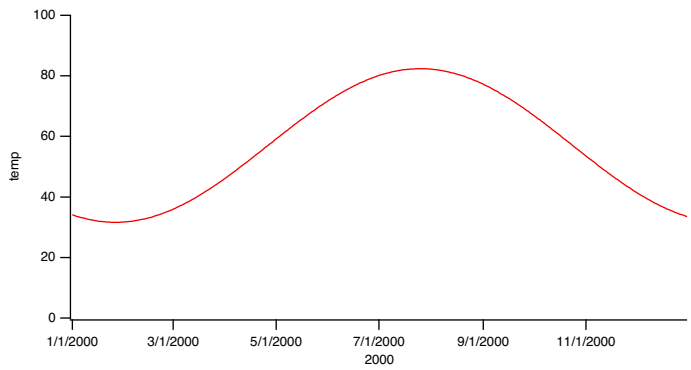
Depending on the extent of the axis, the tick mark labels may show date or date and time. You can suppress the display of the date when both the date and time are showing by selecting the Suppress Date checkbox. This checkbox is irrelevant when you choose the elapsed time mode in which dates are never displayed.

Use the Axis Range tab of the Modify Axis dialog to enter axis range values in terms of seconds, times, dates or date/times. When you enter the dialog, Igor automatically chooses a format appropriate for the axis you are working on.

Date/Time Examples

The following example shows how you can create a date/time graph of a waveform whose Y values are temperature and whose X values, as set via the SetScale operation, are dates:

```
// Make a wave to contain temperatures for the year
Make /N=365 temperature                // single precision data values

// Set its scaling so X values are dates
Variable t0, t1
t0 = Date2Secs(2000,1,1); t1 = Date2Secs(2001,1,1)
SetScale x t0, t1, "dat", temperature  // double-precision X scaling

// Enter the temperature data in the wave's Y values
t0 = Date2Secs(2000,1,1); t1 = Date2Secs(2000,3,31)   // winter
temperature(t0, t1) = 32                              // it's cold
t0 = Date2Secs(2000,4,1); t1 = Date2Secs(2000,6,30)   // spring
temperature(t0, t1) = 65                              // it's nice
t0 = Date2Secs(2000,7,1); t1 = Date2Secs(2000,9,31)   // summer
temperature(t0, t1) = 85                              // it's hot
t0 = Date2Secs(2000,10,1); t1 = Date2Secs(2000,12,31) // fall
temperature(t0, t1) = 45                              // cold again

// Smooth the data out
CurveFit sin temperature
temperature= K0+K1*sin(K2*x+K3)

// Graph the wave
Display temperature
SetAxis left, 0, 100;Label left "temp"
Label bottom "2000"
```



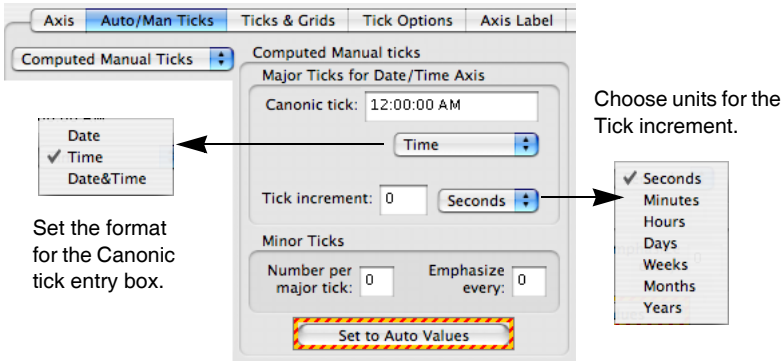
The SetScale operation sets the temperature wave so that its X values span the year 1989. In this example, the date/time information is in the X *values* of the wave. X values are always double precision. The wave itself is not declared double precision because we are storing temperature information, not date/time information in the Y values.

Manual Ticks for Date/Time Axes

Just as with regular axes, there are times when Igor's automatic choices of ticks for date/time axes simply are not what you want. For these cases, you can use computed manual ticks with date/time axes.

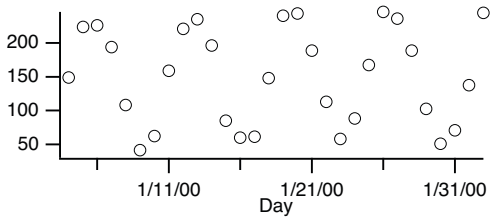
To use computed manual ticks, display the Modify Axis dialog by double-clicking the axis, or by choosing Modify Axis from the Graph menu. Select the Auto/Man Ticks tab, and choose Computed Manual Ticks from the menu in that tab. This much is just like computed manual ticks for regular axes (see **Manual Ticks** on page II-273).

The Computed Manual Ticks box is somewhat different for a date/time axis:

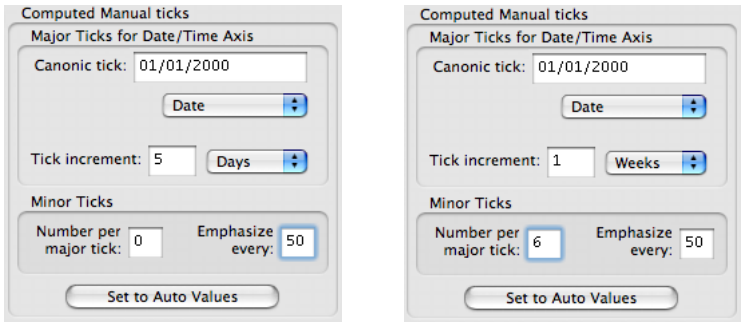


The first step is to click the Set to Auto Values button. Choose whether you need to enter a just the date, a date and time or just the time for the canonic tick. This will depend on the range of the data. Choose the units of the tick increment and the increment.

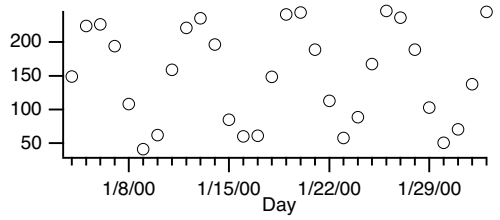
As an example, you might have data acquired over a period of some months showing data that have a strong weekly variation. The automatic date/time axis never chooses weeks as the basis for the tick increment, so you will need to use manual ticks. Here is the graph that we start with (we fixed it up a bit — one major change was to set the date format for the tick labels to show only two characters for the year, see **Custom Date Formats** on page II-277):



And the Computed Manual Ticks box after clicking the Set to Auto Values button, and after setting it up for weekly ticks with daily minor ticks:



The result:



"Fake" Axes

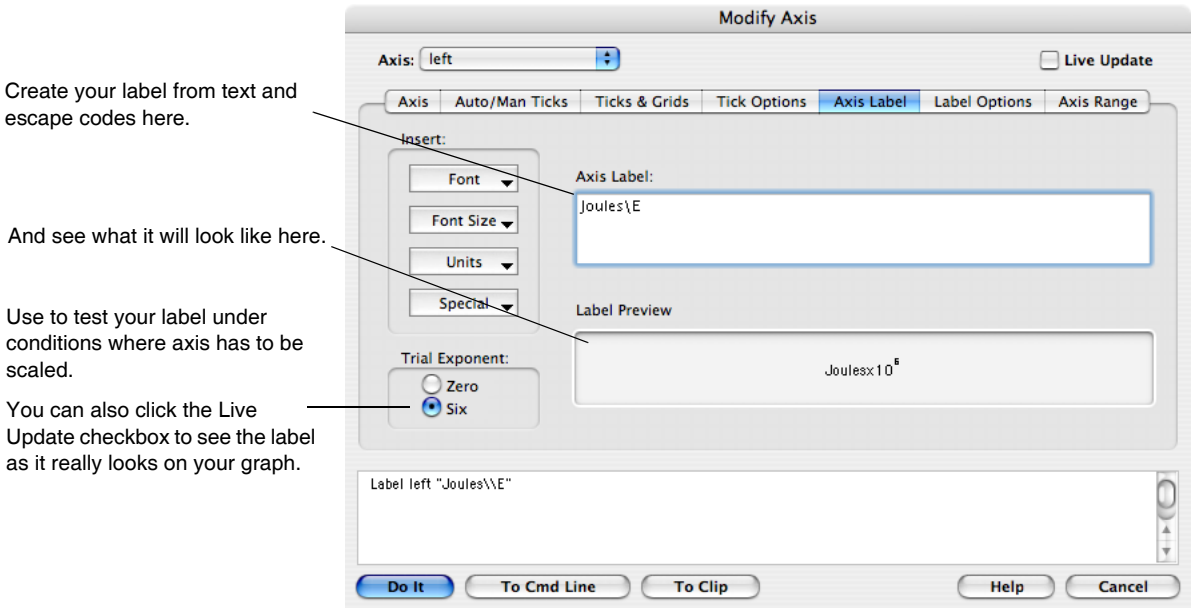
It is sometimes necessary to create an axis that is not related to the data in a simple way. One method uses free axes that are not associated with a wave (see the operations NewFreeAxis, ModifyFreeAxis and KillFreeAxis). The Transform Axis package uses this technique to make a mirror axis reflecting a different view of the data. An example would be a mirror axis showing wave number to go with a main axis showing wavelength. A demonstration is available from the File menu, Example Experiments→Graphing Techniques→Transform Axis Demo.

Another technique is to use Igor's drawing tools to create fake axes. The Polar Graph package is an example of a this technique (for an example, see in the File menu, Example Experiments→Graphing Techniques→New Polar Graph Demo). Another example is in Igor Technical Notes TN021: Ternary Graphs.

Axis Labels

The text for an axis label in a graph can come from one of two places. If you specify units for the wave which controls an axis (using the Change Wave Scaling dialog), Igor will use these units to label the axis. You can override this labeling by explicitly entering axis label text using the Axis Label tab of the Modify Axis dialog. The text that you explicitly enter can, in turn, contain an escape sequence to refer to the wave's units.

You can explicitly specify an axis label using the Axis Label tab in the Modify Axis dialog. You can display the dialog with the Axis Label tab showing by choosing Label Axis from the Graph menu or by double-clicking an axis label. If there is no axis label you must use the menu. This brings up the Modify Axis dialog with the Axis Label tab displayed.

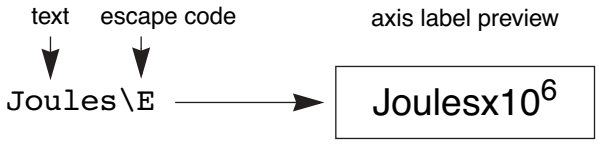


After you set the label for a particular axis you can select another axis and set its label. Further label formatting options are available on the Label Options tab described on page II-270.

There are two parts to an axis label: the text for the label and the special effects such as font, font size, superscript or subscript. You specify the text by typing in the text box. At any point in entering the text you can choose a special effect from a pop-up menu in the Insert box.

The Label Preview box shows what the axis label will look like, taking the text and special effects into account. You can not enter text in this box. You can also see your label on the graph if you click the Live Update checkbox.

When you choose a special effect, Igor inserts an **escape code** in the text. An escape code consists of a backslash character followed by one or more characters. It represents the special effect you chose. The escape codes are cryptic but you can see their effects in the Label Preview box.

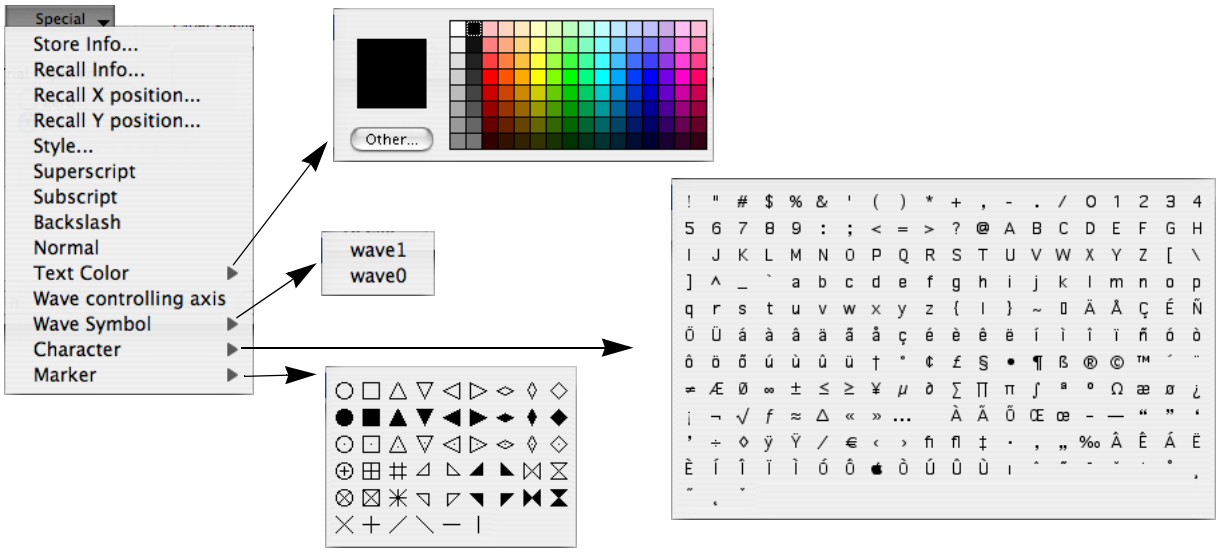


Select the axis that you want to label from the menu of axes and then type the text for the axis label in the textbox. You can insert special affects at any point in the text by clicking at that point and selecting the special effect from the Insert pop-ups.

Choosing an item from the Font pop-up menu inserts a code that changes the font for subsequent characters in the label. If you specify no font, Igor uses whatever font is in effect for the given axis. See **Axis Tab** on page II-264. The font pop-up also has a "Recall font" item. This item is used to make elaborate axis labels. See **Elaborate Annotations and Axis Labels** on page III-63.

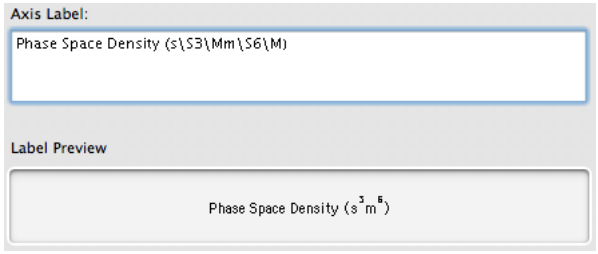
Choosing an item from the Font Size pop-up menu inserts a code that changes the font size for subsequent characters in the label. If you specify no font size, Igor uses the font size specified for the current axis. See **Axis Tab** on page II-264. The font size pop-up also has a "Recall size" item. This item is used to make elaborate axis labels. See **Elaborate Annotations and Axis Labels** on page III-63.

The **Special** pop-up menu is shown here:



The most commonly used items are Superscript, Subscript and Normal. To create a superscript or subscript, use the Special pop-up menu to insert the desired code, type the text of the super- or subscript and then finish with the Normal code. For example, suppose you want to create an axis label that reads "Phase space density (s^3m^{-6})". To do this, type "Phase space density (s", choose the Superscript item from the Special pop-up menu, type "3", choose Normal, type "m", choose Superscript, type "-6", choose Normal and then type ")".

Here is the label text and preview from the dialog:

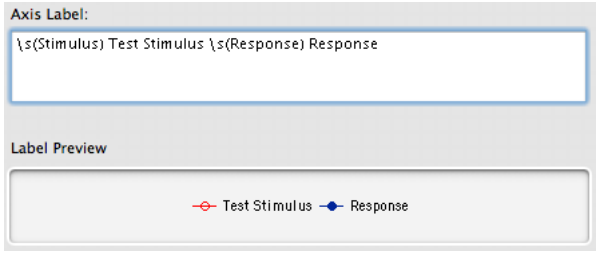


See Chapter III-2, **Annotations**, for a complete discussion of these items.

The Text Color hierarchical menu inserts a code that sets the color of the label's text.

The "Wave controlling axis" item inserts a code that prints the name of the first wave plotted on the given axis. You might use this in conjunction with preferences or style macros to ensure that new graphs are automatically labeled with the data names.

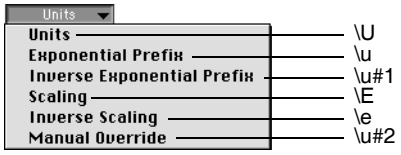
The Wave Symbol hierarchical menu inserts a code that draws the symbol used to plot the selected trace. You might use this to create an axis label that is a one-line legend:



The Character hierarchical menu presents a table from which you can select text and special characters to add to the axis label.

The Marker hierarchical menu inserts a code to draw a marker symbol. These symbols are independent of any traces in the graph.

The items in the Units pop-up menu insert escape codes as shown here:



These codes allow you to create an axis label that automatically changes when the extent of the axis changes.

For example, if you specified units for the controlling wave of an axis you can make those units appear in the axis label by choosing the Units item from the Units pop-up menu. If appropriate Igor will automatically add a prefix (μ for micro, m for milli, etc.) to the label and will change the prefix appropriately if the extent of the axis changes. The extent of the axis changes when you explicitly set the axis or when it is autoscaled.

If you are plotting a waveform (wave's Y values plotted versus its X values) then the units for the Y axis come from the wave's data units and the units for the X axis come from the wave's X units. However, if you are plotting an XY pair (Y values of one wave plotted versus Y values of another wave) then the units for the Y axis come from the data units of one wave and the units for the X axis come from the data units of the other wave.

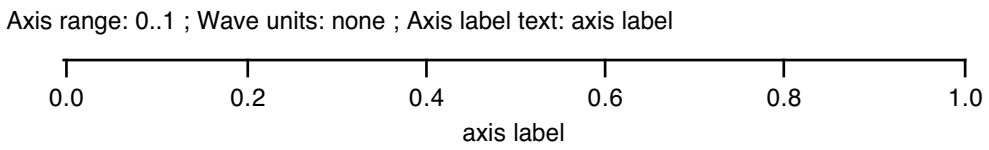
If you choose the Scaling or Inverse Scaling items from the Units pop-up menu, Igor will automatically add a power of 10 scaling (x10^3, x10^6, etc.) to the axis label if appropriate and will change this scaling if the extent of the axis changes. The Trial Exponent buttons determine what power is used *only* in the label preview so you can see what your label will look like under varying axis scaling conditions. Both of these techniques can be ambiguous — it is never clear if the axis has been multiplied by the scale factor or if the units contain the scale factor.

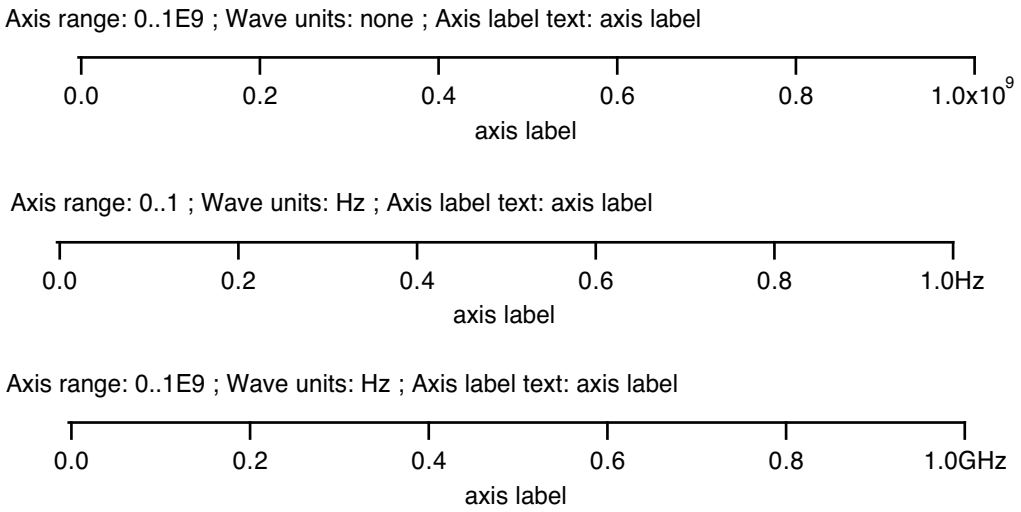
A less ambiguous method is to use the Exponential Prefix escape code. This is identical to the Scaling code except the "x" is missing. You can then use it in a context where it is clear that it is a multiplier of units. For example, if your axis range is 0 to 3E9 in units of cm/s, typing "Speed, \ucm/s" would create "Speed, 10^9cm/s".

It is common to parenthesize scaling information in an axis label. For example the label might say "Joules ($x10^6$)". You can do this by simply putting parentheses around the Scaling or Inverse Scaling escape codes. If the scaling for the axis turns out to be $x10^0$ Igor omits it and also omits the parentheses so you get "Joules" instead of "Joules ($x10^0$)" or "Joules()".

If you do not specify scaling but the range of the axis requires it, Igor labels one of the tick marks on the axis to indicate the axis scaling. This is an emergency measure to prevent the graph from being misleading. You can prevent this from happening by inserting the Manual Override escape code, \u#2, into your label. No scaling or units information will be printed at the location of the escape code (or on the tick marks). You will need to provide your own units or scaling by creating an annotation (Chapter III-2, **Annotations**) or a simple text object (Chapter III-3, **Drawing**).

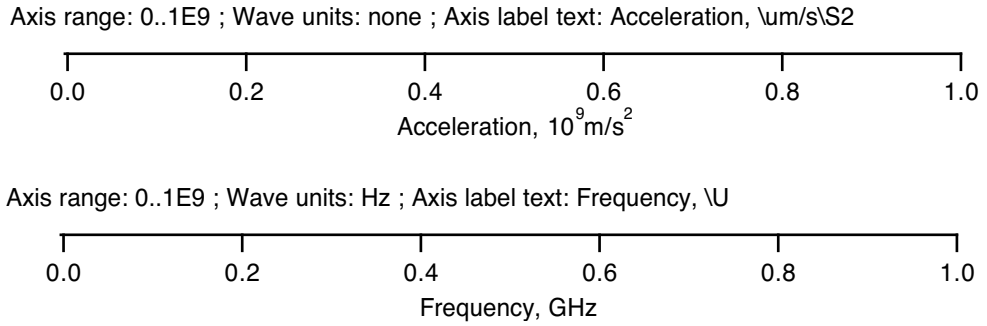
The following four examples illustrate what happens when the axis label *does not* contain any scaling or units escape codes:





In the last three cases, Igor has added units or scaling to the last tick mark label because the axis label itself does not contain the units and scaling information. You can suppress this by choosing Manual Override from the Units pop-up menu.

The following two examples show what happens when the axis label *does* contain scaling and units escape codes. ("\S" is the superscript escape code):

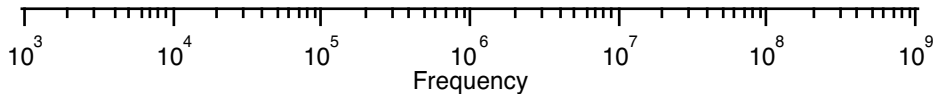


Here is a table showing how the escape codes react to different conditions:

Axis Range	Wave Units	Units \U	Exponential Prefix \u	Inverse Exponential Prefix \u#1	Scaling \E	Inverse Scaling \e
0…1	none					
0…1E9	none	$x10^9$	10^9	10^{-9}	$x10^9$	$x10^{-9}$
0…1	Hz	Hz	Hz	Hz		
0…1E9	Hz	GHz	10^9Hz	10^{-9}Hz	$x10^9$	$x10^{-9}$

The situation with log axes is a bit different. By their nature, log axes never have to be scaled and units/scaling escape codes are not used in axis labels. If the controlling wave for a log axis has units then Igor automatically uses the units along with the appropriate prefix for each major tick mark label. Here are two log axes; one where the controlling wave does not have units and one where it does:

Log Axis range: 1E3..1E9 ; Wave units: none ; Axis label text: Frequency



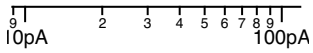
Log Axis range: 1E3..1E9 ; Wave units: Hz ; Axis label text: Frequency

Yet another variation on labeling and units is provided by Exponent Prescale, which is found on the Ticks and Grids tab of the Modify Axis dialog. Using Exponent Prescale you can force the tick and axis label scaling to values different from what Igor would pick. For example, if you have data whose x scaling ranges from, say, 9pA to 120pA and you display this on a log axis, Igor will label the tick marks with 10pA and 100pA. But if you really want the tick marks labeled 10 and 100 with pA in the axis label, you can set the prescaleExp to 12. For details, see **Axis Labels** on page II-280.

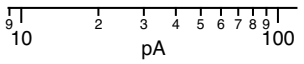
To see this, use the following commands:

```
Make/O jack=x
Display jack
SetScale x,9e-12,120e-12,"A",jack   // X scale 9 to 120 pico Amps
ModifyGraph log(bottom)=1
```

At this point we have a graph with a logarithmic bottom axis, and the major tick labels are 10 and 100 pA (we show only the bottom axis).

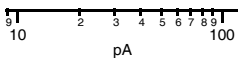
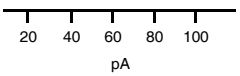
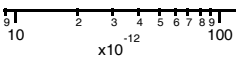
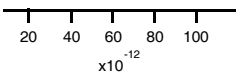
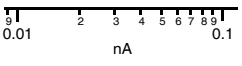
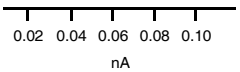
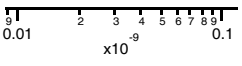
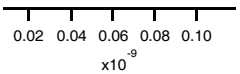
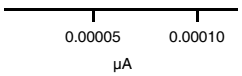
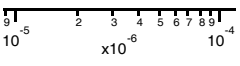
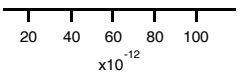
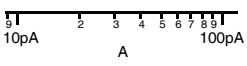
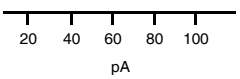
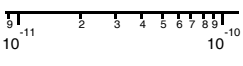
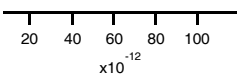


You would like to force the "pA" into the axis label, so you set Exponent Prescale to 12.



Exponent Prescale also works with normal axes (that is, not log). It may interact confusingly with the high and low trip points. Exponent Prescale only applies when the range of the axis is such that it does not trigger a trip point (note that the trip point is compared with the axis values *after* the exponent prescale is applied).

Here are examples of the above axis in both log and normal modes, with several values of Exponent Prescale. For these examples the low trip has been set to 1e-12 to avoid the confusing interactions, and the axis label was "\U" to display units, if any. In the examples with units, the units are "A".

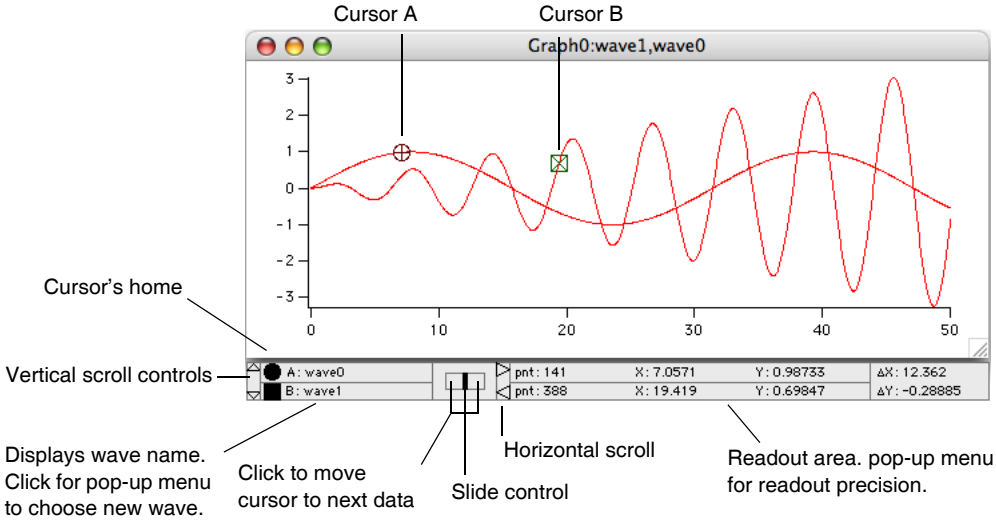
Log Axis with Units	Normal Axis with Units	Log Axis, no Units	Normal Axis, no Units
Prescale Exponent = 12			
			
Prescale Exponent = 9			
			
Prescale Exponent = 6			
			
Prescale Exponent = 0			
			

Annotations in Graphs

You can add text annotation to a graph by choosing Add Annotation from the Graph menu. This brings up the Add Annotation dialog. If text annotation is already on the graph you can modify it by double-clicking it. This brings up the Modify Annotation dialog. See Chapter III-2, **Annotations**, for details.

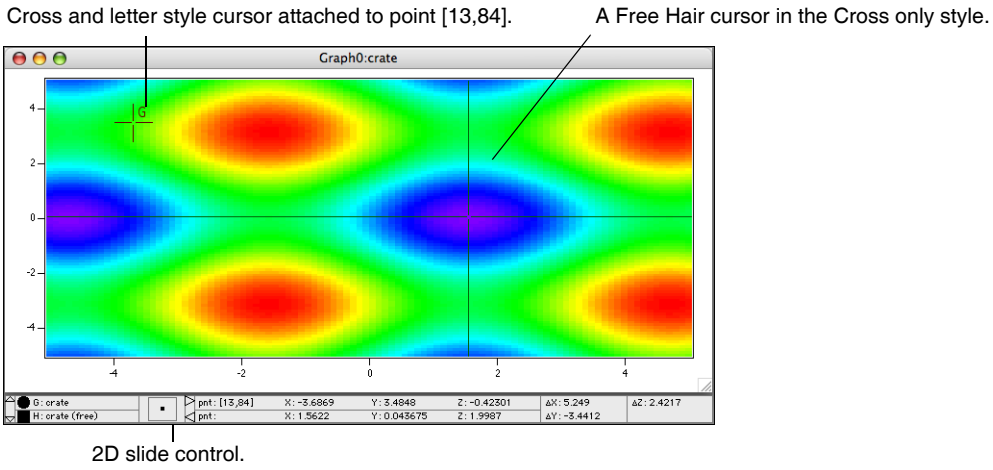
Info Box and Cursors

You can put an information box ("**info box**" for short) on a graph by choosing Show Info from the Graph menu while the graph is the target window. An info box displays a precise readout of values for waves in the graph. It also provides a convenient way to specify a region of interest on a wave for operations such as curve fitting (see **Fitting a Subset of the Data** on page III-177). To remove an info box from a graph while the graph is the target window choose Hide Info from the Graph menu.

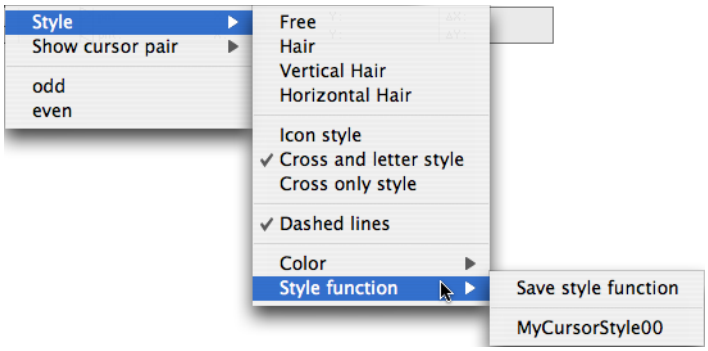


You can use up to five different pairs of cursors (AB through IJ). To view and select cursors, in the info panel, Control-click (*Macintosh*) or right click (*Windows*) in the cursor home area and select cursors from the "Show cursor pair" item in the pop-up menu. By default, cursors beyond B use the cross and letter style.

Icon style is the default for the A cursor, , and B cursor, ; for all other cursors it is the Cross and letter style. You can specify other cursor styles in the cursor Style pop-up menu when you Control-click (*Macintosh*) or right click (*Windows*) in the cursor name area: Hair style, which extends over the full range of the X and Y axes; "Cross only style", which is a smaller version of the Hair style; "Cross and letter style", which also includes cursor identifier. You can also specify colors and a dashed line format for the cursors.



All of the cursor styles can be applied in various ways by choosing appropriate combinations of styles in the Style pop-up menu. You can save your style settings as cursor style macros for easy reuse by choosing Save style function under the Style function submenu in the Cursor pop-up menu.



When you first put an info box on a graph the cursors are at home and not associated with any wave. The slide control is disabled and the readout area shows no values.

To **activate a cursor**, click it and drag it to the desired point on the wave whose values you want to examine. Now the cursor appears on the graph and the cursor's home is black indicating that the cursor is not home and that it is active. The name of the wave which the cursor is on appears next to the cursor's name. The slide control is enabled indicating that you can move the cursor. On images and for free cursors, the slider changes to a small square that you can drag up or down, left or right to move the cursor in the same way across the graph area.

In addition to attaching the cursors to points on a wave, you can use Free cursors, which can move "at will" anywhere within a graph. Simply choose Free in the cursor Style pop-up menu, following which you will see "(free)" appended to the wave name in the info area. Free cursors can move anywhere within the graph area and the cursor info are will update to show interpolated values at the cursor's position. Free cursors can be attached to both 1D and 2D traces in graphs.

The readout area shows the point number, X value, Y value, or Z value (when appropriate) for the point the cursor is on. If you put both cursors on the graph the dX readout shows the difference between the X value at cursor B and the X value at cursor A, the dY readout shows the difference between the Y value at cursor B and the Y value at cursor A, and the dZ readout shows the difference between the Z value at cursor B and the Z value at cursor A.

There are several ways to **move a cursor**. You can click it and drag it to a new point on the wave or to a new wave. You can drag the slide control right or left to move the cursor continuously right or left. You can click to one side or the other of the slide control or use the arrow keys on your keyboard to move the cursor by one point (Shift-arrow moves by 10 points). Whenever you move the cursor the readout area is updated. You can **remove a cursor** from the graph by dragging it away from the plotting area.

If you have both cursors on the graph and both are active, then the slide control moves both cursors at once. If you want to move only one cursor you can use the mouse to drag that cursor to its new location. Another way to move just one cursor is to **deactivate the cursor** that you don't want to move. You do this by clicking in the cursor's empty home. This makes the empty home change from black to white indicating that the cursor is not at home but also is not active. Then the slide control moves only the active cursor.

You can also move both cursors at once using direct drag. As long as both cursors are on the graph, you can move both by holding shift before clicking and dragging one of the cursors. The selected state of the cursor icon docks in the cursor info panel is irrelevant. You do not need to keep the Shift key depressed.

When you use the mouse to drag a cursor to a new location, Igor first searches for the wave the cursor is currently attached to. Only if the new location is not near a point on the current wave are all the other waves are searched. You can use this preferential treatment of the current wave to make sure the cursor lands on the desired wave when many traces are overlapping in the destination region.

You can also put a cursor on a particular wave using a pop-up menu. Position the mouse over the "A: *waveName*" or "B: *waveName*" area of the info box. Then click and choose a wave from the pop-up menu.

Sometimes the graph is not wide enough to see all of the readout area. In this case you can click the left or right **horizontal scroll** control to make the readout area scroll horizontally.

If you click the up or down **vertical scroll** control the entire info box scrolls vertically revealing a list of the traces and X waves in the graph. Click the vertical scroll again to return to the main info box.

The cursors provide a convenient way to specify a **range of points** on a wave which is of particular interest. For example, if you want to do a curve fit to a particular range of points on a wave you can start by putting cursor A on one end of the range and cursor B on the other. Then you can summon the Curve Fitting dialog from the Analysis menu. In this dialog on the Data Options tab there is a range control. If you click the "cursors" button in this dialog then the range of the fit will be set to the range from cursor A to cursor B.

Here are the built-in functions that return the current position of a cursor:

```
hcsr  pcsr  vcsr  xcsr  zcsr
```

Some additional cursor-related functions may prove useful when programming:

```
CsrWave  CsrWaveRef  CsrXWave CsrXWaveRef
```

The Cursor operation sets the position of either cursor, and the ShowInfo and HideInfo operations show and hide the info panel containing the cursor's homes.

These functions and operations are useful from the command line or in procedures. See their descriptions in Chapter V-1, **Igor Reference**.

Identifying a Trace

A help tag or tool tip that identifies a trace and contains information on the waves displayed by a trace can be displayed when you hover the mouse over a trace. To enable this mode, pull down the Graph menu and select Show Trace Info Tags.

Subrange Display

In addition to using an entire wave for display in graphs, you can specify a subrange of your data to display. This feature is mainly intended to allow the display of columns of a matrix as if they were extracted into individual 1D waves but can also be used to display other subsets or to skip every *n*th data point. To use this feature using the New Graph and Append Traces dialogs, you must be in the more complex version of the dialogs used when you click the More Choices button. You may then add entries to the holding list and from there you can edit the subrange settings.

Subrange Display Syntax

The **Display** operation (page V-116), **AppendToGraph** operation (page V-26), and **ReplaceWave** operation (page V-524) support the following subrange syntax for a wave list item:

wavename`[`*rdspec*`][`*rdspec*`][`*rdspec*`][`*rdspec*`]`

where *rdspec* is a range or dimension specification and the brackets are part of the syntax (rather than indicating options). Higher unneeded specs can be omitted. Only one my be a range spec and the others must be a single numeric or dimension label value. A range spec may be `[]` or `[*]` to indicate the entire range of the dimension, may be `[`*start*`,`*stop*`]`, or `[`*start*`,`*stop*`;`*inc*`]` where *stop* may be `*`.

This can be restated as:

1. Only one dimension specifier may contain the range to be displayed.

   Legal syntax for range is: `[]` or `[*]` for an entire dimension.

[*start*,*stop*] for a subrange; *stop* may be *, *stop* must be >= *start*; the range is inclusive.

[*start*,*stop*;*inc*] for a subrange with the specified positive nonzero increment (*inc*) value.

2. Other dimensions must contain a single numeric value, or dimension label using % syntax.

   Legal syntax for nonrange specifier is: [*value*] or [%*name*].

3. Unspecified higher dimensions are treated as if zero was specified.

For non-XY plots, the X-axis label uses the dimension label (if any) for the active dimension (the one with a range).

When cursors or tags are placed on a subranged trace, the point number used is the virtual point number as if the subrange had been extracted into a 1D wave.

Subrange syntax is also supported for waves used with ErrorBars when an error bar wave is selected (see **Error Bars** on page II-260), and color, marker size and marker number as f(Z) (see **Setting Trace Properties from an Auxiliary (Z) Wave** on page II-255). These correspond to the **ErrorBars** operation (page V-143) used with the wave keyword and to the **ModifyGraph (traces)** operation (page V-400) with the zmrkSize, zmrkNum, and zColor keywords.

Limitations

In category plots, the category wave (the text wave) may not be subranged. Waves used to specify text using ModifyGraph textMarker mode may not be subranged.

Subranged traces may not be edited using the draw tools (such as: option click on the edit poly icon in the tool bar on a graph).

Waterfall plots may not use subranges.

When multiple subranges of the same wave are used in a graph, they are distinguished only using instance notation and not using the subrange syntax. For example, given `display w[][0],w[][1]`, you must use `ModifyGraph mode(w#0)=1,mode(w#1)=2` and not `ModifyGraph mode(w[][0])=1,mode(w[][1])=2` as you might expect.

The trace instance and subrange used to plot given trace is included in trace info information. See **Identifying a Trace** on page II-288.

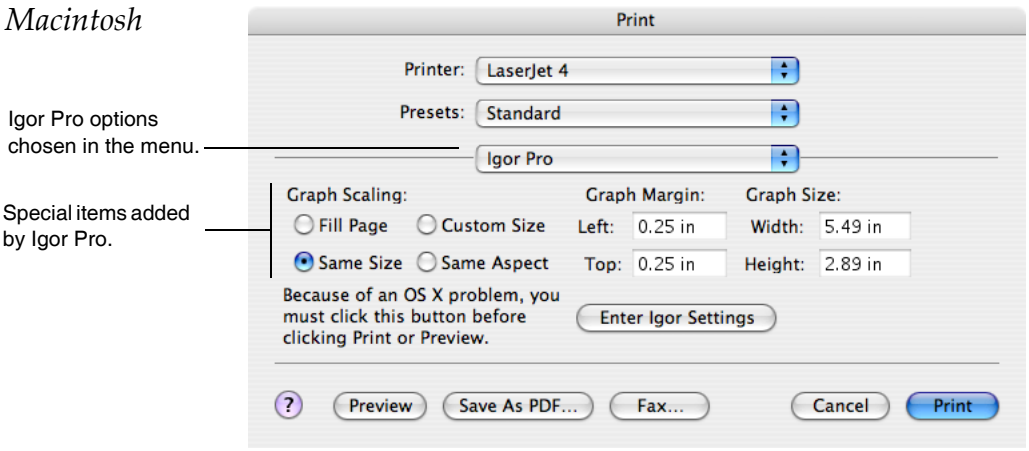
Printing Graphs

Before printing a graph you should set the page size and orientation using the Page Setup dialog. Choose Page Setup from the Files menu. Often graphs are wider than they are tall and look better when printed using the horizontal orientation.

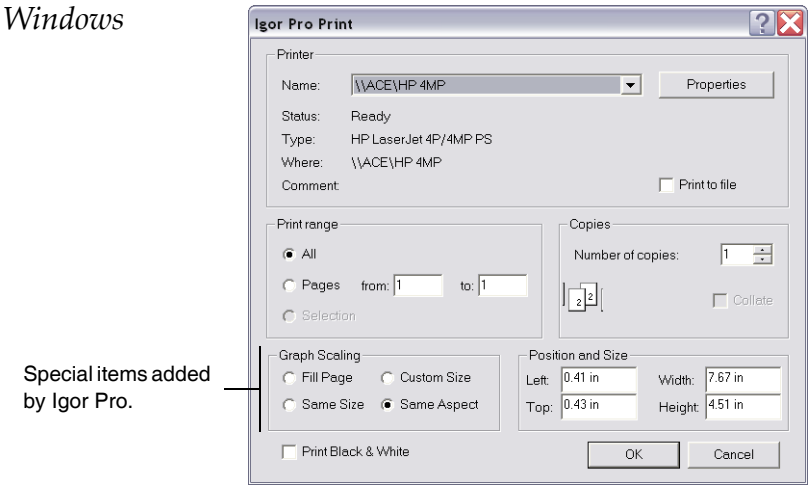
When you invoke the Page Setup dialog you must make sure that the graph that you want to print is the top window. Igor stores one page setup in each experiment for all graphs and stores other page setups for other types of windows. You can set the default graph page setup for new experiments using the Capture Graph Preferences dialog.

To print a graph, choose Print from the File menu while the graph is the active window. This brings up the Print dialog. In addition to the standard items, the Print dialog contains items which allow you to control how the graph is scaled and positioned within the printed page.

Macintosh



Windows



If you enable the Fill Page radio button, Igor fills the printed page with the graph, changing the aspect ratio of the graph as necessary.

If you enable the Same Size radio button (the default), Igor sets the size of the printed graph equal to its size on the screen. If you have constrained the size of the graph, using the Modify Graph dialog, then you normally should use the Same Size setting.

If you enable the Same Aspect radio button, Igor makes the graph as large as possible so that it fits on the printed page and retains the aspect ratio of the graph on the screen.

If you enable the Custom Size radio button, Igor positions and sizes the graph according to the margins and dimensions that you enter in the Graph Margin and Graph Size parameter boxes.

You can use the custom size setting and the related Left, Top, Width and Height parameters to print a graph of any size at any position on the page.

Printing Poster-Sized Graphs

Using the Custom Size setting in the Print dialog, you can specify a size for a graph that will not fit on a single sheet of paper. When you do this, Igor uses multiple sheets of paper to print the graph. This makes very large, poster-sized graphs. Another approach is to specify a custom size that *does* fit on a single sheet and use the Reduce/Enlarge setting in the Page Setup dialog to enlarge the graph by, for example, 200%.

To make the multiple sheets into one big poster, you need to trim the edges of the sheets and tape them together. Igor prints tiny alignment marks on the edges so you can line the pages up. You should trim the unneeded borders so that the alignment marks are flush against the edge of the trimmed sheet. Then align the sheets so that the alignment marks butt up against each other. All of the alignment marks should still be visible. Then tape the sheets together.

If you need to reprint some of the sheets but not all you can use the Pages settings in the Print dialog. For example, if you need to reprint just sheet two of a four sheet poster, enter "2" in the From parameter box and "2" in the To parameter box. Now Igor will reprint only sheet number two.

Other Printing Methods

You can also print graphs by placing them in page layouts. See Chapter II-16, **Page Layouts** for details.

You can print graphs directly from macros using the **PrintGraphs** (see page V-501) operation.

Save Graph Copy

You can save the active graph as an Igor packed experiment file by choosing File→Save Graph Copy. The main uses for saving as a packed experiment are to save an archival copy of data or to prepare to merge data from multiple experiments (see **Merging Experiments** on page II-32). The resulting experiment file preserves the data folder hierarchy of the waves displayed in the graph starting from the "top" data folder, which is the data folder that encloses all waves displayed in the graph. The top data folder becomes the root data folder of the resulting experiment file. Only the graph, its waves, dashed line settings, and any pictures used in the graph are saved in the packed experiment file, not procedures, variables, strings or any other objects in the experiment.

Save Graph Copy does not work well with graphs containing controls. First, the controls may depend on waves, variables or FIFOs (for chart controls) that Save Graph Copy will not save. Second, controls typically rely on procedures which are not saved by Save Graph Copy.

Save Graph Copy does not know about dependencies. If a graph contains a wave, wave0, that is dependent on another wave, wave1 which is not in the graph, Save Graph Copy will save wave0 but not wave1. When the saved experiment is open, there will be a broken dependency.

The **SaveGraphCopy** operation on page V-539 provides options that are not available using the Save Graph Copy menu command.

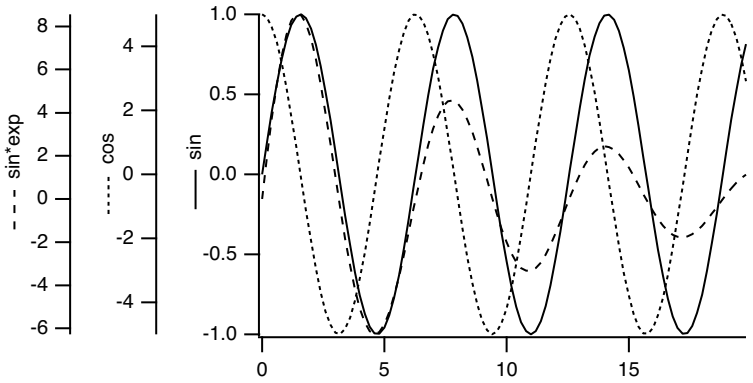
Exporting Graphs

You can export a graph to another application through the Clipboard or by creating a file. To export via the Clipboard, use the Export Graphics item in the Edit menu. To export via a file, use the Save Graphics item of the File menu.

The process of exporting graphics from a graph is very similar to exporting graphics from a layout. Because of this, we have put the details in Chapter III-5, **Exporting Graphics (Macintosh)**, and Chapter III-6, **Exporting Graphics (Windows)**. These chapters describe the various export methods you can use and how to choose a method that will give you the best results.

Creating Graphs with Multiple Axes

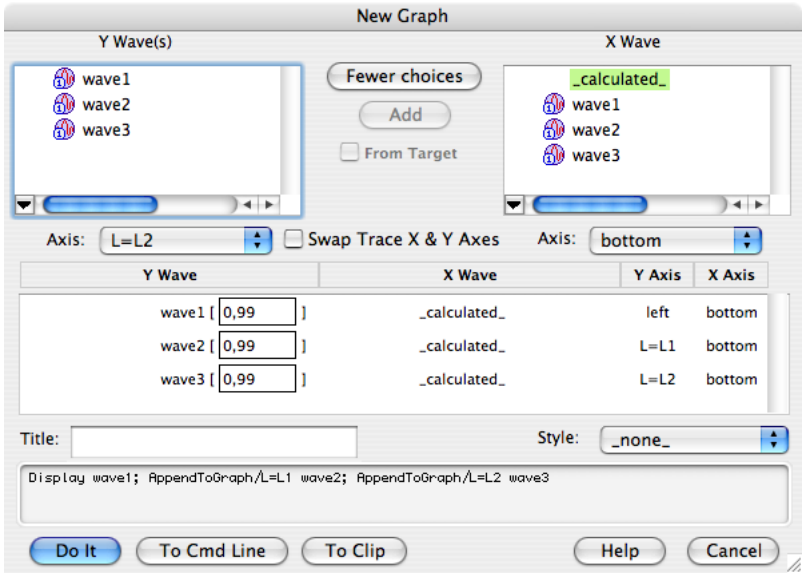
This section describes how to create a graph that has many axes attached to a given plot edge. For example:



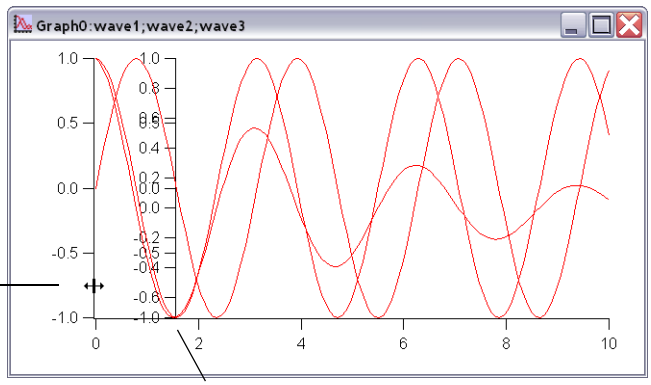
To create the above example we first need some data:

```
Make/N=100 wave1,wave2,wave3; SetScale x,0,20,wave1,wave2,wave3
wave1=sin(x); wave2= 5*cos(x); wave3= 10*sin(x)*exp(-0.1*x)
```

Next, we use the New Graph dialog in the expanded mode to create two free axes attached to the left edge:



This creates the following initial graph:



Position cursor over left axis while pressing Option (*Macintosh*) or Alt (*Windows*) to drag left margin.

Both free axes, L1 & L2, are initially positioned 50 points in from left edge

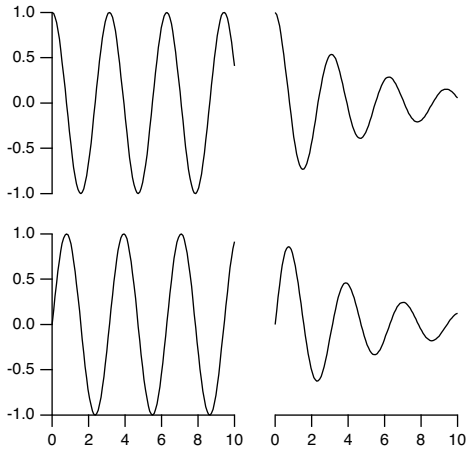
Next, we drag the left margin to the right, drag the two free axes to the left, change the line styles of the traces and finally create the axis labels (see **Axis Labels** on page II-280). In the axis label we use the Wave Symbol from the Special pop-up menu to include the line style. Drag the axis labels into place to complete the plot shown above.

Creating Stacked Plots

Igor's ability to use an unlimited number of axes in a graph combined with the ability to shrink the length of an axis makes it easy to create stacked plots. You can even create a matrix of plots and can also create inset plots.

Another way to make a stacked graph embedded subwindows. See Layout Mode and Guide Tutorial for an example. It is also possible to do make stacked graphs in page layouts, using either embedded graph subwindows or the traditional graph layout objects.

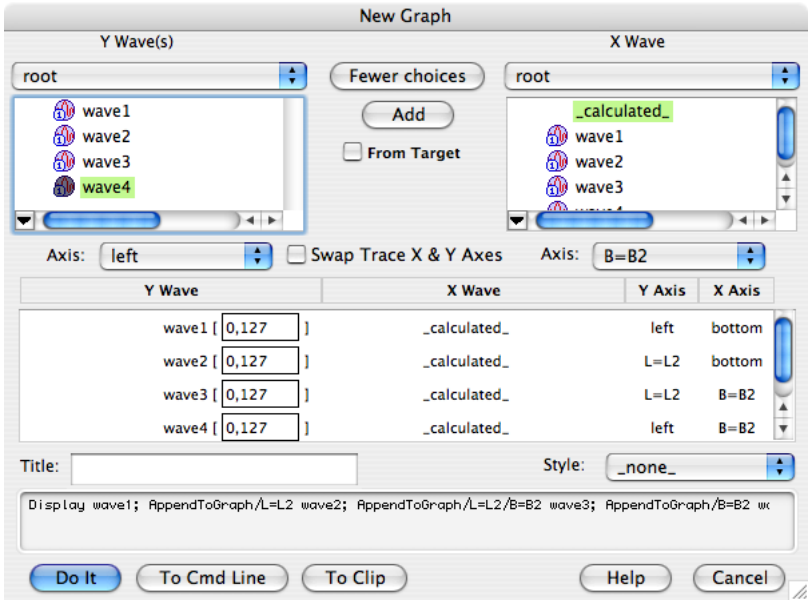
As an example, we will create the following graph:



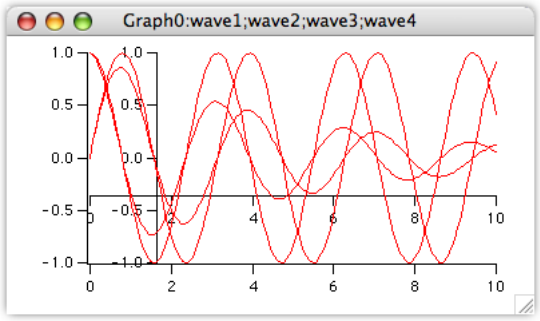
First we create some data:

```
Make wave1,wave2,wave3,wave4
SetScale/I x 0,10,wave1,wave2,wave3,wave4
wave1= sin(2*x); wave2= cos(2*x)
wave3=cos(2*x)*exp(-0.2*x)
wave4=sin(2*x)*exp(-0.2*x)
```

Next, we use the extended form of the New Graph dialog:

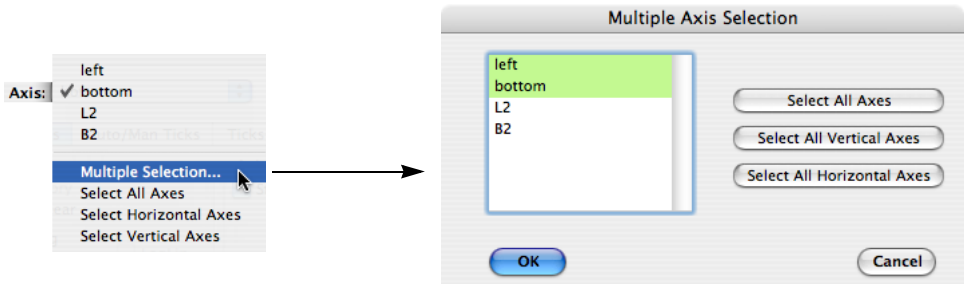


Notice that two free axes were created with the arbitrary names L2 and B2. Clicking the Do It button gave the following jumble of axes and traces:



Next, we double click one of the axes to get to the Modify Axis dialog and then go to the Axis tab if necessary. We set the left and bottom axes to be drawn from 0 to 45% of normal. Next we set the L2 and B2 axes to be drawn from 55 to 100% of normal and set their "Free axis position Distance" to 0.

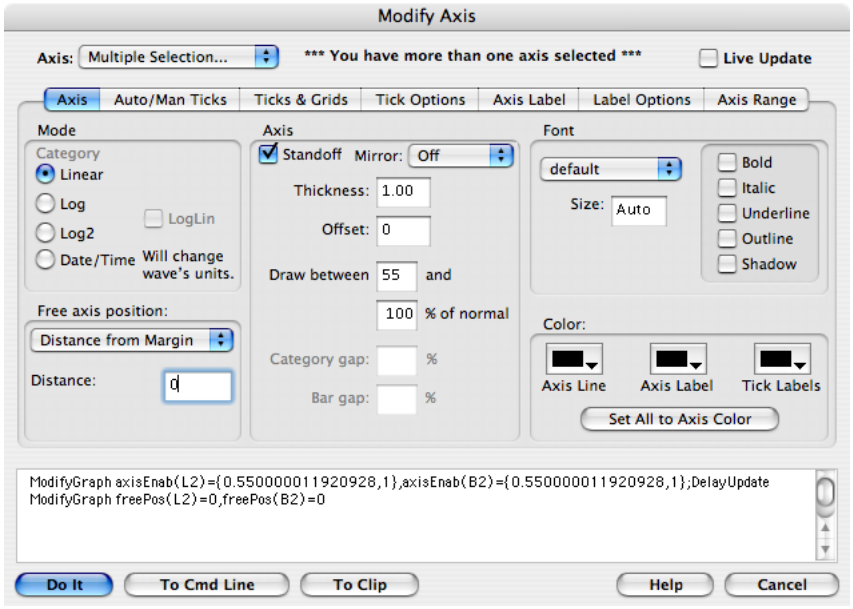
This is a case in which Multiple Axis Selection is convenient. You can select the left and bottom axes, then set these axes to draw from 0 to 45% simultaneously:



Remember that *all* selected axes are changed any time you *touch* any control in the dialog, so be careful when you have multiple axes selected.

When the left and bottom axes are done, return to the Multiple Axis Selection dialog and select the L2 and B2 axes and set them to draw between 55% and 100%. Then set the Free axis position Distance to 0.

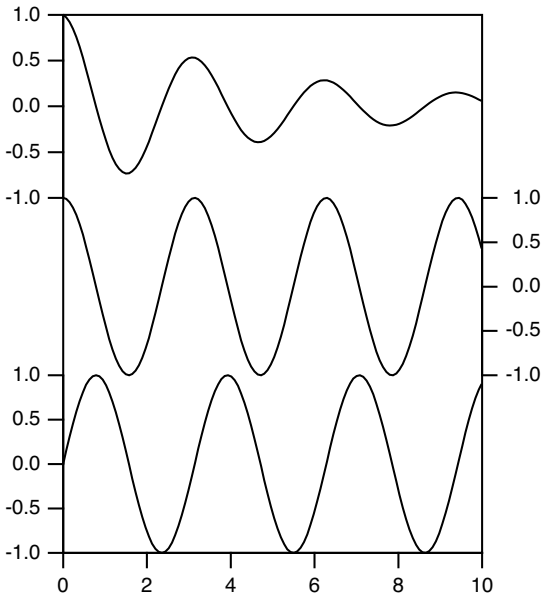
Here is how the Modify Axis dialog should look:



Once everything is properly set up, click Do It. The graph need only be resized to replicate the above example.

Staggered Stacked Plot

Here is a common variant of the stacked plot:



This example was created from three of the waves used in the previous plot. Wave1 was plotted using the left and bottom axes, wave2 used the right and bottom axes and wave3 used L2 and bottom axes. Then the Axis tab of the Modify Axis dialog was used to set the left axis to be drawn from 0 to 33% of normal, the right axis 33 to 66% and the L2 axis 66 to 100%. The Axis Standoff checkbox was deselected for the bottom axis. This was not necessary for the other axes; axis standoff is not used when axes are drawn on a reduced extent.

After returning from the Modify Axis dialog, the graph was resized and the frame around the plot area was drawn using a polygon in plot-relative coordinates. This allows the frame to be exactly centered over the plot rectangle. Here is a function that adds the plot frame:

```
Function AddPlotFrame()
   SetDrawLayer UserBack
   SetDrawEnv xcoord=prel,ycoord=prel,fillpat=0
   DrawPoly 0,0,1,1,{0,0,0,1,1,1,1,0,0,0}
   SetDrawLayer UserFront
End
```

To use this function, include the WaveMetrics-supplied "AddPlotFrame" file. See **Including a Procedure File** on page III-346 for instructions on including a procedure file.

In case you need to change its color or thickness, you should note that it is in the UserBack draw layer. That is so the axes will be drawn on top of the frame. You might find that decreasing the thickness of the plot frame to, say, 0.25 points will highlight the axes nicely.

Waterfall Plots

You can create a graph displaying a sequence of traces in a perspective view. We refer to these types of graphs as waterfall plots, which can be created and modified using, respectively, the **NewWaterfall** operation on page V-444 and **ModifyWaterfall** operation on page V-423. At present, there is no dialog interface that you can use to create waterfall plots, so you must either execute these commands on the Command Line or in an Igor Procedure.

To display a waterfall plot, you must first create or load a matrix wave. In this 2D matrix, each of the individual matrix columns is displayed as a separate trace in the waterfall plot. Each column from the matrix wave is plotted in, and clipped by, a rectangle defined by the X and Z axes with the plot rectangle displaced along the angled Y axis, which is the right-hand axis, as a function of the Y value.

You can display only one matrix wave per plot.

To modify certain properties of a waterfall plot, you must use the ModifyWaterfall operation. For other properties, you will need to use the usual axis and trace dialogs.

Because the traces in the waterfall plot are from a single wave, any changes to the appearance of the waterfall plot using the Modify Trace Appearance dialog or ModifyGraph operation will globally affect all of the waterfall traces. For example, if you change the color in the dialog, then all of the waterfall traces will change to the same color. If you want each of the traces to have a different color, then you will need to use a separate wave to specify (as f(z)) the colors of the traces. See the example in the next section for an illustration of how this can be done.

The X and Z axes of a waterfall are always at the bottom and left while the Y axis runs at a default 45 degrees on the right-hand side. The angle and length of the Y axis can be changed using ModifyWaterfall. Except when hidden lines are active, the traces are drawn in back to front order. Note that hidden lines are active only when the trace mode is lines between points.

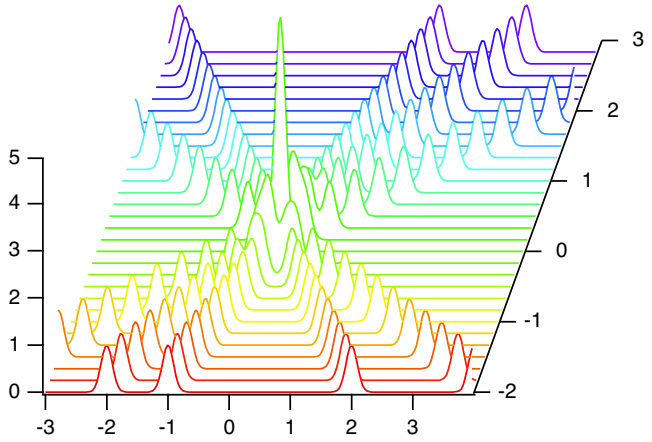
Marquee expansion is based only on the bottom and right (waterfall) axes. The marquee is drawn as a box with the bottom face in the ZY plane at zmin and the top face is drawn in the ZY plane at zmax.

Cursors may be used and the readout panel provides X, Y and Z axis information. The hcsr and xcsr functions are unchanged; the vcsr function returns the Y data value (waterfall) and the zcsr returns the data (Z axis) value.

Example

```
Make/O/N=(200,30) jack
SetScale x,-3,4,jack
SetScale y,-2,3,jack
jack=exp(-((x-y)^2+(x+3+y)^2))
jack=exp(-60*(x-1*y)^2)+exp(-60*(x-0.5*y)^2)+exp(-60*(x-2*y)^2)
jack+=exp(-60*(x+1*y)^2)+exp(-60*(x+2*y)^2)
NewWaterfall /W=(21,118,434,510) jack
ModifyWaterfall angle=70, axlen= 0.6, hidden= 3
```

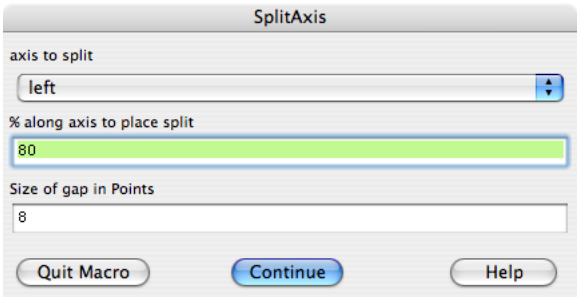
```
duplicate jack,jacky
jacky=y
ModifyGraph zColor(jack)={jacky,*,*,Rainbow}
```



Creating Split Axes

You can create split axes using the same techniques just described for creating stacked plots. Simply plot your data twice using different axes and then adjust the axes so they are stacked. You can then adjust the range of the axes independently. You can use draw tools to add cut marks.

WaveMetrics supplies a macro package to automate all aspects of creating split axes except setting the range and adjusting the ticking details of the axes. To use the macro package, select the Graph→Packages→Split Axes menu item. An example experiment using these macros is provided in the Examples folder.



Before using the macros, you should create the graph in near final form using just the main axes. For best results, especially if you will be using cut marks, you should work with the graph at actual size before adding the split axes. It is recommended that you create a recreation macro just before executing the split axis macros. This is so you can easily revert in case you need to change the presplit setup.

After creating the split, you can execute the AddSplitAxisMarks macro to add cut marks between the two axes. You can then use the drawing tools to duplicate the cut marks if you want marks on the traces as well as the axes. Of course, you can also draw your own cut marks. You should use the default Plot Relative coordinate system for cut marks so they will remain in the correct location should you resize the graph.

Some programs draw straight lines between data points on either side of the split. While such lines provide the benefit of connecting traces for the viewer, they also are misleading and inaccurate. This macro package accurately plots both sections and does not attempt to provide a bridge between the two sections. If you feel it is necessary, you can use drawing tools to add a connecting bridge. You can even erase sections of the existing traces by drawing a rectangle or polygon with zero line thickness and with the fill mode set to erase.

Live Graphs and Oscilloscope Displays

This section will be of interest mainly to those involved in data acquisition.

Normally, when the data in a wave is modified, all graphs containing traces derived from that wave are redrawn from scratch. Although fast compared to other programs, this process generally takes at least one second, thereby limiting the update rate to approximately one Hz. However, if you specify one or more

traces in a graph as being "live" then Igor will configure the graph such that updating the display does not require redrawing from scratch and is therefore much faster than normal. With short (100 point) waves on a fast computer you may be able to attain a 20 Hz update rate or better. The fast update will only be obtained when certain conditions are observed.

Note: When graphs are redrawn in live mode, *autoscaling is not done*.

To specify a trace in a graph as being live you must use the **live** keyword with the ModifyGraph command. There is no dialog support for this setting.

```
ModifyGraph live(traceName)= mode
```

Mode can be 0 or 1. Zero turns live mode off for the given trace.

WaveMetrics provides an example experiment that generates and displays synthetic data. You should use this experiment to get a feel for the performance you might expect on your particular computer as a function of the window size, number of points in the live wave, color setup of your monitor and the live modes. The example experiment is called "Live mode" and can be found in the "Examples:Feature Demos" folder.

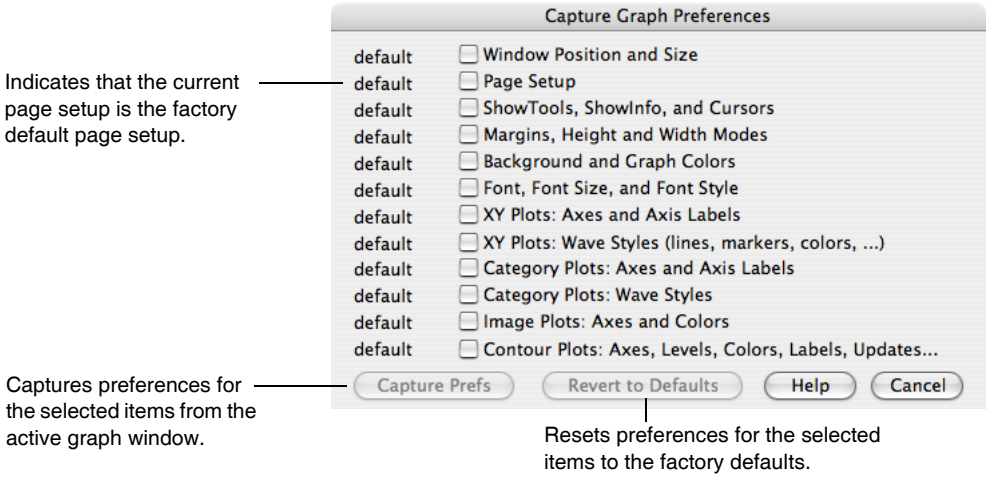
Although live mode 1 is not restricted to unity thickness solid lines or dots modes, you will get the best performance if you do use these settings.

Another feature that may be of use is the quick append mode. It is intended for applications in which a data acquisition task creates new waves periodically. It permits you to add the new waves to a graph very quickly. To invoke a quick append, use the /Q flag in an AppendToGraph command. There is no dialog support for this setting.

A side effect of quick append is that it marks the wave as not being modified since the last update of graphs and therefore prevents other graphs containing the same wave, if any, from being updated. See the "Quick Append" example experiment in the "Examples:Feature Demos" folder.

Graph Preferences

Graph preferences allow you to control what happens when you create a new graph or add new traces to an existing graph. To set preferences, create a graph and set it up to your taste. We call this your *prototype* graph. Then choose Capture Graph Prefs from the Graph menu.



Preferences are normally in effect only for *manual* operations, not for automatic operations from Igor procedures. This is discussed in more detail in Chapter III-17, **Preferences**.

When you initially install Igor, all preferences are set to the factory defaults. The dialog indicates which preferences you have not changed by displaying "default" next to them.

The Window Position and Size preference affects the creation of new graphs only. New graphs will have the same size and position as the prototype graph.

The Page Setup preference is somewhat unusual because all graphs share the same page setup settings, as shown in the Page Setup dialog. The captured page setup is already in use by all other graphs. The utility of this category is that new *experiments* will use the captured page setup for graphs.

The "XY Plots:Wave Styles" preference category refers to the various wave-specific settings in the graph, such as the line type, markers and line size, set with the Modify Trace Appearance dialog. This category also includes settings for Waveform plots. Each captured wave style is associated with the index of the wave it was captured from. The index of the first wave displayed or appended to a graph is 0, the second appended wave has an index of 1, and so on.These indices are the same as are used in style macros. See **Graph Style Macros** on page II-300.

If preferences are on when a new graph with waves is created or when a wave is appended to an existing graph, the wave style assigned to each is based on its index. The wave with an index of 2 is given the captured style associated with index 2 (the third wave appended to the captured graph).

You might wonder what style is applied to the fifth and sixth waves if only four waves appeared in the graph from which wave style preferences were captured. You have two choices; either the factory default style is used, or the styles repeat with the first wave style and then the second style. You make this choice in the Miscellaneous Settings dialog, with the Repeat Wave Style Prefs in Graphs checkbox. With that box selected, the fifth and sixth waves would get the first and second captured styles; if deselected, they would both get the factory default style, as would any other waves subsequently appended to the graph.

The XY Plots:Axes and Axis Labels preferences category captures all of the axis-related settings for axes in the graph. Only axes used by XY or Waveform plots have their settings captured. Axes used solely for an category, image, or contour plot are ignored. The settings for each axis are associated with the name of the axis it was captured from.

Even if preferences are on when a new graph with waves is created or when a wave is newly appended to an existing graph, the wave is still displayed using the usual default left and bottom axes unless you explicitly specify another named axis. The preferred axes are not automatically applied, but they are listed by name in the New Graph, and the various Append to Graph dialogs, in the two Axis pop-up menus so that you may select them.

For example, suppose you capture preferences for an XY plot using axes named "myRightAxis" and "myTopAxis". These names will appear in the X Axis and Y Axis pop-up menus in the New Graph and Append Traces to Graph dialogs.

- If you choose them in the New Graph dialog and click Do It, a graph will be created containing *newly-created* axes named "myRightAxis" and "myTopAxis" and having the axis settings you captured.
- If you have a graph which already uses axes named "myRightAxis" and "myTopAxis" and choose these axes in the Append Traces to Graph dialog, the traces will be appended to those axes, as usual, but no captured axis settings will be applied to these *already-existing* axes.

Captured axes may also be specified by name on the command line or in a macro or function, provided preferences are on:

```
Function AppendWithCapturedAxis(wav)
   Wave wav
   Variable oldPrefState
   Preferences 1; oldPrefState = V_Flag   // Turn preferences on
   Append/L=myCapturedAxis wav      // Note: myCapturedAxis must
                                    //       be vertical to use /L
   Preferences oldPrefState         // Restore old prefs setting
End
```

The Category Plots:Axes and Axis Labels and Category Plots:Wave Styles are analogous to the corresponding settings for XY plots. Since they are separate preference categories, you have can independent preferences for category plots and for XY plots. Similarly, preferences for image and contour plots are

independent of preferences for other types. See Chapter II-13, **Category Plots**, Chapter II-14, **Contour Plots**, and Chapter II-15, **Image Plots**.

How to use Graph Preferences

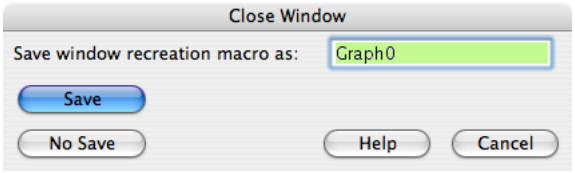
Here is our recommended strategy for using graph preferences:

1. Create a new graph containing a single trace. Use the axes you will normally use.
2. Make the graph appear as you prefer using the Modify Graph dialog, Modify Trace Appearance dialog, the Modify Axis dialog, etc. Move the graph to where you prefer it be positioned.
3. Choose the Graph→Capture Graph Prefs menu. Select the desired categories, and click Capture Prefs.
4. Bring up the Miscellaneous Settings dialog from the Misc menu, choose Graph Settings, and select the Repeat Wave Style Prefs in Graphs checkbox.

Saving and Recreating Graphs

When you save an experiment all of its windows, including graphs, are saved as part of the experiment. When you reopen the experiment all windows will be intact.

If you click in the close button of a graph window, Igor asks you if you want to save a **window recreation macro**.



Igor presents the graph's name as the proposed name for the macro. You can replace the proposed name with any valid macro name.

If you want to make a macro to recreate the graph, click Save or press Return or Enter. If you press Option (*Macintosh*) or Alt (*Windows*) you will notice that the No Save button becomes the default allowing you to press Return or Enter to dismiss the dialog without saving a recreation macro.

If you do choose to save, Igor will create a macro which, when invoked, will recreate the graph with its size, position and stylistic presentation intact. The macro is placed in the procedure window where you can inspect, modify or delete it as you like. The macro name also appears in the Graph Macros submenu of the Windows menu.

You can invoke the macro by choosing it from the Windows menu or by typing its name on the command line. The graph name for the graph that is recreated will be the same as the name of the macro that created it.

If you are sure that you don't want to make a recreation macro for the graph you can press Option (*Macintosh*) or Alt (*Windows*) while you click the close button of the graph window. This skips the dialog.

For a general discussion of saving, recreating, closing windows, see Chapter II-4, **Windows**.

Graph Style Macros

The purpose of a graph style macro is to allow you to create a number of graphs with the same stylistic properties. Igor can automatically generate a style macro from a prototype graph. You can manually tweak the macro if necessary. Later, you can apply the style macro to a new graph.

For example, you might frequently want to make a graph with a certain sequence of markers and colors and other such properties. You *could* use preferences to accomplish this. The style macro offers another way and has the advantage that you can have any number of style macros while there is only one set of preferences.

You create a graph style macro by making a prototype graph, setting each of the elements to your taste and then, using the Window Control dialog, instructing Igor to generate a style macro for the window.

You can apply the style macro when you create a graph using the New Graph dialog. You can also apply it to an existing graph by choosing the macro from the Graph Macros submenu of the Windows menu.

Example of Creating a Style Macro

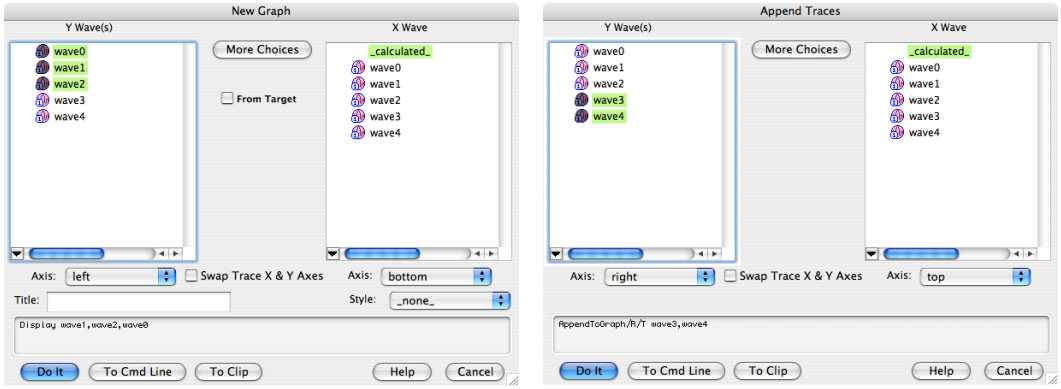
As an example, we will create a style macro that defines the color and line type of five traces and some aspects of the axes.

Since we want our style macro to define a style for five traces, we start by making five waves with the command

```
Make wave0, wave1, wave2, wave3, wave4
```

The length, numeric type and scaling of the waves does not matter. We need them only to have something to put in our prototype graph.

We create the prototype graph using the New Graph item in the Windows menu. We want our graph to have a right and a top axis as well as the standard left and bottom. Therefore we initially display wave0, wave1 and wave2 in the prototype graph. Then, using the Append Traces to Graph item in the Graph menu, we append wave3 and wave4 using the right and top axes.



You could create this graph with the New Graph dialog only, using the features provided by the More Choices button.

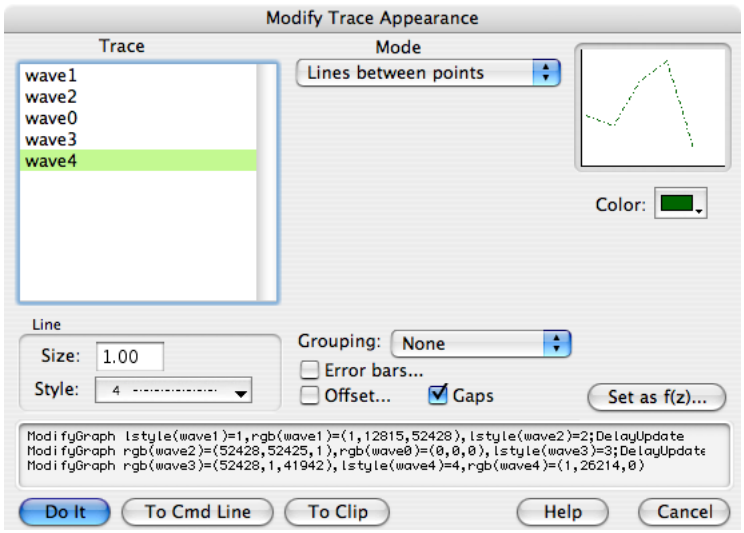
All of the data in the waves is zero. To distinguish the five waves, we fill them with data using the following command:

```
wave0=0; wave1=1; wave2=2; wave3=3; wave4=4
```

It will be clearer if both the left and the right axes cover the same range. So, we execute

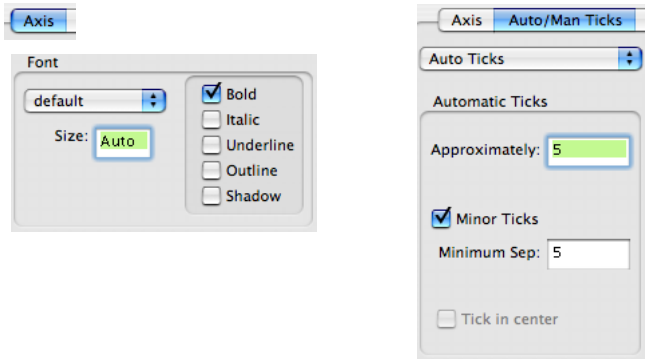
```
SetAxis left, -1, 5; SetAxis right, -1, 5
```

Now, using the Modify Trace Appearance item in the Graph menu, we set the color and line style for each of the waves to our liking:

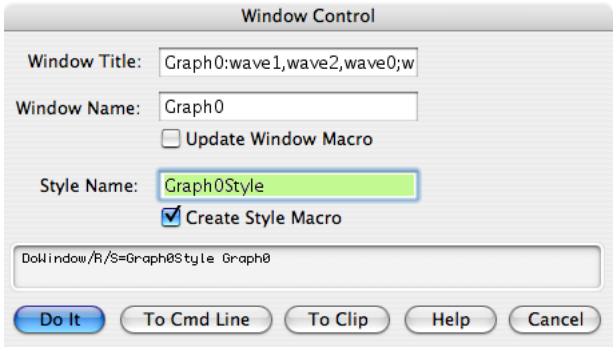


Next, we set our axes to the desired state, using the Modify Axis dialog. Let's turn minor ticks on and make labels bold for all four axes. This is conveniently done by choosing Select All Axes from the Axis pop-up menu at the top left corner of the dialog.

And then set up the appropriate controls in the Modify Axis dialog:



Now, we're ready to generate the style macro. With the graph the active window, we use the Window Control item in the Control submenu of the Windows menu. We enable the Create Style Macro checkbox.



When we click Do It, Igor generates a graph style macro and puts it in the procedure window.

The graph style macro for this example is:

```
Proc Graph0Style() : GraphStyle
   PauseUpdate; Silent 1      // modifying window...
   ModifyGraph/Z lStyle[1]=1,lStyle[2]=2,lStyle[3]=3,lStyle[4]=4
   ModifyGraph/Z rgb[0]=(0,0,0),rgb[1]=(577,43860,60159)
```

```
   ModifyGraph/Z rgb[2]=(56683,2242,1698)
   ModifyGraph/Z rgb[3]=(65495,2134,34028),rgb[4]=(64512,62423,1327)
   ModifyGraph/Z minor=1
   ModifyGraph/Z fStyle=1
   SetAxis/Z left -1,5
   SetAxis/Z right -1,5
EndMacro
```

The macro will sometimes need some touchup. In this example, we see two SetAxis commands at the bottom. For this example, we don't really consider the axis range as part of the style, so we delete these lines.

Notice that the graph style macro does not refer to wave0, wave1, wave2, wave3 or wave4. Instead, it refers to traces by index. For example,

```
ModifyGraph rgb[0]=(0,0,0)
```

sets the color for the trace whose index is 0 to black. A trace's index is determined by the order in which the traces were displayed or appended to the graph. In the Modify Trace Appearance dialog, the trace whose index is zero appears at the top of the list.

The /Z flag used in the graph style macro ignores any potential errors if the command tries to modify a trace that is not actually in the graph. For example, if you make a graph with three traces (indices from 0 to 2) and apply this style macro to it, there will be no trace whose index is 3 at the time you run the macro. The command:

```
ModifyGraph rgb[3]= (65495,2134,34028)
```

would generate an error in this case. Adding the /Z flag continues macro execution and ignores the error.

Style Macros and Preferences

When Igor generates a graph style macro, it generates commands to modify the target graph according to the prototype graph. *It assumes that the objects in the target will be in their factory default states at the time the style macro is applied to the target*. Therefore, it generates commands only for the objects in the prototype which have been modified. If Igor did not make this assumption, it would have to generate commands for every possible setting for every object in the prototype and style macros would be very large.

Because of this, you should create the new graph *with preferences off* and then apply the style macro.

Applying the Style Macro

To use this macro, you would perform the following steps.

1. Turn preferences off by choosing Preferences Off from the Misc menu.
2. Create a new graph, using the New Graph dialog and optionally the Append Traces to Graph dialog.
3. Choose Graph0Style from the Graph Macros submenu in the Windows menu.
4. Turn preferences back on by choosing Preferences On from the Misc menu.

If you use only the New Graph dialog, you can use the shorter method:

1. Open the New Graph dialog, select the wave(s) to be displayed in the graph, and choose Graph0Style from the Style pop-up menu in the dialog. Click Do It.

Igor automatically generates the Preferences Off and Preferences On commands to apply the style to the new graph without being affected by preferences.

Limitations of Style Macros

Igor automatically generates style macro commands to set all of the properties of a graph that you set via the ModifyGraph, Label and SetAxis operations. These are the properties that you set using the Modify Trace Appearance, Modify Graph, and Modify Axis dialogs.

It does not generate commands to recreate annotations or draw elements. Igor's assumption is that these things will be unique from one graph to the next. If you want to include commands to create annotations and draw elements in a graph, it is not too difficult, following these steps.

1. Make your prototype graph with annotations and draw elements.
2. Use the Window Control dialog to create a graph *recreation* macro (not a *style* macro).
3. In the procedure window, copy the relevant commands from the graph recreation macro to the end of the style macro. These are the commands that start with Tag, Textbox, Legend, or any of the drawing-related operations. All of these commands will be at the end of the recreation macro.

There is a problem in using a Tag command in a style macro. This command needs to reference a particular trace in the graph. It might look something like this:

```
Tag/N=text0 wave0, 10, "This is a test."
```

wave0 was in the prototype graph but it would probably not be in the target graph when the style macro is applied. Your options would be to remove the Tag command from the style macro or to use the WaveName function to provide an appropriate wave name when the macro is executed.

For example, to tag the first trace in the graph, you would use:

```
Tag/N=text0 $WaveName("",0,1), 10, "This is a test."
```

Even this would run into problems if the X scaling of the wave associated with the trace in the target graph was different from the X scaling of the wave associated with the trace in the prototype graph, because the parameter that sets the X location of the tag (10 in this case) would not be appropriate.

Also, the Tag command actually uses the name of a *trace*, not a wave. If the same wave is displayed more than once, or if more than one wave with the same name is displayed, using `$WaveName` may attach the tag to the wrong trace.

Where to Store Style Macros

If you want a style macro to be accessible from a single experiment only, you should leave them in the main procedure window of that experiment. If you want a style macro to be accessible from any experiment then you should store it in an auxiliary procedure file. See Chapter III-13, **Procedure Windows** for details.

Graph Pop-Up Menus

There are a number of contextual pop-up menus that you can use to quickly set colors and other graph properties. To bring up a contextual menu on Macintosh, hold the Control key and click or, on Windows, use the right mouse button. This is termed a contextual click.

Different contextual menus are available for clicks on traces, the interior of a graph (but not on a trace) and axes. If you hold the Shift key before a contextual click on a trace or axis, the menu will apply to all traces or axes in the graph.

You are encouraged to explore these menus.

Sometimes it is difficult to contextual click in the plot area of a graph and not hit a trace. In this case, try clicking outside the plot area (but not on an axis.)

You can be sure you are over an axis before a contextual click because the cursor will change to a two-headed arrow.

Graph Expansion

Normally, graphs are shown actual size but sometimes, when working with very small or very large graphs, it is easier to work with an expanded or contracted screen representation. You can set an expansion or contraction factor for a graph by Control-clicking (*Macintosh*) or right-clicking (*Windows*) in the graph body (away from traces or axes) and choosing a value from the Expansion menu item.

The expansion setting affects only the screen representation. The logical size for printing or exporting is not affected.

Graph Shortcuts

Action	Shortcut (*Macintosh*)	Shortcut (*Windows*)
To autoscale a graph	Press Command-A.	Press Ctrl+A.
To modify the appearance or front-to-back drawing order of a trace	Press Control and click the trace to get a contextual menu.	Right-click the trace to get a contextual menu.
	Press Shift-Control to modify all traces.	Press Shift while right-clicking to modify all traces.
	Double-click the trace to summon a dialog.	Double-click the trace to summon a dialog.
To modify the appearance of an axis, axis labels, grid lines, tick marks	Press Control and click the axis.	Right-click the axis.
	Press Shift-Control to modify all axes.	Press Shift and right-click to modify all axes.
	Double-click an axis to summon a dialog.	Double-click an axis to summon a dialog.
To modify the appearance of a contour plot	Control-click and choose Modify Contour from the contextual menu or press Shift and double-click the contour plot.	Right-click and choose Modify Contour from the contextual menu or press Shift and double-click the contour plot.
	See also Chapter II-14, **Contour Plots**.	See also Chapter II-14, **Contour Plots**.
To modify the appearance of an image plot	Control-click and choose Modify Image from the contextual menu.	Right-click and choose Modify Image from the contextual menu.
	See also Chapter II-15, **Image Plots**.	See also Chapter II-15, **Image Plots**.
To set background colors	Press Control and click in the graph body, away from any traces.	Press Ctrl and click in the graph body, away from any traces.
To set the range of an axis	Double-click tick mark labels to summon a dialog.	Double-click tick mark labels to summon a dialog.
To pan the graph	Press Option and drag the body of the graph.	Press Alt and drag the body of the graph.
	Press Shift also to constrain the direction of panning.	Press Shift also to constrain the direction of panning.
To offset a trace	Click and hold the trace for about a second, then drag. You can avoid inadvertently triggering this feature by pressing Caps Lock before clicking a trace.	Click and hold the trace for about a second, then drag. You can avoid inadvertently triggering this feature by pressing Caps Lock before clicking a trace.
To adjust the position of an axis or axis label	Drag the axis or label.	Drag the axis or label.
To change a graph margin	Press Option and click the axis and drag. Drag beyond edge of graph to return to default position.	Press Alt and click the axis and drag. Drag beyond edge of graph to return to default position.
	Press Option and double-click an axis or just double-click outside of the graph body and axes to summon a dialog.	Press Alt and double-click an axis or just double-click outside of the graph body and axes to summon a dialog.
To change an axis label	Double-click the label to summon a dialog.	Double-click the label to summon a dialog.

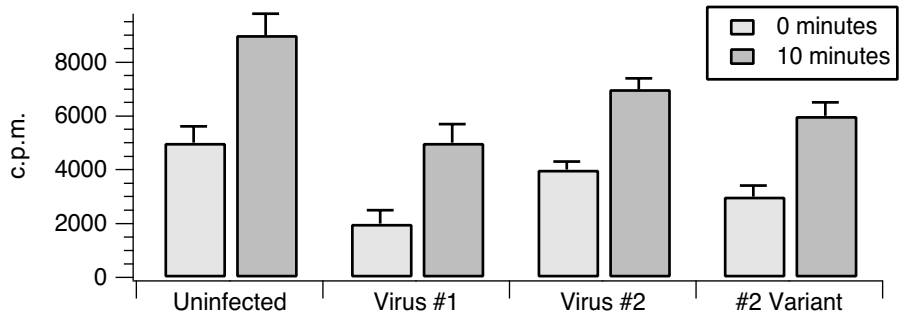
Action	Shortcut (*Macintosh*)	Shortcut (*Windows*)
To change an annotation	Double-click the annotation to summon a dialog.	Double-click the annotation to summon a dialog.
To connect a tag to a different point	Press Option and drag the tag to the new point. Don't drag the arrow or line; drag the tag body. Drag it off the graph window to remove it.	Press Alt and drag the tag to the new point. Don't drag the arrow or line; drag the tag body. Drag it off the graph window to remove it.
To attach a cursor to a particular trace	Drag the cursor from the info panel to the trace or click the cursor name in the info panel and choose a trace name from the pop-up menu.	Drag the cursor from the info panel to the trace or click the cursor name in the info panel and choose a trace name from the pop-up menu.
To get information about a particular trace	Press Command-Option-I to turn on Trace Info tags.	Press Ctrl+Alt+I to turn on Trace Info tags.
To show or hide a graph's tool palette	Press Command-T.	Press Ctrl+T.
To move or resize a user-defined control without using the tool palette	Press Command-Option and click the control. With Command-Option still pressed, drag or resize it. See also Chapter III-14, **Controls and Control Panels**.	Press Ctrl+Alt and click the control. With Ctrl+Alt still pressed, drag or resize it. See also Chapter III-14, **Controls and Control Panels**.
To modify a user-defined control	Press Command-Option and double-click the control. This displays a dialog that you use to modify all aspects of the control. If the control is already selected, you don't need to press Command-Option.	Press Ctrl+Alt and double-click the control. This displays a dialog that you use to modify all aspects of the control. If the control is already selected, you don't need to press Ctrl+Alt.
To edit a user-defined control's action procedure	With the graph in modify mode (tools showing, second icon from the top selected) press Option and double-click the control. This displays the procedure window containing the action procedure or beeps if there is no action procedure.	With the graph in modify mode (tools showing, second icon from the top selected) press Alt and double-click the control. This displays the procedure window containing the action procedure or beeps if there is no action procedure.
To nudge a user-defined control's position	Select the control and press Arrow keys. Press Shift to nudge faster.	Select the control and press Arrow keys. Press Shift to nudge faster.

Chapter
II-13

Category Plots

Overview

Category plots are two-dimensional plots with a continuous numeric variable on one axis and a nonnumeric (text) category on the other. Most often they are presented as bar charts with one or more bars occupying a category slot either side-by-side or stacked or some combination of the two. You can also combine them with error bars:



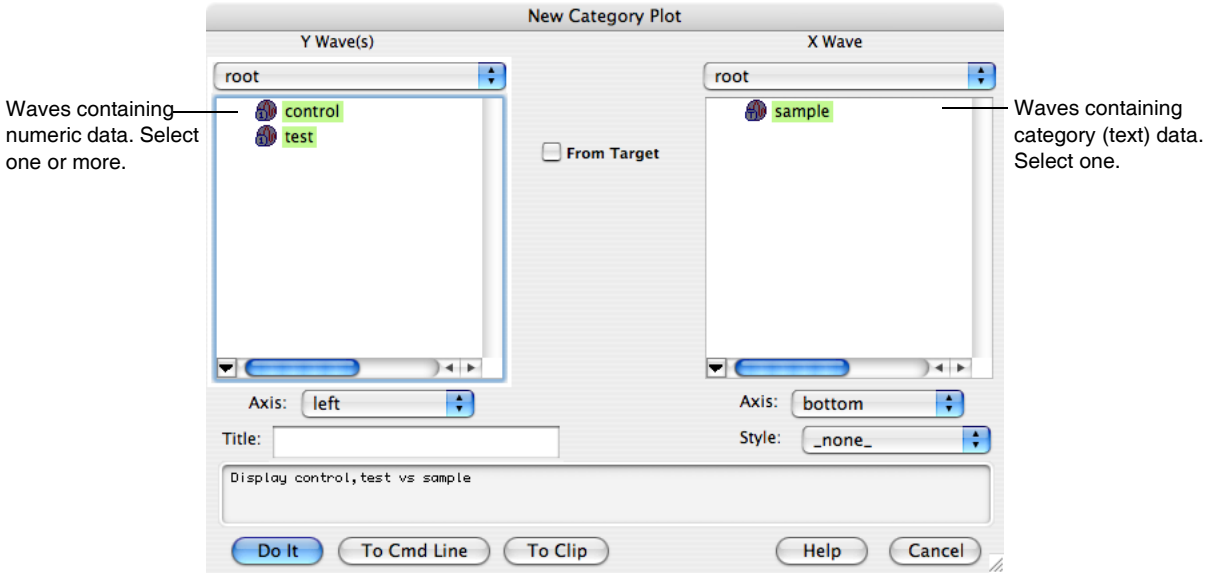
Category plots are created in ordinary graph windows when a text wave is used as the X data. For more on graphs, see Chapter II-12, **Graphs**.

Creating a Category Plot

To create a category plot, first create your numeric wave(s) and your text category wave.

Note: The numeric waves used to create a category plot should have "point scaling" (X scaling with Offset = 0 and Delta = 1). See **Category Plot Pitfalls** on page II-316 for an explanation.

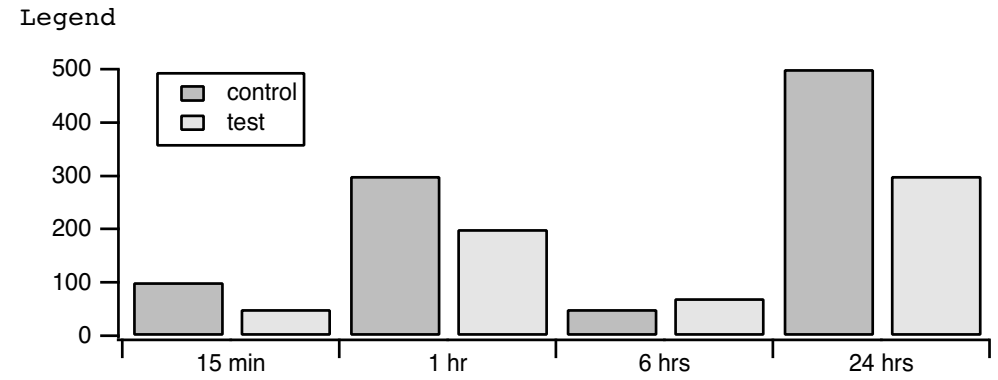
Then use the Category Plot dialog (see the New submenu of the Windows menu). You can append to an existing graph by choosing Category Plot from the Append to Graph submenu in the Graph menu. Select the numeric waves in the Y Wave(s) list, and the category (text) wave in the X Wave list:



You can use the Display command directly to create a category plot:

```
Make/O control={100,300,50,500},test={50,200,70,300}
Make/O/T sample={"15 min","1 hr","6 hrs","24 hrs"}
Display control,test vs sample   //vs text wave creates category plot
ModifyGraph hbFill(control)=5,hbFill(test )=7
```

```
SetAxis/A/E=1 left
Legend
```



Combining Category Plots and XY Plots

You can have ordinary XY plots and category plots in the same graph window. However, once an axis has been used as either numeric or category, it is not usable as the other type.

For example, if you tried to append an ordinary XY plot to the above graph you would find that the bottom (category) axis was not available in the Axis pop-up menu. If you try to append data to an existing category plot using a different text wave as the category wave, the new category wave will be ignored.

The solution to these problems is to create a new axis using the Append Traces to Graph dialog or the Append Category Plot dialog.

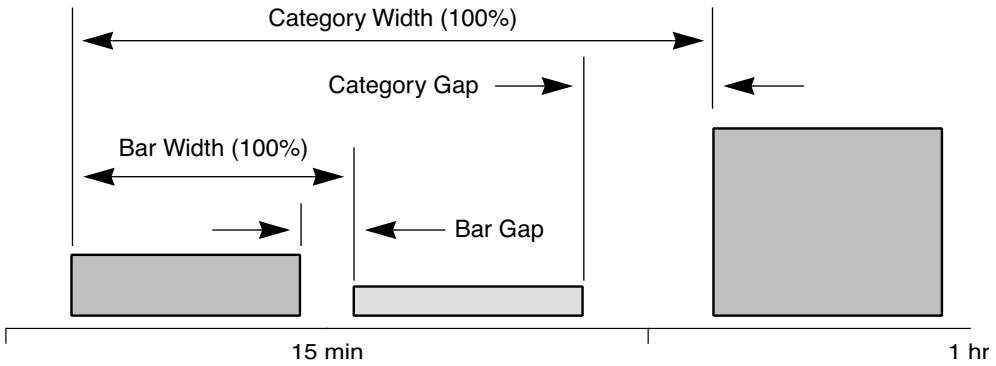
Modifying a Category Plot

Because category plots are created in ordinary graph windows, you can change the appearance of the category plot using the same methods you use for XY plots. For example, you modify the bar colors and line widths using the Modify Trace Appearance dialog. For information on traces, XY plots and graphs, see **Modifying Styles** on page II-248.

The settings unique to category plots are described in the following sections.

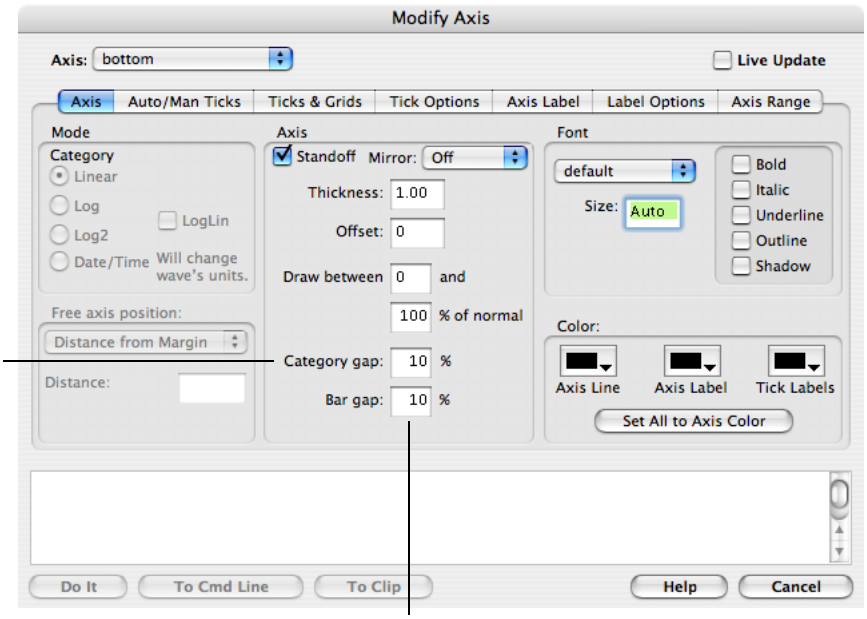
Bar and Category Gaps

You can control the gap size between bars and between categories.



Generally, the category gap should be larger than the bar gap so that it is clear which bars are in the same category. However, a category gap of 100% leaves no space for bars.

The gap sizes are set in the Modify Axis dialog (Graph menu or double-click the category axis):

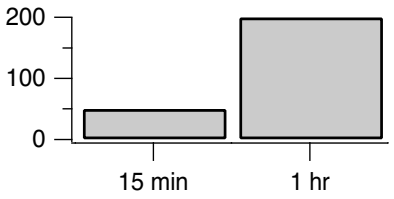


Sets the gap between the last bar in one category and the first bar in the next category as a percentage of the category width.

Sets the gap between bars within a category as a percentage of the bar width. If there is only one bar per category, then this has no effect.

Tick Mark Positioning

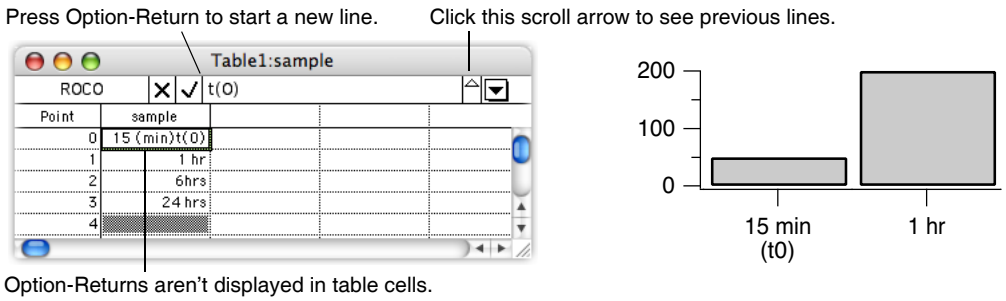
You can cause the tick marks to appear in the center of each category slot rather than at the edges. Bring up the Modify Axis dialog (Graph menu or double-click the category axis) and select the "Tick in center" checkbox in the "Auto/Man Ticks" pane. This looks best when there is only one bar per category.



Fancy Tick Mark Labels

Tick mark labels on the category axis are drawn using the contents of your category text wave. In addition to simple text, you can insert special escape codes in your category text wave to create multiline labels and to include font changes and other special effects. The escape codes are exactly the same as those used for axis labels and for annotation text boxes (see **Modifying Annotations** on page III-43). However, there is no point-and-click way to insert the codes in this version of Igor Pro. You will have to either remember the codes or use the Add Annotation dialog to create a string you can paste into a cell in a table (use the To Clip button in the dialog).

Macintosh:Creating multiline labels is easy enough; just edit your category wave in a table and press Option-Return (to insert a carriage return into the cell's text) as needed:



Multiline labels will be center-aligned on a horizontal category axis and right-aligned on a left axis but left-aligned on a right axis. You can override the default alignment using the alignment escape codes as used in the Add Annotation dialog.

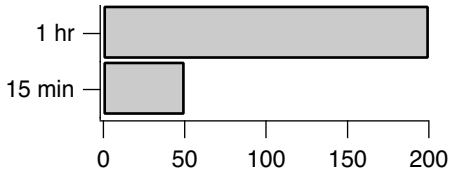
Windows: You can make category labels with more than one line. Because it is impossible on Windows to enter a return character in a table, you must make multiline labels on the command line, using "\r" to separate the lines:

```
Make/T/N=5 CatWave           // Mostly you won't need this line
CatWave[0]="Line 1\rLine2"   // "\r" Makes first label with two lines
```

Multiline labels will be center-aligned on a horizontal category axis and right-aligned on a left axis but left-aligned on a right axis. You can override the default alignment using the alignment escape codes as used in the Add Annotation dialog. See the **TextBox** operation on page V-695 for a description of the formatting codes.

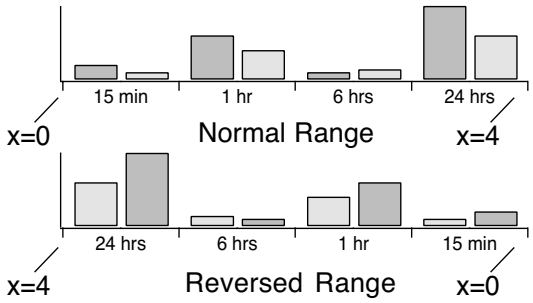
Horizontal Bars

To create a category plot in which the category axis runs vertically and the bars run horizontally, create a normal vertical bar plot and then select the Swap XY checkbox in the Modify Graph dialog (Graph menu).



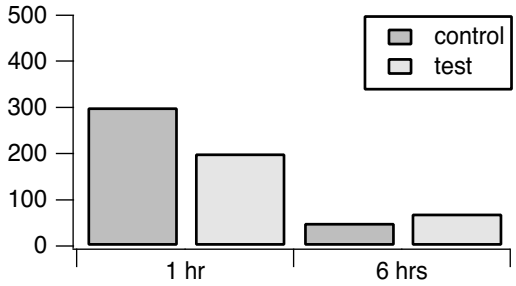
Reversed Category Axis

Although the ordering of the categories is determined by the ordering of the value (numeric) and category (text) waves, you can reverse a category axis just like you can reverse a numeric axis. Double-click one of the category axis tick labels or choose the Set Axis Range from the Graph menu to access the Axis Range pane in the Modify Axes dialog. When the axis is in autoscale mode, select the Reverse Axis checkbox to reverse the axis range.



Category Axis Range

You can also enter numeric values in the min and max value items of the Axis Range pane of the Modify Axes dialog. The X scaling of the numeric waves determine the range of the category axis. We used "point" X scaling for the numeric waves, so the numeric range of the category axis for the 15 min, 1 hr, 6 hrs, 24hrs example is 0 to 4. To display only the second and third categories, set the min to 1 and the max to 3.



Bar Drawing Order

When you plot multiple numeric waves against a single category axis you will have multiple bars within each category group. (In the examples so far, there are two bars per category group.) The order in which the bars are initially drawn is the same as the order of the numeric waves in the Display or AppendToGraph command:

```
Display control,test vs elapsed     //control on left, test on right
```

You can change the drawing order in an existing graph using the **Reorder Traces dialog** (Graph menu) and the Trace pop-up menu (see **Graph Pop-Up Menus** on page II-304).

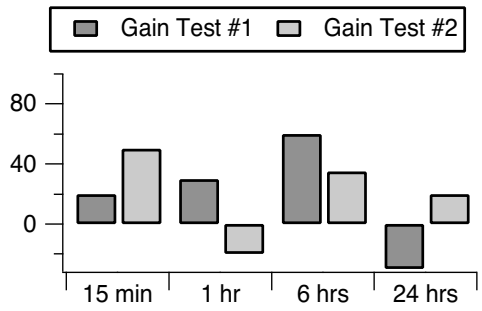
The ordering of the traces is particularly important for stacked bar charts.

Stacked Bar Charts

You can stack one bar on top of the next by choosing one of several grouping modes in the Modify Trace Appearance dialog (Graph menu or double-click in a bar). The modes are chosen from the Grouping pop-up menu in the dialog. The pop-up menu options are:

Mode	Mode Name	Purpose
-1	Keep with next	For special effects
0	None	Side-by-side bars (default)
1	Draw to next	Overlapping bars
2	Add to next	Arithmetically combined bars
3	Stack on next	Stacked bars

For most uses, you will use the **None** and "Stack on next" modes which produce the familiar bar and stacked bar charts.

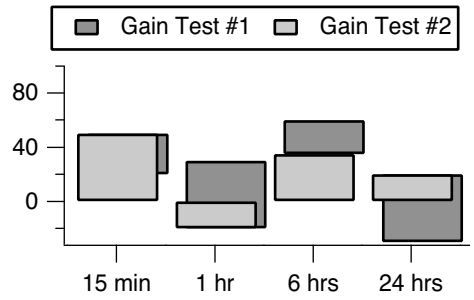


None Mode

In all of the Stacked Bar Chart examples that follow, the stacking mode is applied to the Gain Test #1 bar and Gain Test #2 is the "next" bar.

We have offset Gain Test #1 horizontally by 0.1 so that you can see what is being drawn behind Gain Test #2.

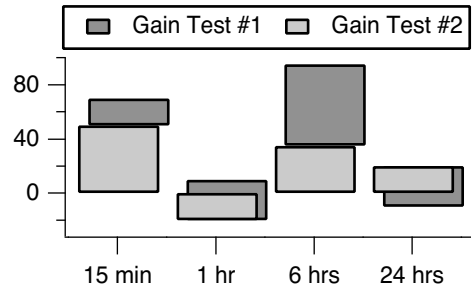
Choosing "**Draw to next**" will cause the current bar to be in the same horizontal position as the next bar and to be drawn from the y value of this trace to the Y value of the next trace.



Draw to Next Mode

If the next bar is taller than the current bar then the current bar will not be visible because it will be hidden by the next bar. The result is as if the current bar is drawn behind the next bar, as is done when bars are displayed using a common numeric X axis.

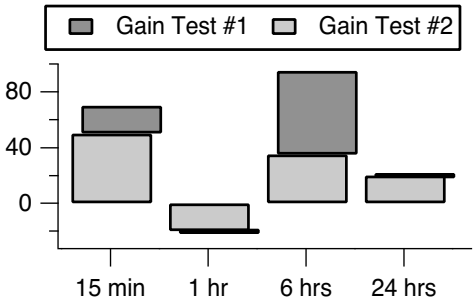
"**Add to next**" is similar to "Draw to next" except the Y values of the current bar are added to the Y values of the next bar(s) before plotting.



Add to Next Mode

If the current Y value is negative and the next is positive then the final position will be shorter than the next bar, as it is here for the 24 hrs bar.

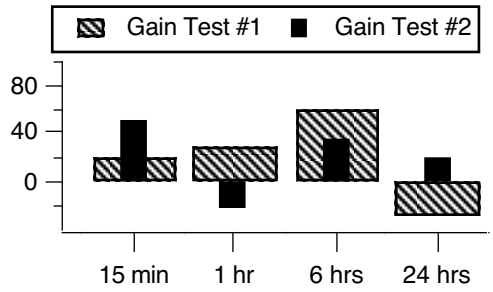
"**Stack on next**" is similar to "Add to next" except bars are allowed only to grow, not shrink.



Stack on Next Mode

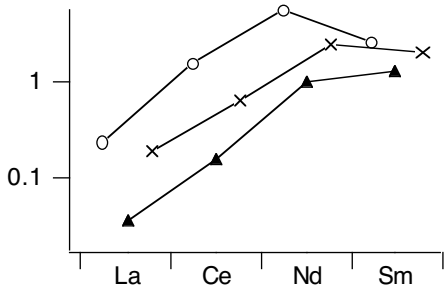
Negative values act like zero when added to a positive next trace (see the 24 hrs bar) and positive values act like zero when added to a negative next trace (see the 1 hr bar). Zero height bars are drawn as a horizontal line. Normally the values are all positive, and the bars stack additively, like the 15 min and 6 hrs bars.

"**Keep with next**" creates special effects in category plots. Use it when you want the current trace to be plotted in the same horizontal slot as the next but you don't want to affect the length of the current bar. For example, if the current trace is a bar and the next is a marker then the marker will be plotted on top of the bar. Here we set the Gain Test #2 wave to Lines from Zero mode, using a line width of 10 points.

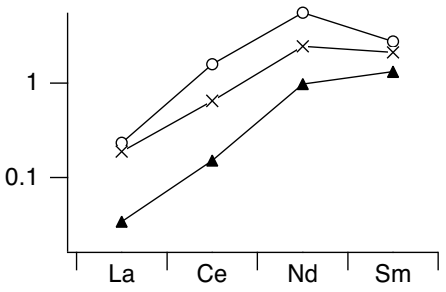


Keep with Next Mode

"Keep with next" mode is also useful for category plots that don't use bars; you can keep markers from different traces vertically aligned within the same category:



None mode



Keep with Next Mode

More details about these modes can be found in **Grouping, Stacking and Adding Modes** on page II-253.

Numeric Categories

You can create category plots with numeric categories by creating a text wave from your numeric category data. Create a text wave containing the numeric values by using the num2str function. For example, if we have years in a numeric wave:

```
Make years={1993,1995,1996,1997}
```

we can create an equivalent text wave:

```
Make/T/N=4 textYears= num2str(years)
```

Then create your category plot using textYears:

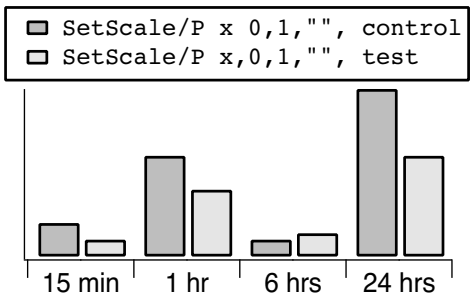
```
Display ydata vs textYears       // vs 1993, 1995, 1996, 1997 (as text)
```

Category Plot Pitfalls

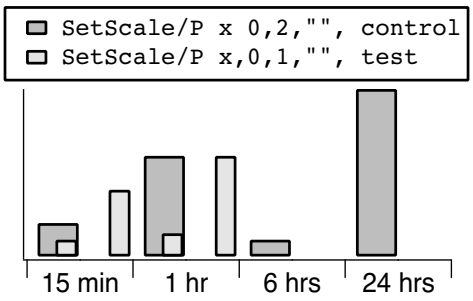
You may encounter situations in which the category plot doesn't look like you expect it to.

Pitfall: X Scaling Moves Bars

Category plots position the bars using the X scaling of the value (numeric) waves. The X scaling of the category (text) wave is completely ignored. It is usually best if you leave the X scaling of the category plot waves at the default "point scaling." In any event, the X scaling of the value (numeric) waves should be identical. Differing X scaling will separate the bars in category plots containing multiple bars per category. In the graph on the right the numeric waves have different X scaling



Same X scaling



Different X scaling

Pitfall: Changing the Drawing Order Breaks Stacked Bars

Stacked bar charts are heavily dependent on the concept of the "current bar" and the "next bar". The modes describe how the current bar is connected to the next bar, such as "Stack on next".

If you change the drawing order of the traces, using the Reorder Traces dialog or the Trace pop-up menu, one or more bars will have a new "next bar" (a different trace than before). Usually this means that a bar

will be stacking on a different bar. This is usually a problem only when the stacking modes of the traces differ, or when smaller bars become hidden by larger bars.

After you change the drawing order, you may have to change the stacking mode(s). Bars hidden by larger bars may have to be moved forward in the drawing order with the Reorder Traces dialog or the Trace pop-up menu.

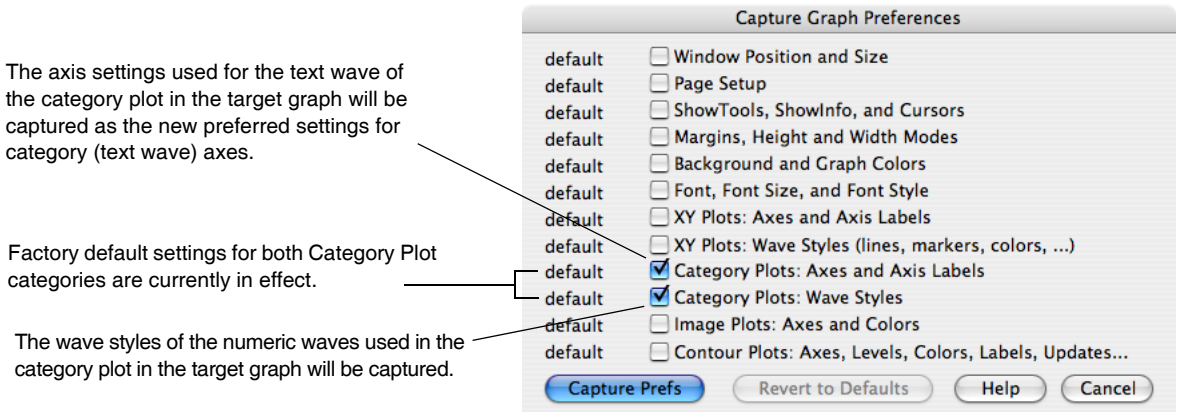
Pitfall: Bars Disappear with "Draw to next" Mode

In "Draw to next" mode, if the next bar is taller than the current bar then the current bar will not be visible because it will be hidden by the next bar.

You can change the drawing order with the Reorder Traces dialog or the Trace pop-up menu to move the shorter bars forward in the drawing order, so they will be drawn in front of the larger bars.

Category Plot Preferences

You can change the default appearance of category plots by "capturing" preferences from a "prototype" graph containing category plots. Create a graph containing a category plot (or plots) having the settings you use most often. Then choose Capture Graph Prefs from the Graph menu. Select the Category Plots categories, and click Capture Prefs.



Preferences are normally in effect only for *manual* operations, not for automatic operations from Igor procedures. This is discussed in more detail in Chapter III-17, **Preferences**.

Category Plot Axes and Axis Labels

When creating category plots with preferences turned on, Igor will use the Category Plot axis settings for the text wave axis *and XY plot axis settings for the numeric wave axis.*

Only axes used by category plot *text waves* have their settings captured. Axes used solely for an XY plot, image plot, or contour plot are ignored. *Usually this means that only the bottom axis settings are captured (the left axis is usually a numeric wave, and is therefore an XY Plot axis).*

The category plot axis preferences are applied only when axes having the same name as the captured axis are created by a Display or AppendToGraph operation when creating a category plot. If the axes existed before the operation is executed, they will not be affected by the category plot axis preferences.

The names of captured category plot axes are listed in the X Axis pop-up menu of the New Category Plot and Append Category Plot dialogs.

For example, suppose you capture preferences for a category plot that was created with the command:

```
AppendToGraph/R=myRightAxis/T=myTopAxis ywave vs textwave
```

Since only the X axis is a category axis, "myTopAxis" will appear in the X Axis pop-up menu in the category plot dialogs. The Y Axis pop-up menu will be unaffected.

- If you choose "myTopAxis" in the X Axis pop-up menu of the New Category Plot dialog and click Do It, a graph will be created containing a *newly-created* X axis named "myTopAxis" and having the axis settings you captured.
- If you have a graph which already uses an axis named "myTopAxis" *as a category axis* and you choose it from the X Axis pop-up menu in the Append Category Plot dialog, the category plot will use the axis, but no captured axis settings will be (re)applied to it. (Of course, the preferred settings were probably applied when the axis was created in the first place.)

You can capture category plot axis settings for the standard left or bottom axis, and Igor will save the settings separately from left and bottom axis preferences captured for XY, image, and contour plots.

Remember, when creating category plots, Igor will use the category plot axis settings for the text wave axis and XY plot axis settings for the numeric wave axis.

Category Plot Wave Styles

The captured category plot wave styles are automatically applied to a category plot when it is first created (provided preferences are turned on; see **How to Use Preferences** on page III-430). "Wave styles" refers to the various trace-specific settings for category plot numeric waves in the graph. The settings include trace mode, line style, stacking mode, fill pattern, colors, etc., as set by the Modify Trace Appearance dialog.

If you capture the category plot preferences from a graph with more than one category plot, the first category plot appended to a graph gets the wave style settings from the category first appended to the prototype graph. The second category plot appended to a graph gets the settings from the second category plot appended to the prototype graph, etc. This is similar to the way XY plot wave styles work.

How to Use Category Plot Preferences

Here is our recommended strategy for using category preferences:

1. Create a new graph containing a single category plot. Use the axes you will normally use, even if they are left and bottom. You can use other axes, too (choose New Axis in the Category Plot dialogs).
2. Use the Modify Trace Appearance dialog and the Modify Axes dialogs to make the category plot appear as you prefer.
3. Choose Capture Graph Prefs from the Graph menu. Select the Category Plot checkboxes, and click Capture Prefs.

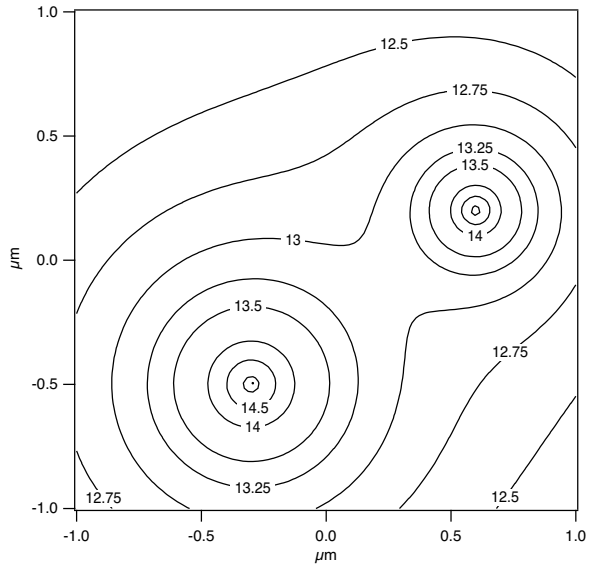
Chapter
II-14

Contour Plots

Overview

A contour plot is a two-dimensional XY plot of a three-dimensional XYZ surface showing lines where the surface intersects planes of constant elevation (Z).

One common example is a contour map of geographical terrain showing lines of constant altitude, but contour plots are also used to show lines of constant density or brightness, such as in X-ray or CT images, or to show lines of constant gravity or electric potential.



Contour Data

The contour plot is appropriate for data sets of the form:

$$z = f(x,y)$$

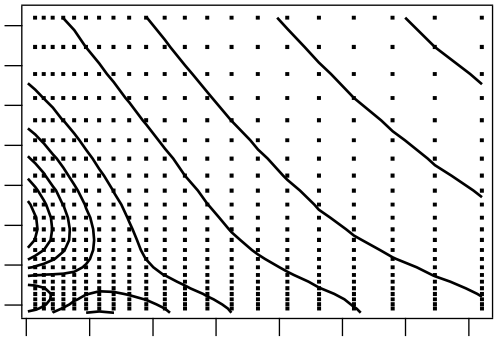
meaning there is only one Z value for each XY pair. This rules out 3D shapes such as spheres, for example.

You can create contour plots from two kinds of data:

- gridded data stored in a matrix
- XYZ triplets

Gridded Data

Gridded data is stored in a 2D wave, or "matrix wave". By itself, the matrix wave defines a regular XY grid. The X and Y coordinates for the grid lines are set by the matrix wave's row X scaling and column Y scaling.



You can also provide optional 1D waves that specify coordinates for the X or Y grid lines, producing a non-linear rectangular grid like the one shown here (the dots mark XY coordinates specified by the 1D waves).

Contouring gridded data is computationally much easier than XYZ triplets and consequently much faster.

XYZ Data

XYZ triplets may be stored in a matrix wave of three columns, or in three separate 1D waves each supplying X, Y, or Z values. You must use this format if your Z data does not fall on a rectangular grid. For example, you can use this format for data on a circular grid or for Z values at random X and Y locations.

For best results, you should avoid having multiple XYZ triplets with the same X and Y values. If the contour data is stored in separate waves, the waves should be the same length.

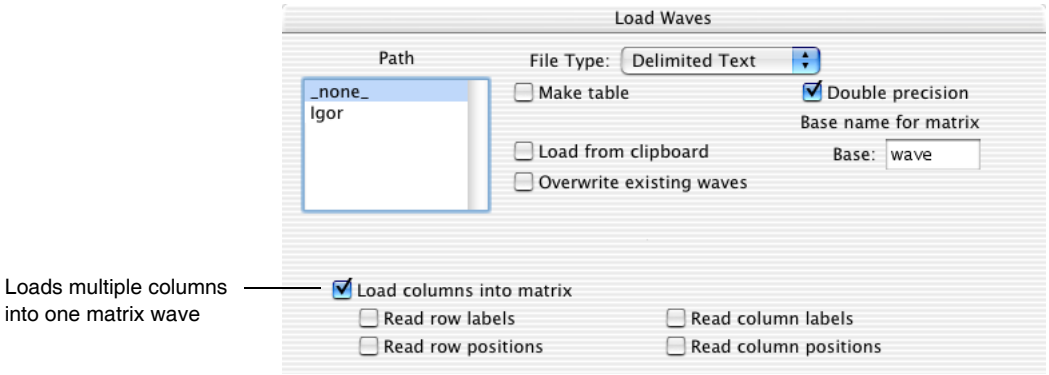
If your data is in XYZ form and you want to convert it to gridded data, you can use the **ContourZ** function on an XYZ contour plot of your data to produce a matrix wave. The AppendImageToContour procedure in the WaveMetrics Procedures folder produces such a matrix wave, and appends it to the top graph as an image. Also see the Voronoi parameter of the **ImageInterpolate** operation on page V-264 for generating an interpolated matrix.

XYZ contouring involves the generation of a Delaunay Triangulation of your data, which takes more time than is needed for gridded contouring. You can view the triangulation by selecting the Show Triangulation checkbox in the Modify Contour Appearance dialog.

Creating a Contour Plot

The first step in creating a contour plot is to create or load the data to be contoured.

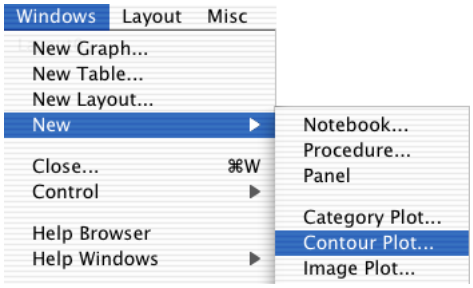
Gridded (matrix) data or a 3-column matrix of XYZ triples can be loaded using the Load Waves dialog by selecting the "Load columns into matrix" checkbox.



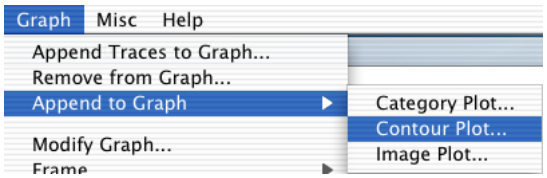
Leave the "Load columns into matrix" checkbox deselected to load columns into separate XYZ waves. You can also load X, Y, and Z waves separately from different files.

Contour plots are appended to ordinary graph windows. All the features of graphs apply to contour plots: axes, line styles, drawing tools, controls, etc. See Chapter II-12, **Graphs**.

You can create a contour plot in a new graph window with the New Contour Plot dialog. This dialog creates a blank graph to which the plot is appended. Add a contour plot to an existing graph with the Append Contour Plot dialog.



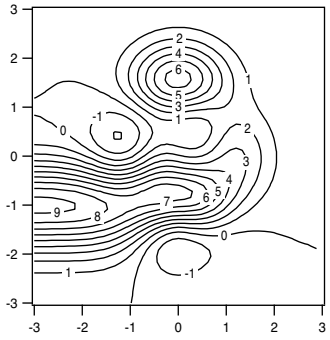
Creating a New Contour Plot



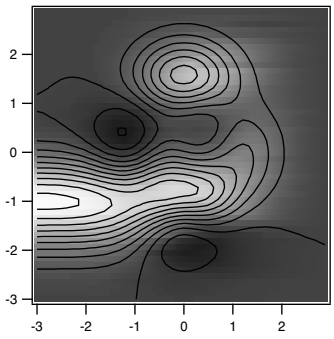
Appending a Contour Plot to an Existing Graph

You can also use the **AppendMatrixContour** (see page V-24) or **AppendXYZContour** (see page V-27) operations:

```
Make/O/D/N=(50,50) mat2d   //make some data to plot
SetScale x,-3,3,mat2d
SetScale y,-3,3,mat2d
mat2d =  3*(1-y)^2 * exp(-((x*(x>0))^2.5) - (y+0.5)^2)
mat2d -= 8*(x/5 - x^3 - y^5) * exp(-x^2-y^2)
mat2d -= 1/4 * exp(-(x+.5)^2 - y^2)
Display;AppendMatrixContour mat2d   //This creates the contour plot
AppendImage mat2d                   //Adds a sense of depth
```



`AppendMatrixContour mat2d`

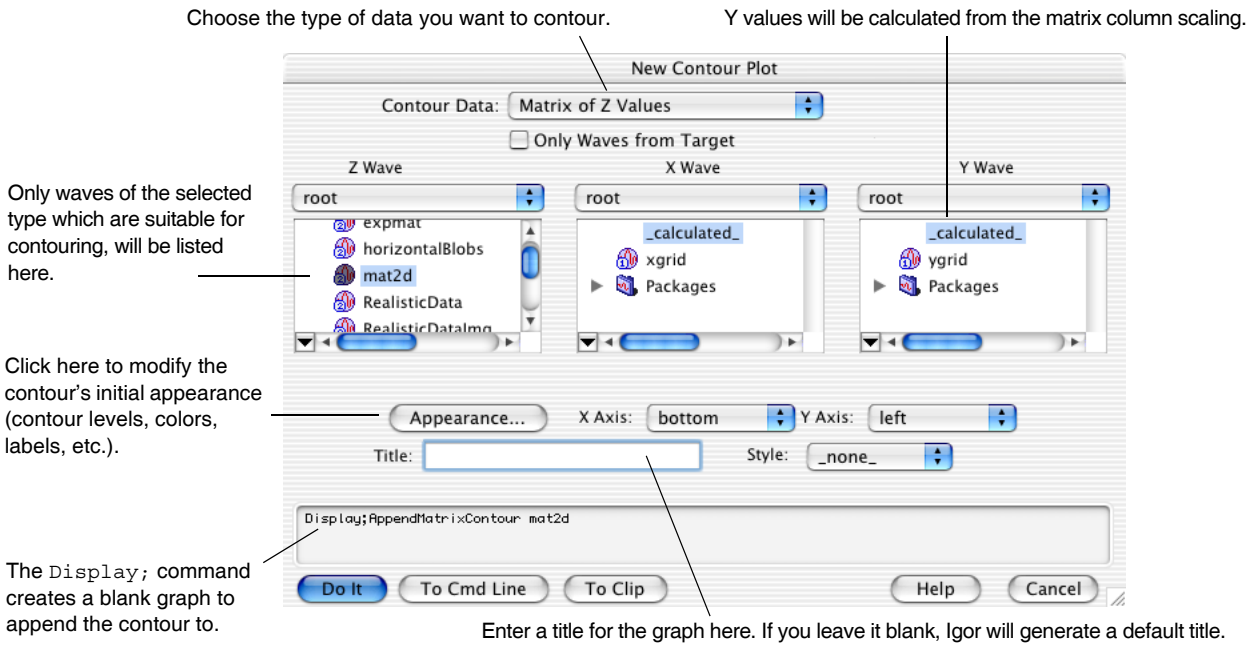


`AppendMatrixContour mat2d`
`AppendImage mat2d`

You can add a sense of depth to a matrix (gridded data) contour plot by using **The Append Image Dialogs** to add an image plot to the same graph. If your data is in XYZ form, use the AppendImageToContour procedure (in the WaveMetrics Procedures folder) to create and append an image. For more on image plots, see Chapter II-15, **Image Plots**.

The Contour Plot Dialogs

Choose New Contour Plot from the Windows menu to bring up the New Contour Plot dialog.

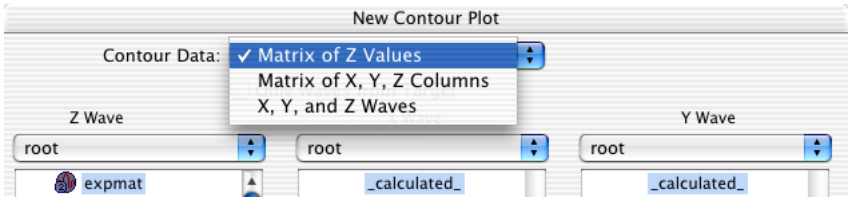


The Title, Style, X Axis and Y Axis items work the same as in the New Graph dialog (see **Creating Graphs** on page II-237). You can specify a new title for the graph and select or create the axes used in the contour plot. Use the Style pop-up menu to apply a style macro to the newly created graph window.

The Append Contour Plot dialog is similar, except that the Only Waves from Target, Title, and Style items are missing.

Contour Data Pop-Up Menu

The first thing you should do when using this dialog is to choose the type of data you want to contour using the Contour Data pop-up menu:



This limits the waves displayed in the Z Wave list to that particular kind of data. The pop-up menu also determines whether the AppendMatrixContour or AppendXYZContour operation is used to generate the final command.Select the Only Waves from Target checkbox to show only the waves in the target window (most useful when the target window is a table).

X, Y, and Z Wave Lists

The second thing you should do is choose from the Z Wave list the data you want to contour. This limits the choices shown in the X Wave and Y Wave lists to those which can be combined with that Z wave and the selected contour operation.

When "Matrix of Z Values" is chosen from the Contour Data pop-up menu, choosing _calculated_ from the X Wave list generates X coordinates from the row X scaling of the matrix selected in the Z Wave list. Choosing _calculated_ from the Y Wave list uses the column Y scaling to provide Y coordinates.

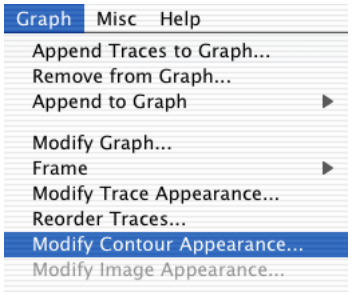
You can also select a 1D wave to provide the X or Y coordinates for a matrix of Z values; only those waves with the proper length for the selected Z Wave are shown in the X Wave and Y Wave lists. For details, see the **AppendMatrixContour** operation on page V-24 and **AppendXYZContour** operation on page V-27.

When "Matrix of X, Y, Z Columns" is chosen from the Contour Data pop-up menu, the X Wave and Y Wave lists aren't necessary and are hidden. Igor expects the matrix selected in the Z Wave list to have X coordinates in the first column, Y in the second, and Z in the third.

When "X, Y, and Z Waves" is selected, the X Wave and Y Wave lists are updated whenever a new Z wave is selected so that only waves with matching lengths are shown.

Modifying a Contour Plot

You can change the appearance of the contour plot using the Modify Contour Appearance dialog. This dialog is also available as a subdialog of the New Contour Plot dialog.

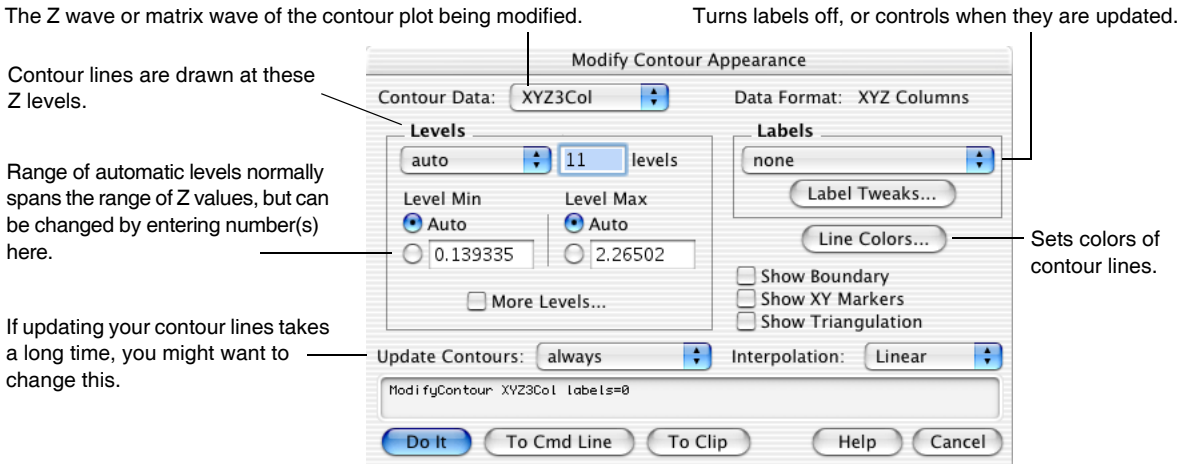


Tip #1 You can open the Modify Contour Appearance dialog by Shift-double-clicking the plot area of a graph. The graph must contain a contour plot for this to work.

Tip #2 Use the preferences to change the default contour appearance, so you won't be making the same changes over and over. See **Contour Preferences** on page II-338.

Another way to open the dialog is by Control-clicking (*Macintosh*) or right-clicking (*Windows*) in the plot area and choosing Modify Contour from the resulting pop-up menu.

Modify Contour Appearance Dialog



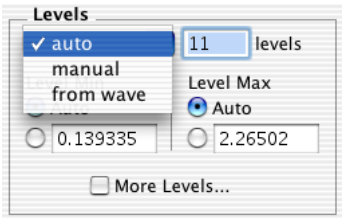
Contour Data Pop-Up Menu

The Contour Data pop-up menu shows the "contour instance name" of the contour plot being modified. The name of the contour plot is the same as the name of the Z wave containing the contour data.

If the graph contains more than one contour plot, you can use this pop-up menu to change all contour plots in the target graph.

If the graph contains two contour plots *using the same Z wave name*, an instance number is appended to those Z wave names in this pop-up menu. See **Instance Notation** on page IV-16, and **Contour Instance Names** on page II-334.

Levels



Each contour trace draws lines at one constant Z level. The Z levels are assigned automatically or manually as specified in this part of the dialog.

Igor computes automatic levels by subdividing the range of Z values into approximately the number of requested levels. You can instruct Igor to compute the Z range automatically from the minimum and maximum of the Z data, or to use a range that you specify in the dialog. Igor attempts to choose "nice" contour levels that minimize the number of significant digits in the contour labels. To achieve this, Igor may create more or fewer levels than you requested.

You can specify manual levels directly in the dialog in several ways:

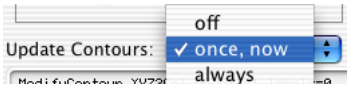
- Linearly spaced levels (constant increment) starting with a first level and incrementing by a specified amount.
- A list of arbitrary levels stored in a wave you choose from a pop-up Wave Browser.
- A list of arbitrary levels you enter in the dialog that appears when you select the More Levels checkbox. These levels are *in addition to* automatic, manual, or from-wave levels.

The More Levels dialog can be used for different purposes:

- To add a contour level to those already defined by the automatic, manual, or from-wave levels. You might do this to indicate a special feature of the data.
- As the only source of arbitrary contour levels, for complete control of the levels. You might do this to slightly change a contour level to avoid problems (see **Contouring Pitfalls** on page II-337). Disable the auto or manual levels by entering 0 for the number of levels. The only contour levels in effect will be those entered in the More Levels dialog.

In the future, we may add a button in the More Levels dialog to copy the automatic or manual levels into the More Levels dialog and automatically disable the automatic or manual levels. The Contour Levels WaveMetrics procedures will currently help in this regard (look in the WM Procedures Index help file for details).

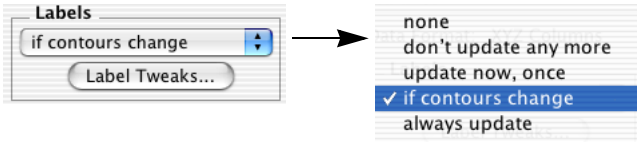
Update Contours Pop-Up Menu



Igor normally recalculates and redraws the contour plot whenever any change occurs that might alter its appearance. This includes changes to any data waves and the wave supplying contour levels, if any. Since calculating the contour lines can take a long time, you may want to disable this automatic update with the Update Contours pop-up menu.

"Off" completely disables updating of the contours for any reason. Choose "once, now" to update the contour when you click the Do It button. "Always" selects the default behavior of updating whenever the contour levels change.

Labels Pop-Up Menu



Igor normally adds labels to the contour lines, and updates them whenever the contour lines change (see **Update Contours Pop-Up Menu** on page II-326). Since updating plots with many labels can take a long time, you may want to disable or modify this automatic update with the Labels pop-up menu.

"None" removes any existing contour labels, and prevents any more from being generated.

"Don't update any more" keeps any existing labels, and prevents any more updates. This is useful if you have moved, removed, or modified some labels and you want to keep them that way.

"Update now, once" will update the labels when you click the Do It button, then prevents any more updates. Use this if updating the labels takes too long for you to put up with automatic updates.

"If contours change", the default, updates the labels whenever Igor recalculates the contour lines.

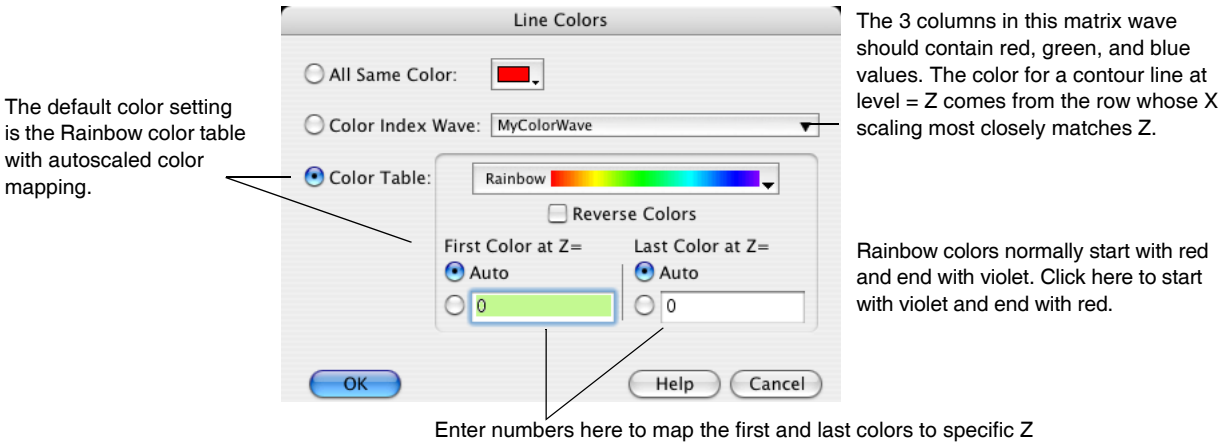
"Always update" is the most aggressive setting. It updates the labels if the graph is modified in almost any way, such as changing the size of the graph or adjusting an axis. You might use this setting temporarily while making adjustments that might otherwise cause the labels to overlap or be too sparse.

Click Label Tweaks to change the number format and appearance of the contour labels with the Contour Labels dialog. See **Modifying Labels** on page II-336.

For more than you ever wanted to know about contour labels, see **Contour Labels** on page II-335.

Line Colors Button

Click the Line Colors button to assign colors to the contour lines according to their Z level, or to make them all the same color:



Autoscaled color mapping assigns the first color in a color table to the minimum Z value of the contour data (not to the minimum contour level), and the last color to the maximum Z value.

For details about the color index wave, see **The Color of Contour Traces** on page II-329.

You can override the color set by this dialog with the **Modify Trace Appearance Dialog**. Also see **Overriding the Color of Contour Traces** on page II-332.

Show Boundary Checkbox

Click this to generate a trace along the perimeter of the contour data in the XY plane. For a matrix contour, the perimeter is simply a rectangle enclosing the minimum and maximum X and Y. The perimeter of XYZ triplet contours connects the outermost XY points. This trace is updated at the same time as the contour level traces.

Show XY Markers Checkbox

Click this to generate a trace that shows the XY locations of the contour data. For a matrix contour, the locations are by default marked with dots; for XYZ triplet contours they are shown using markers. As with any other contour trace, you can change the mode and marker of this trace with the Modify Trace Appearance dialog. This trace is updated at the same time as the contour level traces.

Show Triangulation Checkbox

Click this to generate a trace that shows the Delaunay triangulation of the contour data. This is available for only XYZ triplet contours. This trace is updated at the same time as the contour level traces.

Interpolation Pop-Up Menu

XYZ triplet contours can be interpolated to increase the apparent resolution, resulting in smoother contour lines. The interpolation uses the original Delaunay triangulation. Increasing the resolution requires more time and memory; settings higher than x16 are recommended only to the very patient.

All About Contour Traces

Igor Pro creates XY pairs of double-precision waves to contain the contour trace data, and displays them as ordinary graph traces. Each trace draws all the curves for one Z level. If a single Z level generates more than one contour line, Igor uses a blank (NaN) at the end of each contour line to create a gap between it and the following line.

The same method is used to display markers at the data's XY coordinates, the XY domain's boundary, and (for XYZ triplet contours only) the Delaunay triangulation.

The names of these traces are fabricated from the name of the Z data wave or matrix. See **Contour Trace Names** on page II-328.

One important special property of these waves is that they are private to the graph. These waves will not show up in any other dialog and are not accessible from the command line. There is a trick you can use to copy these waves, however. See **Extracting Contour Trace Data** on page II-334.

The contour traces (which are the visible manifestation of these waves), *do* show up in the Modify Trace Appearance dialog, and *can* be named in commands just like other traces.

There is often no need to bother with the individual traces of a contour plot because the Modify Contour Appearance dialog provides adequate control over the traces for most purposes. However, if you want to distinguish one or more contour levels (to make them dashed lines, for example) you can do this by modifying the traces using the Modify Trace Appearance dialog. Also see **Overriding the Color of Contour Traces** on page II-332.

Contour Trace Names

The name of a contour trace is usually something like "zwave=2.5", indicating that the trace is the contour of the z data set "zwave" at the z=2.5 level. A name like this must be enclosed in single, noncurly quotes when used in a command:

```
ModifyGraph mode('zwave=2.5')=2
```

This trace naming convention is the default, but you can create a different naming convention when you first append a contour to a graph by adding the proper /F parameters to AppendMatrixContour or AppendXYZContour commands.

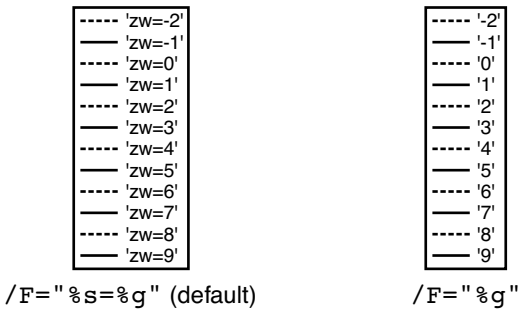
The contour dialogs do not generate these /F commands, but you can add them to commands the dialogs generate by using the To Cmd Line button instead of the Do It button.

See the **AppendMatrixContour** operation on page V-24 and the **AppendXYZContour** operation on page V-27 for a more thorough discussion of /F.

Example: Contour Legend with Numeric Trace Names

To make a legend contain slightly nicer wording you can omit the "zwave=" portion from trace names with `/F="%g"`:

```
AppendMatrixContour/F="%g" zw    // trace names become just numbers
```



Note: You can manually edit the legend text to remove the single quotes around the trace names. Double-click the legend to bring up the Modify Annotation dialog.

For details on creating contour plot legends, see **Legends** on page II-335.

Programming Notes

The **TraceNameList** function returns a string which contains a list of the names of contour traces in a graph. You can use the name of the trace to retrieve a "wave reference" to the values in the trace with the **TraceNameToWaveRef** (see page V-711) . See also **Extracting Contour Trace Data** on page II-334.

If a graph happens to contain two traces with the same name, "instance notation" uniquely identifies them. For example, two traces named "2.5" would show up in the Modify Trace Appearance dialog as "2.5" and "2.5#1". On the command line, you would use something like:

```
ModifyGraph mode('2.5'#1)=2
```

Notice how the instance notation (the "#1" part) is outside the single quotes. This instance notation is needed when the graph contains two contour plots that generate identically named traces, usually when they use the same /F parameters and draw a contour line at the same level (z=2.5).

See **Instance Notation** on page IV-16. Also see **Contour Instance Names** on page II-334.

The Color of Contour Traces

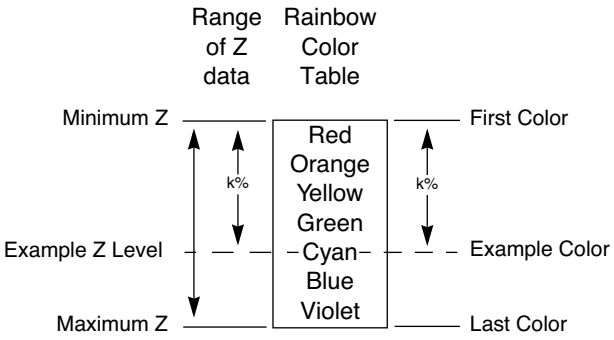
By default, contour traces are assigned a color from the Rainbow color table based on the contour trace's Z level. You can choose different colors in the Line Colors subdialog of the Modify Contour Appearance dialog.

The Line Colors dialog provides three choices for setting the contour trace colors:

1. All contour traces can be set to the same color.
2. The color for a trace can be selected from a "color index wave" (that you must create) by matching the trace's Z level with the color index wave X index.
3. The color for a trace can be computed from a chosen built-in color table by matching the trace's Z level with the range of available colors.

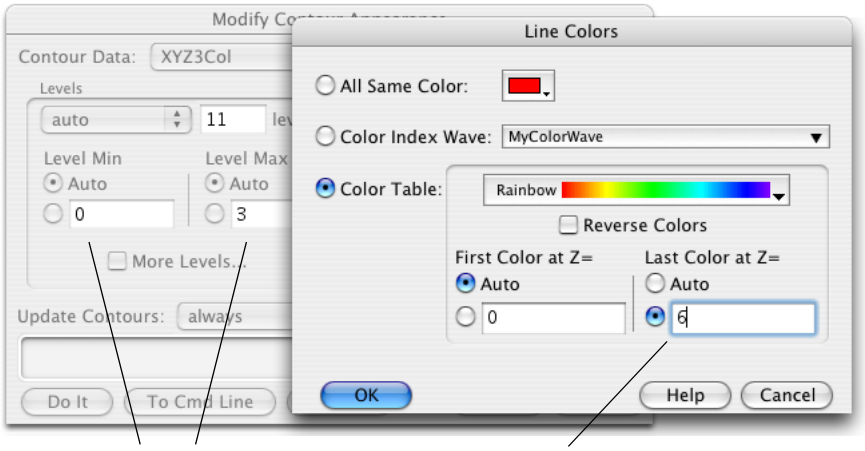
Color Tables

When you use a built-in color table to supply colors, Igor maps the Z level to an entry in the color table. By default, Igor linearly maps the entire range of Z levels to the range of colors in the table by assigning the minimum Z value of the contour data to the first color in the table, and the maximum Z value to the last color:



Automatic Mode Color Table Mapping

With the Modify Contour Appearance Dialog, you can assign a specific Z value to the color table's first and last colors. For example, you can use the first half of the color table by leaving First Color on Auto, and typing a larger number (2* Z Max -Z Min, actually) for the Last Color.



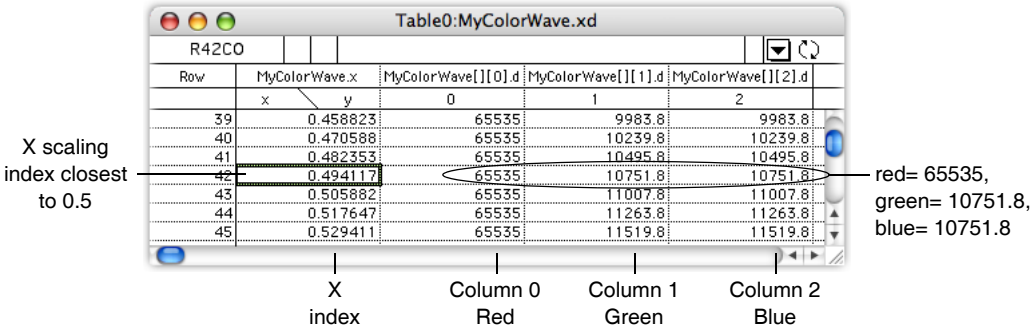
Auto Levels mode shows the minimum and maximum Z values of the contour data here.

Enter 2 * maximum Z - minimum Z to use only the first half of the color table.

Using the First Half of a Color Table

Color Index Wave

You can create your own range of colors by creating a color index wave. The wave must be a 2D wave with three columns containing red, green, and blue values that range from 0 (zero intensity) to 65535 (full intensity), and a row for each color. Igor finds the color for a particular Z level by choosing the row in the color index wave whose X index most closely matches the Z level:



Choosing a color for Z contour level = 0.5

To choose the row, Igor converts the Z level into a row number as if executing:

```
colorIndexWaveRow= x2pnt(colorIndexWave,Z)
```

which rounds to the nearest row and limits the result to the rows in the color index wave.

When the color index wave has default X scaling (the X index is equal to row number), then row 0 contains the color for z=0, row 1 contains the color for z=1, etc. By setting the X scaling of the wave (Change Wave Scaling dialog), you can control how Igor maps Z level to color. This is similar to setting the First Color and Last Color values for a color table.

Color Index Wave Programming Notes

Looking up a color in a color index wave can be expressed programmatically as:

```
red=   colorIndexWave (Z level)[0]
green= colorIndexWave (Z level)[1]
blue=  colorIndexWave (Z level)[2]
```

where ( ) indexes into rows using X scaling, and [ ] selects a column using Y point number (column number).

Here is a code fragment that creates a color index wave that varies from blue to red:

```
Function BlueRedColorIndexWave(numberOfColors,zMin,zMax)
   Variable numberOfColors
   Variable zMin,zMax      // from min, max of contour or image data

   Make/O/N=(numberOfColors,3) myColors
   Variable white=65535          // black is zero
   Variable colorStep= white/(numberOfColors-1)

   myColors[][0]= colorStep*p    // red increases with row number,
   myColors[][1]= 0              // no green
   myColors[][2]= colorStep*(numberOfColors-1-p)  // blue decreases

   SetScale/I x,zMin,zMax,myColors  // Match X scaling to Z range
End
```

Log Color for Contour Traces

In Igor Pro 6.22, the ability to map the color table and color index colors in a logarithmic fashion was added with the `ModifyContour logLines=1` command, which is controlled by the Log Color checkbox in the Line Colors dialog. When checked, the colors change more rapidly at smaller contour z level values than at larger values.

For a color index wave, the colors are mapped using the log(color index wave's x scaling) and log(contour z level) values this way:

```
colorIndexWaveRow = (nRows-1)*(log(Z)-log(xMin))/(log(xmax)-log(xMin))
```

where,

```
nRows = DimSize(colorIndexWave,0)
xMin = DimOffset(colorIndexWave,0)
xMax = xMin + (nRows-1) * DimDelta(colorIndexWave,0)
```

The colorIndexWaveRow value is rounded before it is used to select a color from the color index wave.

A similar mapping is performed with color tables, where the xMin and xMax are replaced with the automatically determined or manually provided zMin and zMax values.

Overriding the Color of Contour Traces

You can override the color set by the Line Color subdialog by using the **Modify Trace Appearance** dialog, the **ModifyGraph** command, or by Control-clicking (*Macintosh*) or right-clicking (*Windows*) in the graph's plot area to pop up the **Trace Pop-Up Menu**. The color you choose will continue to be used until either:

1. The trace is removed when the contours are updated (because the levels changed, for instance),
2. Or you choose a new setting in the Line Color subdialog.

Removing Contour Traces from a Graph

Removing traces from a contour plot with the RemoveFromGraph operation or the Remove from Graph dialog will work only temporarily. As soon as Igor updates the contour traces, any removed traces may be replaced.

You can prevent this replacement by disabling contour updates with the Modify Contour Appearance dialog. It is better, however, to use the Modify Contour Appearance dialog to control which traces are drawn in the first place.

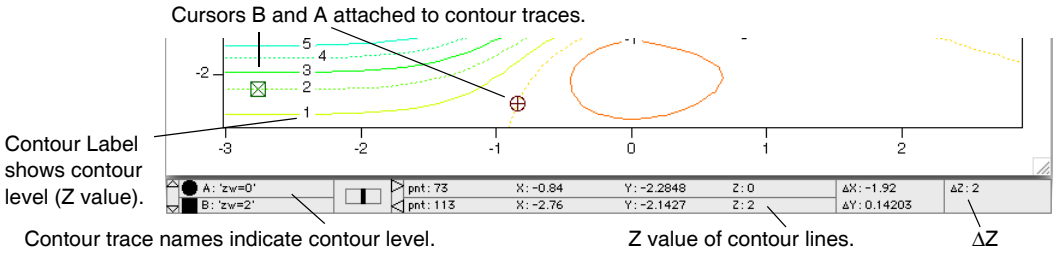
To permanently remove a particular automatic or manual contour level, you are better off not using manual levels or automatic levels at all. Use the **More Levels** dialog to explicitly enter all the levels, and enter zero for the number of manual or automatic levels.

Tip: Make a legend in the graph to list the contour traces. The trace names normally contain the contour levels. To do this, select the command window, type `Legend`, and press Return or Enter.

Similarly, if you don't want the triangulation or XY marker traces to be drawn, use the Modify Contour Appearance dialog to turn them off, rather than removing them with the Remove from Graph dialog.

Cursors on Contour Traces

You can attach cursors to a contour trace. Just like any other trace, the X and Y values are shown in the graph info panel (choose Show Info from the Graph menu). When the cursor is attached to a contour line the Z value (contour level) of the trace is also shown in the info panel:



There are several additional methods for displaying the Z value of a contour trace:

- The zcsr function returns the Z value of the trace the cursor is attached to. Zcsr returns a NaN if the cursor is attached to a noncontour trace. You can use this in a macro, or print the result on the command line using: `Print zcsr(A)`.
- If you add an image behind the contour, you can use cursors to display the X, Y, and Z values at any point.
- The name of the trace the cursor is attached to shows up in the info panel, and the name usually contains the Z value.
- Contour labels show the Z value. Contour labels are tags that contain the \OZ escape code or TagVal(3). See **Contour Labels** on page II-335. You can drag these labels around by holding down Option (*Macintosh*) or Alt (*Windows*) to change the tag attachment point. See **Changing a Tag's Attachment Point** on page III-57.

Contour Trace Updates

Igor normally updates the contour traces whenever the contour data (matrix or XYZ triplets) changes or whenever the contour levels change. Because creating the contour traces can be a lengthy process, you can prevent these automatic updates through a pop-up menu item in the Modify Contour Appearance dialog. See **Update Contours Pop-Up Menu** on page II-326.

Preventing automatic updates can be useful when you are repeatedly editing the contour data in a table or from a procedure. Use the "once, now" pop-up item to manually update the traces.

Programming Note

Programmers should be aware of another update issue: contour traces are created in two steps. To understand this, look at this graph recreation macro that appends a contour to a graph and then defines the contour levels, styles and labels:

```
Window Graph1() : Graph
   PauseUpdate; Silent 1      // building window...
   Display /W=(11,42,484,303)
   AppendMatrixContour zw
   ModifyContour zw autoLevels={*,*,5}, moreLevels={0.5}
   ModifyContour zw rgbLines=(65535,0,26214)
   ModifyContour zw labels=0
   ModifyGraph lSize('zw=0')=3
   ModifyGraph lStyle('zw=0')=3
   ModifyGraph rgb('zw=0')=(0,0,65535)
   ModifyGraph mirror=2
EndMacro
```

First, the AppendMatrixContour operation runs and creates stub traces consisting of zero points. The ModifyContour and ModifyGraph operations that follow act on the stub traces. Finally, after all of the commands that define the graph have executed, Igor does an update of the entire graph (the effect of the PauseUpdate operation expires when the window macro finishes). This is the time when Igor does the actual contour computations which convert the stub traces into fully-formed contour traces.

This delayed computation prevents unnecessary computations from occurring when ModifyContour commands execute in a macro or function. The ModifyContour command often changes default contour level settings, rendering any preceding computations obsolete. For the same reason, the New Contour Plot dialog appends ";DelayUpdate" to the Append Contour commands when a ModifyContour command is also generated.

The DoUpdate operation updates graphs and objects. You can call DoUpdate from a macro or function to force the contouring computations to be done at the desired time.

Drawing Order of Contour Traces

The contour traces are drawn in a fixed order. From back-to-front, that order is:

1. triangulation (Delaunay Triangulation trace, only for XYZ contours),
2. boundary,
3. XY markers,
4. contour trace of lowest Z level,
... intervening contour traces are in order from low-to-high Z level...
N. contour of highest Z level.

You can temporarily override the drawing order with the **Reorder Traces Dialog**, the **Trace Pop-Up Menu**, or the **ReorderTraces** operation. The order you choose will be used until the contour traces are updated. See **Contour Trace Updates** on page II-332.

Note: The order of a contour plot's traces relative to any other traces (the traces belonging to another contour plot for instance) is *not* preserved by the graph's window recreation macro.

Any contour trace reordering is lost when the experiment is closed.

Extracting Contour Trace Data

Advanced users may want to create a copy of a private XY wave pair that describes a contour trace. You might do this to extract the Delaunay triangulation, or simply to inspect the X and Y values in a table, for example. To extract contour wave pair(s), include the Extract Contours As Waves procedure file:

```
#include <Extract Contours As Waves>
```

which adds "Extract One Contour Trace" and "Extract All Contour Traces" menu items to the Graph menu.

Another way to copy the traces into a normal wave is to use the Data Browser to browse the saved experiment; the contour traces are saved as waves in a temporary data folder whose name begins with "WM_CTraces_" and ends with the contour's "contour instance name". See Chapter II-8, **Data Folders**, for more about data folders.

Contour Instance Names

Igor identifies a contour plot by the name of the wave providing Z values (the matrix wave or the Z wave). This "contour instance name" is used in commands that modify the contour plot.

Note: Contour instance names are *not* the same as contour *trace* instance names: contour instance names refer to the data from which the contour traces are derived. See **Contour Trace Names** on page II-328.

The Modify Contour Appearance dialog generates the correct contour instance name automatically.

Contour instance names work much the same way wave instance names for traces in a graph do. See **Instance Notation** on page IV-16.

Examples

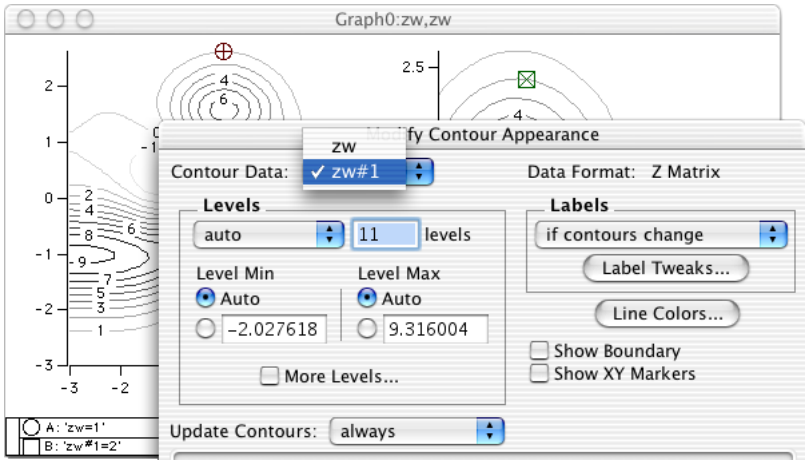
In the first example the contour instance name is "zw":

```
Display; AppendMatrixContour zw           // new contour plot
ModifyContour zw ctabLines={*,*,BlueHot}  // change color table
```

In the unusual case that a graph contains two contour plots of the same data, an instance number must be appended to the name to modify the second plot: zw#1 is the contour instance name of the second contour plot:

```
Display
AppendMatrixContour zw; AppendMatrixContour zw     //two contour plots
ModifyContour zw ctabLines={*,*,RedWhiteBlue}      //change first plot
ModifyContour zw#1 ctabLines={*,*,BlueHot}         //change second plot
```

You might have two contour plots of the same data to show different subranges of the data side-by-side (this example uses separate axes for each plot):



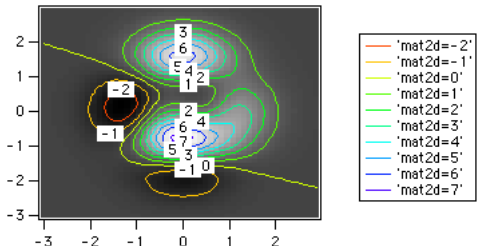
Programming Note

The **ContourNameList** function returns a string containing a list of contour instance names. Each name corresponds to one contour plot in the graph. **ContourInfo** (see page V-60) returns information about a particular named contour plot.

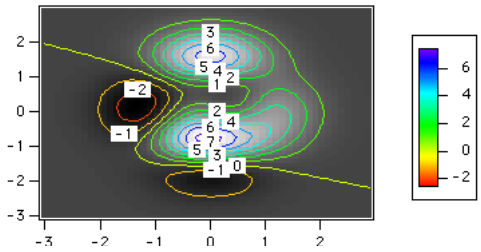
Legends

You can create two kinds of legends appropriate for contour plots using the Add Annotation dialog: a Legend or a ColorScale. For more details about the Add Annotation dialog and creating legends, see Chapter III-2, **Annotations**, and the **Legends** (see page III-52) and **Color Scales** (see page III-59) sections.

A Legend annotation will display the contour traces with their associated color. A ColorScale will display the entire color range as a color bar with an axis that spans the range of colors associated with the contour data.



Legend



Color Scale

Contour Labels

Igor uses specialized tags to create the numerical labels for contour plots. Igor puts one label on every contour curve. Usually there are several contour curves drawn by one contour trace. The tag uses the \OZ escape code or the TagVal(3) function to print the contour level value in the tag instead of printing the literal value. See **Text Content** on page III-43 for more about escape codes and tags.

You can select the rotation of contour labels using the Label Tweaks subdialog of the Modify Contour Appearance Dialog. You can request tangent, horizontal, vertical or both orientations. If permitted, Igor will prefer horizontal labels. The "Snap to" alternatives convert horizontal or vertical labels within 2 degrees of horizontal or vertical to exactly horizontal or vertical.

Igor will position the labels so that they don't overlap other annotations and aren't outside the graph's plot area. Contour labels are slightly special in that they are always drawn below all other annotations, so that they will never show up on top of a legend or axis label. Igor chooses label locations and tangent label orientations based on the slope of the contour trace on the screen.

Controlling Label Updates

By default, Igor automatically relabels the graph only when the contour data or contour levels change, but you can control when labels update with the Labels pop-up menu in the Modify Contour Appearance dialog. See **Labels Pop-Up Menu** on page II-326. Be aware that updating a graph containing many labels can be slow.

Repositioning and Removing Contour Labels

Contour labels are "frozen" so that they can't be dragged, but since they are tags, you *can* Option-drag (*Macintosh*) or Alt-drag (*Windows*) them to a new attachment point. See **Changing a Tag's Attachment Point** on page III-57. The labels are frozen to make them harder to accidentally move.

You can reposition contour labels, but they will be moved back, moved to a completely new position, or deleted when labels are updated. You should turn off label updating before that happens. See **Controlling Label Updates** on page II-335.

Here's a recommended strategy for creating contour labels to your liking:

1. Create the contour plot and set the graph window to the desired size.
2. Choose Modify Contour Appearance (or Shift-double-click the plot area), and click the Label Tweaks button and choose the rotation for labels, and any other label formatting options you want.
3. Choose "update now, once" from the Labels pop-up menu, and then the Do It button.
4. Option-drag or Alt-drag any labels you don't want completely off the graph.
5. Option-drag or Alt-drag any labels that are in the wrong place to another attachment point. You can drag them to a completely different trace, and the value printed in the label will change to the correct value.

To drag a label away from its attachment point, you must first unfreeze it with Position pop-up menu in the **Annotation Tweaks** dialog. See **Overriding the Contour Labels** on page II-336.

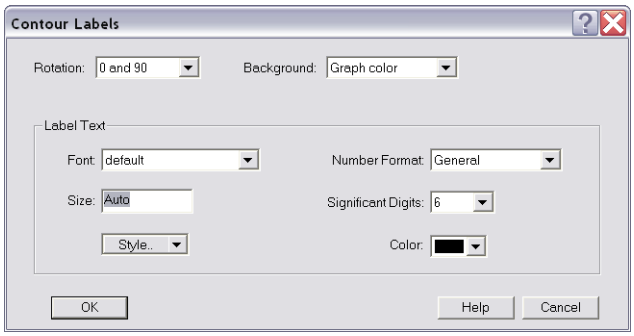
Adding Contour Labels

You can add a contour label with a Tag command like:

```
Tag/Q=ZW#1/F=0/Z=1/B=2/I=1/X=0/Y=0/L=1 'zw#1=2', 0 , "\\OZ"
```

or (if you are sane) use the Modify Annotation dialog to duplicate the annotation and then drag it to a new location.

Modifying Labels



You can change the label font, font size, style, color, and rotation of all labels for a contour plot by clicking Label Tweaks in the Modify Contour Appearance dialog. This brings up the Contour Labels subdialog.

You can choose the rotation of contour labels from tangent, horizontal, vertical or both orientations. If both vertical and horizontal labels are permitted, Igor will choose vertical or horizontal with a preference for horizontal labels. Selecting one of the Tangent choices creates labels that are rotated to follow the contour line. The "Snap to" alternatives convert labels within 2 degrees of horizontal or vertical to exactly horizontal or vertical.

You can choose a specific font, size, and style. The "default" font is the graph's default font, as set by the Modify Graph dialog.

The background color of contour labels is normally the same as the graph plot color (usually white). With the Background pop-up menu, you can select a specific background color for the labels, or choose the window background color or the transparent mode.

You can choose among general, scientific, fixed-point, and integer formats for the contour labels. These correspond to printf conversion specifications, "%g", "%e", "%f", and "%d", respectively (see the **printf** operation on page V-499). These specifications are combined with TagVal(3) into a dynamic text string that is used in each tag.

For example, choosing a Number Format of "###0.0...0" with 3 Digits after Decimal Point in the Contour-Labels dialog results in the contour tags having this as their text: `\{"%.3f",tagVal(3)}`. This format will create contour labels such as "12.345".

Overriding the Contour Labels

Since Igor implements contour labels using standard tags, you can adjust labels individually by simply double-clicking the label to bring up the Modify Annotation dialog.

However, once you modify a label, Igor no longer considers it a contour label and will not automatically update it any more. When the labels are updated, the modified label will be ignored, which may result in two labels on a contour curve.

You may want to take complete, manual control of contour labels. In this case, set the Labels pop-up menu in the Modify Contour Appearance dialog to "no more updates" so that Igor will no longer update them. You can then make any desired changes without fear that Igor will undo them.

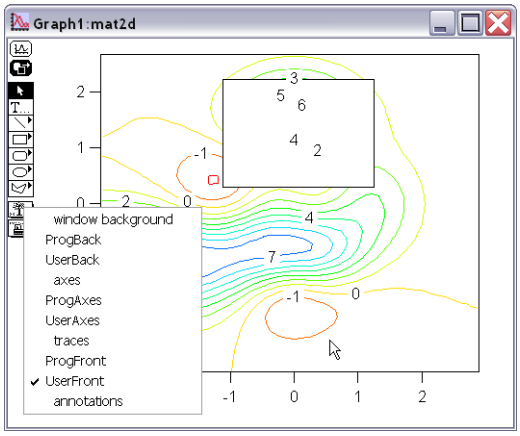
Contour labels are distinguished from other tags by means of the /Q flag. Tag/Q=*contourInstanceName* assigns the tag to the named contour. Igor uses the /Q flag in recreation macros to assign tags to a particular contour plot.

When you edit a contour label with the Modify Annotation dialog, the dialog adds a plain /Q flag (with no =*contourInstanceName* following it) to the Tag command to divorce the annotation from its contour plot.

Add the `/Q=ContourInstanceName` to Tag commands to temporarily assign ownership of the annotation to the contour so that it is deleted when the contour labels are updated.

Labels and Drawing Tools

One problem with Igor's use of annotations as contour labels is that there isn't any drawing layer above them. If you use the drawing tools to create a rectangle in the same location as some contour labels, you will encounter something like the following window.



You will need to remove the offending labels by Option-dragging (*Macintosh*) or Alt-dragging (*Windows*) them elsewhere or entirely off the graph (see **Repositioning and Removing Contour Labels** on page II-335). Do this after you have disabled label updates (see **Update Contours Pop-Up Menu** on page II-326).

Contouring Pitfalls

You may encounter situations in which the contour plot doesn't look as you expect.

Insufficient Resolution

Contour curves are generally closed curves, or they intersect the data boundary. Under certain conditions, typically when using XYZ triplet data, the contouring algorithm may generate what appears to be an open curve (a line rather than a closed shape). This open curve typically corresponds to a peak ridge or a valley trough in the surface. At times, an open curve may also correspond to a line that intersects a nonobvious boundary.

The line may actually be a very narrow closed curve: zoom in by dragging out a marquee, click inside and choose "expand" from the pop-up menu.

If it really is a line, increasing the resolution of the data in that region (by adding more X, Y, Z triplets) may result in a closed curve. Selecting higher interpolation settings with the **Interpolation Pop-Up** may help.

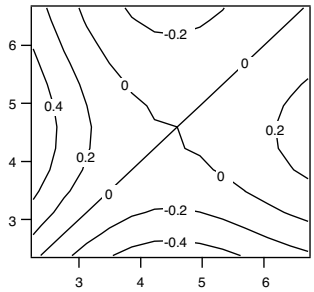
Another solution is to shift the contour level slightly down from a peak or up from a valley. Or you could choose a new set of levels that don't include the level exhibiting the problem. See **Levels** on page II-326.

Crossing Contour Lines

Contour lines corresponding to different levels will not cross each other, but contour lines of the same level may appear to intersect. This typically happens when a contour level is equal to a "saddle point" of the surface. An example of this is a contour level of zero for the function:

```
z= sinc(x) - sinc(y)
```

```
z= sinc(x) - sinc(y)
```



You should shift the contour level away from the level of the saddle point. See **Levels** on page II-326.

Flat Areas in the Contour Data

Patches of constant Z values in XYZ triplet data don't contour well at those levels. If the data has flat areas equal to 2.0, for example, a contour level at Z=2.0 may produce ambiguous results. Gridded contour data does not suffer from this problem.

You should shift the contour level above or below the level of the flat area. See **Levels** on page II-326.

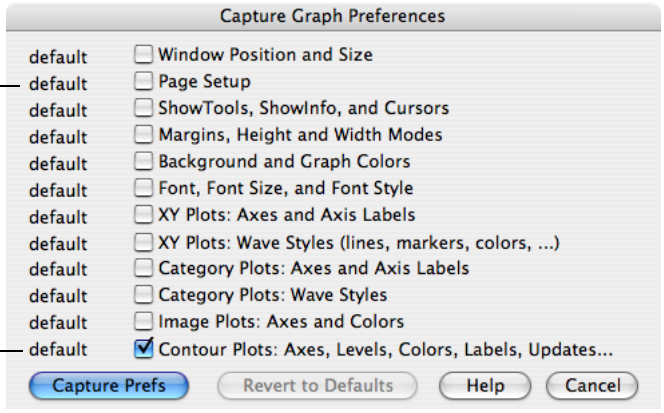
Contour Preferences

You can change the default appearance of contour plots by "capturing" preferences from a "prototype" graph containing contour plots.

Create a graph containing a contour plot (or plots) having the settings you use most often. Then choose Capture Graph Prefs from the Graph menu. Select the Contour Plots category, and click Capture Prefs.

The factory default settings for the Contour Plot category are currently in effect.

The Contour Appearance settings and the axis settings used for the contour plot in the target Graph will be captured as the new preferred settings.



Preferences are normally in effect only for *manual* operations, not for automatic operations from Igor procedures. This is discussed in more detail in Chapter III-17, **Preferences**.

The Contour Plots category includes both Contour Appearance settings and axis settings.

Contour Appearance Preferences

The captured Contour Appearance settings are automatically applied to a contour plot when it is first created (provided preferences are turned on). They are also used to preset the Modify Contour Appearance dialog (which is also a subdialog of the New Contour Plot dialog).

If you capture the Contour Plot preferences from a graph with more than one contour plot, the first contour plot appended to a graph gets the settings from the contour first appended to the prototype graph. The second contour plot appended to a graph gets the settings from the second contour plot appended to the prototype graph, etc. This is similar to the way XY plot wave styles work.

Contour Axis Preferences

Only axis settings used by the contour plot are captured. Axes used solely for an XY, category, or image plot are not captured when the Contour Plots category is selected.

The contour axis preferences are applied only when axes having the same name as the captured axis are created by AppendMatrixContour or AppendXYZContour commands. If the axes existed before those commands are executed, they will not be affected by the axis preferences. The names of captured contour axes are listed in the X Axis and Y Axis pop-up menus of the New Contour Plot and Append Contour Plot dialogs. This is similar to the way XY plot axis preferences work.

You can capture contour axis settings for the standard left and bottom axes, and Igor will save these separately from left and bottom axis preferences captured for XY, category, and image plots. Igor will use the contour axis settings for AppendMatrixContour or AppendXYZContour commands only.

How to Use Contour Plot Preferences

Here is our recommended strategy for using contour preferences:

1. Create a new graph containing a single contour plot (if you want to capture the Show Triangulation and Interpolation settings, you must make an XYZ contour plot). Use the axes you will want for a contour plot.
2. Use the Modify Contour Appearance dialog and the Modify Axis dialog to make the contour plot appear as you prefer.
3. Choose Graph→Capture Graph Prefs, select the Contour Plots category, and click Capture Prefs.

References

Watson, David F., *Contouring: A Guide to the Analysis and Display of Spatial Data*, 340 pp., Pergamon Press, New York, 1992.

Contour Plot Shortcuts

Because contour plots are drawn in a normal graph, all of the **Graph Shortcuts** (see page II-306) apply. Here we list those which apply specifically to contour plots.

Action	**Shortcut** (Macintosh)	**Shortcut** (Windows)
To modify the appearance of the contour plot as a whole	Control-click and choose Modify Contour from the pop-up menu **or**	Right-click and choose Modify Contour from the pop-up menu **or**
	Press Shift and double-click the plot area of the graph, away from any labels or traces. This brings up the Modify Contour Appearance dialog.	Press Shift and double-click the plot area of the graph, away from any labels or traces. This brings up the Modify Contour Appearance dialog.
To modify a contour label	Press Shift and double-click the plot area, and click Label Tweaks in the Modify Contour Appearance dialog.	Press Shift and double-click the plot area, and click Label Tweaks in the Modify Contour Appearance dialog.
	Don't double-click the label to use the Modify Annotation dialog unless you intend to maintain the label yourself. See **Overriding the Contour Labels** on page II-336.	Don't double-click the label to use the Modify Annotation dialog unless you intend to maintain the label yourself. See **Overriding the Contour Labels** on page II-336.
To remove a contour label	Press Option, click in the label, and drag it completely off the graph.	Press Alt, click in the label, and drag it completely off the graph.
To move a contour label to another contour trace	Press Option, click in the label and drag it to another contour trace.	Press Alt, click in the label and drag it to another contour trace.
To duplicate a contour label	Double-click any label, click the Duplicate button, and click the Do It button.	Double-click any label, click the Duplicate button, and click the Do It button.
To modify the appearance or front-to-back drawing order of a contour trace	Press Control and click the trace to get a pop-up menu.	Right-click the trace to get a contextual menu.
	Press Command-Shift to modify all traces.	Press Shift while right-clicking to modify all traces.
	Double-click the trace to summon the Modify Trace Appearance dialog.	Double-click the trace to summon the Modify Trace Appearance dialog.
	Also see **Overriding the Color of Contour Traces** on page II-332 and **Drawing Order of Contour Traces** on page II-333.	Also see **Overriding the Color of Contour Traces** on page II-332 and **Drawing Order of Contour Traces** on page II-333.

Chapter
II-15

Image Plots

Overview

You can display matrix data as an image plot in a graph window. Each matrix data value defines a colored rectangle in the image plot. The size and position of these rectangles is controlled by the range of the graph axes, the graph width and height modes, and the X and Y coordinates of the image rectangles.

Note: The terms "matrix data value", "pixel", and "Z value" are used interchangeably in this chapter.

Image data can be false color, indexed color or direct color.

False Color Images

In false color images the data values in the 2D matrix are linearly mapped onto a color table. This is a very powerful way to view matrix data and is often more effective than either surface plots or contour plots. For best results you can superimpose a contour plot on top of a false color image of the same data.

Igor has several built-in color tables (see **Color Tables** on page II-349). You can provide color index waves that define custom color tables (described in **Indexed Color Details** on page II-356).

Indexed Color Images

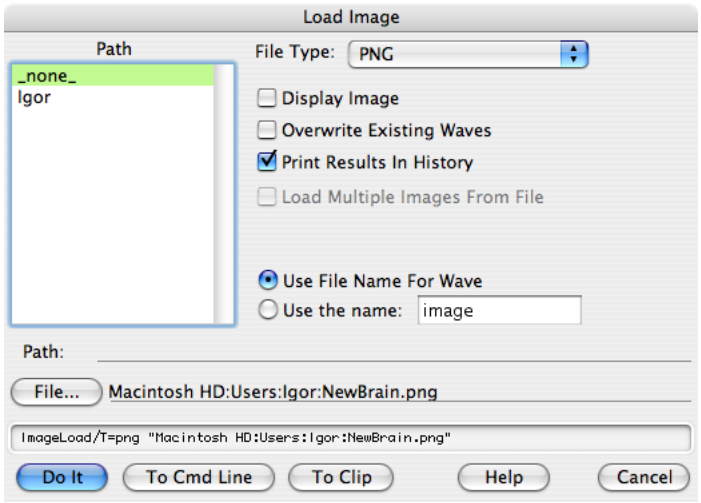
Indexed color images use the data values stored in your 2D matrix as indices (usually integers) into a three-column wave of color values that you supply. "True color" images such as those that come from video cameras or scanners generally use indexed color. Indexed color images are more common than direct color because they consume less memory. See **Indexed Color Details** on page II-356.

Direct Color Images

Direct color images contain the actual red, green and blue values at each point in the image matrix. Direct color images use a 3D wave with 3 color planes containing absolute values for red, green and blue, providing 24-bit color. While an indexed color image contains at most the number of colors in the color index wave, a direct color image can have a unique color for every pixel. See **Direct Color Details** on page II-358.

Loading an Image

To create an image plot, first create or load your matrix wave. You can load TIFF, PICT, JPEG, PNG, GIF, BMP, PhotoShop, Targa, Silicon Graphics, and Sun Raster image files into matrix waves using the **ImageLoad** operation (see page V-269) or the Load Image dialog under the Data menu. Depending on the type of image that you would like to read, this operation may require that you have Apple's QuickTime software (version 4 or newer) installed on your computer.

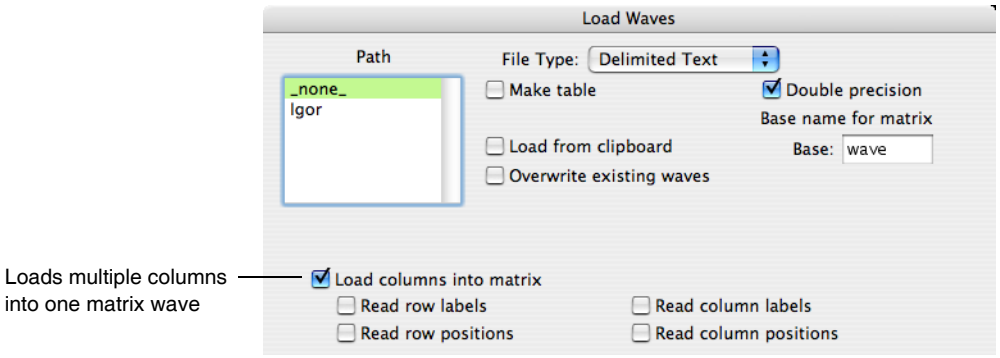


Except for TIFF and Sun Raster images, all images are loaded onto a 3D Igor wave (unsigned byte with plane 0 containing the red channel, plane 1 the green and plane 2 the blue channel). You can convert the 3D wave into other forms using the **ImageTransform** operation (see page V-290).

When loading TIFF files into Igor, a number of additional options are available. For stacked TIFF files that contain multiple images, you can read all of the images into a single 3D wave (where each image occupies a sequential plane), into individual 2D waves, or you can specify a particular image, or range of images, that you would like to read. Reading a TIFF image stack into a single 3D wave is applicable only for images that are 8 or 16 bits/pixel deep.

If you want to load an image file but you are not sure about it's file format, use the "any" or "*.*" specification for the file type. Most of the supported file types can be uniquely recognized by the ImageLoad operation.

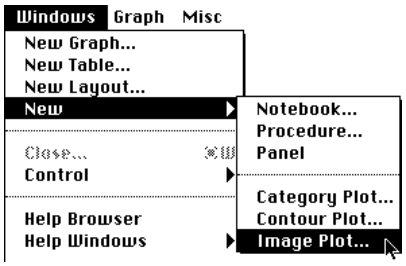
You can also load numeric text and binary files using the Load Waves dialog in the Data menu. Select the "Load columns into matrix" checkbox:



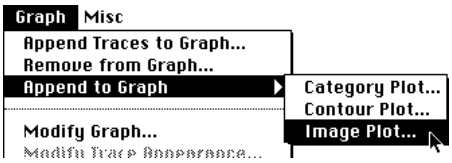
You can also obtain images by acquisition of video using appropriate frame grabber's XOPs (e.g., QTGrabber) or load images using specific file loader XOPs (e.g., HDF5 Loader).

Creating an Image Plot

You can create an image plot in a new graph window with the New Image Plot dialog. This dialog creates a blank graph to which the plot is appended. You can add an image plot to an existing graph with the Append Image Plot dialog.



Creating a new image plot



Appending an image plot to an existing graph

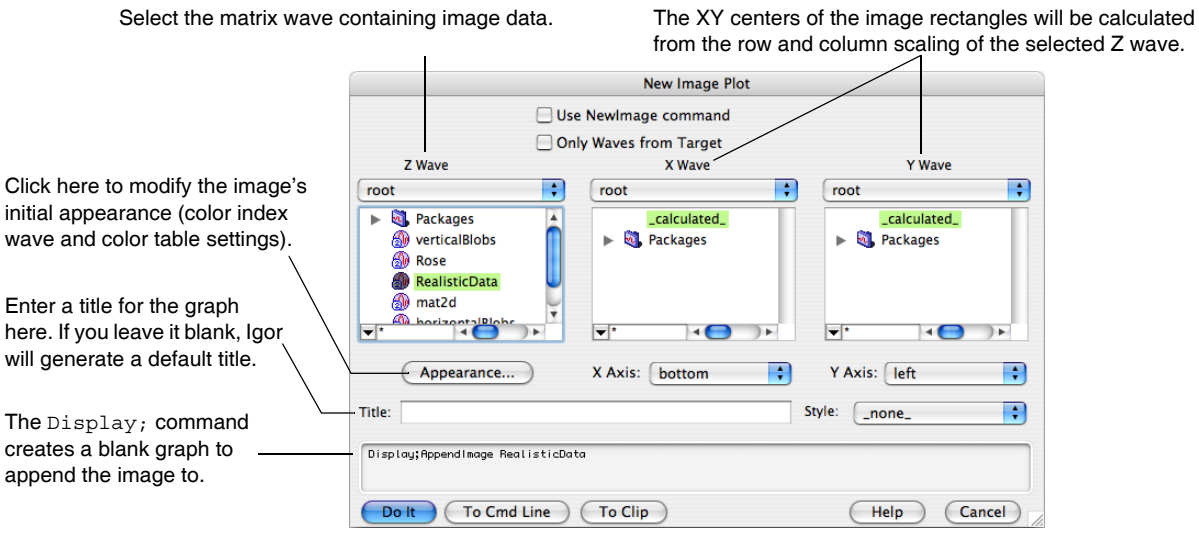
You can also use the **NewImage** operation (see page V-436) or the **AppendImage** operation (see page V-23). Also see **How Images Are Displayed** on page II-346.

Image plots are displayed in an ordinary graph window. All the features of graphs apply to image plots: axes, line styles, drawing tools, controls, etc. See Chapter II-12, **Graphs**.

You can show lines of constant image value by appending a contour plot to a graph containing an image. Igor will draw the contour above all the image plots. See **Creating a Contour Plot** on page II-322 for an example of combining contour plots and images in a graph.

Image Plot Dialogs

Choose New Image Plot from the Windows menu to bring up the New Image Plot dialog:



The Title, Style, X Axis and Y Axis items work the same as in the New Graph dialog (See **Creating Graphs** on page II-237). You can specify a new title for the graph and select or create the axes used in the image plot. Use the Style pop-up menu to apply a style macro to the newly created graph window.

This dialog normally generates two commands — a Display command to make a blank graph window, and an AppendImage command to append a image plot to that graph window. Selecting the "Use NewImage command" checkbox replaces Display and AppendImage with the NewImage command, which changes what options you have for creating an image plot in this dialog (see **Use NewImage Command** on page II-344).

The Append Image Plot dialog is similar, except that the Only Waves from Target, Use NewImage command, Title, and Style items are missing.

X, Y, and Z Wave Lists

The X Wave, Y Wave, and Z Wave lists show the available waves that will be accepted by the AppendImage operation. Select the Only Waves from Target checkbox to show only the waves in the target window (most useful when the target window is a table).

You should select the matrix wave containing your image data in the Z Wave list. This will update the X Wave and Y Wave lists to show only those waves, if any, that may be combined with the image data matrix wave.

Choosing _calculated_ from the X Wave list uses the row scaling (X scaling) of the matrix selected in the Z Wave list to provide the X coordinates of the image rectangle centers.

Choosing _calculated_ from the Y Wave list uses the column scaling (Y scaling) of the matrix to provide Y coordinates of the image rectangle centers.

You can also select a 1D wave to provide the X or Y values for a matrix of Z values; only those waves with the proper length for the selected Z Wave are shown in the X Wave and Y Wave lists. See **Image X and Y Coordinates** on page II-346 and the **AppendImage** operation on page V-23.

Use NewImage Command

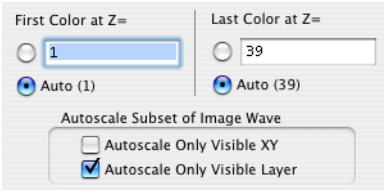
The NewImage command nicely sets the window margins and axes to maximize the image in the window, presets the window size to match the number of pixels in the image, and automatically reverses the left (vertical) axes so that pictures aren't displayed upside down.

However NewImage does not support window titles or 1D X and Y waves, axes other than left and top, or graph style macros.

A useful option of NewImage is the "Do not treat this three-layer image as direct (rgb) color" checkbox that appears only for three-layer Image Waves.

Selecting that option displays one layer from a three-layer wave as a false-color or indexed-color image (the data values in one layer are linearly mapped onto a color table), instead of using all three layers to display a direct color image (see **Direct Color Details** on page II-358).

Click the Appearance button to chose which layer is displayed.



The other two layers are not visible and are ignored except for possibly autoscaling the color table Z limits. If Autoscale Only Visible Layer is checked in the Modify Image Appearance dialog, the other layers are then completely ignored.

Modifying an Image Plot

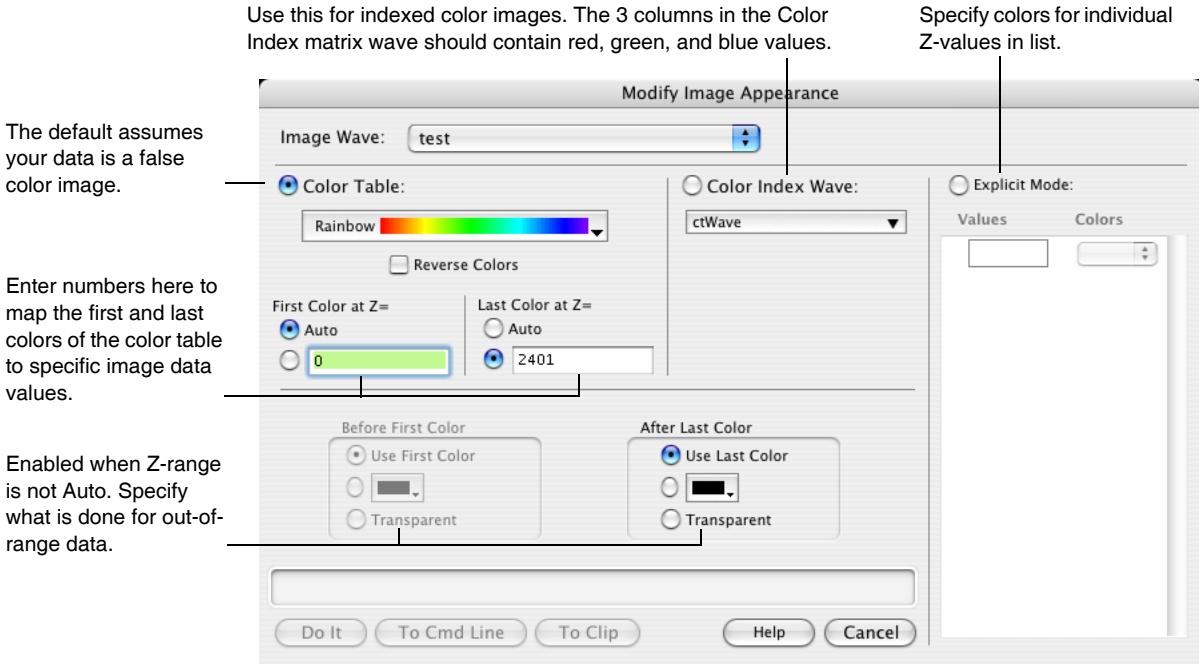
You can change the appearance of the image plot using the Modify Image Appearance dialog. This dialog is also available as a subdialog of the New Image Plot dialog.

Tip #1: You can open the Modify Image dialog quickly by Control-clicking (*Macintosh*) or right-clicking (*Windows*) and choosing Modify Image from the pop-up menu. The graph must contain an image plot for this to work.

Tip #2: Use the preferences to change the default image appearance, so you won't be making the same changes over and over. See **Image Preferences** on page II-361.

The Modify Image Appearance Dialog

The Modify Image Appearance dialog applies to false color and indexed color images, but not direct color images. See **Direct Color Details** on page II-358.



If your image is indexed color, choose the matching color index wave from the Color Index Wave pop-up menu. For Color Index Wave details, see **Indexed Color Details** on page II-356.

If your image is false color, choose a built-in color table from the pop-up menu. Autoscaled color mapping assigns the first color in a color table to the minimum value of the image data and the last color to the

maximum value. (The dialog uses "Z" to refer to the values in the image wave.) For more information, see **Color Tables** on page II-349.

Indexed and color table colors are distributed between the minimum and maximum Z values either linearly or logarithmically, based on the `ModifyImage log` parameter, which is set by the Log Colors checkbox.

Only in rare cases will you want to choose a Lookup Wave; "_none_" is the usual choice. See **Color Table Ranges** on page II-349 for a reason to use a Lookup Wave.

You can achieve special effects using a Color Index Wave to select colors for your image; see **Indexed Color Details** on page II-356. The built-in color tables are easier to use, however.

Use Explicit Mode to select specific colors for specific Z values in the image. If an image element is exactly equal to the number entered in the dialog, it will be given the assigned color. This is not very useful for images made with floating-point data; it is intended for integer data. It is almost impossible to enter exact matches for floating-point data.

When you select Explicit Mode for the first time, two entries are made for you assigning white to 0 and black to 255. A third blank line is added for you to enter a new value. If you put something into the blank line, another blank line is added.

To remove an entry, click in the blank areas of a line in the list to select it and press Delete (*Macintosh*) or Backspace (*Windows*).

If you make an image using a three-layer wave containing direct RGB values, you cannot alter it with this dialog (unless it has multiple chunks, in which case you can modify only the displayed chunk).

How Images Are Displayed

Igor displays images by drawing one "image rectangle" for each data value in the image matrix wave. The size and position of these rectangles is controlled by the range of the graph axes, the graph width and height modes, and the X and Y coordinates of the image rectangles.

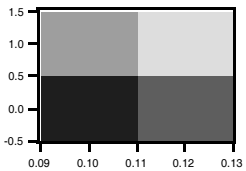
Images are displayed behind all other objects in a graph except the background color.

If your image data is a floating point type, you can use NaN to represent missing data. This allows the graph background color to show through. This feature is particularly useful when you want to create overlays.

Image X and Y Coordinates

Images are displayed versus axes just like XY plots. Individual pixels fill a rectangle defined by adjacent X and Y coordinate values and each rectangle is centered on the X and Y coordinate for the pixel. Here is an example that uses a 2x2 matrix to exaggerate the effect:

```
Make/O small={{0,1},{2,3}}
SetScale/I x 0.1,0.12,"", small
Display
AppendImage small    // _calculated_ X & Y
ModifyImage small ctab={-0.5,3.5,Grays}
```

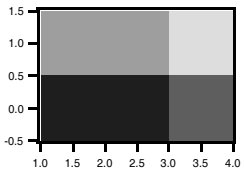


Note that on the X axis the rectangles are centered on 0.1 and 0.12, the row X indices of the matrix as defined by its X scaling. On the Y axis they are centered on 0 and 1 since the Y scaling was left at the default (point scaling).

When appending an image to a graph, you can supply optional X and Y waves to define the coordinates *of the rectangle edges*. These waves need to contain one more data point than the X (row) or Y (column) dimension of the matrix to define the end of the last rectangle.

Here is an example, using the matrix from above, that should make this point clear:

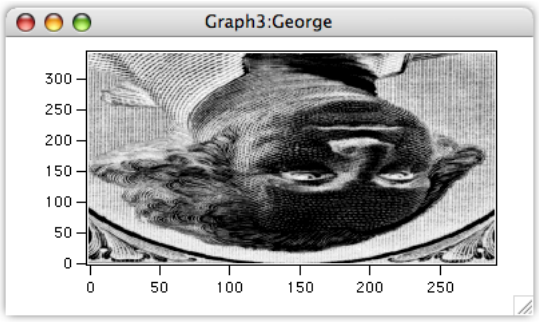
```
Make smallx={1,3,4}
Display
// define X edges with smallx
AppendImage small vs {smallx,*}
ModifyImage small ctab={-0.5,3.5,Grays,0}
```



In this case the Y is unchanged because we did not supply a Y coordinate wave. For the X axis, the X coordinate wave (smallx) now controls the start and end of each rectangle. The rectangles are not centered because they can not be: if we tried to center the rectangles horizontally on 1 and 3 then there would be a gap between them.

Image Orientation

By default, the AppendImage operation draws increasing Y values (matrix column indices) upward, and increasing X (matrix row indices) to the right. Most image formats expect Y to increase downward:

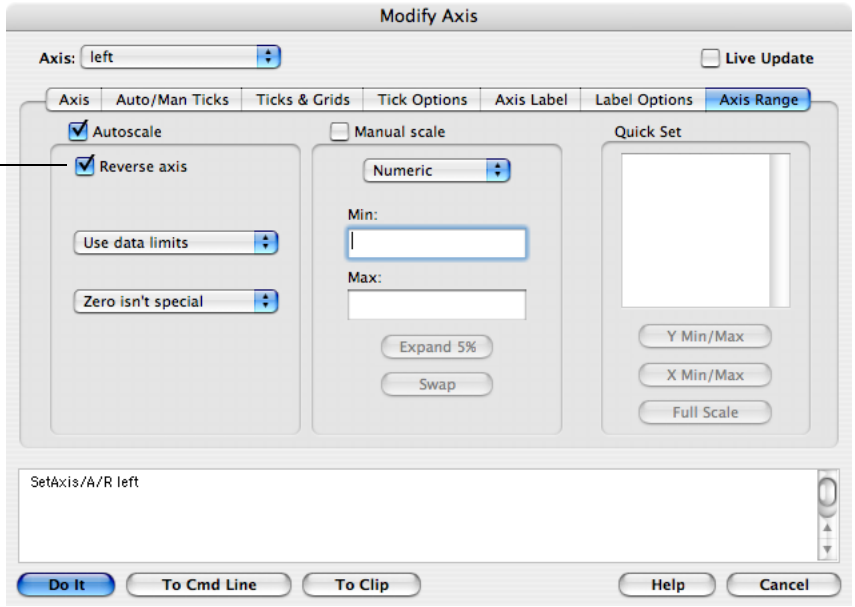


TIFF file displayed with default AppendImage settings

Alternatively you can select the "Use NewImage command" checkbox in the New Image Plot dialog. NewImage automatically reverses the left axes (see **Use NewImage Command** on page II-344).

You can flip an image vertically by reversing the Y axis, and horizontally by reversing the X axis, using the Axis Range tab in the Modify Axes dialog:

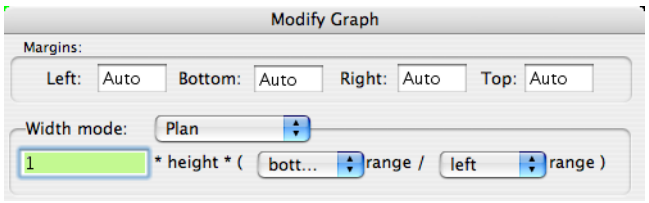
Reverse autoscaled vertical axes with this checkbox to make images appear right-side-up.



You can also flip the image vertically by reversing the Y scaling of the matrix wave. Note that if you use the **NewImage** operation (see page V-436), there is no need to flip the image.

Image Rectangle Aspect Ratio

By default, Igor does not make the image rectangles square. Use the Modify Graph dialog (in the Graph menu) to correct this by setting Plan width mode:



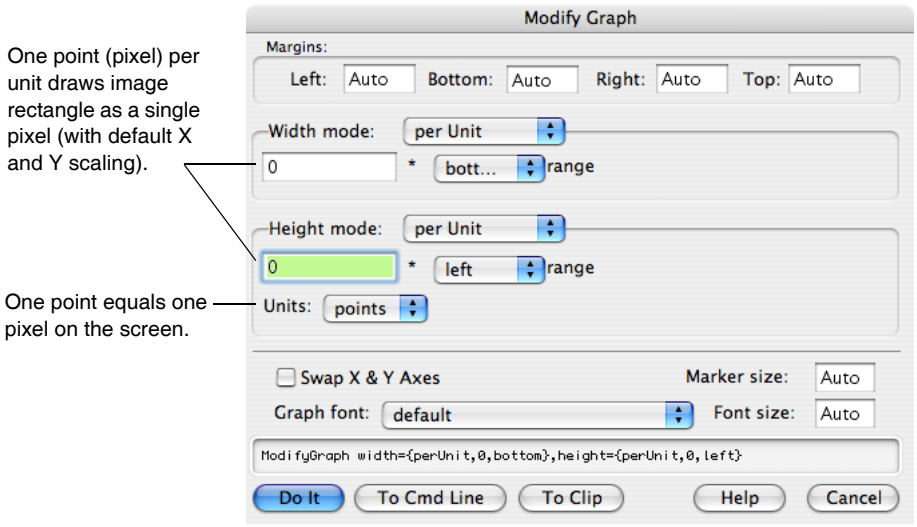
You can use the Plan height mode to accomplish the same result.

If DimDelta(*matrixWave*,0) ¦ DimDelta(*matrixWave*,1), you will need to enter the ratio (or inverse ratio) of these two values in the Plan height (or width):

```
SetScale/P x 0,3,"", mat2dImage
SetScale/P y 0,1,"", mat2dImage
ModifyGraph width=0, height={Plan,3,left,bottom}
// or
ModifyGraph height=0, width={Plan,1/3,bottom,left}
```

Do not use the Aspect width or height modes; they make the entire image plot square even if it shouldn't be.

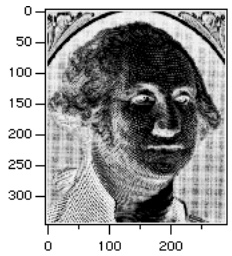
Plan mode ensures the image rectangles are square, but it allows them to be of any size. If you want each image rectangle to be a single (square) pixel, use the per Unit width and per Unit height modes. With point X and Y scaling of an image matrix, use one point (not one inch) per unit:



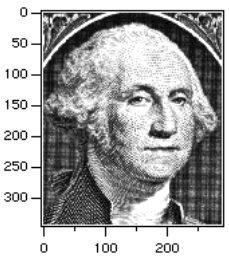
You can also flip an image along its diagonal by setting the Swap XY checkbox.

Image Polarity

Sometimes the image's pixel values are inverted, too. False color images can be inverted by reversing the color table. Select the Reverse Colors checkbox in the Modify Image Appearance dialog. See **Color Tables** on page II-349. To reverse the colors in an index color plot is harder: the rows of the color index wave must be reversed.



After SetAxis/A/R left
ModifyGraph width={Plan,1,bottom,left}



After reversing
the Grays color table

Color Tables

The data values contained in your matrix are normally linearly mapped into a table of colors containing a sweep of colors that lets the viewer easily identify the data values. The data values can be logarithmically mapped by using the `ModifyImage log=1` option, which is useful when the data values span multiple orders of magnitude.

There are 56 built-in color tables you can use with false color images.

Note: You can not create your own color tables, but you can use a color index wave to create a custom range of colors. See **Indexed Color Details** on page II-356.

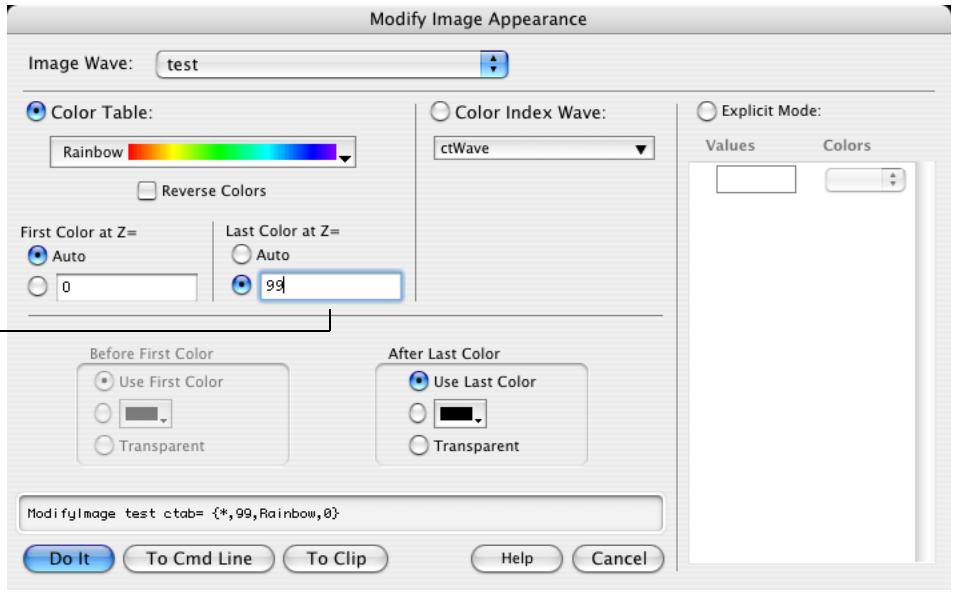
The **CTabList** function (see page V-81) returns a list of all color table names. You can create a color index wave from any built-in color table with the **ColorTab2Wave** operation (see page V-57).

The ColorsMarkersLinesPatterns example Igor experiment (In the Testing & Misc folder) implements a Color Table Visual Guide. These various color tables are summarized in the section **Color Table Details** on page II-353.

Color Table Ranges

The range of data values that maps into the range of colors in the table can be set either manually or automatically using the Modify Image Appearance dialog.

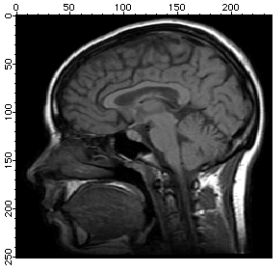
Enter numbers here to map the first and last colors to specific matrix data values (Z values).



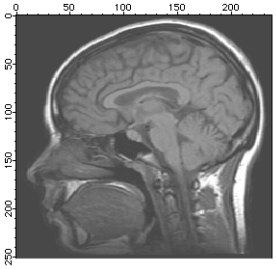
When you choose to autoscale the first or last color, your image is scaled using, respectively, the minimum or maximum data values.

By changing the "First Color at Z=" and "Last Color at Z=" values you can examine subtle features in your data.

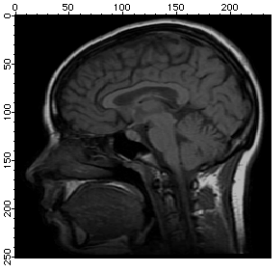
For example, when using the Grays color table, you can lighten the image by assigning the First Color (which is black) to a number lower than the image minimum value. This maps a lighter color to the minimum image value. To darken the maximum image values, assign the Last Color to a number higher than the image maximum value, mapping a darker color to the maximum image value.



ctab = {0,255,Grays}



ctab = {-100,255,Grays}



ctab = {0,355,Grays}

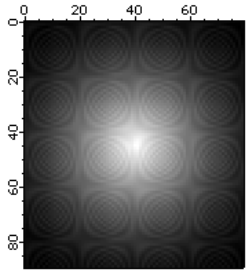
Data values greater than the range maximum are given the last color in the color table, or they can all be assigned to a single color or made transparent. Similarly, data values less than the range minimum are given the first color in the color table, or they can all be assigned to a single color (possibly different from the max color), or made transparent.

Example: Overlaying Data on a Background Image

By setting the image range to render small values transparent, you can see the underlying image in those locations, which helps visualize where the nontransparent values are located with reference to a background image. Here's a fake weather radar example.

First, we create some "land" to serve as a background image:

```
Make/O/N=(80,90) landWave
landWave = 1-sqrt((x-40)*(x-40)+(y-45)*(y-45))/sqrt(40*40+45*45)
landWave = 7000*landWave*landWave
landWave += 200*sin((x-60)*(y-60)*pi/10)
landWave += 40*(sin((x-60)*pi/5)+sin((y-60)*pi/5))
NewImage landWave
```



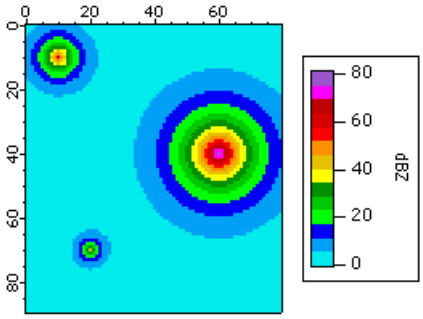
Then we create some "weather" radar data ranging from about 0 to 80 dBZ:

```
Duplicate/O landWave overlayWeather                    // "weather" radar values
overlayWeather=60*exp(-(sqrt((x-10)*(x-10)+(y-10)*(y-10))/5))      // storm 1
overlayWeather+=80*exp(-(sqrt((x-60)*(x-60)+(y-40)*(y-40)))/10)    // storm 2
overlayWeather+=40*exp(-(sqrt((x-20)*(x-20)+(y-70)*(y-70)))/3)     // storm 3
SetScale d, 0, 0, "dBZ", overlayWeather
```

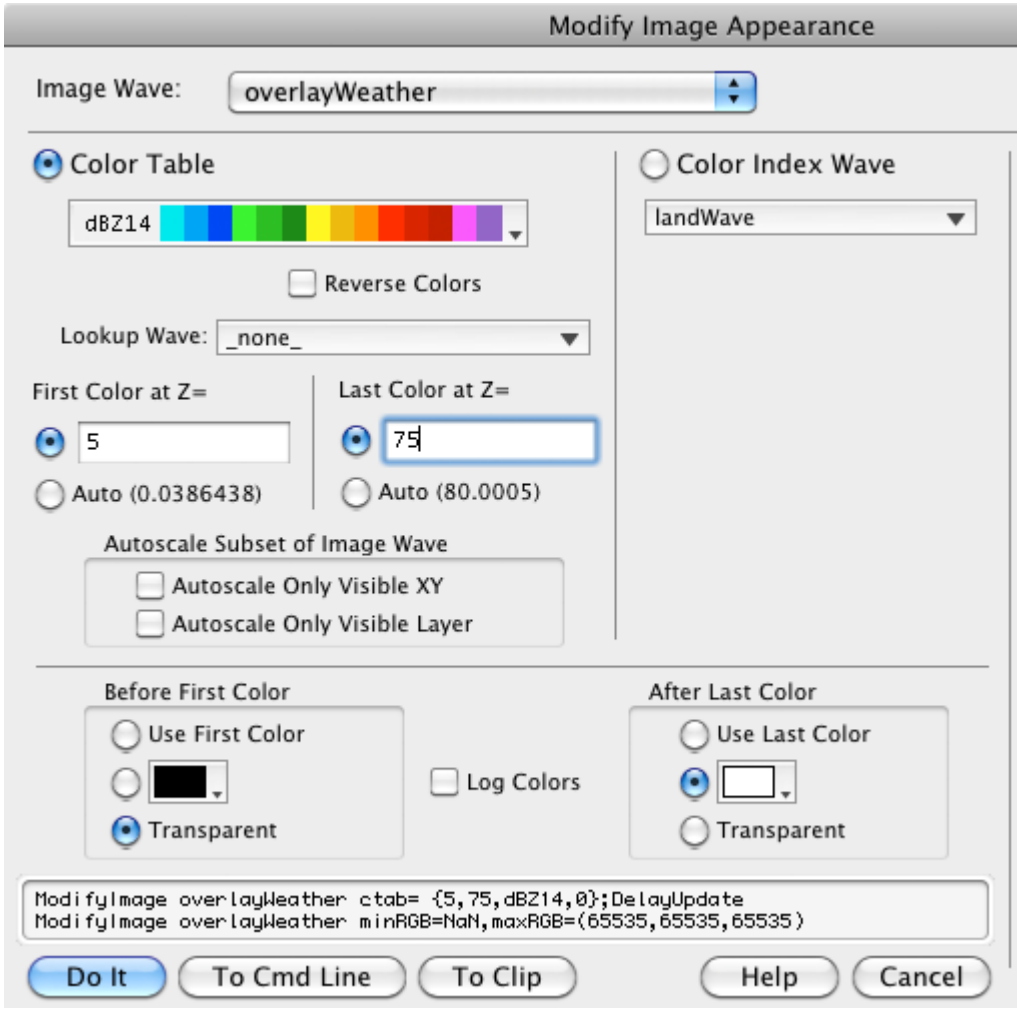
And append it using the same axes as the landWave to overlay the images. With the default color table range, the landWave is totally obscured:

```
AppendImage/T overlayWeather
ModifyImage overlayWeather ctab= {*,*,dBZ14,0}
// Show the image's data range with a ColorScale
```

```
ModifyGraph width={Plan,1,top,left}, margin(right)=100
ColorScale/N=text0/X=107.50/Y=0.00 image=overlayWeather
```

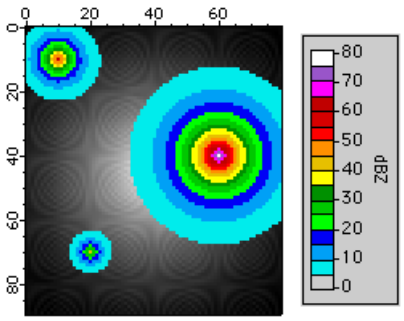


Calibrate the image plot colors to National Weather Service values for precipitation mode by selecting the dBZ14color table for data values ranging from 5 to 75, where values below 5 are transparent and values above 75 are white:



Modify the ColorScale to show a range larger than the color table values (0-80):

```
ColorScale/C/N=text0 colorBoxesFrame=1,heightPct=90,nticks=10
ColorScale/C/N=text0/B=(52428,52428,52428) axisRange={0,80},tickLen=3.00
```



Color Table Ranges - Lookup Table (Gamma)

Normally the range of data values and the range of colors are linearly related or logarithmically related if the `ModifyImage log` parameter is set to 1. You can also cause the mapping to be nonlinear by specifying a lookup (or "gamma") wave, as described in this next example.

Example: Using a Lookup for Advanced Color/Contrast Effects

The **ModifyImage** operation (see page V-415) with the lookup parameter specifies a 1D wave that modifies the mapping of scaled Z values into the current color table. Values in the lookup wave should range from 0.0 to 1.0. A linear ramp from 0 to 1 would have no effect while a ramp from 1 to 0 would reverse the colormap. Used to apply gamma correction to grayscale images or for special effects.

```
Make luWave=0.5*(1+sin(x/30))
Make /n=(50,50) simpleImage=x*y
NewImage simpleImage
ModifyImage simpleImage ctab= {*,*,Rainbow,0}

// After inspecting the simple image, apply the lookup:
ModifyImage simpleImage lookup=luWave
```

This example is a simplified version of the Image Contrast Lookup panel available in the Image Processing package. Choose Analysis→Packages→Image Processing to load the package, then choose Image Contrast to open the panel.

Specialized Color Tables

Some of the color tables are designed for specific uses and specific numeric ranges.

The BlackBody color table shows the color of a heated "black body" (though not the brightness of that body) over the temperature range of 1,000 to 10,000 K.

The Spectrum color table is designed to show the color corresponding to the wavelength of visible light as measured in nanometers over the range of 380 to 780 nm.

The SpectrumBlack color table does the same thing, but over the range of 355 to 830 nm. The fading to black is an attempt to indicate that the human eye loses the ability to perceive colors at the range extremities.

The GreenMagenta16, EOSOrangeBlue11, EOSSpectral11, dBZ14, and dBZ21 tables are designed to represent discrete levels in weather-related images, such as radar reflectivity measures of precipitation and wind velocity and discrete levels for geophysics applications.

The LandAndSea, Relief, PastelsMap, and SeaLandAndFire color tables all have a sharp color transition which is intended to denote sea level. The LandAndSea and Relief tables have this transition at 50% of the range. You can put this transition at a value of 0 by setting the minimum value to the negative of the maximum value:

```
ModifyImage imageName, ctab={-1000,1000,LandAndSea,0}     // image plot
ColorScale/C/N=scale0 ctab={-1000,1000,LandAndSea,0}      // colorscale
```

The PastelsMap table has this transition at 2/3 of the range. You can put this transition at a value of 0 by setting the minimum value to twice the negative of the maximum value:

```
ModifyImage imageName, ctab={-2000,1000,PastelsMap,0}    // image plot
ColorScale/C/N=scale0  ctab={-2000,1000,PastelsMap,0}    // colorscale
```

This principle can be extended to the other color tables to position a specific color to a desired value. Some trial-and-error is to be expected.

Note: The BlackBody, Spectrum, and SpectrumBlack color tables are based on algorithms from the Color Science web site:

<http://www.physics.sfasu.edu/astro/color.html>.

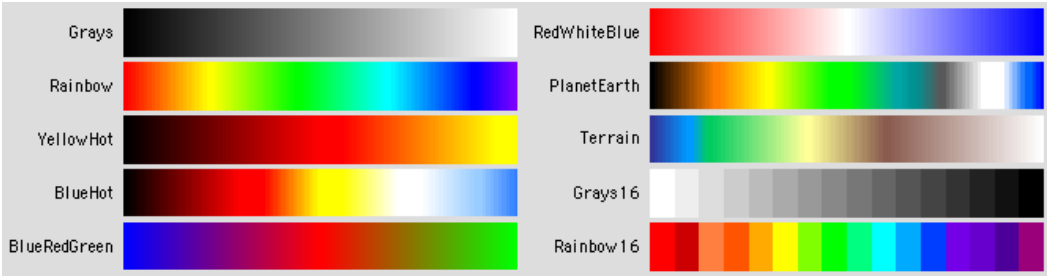
Also see *Color Science* by Wyszecki and Stiles.

Color Table Details

The built-in color tables can be grouped into several categories. For most purposes you may only need to use the "compatible" color tables; this is especially true if you wish to use your experiments with earlier versions of Igor Pro.

Igor Pro 4-Compatible Color Tables

Igor Pro 4 supports10 built-in color tables: Grays, Rainbow, YellowHot, BlueHot, BlueRedGreen, RedWhiteBlue, PlanetEarth, Terrain, Grays16, and Rainbow16. These color tables have 100 color levels except for Grays16 and Rainbow16, which only have 16 levels.



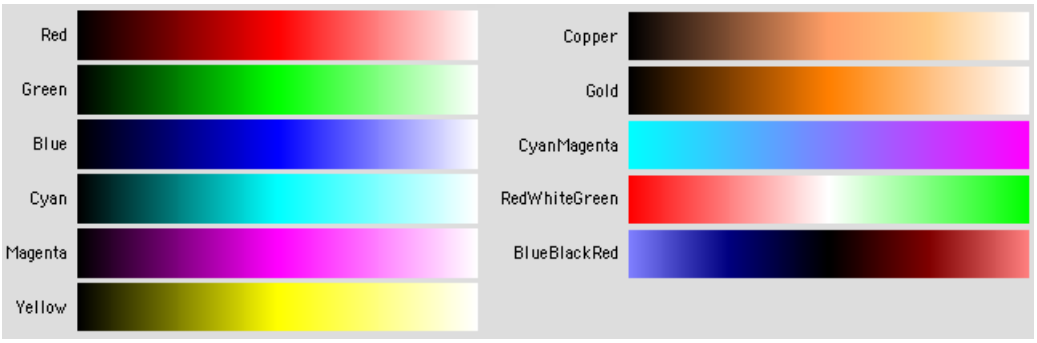
Igor Pro 5-Compatible Color Tables

Igor Pro 5 added 256-color versions of the eight 100-level color tables in Igor Pro 4 (Grays256, Rainbow256, etc.), new gradient color tables, and new special-purpose color tables.

Gradient Color Tables

These are 256-color transitions between two or three colors.

Color Table Name	Colors	Notes
Red	256	Black → red → white.
Green	256	Black → green → white.
Blue	256	Black → blue → white.
Cyan	256	Black → cyan → white.
Magenta	256	Black → magenta → white.
Yellow	256	Black → yellow → white.
Copper	256	Black → copper → white.
Gold	256	Black → gold → white.
CyanMagenta	256	
RedWhiteGreen	256	
BlueBlackRed	256	

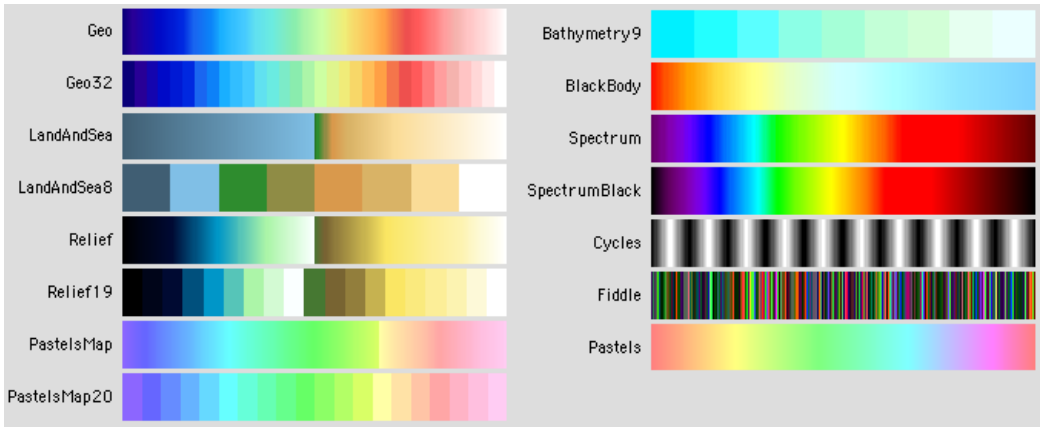


Special-Purpose Color Tables

The special purpose color tables are ones that will find use for particular needs, such as coloring a digital elevation model (DEM) of topography or for spectroscopy. These color tables can have any number of color entries.

The following table summarizes the various special-purpose color tables.

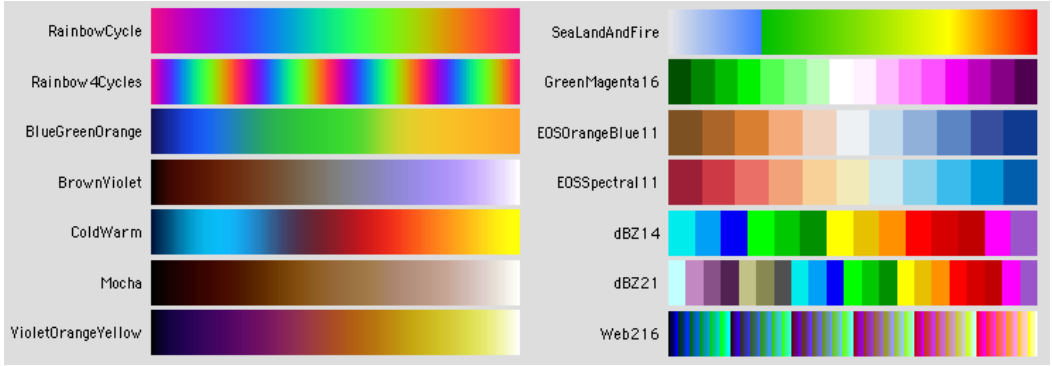
Color Table Name	Colors	Notes
Geo	256	Popular mapping color table for elevations. Sea level is around 50%.
Geo32	32	Quantized to classify elevations. Sea level is around 50%.
LandAndSea	255	Rapid color changes above sea level, which is at exactly 50%. Ocean depths are blue-gray.
LandAndSea8	8	Quantized, sea level is at about 22%.
Relief	255	Slower color changes above sea level, which is at exactly 50%. Ocean depths are black.
Relief19	19	Quantized, sea level is at about 47.5%.
PastelsMap	301	Desaturated rainbow-like colors, having a sharp green→yellow color change at sea level, which is around 66.67%. Ocean depths are faded purple.
PastelsMap20	20	Quantized. Sea level is at about 66.67%.
Bathymetry9	9	Colors for ocean depths. Sea level is at 100%.
BlackBody	181	Red → Yellow → Blue colors calibrated to black body radiation colors (neglecting intensity). The color table range is from 1,000 K to 10,000 K. Each color table entry represents a 50 K interval.
Spectrum	201	Rainbow-like colors calibrated to the visible spectrum when the color table range is set from 380 to 780 nm (wavelength). Each color table entry represents 2nm. Colors do not completely fade to black at the ends of the color table.
SpectrumBlack	476	Rainbow-like colors calibrated to the visible spectrum when the color table range is set from 355 to 830 nm (wavelength). Each color table entry represents 1 nm. Colors fade to black at the ends of the color table.
Cycles	201	Ten grayscale cycles from 0 to 100% to 0%.
Fiddle	254	Some randomized colors for “fiddling” with an image to detect faint details in the image.
Pastels	256	Desaturated Rainbow.



Igor Pro 6-Compatible Color Tables

Igor Pro 6 added 14 new color tables.

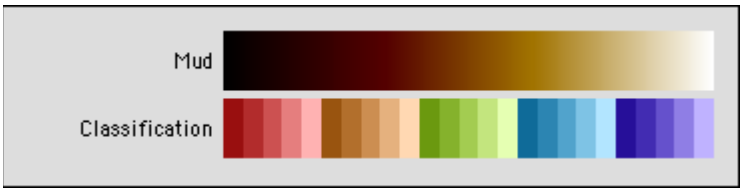
Color Table Name	Colors	Notes
RainbowCycle	360	Red, green, blue vary sinusoidally, each 120 degrees (120 values) out of phase. The first and last colors are identical.
Rainbow4Cycles	360	4 cycles with one quarter of the angular resolution.
BlueGreenOrange	300	Three-color gradient.
BrownViolet	300	Two-color gradient.
ColdWarm	300	Multicolor gradient for temperature.
Mocha	300	Two-color gradient.
VioletOrangeYellow	300	Multicolor gradient for temperature.
SeaLandAndFire	256	Another topographic table. Sea level is at 25%.
GreenMagenta16	16	Similar to the 14-color National Weather Service Motion color tables (base velocity or storm relative values), but friendly to red-green colorblind people.
EOSOrangeBlue11	11	Colors for diverging data (friendly to red-green colorblind people).
EOSSpectral11	11	Modified spectral colors (friendly to red-green colorblind people).
dBZ14	14	National Weather Service Reflectivity (radar) colors for Clear Air (-28 to +24 dBZ) or Precipitation (5 to 70 dBZ) mode.
dBZ21	21	National Weather Service Reflectivity (radar) colors for combined Clear Air and Precipitation mode (-30 to 70 dBZ).
Web216	216	The 216 “web-safe” colors, provides a wide selection of standard colors in a single color table. Intended for trace f(z) coloring using the ModifyGraph zColor parameter.



Igor Pro 6.2-Compatible Color Tables

Igor Pro 6.2 added 2 new color tables:

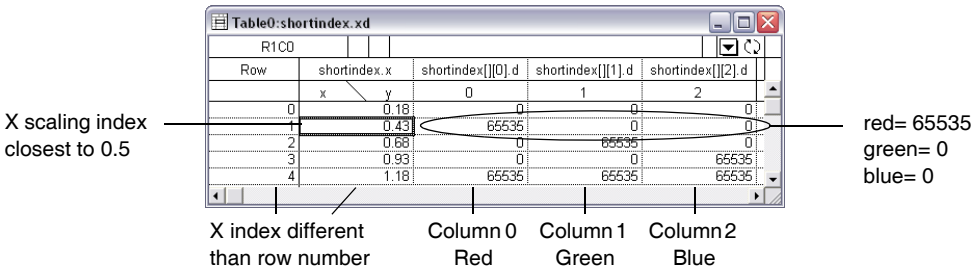
Color Table Name	Colors	Notes
Mud	256	Dark brown to white, without the pink cast of the Mocha color table. For Veeco atomic force microscopes.
Classification	25	5 hues for classification, 5 saturations for variations within each class.



Indexed Color Details

The data values contained in your matrix are used to select a color from a color index wave that you supply.

The color index wave must be a 2D wave with three columns containing red, green, and blue values that range from 0 (zero intensity) to 65535 (full intensity), and a row for each color.



Choosing a color from an X-scaled color index wave for image matrix Z value = 0.5

In the normal linear case, Igor finds the color for a particular matrix data value by choosing the row in the color index wave whose X index most closely matches the image matrix data value.

To choose the row, Igor converts the matrix data value into a row number as if executing:

```
colorIndexWaveRow= x2pnt(colorIndexWave,zImageValue)
```

which rounds to the nearest row and limits the result to the rows in the color index wave.

By setting the X scaling of the color index wave (Change Wave Scaling dialog), you can control how Igor maps the image matrix data value to a color. This is similar to setting the First Color and Last Color values for a color table.

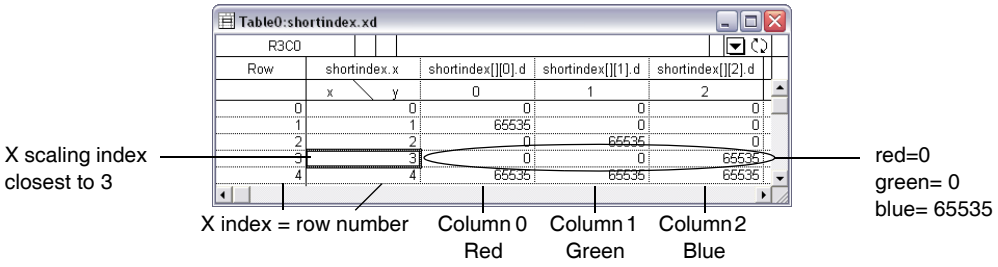
When the `ModifyImage log` parameter is set to 1, the colors are mapped using the log(x scaling) and log(image z) values this way:

```
colorIndexWaveRow = (nRows-1)*(log(zImageValue)-log(xMin))/(log(xmax)-
log(xMin))
```

where,

```
nRows = DimSize(colorIndexWave,0)
xMin = DimOffset(colorIndexWave,0)
xMax = xMin + (nRows-1) * DimDelta(colorIndexWave,0)
```

The images are drawn fastest if the image matrix is an integer number type, the color index wave has point X scaling (the X index is equal to row number), and the normal linear color mapping is used. In this case, row 0 contains the color for image matrix values equal to 0, row 1 contains the color for image values equal to 1, etc.



Choosing a color from a point-scaled color index wave for image matrix Z value = 3

Programming Note:

XOP (eXternal OPeration) code modules that read indexed image data from a file should create a color index wave to go with each image matrix.

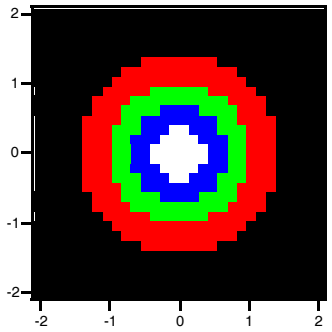
Example: Point-Scaled Color Index Wave

Here are the commands that created the point-scaled, unsigned 16-bit integer color index wave shown in the preceding table:

```
Make/O/W/U/N=(1,3) shortindex  // initially 1 row; more will be added
shortindex[0][]= {{0},{0},{0}}                 // black in first row
shortindex[1][]= {{65535},{0},{0}}             // red in new row
shortindex[2][]= {{0},{65535},{0}}             // green in new row
shortindex[3][]= {{0},{0},{65535}}             // blue in new row
shortindex[4][]= {{65535},{65535},{65535}}     // white in new row
```

These commands generate sample data and display it using the color index wave:

```
Make/O/N=(30,30)/B/U expmat    // /B/U makes unsigned byte image
SetScale/I x,-2,2,"" expmat
SetScale/I y,-2,2,"" expmat
expmat= 4*exp(-(x^2+y^2))      // test image ranges from 0 to 4
Display;AppendImage expmat
ModifyImage expmat cindex= shortindex
```



Programming Example: Adding Colors to a Built-In Color Table

A possible future extension to color tables would be to allow you to specify a special color to be used for out-of-range data. This example implements the feature with a color index wave generated by the **ColorTab2Wave** operation (see page V-57). The operation extracts colors from a built-in color table and places the red, green and blue values in a 3 column color index wave named M_colors.

We will insert special colors at the beginning and end of the color index wave, and then modify its X scaling so that the colors of the in-range data match those from the original color table.

First we create an image matrix containing data that ranges from -1 to 1 and display it over a range of -.5 to +.5 using the built-in Rainbow color table.

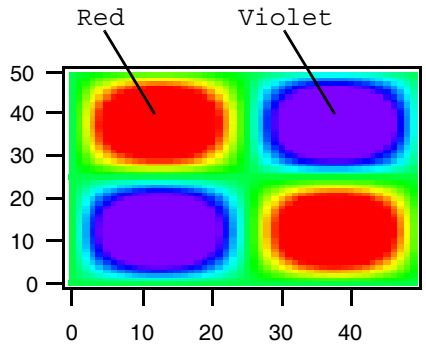
Tip: The commands in this example are in the Using Igor help file under the "Image Plots" topic. You can execute commands in a help file by selecting them and pressing Control or Ctrl plus Enter or Return.

```
Make/O/N=(50,50) fpmat=sin(x/8)*sin(y/8)
Display;AppendImage fpmat;ModifyImage fpmat ctab= {-0.5,0.5,Rainbow}
```

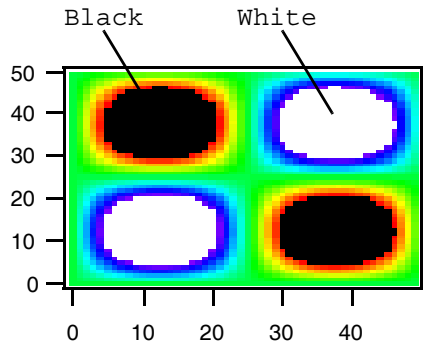
All fpmat values below -0.5 are red, and all values above +0.5 are violet. See the Before picture below (you'll have to believe us about the colors).

We want under-range data (values below -0.5) to show as black and over-range data as white, so we execute the following lines:

```
ColorTab2Wave Rainbow          // Reads built-in color table into...
Rename M_colors,mycolors       // ...M_colors which we rename
InsertPoints 0,1,mycolors      // Insert a row of zero (which is black)
k0= DimSize(mycolors, 0)       // Number of rows we have now (101 currently)
InsertPoints k0,1,mycolors     // Append a row; k0 is now index of last row
mycolors[k0][]= {{65535},{65535},{65535}}    // last row is white

// Set X scaling of mycolors so that red is still at -0.5, and violet is still at 0.5
k1 = (0.5-(-0.5)) / (k0-1)     // color table increment
SetScale/I x,-0.5-k1,0.5+k1,"",mycolors
ModifyImage fpmat cindex= mycolors
```



Before adding special first and last colors to a Rainbow color index wave



After adding black first color and white last color

You can use this same method by substituting your value of "First Color at Z =" for -0.5 and your value of "Last Color at Z=" for 0.5 in the example commands. This method works with any built-in color table.

Direct Color Details

Direct color images use a 3D wave with 3 color planes containing absolute values for red, green and blue. Generally, direct color waves will be either unsigned 8 bit integers or unsigned 16 bit integers.

For 8-bit integer waves, 0 represents zero intensity and 255 represents full intensity. For all other number types, 0 represents zero intensity but 65535 represents full intensity. Out-of-range values are clipped to the limits.

Try the following example, executing each line one at a time. For best results, set your monitor to thousands or millions of colors.

```
Make/O/B/U/N=(40,40,3) matrgb;Display;Appendimage matrgb
matrgb[][][0]= 127*(1+sin(x/8)*sin(y/8))  // specify red,   0-255
matrgb[][][1]= 127*(1+sin(x/7)*sin(y/6))  // specify green, 0-255
matrgb[][][2]= 127*(1+sin(x/6)*sin(y/4))  // specify blue,  0-255
```

```
Redimension/S matrgb    // switch to floating point, image turns black
matrgb*=256             // floating point must be larger to display correctly
```

Because the appearance of a direct color image is completely determined by the image data, the Modify Image Appearance dialog has no effect on direct color images, and the dialog appears blank.

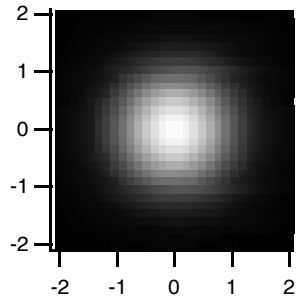
Creating Color Legends

You can create a color legend using a color scale annotation. You can find further details about creating legends using the Add Annotation dialog in Chapter III-2, **Annotations**, particularly in the **Legends** (see page III-52) and **Color Scales** (see page III-59) sections.

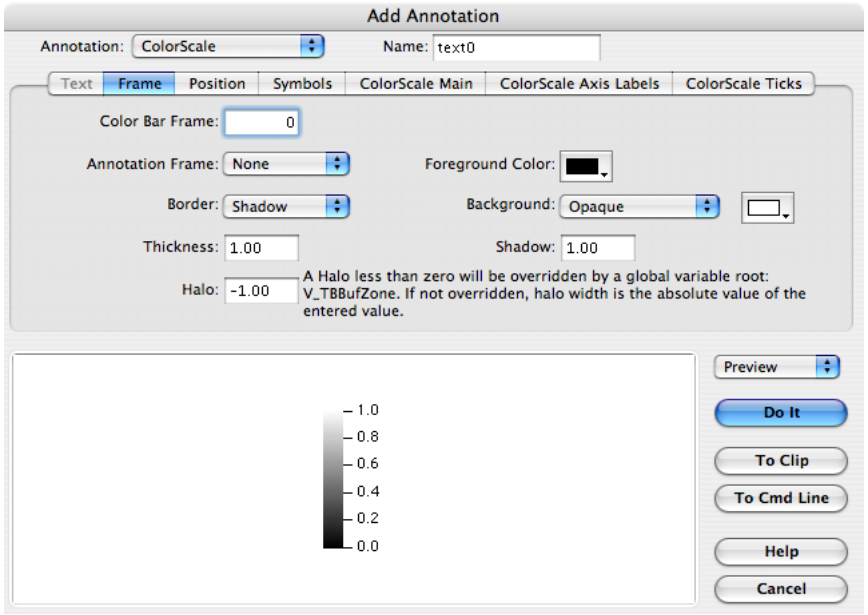
We will demonstrate with a simple image plot created by executing the following commands:

```
Make/O/N=(30,30) expmat
SetScale/I x,-2,2,"" expmat; SetScale/I y,-2,2,"" expmat
expmat= exp(-(x^2+y^2))       // data ranges from 0 to 1
Display;AppendImage expmat    // by default, left and bottom axes
ModifyGraph width={Plan,1,bottom,left},mirror=0
```

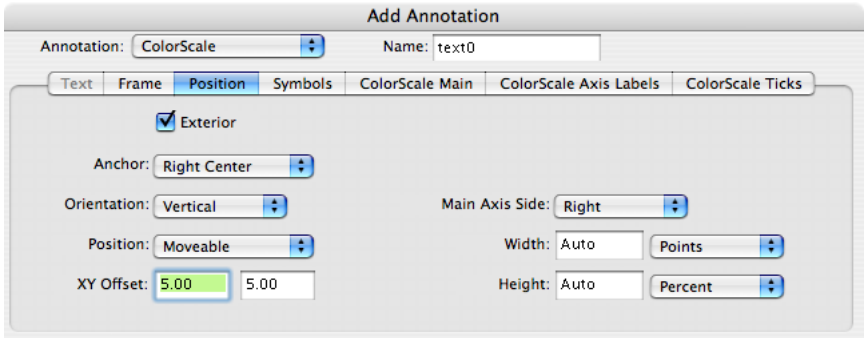
This creates the following image, using the autoscaled Grays color table:



Choose Add Annotation from the Graph menu. Choose “ColorScale” from the Annotation pop-up menu. Switch to the Frame tab, set the Color Bar Frame to 0 and the Annotation Frame to None.



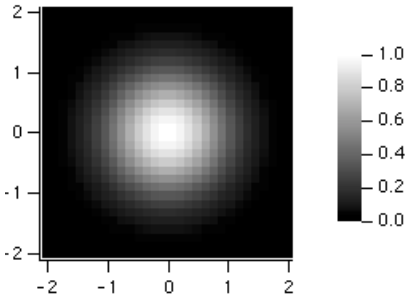
Switch to the Position tab, select Exterior and set the Anchor to Right Center:



Then click Do It. Igor will execute the following commands:

```
ColorScale/C/N=text0/F=0/A=RC/E image=expmat,frame=0.00
```

to generate the following image plot:



Double-click the color scale to edit it with the Modify Annotation dialog. Use the Position tab to change the color scale's orientation.

Image Instance Names

Igor identifies an image plot by the name of the wave providing Z values (the image matrix wave selected in the Z Wave list of the Image Plot dialogs). This "image instance name" is used in commands that modify the image plot.

In this example the image instance name is "zw":

```
Display; AppendImage zw              // new image plot
ModifyImage zw ctab={*,*,BlueHot}    // change color table
```

In the unusual case that a graph contains two image plots of the same data (to show different subranges of the data side-by-side, for example), an instance number must be appended to the name to modify the second plot:

```
Display; AppendImage zw; AppendImage/R/T zw  // two image plots
ModifyImage zw ctab={*,*,RedWhiteBlue}       // change first plot
ModifyImage zw#1 ctab={*,*,BlueHot}          // change second plot
```

The Modify Image Appearance dialog generates the correct image instance name automatically. Image instance names work much the same way wave instance names for traces in a graph do. See **Instance Notation** on page IV-16.

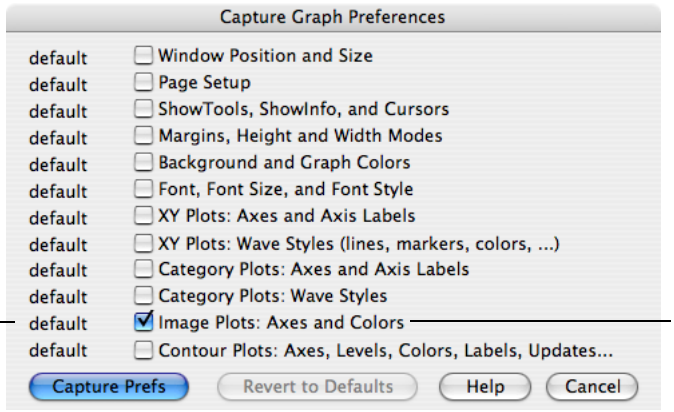
Programming Note

The **ImageNameList** function (see page V-274) returns a string list of image instance names. Each name corresponds to one image plot in the graph. The **ImageInfo** function (see page V-262) returns information about a particular named image plot.

Image Preferences

You can change the default appearance of image plots by "capturing" preferences from a "prototype" graph containing image plots. Create a graph containing an image plot (or plots) having the settings you use most often. Then choose Capture Graph Prefs from the Graph menu. Select the Image Plots category, and click Capture Prefs.

The factory default settings for the Image Plot category are currently in effect.



The Image Appearance settings and the axis settings used for the image plot in the target graph will be captured as the new preferred settings.

Preferences are normally in effect only for *manual* operations, not for automatic operations from Igor procedures. Preferences are discussed in more detail in Chapter III-17, **Preferences**.

The Image Plots category includes both Image Appearance settings and axis settings.

Image Appearance Preferences

The captured Image Appearance settings are automatically applied to an image plot when it is first created (provided preferences are turned on). They are also used to preset the Modify Image Appearance subdialog of the New Image Plot dialog.

If you capture the Image Plot preferences from a graph with more than one image plot, the first image plot appended to a graph gets the settings from the image first appended to the prototype graph. The second image plot appended to a graph gets the settings from the second image plot appended to the prototype graph, etc. This is similar to the way XY plot wave styles work.

Image Axis Preferences

Only axes used by the image plot have their settings captured. Axes used solely for an XY, category, or contour plot are ignored.

The image axis preferences are applied only when axes having the same name as the captured axis are created by an AppendImage command. If the axes existed before AppendImage is executed, they will not be affected by the image axis preferences.

The names of captured image axes are listed in the X Axis and Y Axis pop-up menus of the New Image Plot and Append Image Plot dialogs. This is similar to the way XY plot axis preferences work.

For example, suppose you capture preferences for an image plot using axes named "myRightAxis" and "myTopAxis". These names will appear in the X Axis and Y Axis pop-up menus in image plot dialogs.

- If you choose them in the New Image Plot dialog and click Do It, a graph will be created containing *newly-created* axes named "myRightAxis" and "myTopAxis" and having the axis settings you captured.
- If you have a graph which already uses axes named "myRightAxis" and "myTopAxis" and choose these axes in the Append Image Plot dialog, the image will be appended to those axes, as usual, but no captured axis settings will be applied to these *already-existing* axes.

You can capture image axis settings for the standard left and bottom axes, and Igor will save these separately from left and bottom axis preferences captured for XY, category, and contour plots. Igor will use the image axis settings for AppendImage commands only.

How to Use Image Preferences

Here is our recommended strategy for using image preferences:

1. Create a new graph containing a single image plot. Use the axes you will normally use, even if they are left and bottom. You can use other axes, too (select New Axis in the New Image Plot and Append Image Plot dialogs).
2. Use the Modify Image Appearance, Modify Graph, and Modify Axis dialogs to make the image plot appear as you prefer.
3. Choose Capture Graph Prefs from the Graph menu. Select the Image Plots category, and click Capture Prefs.

Image Plot Shortcuts

Since image plots are drawn in a normal graph, all of the **Graph Shortcuts** (see page II-306) apply. Here we list those which apply specifically to image plots.

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To modify the appearance of the image plot as a whole	Control-click in the plot area of the graph and choose Modify Image from the pop-up menu.	Right-click in the plot area of the graph and choose Modify Image from the pop-up menu.

References

Light, Adam, and Patrick J. Bartlein, The End of the Rainbow? Color Schemes for Improved Data Graphics, *Eos*, *85*, 385-391, 2004.

See also <http://geography.uoregon.edu/datagraphics/color_scales.htm>.

Wyszecki, Gunter, and W. S. Stiles, *Color Science: Concepts and Methods, Quantitative Data and Formula*, 628 pp., John Wiley & Sons, 1982.

Chapter
II-16

Page Layouts

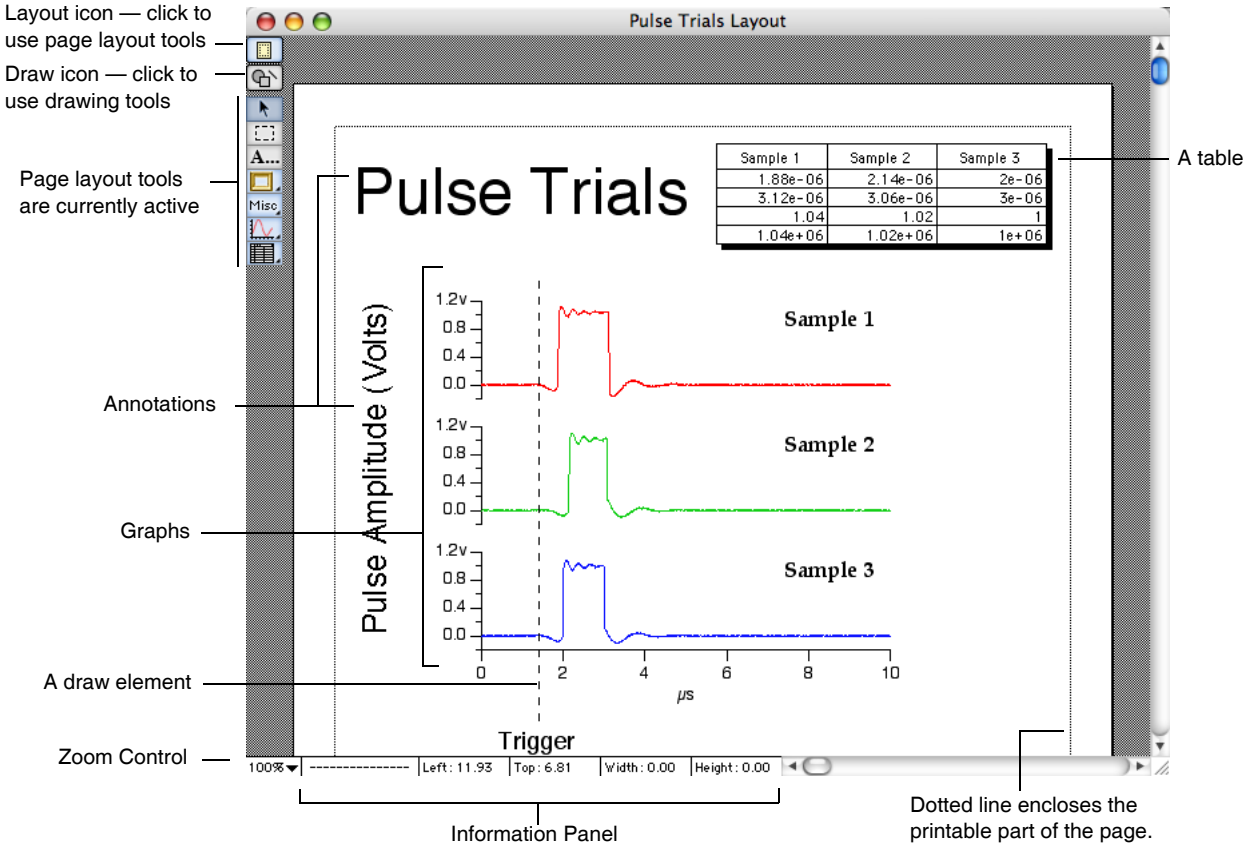
Overview

A page layout, or layout for short, is a type of window that you can use to compose pages containing:

- graphs
- tables
- annotations (textboxes and legends)
- pictures
- drawing elements (lines, arrows, rectangles, polygons, etc.)

Each layout represents one page. You can have as many layouts as memory allows.

Here is an example of a layout window.



A page layout has a number of layers. One layer, the layout layer, is for graphs, tables, annotations and pictures. The other layers are for drawing elements. Drawing is discussed in detail in Chapter III-3, **Drawing**. This chapter is primarily devoted to the layout layer.

Here are the notable features of page layouts.

- You can combine graphs, tables, pictures, annotations and drawing elements.
- Graphs, tables and legends in layouts are updated automatically.
- Complex graphs can be quickly and smoothly positioned.
- Layouts print at the full resolution of the printer.
- You can export all or part of a layout to another program.

There are two ways to add a graph or table to the layout layer:

- By creating a graph or table *object*. An object is a representation of a separate standalone graph or table window. Layout objects are described under **Objects in the Layout Layer** on page II-370.

- By creating an embedded graph or table *subwindow*. A subwindow is a self-contained graph or table embedded in a layout window. Embedded subwindows are described under **Subwindows in the Layout Layer** on page II-372.

The subwindow is a power-user feature added in Igor Pro 5. It is described in detail in Chapter III-4, **Embedding and Subwindows** and can not be effectively used without a careful reading of that chapter. Graph and table objects are less powerful but simpler to use and more intuitive. We recommend using graph and table objects until you have had time to read and understand Chapter III-4.

In this chapter, the term "object" refers to graph, table, annotation and picture objects, not to graph or table subwindows.

Memory Usage in Page Layouts

Igor uses techniques that make the layout layer operate quickly and smoothly even when you are working with large, complex graphs. These techniques require that Igor store an image of the page in memory.

If you view a layout in color and then change the number of colors displayed on your monitor, Igor will update the layout using the new number of colors. If Igor runs out of memory while doing this, it will reduce the page magnification so that less memory is required.

Layout Background Color

You can choose a background color for a page layout. This is useful for creating slides.

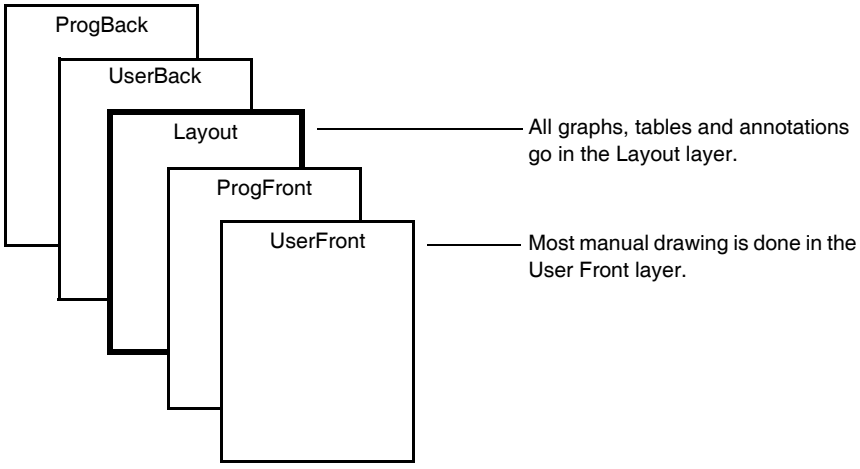
You can specify a background color for the page layout by:

- Using the Background Color submenu in the Layout menu.
- Using the Background Color submenu in the Misc pop-up menu.
- Using the NewLayout command line operation.
- Using the ModifyLayout command line operation.

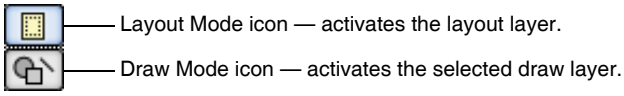
The background color is white by default. If you wish, after selecting a background color, you can capture your preferred background color by choosing Capture Layout Prefs from the Layout menu.

Layers

A page in a layout has five layers. There is one layer for layout objects and four layers for drawing elements.



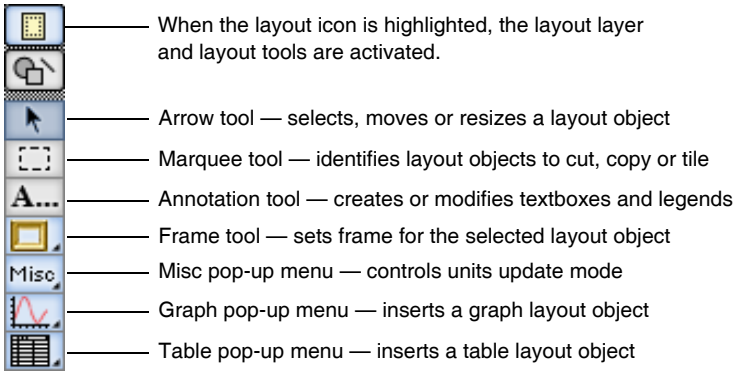
The two icons in the top-left corner of the layout window control whether you are in layout mode or drawing mode.



The layout layer is most useful for presenting multiple graphs and for annotations that refer to multiple graphs. The drawing layers are useful for adding simple graphic elements such as arrows between graphs.

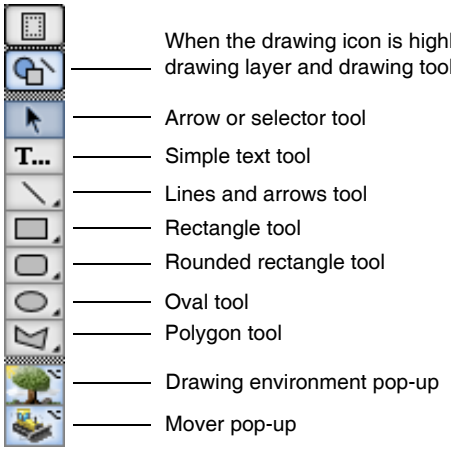
Activating the Layout Layer

When you click the layout icon, the layout layer is activated. You can use the layout tools to add objects to or modify objects in the layout layer only.



Activating the Current Drawing Layer

When you click the drawing icon, the current drawing layer is activated. You can use the drawing tools to add elements to or modify elements in the current drawing layer only.



Changing the Current Drawing Layer

Initially, the UserFront drawing layer will be the current drawing layer. To select a different drawing layer, press Option (*Macintosh*) or Alt (*Windows*) and click the drawing environment pop-up icon.

Option-click or Alt-click the drawing environment icon  to get the Draw Layer pop-up menu:

You may never need to use the Drawing Layer pop-up menu. Most users will need to use just the layout layer and the UserFront drawing layer. The ProgFront and ProgBack layers are intended to be used from Igor procedures only.

If you click an element that is not in the active layer, Igor will ignore the click.

DelayUpdate and Drawing Commands

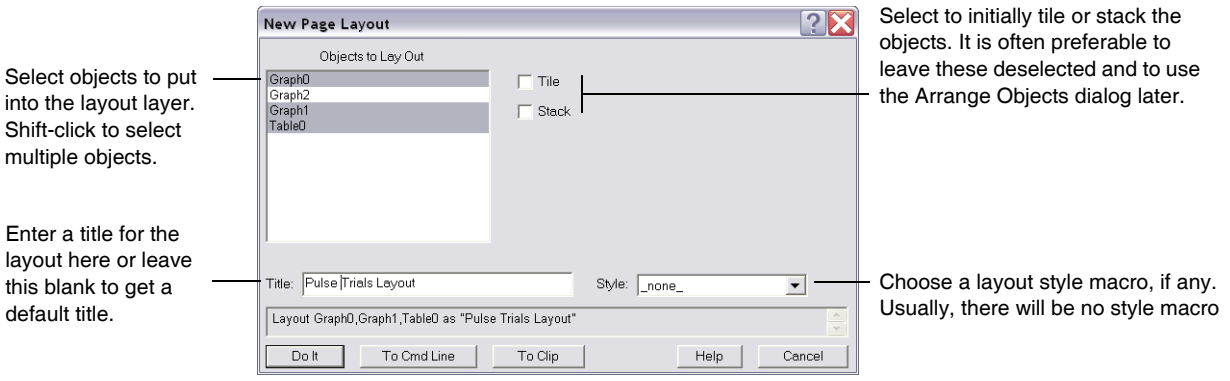
There is a DelayUpdate setting, accessible via the Misc pop-up menu, that controls when the layout will be redrawn. If you execute drawing commands from the command line or from a procedure and if the DelayUpdate setting is on, the layout will not be updated until you make it the active window. Therefore, if you want to type drawing commands in the command line and see the effect in the layout immediately, turn the DelayUpdate setting off.

For Further Information on Drawing Layers

The drawing layers function nearly identically in graph, page layout and control panel windows. For details on drawing, see Chapter III-3, **Drawing**. The rest of this chapter discusses drawing only to describe behavior that is unique to layout windows.

Creating a Layout

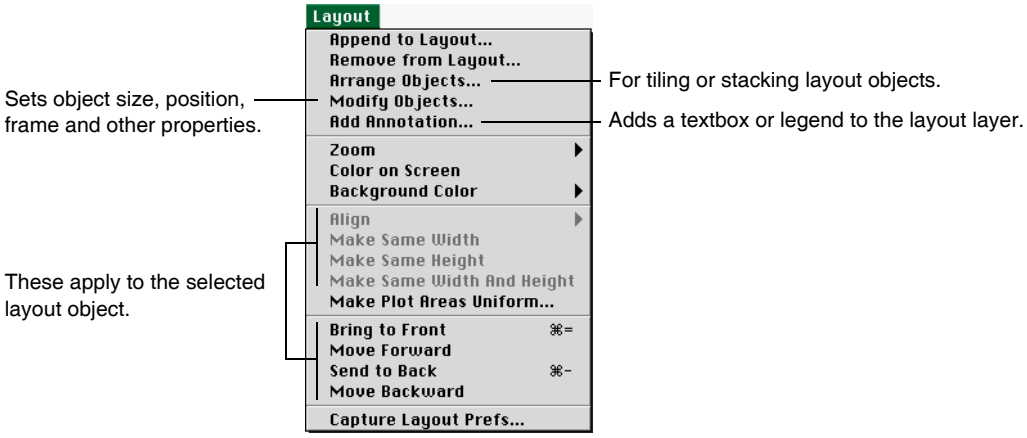
You create a layout by choosing New Layout from the Windows menu.



You can create a layout with no objects in it by clicking Do It without selecting any objects. You can append objects to the layout later.

Layout Menu

The Layout menu contains items that apply to page layout windows only. It appears in the menu bar only when the active window is a layout.



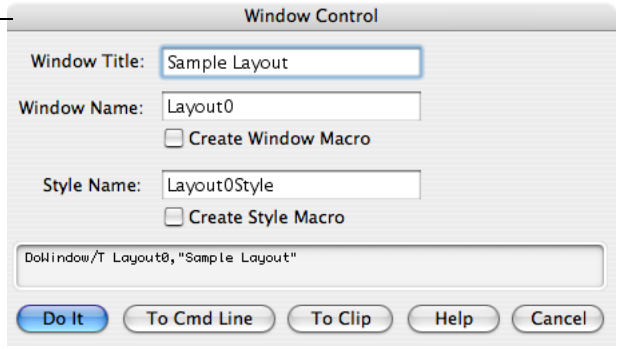
Layout Names and Titles

Every layout that you create has a name. This is a short Igor-object name that you or Igor can use to reference the layout from a command or procedure. When you create a new layout, Igor assigns it a name of the form Layout0, Layout1 and so on. You will most often use a layout's name when you kill and recreate the layout, see **Killing and Recreating a Layout** on page II-369.

A layout also has a title. The title is the text that appears at the top of the layout window. Its purpose is to identify the layout visually. It is not used to identify the layout from a command or procedure. The title can consist of any text, up to 255 characters.

You can change the name and title of a layout using the Window Control dialog. This dialog is a collection of assorted window-related things. Choose Window Control from the Control submenu of the Windows menu.

The Window Control dialog changes a layout's title and name.



Hiding and Showing a Layout

You can hide a layout by Shift-clicking the close button.

If the Minimize Is Hide checkbox is selected in the Miscellaneous Settings dialog (Misc menu), you can hide a layout by clicking its minimize icon and hide all layouts by clicking the minimize button while pressing Option (*Macintosh*) or Alt (*Windows*).

You can show a layout by choosing its name from the Windows→Layouts submenu.

Killing and Recreating a Layout

Igor provides a way for you to kill a layout and then later to recreate it. This temporarily gets rid of a layout that you expect to be of use later.

You kill a layout by clicking the layout window's close button or by using the Close item in the Windows menu. When you kill a layout, Igor offers to create a **window recreation macro**. Igor stores the window recreation macro in the procedure window of the current experiment. You can invoke the window recreation macro later to recreate the layout. The name of the window recreation macro is the same as the name of the layout.

For further details, see **Closing a Window** on page II-59 and **Saving a Window as a Recreation Macro** on page II-61.

Page Setups

The page setup is a collection of information created by the printer driver. It controls the page orientation, the dimensions of the page, and the size of page margins. You can modify it using the Page Setup dialog via the File menu or using the **PrintSettings** operation on page V-503.

Each page layout has its own associated page setup. When you create a new page layout, Igor creates a new, default page setup. You can change the default page setup to get the page orientation and margins that you

prefer using the Capture Layout Prefs item in the Layout menu. Igor uses this captured page setup when you create a layout interactively. To use the captured page setup from an Igor procedure, see the **Preferences** operation on page V-497.

When you close a layout window, Igor asks if you want to create a layout recreation macro. If you do create the macro, you can execute it later to recreate the layout. Igor then reuses the layout's original page setup for the recreated layout.

Page setups are stored in the experiment file when you save the current experiment.

Changing Printers

Page setups are customized for each printer. When you change printers and then attempt to print, some printer drivers try to convert the page setup for the old printer into an equivalent page setup for the new printer. This conversion is not always accurate. If you find that printing behaves unexpectedly when you change printers, then you will need to use the Page Setup dialog to fix the page setup before you print.

Changing Computer Platforms

When you transfer an experiment saved by Igor Pro 3.0 or earlier from Macintosh to Windows, page setups are lost. Igor will use a default setup for all page layouts in the experiment.

When you transfer an Igor Pro 3.1 (or later) experiment from one platform to another, page setup records are only partially preserved. Igor attempts to preserve the page orientation and margins.

There is a more detailed discussion of this issue in Chapter III-15, **Platform-Related Issues**.

Zooming

You can zoom the page in to 200% or out to 50% or 25%. Use the Zoom submenu in the Layout menu or the Zoom pop-up menu in the lower-left corner of the layout window.

By zooming out you see the entire page at once. You can zoom in to place drawing elements with higher precision.

Igor stores the position of layout objects with a precision of one point. Therefore, zooming in does not allow you to position them more precisely. Also, when you zoom in, Igor does not redraw graphs and other objects in the layout layer; it merely shrinks or expands a stored representation of the object. However, Igor does redraw graph and table subwindows and drawing elements.

Objects in the Layout Layer

The layout layer can handle four kinds of objects: graphs, tables, annotations and pictures. This table shows how you can add each of these objects to the layout layer.

Object Type	To Add Object to the Layout Layer
Graph	Use the Graph pop-up menu in the layout window. Use the Append to Layout dialog. Use the **AppendLayoutObject** operation in user-functions, otherwise use the AppendToLayout operation.
Table	Use the Table pop-up menu in the layout window. Use the Append to Layout dialog. Use the **AppendLayoutObject** operation.

Object Type	To Add Object to the Layout Layer
Annotations	Click the text ("A") tool and then click in the page area. Use the Add Annotation dialog. Use the **TextBox** or **Legend** operations.
Pictures	Paste from the Clipboard. Use the Pictures dialog (Misc menu). Use the **AppendLayoutObject** operation if the picture already exists in the current experiment's picture collection.

Layout Object Names

Each object in the layout layer has a name so that you can manipulate it from the command line or from an Igor procedure as well as with the mouse. When you position the cursor over an object, its name, position and dimensions are shown in the info panel at the bottom of the layout window.

For a graph or table, the object name is the same as the name of the graph or table window. For an annotation, the object name is determined by the Textbox or Legend operation that created the annotation. When you paste a picture from the Clipboard into a page layout, Igor automatically gives it a name like PICT_0 and adds it to the current experiment's picture collection.

Layout Object Properties

This table shows the properties of each object in the layout layer.

Object Property	Comment
Left coordinate	Measured from the left edge of the paper. Set using mouse or Modify Objects dialog.
Top coordinate	Measured from the top edge of the paper. Set using mouse or Modify Objects dialog.
Width	Set using mouse or Modify Objects dialog.
Height	Set using mouse or Modify Objects dialog.
Frame	None, single, double, triple, or shadow. Set using Frame pop-up menu or Modify Objects dialog.
Transparency	Set using Modify Objects dialog.
Fidelity	Set using Modify Objects dialog. This affects only what happens when you resize an object. If you resize a high fidelity object, Igor redraws the object completely at the new size. If you resize a low fidelity object, Igor stretches an existing picture of the object to fit the new size. As of Igor Pro 6.1, the fidelity setting no longer affects graph objects.

All of the properties can also be set using the ModifyLayout operation from the command line or from an Igor procedure.

Some special cases involving layout object transparency are discussed under **Problems with Layouts** on page II-389.

Dummy Objects

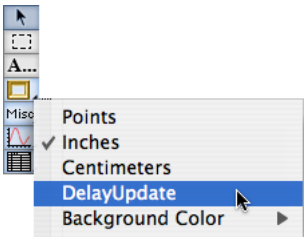
If you append a graph or table to the layout layer, this creates a corresponding layout object. If you then kill the graph or table window, the layout object remains and is said to be a "dummy object". A dummy object can be moved, resized or changed just as any other object.

If you later recreate the graph or table window or create a new graph or table with the same name as the original, the object is reassociated with the window and ceases to be a dummy object.

Automatic Updating of Layout Objects

Graph and table objects are dynamic. When a graph or table changes, Igor automatically updates the corresponding layout object. Also, if you change the symbol for a wave in a graph and if that symbol is used in a layout legend, Igor automatically updates the legend.

Normally, Igor waits until the layout window is activated before doing an automatic update. You can force Igor to do the update immediately by deselecting the DelayUpdate item in the Misc pop-up menu in the layout's tool palette.



See **DelayUpdate and Drawing Commands** on page II-368 for information on how DelayUpdate affects drawing elements.

Subwindows in the Layout Layer

The layout layer can handle two kinds of subwindows: graphs and tables. To add a subwindow to a layout:

1. Activate the layout layer by clicking the layout icon.
2. Select the marquee tool (dashed-line rectangle).
3. Drag out a marquee.
4. Click inside the marquee and choose New Graph Here or New Table Here.

You can also create a subwindow by right-clicking (*Windows*) or Control-clicking (*Macintosh*) while in drawing mode and choosing an item from the New submenu.

You can convert a graph or table object to an embedded subwindow by right-clicking (*Windows*) or Control-clicking (*Macintosh*) while in layout mode and choosing Convert To Embedded. Note that a graph containing a control panel or controls cannot be converted into an embedded graph, even though a graph object with controls or control panels can be added to a layout. Such a graph object does not display the controls or control panels, however.

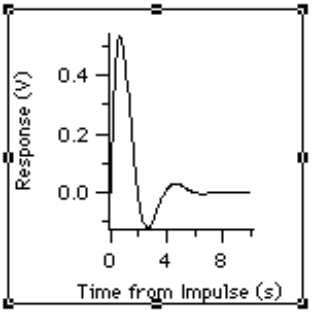
You can convert a graph or table subwindow to a graph or table object by right-clicking (*Windows*) or Control-clicking (*Macintosh*) while in layout mode and choosing Convert To Graph And Object or Convert To Table and Object. In a graph window, you must click in the graph background, away from any traces or axes.

The subwindow is a power-user feature added in Igor Pro 5. It is described in detail in Chapter III-4, **Embedding and Subwindows** and can not be effectively used without a careful reading of that chapter.

Layout Layer Tool Palette

Arrow Tool 

When you click the arrow tool, it becomes highlighted. The arrow tool is used to select, move or resize a graph, table, annotation or picture object. To select an object, click it. Adjustment handles appear on the selected object.



When you position the cursor over an object, the info panel shows the name of the object under the cursor. If you drag, the cursor changes to a hand and the object will follow the cursor as you drag it. While you drag, the info panel shows the left and top coordinates of the object as well as its width and height.

If you press Shift while dragging an object, the direction of movement is constrained either horizontally or vertically, depending on the first motion direction. If you momentarily release Shift and press it again, you can change the direction of constraint.

By dragging the selected object's handles, you can set the object's width and height. While you drag, the info panel shows the width and height of the object. If the object you are adjusting is a table, the info panel also shows the width and height of the table in terms of rows and columns. If the object you are adjusting is an annotation or picture, the info panel also shows the width and height of the object in terms of percent of its unadjusted size.

If you press Shift while dragging a corner handle, the resize will be proportional so that the object will maintain its original aspect ratio. The resize may deviate from precisely proportional if the page layout magnification is less than 100 percent.

When you adjust the size of a table and then release the mouse, the table is auto-sized to an appropriate integral number of rows and columns. To disable the auto-sizing, press Option (*Macintosh*) or Alt (*Windows*) while resizing the table.

You can quickly force an annotation or picture back to its unadjusted size (100% by 100%) by pressing Option (*Macintosh*) or Alt (*Windows*) and double-clicking the object. You can auto-size a table using the same method.

Double-clicking an object while the arrow tool is selected brings up the Modify Objects dialog, described in **Modifying Layout Objects** on page II-380. Use this dialog to set the object's properties.

When two or more objects are selected, you can align them using the Align submenu in the Layout menu. You can also make the widths, heights, or widths and heights of the selected objects the same using items in the Layout menu. In all of these actions, the first object that you select is used as the basis for aligning or resizing other objects. Click in a blank area of the page to deselect all objects. Then click the object whose position or size you want to replicate. Now Shift-click to select additional objects. Finally, choose the desired action from the Layout menu.

While an object is selected, you can control its front-to-back ordering in the layout layer by choosing Bring to Front, Move Forward, Send to Back or Move Backward from the Layout menu. This changes the order of objects *within the layout layer only*. It has no effect on the drawing layers.

If you select a graph or table object, you can then double-click the name of the object in the info panel at the bottom of the layout window. This activates the associated graph or table window.

After selecting an object by clicking it, you can select additional objects by Shift-clicking. If you Shift-click an object that is already selected, it becomes deselected.

You can also select multiple objects by drag-clicking. Start by clicking in an area of the page where there are no objects. With the mouse button held down, drag the mouse diagonally. While you do this, Igor displays a gray selection rectangle. Drag the rectangle until it completely encloses all of the objects that you want to select. You can select additional objects by pressing Shift while drag-selecting unselected objects. You can deselect objects by pressing Shift while drag-selecting selected objects. You can also deselect objects by pressing Shift and clicking them.

With multiple objects selected, you can perform the following actions:

- Delete the objects, using the Delete key or the Edit→Clear menu item.
- Copy the objects, using Edit→Copy.
- Cut the objects, using Edit→Cut.
- Drag the objects to a new location.
- Nudge the objects with the arrow keys.
- Change the frame on the objects using the frame tool.

You can not do the following actions on multiple objects:

- Change the order of the objects in the object list (move to front, move to back).
- Adjust the size of the objects using the mouse.

Marquee Tool 

When you click the marquee tool, it becomes highlighted and the cursor changes to a crosshair. You can use the marquee tool to identify multiple objects for cutting, copying, clearing or arranging. You can also use it to indicate a part of the layout for export to another application and to control pasting of objects from the Clipboard.

To use the marquee tool, click the mouse and drag it diagonally to indicate the region of interest. Igor displays a dashed outline around the region. This outline is called a marquee. A marquee has handles and edges that allow you to refine its size and position.

To refine the size of the marquee, move the cursor over one of the handles. The cursor changes to a double arrow which shows you the direction in which the handle adjusts the edge of the marquee. To adjust the edge, simply drag it to a new position.

To refine the position of the marquee, move the cursor over one of the edges away from the handles. The cursor changes to a hand. To move the marquee, drag it.

To make it possible to export any section of a layout, an object is considered selected if it intersects the marquee. This is in contrast to selection with the arrow tool, which requires that you completely enclose the object.

This table shows how to use the marquee.

To Accomplish This	Do This
Cut, copy or clear multiple objects	Drag a marquee around them and then use the Edit menu to cut, copy or clear.
Paste objects into a particular area	Drag a marquee where you want to paste and then use the Edit menu to paste.
Tile or stack objects	Drag a marquee to indicate the area into which you want to tile or stack and then choose Arrange Objects from the Layout menu.
Export a section of the layout as a picture	Drag a marquee to indicate the section that you want to export and then choose Export Graphics from the Edit menu (to use the Clipboard) or choose Save Graphics from the File menu (to save in a disk file).

When you click inside the marquee Igor presents a pop-up menu, called the Layout Marquee menu, from which you can choose Cut, Copy, Paste, or Clear. This cuts, copies, pastes or clears the all objects that intersect the marquee. These marquee items do the same thing as the corresponding items in the Edit menu.

The Marquee menu also contains items that allow you to insert a graph or table subwindow.

It is possible to add your own menu items to the Layout Marquee menu. See **Marquee Menus** on page IV-119 for details.

See **Copying Objects from the Layout Layer** on page II-387 and **Pasting Objects into the Layout Layer** on page II-387 for more details on copying and pasting. See **Arranging Objects** on page II-385 for details on tiling and stacking.

When the marquee tool is selected, any selected object is deselected. Double-clicking while the marquee tool is selected has no effect.

Annotation Tool 

When you click the annotation tool, it becomes highlighted and the cursor changes to an I-beam. The annotation tool creates new annotations or modifies existing annotations. Annotations include textboxes and legends.

Clicking an existing annotation invokes the Modify Annotation dialog. Clicking anywhere else on the page invokes the Add Annotation dialog which you use to create a new annotation. See **Annotations in the Layout Layer** on page II-381 for more details.

Frame Pop-Up Menu 

When an object is selected, you can change its frame by selecting an item from the Frame pop-up menu. Each object can have no frame or a single, double, triple or shadow frame.

When you change the frame of a graph, table or picture object, its outer dimensions (width and height) do not change. Since the different frames have different widths, the inner dimensions of the object *do* change. In the case of graphs this is usually the desired behavior. For tables, changing the frame shows a nonintegral number of rows and columns. You can restore the table to an integral number of rows and columns by pressing Option (*Macintosh*) or Alt (*Windows*) and double-clicking the table. For pictures, changing the frame slightly resizes the picture to fit into the new frame. To restore the picture to 100% sizing, press Option (*Macintosh*) or Alt (*Windows*) and double-click the picture.

When you change the frame of an annotation object, Igor *does* change the outer dimensions of the object to compensate for the change in width of the frame.

Misc Pop-Up Menu 

The Misc pop-up menu adjusts some miscellaneous settings related to the layout.

You can choose Points, Inches, or Centimeters. This sets the units used in the info panel.

You can enable or disable the DelayUpdate item. If DelayUpdate is on, when a graph or table which corresponds to an object in the layout changes, the layout is not updated until you activate it (make it the front window). If you disable DelayUpdate then changes to graphs or tables are reflected immediately in the layout. This also affects drawing commands. If you want to see the effect of drawing commands immediately, turn the DelayUpdate setting off.

DelayUpdate does not affect embedded graph and table subwindows.

Prior to Igor Pro 6.10, DelayUpdate was a per-document setting. It is now a global setting that affects all existing and future layouts instead of just the layout you set.

You can use the Background Color submenu to change the layout's background color. See **Layout Background Color** on page II-366 for details.

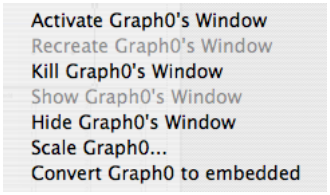
Graph Pop-Up Menu 

The Graph pop-up menu provides a handy way to append a graph object to the layout layer. It contains a list of all the graph windows that are currently open. Choosing the name of a graph appends the graph object to the layout layer. The initial size of the graph object in the layout is taken from the size of the graph window.

Table Pop-Up Menu 

The Table pop-up menu provides a handy way to append a table object to the layout layer. It contains a list of all the table windows that are currently open. Choosing the name of a table appends the table object to the layout layer.

The Layout Layer Contextual Menu



When the layout layer is active, Control-clicking (*Macintosh*) or right-clicking (*Windows*) displays the Layout Layer contextual menu. The contents of the menu depend on whether you click directly on an object or on a part of the page where there is no object.

Activate Object's Window

This item activates the corresponding graph or table window.

Recreate Object's Window

This item recreates the corresponding graph or table window by running the window recreation macro that was created when the window was killed.

Kill Object's Window

This item kills the corresponding graph or table window. Before it is killed, you will see a dialog you can use to create or update its window recreation macro.

If you press Option (*Macintosh*) or Alt (*Windows*) while selecting this item, the window will be killed with no dialog and without creating or updating the window recreation macro. Any changes you made to the window will be lost so use this feature carefully.

Windows Only:Pressing Alt while right-clicking does not work. To use the Alt key feature, proceed as follows: right-click, then select the menu item without releasing the left mouse button, then press Alt, then release the left mouse button.

Show Object's Window

This item shows the corresponding graph or table window if it is hidden.

Hide Object's Window

This item hides the corresponding graph or table window if it is hidden.

Scale Object

This item changes the size of the layout object in terms of percent of its current size or percent of its normal size. Although this can work on any type of object, it is most useful for scaling pictures relative to their normal size.

For a picture or annotation object, "normal" size is the inherent size of the picture or annotation before any shrinking or expanding. For a graph or table object, "normal" size means the size of the corresponding graph or table window.

Regardless of the scaling values you enter, Igor does not allow the size of any object to exceed the size of the entire page.

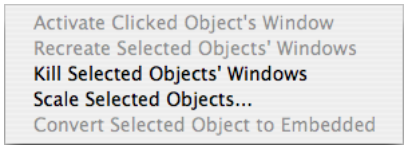
If a graph's size is hardwired via the Modify Graph dialog, the corresponding layout object can not be scaled.

Tip: You can quickly return a picture or annotation to its normal size by double-clicking it while pressing Option (*Macintosh*) or Alt (*Windows*).

Convert Object to Embedded

This item converts a graph or table object to an embedded subwindow. In doing so, the separate graph or table window which the object represented is killed, leaving just the embedded subwindow.

If you Control-click or right-click on a part of the page where there is no object, the Layout contextual menu looks like this:



Recreate Selected Objects' Windows

This item runs the recreation macro for each selected graph or table object for which the corresponding window was killed. It does nothing for selected picture or annotation objects.

Kill Selected Objects' Windows

This item kills the window corresponding to each selected graph or table object. It does nothing for selected picture or annotation objects. Before each window is killed, you will see a dialog you can use to create or update its window recreation macro.

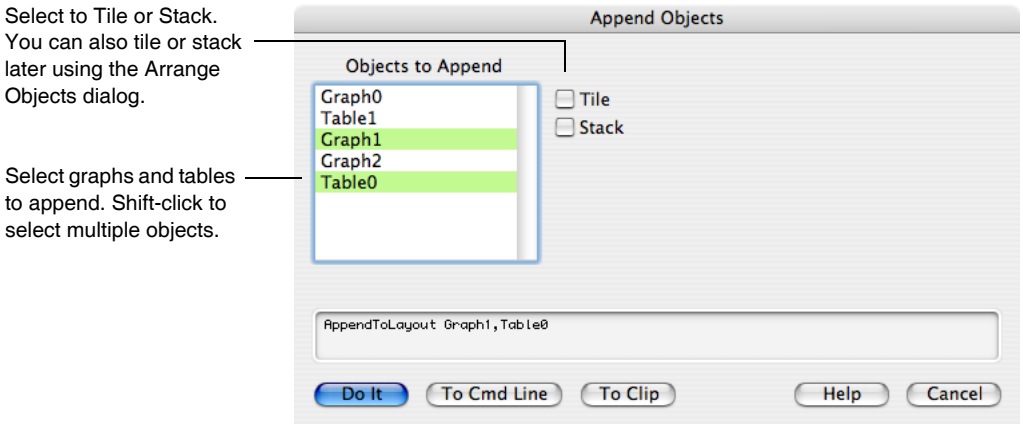
If you press Option (*Macintosh*) or Alt (*Windows*) while selecting this item, each window will be killed with no dialog and without creating or updating the window recreation macro. Any changes you made to the window will be lost so use this feature carefully.

Scale Selected Objects

This item changes the size of each selected layout object in terms of percent of its current size or percent of its normal size.

Appending a Graph or Table to the Layout Layer

You can append graphs and tables to a layout by choosing the Append to Layout item from the Layout menu.



You can also append a graph or table using the pop-up menus in the layout's tool palette.

Inserting a Picture in the Layout Layer

You can insert a picture that you have created in another application, for example a drawing program, into the layout layer. (You can also insert a picture into the drawing layers. This is recommended if you wanted to group the picture with other drawing elements.)

You can insert a picture in a layout by copy-and-paste or by loading from a file. When loading from a file you must go through Igor's picture collection (Misc→Pictures) to load and then place the picture.

Here are the supported picture formats:

Format	How To Place	Notes
PDF	Paste or use Misc→Pictures	Macintosh only
EMF (Enhanced Metafile)	Paste or use Misc→Pictures	Windows only
BMP (bitmap)	Use Misc→Pictures	Windows only BMP is sometimes called DIB (device-independent bitmap).
PNG (Portable Network Graphics)	Use Misc→Pictures	Cross-platform bitmap format
JPEG	Use Misc→Pictures	Cross-platform bitmap format
TIFF (Tagged Image File Format)	Use Misc→Pictures	Cross-platform bitmap format
EPS (Encapsulated PostScript)	Use Misc→Pictures	High resolution vector format. Requires PostScript printer. A screen preview is displayed on screen.

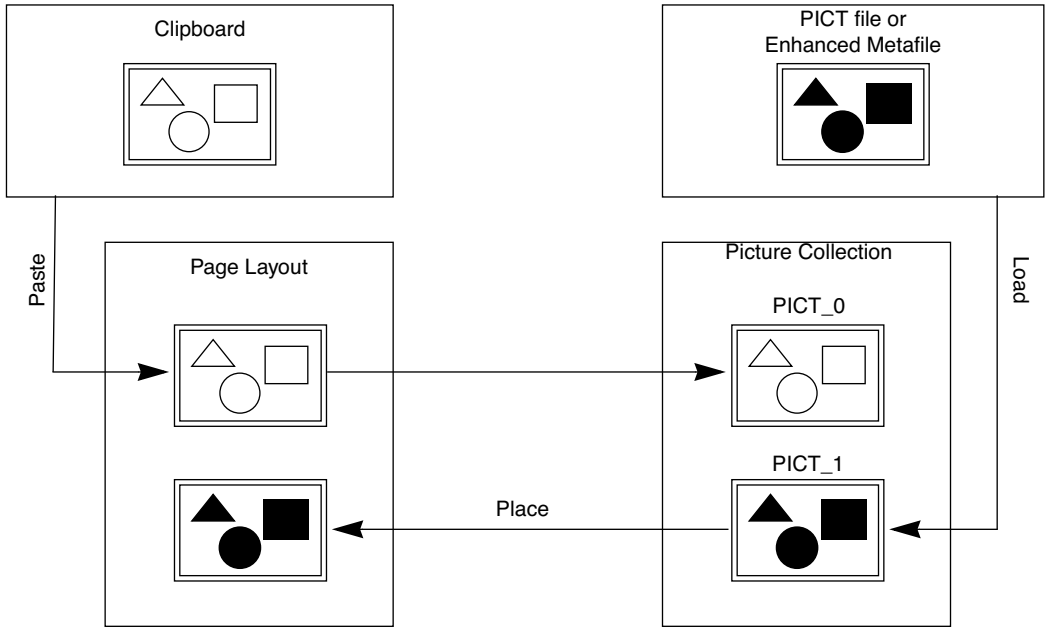
PDF is the standard format for Mac OS X graphics. PICT was used prior to OS X and can still be pasted or place in layouts but is obsolete.

EMF is the standard format for Windows graphics. Both of these are platform-dependent and will display as gray boxes if you move the Igor experiment to the other platform. The other formats are platform-independent. For more details, see **Picture Compatibility** on page III-395.

If you will be exporting your page layout to an EPS file or printing to a Postscript printer, you will get the best results if your imported pictures are EPS. There are some restrictions on exporting a layout that contains pictures as EPS. If you plan to do this, see Chapter III-5, **Exporting Graphics (Macintosh)**, or Chapter III-6, **Exporting Graphics (Windows)**, for details.

All pictures used in the layout layer or in the drawing layers are stored in the current experiment's picture collection. You can examine the picture collection, load pictures into it and remove pictures from it using the Pictures dialog. This is described in greater detail in the section **Pictures** on page III-421.

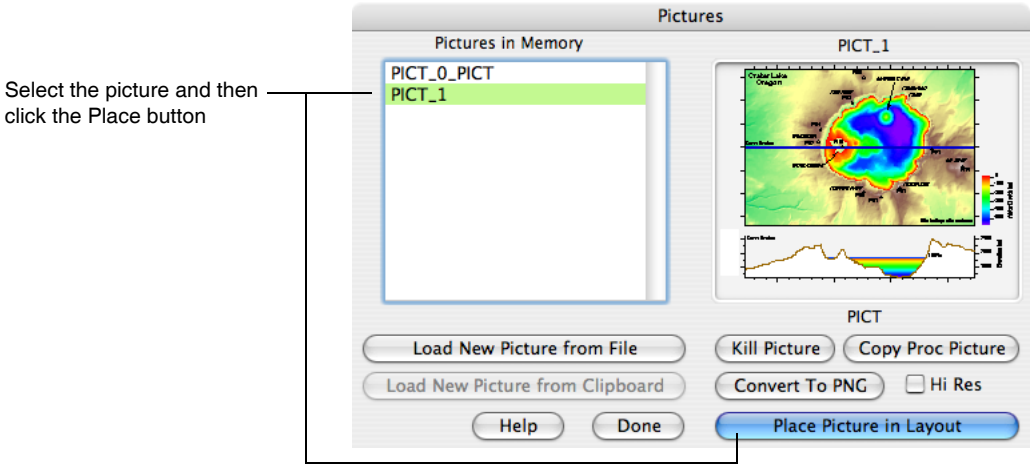
This diagram shows the two methods for putting a picture into a layout and how this relates to the picture collection.



Pasting a picture from the Clipboard or loading it from a file puts the picture into the experiment's picture collection and auto-names it.

Placing a Picture

Here is the Pictures dialog.



When you click Place Picture In Layout, Igor closes the Pictures dialog and displays an angle-bracket cursor. If you drag out a rectangle in the page with the angle-bracket, Igor will paste the picture into that rectangle. If you just click with the angle-bracket, Igor will paste the picture at its default size, putting the top-left corner of the picture where you clicked.

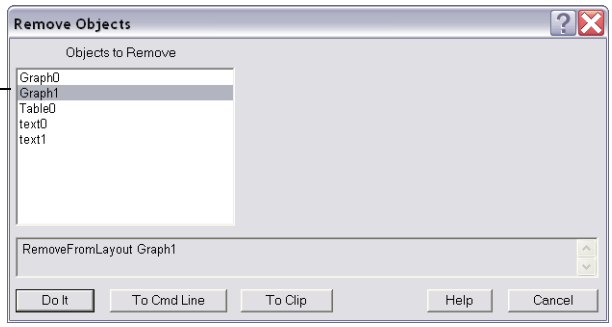
You can't scroll or zoom the page layout while the angle-bracket cursor is active. Therefore, you may need to adjust the picture after placing it.

You can always reset a picture to its default size by pressing Option (*Macintosh*) or Alt (*Windows*) and double-clicking it with the arrow tool.

Removing Objects from the Layout Layer

You can remove objects from a layout by choosing the Remove from Layout item from the Layout menu.

Select the objects to remove. Shift-click to select multiple objects.



You can also remove objects by selecting them with the arrow tool or enclosing them with the marquee tool and pressing Delete or by selecting the Clear or Cut items from the Edit menu.

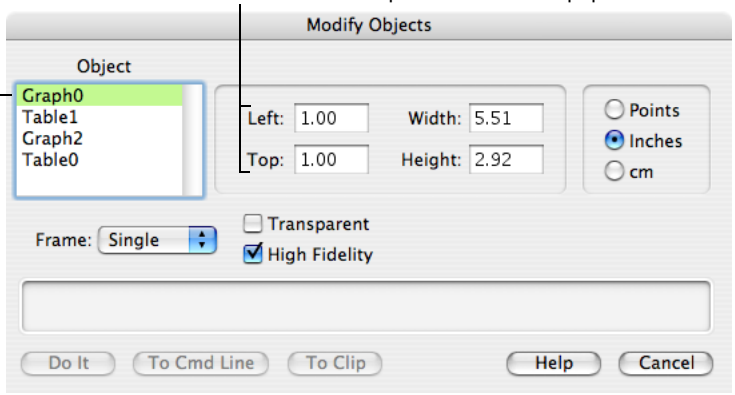
Removing a picture from a layout does not remove it from the picture collection. To do this, use the Pictures dialog.

Modifying Layout Objects

You can modify the properties of layout objects using the Modify Objects dialog. To invoke it, choose Modify Objects from the Layout menu or double-click an object with the arrow tool.

Measured from the top-left corner of the paper.

Select the object to remove. You can select only one object at a time.



The effect of each property is described under **Layout Object Properties** on page II-371.

Once you have modified an object you can select another object from the Object list and modify it.

High Fidelity

The High Fidelity property determines how a layout object is redrawn when it is resized. If selected, the object is fully redrawn. If deselected (low fidelity mode), a stored picture of the object is stretched to fit the new size. As of Igor Pro 6.1, this property does not affect graph objects which are always drawn in high fidelity.

The low fidelity mode was created for speed considerations at a time when personal computers ran at 16 MHz. Now there is rarely any reason to use it.

Annotations in the Layout Layer

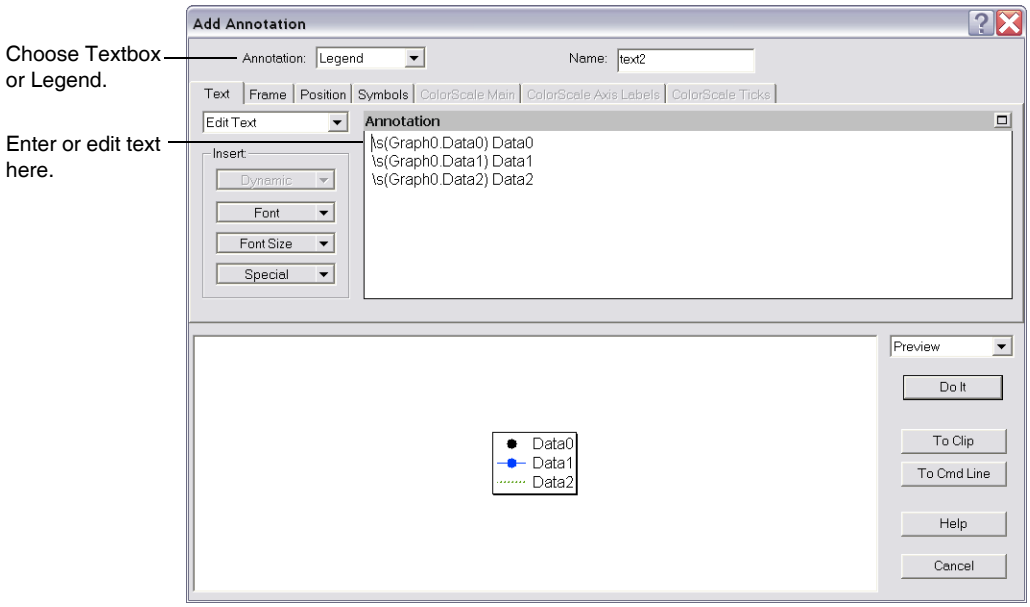
The term "annotation" includes textboxes, legends and tags. You can create annotations in graphs and layouts. Annotations are discussed in detail in Chapter III-2, **Annotations**. This section discusses aspects of annotations that are unique to page layouts.

In a graph, an annotation can be a textbox, legend or tag. A legend shows the plot symbols for the waves in the graph. A tag is connected to a particular point of a particular wave. In a layout, tags are not applicable. You can create textboxes and legends.

Don't confuse annotations with the simple text elements that you can create in the drawing layers of graphs, layouts and control panels. These simple text elements are intended for specialized purposes, such as creating axes that Igor doesn't directly support (e.g. polar axes). Annotations are intended for general purpose labeling.

Creating a New Annotation

To create a new annotation, choose Add Annotation from the Layout menu or select the annotation tool and click anywhere on the page, except on an existing annotation. These actions invoke the Add Annotations dialog.



The many options in this dialog are explained in Chapter III-2, **Annotations**.

Modifying an Existing Annotation

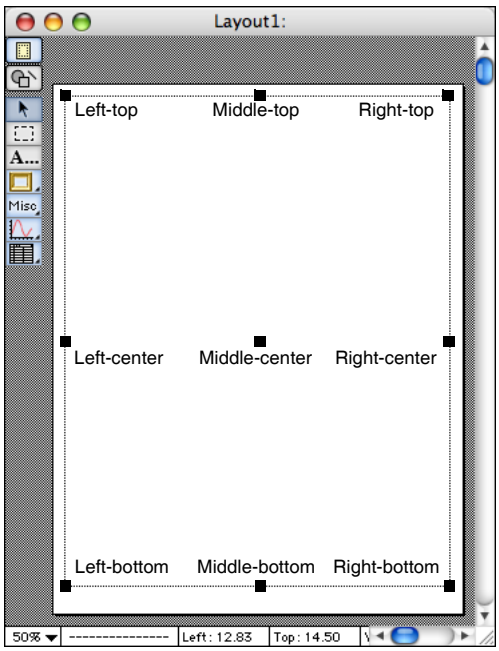
If an annotation is selected when you pull down the Layout menu, you will see a Modify Annotation item instead of the Add Annotation item. Use this to modify the text or style of the selected annotation. You can also get to the Modify Annotation dialog by clicking the annotation while the annotation tool is selected. Double-clicking an annotation while the arrow tool is selected brings up the Modify Object dialog, not the Modify Annotation dialog.

Positioning an Annotation

An annotation is positioned relative to an anchor point on the edge of the printable part of the page. The distance from the anchor point to the textbox is determined by the X and Y offsets which are in percent of the printable page. The X and Y offsets are automatically set for you when you drag a textbox around the page. You can also set them using the annotation Tweaks subdialog but this is usually not as easy as just dragging.

Positioning Annotations Programmatically

This diagram shows the anchor points. You don't need to know this to position annotations by dragging. You do need to know it to position them programmatically, from an Igor procedure.



Using the top-left anchor, a (0, 0) XY offset would put a tag in the top-left corner of the page:

```
Textbox/A=LT/X=0/Y=0 "Test 1"
```

An XY offset of (50, 50) would put a tag in the middle of the page.

```
Textbox/A=LT/X=50/Y=50 "Test 2"
```

Using the middle-center anchor, a (0, 0) XY offset would put a tag in the middle of the page:

```
Textbox/A=MC/X=0/Y=0 "Test 3"
```

An XY offset of (-50, 50) would put a tag in the top-left corner of the page.

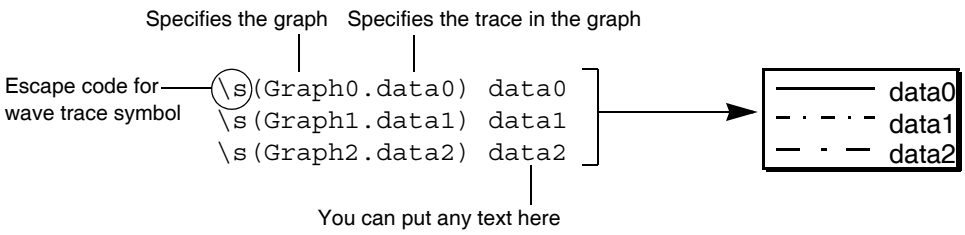
```
Textbox/A=MC/X=-50/Y=50 "Test 4"
```

For most purposes, the left-top anchor is the easiest to use and is sufficient.

The anchor sets not only the reference point on the page but also the reference point on the annotation. For example, if the anchor is right-top then the XY offset sets the position of the right-top corner of the annotation, relative to the right-top corner of the page. For this reason, if you want several textboxes to be right-aligned, you would want to use a right-top, right-center or right-bottom anchor.

Legends in the Layout Layer

When you invoke the Add Annotations dialog and choose Legend, Igor automatically sets the annotation's text to produce a legend containing a symbol for each wave in each graph object in the layout. In the picture of the dialog above, you can see the text that Igor generated. This diagram explains it.



Igor generates the lines of the legend text starting with the bottom graph object in the layout and working toward the top. You can edit the text to remove symbols that you don't want or to change what appears after the symbol.

If you change the symbol for a trace referenced in the legend, Igor will automatically update the layout legend. If you append or remove waves to the graphs represented in the layout, Igor will also update the layout legend. Updating happens when you activate the layout unless you have turned the layout's DelayUpdate setting off, in which case it happens immediately.

You can freeze a legend by converting it to a textbox. This stops Igor from automatically updating it when waves are added to or removed from graphs. To do this, select the annotation tool and click in the legend. In the resulting Modify Annotation dialog, change the pop-up menu in the top-left corner from Legend to Textbox. You can also do this using the following command:

```
Textbox/C/N=text0    // convert legend named text0 into a textbox
```

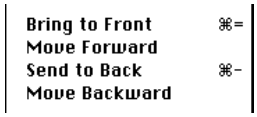
Instead of specifying the name of the trace for a legend symbol, you can specify the trace number. For example, "\s(Graph0.#0)" displays the legend for trace number 0 of Graph0.

Default Font

By default, annotations use the default font chosen in the Default Font dialog via the Misc menu. You can override the default font using the Font pop-up menu in the Add Annotation dialog. If you change the default font, Igor will automatically update the layout. This will happen when you activate the layout or immediately if you have disabled the layout's DelayUpdate setting.

Front-To-Back Relationship of Objects

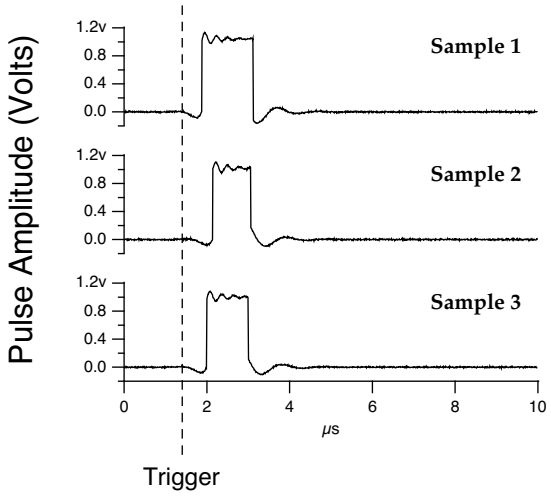
New objects added to the layout layer are added in front of existing objects. You can move objects in front of or in back of other objects using the Layout menu after selecting a single object with the arrow tool.



These menu commands affect the layout layer only. To put drawing elements in front of the layout layer, use the User Front drawing layer. To put drawing elements behind the layout layer, User Back drawing layer.

Aligning Stacked Graph Objects

It is a common practice to stack a group of graphs vertically in a column. Sometimes, only one X axis is used for a number of vertically stacked graph. Here is an example.



This section gives step-by-step instructions for creating a layout like the one above. It is also possible to do this using a single graph (see **Creating Stacked Plots** on page II-293 for details) or using subwindows (see Chapter III-4, **Embedding and Subwindows**).

To align the axes of multiple graph objects in a layout, it is critical to set the graph margins. This is explained in detail as follows.

The basic steps are:

1. Prepare the graphs.
2. Append the graph objects to the layout.
3. Align the left edges of the graph objects.
4. Set the width and height of the graph objects.
5. Set the vertical positions of the graph objects.
6. Set the graph plot areas and margins to uniform values.

It is possible to do steps 3, 4, and 5 at once by using the Arrange Objects dialog. However, in this section, we will do them one-at-a-time.

Prepare the Graphs

It is helpful to set the size of the graph windows approximately to the size you intend to use in the layout so that what you see in the graph window will resemble what you get in the layout. You can do this manually or you can use the MoveWindow operation. For example, here is a command that sets the target window to 5 inches wide by 2 inches tall, one inch from the top-left corner of the screen.

```
MoveWindow/I 1, 1, 1 + 5, 1 + 2
```

In the example shown above, we wanted to hide the X axes of all but the bottom graph. We used the Axis tab of the Modify Graph dialog to set the axis thickness to zero and the Label Options tab to turn the axis labels off.

Append the Graphs to the Layout

Click in the layout window or create a new layout using the New Layout item in the Windows menu. If necessary, activate the layout tools by clicking the layout icon in the top-left corner of the layout. Use the Graph pop-up menu or the Append to Layout item in the Layout menu to add the graphs. Drag each graph to the general area where you want it.

Align Left Edges of Layout Objects

Drag one of the graphs to set its left position to the desired location. Then Shift-click the other graphs to select them. Now choose Align→Left Edges from the Layout menu.

Set Width and Height of Layout Objects

Set the width and height of one of the graph objects by selecting it and dragging the resulting handles or by double-clicking it and entering values in the Modify Objects dialog.

Click in a blank part of the page to deselect all objects. Now click the object whose dimensions you just set. Now Shift-click to select the other graph objects. With all of the graph objects selected, choose Make Same Width And Height from the Layout menu.

Set Vertical Positions of Layout Objects

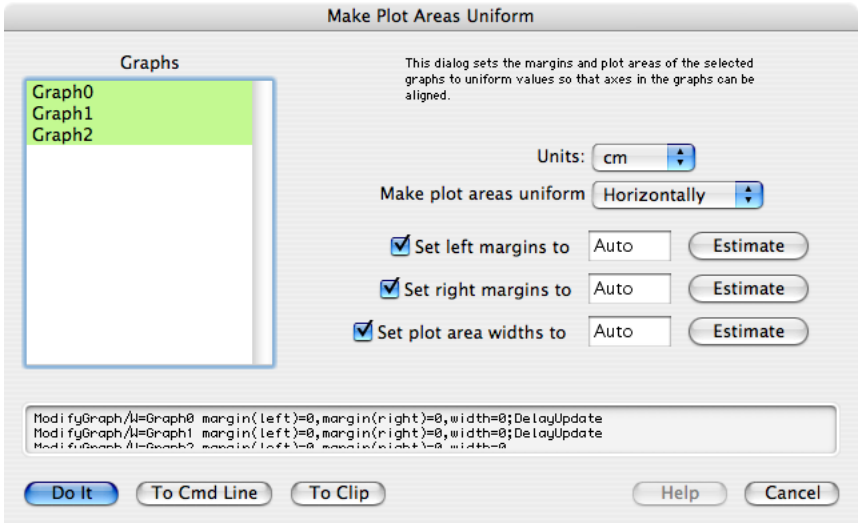
Drag the graph objects to their approximate desired positions on the page. You can drag an object vertically without affecting its horizontal position by pressing Shift while dragging. Once you have set the approximate position, fine tune the vertical positions using the arrow keys to nudge the selected object.

Set Graph Plot Areas and Margins

At this point, your axes would be aligned except for one subtle thing. The width of text (e.g., tick mark labels) in the left margin of each graph can be different for each graph. For example, if one graph has left

axis tick mark labels in the range of 0.0 to 1.0 and another graph has labels in the range 10,000 to 20,000, Igor would leave more room in the left margin of the second graph. The solution to this problem is to set the graph margins, as well as the width of the plot areas, of each graph to the same specific value.

To do this, select all of the graph objects and then choose Make Plot Areas Uniform from the Layout menu. This invokes the following dialog:



Note that, because we are stacking graphs vertically, we want their horizontal margins and plot areas to be the same, which is why we have selected Horizontally from the pop-up menu. The three checkboxes are selected because we want to set both the left and right margins as well as the plot area width.

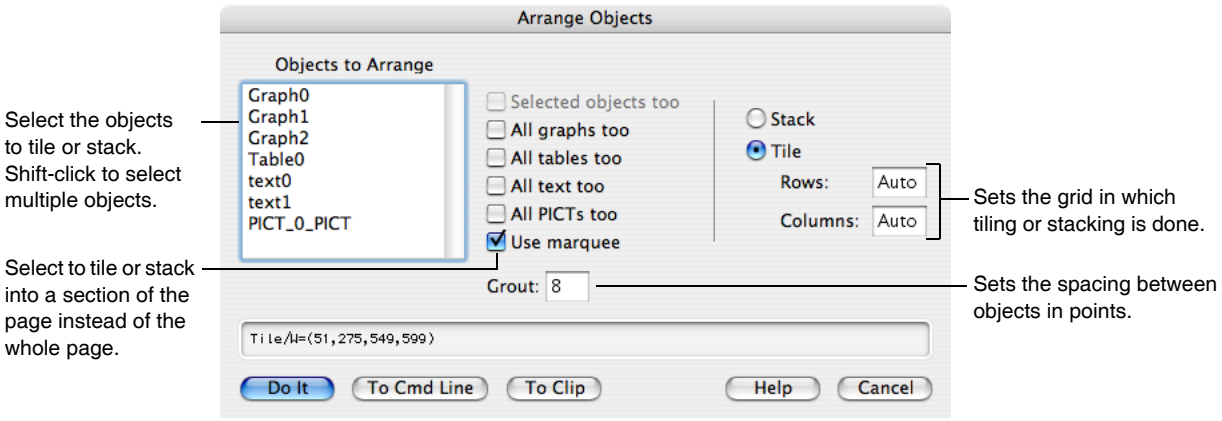
Now click each of the three Estimate buttons. When you click the Estimate button next to the Set Left Margins To checkbox, Igor sets the corresponding edit box to the largest left margin of all of the graphs selected in the list. Igor does a similar thing for the other two Estimate buttons. As a result, after clicking the three buttons, you should have reasonable values. Click Do It.

Now examine the stacked graph objects. It is possible that you may want to go back into the Make Plot Areas Uniform dialog to manually tweak one or more of the settings.

After doing these steps, the horizontal plot areas in the stacked graphs will be perfectly aligned. This does not, however, guarantee that the left axes will line up. The reason for this is the graphs' axis standoff settings. The axis standoff setting, if enabled, moves the left axis to the left of the plot area to prevent the displayed traces from colliding with the axis. If the graphs have different sized markers, for example, it will offset the left axis of each graph by a different amount. Thus, although the plot areas are perfectly-aligned horizontally, the left axes are not aligned. The solution for this is to use the Modify Axis dialog (Graph menu) to turn axis standoff off for each graph.

Arranging Objects

You can tile or stack objects in a layout by choosing the Arrange Objects item from the Layout menu.



To arrange objects in a section of the page rather than the whole page, you must use the marquee tool to specify the section *before* invoking the dialog. Then make sure the "Use marquee" checkbox is selected.

There are several ways to specify which objects to arrange. Any objects that you select in the Objects to Arrange list will be arranged. In addition, you can include the selected object, all graphs, tables, textboxes or pictures by enabling the appropriate checkbox. If you select no objects in the list and select none of the checkboxes, then all of the objects in the layout will be arranged.

You can set the number of rows and columns of tiles or you can leave them both on auto. If auto, Igor figures out a nice arrangement based on the number of objects to be tiled and the available space. Setting rows or columns to zero is the same as setting it to auto.

If you set both the rows and columns to a number between 1 and 100, Igor tiles the objects in a grid determined by your row/column specification. If you set either rows *or* columns to a number between 1 and 100 but leave the other setting on auto, Igor figures out what the other setting should be to properly tile the objects. In all cases, Igor tiles starting from the top-left cell in a grid defined by the rows and columns, moving horizontally first and then vertically.

If the grid that you specify has fewer tiles than the number of objects to be tiled, once all of the available tiles have been filled, Igor starts tiling from the top-left corner again.

Regardless of the parameters you specify, Igor clips coordinates so that a tiled object is never completely off the page. Also, objects are never set smaller than a minimum size or larger than the page.

The order in which objects are tiled is determined by the order in which they appear in the command generated by the Arrange Objects dialog. This in turn depends on the front to back ordering of the objects in the layout. Objects are tiled from left to right, top to bottom. Therefore, you can control exactly where each object winds up by controlling the front to back ordering. Another approach is to use the Arrange Objects dialog to compose the Tile command. Then, instead of clicking Do It, click To Cmd Line, putting the Tile command in the command line. Now arrange the objects by editing the command line so that they are in the order in which you want them tiled. Then press Return or Enter to execute the Tile command.

Printing Graphs as Bitmaps

You can print graphs in layouts using a high-res bitmap rather than the usual object draw method. Use this when a printer driver has bugs that affect normal operations. It may also be useful for printing graphs with very large numbers of data points. There are drawbacks to the bitmap method. A large amount of memory will be needed and on the Macintosh, patterns will be too small to be useful. Also, the quality of lines, dashed lines in particular, may be inferior.

To have Igor to print graphs in Layouts using the bitmap method, execute the following on the command line:

```
Variable/G V_PrintUsingBitmap = 1
```

This command creates a variable that is stored in the current experiment. You must execute this command for each experiment in which you want to use bitmap printing. Also, this variable must be created in the root Data Folder.

To return Igor to its default printing settings, set V_PrintUsingBitmap=0 or kill the variable.

Exporting Layouts

You can export a layout to another application through the Clipboard or by creating a file. To export via the Clipboard, use the Export Graphics item in the Edit menu. To export via a file, use the Save Graphics item in the File menu.

If you want to export a section of the page, use the marquee tool to specify the section first. To do this, the layout icon in the top-left corner of the layout window must be selected. If you don't use the marquee, Igor exports the part of the page that has layout objects or drawing elements in it.

The process of exporting graphics from a layout is very similar to exporting graphics from a graph. Because of this, we have put the details elsewhere: Chapter III-5, **Exporting Graphics (Macintosh)**, and Chapter III-6, **Exporting Graphics (Windows)**. These chapters describe the various export methods and how to select the method that will give you the best results.

Copying Objects from the Layout Layer

You can copy objects to the Clipboard by selecting them with the arrow tool or enclosing them with the marquee tool and then choosing Copy from the Edit menu. You can also choose Copy from the pop-up menu that appears when you click inside the marquee.

When you copy an object to the Clipboard, it is copied in two formats:

- As an Igor object in a format used internally by Igor
- As a picture that can be understood by other applications

Although you can do a copy for the purposes of exporting to another application, this is not the best way. See **Exporting Layouts** on page II-387 for a discussion of exporting graphics to another application. This section deals with copying objects for the purposes of pasting them in the same or another layout. Since it is easy to append graphs and tables to a layout using the pop-up menus in the tool palette, the main utility of this is for copying annotations or pictures from one layout to another.

Copying as an Igor Object Only

There are times when a straightforward copy operation is not desirable. Imagine that you have some graph objects in a layout and you want to put the same objects in another layout. You could copy the graph objects and paste them into the other layout. However, if the graphs are very complex, it could take a lot of time and memory to copy them to the Clipboard as a picture. If your purpose is not to export to another application, there is really no need to copy as a picture. If you press Option (*Macintosh*) or Alt (*Windows*) while choosing Copy, then Igor will do the copy only as Igor objects, not as a picture. You can now paste the copied graphs in the other layout.

Pasting Objects into the Layout Layer

This section discusses pasting Igor objects that you have copied from the same or a different page layout. For pasting a new picture that you have generated with another application, see **Inserting a Picture in the Layout Layer** on page II-377.

To paste layout objects that you have copied to the Clipboard from the same Igor experiment, just choose Paste from the Edit menu.

When you copy a graph, table or picture layout object from a layout to the Clipboard, it is copied as a picture and as an Igor object, in an internal Igor format. The Igor format includes the name by which Igor knows

the layout object. If you later paste into a layout, Igor will use this name to determine what object should be added to the layout. It normally does not paste the picture representation of the object. In other words, the Igor format of the object that is copied to the Clipboard *refers* to a graph, table or picture by its name.

In rare cases, you may actually want to paste as a picture, not as an Igor object. You might plan to change the graph but want a representation of it as it is now in the layout. To do this, press Option (*Macintosh*) or Alt (*Windows*) while choosing Edit→Paste. This creates a new named picture in the current experiment.

Pasting into a Different Experiment

The reference in the Clipboard to Igor objects by name doesn't work across Igor experiments. The second experiment may have a different object with the same name or it may have no object with the name stored in the Clipboard. The best you can do when pasting from one experiment to another is to paste a *picture* of the object from the first experiment.

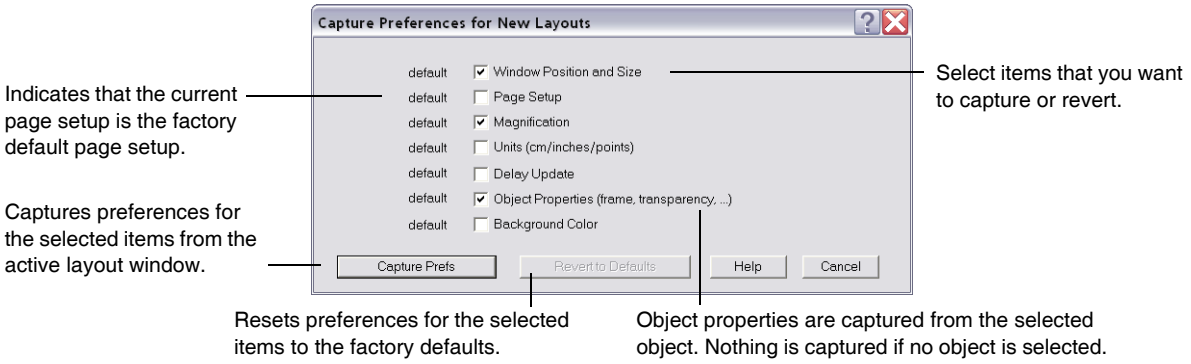
You can force Igor to paste the picture representation instead of the Igor object representation as described above, by pressing Option (*Macintosh*) or Alt (*Windows*) while choosing Edit→Paste.

Pasting Color Scale Annotations

For technical reasons, Igor is not able to faithfully paste a color scale annotation that uses a color index wave or that uses the lookup keyword of the ColorScale operation. If you paste such a color scale, Igor will change it to a color table color scale annotation with no lookup.

Page Layout Preferences

Page layout preferences allow you to control what happens when you create a new layout or add new objects to the layout layer of an existing layout. To set preferences, create a layout and set it up to your taste. We call this your *prototype* layout. Then choose Capture Layout Prefs from the Layout menu.



Preferences are normally in effect only for *manual* operations, not for programmed operations in Igor procedures. This is discussed in more detail in Chapter III-17, **Preferences**.

When you initially install Igor, all preferences are set to the factory defaults. The dialog indicates which preferences you have changed.

The "Window Position and Size" preference affects the creation of new layouts only.

The Object Properties preference affects the creation of new objects in the layout layer. To capture this, add an object to the layout layer and use the Modify Objects dialog to set its properties. Then select the object and choose Capture Layout Prefs. Select the Object Properties checkbox and click Capture Prefs.

The page setup preference affects what happens when you create a new layout, not when you recreate a layout using a recreation macro.

Layout Style Macros

The purpose of a layout style macro is to allow you to create a number of layouts with the same stylistic properties. Using the Window Control dialog, you can instruct Igor to automatically generate a style macro from a prototype layout. You can then apply the macro to other layouts.

Igor can generate style macros for graphs, tables and page layouts. However, their usefulness is mainly for graphs. See **Graph Style Macros** on page II-300. The principles explained there apply to layout style macros also.

Problems with Layouts

This section discusses problems that some people may encounter in using page layouts.

Picture Transparency

A picture is inherently opaque if the picture itself erases its own background. For a picture that you create in a drawing program, this would be the case if you drew a white rectangle behind all of the other elements in the picture. If a picture is inherently opaque, you can't make it transparent by changing the layout transparency property. It will always print opaque.

Graphs Transparency

A graph with a nonwhite background color is inherently opaque. You can't make it transparent by changing the layout transparency property. It will always print opaque.

Transparency on Screen and in the Printout

The part of Igor that draws layout objects in the layout window is not smart enough to recognize when an object is inherently opaque. Because of techniques used in drawing the screen, if you set an inherently opaque object to transparent, it will appear transparent in the layout window but will print opaque.

Page Layout Shortcuts

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
Change the layout magnification	Click the magnification readout in the lower-left corner of the layout window.	Click the magnification readout in the lower-left corner of the layout window.
Modify layout object properties	Select the arrow tool in Layout mode and double-click the object.	Select the arrow tool in Layout mode and double-click the object.
Edit an existing annotation	Select the annotation tool in the Layout mode and click in the annotation.	Select the annotation tool in the Layout mode and click in the annotation.
Bring up a graph or table window	Select corresponding object in the layout layer and then double-click the name of the object in the info panel.	Select corresponding object in the layout layer and then double-click the name of the object in the info panel.
Auto-size a picture or annotation object to 100%	Select arrow tool in Layout mode, press Option and double-click the picture or annotation object.	Select arrow tool in Layout mode, press Alt and double-click the picture or annotation object.
Auto-size a table object to an integral number of rows and columns	Select arrow tool in Layout mode, press Option and double-click the table object.	Select arrow tool in Layout mode, press Alt and double-click the table object.
Constrain the resizing direction or dragging an object	Press Shift while resizing or dragging the object.	Press Shift while resizing or dragging the object.
Copy, cut, or clear multiple layout objects	Use the arrow tool or marquee tool to select the objects, then choose copy, cut or clear from the Edit menu.	Use the arrow tool or marquee tool to select the objects, then choose copy, cut or clear from the Edit menu.
Export a subset of the layout via the Clipboard	Using the marquee tool, select a page area, then choose Export Graphics from the Edit menu.	Using the marquee tool, select a page area, then choose Export Graphics from the Edit menu.
Export a subset of the layout via the a graphics file	Using the marquee tool, select a page area and then choose Save Graphics from the File menu.	Using the marquee tool, select a page area and then choose Save Graphics from the File menu.
Drawing tool shortcuts	See Chapter III-3, **Drawing**.	See Chapter III-3, **Drawing**.

Volume III User's Guide: Part 2

Table of Contents

Chapter
III-1

Notebooks

Overview

A notebook is a window in which you can store text and graphics, very much like a word processor document. Typical uses for a notebook are:

- Keeping a log of your work.
- Generating a report.
- Examining or editing a text file created by Igor or another program.
- Documenting an Igor experiment.

A notebook can also be used as a worksheet in which you execute Igor commands and store text output from them.

Plain and Formatted Notebooks

There are two types of notebooks:

- Plain notebooks.
- Formatted notebooks.

Formatted notebooks can store text and graphics and are useful for fancy reports. Plain notebooks can store text only. They are good for examining data files and other text files where line-wrapping and fancy formatting is not appropriate.

This table lists the properties of each type of notebook.

Property	Plain	Formatted
Can contain graphics	No	Yes
Allows multiple paragraph formats (margins, tabs, alignment, line spacing)	No	Yes
Allows multiple text formats (fonts, text styles, text sizes, text colors)	No	Yes
Does line wrapping	No	Yes
Has rulers	No	Yes
Has headers and footers	Yes	Yes
File name extension	.txt	.ifn
Can be opened by most other programs	Yes	No
Can be exported to word processors via Rich Text file	Yes	Yes

Plain text files can be opened by many programs, including virtually all word processors, spreadsheets and databases. The Igor formatted notebook file format is a proprietary WaveMetrics format that other applications can not open. However, you can save a formatted notebook as a Rich Text file, which is a file format that many word processors can open.

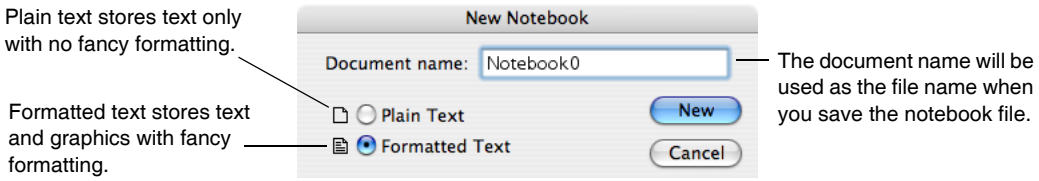
On Macintosh, Igor stores the settings (font, size, style, etc.) for a plain text file in the file's resource fork. The data fork contains just the plain text. *Under Windows*, files have no resource fork. Therefore there is no way for Igor to store settings for a plain text file on Windows. When you open a plain text notebook or an experiment containing a plain text notebook on Windows, Igor uses preferences to set the notebook's text format and document settings, including headers and footers. Thus, format changes that you make to a plain text notebook are lost on Windows unless you capture them as your preferred format.

UTF-16 Files

You can open UTF-16 (two-byte Unicode) text files as plain text notebooks. Igor does not recognize non-ASCII characters, but does ignore the byte-order mark at the start of the file (BOM) and null bytes contained in UTF-16 text files. If you open a UTF-16 file and then save it from Igor, it will be saved as plain ASCII, not UTF-16, and some information may be lost. This feature is intended mainly to enable you to inspect UTF-16 data files.

Creating a New Notebook File

To create a new notebook, choose Notebook from the New submenu of the Windows menu. This displays the New Notebook dialog.



This creates a new notebook *window*. The notebook *file* is not created until you save the notebook window or save the experiment.

Normally you should store a notebook as part of the Igor experiment in which you use it. This happens automatically when you save the current experiment unless you do an explicit Save Notebook As before saving the experiment. Save Notebook As stores a notebook separate from the experiment. This is appropriate if you plan to use the notebook in multiple experiments.

Note: There is a risk in sharing notebook files among experiments. If you copy the experiment to another computer and forget to also copy the shared files, the experiment will not work on the other computer. See **References to Files and Folders** on page II-37 for more explanation.

If you do create a shared notebook file then you are responsible for copying the shared file when you copy an experiment that relies on it.

Opening an Existing File as a Notebook

You can create a notebook window by opening an existing file. This might be a notebook that you created in another Igor experiment or a plain text file created in another program. To do this, choose Notebook from the Open File submenu of the File menu.

Opening a File for Momentary Use

You might want to open a text file momentarily to examine or edit it. For example, you might read a Read Me file or edit a data file before importing data. In this case, you would open the file as a notebook, do your reading or editing and then kill the notebook. Thus the file would not remain connected to the current experiment.

Sharing a Notebook File Among Experiments

On the other hand, you might want to share a notebook among multiple experiments. For example, you might have one notebook in which you keep a running log of all of your observations. In this case, you could save the experiment with the notebook open. Igor would then save a reference to the shared notebook file in the experiment file. When you later open the experiment, Igor would reopen the notebook file.

As noted above, there is a risk in sharing notebook files among experiments. You might want to "adopt" the opened notebook. See **References to Files and Folders** on page II-37 for more explanation.

Notebooks as Worksheets

Normally you enter commands in Igor's command line and press Return or Enter to execute them. You can also enter and execute commands in a notebook window. Some people may find using a notebook as a worksheet more convenient than using Igor's command line.

You can also execute commands from procedure windows and from help windows. The former is sometimes handy during debugging of Igor procedures. The latter provides a quick way for you to execute commands while doing a guided tour or to try example commands that are commonly presented in help files. The techniques described in the next paragraphs for executing commands from a notebook also apply to procedure and help windows.

To execute a command from a notebook, enter the command in a notebook and press Control-Enter. You can also select text already in the notebook and press Control-Enter. Under *Windows*, you can also right-click to get the pop-up menu from which you can choose Execute Selection. On *Macintosh*, you can Control-click to get the pop-up menu.

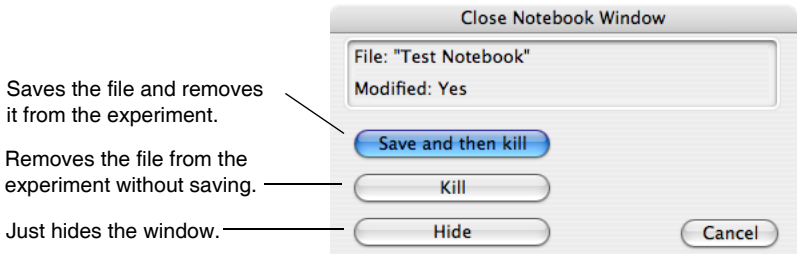
When you press Control-Enter, Igor transfers text from the notebook to the command line and starts execution. Igor stores both the command and any text output that it generates in the notebook and also in the history area of the command window. However, if you opened the notebook for read-only or if you clicked the write-protect icon, the command and output are sent to the history area only. If you don't want to keep the command output in the notebook, just undo it.

If you don't want to store the commands or the output in the history area, you can disable this using the Command Settings section of the Miscellaneous Settings dialog (Misc menu). However, if you opened the notebook for read-only or if you clicked the write-protect icon, the command and output are sent to the history area even if you have disabled this.

Showing, Hiding and Killing Notebook Windows

Notebook files can be opened (added to the current experiment), hidden, and killed (removed from the experiment).

When you click the close button of an notebook window, Igor presents the Close Notebook Window dialog to find out what you want to do.



If you just want to hide the window, you can press Shift while clicking the close button. This skips the dialog and just hides the window.

Killing a notebook window closes the window and removes it from the current experiment but does *not* delete the notebook file with which the window was associated. If you want to delete the file, do this on the desktop.

The Close item of the Windows menu and the keyboard shortcut, Command-W (*Macintosh*) or Ctrl+W (*Windows*), behave the same as the close button, as indicated in these tables.

Macintosh:

Action	Modifier Key	Result
Click close button, choose Close or press Command-W	None	Displays dialog
Click close button, choose Close or press Command-W	Shift	Hides window

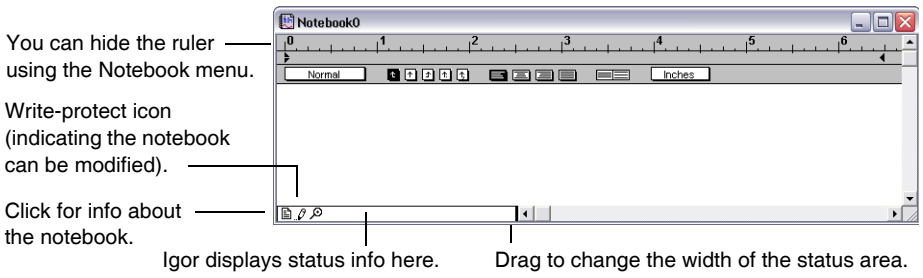
Windows:

Action	Modifier Key	Result
Click close button, choose Close or press Ctrl+W	None	Displays dialog
Click close button, choose Close or press Ctrl+W	Shift	Hides window

On the Macintosh when the Close Notebook Window dialog is showing, you can press Option to make the Kill button the default. The Kill button will become bold while the Save and then kill button will become normal. You can then press Return or Enter to kill the window. Similarly, press Shift to make the Hide button the default button.

Parts of a Notebook

This illustration shows the parts of a formatted notebook window. A plain notebook window has the same parts except for the ruler.



Write-Protect Icon

Notebooks (as well as procedure windows) have a write-enable/write-protect icon which appears in the lower-left corner of the window and resembles a pencil. If you click this icon, Igor Pro will draw a line through the pencil, indicating that the notebook is write-protected. The main purpose of this is to prevent accidental manual alteration of shared procedure files, but you can also use it to prevent accidental manual alteration of notebooks.

Note that write-protect is not the same as read-only. Write-protect prevents manual modifications while read-only prevents all modifications. See **Notebook Read/Write Properties** on page III-12 for details.

Magnifier Icon

You can magnify procedure text to make it more readable. See **Text Magnification** on page II-71 for details.

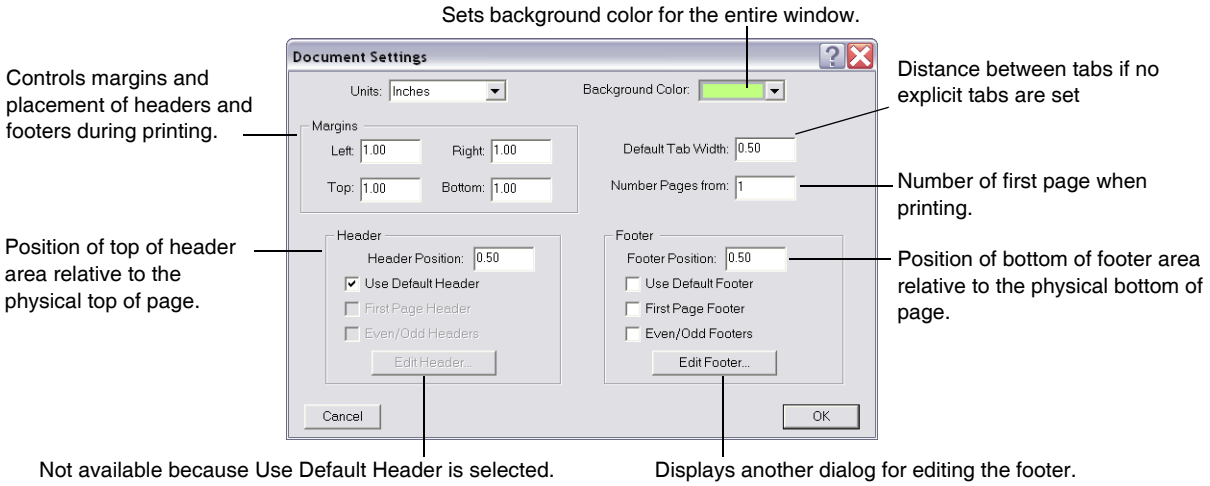
Notebook Properties

Everything in a notebook that you can control falls into one of three categories, as shown in this table.

Category	Settings
Document properties	Page margins, background color, default tab stops, headers and footers.
Paragraph properties	Paragraph margins, tab stops, line alignment, line spacing, default text format.
Character properties	Font, text size, text style, text color, vertical offset.
Read/write properties	Read-only, write-protect and changeableByCommandOnly.

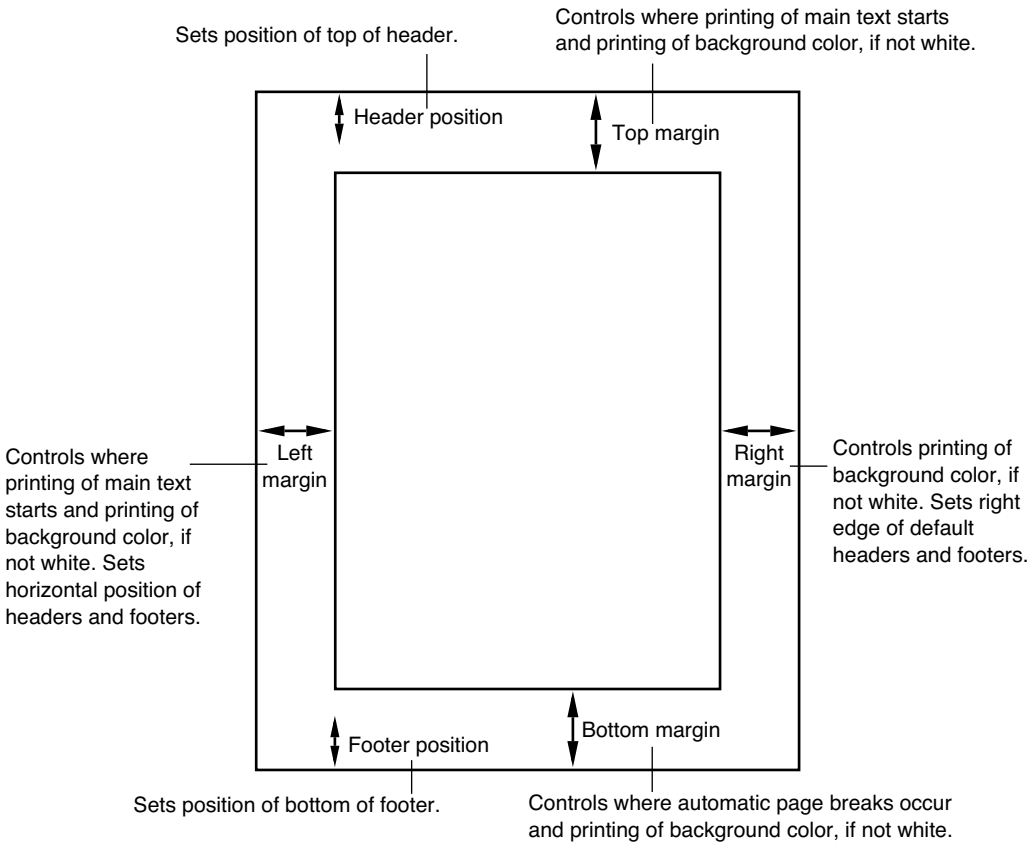
Document Properties

To set document properties, choose Document Settings from the Notebook menu.



In a formatted notebook, the ruler displays the default tab stops and you can adjust them by dragging. Although the default tab stops are indicated in the ruler, there is only one default tab width setting for the entire document, not one for each ruler.

The next illustration shows the effects of the page margins and header and footer position settings. In addition, these settings affect the Rich Text format file which you can use to export a notebook to a word processor.



Paragraph Properties

A set of paragraph properties is called a "ruler". In some word processors, this is called a "style". The purpose of rulers is to make it easy to keep the formatting of a notebook consistent. This is described in more detail under **Working with Rulers** on page III-13.

The paragraph properties are listed in this table.

Paragraph Property	Description
First-line indent	Horizontal position of the first line of the paragraph.
Left margin	Horizontal position of the paragraph after the first line.
Right margin	Horizontal position of the right side of the paragraph.
Line alignment	Left, center, right or full.
Space before	Extra vertical space to put before the paragraph.
Min line space	Minimum height of each line in the paragraph.
Space after	Extra vertical space to put after the paragraph.
Tab stops	Left, center, right, decimal-aligned or comma-aligned tab stops.
Ruler font	The default font to use for the paragraph.
Ruler text size	Default text size to use for the paragraph.
Ruler text style	Default text style to use for the paragraph.
Ruler text color	Default text color to use for the paragraph.

Plain Notebook Paragraph Properties

For each plain notebook, there is one set of paragraph properties that govern all paragraphs. Many of the items are fixed — you can't adjust them.

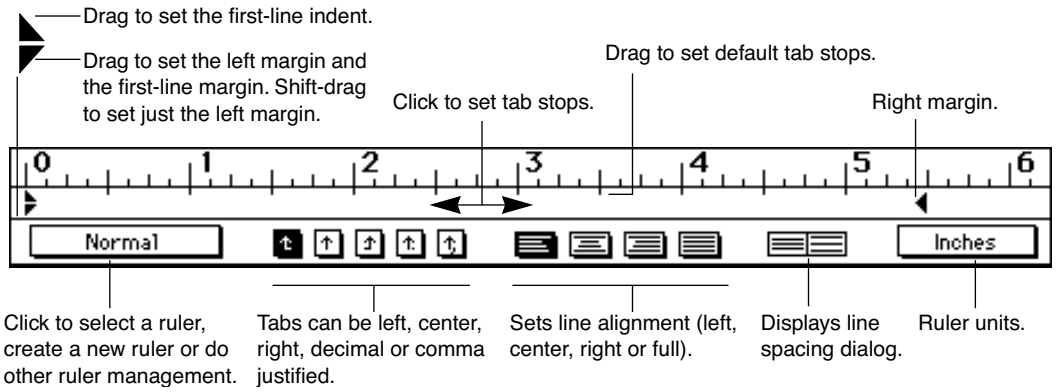
Paragraph Property	Comment
First-line indent	Fixed at zero.
Left margin	Fixed at zero.
Right margin	Fixed at infinity.
Line alignment	Fixed as left-aligned.
Space before	Fixed at zero.
Min line space	Fixed at zero.
Space after	Fixed at zero.
Tab stops	None.
Font	Set using Notebook menu.
Text size	Set using Notebook menu.
Text style	Set using Notebook menu.
Text color	Set using Notebook menu.

There is only one font, text size, text style and text color for the entire document which you can set using the Notebook menu.

Although you can not set paragraph tab stops in a plain notebook, you *can* set and use the notebook's default tab stops, which affect the entire notebook.

Formatted Notebook Paragraph Properties

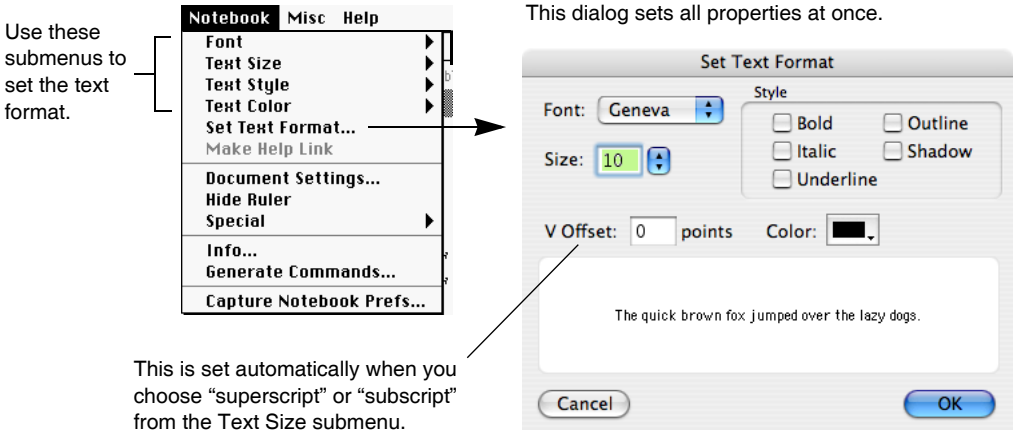
The paragraph properties for a formatted notebook are all under your control and can be different for each paragraph. A new formatted notebook has one ruler, called the Normal ruler. You can control the properties of the normal ruler and you can define additional rulers.



The ruler font, ruler text size, ruler text style and ruler text color can be set using the pop-up menu on the left side of the ruler. They set the *default* text format for paragraphs governed by the ruler. You can use the Notebook menu to override these default properties. The Notebook menu permits you to hide or show the ruler in a formatted notebook.

Character Properties

The character properties are font, text size, text style, text color and vertical offset. The vertical offset is used mainly to implement superscript and subscript. A specific collection of character properties is called a "text format". You can set the text format using the Notebook menu.



Plain Notebook Text Formats

A plain notebook has one text format which applies to all of the text in the notebook. You can set it, using the Notebook menu, except for the vertical offset which is always zero. On Windows there is no way for Igor to store settings for a plain text file. When you open a plain text notebook or an experiment containing a plain text notebook on Windows, Igor uses preferences to set the notebook's text format. Thus, text format changes that you make to a plain text notebook are lost on Windows unless you capture them as your preferred format.

Formatted Notebook Text Formats

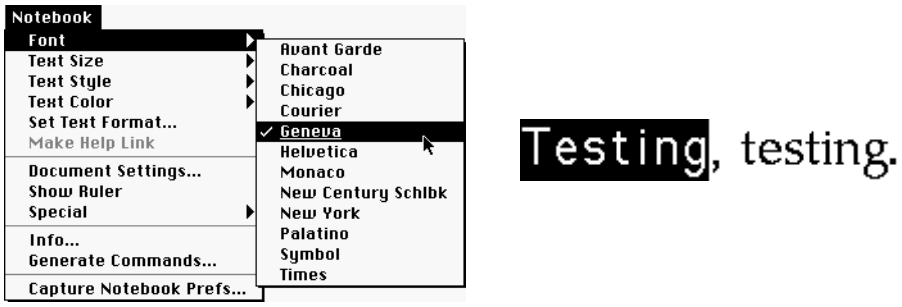
A formatted notebook can have any number of text formats. You can set the text format for the selected text using the Notebook menu. This *overrides* the default text format of the ruler.

You should use the ruler to set the basic text format and use overrides for highlighting or other effects. For example, you might override the ruler text format to underline a short stretch of text or to switch to Symbol font for a Greek character.

On *Macintosh*, the Font, Text Size, and Text Styles submenus in the Notebook menu indicate the currently selected font, size, and style using checkmarks and indicate the *ruler* font, size, and style using an underline. On Macintosh or Windows you can see the ruler font, size, style and color settings using the Ruler pop-up menu on the left side of the ruler.

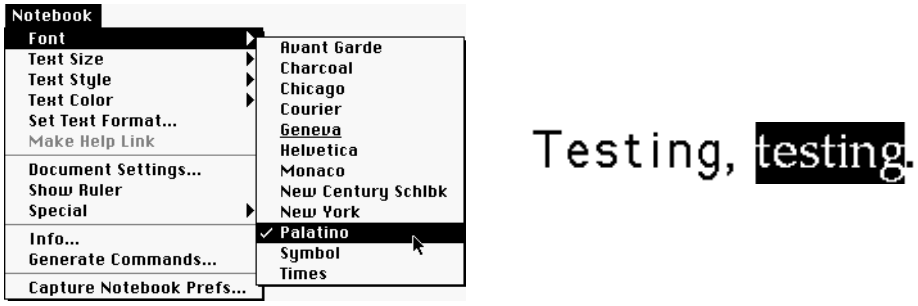
To illustrate the distinction between the setting for the selection and the ruler setting, consider the font submenu. The font for the selected text is checked. The ruler font is underlined (*Macintosh only*).

In this example, the current font, Geneva, is the same as the ruler font, so it is both checked and underlined.



If we redefined the ruler font, the selected text would automatically change to the new font.

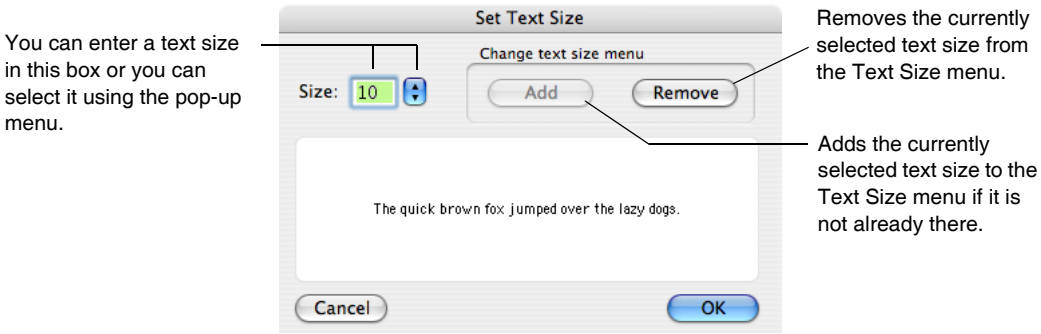
In the next example, the current font is not the same as the ruler font. We have overridden the ruler font with Palatino. Therefore, Palatino is checked but the ruler font, Geneva, is still underlined.



If we redefined the ruler font, the selected text would still override the ruler font and would remain Palatino.

Text Sizes

The Text Size submenu in the Notebook menu contains an Other item. This leads to the Set Text Size dialog.



The text sizes in your Text Size menu are stored in the Igor Preferences file so that the menu will include your preferred sizes each time you run Igor.

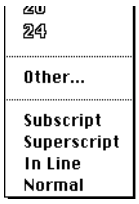
Vertical Offset

The vertical offset property is available only in formatted notebooks and is used mainly to implement superscript and subscript, as described in the next section.

Vertical offset is also useful for aligning a picture with text within a paragraph. For example, you might want to align the bottom of the picture with the baseline of the text. The easiest way to do this is to use Control-Up Arrow and Control-Down Arrow (*Macintosh*) or Ctrl+Alt+Up-Arrow and Ctrl+Alt+Down-Arrow (*Windows*) combinations to tweak the vertical offset by one point at a time.

Superscript and Subscript

The last four items in the Text Size submenu of the Notebook menu have to do with superscript and subscript. Igor implements superscript and subscript by setting the text size and the vertical offset of the selected text to achieve the desired effect. They are not character properties but rather are effects accomplished using character properties.



The following table illustrates the use and effects of each of these items. Do the actions to get a feel for how they work.

Action	Effect on Character Properties	Result
Type "XYZ".		XYZ
Highlight "Y" and then choose Superscript.	Reduces text size and sets vertical offset for "Y".	$X^{Y}Z$
Highlight "Z" and then choose Superscript.	Sets text size and vertical offset for "Z" to make it superscript relative to "Y".	$X^{Y^{Z}}$
Highlight "Z" and then choose In Line.	Sets text size and vertical offset for "Z" to be same as for "Y".	X^{YZ}
Highlight "YZ" and then choose Normal.	Sets text size for "YZ" same as "X" and sets vertical offset to zero.	XYZ

Notebook Read/Write Properties

There are three properties that control whether a notebook can be modified.

Read-only

The read-only property is set if you open the file for read-only using the Open Notebook dialog (File→Open→Notebook) or if you execute **OpenNotebook**/R. It is also set if you open a file for which you do not have read/write permission.

When the read-only property is set, a lock icon appears in the bottom/left corner of the notebook window and you can not modify the notebook manually or via commands.

The read-only property can not be changed after the notebook is opened.

Use read-only if you want no modifications to be made to the notebook.

Write-protect

You can set the write-protect property to on or off by clicking the pencil icon in the bottom/left corner of the notebook window or using the **Notebook** operation with the writeProtect keyword.

The write-protect property is intended to give the user a way to prevent inadvertent manual modifications to the notebook. The user can turn the property on or off at will.

The write-protect property does not affect commands such as **Notebook** and **NotebookAction**. Even if write-protect is on, they can still modify the notebook.

Use write-protect if you want to avoid inadvertent manual modifications to the notebook but want the user to be able to take full control.

Changeable By Command Only

You can control the changeableByCommandOnly property using **NewNotebook**/OPTS=8 or using the **Notebook** operation with the changeableByCommandOnly keyword.

This property is intended to allow programmers to control whether the user can manually modify the notebook or not. Its main purpose is to allow a programmer to create a notebook subwindow in a control panel for displaying status messages and other information that is not intended to be modified by the user. There is no way to manually change this property - it can be changed by command only.

When the changeableByCommandOnly property is on, a lock icon appears in the bottom/left corner of the notebook window.

Use changeableByCommandOnly if you want no manual modifications to be made to the notebook but want it to be modifiable via commands.

The changeableByCommandOnly property is intended for programmatic use only and is not saved to disk.

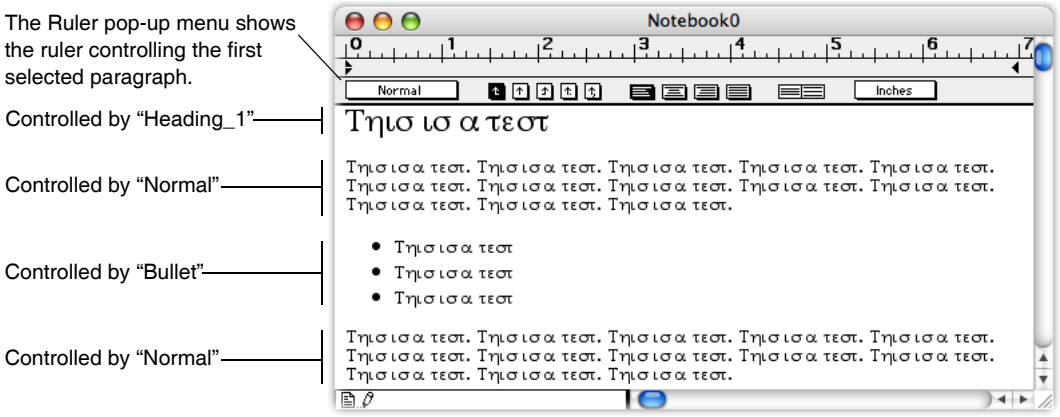
For further information on notebook subwindows, see **Notebooks as Subwindows in Control Panels** on page III-94.

Working with Rulers

A ruler is a set of paragraph properties that you can apply to paragraphs in a formatted notebook. Using rulers, you can make sure that paragraphs that you *want* to have the same formatting *do* have the same formatting. Also, you can redefine the format of a ruler and all paragraphs governed by that ruler will be automatically updated.

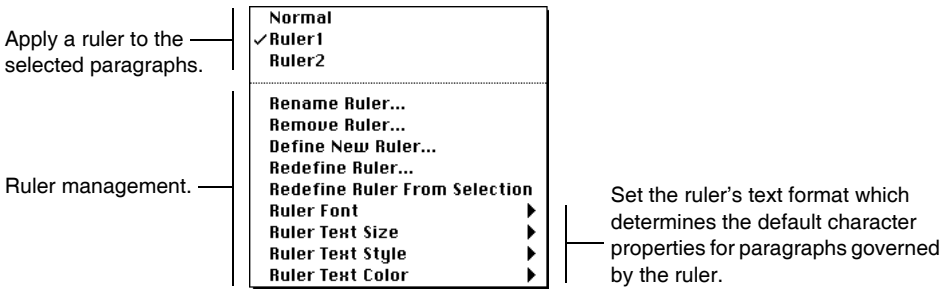
In a simple notebook, you might use just the one built-in ruler, called Normal. In a fancier notebook, where you are concerned with presentation, you might use several rulers.

Here is a sample notebook that uses three rulers: Normal, Heading_1 and Bullet.



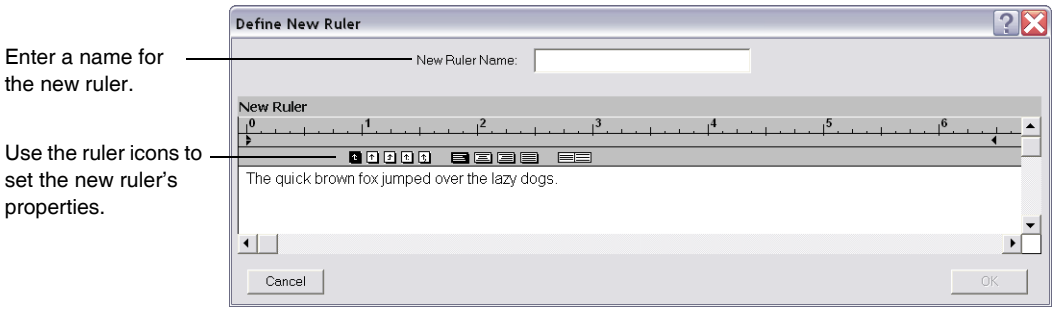
The pop-up menu on the left side of the ruler shows which ruler governs the first currently selected paragraph. You can use this pop-up menu to:

- Apply an existing ruler to the selected paragraph(s).
- Create a new ruler.
- Redefine an existing ruler.
- Find where a ruler is used.
- Rename a ruler.
- Remove a ruler from the document.



Defining a New Ruler

To create a new ruler, choose Define New Ruler from the Ruler pop-up menu.



On *Macintosh*, while in the dialog you can use the Notebook menu to set the ruler's font, text size, text style, and text color. On *Windows*, the Notebook menu is not available from the dialog, so you must use the Ruler pop-up menu to set these properties.

Ruler names must follow rules for standard (not liberal) Igor names. They may be up to 31 characters in length, must start with a letter and may contain letters, numbers and the underscore character.

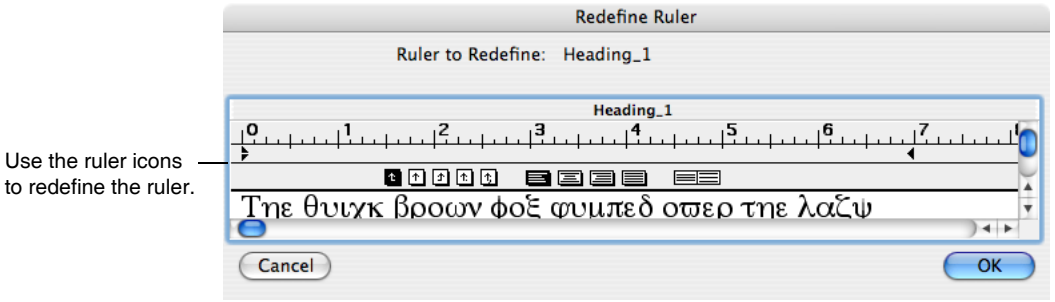
In a sophisticated word processor, a ruler can be based on another ruler so that changing the first ruler automatically changes the second. Igor rulers do not have this capability.

Redefining a Ruler

When you redefine a ruler, all paragraphs governed by the ruler are automatically updated. There are three ways to redefine a ruler:

- Use the Redefine Ruler dialog.
- Use the Ruler Font, Ruler Text Size, Ruler Text Style or Ruler Text Color pop-up menu items.
- Use the Redefine Ruler from Selection item in the Ruler pop-up menu.

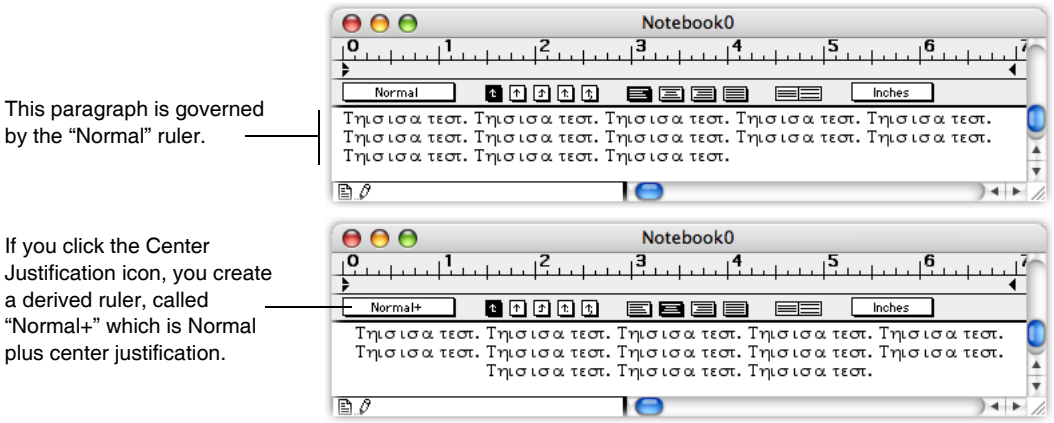
To invoke the Redefine Ruler dialog, choose Redefine Ruler from the Ruler pop-up menu.



Another handy way to redefine an existing, explicitly created ruler (e.g. Normal) is to adjust it, creating a derived ruler (e.g. Normal+). Then choose Redefine Ruler from Selection from the Ruler pop-up menu. This redefines the explicitly named ruler (Normal) to match the current ruler (Normal+).

Creating a Derived Ruler

You can adjust a ruler using its icons. When you do this, you create a *derived* ruler. A derived ruler is usually a minor variation of an explicitly created ruler. Here is an example.



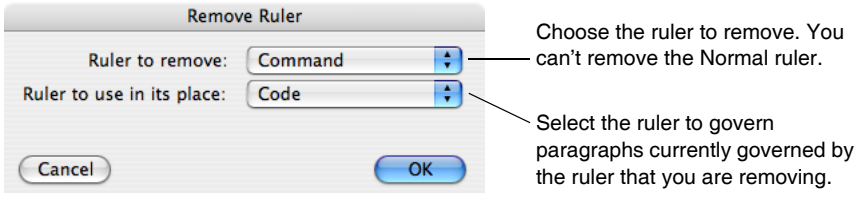
If you redefine the Normal ruler, the Normal+ ruler is *not* automatically redefined. This is a limitation in Igor’s implementation of rulers compared to a word-processor program.

Finding Where a Ruler Is Used

You can find the next or previous paragraph governed by a particular ruler. To do this press Option (*Macintosh*) or Alt (*Windows*) while selecting the name of the ruler from the Ruler pop-up menu. To search backwards, press Shift-Option (*Macintosh*) or Shift+Alt (*Windows*) while selecting the ruler. If there is no next or previous use of the ruler, Igor will emit a beep.

Removing a Ruler

Rulers that you no longer need clutter up the Ruler pop-up menu. You can remove them from the document by choosing Remove Ruler from the Ruler pop-up menu.



You might want to know if a particular ruler is used in the document. The only way to do this is to search for the ruler. See **Finding Where a Ruler Is Used** on page III-15.

Transferring Rulers Between Notebooks

The only way to transfer a ruler from one notebook to another is by copying text from the first notebook and pasting it in the second. Rulers needed for the text are also copied and pasted. If a ruler that exists in the source notebook also exists in the destination, the destination ruler takes precedence.

If you expect to create a lot of notebooks that share the same rulers then you should create a template document with the common rulers. See **Notebook Template Files** on page III-37 for details.

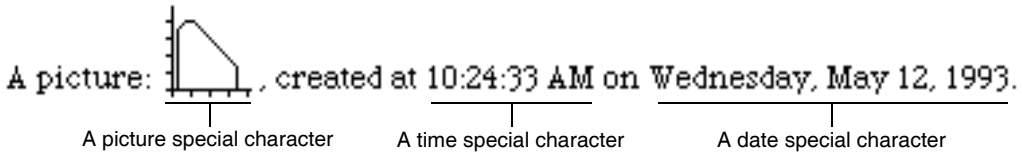
Special Characters

Aside from regular text characters, there are some special things that you can put into a paragraph in a formatted notebook. This table lists of all of the types of special characters and where they can be used.

Special Character Type	Where It Can Be Used
Picture	Main body text, headers and footers.
Igor-object picture (from graph, table, layout)	Main body text, headers and footers.
The date	Main body text, headers and footers.
The time	Main body text, headers and footers.
Notebook window title	Headers and footers only.
Current page number	Headers and footers only.
Total number of pages	Headers and footers only.
Actions	Main body text only.

The main way in which a special character differs from a normal character is that it is not simply a character from a font. Another significant difference is that some special characters are dynamic, meaning that Igor can update them automatically. Other special characters, while not dynamic, are linked to Igor graphs, tables or page layouts (see **Using Igor-Object Pictures** on page III-21).

This example shows three kinds of special characters.



The time and date look like normal text but they are not. If you click any part of them, the entire time or date is selected. They act like a single character.

An action is a special character which, when clicked, runs Igor commands. See **Notebook Action Special Characters** on page III-18 for details.

Except for pictures, special characters are inserted using the Special submenu in the Notebook menu.

Inserting Pictures

You can insert pictures, including Igor-object pictures, by merely doing a paste. You can also insert pictures using Edit→Insert File or using the Notebook insertPicture operation. The supported graphics formats are:

Format	Platform
Windows Bitmap (.bmp)	Windows only
Enhanced Metafile (.emf)	Windows only
Windows Metafile (.wmf)	Windows only
Encapsulated Postscript (.eps)	Macintosh and Windows
JPEG (.jpg)	Macintosh and Windows
PDF (.pdf)	Macintosh only
PICT (.pct)	Macintosh only

PNG (.png)	Macintosh and Windows
TIFF (*.tif)	Macintosh and Windows

On Windows, Encapsulated Postscript appears as a gray box on screen unless it includes a Windows screen preview and prints only on a PostScript printer.

When you insert a picture, the contents of the picture file are copied into the notebook. No link to the picture file is created.

Saving Pictures

You can save a picture in a formatted text notebook as a standalone picture file. Select one picture and one picture only. Then choose File→Save Graphics. You can also save a picture using the Notebook savePicture operation.

Special Character Names

Each special character has a name. For most types, the name is automatically assigned by Igor when the special character is created. However for action special characters you specify the name through the Special→New Action dialog. When you click a special character, you will see the name in the notebook status area. Special character names must be unique within a particular notebook.

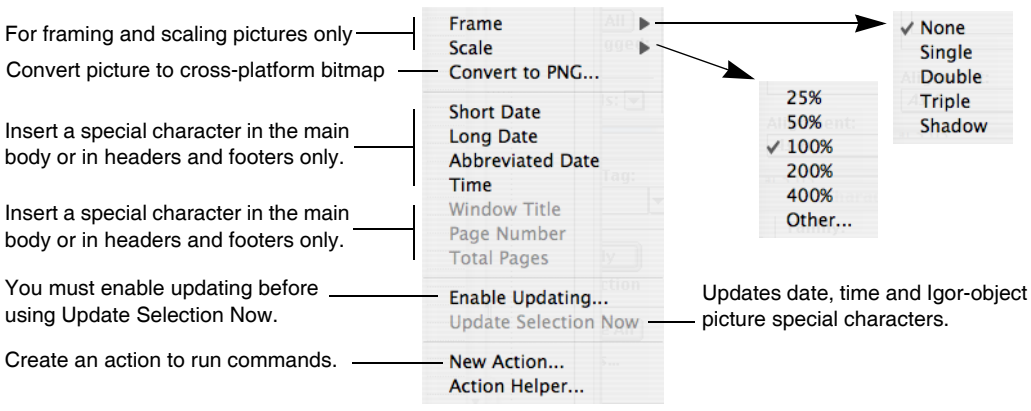
The special character name is used only for specialized applications and usually you can ignore it. You can use the name with the Notebook findSpecialCharacter operation to select special characters. You can get a list of special character names from the **SpecialCharacterList** function (see page V-586) and get information using the **SpecialCharacterInfo** function (see page V-584).

On Macintosh when you copy a graph, table, or layout and paste it into a notebook, an Igor-object picture is created (see **Using Igor-Object Pictures** on page III-21). The Igor-object picture, like any notebook picture, is a special character and thus has a special character name, which whenever possible is the same as the source graph, table, or layout window name. However, this may not always possible such as when, for example, you paste Graph0 twice into a notebook, the first special character will be named *Graph0* and the second *Graph0_1*.

The Special Submenu

Using the Special submenu of the Notebook menu you can:

- Frame or scale pictures.
- Insert special characters.
- Control updating of special characters.
- Convert a picture to cross-platform PNG format.
- Specify an action character that executes commands.

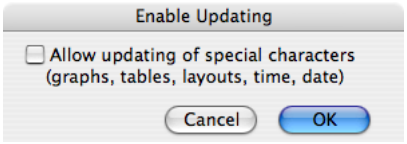


Scaling Pictures

You can scale a picture by choosing an item from the Scale submenu or by using the Notebook command line operation. There is currently no way to scale a picture using the mouse.

Updating Special Characters

The window title, page number and total number of pages are dynamic characters—Igor automatically updates them when you print a notebook. These are useful for headers and footers. All other kinds of special characters are not dynamic but Igor makes it easy for you to update them if you need to, using the Update Selection Now or Update All Now items in the Special menu.



To prevent inadvertent updating, Igor disables these items until you enable updating, using the Enable Updating item in the Special menu. This enables updating for the active notebook.

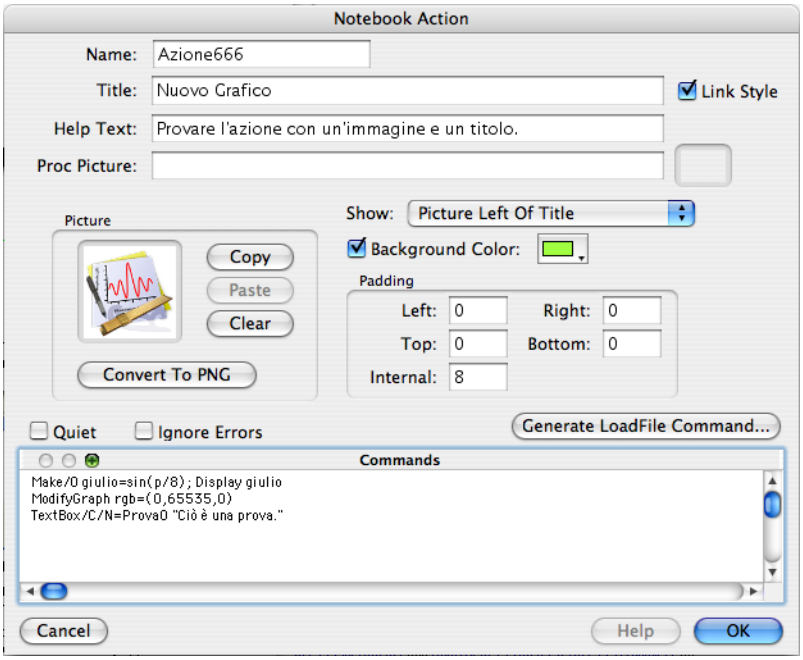
If you are using a notebook as a form for generating reports, you will probably want to enable updating. However, if you are using it as a log of what you have done, you will want to leave updating in the disabled state.

Notebook Action Special Characters

An action is a special character that runs commands when clicked. Use actions to create interactive notebooks, which can be used for demonstrations or tutorials. Help files are formatted notebook files so actions can also be used in help files.

You create actions in a formatted text notebook. You can invoke actions from formatted text notebooks or from help files.

For a demonstration of notebook actions, see the Notebook Actions Demo experiment.



To create an action use the **NotebookAction** operation (see page V-456) or choose Notebook→Special→New Action to get the Notebook Action dialog.

Each action has a name that is unique within the notebook.

The title is the text that appears in the notebook. The text formatting of the notebook governs the default text formatting of the title.

If the Link Style checkbox is selected, the title is displayed like an HTML link — blue and underlined. This style overrides the color and underline formatting applied to the action through the Notebook menu.

The help text is a tip that appears when the cursor is over an action. On Macintosh you must first turn on Igor Tips in the Help menu. On Windows, help text appears in the status bar.

An action can have an associated picture that is displayed instead of or in addition to the title. There are two ways to specify a picture. You can paste one into the dialog using the Paste button or you can reference a Proc Picture stored in a procedure file. The latter source may be useful for advanced programmers (see **Proc Pictures** on page IV-43 for details).

For most purposes it is better to use a picture rather than a Proc Picture. One exception is if you have to use the same picture many times in the notebook, in which case you can save disk space and memory by using a Proc Picture.

If you designate a Proc Picture using a module name (e.g., `MyProcPictures#MyPicture`), then the Proc Picture must be declared static.

If you specify both a Proc Picture and a regular picture, the regular picture is displayed. If you specify no regular picture and your Proc Picture name is incorrect or the procedure file that supplies the Proc Picture is not open or not compiled, `"???"` is displayed in place of the picture.

In order for a picture to display correctly on both Macintosh and Windows, it must be in a cross-platform format such as PNG. You can convert a picture to PNG by clicking the Convert To PNG button. This affects the regular picture only. Proc Pictures are always in a cross-platform format.

Pictures and Proc Picture in actions are drawn transparently. The background color will show through white parts of the picture unless the picture explicitly erases the background.

The action can display one of six things as determined by the Show popup menu:

- The title.
- The picture.
- The picture below the title.
- The picture above the title.
- The picture to the left of the title.
- The picture to the right of the title.

If there is no picture and you choose one of the picture modes, just the title will be displayed.

You can add padding to any external side of the action content (title or picture). The Internal Padding value sets the space between the picture and the title when both are displayed. All padding values are in points.

If you enable the background color, the rectangle enclosing the action content is painted with the specified color.

You can enter any number of commands to be executed in the Commands area. When you click the action, Igor sends each line in the Commands area to the Operation Queue, as if you called the Execute/P operation, and the commands are executed.

In addition to regular commands, you can enter special operation queue commands like INSERTINCLUDE, COMPILEPROCEDURES, and LOADFILE. These are explained under **Operation Queue** on page IV-250.

For sophisticated applications, the commands you enter can call functions that you define in a companion "helper procedure file" (see **Notebook Action Helper Procedure Files** on page III-20).

If the Quiet checkbox is selected, commands are not sent to the history area after execution.

If the Ignore Errors checkbox is selected then command execution errors are not reported via error dialogs.

The Generate LoadFile Command button displays an Open File dialog and then generates an Execute/P command to load the file into Igor. This is useful for generating a command to load a demo experiment, for example. This button inserts the newly-generated command at the selection point in the command area so, if you want the command to replace any pre-existing commands, delete any text in the command area before clicking the button. If the selected file is inside the Igor Pro Folder or any subdirectory, the generated path will be relative to the Igor Pro Folder. Otherwise it will be absolute.

Modifying Action Special Characters

You can modify an existing action by Control-clicking (*Macintosh*) or right-clicking (*Windows*) on it and choosing Modify Action from the pop-up menu, or by selecting the action special character, and nothing else, and then choosing Notebook→Special→Modify Action.

If you have opened a notebook as a help file and want to modify an action, you must close the help file (press Option or Alt and click the close button) and reopen it as a notebook (choose File→Open File→Notebook). After editing the action, save the changes, close the notebook, and reopen it as a help file (choose File→Open File→Help File).

Modifying the Action Frame

If the notebook action has a picture, you can frame the action by choosing a frame style from the Notebook→Special→Frame submenu.

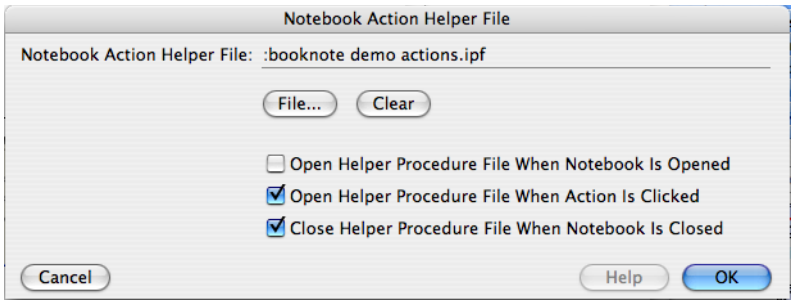
Modifying the Action Picture Scaling

If the notebook action has a picture, you can scale the picture by choosing an item from the Notebook→Special→Scale submenu.

Notebook Action Helper Procedure Files

In some instances you may want an action to call procedures in an Igor procedure file. The notebook action helper procedure file feature provides a convenient way to associate a notebook or help file with a procedure file.

Each formatted notebook (and consequently each help file) can designate only one procedure file as an action helper procedure file. Before choosing the helper file you must save the notebook as a standalone file on disk. Then choose Notebook→Special→Action Helper.



Click the File button to choose the helper procedure file for the notebook.

For most cases we recommend that you name your action helper procedure file with the same name as the notebook but with the .ipf extension. This will indicate that the files are closely associated.

The helper file will usually be located in the same directory as the notebook file. Less frequently, it will be in a subdirectory or in a parent directory. It must be located on the same volume as the notebook file because Igor finds the helper using a relative path, starting from the notebook directory. If the notebook file is moved, the helper procedure file must be moved with it so that Igor will be able to find the helper using the relative path.

If Open Helper Procedure File When Notebook Is Opened is selected, the helper procedure file is opened along with the notebook. This checkbox can usually be left deselected. However, if you use Proc Pictures stored in the helper file, you should select it so that the pictures can be correctly rendered when the notebook is opened.

If Open Helper Procedure File When Action Is Clicked is selected, then, when you click an action, the procedure file loads, compiles, and executes automatically. This should normally be selected.

In both of these situations, the procedure file loads as a "global" procedure file, which means that it is not part of the current experiment and is not closed when creating a new experiment.

If Close Helper procedure File When Notebook Is Closed is selected and you kill a notebook or help file that has opened a helper file, the helper file is also killed. This should normally be selected.

To avoid unanticipated name conflicts between procedures in your helper file and elsewhere, it is a good idea to declare the procedures static (see **Static Functions** on page IV-83). In order to call such private routines you also need to assign a module name to the procedure file and use the module name when invoking the routines (see **Regular Modules** on page IV-212). For an example see the Notebook Actions Demo experiment.

Using Igor-Object Pictures

You create a picture from an Igor graph, table or page layout by choosing Edit→Export Graphics to copy a picture to the clipboard. For graphs or layouts you can also choose Edit→Copy. When you do this, Igor embeds in the picture some information about the graph, table or layout from which the picture was generated. We call this kind of picture an "Igor-object" picture.

The embedded information contains the name of the window from which the picture was generated, the date/time at which it was generated, the size of the picture and the export mode used to create the picture. Igor uses this information to automatically update the picture when you request it.

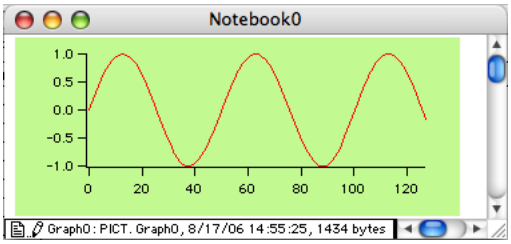
Igor can not link Igor-object pictures to a window in a different Igor experiment.

See **Quality of Printed Pictures (Macintosh)** on page III-24 for instructions on getting the best quality pictures from Igor graphs and layouts.

Updating Igor-Object Pictures

Before updating Igor object pictures, you must enable updating using the Notebook→Special→Enable Updating menu item. This is a per-notebook setting.

When you click an Igor-object picture, Igor displays the name of the object from which the picture was generated and the time at which it was generated in the notebook's status area.



The first Graph0 shown in the status area is the name of the picture special character and the second Graph0 is the name of the source graph for the picture. There is no requirement that these be the same but they usually will be.

If you change the Igor graph, table or layout, you can update the associated picture by selecting it and choosing Update Selection Now from the Notebook→Special menu or by right-clicking and choosing Update Selection from the contextual menu. You can update all Igor-object pictures as well as any other special characters in the notebook by clicking anywhere so that nothing is selected and then choosing Update All Now from the Notebook→Special menu.

An Igor object picture can be updated even if it was created on another platform using a platform-dependent format. For example, you can create an EMF Igor object picture on Windows and paste it into a notebook. If you open the notebook on Macintosh, the EMF will display as a gray box because EMF is a Windows-specific format. However, if you right-click the EMF picture and choose Update Selection, Igor will regenerate it using a Macintosh format.

An Igor-object picture never updates unless you do so. Thus you can keep pictures of a given object taken over time to record the history of that object.

The Size of the Picture

The size of the picture is determined when you initially paste it into the notebook. If you update the picture, it will stay the same size, even if you have changed the size of the graph window from which the picture is derived. Normally, this is the desired behavior. If you want to change the size of the picture in the notebook, you need to repaste a new picture over the old one.

Activating The Igor-Object Window

You can activate the window associated with an Igor-object picture by double-clicking the Igor object picture in the notebook. If the window exists it is activated. If it does not exist but the associated window recreation macro does exist, Igor runs the window recreation macro.

Breaking the Link Between the Object and the Picture

Lets say you create a picture from a graph and paste it into a notebook. Now you kill the graph. When you click the picture, Igor displays a question mark after the name of the graph in the notebook's status area to indicate that it can't find the object from which the picture was generated. Igor can not update this picture. If you recreate the graph or create a new graph with the same name, this reestablishes the link between the graph and the picture.

If you change the name of a graph, this breaks the link between the graph and the picture. To reestablish it, you need to create a new picture from the graph and paste it into the layout.

Compatibility Issues

Prior to Igor Pro 6.10, Igor-object pictures worked only on Macintosh and only for pictures created using the Macintosh PICT format. In Igor Pro 6.10 and later, the Igor-object picture information is embedded for the following picture formats:

Format	Platform
PDF	Macintosh only
Enhanced Metafile	Windows only
Bitmap PICT	Macintosh only
DIB	Windows only
PNG Image	Cross-platform
JPEG Image	Cross-platform
TIFF Image	Cross-platform

If you open a pre-Igor Pro 6.10 notebook in Igor Pro 6.10, existing Igor object pictures in Macintosh PICT format are recognized as Igor object pictures. If you do an update, they will be converted to PDF on Macintosh and to EMF on Windows. PDF pictures are not supported in notebooks prior to Igor Pro 6.10 and therefore these pictures will show up as gray boxes if you then open the notebook in an older version of Igor.

A Windows format picture, when updated on Macintosh, is converted to a Macintosh format, and vice versa.

Cross-Platform Pictures

If you want to create a notebook that contains pictures that display correctly on both Macintosh and Windows, you can use the PNG (Portable Network Graphics) format. If some pictures are already in JPEG or TIFF format, these too will display correctly on either platform.

You can convert other types of pictures to PNG using the Convert to PNG item in the Special submenu of the Notebook menu.

Page Breaks

When you print a notebook, Igor automatically goes to the next page when it runs out of room on the current page. This is an automatic page break.

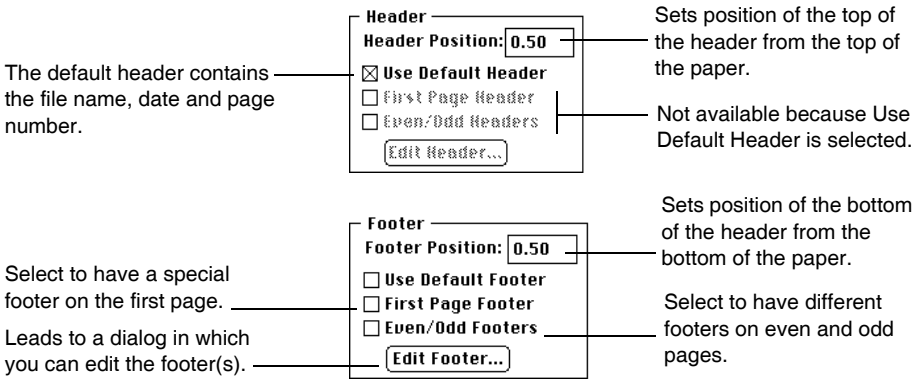
In a formatted notebook, you can insert a *manual* page break using the Insert Page Break item in the Edit menu. Igor displays a manual page break as a dashed line across the notebook window. You can't insert a manual page break into a plain notebook.

Unfortunately, there is no way to see where Igor will put automatic page breaks other than by printing the document. This is a missing feature.

Headers and Footers

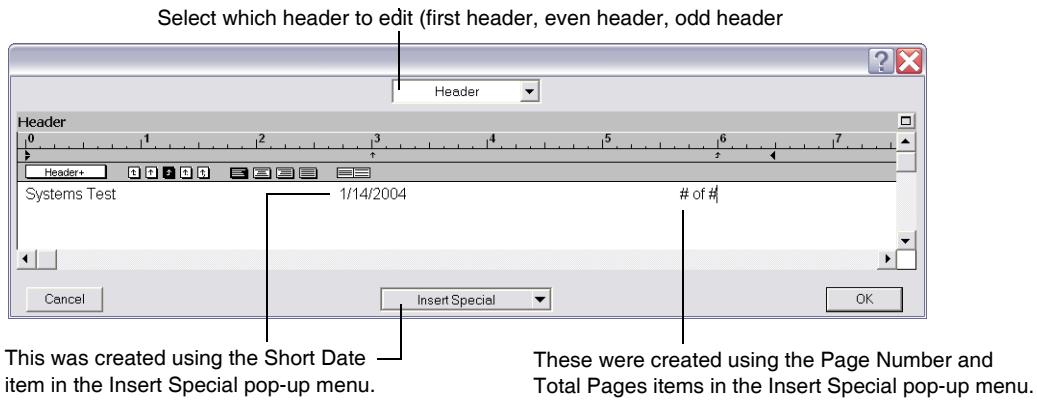
Both formatted and plain notebooks can have headers or footers.

You create headers and footers using the Document Settings dialog. This illustration shows just the parts of this dialog that affect headers and footers.



For most uses, the default header or default footer will be sufficient. The default header consists of the date, window title and page number. The other options are intended for use in fancy reports.

The Edit Header button and Edit Footer buttons lead to a dialog that looks like this.



The Page Number and Number of Pages special characters, shown above, are displayed on the screen as # characters but are printed using the actual page number and number of pages in the document. The Date, Time and Window Title special characters are automatically updated when the document is printed.

In the notebook header or footer dialog you can change text properties (font, size, style, color) for the entire header using the Ruler pop-up menu. *On the Macintosh* you can also use the Notebook menu in the main menu bar to set text properties for selected text. *Under Windows* the Notebook menu is not available and so there is no way to set text properties for selected text from within the dialog. A workaround is to edit the header or footer text in the notebook window, copy it, and then paste it into the header or footer dialog.

On the Macintosh, plain text files can have headers and footers. They are stored in the resource fork of the notebook file; the data fork is plain text so that other programs see only text. Igor stores the header and footer in a format that other applications do not understand.

Under Windows, files have no resource fork so plain text headers and footers can not be saved. Consequently, when you open a plain text notebook, its headers and footers revert to the preferred headers and footers, as set by the Capture Notebooks Prefs dialog. See also **Notebook Issues** on page III-407.

Printing Notebooks

On *Macintosh*, to print an entire notebook, click so that no text is selected and then choose Print Notebook from the File menu. To print part of a notebook, select the section that you want to print and then choose Print Notebook Selection from the File menu.

On Windows, choose whether to print the entire notebook or just the selection in the Print dialog.

Quality of Printed Pictures (*Macintosh*)

Igor prints graphs and page layouts at the highest resolution available on the chosen printer. However, on Macintosh notebooks are printed at a resolution of 72 dots per inch. The reason for this is that font measurements change at higher resolutions causing a lack of fidelity between the screen and the printed document.

Printing at low resolution causes smooth curves in pictures to appear jagged. This is a problem mostly for Igor graphs that you paste into a notebook. This problem can be avoided by using the HiRes PICT format or a PNG format with 4X resolution when creating the picture using Edit→Export Graphics.

Macintosh printer drivers do not handle HiRes PICTs properly when they are printed at 72 dots per inch. They print the pictures at 72 dpi even though the pictures have higher resolution. To cope with this, Igor prints HiRes PICTs in notebooks using a high resolution bitmap technique that circumvents the 72 dpi limitation.

When you use the **PrintNotebook** operation (see page V-502), you can override Igor's default behavior to make it print HiRes PICTs using the technique of your choice. The following commands illustrate this:

```
PrintNotebook/B=1 Notebook0   // use bitmap print HiRes PICT
PrintNotebook/B=0 Notebook0   // don't use bitmap print HiRes PICT
```

For a more detailed explanation of the different picture export methods and when to use which method, see Chapter III-5, **Exporting Graphics (Macintosh)**, and Chapter III-6, **Exporting Graphics (Windows)**.

Quality of Printed Pictures (*Windows*)

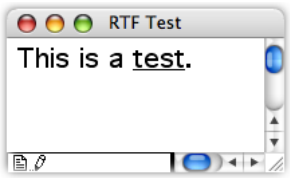
On Windows, Igor prints notebooks at the default printer resolution. You don't need to take any special measures to get smooth curves to print smoothly. When creating a picture from a graph for the purpose of pasting into a notebook, use the Enhanced Metafile format or a PNG format with 4X resolution in the Export Graphics dialog.

Import and Export Via Rich Text Format Files

The Rich Text Format (RTF) is a file format created by Microsoft Corporation for exchanging formatted text documents between programs. Microsoft also calls it the Interchange format. Many word processors and some drawing and page layout programs can import or export RTF. RTF can also be used to move a document from one type of computer to another, Mac to PC, for example.

You can save an Igor plain or formatted notebook as an RTF file and you can open an RTF file as an Igor formatted notebook. You may find it useful to collect text and pictures in a notebook and to later transfer it to your word processor for final editing.

An RTF file is a plain text file that contains RTF codes. For example, here is an Igor notebook and the corresponding RTF codes.




```
{\rtf1\mac\deff1{\fonttbl{\f1\fnil Geneva;}{\f2\fnil Times;}}
{\stylesheet{\s0\sbasedon222\snext0\f1\fs20 Normal;}}
\deftab720\margl1080\margr1080\margt1080\margb1080\pgnstarts1
\sectd\headery720\footery720
\pard\plain
{\header\pard\plain\s1\tqc\tx5040\tqr\tx10060\fs20
\chdate Untitled0 \chpgn \par
}
\pard\plain
\pard\plain\s0\fs20
\fs48 This is a \ul test\ulnone .\par
}
```

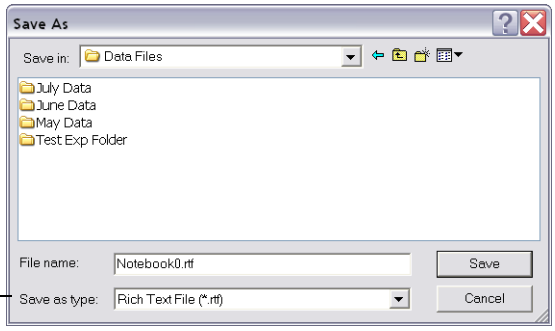
The "\rtf" code at the start of the file is what identifies a text file as an RTF file. Other codes define the text, pictures, document formats, paragraph formats, and text formats and other aspects of the file.

When Igor writes an RTF file from a notebook, it must generate a complex sequence of codes. When it reads an RTF file, it must interpret a complex sequence of codes. The RTF format is very complicated, has evolved and allows some flexibility. As a result, each program writes and interprets RTF codes somewhat differently. Because of this and because of the different feature sets of different programs, RTF translation is sometimes imperfect and requires that you do manual touchup.

Saving an RTF File

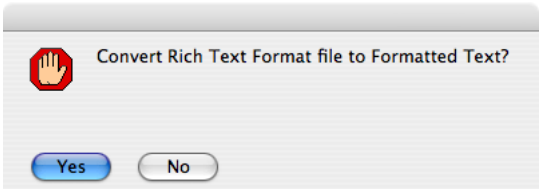
To create an RTF file, choose Save Notebook As from the File menu.

When selected in the "Save as type:" or "Format:" pop-up menu, Igor creates an RTF file from the active notebook.



Opening an RTF File

When Igor opens a plain text file as a notebook, it looks for the "\rtf" code that identifies the file as an RTF file. If it sees this code, it asks if you want to convert the rich text codes into an Igor formatted notebook.



If you answer Yes, Igor creates a new, formatted notebook. It then interprets the RTF codes and sets the properties and contents of the new notebook accordingly. When the conversion is finished, you sometimes need to fix up some parts of the document that were imperfectly translated.

If you answer No, Igor opens the RTF file as a plain text file. Use this to inspect the RTF codes and, if you are so inclined, to tinker with them.

Rich Text Format Graphics

This table shows how Igor deals with various graphics formats when importing an RTF file:

	Macintosh Igor	Windows Igor
Macintosh PICT	Loads picture. Draws picture.	Loads picture. Draws gray box.
Windows DIB	Loads picture. Draws gray box.	Loads picture. Draws picture.
Windows DDB	Not supported. Skips picture.	Not supported. Skips picture.
Windows Metafile	Not supported. Skips picture.	Loads picture. Converts to Enhanced Metafile. Draws picture.
Enhanced Metafile	Loads picture. Draws gray box.	Loads picture. Draws picture.
PNG	Loads picture. Draws picture.	Loads picture. Draws picture.
JPEG	Loads picture. Draws picture.	Loads picture. Draws picture.

The Windows version of Igor displays Macintosh PICTs as gray boxes. However, if you save the picture in an Igor file and open it with the Macintosh version of Igor, it will be displayed correctly. Similarly, The Macintosh version of Igor displays Enhanced Metafiles as gray boxes but the Windows version displays them correctly.

In Microsoft Word, a picture can be "inline" or "floating". The floating variety introduces considerable complexity in the RTF file. Igor loads inline pictures from RTF files but ignores floating pictures.

This table shows how Igor writes pictures when exporting an RTF file:

	Macintosh Igor	Windows Igor
Macintosh PICT	Writes PICT.	Exports gray box.
Windows DIB	Exports gray box.	Writes 4X PNG.
Enhanced Metafile	Exports gray box.	Writes Enhanced Metafile.
PNG	Writes PNG.	Writes PNG.
JPEG	Writes JPEG.	Writes JPEG.
PDF	Writes 4X PNG.	Exports gray box.
TIFF	Writes 4X PNG.	Writes 4X PNG.
EPS	Writes 4X PNG. *	Writes 4X PNG. *

* When exporting EPS, the PNG is created from the EPS preview only. If there is no EPS preview, a gray box is exported.

Exporting a Notebook as HTML

Igor can export a notebook in HTML format. HTML is the format used for Web pages. For a demo of this feature, see "Igor Pro Folder:Examples:Feature Demos:Web Page Demo.pxp".

This feature is intended for two kinds of uses. First, you can export a simple Igor notebook in a form suitable for immediate publishing on the Web. This might be useful, for example, to automatically update a Web page or to programmatically generate a series of Web pages.

Second, you can export an elaborate Igor notebook as HTML, use an HTML editor to improve its formatting or tweak it by hand, and then publish it on the Web. It is unlikely that you could use Igor alone to create an

elaborately formatted Web page because there is a considerable mismatch between the feature set of HTML and the feature set of Igor notebooks. For example, the main technique for creating columns in a notebook is the use of tabs. But tabs mean nothing in HTML, which uses tables for this purpose.

Because of this mismatch between notebooks and HTML, and so your Web page works with a wide variety of Web browsers, we recommend that you keep the formatting of notebooks which you intend to write as HTML files as simple as possible. For example, tabs and indentation are not preserved when Igor exports HTML files, and you can't rely on Web browsers to display specific fonts and font sizes. If you restrict yourself to plain text and pictures, you will achieve a high degree of browser compatibility.

There are two ways to export an Igor notebook as an HTML file:

Choose File→Save Notebook As
Using the `SaveNotebook/S=5` operation

The **SaveNotebook** operation (see page V-540) includes a /H flag which gives you some control over the features of the HTML file:

The file's character encoding.
Whether or not paragraph formatting (e.g., alignment) is exported.
Whether or not character formatting (e.g., fonts, font sizes) is exported.
The format used for graphics.

When you choose File→Save Notebook As, Igor uses the following default parameters:

Character encoding: UTF-8 (see **HTML Character Encoding** on page III-29).
Paragraph formatting is not exported.
Character formatting is not exported.
Pictures are exported in the PNG (Portable Network Graphics) format.

By default, paragraph and character formatting is not exported because this formatting is often not supported by some Web browsers, is at cross-purposes with Web browser behavior (e.g., paragraph space-before and space-after), or is customarily left in the hands of the Web viewer (e.g., fonts and font sizes).

For creating simple Web pages that work with a majority of Web browsers, this is all you need to know about Igor's HTML export feature. To use advanced formatting, to use non-Roman characters, to use different graphics formats, and to cope with diverse Web browser behavior, you need to know more. Unfortunately, this can get quite technical.

HTML Standards

Igor's HTML export routine writes HTML files that conform to the HTML 4.01 specification, which is available from:

`http://www.w3.org/TR/1999/PR-html40-19990824`

It writes style information that conforms to the CSS1 (Cascading Style Sheet - Level 1) specification, which is available from:

`http://www.w3.org/TR/1999/REC-CSS1-19990111`

HTML Horizontal Paragraph Formatting

Tabs mean nothing in HTML. A tab behaves like a single space character. Consequently, you can not rely on tabs for notebooks that are intended to be written as HTML files. HTML has good support for tables, which make tabs unnecessary. However, Igor notebooks don't support tables. Consequently, there is no simple way to create an HTML file from an Igor notebook that relies on tabs for horizontal formatting.

HTML files are often optimized for viewing on screen in windows of varying widths. When you make the window wider or narrower, the browser automatically expands and contracts the width of the text. Consequently, the roles played by the left margin and right margin in notebooks are unnecessary in HTML files. When Igor writes an HTML file, it ignores the left and right paragraph margin properties.

HTML Vertical Paragraph Formatting

The behavior of HTML browsers with regard to the vertical spacing of paragraphs makes it difficult to control vertical formatting. For historical reasons, browsers typically add a blank line after each paragraph (<P>) element and they ignore empty paragraph elements. Although it is possible to partially override this behavior, this only leads to more problems.

In an Igor notebook, you would usually use the space-before and space-after paragraph properties in place of blank lines to get paragraph spacing that is less than one line. However, because of the aforementioned browser behavior, the space-before and space-after would add to the space that the browser already adds and you would get more than one line's space when you wanted less. Consequently, Igor ignores the space-before and space-after properties when writing HTML files.

Because of this browser behavior, you will get close to WYSIWYG results only if you use one blank line between paragraphs in your Igor notebook.

The minimum line height property is written as the CSS1 line-height property, which does not serve exactly the same purpose. This will work correctly so long as the minimum line height that you specify is greater than or equal to the natural line height of the text.

HTML Character Formatting

In an Igor notebook, you might use different fonts, font sizes, and font styles to enhance your presentation. An HTML file is likely to be viewed on a wide range of computer systems and it is likely that your enhancements would be incorrectly rendered or would be a hindrance to the reader. Consequently, it is customary to leave these things to the person viewing the Web page.

If you use the **SaveNotebook** operation (see page V-540) and enable exporting font styles, only the bold, underline and italic styles are supported.

In notebooks, the vertical offset character property is used to create subscripts and superscripts. When writing HTML, Igor uses the CSS vertical-align property to represent the notebook's vertical offset. The HTML property and the Igor notebook property are not a good match. Also, some browsers do not support the vertical-align property. Consequently, subscripts and superscripts in notebooks may not be properly rendered in HTML. In this case, the only workaround is to use a picture instead of using the notebook subscript and superscript.

HTML Pictures

If the notebook contains one or more pictures, Igor writes PNG or JPEG picture files to a "media" folder. For example, if the notebook contains two pictures and you save it as "Test.htm", Igor writes the file Test.htm and creates a folder named TestMedia. It stores in the TestMedia folder two picture files: Picture0.png (or .jpg) and Picture1.png (or .jpg). The names of the picture files are always of the form Picture<N> where N is a sequential number starting from 0. If the folder already exists when Igor starts to store pictures in it, Igor deletes all files in the folder whose names start with "Picture", since these files are most likely left over from a previous attempt to create the HTML file.

When you choose Save Notebook As from the File menu, Igor always uses the PNG format for pictures. If you want to use the JPEG format, you must execute a **SaveNotebook** operation (see page V-540) from the command line, using the /S=5 flag to specify HTML and the /H flag to specify the graphics format.

PNG is a lossless format that is excellent for storing web graphics and is supported by virtually all recent web browsers. However, very old browsers (e.g., Internet Explorer 4.5) do not support it.

JPEG is a lossy format commonly used for web graphics. Prior to Igor Pro 5, Igor relied on QuickTime to write JPEG files. This was true even when running on Windows. Now, Igor has built-in support for JPEG. Igor still uses QuickTime if it is available but uses built-in routines if not.

HTML does not support Igor's double, triple, or shadow picture frames. Consequently, when writing HTML, all types of notebook frames are rendered as HTML thin frames.

HTML Character Encoding

Character encoding refers to the way in which a particular character is represented in a computer's memory or in a computer file. For example, in the Macintosh Roman character encoding, the number 165 is the character code for a bullet symbol. However, in the Windows character encoding, the number 165 is the character code for a yen symbol. These are but two of dozens of possible character encodings. This raises the question of what a Web browser should display when it encounters a particular character code.

An Internet standard called "RFC 2070" addresses this question. Unfortunately, the recommendations in RFC 2070 are not generally understood by people who create Web pages, are difficult for an HTML-generator program to implement and are ignored by many popular Web browsers. For more information on these issues, a good jumping off point is:

```
<http://www.cs.tut.fi/~jkorpela/chars/index.html>.
```

Igor supports three kinds of character encodings:

UTF-2 UTF-2 encoding is 16-bit Unicode. Unicode is a system for uniquely representing nearly all of the characters in most of the commonly-used languages of the world. It provides a simple way to create a document that contains characters from more than one writing system, such as English and Japanese or English and Greek symbols (the characters in the Symbol font).

There are two main problems with UTF-2. First, most text editors don't support it. Second, many Web browsers don't support it. For these reasons, it is better to use UTF-8.

UTF-8 UTF-8 encoding is a kind of packed Unicode. Whereas 16-bit Unicode represents each character using two bytes, UTF-8 represents each character using 1 to 6 bytes.

UTF-8 has a very nice property — all of the US-ASCII characters (0x00-0x7F) are represented using a single byte, the same byte used in US-ASCII. This means that you can edit the English text in UTF-8 documents in most text editors.

UTF-8 is supported by most recent Web browsers and UTF-8 support will continue to expand. For this reason, and because it can represent non-Roman text, Igor uses UTF-8 by default when exporting a notebook as HTML.

Native There may be some cases in which you will not want to use UTF-8. The most reason for doing this is to work around Web browser bugs. If you need to do this, you can use "native" encoding in the **SaveNotebook** operation (see page V-540).

When you use native encoding, the document is created using the native character set of the operating system. On Macintosh, this equates to "mac" (the Mac OS Roman character set). On Windows, it equates to "Windows-1252" (the Windows Western character set).

The display of a document using one of these encodings is more browser-dependent than UTF-8 when the document contains "high-ASCII" and non-Roman characters. Also, non-Roman text such as symbols from the Symbol font and Asian characters depend on the use of specific fonts which may not be available in the Web viewer's system.

In these encodings, the browser can not tell from a character code if the character is Roman, Symbol or Asian. For this reason, if you use a mix of character sets, you must use the SaveNotebook /H flag to enable exporting fonts to the HTML file.

Others Finally, there is a technique that may be of use for writing Asian text if, for some reason, UTF-8 does not work. Assume you have a notebook that consists almost entirely of Japanese text, which on both Macintosh and Windows uses the "Shift-JIS" encoding. Using SaveNotebook/H, you can specify that the characters in the file use Shift-JIS. Igor will write a Content-Type meta tag in the file that specifies Shift-JIS. Web browsers that correctly interpret the Content-Type meta tag will display the characters correctly.

Embedding HTML Code

If you are knowledgeable about HTML, you may want to access the power of HTML without completely giving up the convenience of having Igor generate HTML code for you. You can do this by embedding HTML code in your notebook, which you achieve by simply using a ruler named HTMLCode.

Normally, Igor translates the contents of the notebook into HTML code. However, when Igor encounters a paragraph whose ruler is named HTMLCode, it writes the contents of the paragraph directly into the HTML file. Here is a simple example:

Living things are generally classified into 5 kingdoms:

```
<OL>
<LI>Monera
<LI>Protista
<LI>Fungi
<LI>Plantae
<LI>Animalia
</OL>
```

In this example, the gray lines are governed by the HTMLCode ruler. Igor writes the text in these line directly to the HTML file. This example produces a numbered list, called an "ordered list", which is announced using the HTML "OL" tag.

By convention, we make the ruler font color for the HTMLCode ruler gray. This allows us to distinguish at a glance the HTML code from the normal notebook text. The use of the color gray is merely a convention. It is the fact that the ruler is named HTMLCode that makes Igor write the contents of these paragraphs directly to the HTML file.

Here is an example that shows how to create a simple table:

```
<TABLE border="1" summary="Example of creating a table in HTML.">
<CAPTION><EM>A Simple Table</EM></CAPTION>
<TR><TH><TH>Col 1<TH>Col 2<TH>Col 3
<TR><TH>Row 1<TD>10<TD>20<TD>30
<TR><TH>Row 2<TD>40<TD>50<TD>60
</TABLE>
```

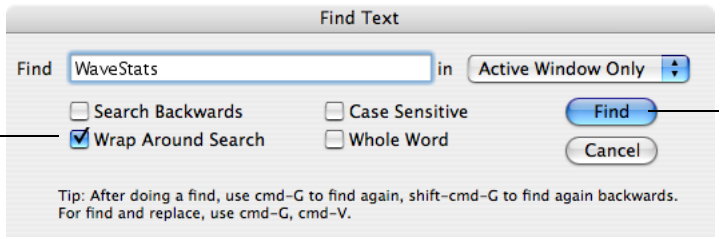
Here is an example that includes a link:

```
<P>Visit the <A HREF="http://www.wavemetrics.com/">WaveMetrics</A> web site</P>
```

Finding Text

You can access the Find Text dialog via the Edit menu or by pressing Command-F (*Macintosh*) or Ctrl+F (*Windows*).

Select to search the entire window. Deselect to search to the end of the document (or start of the document if searching backwards).



Press Shift while clicking to temporarily reverse the search direction.

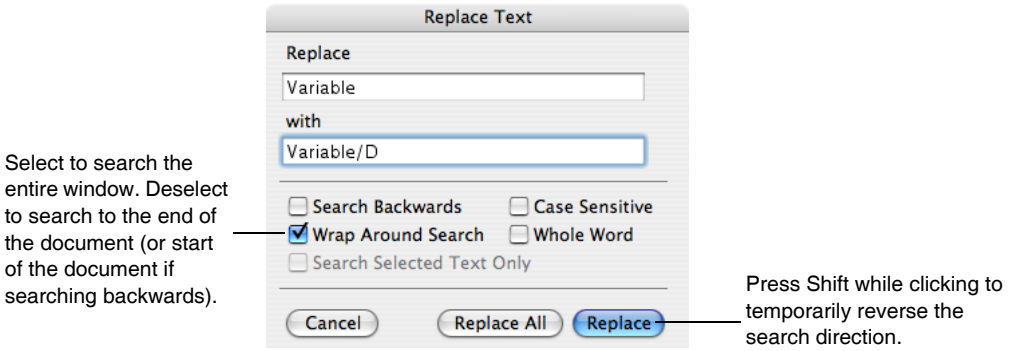
You can search for the next occurrence of a string, without using the dialog, by selecting the string and choosing Find Selection in the Edit menu. The keyboard shortcut is Command-Control-H (*Macintosh*) or Ctrl+H (*Windows*).

After doing a find, you can continue searching for the same text again by choosing Find Same in the Edit menu. The keyboard shortcut is Command-G (*Macintosh*) or Ctrl+G (*Windows*). You can continue searching for the same text, but in the reverse direction, by pressing Command-Shift-G (*Macintosh*) or Ctrl+Shift+G (*Windows*).

You can also perform a Find on multiple help, procedure and notebook windows at one time. See **Finding Text in Multiple Windows** on page II-69.

Replacing Text

You can access the Replace Text dialog via the Edit menu or by pressing Command-R (*Macintosh*) or Ctrl+R (*Windows*).



Another method for searching and replacing consists of copying the replacement text to the Clipboard and using Find, Command-F (*Macintosh*) or Ctrl+F (*Windows*), followed by a series of Paste, Command-V (*Macintosh*) or Ctrl+V (*Windows*), and Find Same commands, Command-G (*Macintosh*) or Ctrl+G (*Windows*).

Notebook Names, Titles and File Names

This table explains the distinction between a notebook's name, its title and the name of the file in which it is saved.

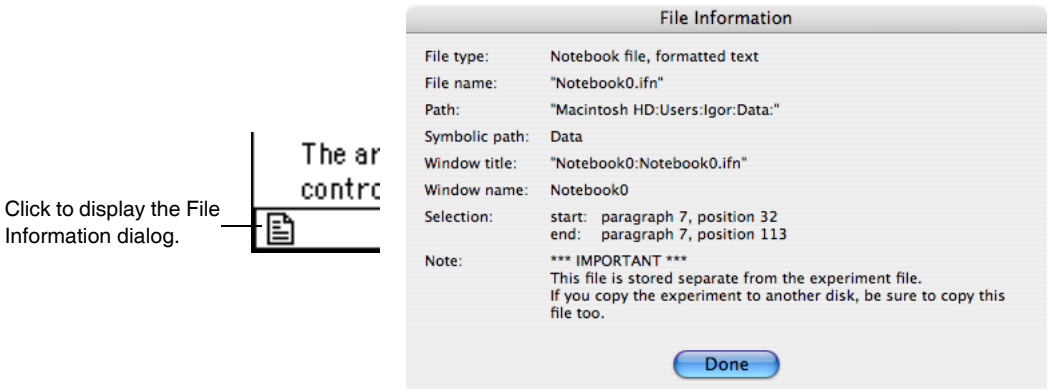
Item	What It Is For	How It Is Set
Notebook name	Used to identify a notebook from an Igor command.	Igor automatically gives new notebooks names of the form Notebook0. You can change it using the Window Control dialog or using the DoWindow/C operation.
Notebook title	For visually identifying the window. The title appears in the window bar at the top of the window and in the Other Windows submenu of the Windows menu.	Initially, Igor sets the title to the concatenation of the notebook name and the file name. You can change it using the Window Control dialog or using the DoWindow/T operation.
File name	This is the name of the file in which the notebook is stored.	You enter this in the New Notebook dialog. Change it on the desktop.

Igor automatically opens notebooks that are part of an Igor experiment when you open the experiment. If you change a notebook's file name outside of the experiment, Igor will be unable to automatically open it and will ask for your help when you open the experiment.

A notebook file stored inside a packed experiment file does not exist separately from the experiment file, so there is no way or reason to change the notebook's file name.

Notebook Info Dialog

You can get general information on a notebook by selecting the Info item in the Notebook menu or by clicking the icon in the bottom/left corner of the notebook.



The Note information shows you whether the notebook has been saved and if so whether it is stored in a packed experiment file, in an unpacked experiment folder or in a stand-alone file. The selection information may be of use to programmers writing Igor procedures to manipulate notebooks.

Programming Notebooks

Advanced users may want to write Igor procedures to automatically log results or generate reports using a notebook. The operations that you would use are briefly described here. See Chapter V-1, **Igor Reference**, for details.

Operation	What It Does
NewNotebook	Creates a new notebook window.
OpenNotebook	Opens an existing file as a notebook.
SaveNotebook	Saves an existing notebook to disk as a stand-alone file or packed into the experiment file.
PrintNotebook	Prints all of a notebook or just the selected text.
Notebook	Provides control of the contents and all of the properties of a notebook except for headers and footers. Also sets the selection and to search for text or graphics.
NotebookAction	Creates or modifies notebook action special characters.
GetSelection	Retrieves the selected text.
DoWindow/K	Kills a notebook.

There is currently no way to set headers and footers from Igor procedures. A workaround is to create a stationery (*Macintosh*) or template (*Windows*) notebook file with the headers and footers that you want and to open this instead of creating a new notebook.

In addition, the **SpecialCharacterList** function (see page V-586) and **SpecialCharacterInfo** function (see page V-584) may be of use.

The Notebook Demo #1 experiment, in the Examples:Feature Demos folder, provides a simple illustration of generating a report notebook using Igor procedures.

See **Notebooks as Subwindows in Control Panels** on page III-94 for information on using a notebook as a user-interface element.

Some example procedures follow.

Logging Text

This example shows how to add an entry to a log. Since the notebook is being used as a log, new material is always added at the end.

```
// Function AppendToLog(nb, str, stampDateTime)
// Appends the string to the named notebook.
// If stampDateTime is nonzero, appends date/time before the string.
Function AppendToLog(nb, str, stampDateTime)
   String nb               // name of the notebook to log to
   String str              // the string to log
   Variable stampDateTime  // nonzero if we want to include stamp

   Variable now
   String stamp

   Notebook $nb selection={endOfFile, endOfFile}
   if (stampDateTime)
      now = datetime
      stamp = Secs2Date(now,0) + ", " + Secs2Time(now,0) + "\r"
      Notebook $nb text=stamp
   endif
   Notebook $nb text= str+"\r"
End
```

You can test this function with the following commands:

```
NewNotebook/F=1/N=Log1 as "A Test"
AppendToLog("Log1", "Test #1\r", 1)
AppendToLog("Log1", "Test #2\r", 1)
```

The **sprintf** operation (see page V-590) is useful for generating the string to be logged.

Inserting Graphics

There are two kinds of graphics that you can insert into a notebook under control of a procedure:

- A picture generated from a graph, table or layout (an "Igor-object" picture).
- A copy of a named picture stored in the current experiment's picture collection.

The command

```
Notebook Notebook0 picture={Graph0(0,0,360,144), -1, 0}
```

creates a new picture of the named graph and inserts it into the notebook. The numeric parameters allow you to control the size of the picture, the type of picture and whether the picture is black and white or color. This creates an anonymous (unnamed) picture. It has no name and does not appear in the Pictures dialog. However, it is an Igor-object picture with embedded information that allows Igor to recognize that it was generated from Graph0 (the embedded information feature is not implemented on *Windows*).

The command

```
Notebook Notebook0 picture={PICT_0, 1, 0}
```

makes a *copy* of the named picture, PICT_0, stored in the experiment's picture collection, and inserts the copy into the notebook as an anonymous picture. The inserted anonymous picture is no longer associated with the named picture from which it sprang.

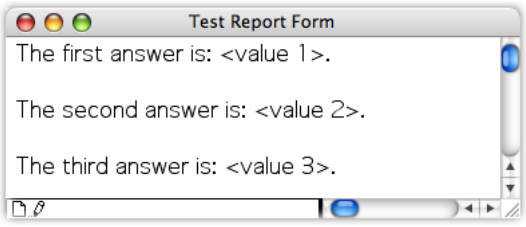
See **Pictures** on page III-421 for more information on pictures.

Updating a Report Form

In this example, we assume that we have a notebook that contains a form with specific values to be filled in. These could be the results of a curve fit, for example. This procedure opens the notebook, fills in the values, prints the notebook and then kills it.

```
// DoReport(value1, value2, value3)
// Opens a notebook file with the name "Test Report Form",
// searches for and replaces "<value 1>", "<value 2>" and "<value3>".
// Then prints the notebook and kills it.
// "<value 1>", "<value 2>" and "<value 3>" must appear in the form
// notebook, in that order.
// This procedure assumes that the file is in the Igor folder.
Function DoReport(value1, value2, value3)
   String value1, value2, value3

   OpenNotebook/P=Igor/N=trf "Test Report Form"
   Notebook trf, findText={"<value 1>", 1}, text=value1
   Notebook trf, findText={"<value 2>", 1}, text=value2
   Notebook trf, findText={"<value 3>", 1}, text=value3

   PrintNotebook/S=0 trf
   DoWindow/K trf
End
```

To try this function, enter it in the Procedure window. Then create a notebook that contains “<value 1>”, “<value 2>” and “<value 3>” and save it in the Igor folder using the file name “Test Report Form”. The notebook should look like this:



Now the notebook and then, execute the following command:

```
DoReport("123", "456", "789")
```

This will print the form using the specified values.

Updating Igor-Object Pictures

This feature is not implemented on Windows.

The following command will update all pictures in the notebook made from Igor graphs, tables or layouts from the current experiment.

```
Notebook Notebook0 specialUpdate=0
```

More precisely, it will update all dynamic special characters, including date and time characters as well as Igor-object pictures.

This next fragment shows how to update just one particular Igor-object picture.

```
String nb = "Notebook0"
Notebook $nb selection={startOfFile, startOfFile}
Notebook $nb findPicture={"Graph0", 1}
if (V_Flag)
   Notebook $nb specialUpdate=1
else
   Beep              // can't find Graph0
endif
```

Igor will normally refuse to update special characters unless updating is enabled, via the Enable Updating dialog (Notebook menu). You can override this and force Igor to do the update by using 3 instead of 1 for the specialUpdate parameter.

Retrieving Text

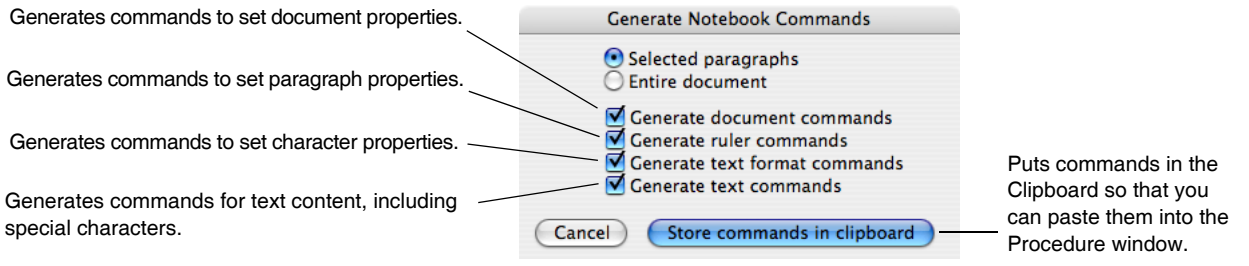
Since you can retrieve text from a notebook, it is possible to use a notebook as an input mechanism for a procedure. To illustrate this, here is a procedure that tags each point of a wave in the top graph with a string read from the specified notebook. The do-loop in this example shows how to pick out each paragraph from the start to the end of the notebook.

```
#pragma rtGlobals=1  // Make V_Flag and S_Selection be local variables.

// TagPointsFromNotebook(nb, wave)
// nb is the name of an open notebook.
// wave is the name of a wave in the top graph.
// TagPointsFromNotebook reads each line of the notebook and uses it
// to tag the corresponding point of the wave.
Function TagPointsFromNotebook(nb, wave)
   String nb         // name of notebook
   String wave       // name of the wave to tag

   String name       // name of current tag
   String text       // text for current tag
   Variable p

   p = 0
   do
      // move to current paragraph
      Notebook $nb selection={(p, 0), (p, 0)}
      if (V_Flag)          // no more lines in file?
         break
      endif

      // select all characters in paragraph up to trailing CR
      Notebook $nb selection={startOfParagraph, endOfChars}

      GetSelection notebook, $nb, 2    // Get the selected text
      text = S_Selection      // S_Selection is set by GetSelection
      if (strlen(text) > 0)   // skip if this line is empty
         name = "tag" + num2istr(p)
         Tag/C/N=$name/F=0/L=0/X=0/Y=8 $wave, pnt2x($wave, p), text
      endif

      p += 1
   while (p < numpnts($wave)) // break if we hit the end of the wave
End
```

For examples using notebook action special characters, see the Notebook Actions Demo example experiment.

Generate Notebook Commands Dialog

The Generate Notebook Commands dialog automatically generates the commands required to reproduce a notebook or a section of a notebook. This is intended to make programming a notebook easier. To use it, start by manually creating the notebook that you want to later create automatically from an Igor procedure. Then choose Generate Commands from the Notebook menu.



After clicking Store commands in Clipboard, open the procedure window and paste the commands into a procedure.

For a very simple formatted notebook, the commands generated look like this:

```
String nb = "Notebook2"
NewNotebook/N=$nb/F=1/V=1/W=(5,40,563,359)
Notebook $nb defaultTab=36,statusWidth=222,pageMargins={54,54,54,54}
Notebook $nb showRuler=0,rulerUnits=1,updating={1,60}
Notebook $nb newRuler=Normal,justification=0,margins={0,0,504}
Notebook $nb spacing={0,0,0},tabs={}
Notebook $nb rulerDefaults={"Helvetica",10,0,(0,0,0)}
Notebook $nb ruler=Normal,text="This is a test."
```

To make it easier for you to modify the commands, Igor uses the string variable nb instead of repeating the literal name of the notebook in each command.

If the notebook contains an Igor-object picture, you will see a command that looks like

```
Notebook $nb picture={Graph0(0,0,360,144), 0, 1}
```

However, if the notebook contains a picture that is not associated with an Igor object, you will see a command that looks like

```
Notebook $nb picture={putGraphicNameHere, 1, 0}
```

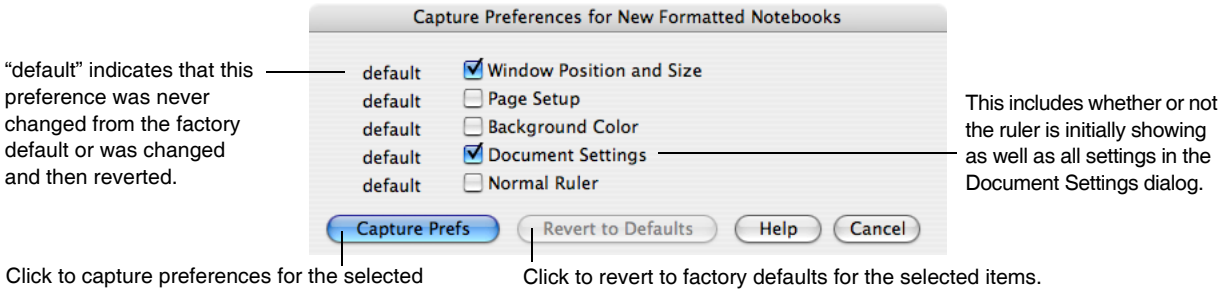
You will need to replace "putGraphicNameHere" with the name of a picture. Use the Pictures dialog, via the Misc menu, to see what named pictures are stored in the current experiment or to add a named picture. See **Pictures** on page III-421 for more information.

There is a shortcut that generates commands without going through the dialog. Select some text in the notebook, press Option (*Macintosh*) or Alt (*Windows*) and choose Copy from the Edit menu. This generates commands for the selected text and text formats. Press the Shift-Option (*Macintosh*) or Shift+Alt (*Windows*) to also generate document and ruler commands.

Notebook Preferences

The notebook preferences affect the creation of *new* notebooks. There is one set of preferences for plain notebooks and another set of preferences for formatted notebooks.

To set notebook preferences, set the attributes of any notebook of the desired type (plain or formatted) and then use the Capture Notebook Prefs item in the Notebook menu.



To determine what the preference settings are you must create a new notebook and examine its settings.

Notebook windows each have their own Page Setup values. New notebook windows will have their own copy of the captured (or reverted) Page Setup values.

Preferences are stored in the Igor Preferences file. See Chapter III-17, **Preferences**, for further information on preferences.

Notebook Template Files

A template notebook provides a way to customize the initial contents of a new notebook. When you open a template notebook, Igor opens it normally but leaves it untitled and disassociates it from the template notebook file. This leaves you with a new notebook based on your prototype. When you save the untitled notebook, Igor creates a new notebook file.

Template notebooks have ".ift" as the file name extension instead of ".ifn".

To make a template notebook, start by creating a prototype formatted text notebook with whatever contents you would like in a new notebook.

On Macintosh, choose File→Save Notebook As, check the Save as Stationery checkbox, and save the template notebook.

On Windows, choose File→Save Notebook As, choose IGOR Formatted Notebook Template from the "Save as type" menu, and save the template notebook.

You can convert an existing formatted text notebook file into a template file by changing the extension from ".ifn" to ".ift".

The Macintosh Finder's file info window has a Stationery Pad checkbox. Checking it turns a file into a stationery pad. When you double-click a stationery pad file, Mac OS X creates a copy of the file and opens the copy. For most uses, the template technique is more convenient.

Notebook Shortcuts

To view text window keyboard navigation shortcuts, see **Text Window Navigation** on page II-68.

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To get a contextual menu of commonly-used actions	Press Control and click in the body of the notebook window.	Right-click the body of the notebook window.
To execute commands in a notebook window	Select the commands or click in the line containing the commands and press Control-Return or Control-Enter.	Select the commands or click in the line containing the commands and press Ctrl+Enter.
To display the Find dialog	Press Command-F.	Press Ctrl+F.
To find the same text again	Press Command-G.	Press Ctrl+G.
To find again but in the reverse direction	Press Command-Shift-G.	Press Ctrl+Shift+G.
To find selected text	Press Command-Control-H.	Press Ctrl+H.
To find the selected text but in the reverse direction	Command-Control-Shift-H.	Press Ctrl+Shift+H.
To select a word	Double-click.	Double-click.
To select an entire line	Triple-click.	Triple-click.
To change a named ruler without using the Redefine Ruler dialog	Press Command while adjusting the icons in the ruler.	Press Ctrl while adjusting the icons in the ruler.
To find the next occurrence of a ruler	Press Option while selecting a ruler from the pop-up menu.	Press Alt while selecting a ruler from the pop-up menu.
To find the previous occurrence of a ruler	Press Shift-Option while selecting a ruler from the pop-up menu.	Press Shift+Alt while selecting a ruler from the pop-up menu.
To get miscellaneous information on notebook	Click the document icon in the bottom-left corner of the window.	Click the document icon in the bottom-left corner of the window.
To generate Notebook commands that will recreate the selected text	Press Option and choose Copy from the Edit menu. This puts the commands in the Clipboard for text and text formats.	Press Alt and choose Copy from the Edit menu. This puts the commands in the Clipboard for text and text formats.
	Press Option-Shift while copying to also generate document and ruler commands.	Press Alt+Shift while copying to also generate document and ruler commands.
To nudge a picture by one point up or down	Select the picture and press Control-Up Arrow or Control-Down Arrow.	Select the picture and press Ctrl+Alt+Up Arrow or Ctrl+Alt+Down Arrow.

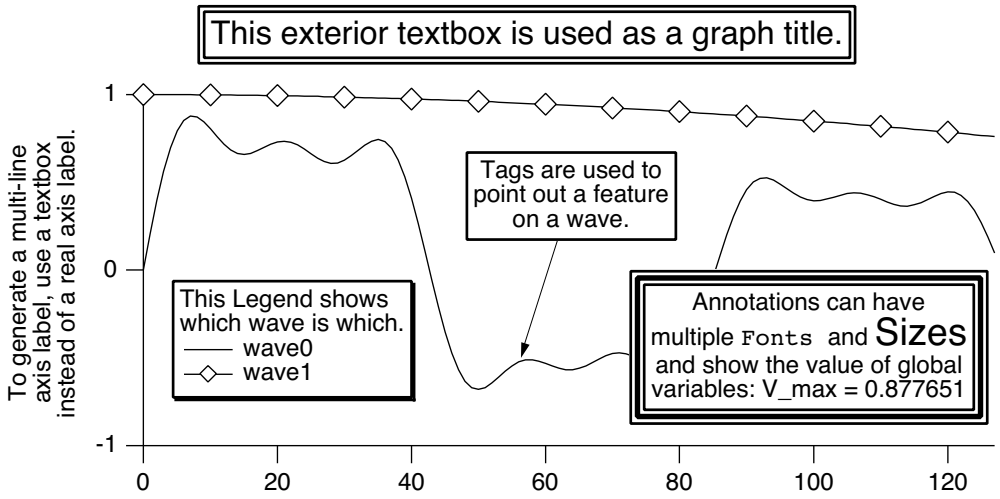
Chapter
III-2

Annotations

Overview

Annotations are custom objects that add information to a graph or a page layout. Most annotations contain text that you might use to describe the contents of a graph, point out a feature of a wave, identify the axis that applies to a wave, or create a legend. Igor automatically creates annotations for labeling contour plots. An annotation can also contain color scales showing the data range associated with colors in contour and image plots.

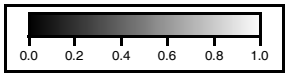
There are four types of annotation: textboxes, legends, color scales, and tags.



A textbox contains one or more lines of text which optionally may be surrounded by a frame, rotated, colorized and aligned.

A legend is similar to a textbox except that it contains wave symbols for one or more waves in a graph. Legends are automatically updated when waves are added to or removed from the graph, or when a wave's appearance is modified.

A tag is also similar to a textbox except that it is attached to a point in a wave and can contain dynamically updated text describing that point. Tags can be added to a graph, but not to a page layout. In contour plots, Igor automatically generates tags to label the contour lines.



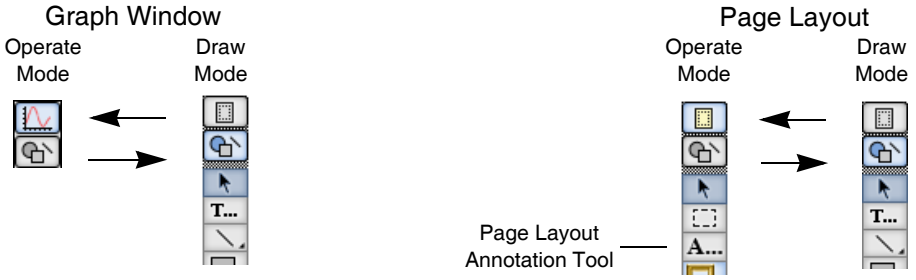
A color scale is similar to a legend except that it contains a color bar with an axis that spans the range of colors associated with the data. Color scales are automatically updated when the associated image plot, contour plot, f(z) trace, or color index wave is modified. A color scale can also be completely disassociated from any data by directly specifying a named color table and an explicit numeric range for the axis.

Annotations Quick Start

To Do This	Do This
To add an annotation to a graph	Choose Add Annotation from the Graph menu.
To add an annotation to a page layout	Choose Add Annotation from the Layout menu or click with the annotation ("A") tool. For more information about annotations in page layouts, see **Annotations in the Layout Layer** on page II-381.
To modify an annotation in a graph	Double-click the annotation to bring up the Modify Annotation dialog.
To modify an annotation in a page layout	Single-click the annotation with the annotation tool. This brings up the Modify Annotation dialog.

To Do This	Do This
To change the annotation type	Use the Annotation pop-up menu in the Modify Annotation dialog, or use the proper Tag, TextBox, ColorScale, or Legend operation.
To move an existing annotation	Click in the annotation and drag it to the new position. If the annotation is frozen, this won't work — double-click it and make it moveable in the Annotation Position tab. To change a tag's attachment point, press Option (*Macintosh*) or Alt (*Windows*) and drag the tag text to the new attachment point on the wave. This works whether or not the tag is frozen.
To duplicate an existing annotation	Double-click the annotation, then click the Duplicate Textbox button in upper-right corner of the Modify Annotation dialog. If the annotation is a tag, the button is titled Duplicate Tag, etc.
To delete an annotation	Double-click the annotation, then click the Delete button in the Modify Annotation dialog. A tag can be deleted by dragging its attachment point off the graph.
To show the value of a global variable in an annotation	Type "\{variableName}" in the annotation text. See also **Other Dynamic Escape Codes** on page III-46.

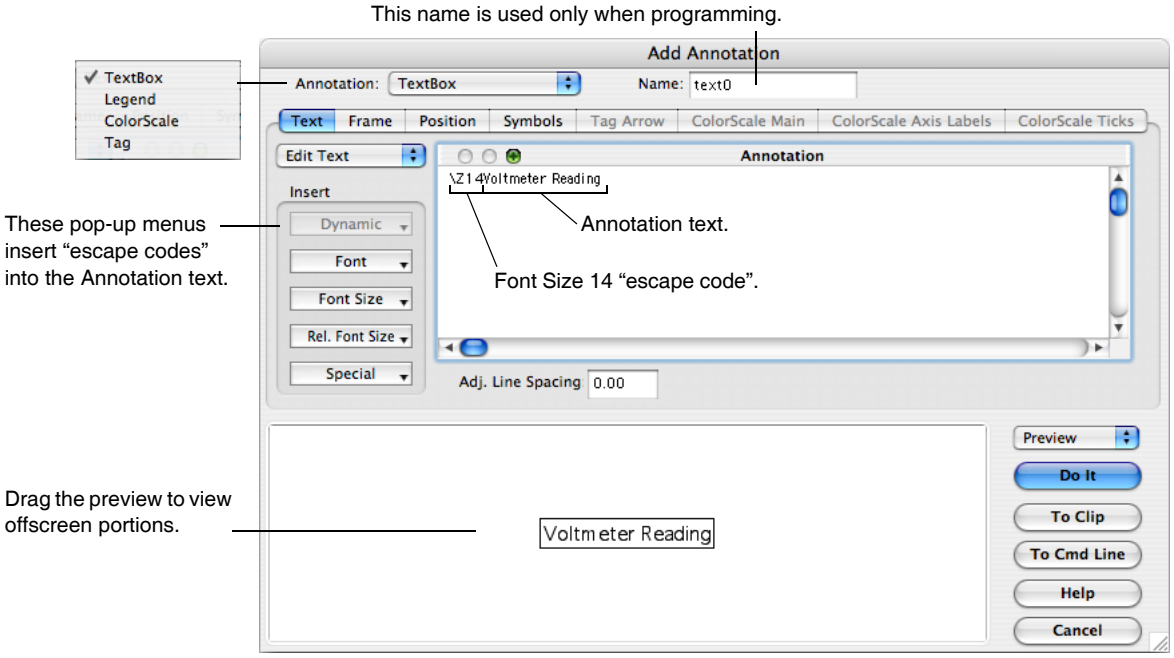
When manipulating annotations with the mouse, be sure that the graph or page layout are in the "operate" mode; *not* the "drawing" mode. The Tool bar indicates which mode the window is in:



The Annotation Dialog

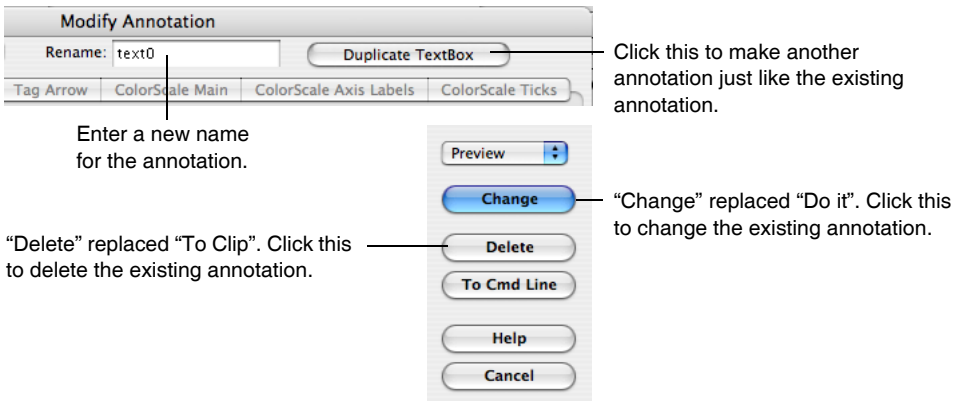
You can use the annotation dialog to create new or modify existing annotations. The dialog title is Add Annotation when a new annotation will be created and Modify Annotation when an existing annotation will be changed or duplicated. See **Modifying Annotations** on page III-43. The annotation dialog seems complex but comprises only a few major functions:

- A pop-up menu to choose the annotation type.
- A Name: setting.
- Tabs that group related Annotations settings. Some of the tabs apply to color scales only, and some of the settings in various tabs apply only to specific types of annotations.
- A Preview box to show what the annotation will look like or to display commands.
- The normal Igor dialog buttons.



Modifying Annotations

If an annotation is already in a graph you can modify it by double-clicking it while the graph is in the "operate" mode (see **Annotations Quick Start** on page III-41). The resulting Modify Annotation dialog is similar to the Add Annotation dialog except for a few items:



Single-clicking an annotation with the annotation ("**A**") tool in a page layout also brings up the Modify Annotation dialog.

Text Content

You enter text into the Annotation text entry area in the Text tab. This "windoid" supports copy, cut, paste and undo using Command-C, Command-X, Command-V and Command-Z (*Macintosh*) or Ctrl+C, Ctrl+X, Ctrl+V and Ctrl+Z (*Windows*). Tab stops are provided; see **Tabs** on page III-48. If your annotation has a lot of text, you can scroll and zoom this area for easier editing.

The annotation text may contain both plain text and "escape code" text which produces special effects such as superscript, font, font size and style, alignment, text color and so on. The text can contain multiple lines; just press Return. At any point when entering plain text you can choose a special effect from a pop-up menu within the Insert group, and Igor will enter the correct escape code. Igor wizards can type them in directly.

You can enter numbers into the text by simply typing them, or by referencing global variables or functions using **dynamic text**. Dynamic text is explained in **Other Dynamic Escape Codes** on page III-46.

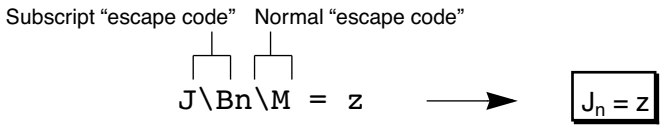
As you type annotation text, the Preview box shows what the resulting annotation will look like. You can not enter text in the Preview box.

Sometimes the textbox you are creating will not fit in the preview box. You can move the previewed annotation around to see the hidden parts. When you position the cursor over the preview box, it changes to a hand; just drag the annotation.

If the preview area isn't showing the annotation, change the pop-up menu (just above the Do It or Change button) from Commands to Preview.

About Text Escape Codes

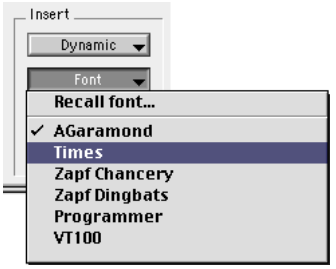
An escape code consists of a backslash character followed by one or more characters. It represents the special effect you selected. The effects of the escape code persist until overridden by a following escape code. The escape codes are cryptic but you can see their effects in the Preview box.



In the adjacent example, the subscript escape code "\B" begins a subscript and is not displayed in the annotation; the "n" that follows is plain text displayed as a subscript. The normal escape code "\M" overrides the subscript mode so that the plain text "= z" that follows has the original size and Y position (vertical offset) used for the "J".

Font Escape Codes

Choosing an item from the Font pop-up menu inserts a code that changes the font for subsequent characters in the annotation. The checked font is the font currently in effect at the current insertion point in the annotation text entry area.

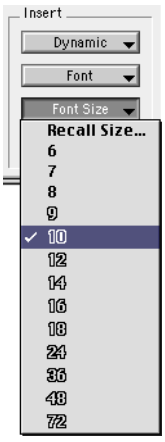


inserts `\F'Times'`

If you don't choose a font, Igor uses the default font or the graph font for annotations in graphs. You can set the default font using the Default Font item in the Misc menu, and the graph font using the Modify Graph item in the Graph menu. The Font pop-up menu also has a "Recall font" item. This item is used to make elaborate annotations as described under **Text Info Variables Escape Codes** on page III-64.

Font Size Escape Codes

Choosing an item from the Font Size pop-up menu inserts a code that changes the font size for subsequent characters in the annotation. The checked font size is the size currently in effect at the current insertion point in the annotation text entry area.



inserts `\Z10`

(To insert a size not shown, choose any shown size, and edit the escape code to contain the desired font size. Annotation font sizes may be 03 to 99 points; two digits are required after the "\Z" escape code.)

If you specify no font size escape code for annotations in graphs, Igor chooses a font size appropriate to the size of the graph unless you've specified a graph font size in the Modify Graph dialog. The default font size for annotations in page layouts is 10 points. The Font Size pop-up menu contains a "Recall size" item which is used to make elaborate annotations as described in the section **About Text Info Variables** on page III-64.

Relative Font Size Escape Codes

Choosing an item from the Rel. Font Size pop-up menu inserts a code that changes the relative font size for subsequent characters in the annotation. Use values larger than 100 to increase the font size, and values smaller than 100 to decrease the font size.

To insert a size not shown, choose any shown relative size, and edit the escape code to contain the desired relative font size. Annotation relative font sizes may be 001 to 999 (1% to 999%). Three digits are required after the "\Zr" escape code.

Don't use, say, 50% followed by 200% and expect to get exactly the original font size back; rounding inaccuracies will prevent success (because font sizes are handled as only integers). For example, if you start with 15 point text and use `\Zr050` (50%) the result is 7 point text. 200% of 7 points is only 14 point text:

Voltmeter Readings
(past 30 days)
Voltmeter Readings

```
\JC\Z15Voltmeter Readings
\Zr050(past 30 days)
\Zr200Voltmeter Readings
```

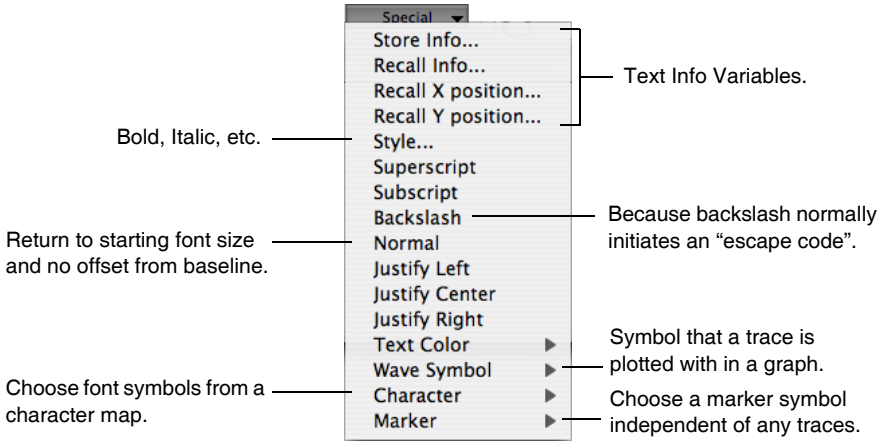
Instead, use the Normal "\M" escape code (or an absolute font size or a recalled font size) to return to a known font size. In the following example, the initial 15 point font size is saved in the main text info variable (the "\[0" escape code, see **Text Info Variables Escape Codes** on page III-64), whose size is recalled by the Normal escape code.

Voltmeter Readings
(past 30 days)
Voltmeter Readings

```
\JC\Z15\[0Voltmeter Readings
\Zr050(past 30 days)
\MVoltmeter Readings
```

Special Escape Codes

Choosing an item from the Special pop-up menu inserts an escape code that makes subsequent characters superscript, subscript or normal, affects the style, position or color of subsequent text, or inserts the symbol with which a wave is plotted in a graph.



The first four items, Store Info, Recall Info, "Recall X position", and "Recall Y position" are used to make elaborate annotations as described under **Elaborate Annotations and Axis Labels** on page III-63.

The Style item brings up a subdialog that you use to change the style (bold, italic, etc.) for the annotation at the current insertion point in the annotation text entry area. This subdialog has a Recall Style checkbox that is used in elaborate annotations with text info variables.

The Superscript and Subscript items insert an escape code that makes subsequent characters superscript or subscript. Use the Normal item to return the text to the original text size and Y position.

The Backslash item inserts a code to insert a backslash that prints, rather than one which introduces an escape code. Igor does this by inserting two backslashes, which is an escape code that prints a backslash. Weird, huh?

The Normal item inserts a code to return to the original font size and baseline (which has no vertical offset such as is used to produce super- and subscripts). More precisely, Normal sets the font size and baseline to the values stored in text variable 0 (see **About Text Info Variables** on page III-64). The font and style are not affected.

The Justify items insert a code to align the current and following lines.

The Color item inserts a code to color the following text. The initial text color and the annotation *background* color are set in the Frame Tab.

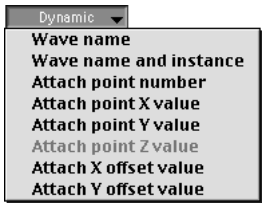
The Wave Symbol item inserts a code that prints the symbol (line, marker, etc.) used to display the wave trace in the graph. This code is inserted automatically in a legend. You can use this menu item to manually insert a symbol into a tag, textbox, or color scale. For graph annotations, the submenu lists all the trace name instances in the top graph. For layout annotations, all the trace name instances in all graphs in the layout are listed.

The Character item presents a table from which you can select text and special characters to add to the annotation.

The Marker item inserts a code to draw a marker symbol. These symbols are independent of any traces in the graph.

Dynamic Escape Codes for Tags

The Dynamic pop-up menu inserts escape codes that apply only to tags. These codes insert information about the wave or point in the wave to which the tag is attached. This information automatically updates whenever the wave or the attachment point changes.



Dynamic Item	Effect
Wave name	Displays the name of the wave to which the tag is attached.
Trace name and instance	Same as wave name but appends an instance number (e.g., #1) if there is more than one trace in the graph associated with a given wave name.
Attach point number	Displays the number of the tag attachment point.
Attach point X value	Displays the X value of the tag attachment point.
Attach point Y value	Displays the Y value of the tag attachment point.
Attach point Z value	Displays the Z value of the tag attachment point. Available only for contour traces, waterfall plots, or image plots.
Attach X offset value	Displays the trace's X offset.
Attach Y offset value	Displays the trace's Y offset.

See also **TagVal and TagWaveRef Functions** on page III-47. These functions provide the same information as the Dynamic pop-up menu items but with greater flexibility.

Other Dynamic Escape Codes

You can type the "dynamic text escape code" which inserts dynamically evaluated text into any kind of annotation using the escape code sequence:

```
\{dynText}
```

where *dynText* may contain numeric and string expressions. If *dynText* references a numeric variable, string variable or wave "object", this makes the annotation dependent on the referenced object (see Chapter IV-9, **Dependencies**, for further details). If the object changes, Igor automatically updates the dynamic text.

Note: Making an annotation dependent on a variable or wave is often a bad idea because you might inadvertently forget to create the variable or wave, delete it, or change its value. Thus, you should use this feature only when other techniques are insufficient. In most cases, it is better to generate text programmatically, as described in **Generating Text Programmatically** on page III-66.

The numeric and string expressions are evaluated in the context of the "root" data folder. If you are not using data folders, just use the names of the waves and variables. If you are using data folders, use the full data folder path of any nonroot objects in the expressions. For more on Data Folders, see Chapter II-8, **Data Folders**.

dynText can take two forms: an easy form for a single numeric expression, and a more complex form that provides precise control over the formatting of the result.

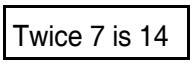
The easy form is:

```
\{numeric-expression}
```

This evaluates the numeric expression and prints with generic ("%g") formatting. For example:

```
Twice \{K0} is \{K0*2}
```

creates this textbox when K0 is 7:



If K0 changes, Igor automatically updates the textbox.

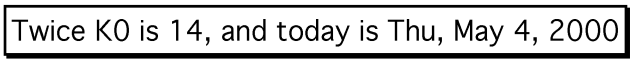
The full form for *dynText* is:

```
\{formatStr, list-of-numeric-or-string-expressions}
```

formatStr and *list-of-numeric-or-string-expressions* are treated the same as for the Printf operation. For instance, this example has a format string, a numeric expression and a string expression:

```
\{"Twice K0 is %g, and today is %s", 2*K0, date()}
```

It produces this result:



Don't try to use any annotation escape codes in the format string or numeric or string expressions; they don't work within the \{ … } context.

Also, the format string and string expressions do not support multiline text. If you need to use multiline text, use the technique described in **Generating Text Programmatically** on page III-66.

As an aid in typing the expressions, Igor considers carriage returns within the braces to be equivalent to spaces. Thus you can type in the Add Annotation dialog:

```
\{
   "Twice K0 is %g, and today is %s",
   2*K0,
   date()
}
```

and get the same result as above. These carriage returns can be typed directly in the Add Annotations dialog, or be typed as "\r" in a macro, function, or the command line.

TagVal and TagWaveRef Functions

If the annotation is a tag, you can use the functions **TagVal** (page V-694) and **TagWaveRef** (page V-695) to display information about the data point to which the tag is attached. For example, the following displays the Y value of the tag's data point:

```
\{"%g", TagVal(2)}
```

This is identical in effect to the "\0Y" escape code which you can insert by choosing the "Attach point Y value" item from the Dynamic pop-up menu. The benefit of using the TagVal function is that you can use a formatting technique other than %g. For example:

```
\{"%5.2f",TagVal(2)}
```

TagVal is capable of returning all of the information that you can access via the Dynamic menu escape codes. Use it when you want to control the numeric format of the text.

The TagWaveRef function returns a reference to the wave to which the tag is attached. You can use this reference just as you would use the name of the wave itself. For example, given a graph displaying a wave named wave0 (in the root data folder), the following tag text displays the average value of the wave:

```
\{"%g",mean(wave0,-INF,INF)}
```

This is fine, but if you move the tag to another wave it will still show the average value of wave0. Using TagWaveRef, you can make this show the average value of whichever wave is tagged:

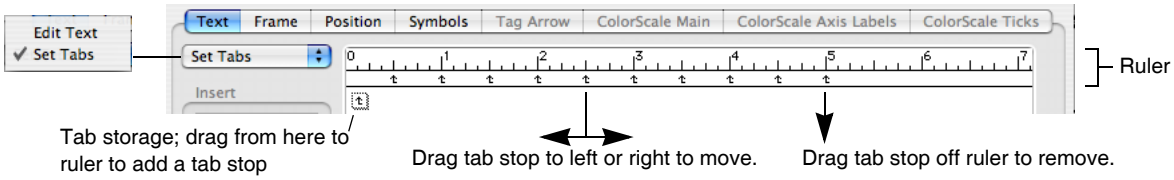
```
\{"%g",mean(TagWaveRef(),-INF,INF)}
```

The TagVal and TagWaveRef functions work only while Igor is in the process of evaluating the annotation text, so you should use them only in annotation dynamic text or in a function called from annotation dynamic text.

Also see the **TraceNameToWaveRef** function (page V-711), which returns a reference to a wave given a graph's trace name and can be freely used in macros and functions.

Tabs

The Text Tab's Annotation text area actually has two functions which are controlled by the pop-up menu at its top-left corner. If you choose Set Tabs from this pop-up menu, Igor shows the tab stops for the annotation.



By default, an annotation has 10 tab stops spaced 1/2 inch apart. You can change the tab stops by dragging them along the ruler. You can remove a tab stop by dragging it down off the ruler. You can add a tab by dragging it from the tab storage area at the left onto the ruler.

Igor supports a maximum of 10 tab stops per annotation and they are always left-aligned tabs. There is only one set of tab stops per annotation and they affect the entire annotation.

When setting the tabs, the Insert Text pop-up menus are disabled; return the pop-up menu to Edit Text to reenable them.

General Annotation Properties

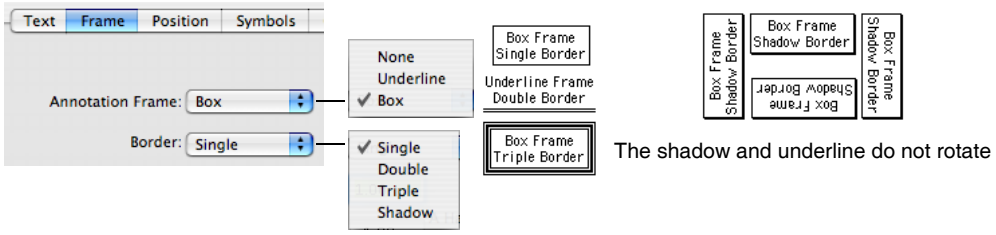
Most annotation properties are common to all kinds of annotations.

Name

You can assign a name to the annotation with the Name item. In the Modify Annotation dialog, this is the Rename item. The name is used to identify the annotation in a Tag, TextBox, ColorScale, or Legend operation. Annotation names must be unique to the window they are in. The names Igor automatically puts here already are unique, but you can change them if you want. See **Programming with Annotations** on page III-66 for more information.

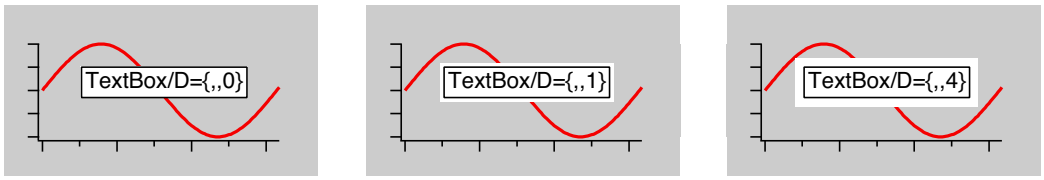
Frame

In the Frame Tab, the Frame and Border pop-up menus allow you to frame the annotation with a box or shadow box, to underline the textbox, or to have no frame at all. The line size of the frames and the shadow are set by the Thickness and Shadow values.



By default, framed annotations also have a 1-point "halo" that surrounds them to separate them from their surroundings. The halo takes on the color of the annotation's background color. You can change the width of this halo to a value between 0 and 10 points by setting the desired thickness in the Halo box in the Frame tab. A fractional value such as 0.5 is permitted.

Annotation Halo Examples



0 points | 1 point (Default) | 4 points

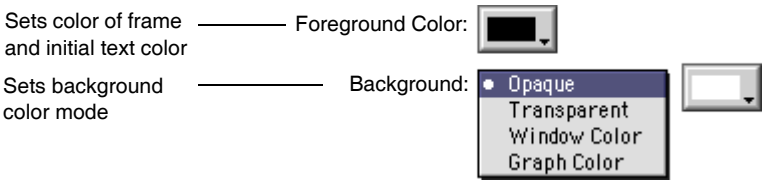
Specifying a negative value for Halo allows the halo thickness to be overridden by the global variable V_TBBufZone in the root data folder. If the variable doesn't exist, the absolute value of the entered value is used. The default halo value is -1. You can override the default halo by setting the V_TBBufZone global in a IgorStartOrNewHook hook function. See the example in **User-Defined Hook Functions** on page IV-251.

Color

The Frame tab contains most of the annotation's color settings.

Use the Foreground Color pop-up menu to set the *initial* text color. You can *change* the color of the text from the initial foreground color by inserting a color escape code using the Special pop-up menu in the Text tab.

Use the Background pop-up menu to set the background mode and color.



Background Color Mode	Effect
Opaque	The annotation background covers objects behind. You choose the background color from a pop-up menu.
Transparent	Objects behind the annotation show through.
Graph color	The background is opaque and is the same color as the graph background color. This is not available for annotations added to page layout windows.
Window color	The background is opaque and is the same color as the window background color.

Annotation Positioning

You can rotate the annotation into the four principal orientations with the in the Position tab's Rotation pop-up menu. You can also enter an arbitrary rotation angle (in integral degrees) directly. Tags attached to

contour traces and color scales have specialized rotation settings; see **Modifying Labels** on page II-336 and **ColorScale Size and Orientation** on page III-60.

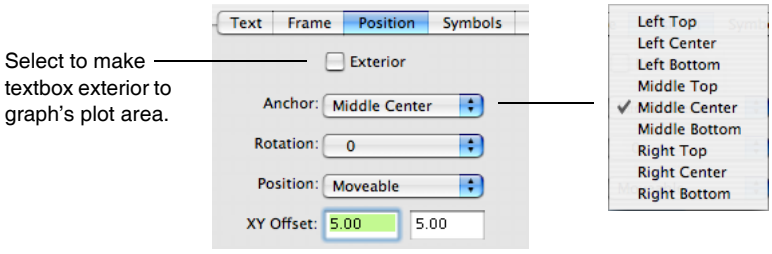
You can position an annotation anywhere in a window by dragging it and in many cases this is all you need to know. However, if you attend to a few extra details you can make the annotation go to the correct position even if you resize the window or print the window at a different size.

This is particularly important when a graph is placed into a page layout window, where the size of the placed graph usually differs from the size of the graph window.

Annotations are positioned using X and Y offsets from "anchor points". The meaning of these offsets and anchors depends on the type of annotation and whether the window is a graph or layout. Tags, for instance, are positioned with offsets expressed as a percentage of the horizontal and vertical sizes of the graph. See **Tag Positioning** on page III-58.

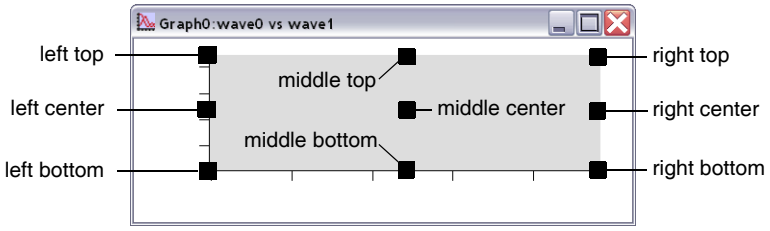
Textbox, Legend, and Color Scale Positioning in a Graph

A textbox, legend, and color scale are positioned identically, so this description will use "textbox" to refer to all of them. A textbox in a graph can be "interior" or "exterior" to the graph's plot area. You choose this positioning option with the Exterior checkbox:



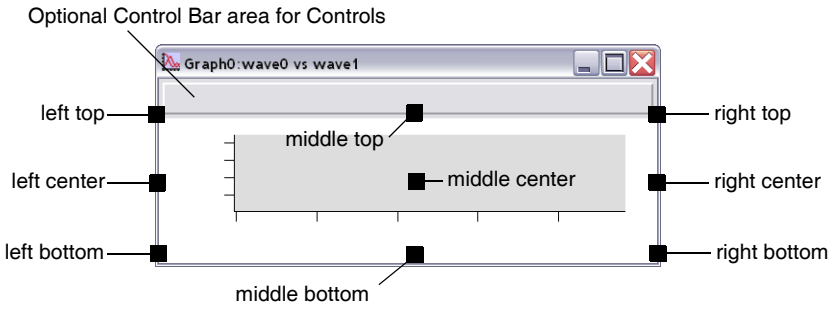
The Anchor pop-up menu specifies the precise location of the reference point on the plot area or graph window edges. It also specifies the location *on the textbox* which Igor considers to be the "position" of the textbox.

An interior textbox is positioned relative to a reference point on the edge of a graph's plot area. (The plot area is the central rectangle in a graph window where traces are plotted. The standard left, right, bottom, and top axes surround this rectangle.)



Anchor Points for Interior Textboxes

An exterior textbox is positioned relative to a reference point on the edge of the window and the textbox is normally outside the plot area.



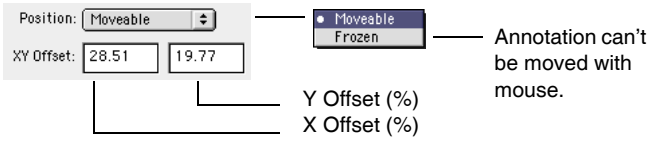
Anchor Points for Exterior Textboxes

The purpose of the exterior textbox is to allow you to place a textbox away from the plot area of the graph. For example, you may want it to be above the top axis of a graph or to the right of the right axis. Igor tries to keep exterior textboxes away from the graph by pushing the graph away from the textbox.

The direction in which it pushes the graph is determined by the textbox's anchor. If, for example, the textbox is anchored to the top then Igor pushes the graph down, away from the textbox. If the anchor is middle-center, Igor does not attempt to push the graph away from the textbox. So, an exterior textbox anchored to the middle-center behaves like an interior textbox.

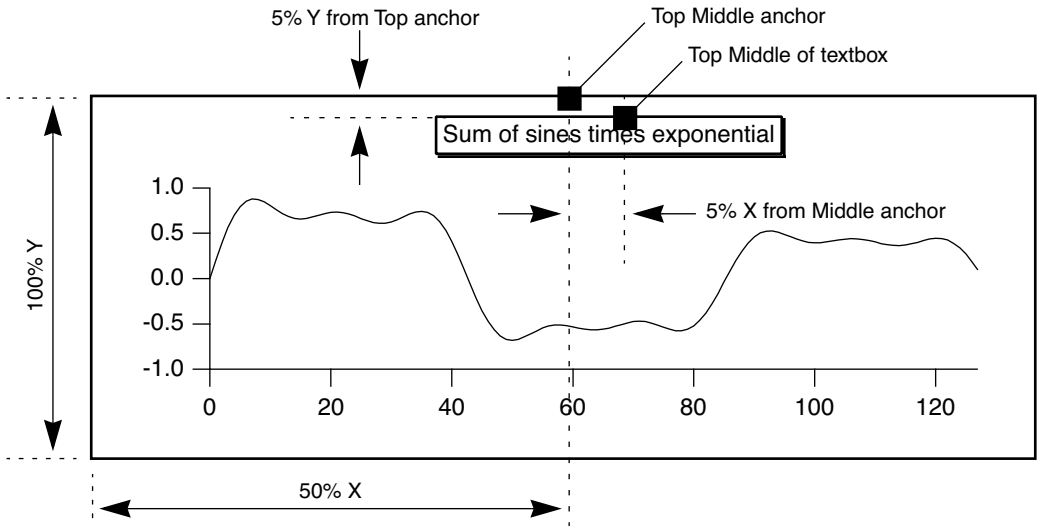
If you specify a margin, using the Modify Graph dialog, this overrides the effect of the exterior textbox, and the exterior textbox will not push the graph.

The XY Offset in the Position Tab gives the horizontal and vertical offset from the anchor to the textbox as a *percentage* of the horizontal and vertical sizes of the graph's plot area for interior textboxes or the window sizes for exterior textboxes.



The **Position** pop-up menu "freezes" the position of the textbox so that it can not be moved with the mouse. This is useful if you are using the textbox to label an axis tick mark and don't want to accidentally move it.

In the following example we wanted to center the textbox above and outside the plot area, so we chose an exterior textbox, a middle top anchor point and X and Y offsets of 5 (percent of the graph window's width and height). The choice of a top anchor pushed the graph below the textbox; a middle anchor does not push the graph left or right.

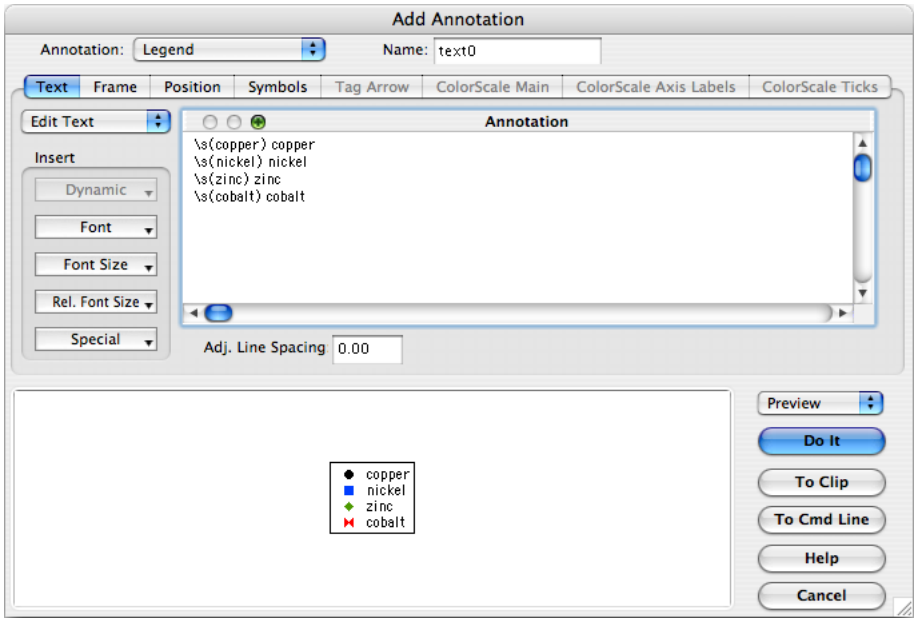


Textbox and Legend Positioning in a Page Layout

Annotations in a page layout window are positioned relative to an anchor point on the edge of the printable part of the page. The distance from the anchor point to the textbox is determined by the X and Y offsets which are in percent of the printable page. Annotations in a page layout can not be "frozen" as they can in a graph (see above).

Legends

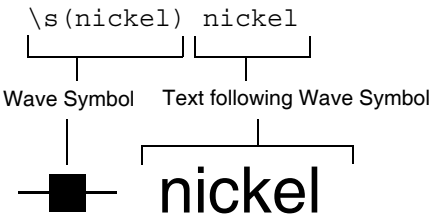
A legend is very similar to a textbox. It shows the "wave symbol" for some or all of the waves in a graph or page layout. (We also call this a "legend symbol".) To make a legend, choose Add Annotation from the Graph or Layout menu.



The pop-up menu at the top left of the dialog sets the type of the annotation (TextBox, Legend, ColorScale or Tag). If you choose Legend *when there is no text in the text entry area*, Igor automatically generates the text needed for a "standard legend". To keep the standard legend, just click Do It. However, you can also modify the legend text as you can for any type of annotation.

Legend Text

The legend text consists of an escape sequence to specify the wave whose symbol you want in the legend plus plain text. In this example dialog above, `\s(nickel)` is the escape sequence that inserts the wave symbol (a line and a filled square marker) for the wave whose name is nickel. This escape sequence is followed by a space and the name of the wave. The part after the escape sequences is plain text that you can edit as needed.



Instead of specifying the name of the trace for a legend symbol, you can specify the trace number. For example, "`\s(#0)`" displays the legend for trace number 0.

There are only two differences between a legend and a textbox. First, text for a legend is automatically generated when you choose Legend from the pop-up menu while there is no text in the text entry area. Second, if you append or remove a wave from the graph or rename a wave, the legend is automatically updated by adding or removing wave symbols. Neither of these two actions occur for a textbox (or a tag or color scale, for that matter).

Symbol Conditions at a Point

You can specify that a legend symbol shows the conditions at a specific point on a trace by appending the point number in brackets to the trace name. For example \s(nickel[3]). This feature is useful when a trace uses f(z) mode or when a single point on a trace has been customized. This feature was added in Igor Pro 6.20.

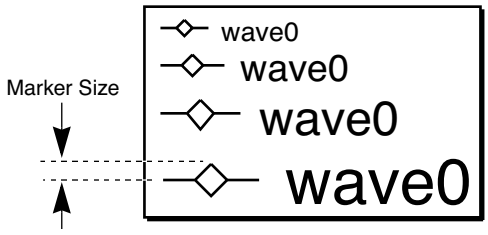
Freezing the Legend Text

Occasionally you may not want the legend to update automatically when you append waves to the graph. You can freeze the legend text by converting the annotation to a textbox. To create a nonupdating legend, bring up the Add Annotation dialog. Choose Legend from the pop-up menu to get Igor to create the legend text, then choose TextBox from the pop-up menu. Now you have converted the legend to a textbox so it will not be automatically updated.

Marker Size

A wave symbol will contain a marker if the wave is drawn with one. Normally the size of the marker drawn in the annotation is based on the font size in effect when the marker is drawn. When you set the font size before the wave symbol escape code, both the marker and following text will be adjusted in proportion:

```
\Z10\s(wave0) wave0
\Z14\s(wave0) wave0
\Z18\s(wave0) wave0
\Z24\s(wave0) wave0
```

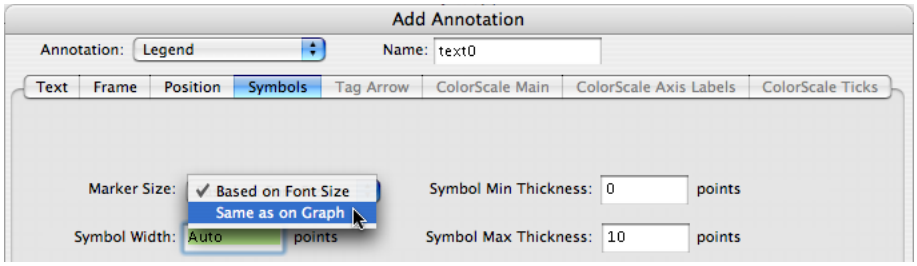


You can also change the font size after the wave symbol, which sets the size of the following text. Here is an example that uses a small marker size with large text.

```
\Z12\s(wave0) \Z24wave0
```

The \Z12 sets the font size to 12 points. This controls the size of the marker. The \Z24 sets the font size of the following text to 24 points.

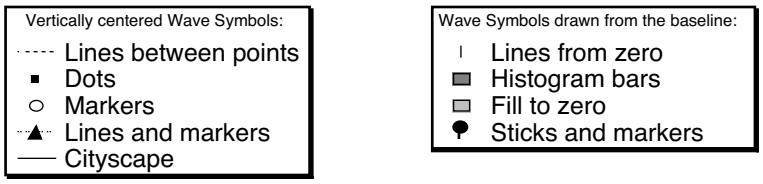
The second method for setting the size of a marker is to choose "Same as on Graph" from the Marker Size pop-up menu in the Symbols Tab.



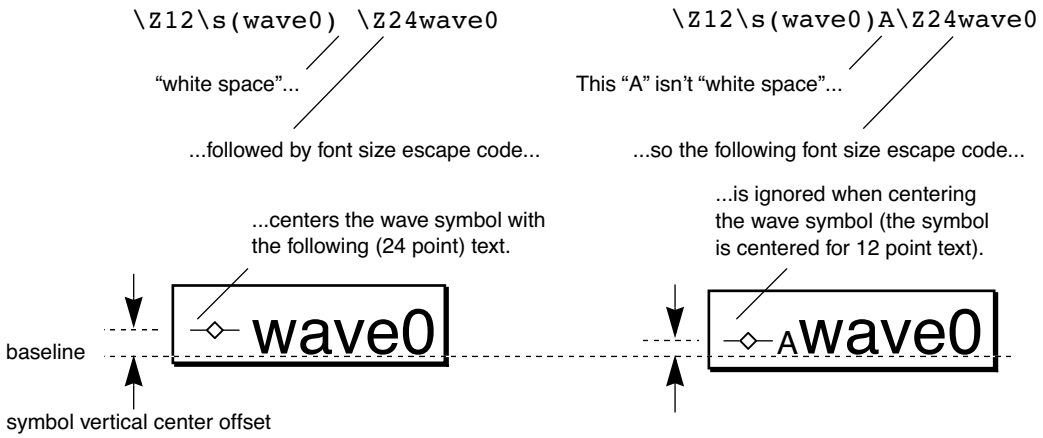
With "Same as on Graph" chosen, the marker size will match the size of the corresponding marker in the graph, regardless of the size of the annotation's text font.

Wave Symbol Centering

Some wave symbols are vertically centered relative to either the text that precedes or the text that follows the wave symbol escape code, and other symbols are drawn with their bottom at the baseline:



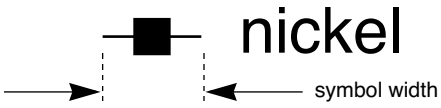
Vertically centered symbols are centered based on the height of the current font and size except that, if the symbol is followed by optional white space plus a font size escape code (i.e. `\Z09`, `\Z]n`, `\]n`, or `\M`), then the centering is based on the *following* font size. This automatically centers a symbol with following text that is much bigger or smaller than the symbol.



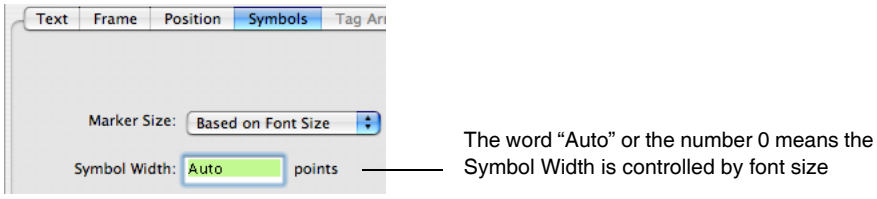
"White space" in this context is a run of space characters or tab characters. Note that on the Macintosh Option-Space is *not* considered to be "white space", and line breaks are not white space on any platform.

Wave Symbol Width

The wave symbol width is the width in which all wave symbols are drawn.



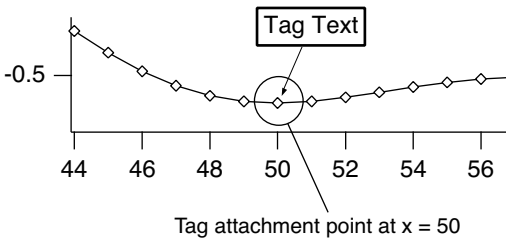
This width is controlled by the font size of the text preceding the wave symbol, or it is set explicitly to a given number of points using the Symbol Width value in the Symbols Tab.

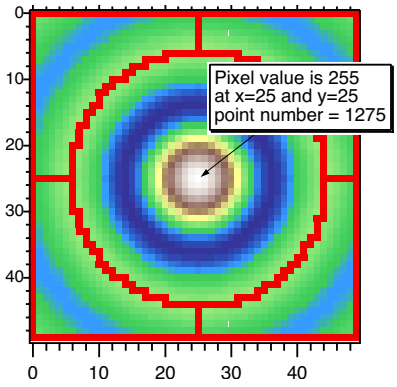


You can widen or narrow the overall symbol size by entering a nonzero width value. If you use large markers with small text, you may find it necessary to reduce the wave symbol width using this setting. For some line styles that have long dash/gap patterns, you will want to enter an explicit value large enough to show the pattern, such as 36 (1/2 inch) or 72 points. The maximum is 1000 points.

Tags

A tag is like a textbox but with several added capabilities. A tag is attached to a particular point on a particular trace, image, or waterfall plot in a graph:

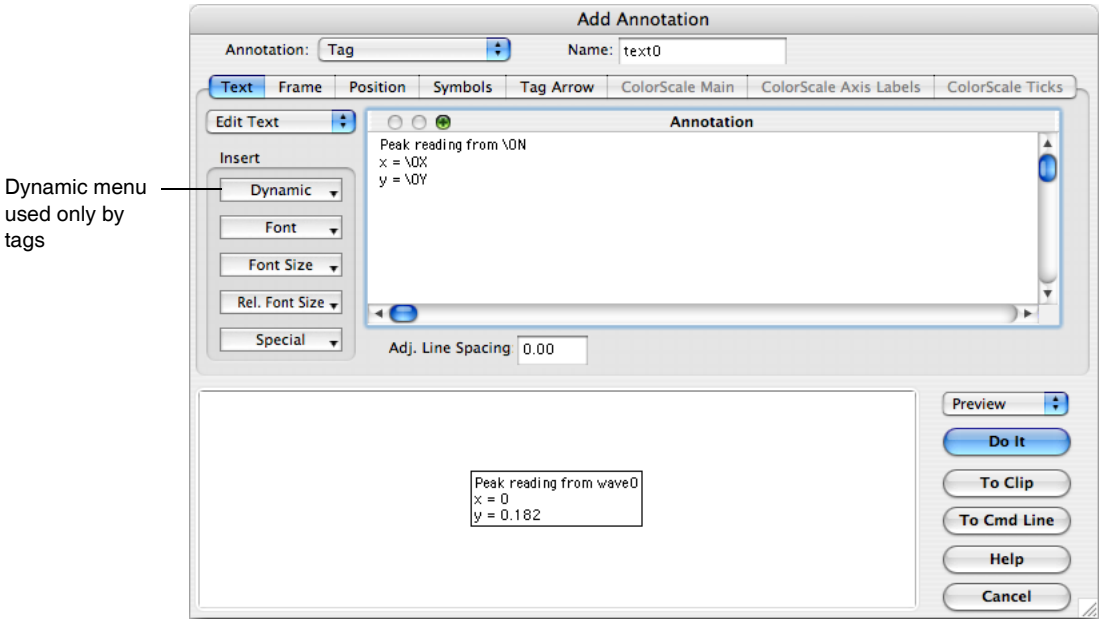




Tags can not be added to page layouts (a graph containing a tag can be added to the page layout, of course). Igor automatically generates tags to label contour plots.

Tag Text

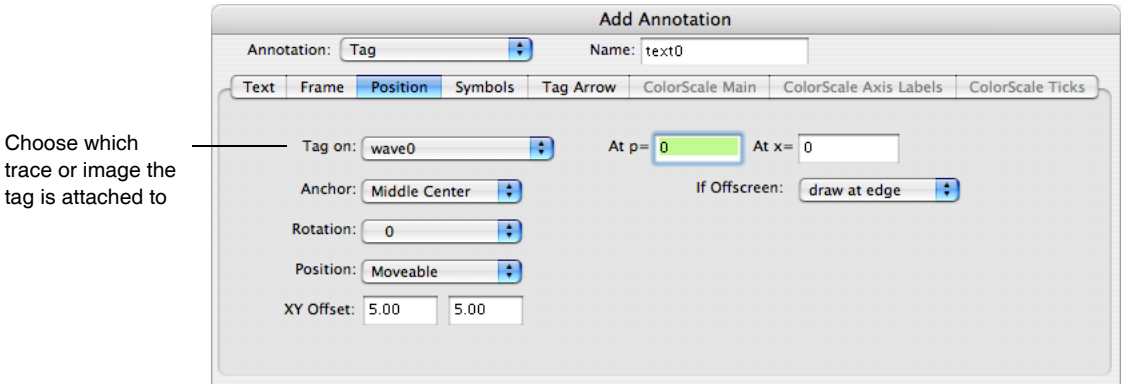
Text in a tag can contain anything a textbox or legend can handle, and more. The Dynamic pop-up menu of the Text Tab inserts escape codes that apply only to tags. These codes insert information about the wave the tag is attached to, or about the point in the wave to which the tag is attached. This information is "dynamically" updated whenever the wave or the attachment point changes. See **Dynamic Escape Codes for Tags** on page III-46.



The TagVal and TagWaveRef functions are also useful when creating a tag with dynamic text. See **TagVal and TagWaveRef Functions** on page III-47.

Tag Wave and Attachment Point

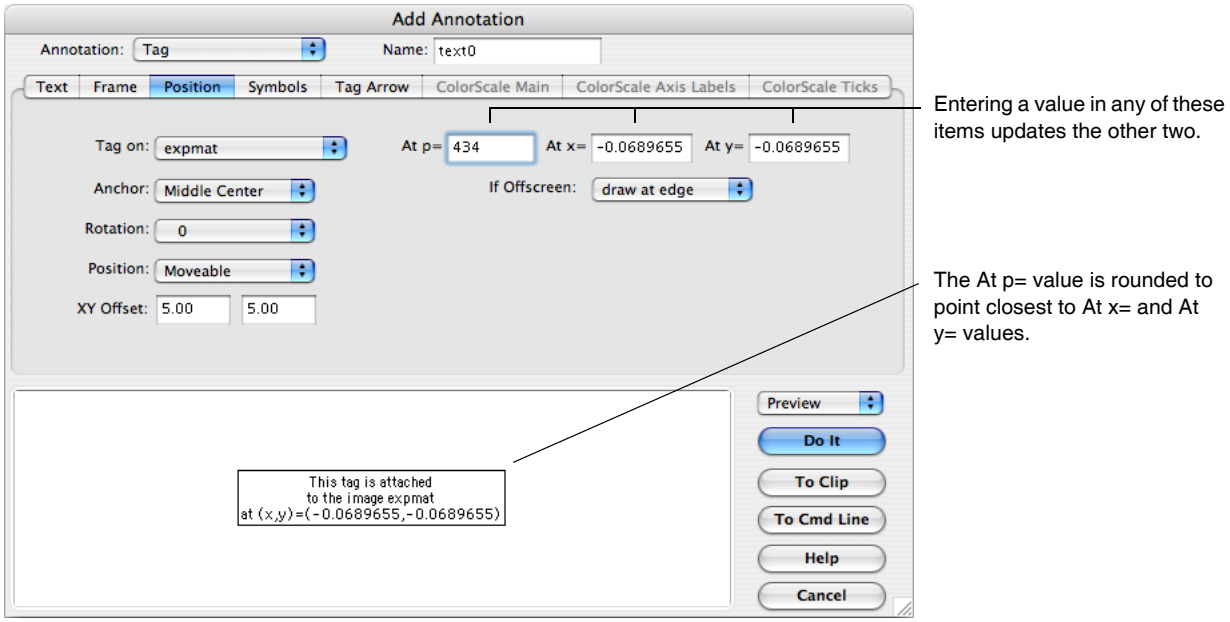
You specify which wave the tag is attached to in the Position Tab, by choosing a wave from the "Tag on" pop-up menu.

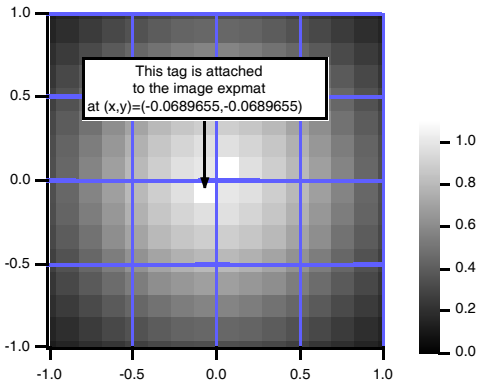


You specify which point the tag is attached to by entering the point number in the "At p=" entry or an X value in the "At x=" entry. The X value is in terms of the X scaling of the wave to which you are attaching the tag. This is not necessarily the same as the X axis position of the point if the wave is displayed in XY mode (versus a wave providing X values). This is the X value *of the wave's point* to which the tag is attached. If this distinction mystifies you, see **Waveform Model of Data** on page II-77.

The attachment point of a tag in a (2D) image or waterfall plot is treated a bit differently than for 1D waves. In images it is the sequential point number linearly indexed into the matrix array. The dialog will convert this point number displayed in the "At p=" entry into the X and Y values, and vice versa.

Because it is the point number that determines the actual attachment point, entered "At x=" and "At y=" values are not necessarily exactly where the tag is attached. In the following dialog the X and Y values were entered as 0, but the nearest point (434) results in an actual attachment point of (-0.0689655,-0.0689655):



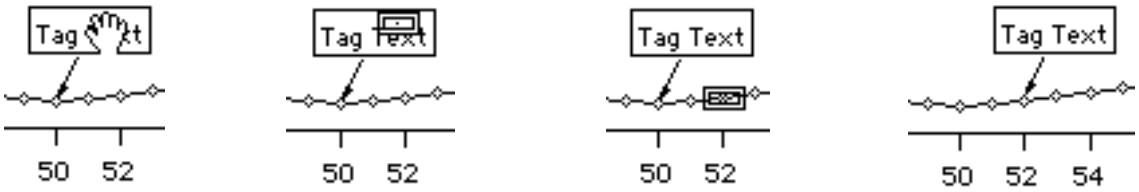


Notice that no point in the image is centered on (x,y) = (0,0).

Sometimes, however, it is just easier to manually position the tag by dragging it with the cursor, as described in the next section.

Changing a Tag's Attachment Point

Once a tag is on a graph you can attach it to a different point by pressing Option (*Macintosh*) or Alt (*Windows*), clicking in the tag, and dragging the special tag cursor you will see to the new attachment point of the wave. You must drag the tag cursor to the point *on the trace* to which you want to attach the tag, not to the position on the screen where you want the tag text to appear. The dot in the center of the tag cursor shows where the tag will be attached.

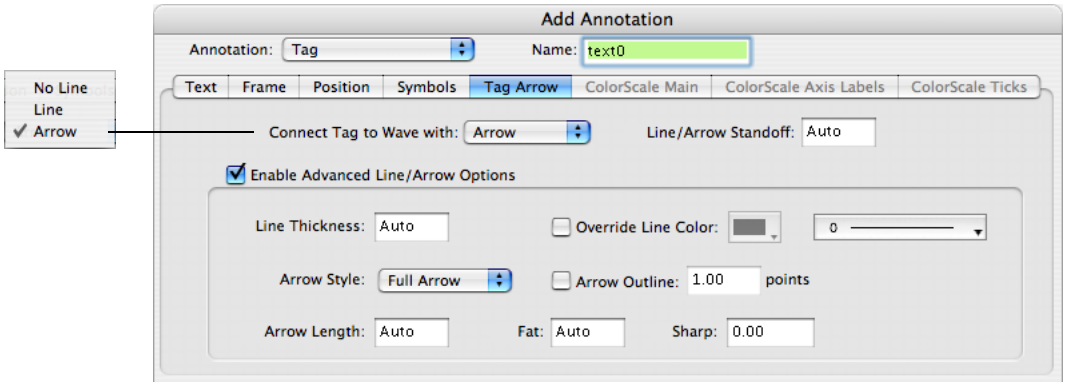


1. Move cursor over tag text.
2. Hold down Option or Alt.
3. Drag to new attachment point.
4. Tag attaches to and moves to the new point.

If you drag the tag cursor off the graph, the tag will be deleted from the graph.

Tag Arrows

You can indicate a tag's attachment point with an arrow or line drawn from the tag's anchor point using the "Connect Tag to Wave with" pop-up menu in the Tag Arrows tab.



You can adjust how close the arrow or line comes to the data point by setting the Line/Arrow Standoff distance (in points). In the Preview, the standoff is demonstrated by how close the arrow is drawn to the edge of the preview area

The Advanced Line/Arrow options are compatible with Igor 6.02 or later; they give you added control of the line and arrow characteristics.

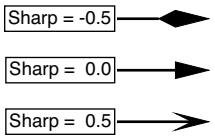
A Line Thickness value of 0 corresponds to the default (nonadvanced) line thickness of 0.5 points, otherwise, enter a value up to 10.0 points. To make the line disappear, select No Line from the "Connect Tag to Wave with" popup menu.

The line color is normally set by the annotation frame color (in the Frame tab). You can override this by checking the Override Line Color checkbox and choosing a color from the popup menu.

Change the attachment line's style from the default solid line using the line style popup menu.

If "Connect Tag to Wave with" popup is set to Arrow, you can alter the default appearance of the arrow head using the remaining controls: full or half arrow head (left or right), filled or outlined arrow head, and alter the arrow head's length from the default (Auto) to the given length in points.

The Sharp option is a small value between -1.0 and 1.0 (0 is the default).



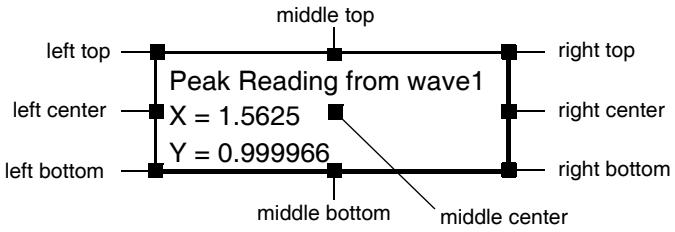
The Fat option can be Auto (or 0) for the default width-to-length ratio of 0.5. Larger numbers result in fatter arrows. If the number is small (say, 0.1), the arrow may seem to disappear unless the Arrow Length is made longer. Printed arrows can be narrower than screen-displayed arrows.

Tag Line and Arrow Standoff

You can specify how close to bring the line or arrow to that trace with the "Line/Arrow standoff" setting in the Frame Tab. You can set the distance by entering an explicit distance in points, or typing "auto" (a value of "0" also means "auto") which varies the distance according to the output device resolution and graph size. Use a value of 1 to bring the line as near to the trace as possible. When the wave is graphed with markers, you might prefer to set the standoff to a value larger than the marker size so that the line or arrow does not intersect the marker. The left tag in the above example is using the auto standoff, and the right tag has a standoff of 6 points.

Tag Anchor Point

A tag has an Anchor point that is on the tag itself. If there is an arrow or line, it is drawn from the anchor point on the tag to the attachment point on the trace. The anchor setting also determines the precise spot on the tag which represents the position of the tag.



Anchors on Tags

The line is always drawn behind the tag so that if the anchor point is middle center the line doesn't deface the text; the examples above have a middle center anchor.

Tag Positioning

The position of a tag is determined by the position of the point to which it is attached and by the "Tag XY offset" settings. The "XY offset" gives the horizontal and vertical distance from the attachment point to the tag's anchor in percentage of the horizontal and vertical sizes of the graph's plot area.

Once a tag is on a graph you can change its "XY offset" and therefore its position by merely dragging it. You can prevent the tag from being dragged by choosing "frozen" in the Position pop-up menu in the Position Tab. Igor freezes tags when it creates them for contour labels.

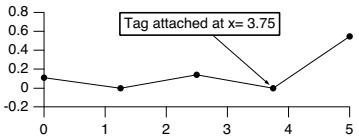
The interior/exterior setting used with textboxes does not apply to tags (see **Textbox, Legend, and Color Scale Positioning in a Graph** on page III-50).

Tags Attached to Offscreen Points

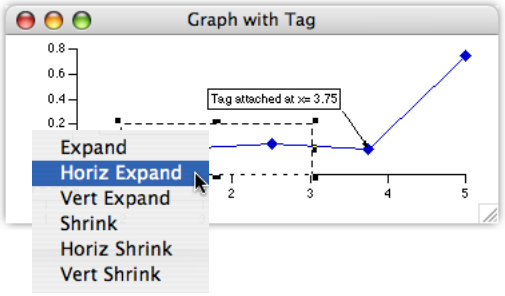
When only a portion of a wave is shown in a graph, it is possible that the attachment point of a tag isn't shown in the graph; it is "off screen" or "out-of-range".

This usually occurs because the graph has been manually expanded or the axes are not autoscaled. Igor draws the attachment line toward the offscreen attachment point.

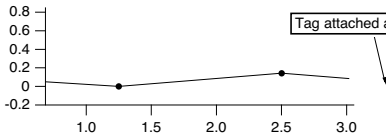
In this example graph, the attachment point at x=3.75 falls within the range of displayed X axis values.



If we zoom the graph's X range to exclude the x=3.75 attachment point...



... the tag attachment point is offscreen but the tag is still drawn.



However, you can suppress the drawing of a tag whose attachment point is offscreen by choosing "Hide the Tag" from the If Offscreen pop-up menu in the Position Tab.

If you want to see or modify a tag that is hidden, autoscale the graph so that it will no longer be hidden.

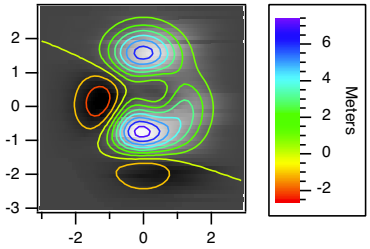
Contour Labels Are Tags

Igor uses specialized tags to create the numerical labels for contour plots. The specialization adds a "tangent" feature to automatically orient the tag along the path of the contour lines. See **Contour Labels** on page II-335 for details.

Color Scales

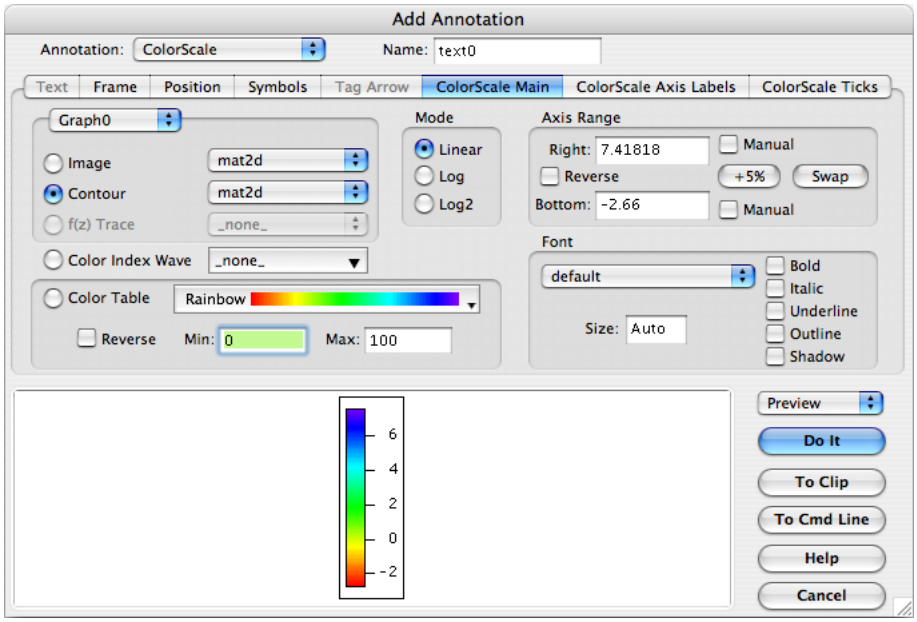
A color scale is like a tag because it is associated with data, except the color scale describes the range of the data rather than one particular value.

A color scale summarizes the range of data using a color bar and one or more axes.



ColorScale Main Tab

A color scale is associated with an f(z) trace, image plot, contour plot in a graph, or with any color index wave or color table. This association can be changed in the ColorScale Main tab.

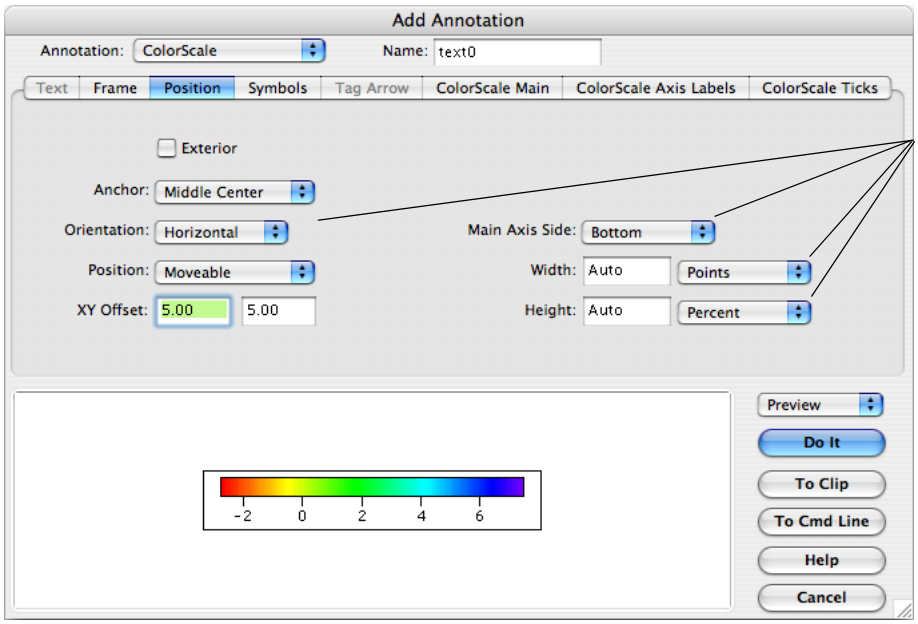


You can infer from the Graph pop-up menu that color scales can be associated with image and contour plots and f(z) traces in a graph other than the graph (or layout) in which the color scale is displayed.

Many of the color scale settings are similar to graph axis settings, such as the Linear/Log/Log2 setting and the default font settings in this ColorScale Main tab.

ColorScale Size and Orientation

The size and orientation of the color scale is set in the dialog's Position tab:



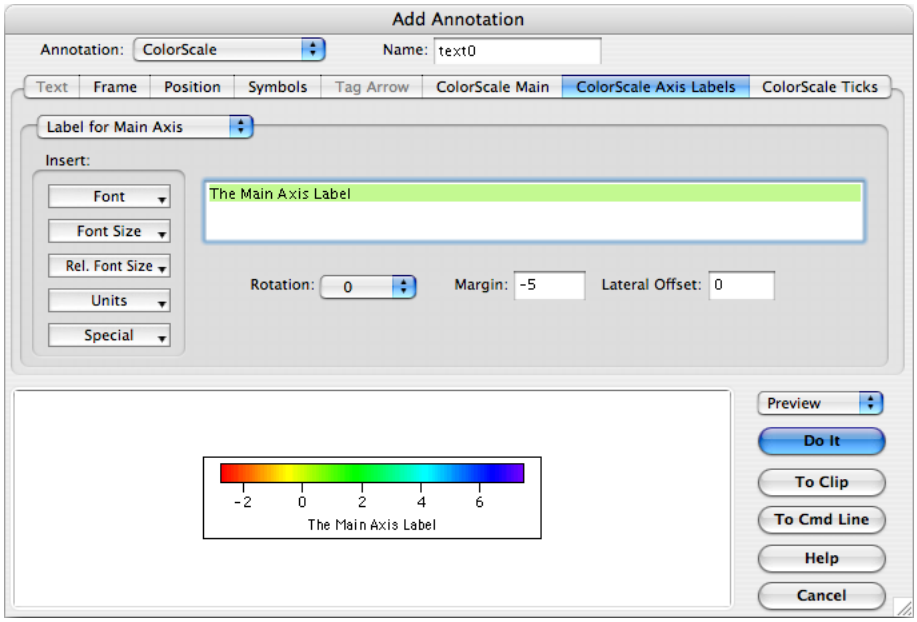
The size of a color scale is indirectly controlled by the size and orientation of the "color bar" inside the annotation, and by the various axis and ticks parameters. The annotation adjusts itself to accommodate the color bar, tick labels, and axis label(s).

When set to "Auto" or "0" the Width and Height settings cause the color scale to auto-size with the graph along the color scale's axis dimension. Horizontal color scales auto-size horizontally but not vertically, and vice versa. The long dimension of the color bar is maintained at 75% of the corresponding plot area dimension. The short dimension is set to 15 points.

You can specify custom setting of either scale dimension. Choosing Percent from the menu resizes the corresponding dimension in response to graph size changes. Points fixes the dimension so that it never changes.

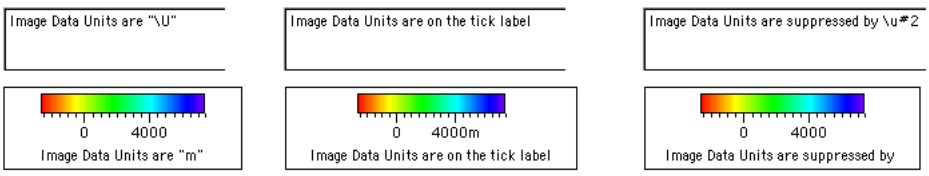
ColorScale Axis Labels Tab

The axis label for the main axis and the second axis (if any) is set in the ColorScale Axis Labels tab:



The axis label text is limited to one line. This text is the same as is used for text boxes, legends, and tags in the Text tab, but it is truncated to one line when the Annotation pop-up menu is changed to ColorScale.

The Units pop-up menu inserts escape codes related to the data units of the item the color scale is associated with. In the case of an image or contour plot, the codes relate to the data units of the image or contour matrix, or of an XYZ contour's Z data wave. See **Changing Dimension and Data Scaling** on page II-83**.** These codes work as they do in the Modify Axis dialog. For example, inserting the code "\U" adds the data units to the axis label, and "\u#2" removes the data units from the axis:

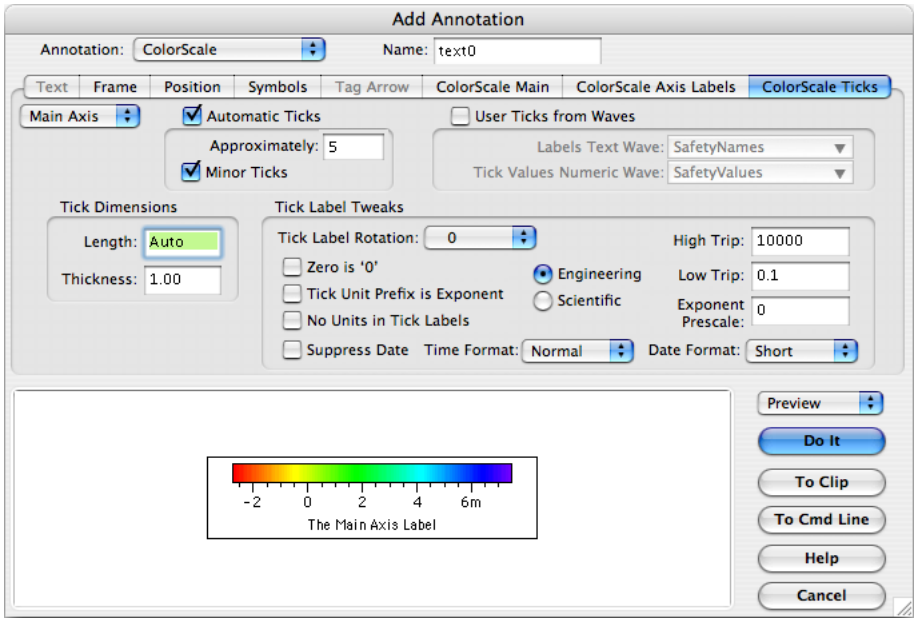


Rotation, Margin, and Lateral Offset adjust the axis label's orientation and position relative to the color axis.

The second axis label is enabled only if the Color Scale Ticks tab has created a second axis through user-supplied tick value and label waves.

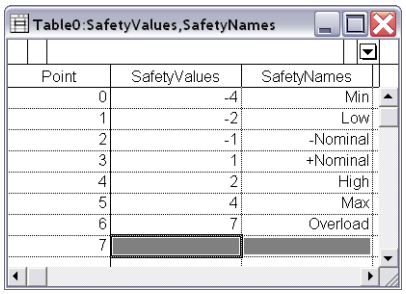
ColorScale Ticks Tab

The color scale's axis ticks settings are similar to those for a graph axis:

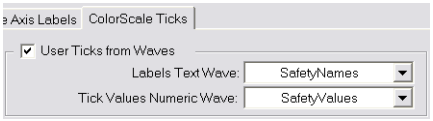


The main axis tick marks can be automatically computed by selecting Automatic Ticks, or you can supply two waves (one numeric with the tick positions, and one text wave with the corresponding tick labels):

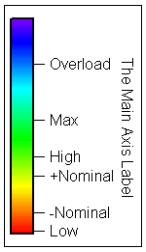
Create waves…



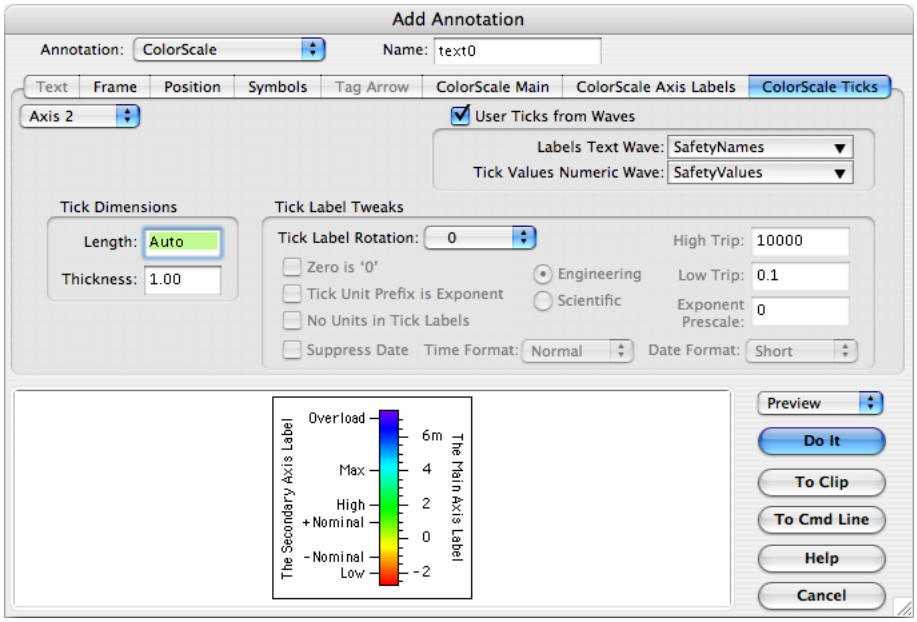
… select User Ticks from Waves…



… result.



You can also specify user-defined tick values and labels to create a second axis, which might be useful to display a temperature range in Fahrenheit and in Celsius or a distance in feet and meters. The second axis is drawn on the opposite side of the color bar from the main axis:



The Tick Dimensions are common to both axes. The length of -1 means Auto (0 means 0), otherwise the dimensions are in points. To eliminate tick marks (but not the labels), set the Thickness (not the Length) to 0.

Elaborate Annotations and Axis Labels

For the purposes of this discussion we will use the term "annotation" to include textboxes, tags, legends *and axis labels*.

You control the nuances of text in annotations by embedding escape sequences in the text. You can insert these escape sequences by choosing an item from a pop-up menu in the Insert group in the Label Axis and Add Annotation dialogs or by directly typing them into the text. When you choose an item from a pop-up menu the escape sequence is inserted at the current insertion point in the annotation text or, if a range of text is selected, the escape sequence replaces the range of text.

Simple escape sequences allow you to set things like the font, font size, subscript or superscript and inserting wave symbols. Tags support sequences for inserting dynamic text. All of this simple stuff is described in the sections following **Modifying Annotations** on page III-43. The secret to creating elaborate annotations is something we call **text info variables**.

Remember, you can use text info variables in annotations *and in axis labels*.

Elaborate Annotations Versus Equation Editors

Creating simple annotations and axis labels is very easy in Igor. Creating more elaborate mathematical, scientific, or engineering annotations involves subscripts, superscripts, changes of X and Y position and of font and style.

The good news is that Igor can do all of this. The bad news is that Igor uses escape codes to do it rather than a nifty WYSIWYG equation editor.

If you create many of these elaborate annotations, you might consider using an equation editor. You can export their output as a picture and then import it into the drawing layer of a graph or into any layer of a page layout. The main disadvantages of this approach are the additional cost of the editor and the fact that the picture format may not be cross-platform or give the highest quality. If you will be exporting your graph or page layout as EPS or printing to a PostScript printer, you should seek out an editor that can export as EPS for best results in Igor.

About Text Info Variables

The **text info variable** is a mechanism that uses escape sequences that have a higher degree of "intelligence" than simple changes of font, font size, or style. Using text info variables, you can create quite elaborate annotations if you have the patience to do it. Since you need to know about them only to do fancy things, *if you are satisfied with simple annotations, skip the rest of these Text Info Variable sections*.

A text info variable saves information about a particular "spot" (text insertion point) in an annotation. Specifically, it saves the font, font size, style (bold, outlined, etc.), and horizontal and vertical positions of the spot. Each annotation has 10 text info variables, numbered 0 through 9. You can embed an escape sequence in an annotation's text to store information about the insertion point in a particular variable. Later, you can embed an escape sequence to recall part or all of that information. In the Label Axis and Add Annotation dialogs, there are items in the Font, Font Size and Special pop-up menus to do this.

Simple Text Info Variables Example

To get a feel for this, let's look at a simple example. We want to create a textbox that says:

x = A cos(ωt)

To do this, we need to switch to the Symbol font to do the omega. Then we want to switch back to the normal font, whatever that was, to finish the annotation. We could do this without a text info variable as follows:

```
Macintosh:  x = A cos(\F'Symbol'w\F'Geneva't)
Windows:    x = A cos(\F'Symbol'w\F'Arial't)
```

Here the escape sequence `\F'Symbol'` sets the font to Symbol and `\F'Geneva'` sets it back to Geneva (`\F'Arial'` sets it back t o Arial). We have assumed that the default font is Geneva (or Arial). Using a text info variable we can accomplish the same thing without making that assumption. The text to do this is:

```
\[1x = A cos(\F'Symbol'w\F]1t)
```

The `\[1` is an escape sequence that says "save all of the information about the current insertion point in text info variable 1". So text info variable 1 now contains the font, font size, style, and horizontal and vertical positions of the insertion point. You can insert this escape sequence by choosing Store Info from the Special pop-up menu. (Note that one advantage is enhanced cross-platform compatibility.)

The `\F'Symbol'` escape sequence says "start using Symbol font". You can insert this escape sequence by choosing Symbol from the Font pop-up menu (assuming you have Symbol font installed). The `w` is an omega in the Symbol font.

The `\F]1` is an escape sequence that says "restore the font to what it was when text info variable 1 was last saved". You can insert this escape sequence by choosing "Recall font" from the Font pop-up menu.

Text Info Variables Escape Codes

At the start of the annotation, each text info variable is initialized to the default font and size for the graph or page layout the annotation is (or will be) in. The default font and font size of a *graph* is established in the Modify Graph dialog. The default font for a *page layout* is the experiment's default font (often Geneva (*Macintosh*) or Arial (*Windows*), see the Misc menu); the default font size is 10 points. The X and Y positions are initially undefined.

Here are all of the text info variable escape codes; *digit* means one of 0, 1, 2, … 9.

`\[`*digit*	Saves font, size, style and current X and Y positions in text info variable.
`\]`*digit*	Restores all but XY position from text info variable.
`\X`*digit*	Restores X position from text info variable.
`\Y`*digit*	Restores Y position from text info variable. X and Y positions of a variable are undefined until you store into it.
`\F]`*digit*	Restores font from text info variable.
`\Z]`*digit*	Restores font size from text info variable.
`\f]`*digit*	Restores style of type from text info variable.

Text info variable 0 has a special property. It defines the "main" font size which is restored using the \M escape sequence. \M also sets the Y offset from the baseline to zero. No other text info variable 0 settings (font, style, or offsets) are used by \M.

Because the initial font, size and style are stored in text info variable 0, the previous example could be made even more simple:

x = A cos(\F'Symbol'w\F]0t)

Note the right bracket,], used with \F, \Z, and \f. This indicates that what follows is the number of a text info variable. The sequence \Z09 means "set the font size to 9" whereas the sequence \Z]9 means "set the font size as stored in text info variable 9".

Elaborate Text Info Variables Example

Let's look at an elaborate example using text info variables. We want to create a textbox that looks something like this:

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} \cos(nx)\,dx$$

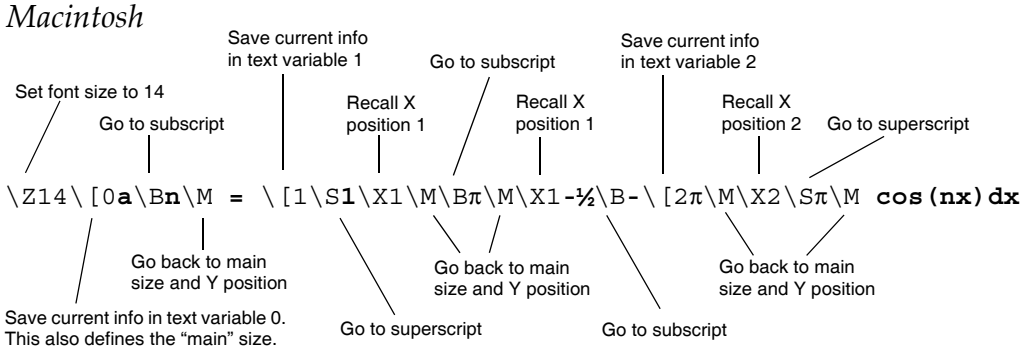
Here is the annotation text with escape sequences to produce an approximation to this equation:

```
Macintosh (the integral sign is Option-b in many fonts):
\Z14\[0a\Bn\M = \[1\S1\X1\M\Bπ\M\X1-½\B-\[2π\M\X2\Sπ\M cos(nx)dx

Windows (the integral sign and Greek letters are in Symbol font):
\Z14\[0a\Bn\M = \[1\S1\X1\M\B\F'Symbol'p\M\X1\X1-ò\B-\[2p\M\X2\Sp\M\F]0cos(nx)dx
```

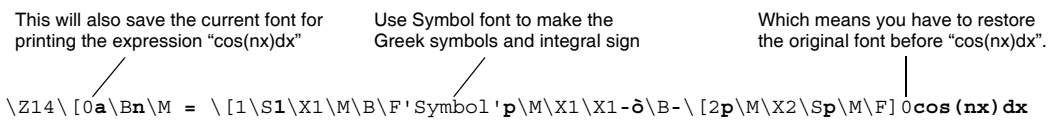
This produces the following textbox: $a_n = \frac{1}{\pi}\int_{-\pi}^{\pi}\cos(nx)dx$

The escape sequences were generated using the Font, Font Size and Special items in the pop-up menus of the Add Annotation dialog. Here is an explanation of each escape sequence:

Macintosh



Windows In addition to the previous, on Windows...



More Examples

$$y = \sum_{i=0}^{N} f(i)$$

\Z14\[0y =\Z20\F'Symbol'\[1S\F]0\X1\Z20\B\Bi=0\M\X1\Z20\S\S\Z09 N\M f(i)

$$y = K_a^b + 2$$

\Z10\[0y = \Z14K\[1\Ba\M\]1\X1\Sb\M\]0 + 2

Programming with Annotations

You can create, modify and delete annotations with the Legend, Tag, and Textbox operations. The AnnotationInfo function returns information about one existing annotation. The **AnnotationList** function (see page V-20) returns a list of the names of existing annotations. Look at the demo experiment Tags as Markers Demo in the Examples:Techniques folder for inspiration.

Changing Annotation Names

Each annotation has a name which is unique within the graph or page layout it is in. You supply this name to the Legend, Tag, Textbox, and AnnotationInfo routines to identify the annotation you want to change. You can rename an annotation by using the /C/N=*oldName*/R=*newName* syntax with the operations. For example:

```
TextBox/C/N=oldTextBoxName/R=newTextBoxName
```

Changing Annotation Types

To change the type of an annotation, apply the corresponding operation to the named annotation. For example, to change a Tag or Legend into a TextBox, use:

```
TextBox/C/N=annotationName
```

Changing Annotation Text

To change the text of an existing annotation, identify the annotation using /N=*annotationName*, and supply the new text. For example, to supply new text for the textbox named text0, use:

```
TextBox/C/N=text0 "This is the new text"
```

To append text to an annotation, use the AppendText operation:

```
AppendText/N=text0 "but this text appears on a new line"
```

Generating Text Programmatically

You can write an Igor procedure to create or update an annotation using text generated from the results of an analysis or calculation. For example, here is a function that creates or updates a textbox in the top graph or layout window. The textbox is named FitResults.

```
Function CreateOrUpdateFitResults(slope, intercept)
   Variable slope, intercept

   String fitText
   sprintf fitText, "Fit results: Slope=%g, Intercept=%g", slope, intercept
   TextBox/C/N=FitResults fitText
End
```

You would call this function, possibly from another function, after executing a CurveFit command that performed a fit to a line, passing K0 as the intercept parameter and K1 as the slope parameter. K0 and K1 are outputs from the CurveFit operation.

Usually it is better to calculate text this way, using the sprintf operation, than to use dynamic text, as described in **Dynamic Escape Codes for Tags** on page III-46, because dynamic text relies on global variables that you might inadvertently delete or whose value you might inadvertently change.

Deleting Annotations

To programmatically delete an annotation, use:

```
TextBox/K/N=text0
```

Chapter
III-3

Drawing

Overview

Igor's drawing tools are useful in page layout and graph windows for highlighting material with boxes, circles and arrows and can also be used to create simple diagrams. These drawing tools are object-oriented and optimized for the creation of publication quality graphics. All line sizes and object coordinates can be specified using real numbers. For example you can specify that a line thickness be 0.76 points.

You can graphically edit a wave's data values when it is displayed in a graph; data points can be deleted, added or modified. You can also create new waves by simply drawing them.

In control panel windows, you can use the drawing tools to create a fancy background for controls.

Like all other aspects of Igor, drawing tools are fully programmable. This allows programmers to create packages of code that add new graph types to Igor's repertoire. Although only programmers can create such packages, everyone can make use of them.

The drawing tools are available only in page layout, graph and control panel windows.

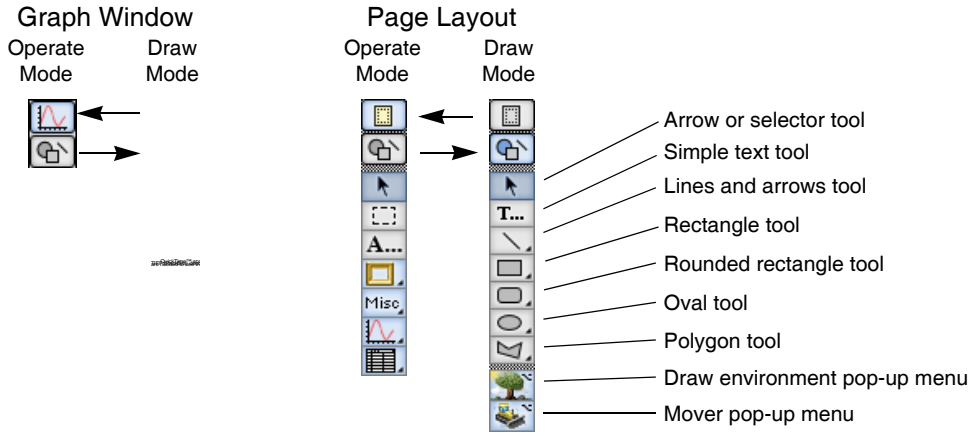
The Tool Palette

To use the drawing tools you first need to invoke the tool palette. In graphs and control panels this is done by the Show Tools menu command in the Graph or Panel menu. This command adds a tool palette along the left edge of the window. The tool palette is always available in page layout windows.

With the exception of Undo, Cut, Copy, Clear, Paste, Select All and Duplicate in the Edit menu, all drawing commands are located in the tool palette.

Once displayed, the tool palette works in two modes:

- **Operate mode:** Click the top (operate) button to enter Operate mode. In this mode, the drawing tool palette is not available, and you interact with the window as normal.
- **Drawing mode:** Click the second (drawing) button to enter Drawing mode. The entire tool palette will then be displayed, and you can add text, arrows, boxes, and other shapes to the window.



The two bottom icons in the tool palette — the Drawing Environment icon (tree and grass) and the Mover icon (bulldozer) — are pop-up menus. Both present alternate menus if you hold down Option (*Macintosh*) or Alt (*Windows*) before clicking. The icons with the triangle symbol will also present a pop-up menu if you click and hold on the icon.

All drawing tools except the polygon tool remain in effect after an object is drawn so that you may draw multiple objects of the same type.

Arrow Tool 

Use this tool to select, move and resize one or more drawing objects or a single user-defined control.

Selecting, Moving, and Resizing Objects

Click a drawing object once to select it. When selected, small black squares called **handles** appear, defining the object's size. You can drag these handles to resize single or multiple objects. By pressing Shift, an object resize can be constrained to the horizontal, vertical, or diagonal directions depending on how close the cursor is to these directions. If Option (*Macintosh*) is used rather than Shift, then the diagonal resize is replaced with a proportional resize.

If you click an object and it does not become selected then the object may be in a different drawing layer or it may not in fact be a drawing object at all.

You can select multiple drawing objects by shift clicking additional objects. You can remove an object from a selection by shift clicking a second time.

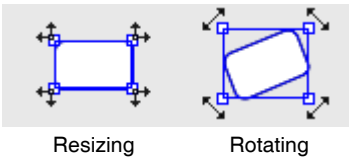
You can also select multiple drawing objects by dragging a selection rectangle around a set of drawing objects. Unless you first press Option (*Macintosh*) or Alt (*Windows*), only objects that are completely enclosed by the selection rectangle are included in the selection. With Option (*Macintosh*) or Alt (*Windows*) pressed, objects that are merely touched by the selection rectangle will also be included. You can append additional objects to a selection by holding down Shift before dragging out a selection rectangle.

You can select all objects in the current layer with the Select All item in the Edit menu.

Rotating Objects

You can rotate draw objects by clicking just beyond a selection's visible resizing handles.

When the mouse is over the invisible rotation handles the cursor changes shape:



Click and drag the invisible rotation handles to rotate the object.

Duplicating Objects

You can duplicate selected drawing objects using the Duplicate command found in the Edit menu. If, right after duplicating an object, you reposition the duplicated object and duplicate again then the new object will be offset from the last by the amount you just defined.

Deleting Objects

You can delete selected drawing objects by pressing Delete.

Modifying Objects

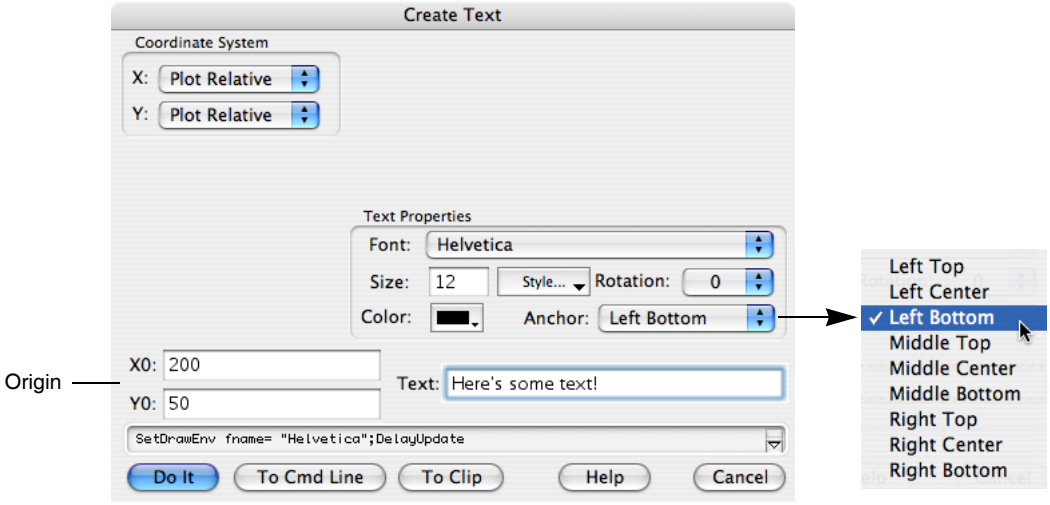
Double-clicking a drawing object will bring up a dialog for that object. You can then change any of the information listed in the dialog, and click the Do it button when you are finished. The object will be redisplayed with the changes.

Simple Text Tool 

The text tool is usually used to create a single line of simple text with the same font, size, style and rotation. As of Igor Pro 6.1, text can actually be fancy text using multiple lines and all the escape codes used by TextBox operation. However the Create Text dialog displayed by the text tool supports simple text only. Therefore, in graph and page layout windows you should use textboxes, legends and tags to create complex text, rather than the text tool.

The text tool is the only way to add text to control panel windows.

To create a line of text, click the simple text icon. The cursor will change shape to a text entry vertical line. Then move the cursor to the location in the window where you want the text origin, and click there. The following dialog will be presented:



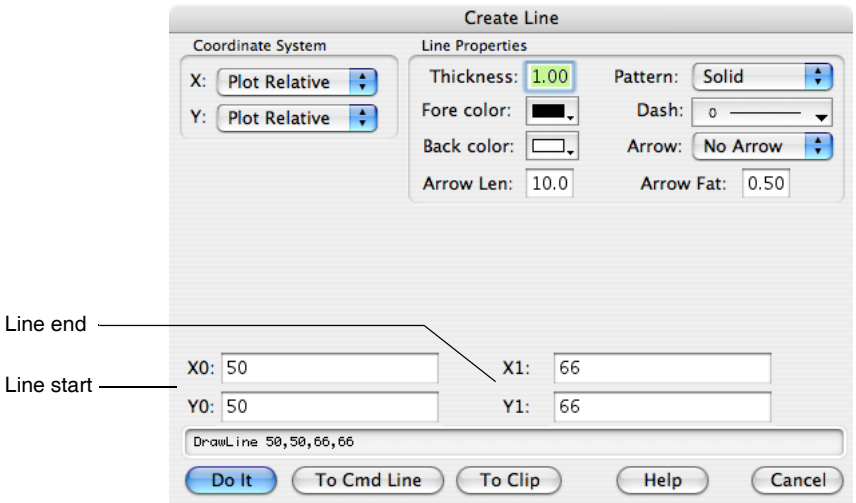
Casual users can ignore the Anchor pop-up menu. It is used mainly by programmers to ensure that text is aligned properly about the numeric origin (X0,Y0). Any rotation is applied first and then the rotated text is aligned relative to the origin as specified by the anchor setting.

To edit or change a text object, double-click the object with the Arrow tool.

Lines (and Arrows) Tool 

You can use the Lines tool to draw lines by clicking at the desired start point and then dragging to the desired stop point. Press Shift while drawing to constrain the line to be vertical or horizontal.

If you click and hold on the Lines icon you will get a pop-up menu that takes you to a dialog where you can specify the line numerically. You will see a similar dialog if you use the arrow tool to double-click a line. It will allow you to change the properties of the line.



The color of the line is principally set by the foreground color setting. The background color setting is only used when the line pattern setting is something other than Solid. You will see a pop-up palette of colors rather than the above menu of color names if the dialog is on a color screen.

The dash pattern pop-up palette is the same as the one used for graph traces. You can adjust the dash patterns by use of the Dashed Lines command in the Misc menu.

The arrow fatness parameter is the desired ratio of the width of the arrow head to its length.

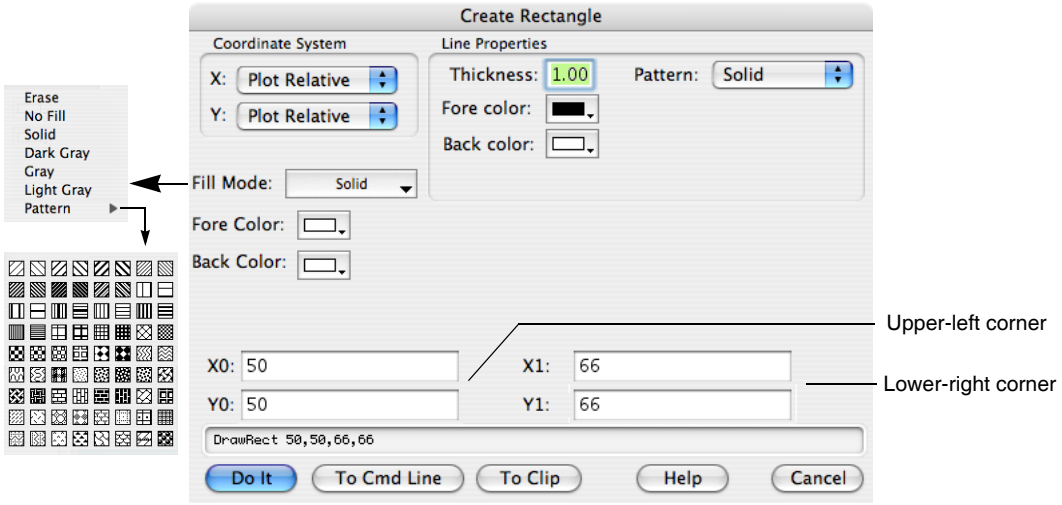
The line thickness and arrow length parameters are specified in terms of points and may be fractional.

Line start and end coordinates depend on the chosen coordinate system. See **Coordinate Systems** on page III-77 for detailed discussion. Programmers should note that the ends of a line are *centered* on the start and end coordinates. This is more obvious when the line is very thick.

Rectangle, Rounded Rectangle, Oval

These tools create objects that are defined by an enclosing rectangle. Click and hold on the appropriate icon to get a dialog where you can specify the object numerically. You will see a similar dialog if you use the arrow tool to double-click an object.

Press Shift while initially dragging out the object to create a square or circle. If you hold down Shift while resizing an object with the arrow tool, you will constrain the object in the horizontal, vertical, or diagonal directions depending on how close the cursor is to one of these directions. Thus, when you Shift-drag along a diagonal the sides will be constrained to equal length, but if you Shift-drag along a horizontal or vertical direction the object will be resized along only one of these directions. If instead you hold down Option, dragging along a diagonal resizes the object proportionally.



The Erase fill mode functions by filling the area with the current background color of the window. The fill background color is used only when a fill pattern other than Solid (or Erase) is chosen.

An object is always drawn *inside* the mathematical rectangle defined by its coordinates no matter how thick the lines. This differs from straight lines which are centered on their coordinates.

To adjust the corners of a rounded rectangle, double-click the object and edit the RRect Radius setting in the resulting dialog (similar to the above). Units are in points.

Arcs and Circles

To draw an arc or a circle by center and radius, click and hold on the Oval icon and choose Draw Arc from the resulting pop-up menu. Click and drag to define the center and the radius or start angle. Click again without moving the mouse to create a circle or move and click to define the stop angle for an arc. A variety of click and click-drag methods are supported and experimentation is encouraged.

To edit an arc, click and hold on the Oval icon in the draw tool panel and then choose Edit Arc from the resulting pop-up menu. If necessary, click on an arc to select it. You can then drag the origin, radius, start angle, and stop angle.

To change the appearance of an arc, double click to get to the Modify Arc dialog. Unlike Ovals, Arcs honor the current dash pattern and arrowhead setting in the same way as polygons and Beziers. The center of an arc or circle can be in any coordinate system (see **Coordinate Systems** on page III-77) but the radius is always in Points.

Polygon Tool

This tool creates or edits drawing objects called polygons and, in graphs, it can create or edit waves.

A polygon is an open or closed shape with one or more line segments, or edges. Polygons can be filled with a color and pattern and can have arrow heads attached to the start or end.

Although you may create a closed polygon by making the beginning and ending points the same, Igor does not recognize it as a closed shape. You can thus open the Polygon by moving either the beginning or ending points. This is subject to change in a future release.

Creating a New Polygon

You can create a polygon in one of two ways:

- **Segment Mode:** Each click defines a new vertex.
- **Freehand Mode:** Igor adds new vertices as you sweep out a smooth curve.

To create a polygon using segment mode, click the polygon icon once. Then click at the desired location for the beginning of the polygon. As you move the cursor, you will drag a line segment. A second click anchors the first line segment, and begins the second. You can keep drawing line segments until the polygon is finished.

Stop drawing by either double-clicking to define the last point or by clicking at the first vertex. You will then automatically enter edit mode (the cursor will change to), where you can reshape the polygon. Vertices are marked with square handles when in edit mode. To exit edit mode click the arrow tool.

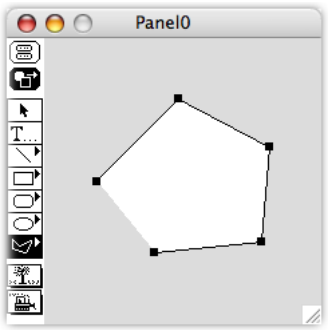
To create a polygon using freehand mode, to edit an existing polygon, or to draw or edit a Bezier curve, click and hold in the polygon icon until the pop-up menu appears.

- **Draw Poly:** This command is identical to a single click the icon.
- **Freehand Poly:** Choose the freehand command to sweep out a smooth curve as long as you hold down the mouse button. When you release the mouse button, you will automatically enter edit mode, where you can change the shape.
- **Edit Poly:** Use this to edit a preexisting polygon and is described in detail in the next section.
- **Draw Bezier:** Click and drag to define anchor and control points. Click on the first point to close a curve.
- **Edit Bezier:** Use this to edit a preexisting Bezier and is described in detail in the next section. If needed, you may need to click on a Bezier curve to select it.

Editing a Polygon

To enter edit mode, click and hold on the polygon icon, and choose Edit Poly from the pop-up menu. Then click the polygon object you want to edit.

While in edit mode you can move vertices, add or delete vertices and move line segments:



Normal cursor

Point zapper — press Option (*Macintosh*) or Alt (*Windows*)

Segment mover — press Command (*Macintosh*) or Ctrl (*Windows*)

There are a number of operations you can perform to change the polygon:

- **Move a vertex:** Click and drag the cursor on a vertex to move the vertex and stretch the associated edges.
- **Create a new vertex:** Click midway in a line segment.
- **Delete vertices:** Press Option (*Macintosh*) or Alt (*Windows*). The cursor will change to the point zapper shape shown in the above illustration. Click the vertex you want to delete.
- **Offset pairs of vertices:** Press Command (*Macintosh*) or Ctrl (*Windows*). The cursor will change shape (see the illustration above). This will allow you to move an edge, offsetting the associated pair of vertices.

There are a number of operations you can perform to edit a Bezier curve:

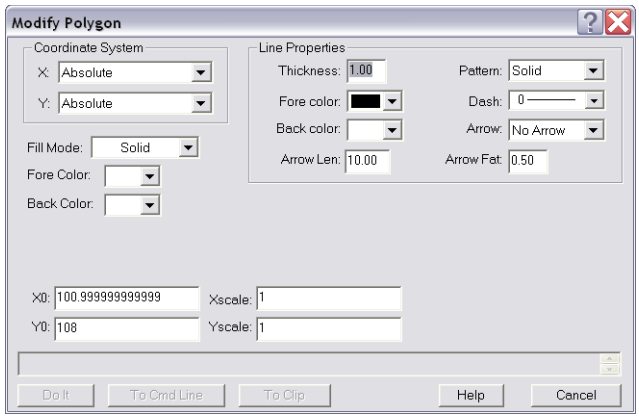
- **Move a control point:** Press Option (*Macintosh*) or Alt (*Windows*) while dragging (normally, both control points on either side of an anchor point are adjusted simultaneously.
- **Create a new anchor point:** Click on the curve between anchor points.
- **Delete an anchor point:** Press Option (*Macintosh*) or Alt (*Windows*) and then click on the anchor point.

Holding Shift while dragging will constrain movement to horizontal or vertical directions.

You can always undo the most recent edit operation.

Exit the edit mode by clicking the arrow tool.

After exiting edit mode, you can use the environment icon to adjust the other attributes of the polygon. You can even add arrows to the start or end of the polygon. Or you can double-click a polygon and the Modify Polygon dialog will appear.

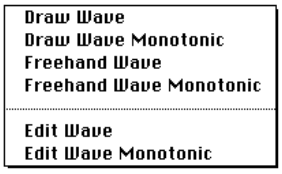


The X0 and Y0 settings determine the location of the first point.

You can change the size of the polygon by modifying the Xscale and Yscale parameters (shown in the dialog). For example, enter 0.5 for both settings to shrink the polygon to half its normal size.

Drawing and Editing Waves

When used in graphs, you can use the polygon tool to create or edit waves using the same techniques just described for drawing polygons. If you press Option (*Macintosh*) or Alt (*Windows*) and hold down the polygon icon, you will see a pop-up menu.



The first four commands will create and add a pair of waves with names of the form W_XPoly*nn* and W_YPoly*nn* where *nn* are digits chosen to ensure the names are unique. Draw Wave and Freehand Wave work exactly like the corresponding polygon drawing described above. The monotonic variants prevent you from backtracking in the X direction. As with polygons, you enter edit mode when you finish drawing.

You can edit an existing wave (or pair of waves if displayed as an XY pair) by choosing one of the Edit Wave commands and then clicking the wave trace you wish to edit. Again, the monotonic variant prevents backtracking in the X direction. If you edit a wave that is not displayed in XY mode then you won't be able to adjust the X coordinates since they are calculated from point numbers.

You can also use the GraphWaveEdit and GraphWaveDraw operations as described in **Programming** on page III-79. Note that no dialogs are available for these commands.

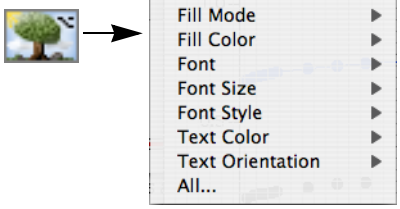
Drawing Environment Pop-Up Menu 

The Environment icon (which shows a tree and grass) does two things:

- **Change the attributes of objects:** Click and hold on the Environment icon. This action will display a menu that you use to change properties such as line thickness, color, fill pattern, and other visual attributes.
- **Change the current drawing layer:** Press and hold down Option (*Macintosh*) or Alt (*Windows*), and click the Environment icon. This will display a list of available drawing layers for you to choose from. See **Layers** on page III-78.

Changing Attributes

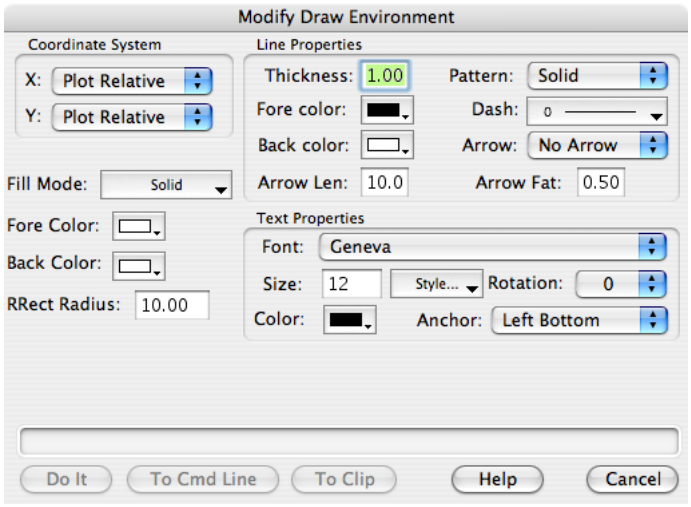
You can change the attributes of existing objects, or you can change the default attributes of objects you are yet to create:



- To change the attributes of existing objects, first select those objects. Then use the Drawing Environment pop-up menu to modify the attributes.
- To change the default attributes of objects yet to be created, make sure no objects are selected before bringing up the Drawing Environment menu. Change the attributes you want to modify. From that point on, all new objects will have the new attributes, until you change them again.

The items on this menu do not affect all types of objects. The following is a list of exceptions:

- The Fill Mode and Fill Color commands affect only enclosed shapes.
- The Line Dash and Line Arrow commands do not affect rectangles and ovals.



You can also bring up the Modify Draw Environment dialog that you use to change multiple attributes. You can bring up this dialog in two ways: You can choose All from the pop-up menu, or you can double-click an object or group of objects. Use this dialog to set or view all of the attributes accessible by the pop-up menu, plus a few more.

See **Coordinate Systems** on page III-77 for further details about coordinates.

Double-clicking a group of objects (or a single grouped object) will also bring up this dialog. In this case, the properties shown will be those of the *first* object in the group but if you change a property then all selected objects will be affected. Double-clicking a single drawing object with the selector tool will bring up a specific dialog for that object that will include coordinates and similar properties specific to the given object. These dialogs are useful to view or change the properties of an object.

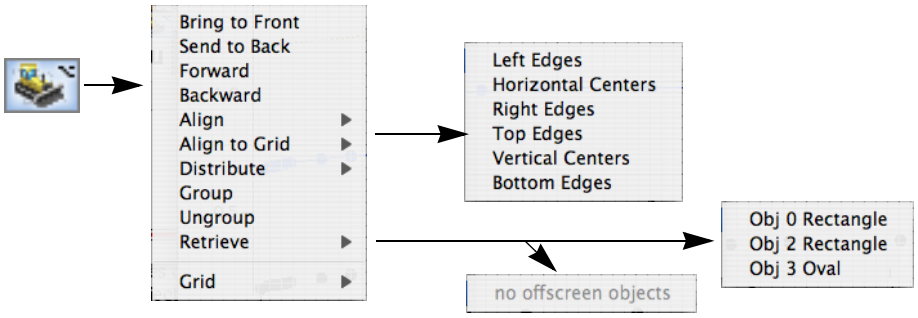
Mover Pop-Up Menu 

The Mover pop-up menu performs two types of actions:

- Adjust the drawing order within the same layer, adjust object positions, group and ungroup objects, and perform other object movement tasks. Display this menu by clicking and holding on the Mover icon.
- Select offscreen objects: If you press Option (*Macintosh*) or Alt (*Windows*) before clicking the Mover icon you will see a list of offscreen objects. In this case selecting an item will select the corresponding drawing object and will then bring up the Properties dialog for that object. Use this to set the numeric coordinates for an object to bring it back onscreen. Alternately you can cancel out of the dialog and then press Delete to remove the object.

Object Orientation

The following illustration shows the menus you get when you click and hold on the Mover icon:



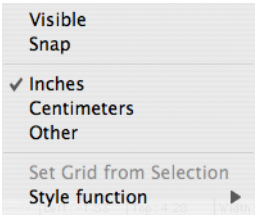
Use the Bring to Front, Send to Back, Forward and Backward commands to adjust the drawing order within the current drawing layer.

The Align command adjusts the positions of all the selected drawing objects relative to the first selected object. If you just drag-select a set of objects you will not know which is the first selected. For this reason, you should first select one of the objects and then with Shift held down, drag select the remainder of the objects. It is OK if the first selected object is included in the drag selection.

The Distribute command evens up the horizontal or vertical spacing between selected objects. The original order is maintained. This operation is especially handy when working with buttons or other controls in a user-defined panel.

The Retrieve command is used to bring offscreen objects back into the viewable area. The above example submenu for the Retrieve item was created by simply dragging two rectangles and an oval out of the window. Other ways that objects can find themselves offscreen are discussed next in the **Coordinate Systems** section and in **Export/Import** on page III-78.

Grid



You can display a grid and force objects to snap to the grid (visible or not). The mover pop-up menu has a Grid item which presents a submenu.

The default grid is in inches with 8 subdivisions. The grid origin is at the top left of the window or subwindow. Use the **ToolsGrid** operation on page V-707 to set grid properties. You can independently specify the X and Y grids and set the origin, major grid spacing, and number of subdivisions.

Set Grid from Selection

If a single object is selected, Set Grid from Selection will set the grid origin at the top left corner of the object. It two objects are selected, the origin will be set to the top left corner of the first object and the major grid spacing will be defined by the distance to the top left corner of the second object. If either the horizontal or vertical separation is small then a uniform (equal X and Y) grid will be defined by the larger distance, otherwise the horizontal and vertical grids will be set from the corresponding distances.

Style Function

Style Function presents a submenu you use to create a style function or to run one that you previously created. Style functions are created in the main procedure window with names like *MyGridStyle00*. You can edit these to provide more meaningful names.

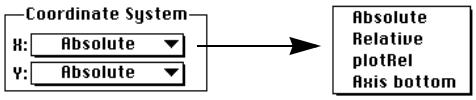
When grid snap is on, you can turn it off temporarily by engaging Caps Lock.

When dragging an object, the corner nearest to where you clicked to start dragging the object is the corner that will be snapped to the grid. You can also snap existing objects to the grid by selecting the Align to Grid menu item (mover popup).

Coordinate Systems

A unique feature of Igor's drawing tools is the ability to choose different coordinate systems. You can choose different systems on an object-by-object basis and for X and Y independently. This capability is mainly for use in graphs to allow your drawings to adjust to changes in window size or to changes in axis scaling.

You specify the coordinate system using pop-up menus found in all the above dialogs.



Absolute

In absolute mode, coordinates are measured in points relative to the top-left corner of the window. Positive x is toward the right and positive y is toward the bottom. In this mode the position and size of objects are unaffected by changes in window size. This is the default and recommended mode in page layouts and control panels.

If you shrink a window, it is possible that some objects will be left behind and may find themselves outside of the window (offscreen). In addition, if you copy an object with absolute coordinates from one window and then paste it in another smaller window it will be placed where the coordinates specify, even if it is off-screen. If you think this has happened, use the Mover pop-up menu to retrieve any offscreen objects or expand the window until the stray objects are visible.

Relative

In this mode, coordinates are measured as fractions of the size of the window. Coordinate values x=0, y=0 represent the top-left corner while x=1,y=1 corresponds to the bottom-right corner. Use this mode if you want your drawing object to remain in the same relative position as you change the window size.

This mode will produce near but not exact WYSIWYG results in graphs. This is because the margins of a graph depend on many factors and only loosely on the window size. This mode gives good results for objects that don't have to be positioned precisely, such as an arrow pointing from near a trace to near an axis. It would not be suitable if you want the arrow to be positioned precisely at a particular data point or at a particular spot on an axis. For that you would use one of the next two coordinate systems.

Plot Relative (Graphs Only)

This system is just like Relative except it is based on the plot rectangle rather than the window rectangle. The coordinates x=0, y=0 represent the top-left corner while x=1,y=1 corresponds to the bottom-right corner. This is the default and recommended mode for graphs. The following diagram illustrates both relative coordinate systems:

The Plot Relative system is ideal for objects that should maintain their size and location relative to the axes. A good example is cut marks as used with split axes. In most cases, Plot Relative is a better choice than the more complex axis-based system discussed next.

Axis-Based (Graphs Only)

The pop-up menu for the X coordinate system will include a list of all the horizontal axes and the pop-up menu for the Y coordinate will include all the vertical axes. When you choose an axis coordinate system,

the position on the screen is calculated just as it is for wave data plotted against that axis (with the exception that drawing object coordinates are not limited to the plot area). This mode is ideal when you want an object to stick to a feature in a wave even if you zoom in and out.

Axes are treated as if they extend to infinity in both directions. For this reason along with the fact that axis ranges can be very dynamic, it is very easy to end up with objects that are offscreen. Again, you can use the Mover pop-up menu to retrieve objects or, if you press Option (*Macintosh*) or Alt (*Windows*) before clicking the Mover icon, you can edit the numerical coordinates of each offscreen object. You can also end up with objects that are huge or tiny. It is best to have the graph in near final form before using axis-based drawing objects.

Axis-based coordinates are of particular interest to programmers but are also handy for a number of interactive tasks. For example you can easily create a rectangle that shades an exact area of a plot. If you use axis coordinate systems then the rectangle will remain correct as the graph is resized and as the axis ranges are changed. You can also create precisely positioned drop lines and scale (calibrator) bars.

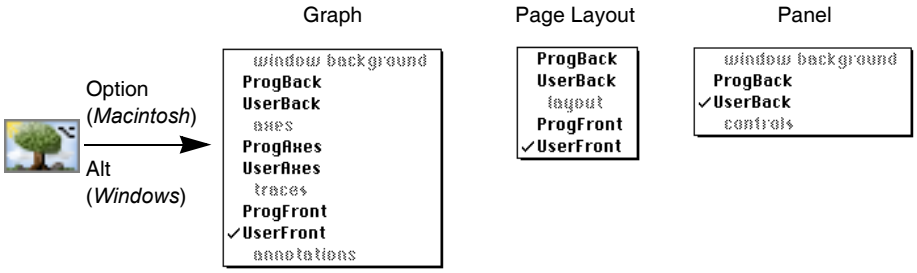
Layers

Layers allow you to control the front-to-back layering of drawing objects relative to other window components. For example, if you want to demarcate a region of interest in a graph, you can draw a shaded rectangle into a layer behind the graph traces. If you drew the same rectangle into a layer above the traces then the traces would be covered up.

Each window type supports a number of separate drawing layers. For example, in graphs, Igor provides three pairs of drawing layers. You can see the layer structure for the current window and change to a different layer by pressing Option (*Macintosh*) or Alt (*Windows*) before clicking the environment icon.

It is not necessary to use the layering system. If you do not have a need for it, it is much simpler to stick to the default layer, UserFront.

The following illustration shows the Layer pop-up menu for each of the window types. The current layer is indicated with the check mark:



You will note that drawing layers appear in pairs named ProgSomething and UserSomething. User layers are provided for interactive drawing while Prog layers are provided for Igor programmers. This usage is just a recommended convention and is not enforced. The purpose of the recommendation is to give Igor procedures free access to the Prog layers. If you were to draw into a Prog layer and then ran a procedure that used that layer then your drawing could be damaged or erased.

Note that only drawing objects in the current layer may be selected with the Arrow tool. If you find you can not select a drawing object then it must be in a different drawing layer. You will have to try the other layers until you find the right one. To move an object between layers you have to cut, switch layers and then paste.

Export/Import

Copy/Paste Within Igor

You can use the Edit menu to cut, copy, clear and paste drawing objects just as you would expect.

Drawn objects retain all of their Igor properties as long as they are not modified by any other program. If, however, you export an Igor drawing to a program and then copy it back to Igor, the picture will no longer be editable by Igor, even if you made no changes to the picture.

When selected drawing objects are copied to the Clipboard and then pasted, they retain their coordinates. However, this can cause the pasted objects to be placed offscreen if the object's coordinates don't fall within the displayed portion of the coordinate systems.

If you find that pasting does not yield what you expected, perhaps it is because some objects were pasted off-screen. You can use the Mover icon to examine or retrieve any of these offscreen objects.

Pasting a Picture Into a Drawing Layer

Pasting a picture from a drawing program may work differently than you expect. Igor does not attempt to take the picture apart to give you access to the component objects. Instead, Igor treats the entire picture as a single object that you can move and resize but not otherwise adjust. The principal reason for this limitation is that Igor's drawing capabilities are too limited to have a good chance of successfully modifying many pasted pictures.

You can change the scale of a pasted picture by either dragging the handles when the object is selected or by double-clicking the object and then setting the x and y scale factors in the resulting dialog.

Warning: There are some compatibilty issues regarding the various import and export types. See **Pictures** on page III-421 for details.

Copying from Igor to a Drawing Program

You can export an Igor drawing to a drawing program such as MacDraw, however the fine resolution available within Igor will be lost.

Drawing objects that are offscreen can create problems when a graph or page layout window is exported to a drawing program. Depending on how far outside objects extend, the drawing program may simply accept them, ignore them, or it may become confused. To reduce the problem, Igor does not include objects that are clearly offscreen when exporting. "Clearly offscreen" means that the bounding rectangle for the given object does not intersect the export rectangle.

Programming

All of the drawing capabilities in Igor can be used from Igor procedures. This provides a remarkable degree of power and flexibility (and even fun).

The programmable nature is especially useful in creating new graph types. For example, even though Igor does not support polar plots as a native graph type we were able to create a polar plot package that produces high-quality polar graphs. Nonprogrammers can use the package as-is while programmers can modify the code to suit their purposes or can extract useful code snippets for their own projects. The polar plot package is provided on the Igor Pro distribution disks along with other packages and examples. See the Polar Graphs Demo experiment in the Examples:Graphing Techniques folder.

This section describes drawing programming in general terms and provides strategies for use along with example code.

You can get a quick start on a drawing programming project by first drawing interactively and then asking Igor to create a recreation macro for the window (click the close button and look in the Procedure window). You can then extract useful code snippets for your project. Frequently all you will have to do is replace literal coordinate values with calculated values and you are in business.

Drawing Operations

Here is a list of the Operations related to drawing. See Chapter V-1, **Igor Reference**, for details.

```
DrawArc [/W=winName/X/Y] xOrg, yOrg, arcRadius, startAngle, stopAngle
or DrawBezier [/W=winName] xOrg, yOrg, hScaling, vScaling, xWaveName, yWaveName
DrawBezier [/W=winName] xOrg, yOrg, hScaling, vScaling, {x0,y0,x1,y1 …}
and DrawBezier/A [/W=winName] {xn,yn,xn+1,yn+1 …}
DrawLine [/W=winName] x0, y0, x1, y1
DrawOval [/W=winName] left, top, right, bottom
DrawPoly [/W=winName] xOrg, yOrg, hScaling, vScaling, xWaveName, yWaveName
or DrawPoly [/W=winName] xOrg, yOrg, hScaling, vScaling,{x0,y0,x1,y1 …}
and DrawPoly/A [/W=winName] {xn,yn,xn+1,yn+1 …}
DrawRect [/W=winName] left, top, right, bottom
DrawRRect [/W=winName] left, top, right, bottom
DrawText [/W=winName] x0, y0, textStr
GraphWaveDraw [/B/F/L/M/O/R/T/W=winName] [yWaveName,xWaveName]
GraphWaveEdit [/M/W=winName] traceName
GraphNormal [/W=winName]
DrawAction [/L=layerName/W=winName] keyword=value [, keyword=value …]
SetDrawLayer [/K/W=winName] layerName
SetDrawEnv [/W=winName] keyword [=value][, keyword [=value]]…
ShowTools [/A/W=winName] [toolName]
HideTools [/A/W=winName]
```

Programming Usage Notes

The following notes provide information that is supplementary to Chapter V-1, **Igor Reference**. These notes are designed to give you a view of how the commands can work together, as well as some tips on efficiency and usage. You may wish to refer to Igor's online reference for these operations as you study this section.

SetDrawLayer

Use this command to specify which layer the following drawing commands will affect. If you use the /K flag then the current contents of the given drawing layer will be killed (erased). See **Programming Strategies** on page III-82 for considerations in the use of this command and the /K flag.

SetDrawEnv

This is the workhorse command of the drawing facility. It is used to specify the characteristics for a single object, to specify the default drawing environment for future objects and to create groups of objects.

You can issue several SetDrawEnv commands in sequence; their effect is cumulative. By default, the group of SetDrawEnv commands affects only the next drawing command. Drawing commands that follow the first will use the default settings that were in effect before the SetDrawEnv commands were issued. For instance, these SetDrawEnv commands change the font and font size for only the first of the two DrawText commands:

```
SetDrawEnv fname="New York"
SetDrawEnv fsize=18     // 18 point New York, commands accumulate
DrawText 0,1,"This is in 18 point New York"
DrawText 0,0,"Has font and size in use before SetDrawEnv commands"
```

Use the save keyword in the SetDrawEnv specification to make the settings permanent. The usual use of the save keyword is at the end of the last SetDrawEnv command in a series. The permanent settings allow you to draw a number of objects all with the same characteristics without having to reissue SetDrawEnv commands before each object.

To create a grouping of objects, simply bracket a group of drawing commands with SetDrawEnv commands using the gstart and gstop keywords. Grouping is purely a user interface concept. Objects are drawn exactly the same regardless of grouping. You should use grouping when you think it will be useful to the user.

Draw<*object*> Operations

These operations, along with SetDrawEnv, operate differently depending on whether or not drawing objects are selected in the target window. If, for example, a rectangle is selected in the target window and a DrawRect command is executed then the *selected* rectangle will be changed. If, on the other hand, no rectangle is selected then *a new* rectangle will be created. This behavior exists to support interactive drawing

and is not useful to Igor programmers, since there is no programmatic way to select a drawing object. Normally, you will be creating new objects rather than modifying existing objects.

As you can see from the format of the commands, generally all you specify in the commands themselves are the coordinates. Properties such as color and line thickness are specified by SetDrawEnv commands preceding the Draw<*object*> commands. The exception is DrawText where you specify the text to be drawn.

DrawPoly and DrawBezier

The **DrawPoly** operation on page V-129 and **DrawBezier** operation on page V-126 come in the following two types:

- **Literal:** You can specify the vertices or control points with a set of literal numbers. (Polygons and bezier curves created interactively are always of the literal variety.)
- **Wave:** You can use waves to define the vertices or control points.

Because polygons and bezier curves can be of unlimited length, the /A flag allows object definitions to extend over multiple lines.

It is legal to specify a polygon with only a single point. Use this to set up a loop to append vertices to the origin vertex. Note that if you fail to add vertices and leave the polygon with just one vertex then the user will not be able to see or select the polygon.

Literal Versus Wave

As a programmer, you must choose either the literal or the wave polygon type when creating a polygon or bezier curve. This section explains the differences between the two.

The advantage of the literal method is that it does not clutter the experiment with numerous waves that may be distracting to the user. It also has the advantage that all such objects are independent of one another.

With the wave method, objects are not independent. If the user duplicates an object, or runs a window recreation macro several times, then all the objects would be linked via the wave. If the user then edits one of the objects, those edits would affect all of the associated objects; this could also be considered an advantage of the method.

A disadvantage of the literal method is that when Igor creates a recreation macro for a window containing literal method objects then all of the vertices or control points have to be specified in text. This can create huge macros that take a lot of time to create and to run. Because Igor uses the recreation macro technique when saving and restoring experiments, the use of large literal method objects can dramatically lengthen experiment save and restore time.

Wave method objects do not have this disadvantage. One nifty feature of the wave method is that you can read back the vertices or control points after the user has edited the object. Another advantage of the wave method is that you can calculate new vertices or control points at any time and the dependent objects will be automatically updated.

Screen Representation

It is important to note that the value of the first polygon vertex does not determine the location of the first vertex on the screen. The location is specified by the *xOrg*, *yOrg* parameters. Effectively the value of the first vertex is *subtracted* from all the vertices and then the value of the origin is *added* to all vertices. Thus both of the following lines will create the exact same representation on the screen:

```
DrawPoly 120,50,1,1,{0,0,20,40,60,15}
DrawPoly 120,50,1,1,{200,300,220,340,260,315}
```

When programming, the first vertex is usually `0,0`. The *hScaling*, *vScaling* parameters are probably not of any interest to programmers; use 1 for both values.

GraphWaveDraw, GraphWaveEdit, and GraphNormal

These operations relate to graph modes that are only tangentially related to drawing.

- **GraphWaveDraw** puts the graph in a mode where the user can draw a wave using the same user interface as polygon drawing.
- **GraphWaveEdit** allows the user to edit a wave using the same user interface as polygon editing.
- **GraphNormal** puts the user into normal operation mode, and is the equivalent of clicking the top icon in the tool palette.

These commands are provided so a program can allow the user to sketch a region in a graph. The program can then read back what the user did. Unlike the other drawing modes, these wave drawing and edit modes allow user defined buttons to be active. This is so you can provide a "done" button for the user. The button procedure should call GraphNormal to exit the drawing or edit mode.

The GraphWaveEdit command operates a little differently depending on whether or not you specify a wave with the command. If you do specify a wave then only that wave can be edited by the user. If you let the user choose a wave then he or she can switch to a new trace by just clicking it.

Programming Strategies

There are two distinct ways you can structure your drawing program:

- **Append:** You can append the contents of one or more layers.
- **Replace Layer:** You can replace the contents of the layers.
- **Replace Group**: You can replace the contents of a named group.

The Replace Layer Method

This method is used when you want to maintain a fairly complex drawing completely under program control. For example you may want to extend Igor by adding a new axis type or a new display method or you may want to create a completely new kind of graph. The Polar Graphs package mentioned above utilizes the replace method.

The key to the replace method is the use of the /K flag with the SetDrawLayer command. This "kills" (deletes) the entire contents of the specified layer. This is the reason for the existence of the Prog layers. After clearing out the layer you must then redraw the entire contents. To do this you will usually have to maintain some sort of data structure or database to hold all the information and status required to maintain the drawing.

For example if you are creating an artificial axis package, you will need to maintain user settings similar to those you see in Igor's modify axis dialog. In many cases setting up a few global variables or waves in a data folder will be sufficient. As an example, see the Drawing Axes procedure file in the WaveMetrics Procedures folder.

The Replace Group Method

With named groups created with the SetDrawEnv gname keyword, you can use DrawAction to delete the group or to set the insertion point for new draw commands. See the **DrawAction** operation on page V-124 for an example.

The Append Method

In this method, you will be adding a small drawing when the user runs a macro or clicks a button. Such drawings are often small and modular — a drop line or a calibration bar or a shading rectangle.

Generally, the drawing will be something the user could have done manually and may want to modify. If you need to specify a layer at all it should be a User layer. Often there will be no need to set the drawing layer at all — just use the current layer.

You may, however, need to set the layer for specific circumstances. A shading rectangle is an example of an object that should go in a specific layer, since it must be below the traces of a graph. In this case, if you use the SetDrawLayer operation, then you should set the current layer back with "SetDrawLayer UserTop".

If you are using the append method, you should avoid using the Prog layers. This is because they are intended for use where the entire layer is to be replaced under program control.

Another consideration is whether or not you should set the default drawing environment. In general you should not since you, the programmer, are just a guest of the user and it would be rude for you to change the user's settings. On the other hand, if you are appending a fairly complex drawing, it might be inconvenient to have to keep making the same settings over and over.

Grouping

Finally, you may want to make sure your drawing is grouped. That way when the user clicks on your drawing the whole thing will be selected and can be easily moved.

Example: Drop Lines

In this example, we want to create a macro to add a line from a particular point on a wave to the bottom axis. To be useful this macro would have to be able to add drop lines to any axis. That was not done here because it would add a lot of complexity that might obscure the example. This macro draws a line from the point that Cursor A is on to the bottom axis.

```
#pragma rtGlobals= 1          // keep V_min, etc local

Macro AddDropLine(doArrow)
   Variable doArrow= 2
   Prompt doArrow,"Include arrow head?",popup "No;Yes"

   PauseUpdate; Silent 1
   doArrow -= 1               // pop-up menu items start from 1
   Variable csrposy,csrposx,axposy
   csrposx= hcsr(A)
   csrposy= vcsr(A)
   GetAxis/Q Left; axposy= V_min
   SetDrawEnv xcoord= bottom,ycoord= left
   if( doArrow )
      SetDrawEnv arrow= 1,arrowlen= 8,arrowfat= 0.5
   else
      SetDrawEnv arrow= 0
   endif
   DrawLine csrposx,csrposy,csrposx,axposy
End
```

Drawing Shortcuts

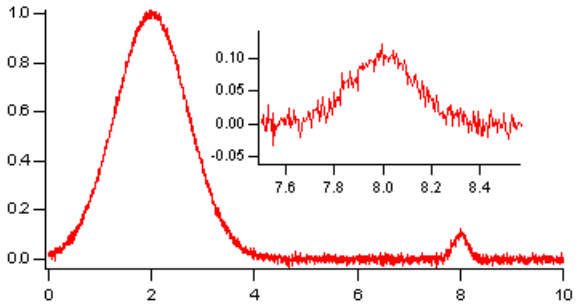
Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To temporarily invoke the Arrow tool while in Operate mode	Press Command-Option. You can now select an object and move or resize it. This shortcut works even if the tool palette is not showing.	Press Ctrl+Alt. You can now select an object and move or resize it. This shortcut works even if the tool palette is not showing.
To invoke the dialog to modify a control or drawing object while in Operate mode	Press Command-Option and double-click the control or drawing object. This shortcut works even if the tool palette is not showing. Also, if the palette is showing, the Drawing mode's Arrow tool becomes selected after the dialog is dismissed.	Press Ctrl+Alt and double-click the control or drawing object. This shortcut works even if the tool palette is not showing. Also, if the palette is showing, the Drawing mode's Arrow tool becomes selected after the dialog is dismissed.
To nudge a selected drawing object or control	Use the Arrow keys. Press Shift to nudge faster.	Use the Arrow keys. Press Shift to nudge faster.

Chapter
III-4

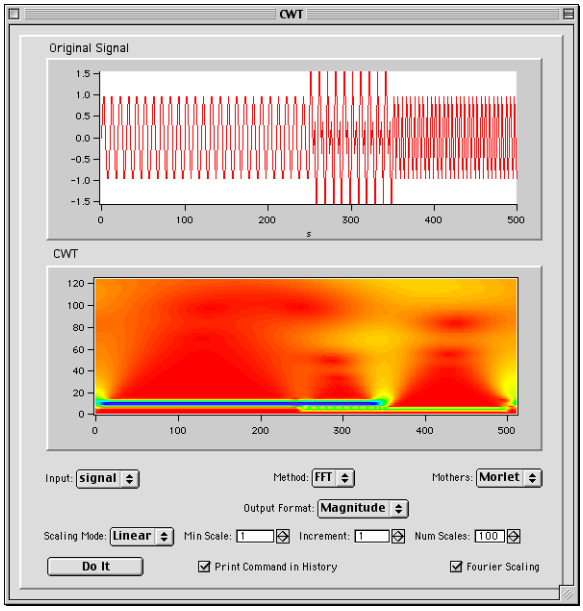
Embedding and Subwindows

Overview

You can embed graphs, tables, and control panels into other graph, control panel, and page layout windows. In addition, you can embed notebooks in control panels only. The embedded window is called a *subwindow* and the enclosing window is called the *host*. Subwindows may be nested in a hierarchy of arbitrary depth. The top host window in the hierarchy is known as the *base*. In the following example, the smaller, inset graph is a subwindow:



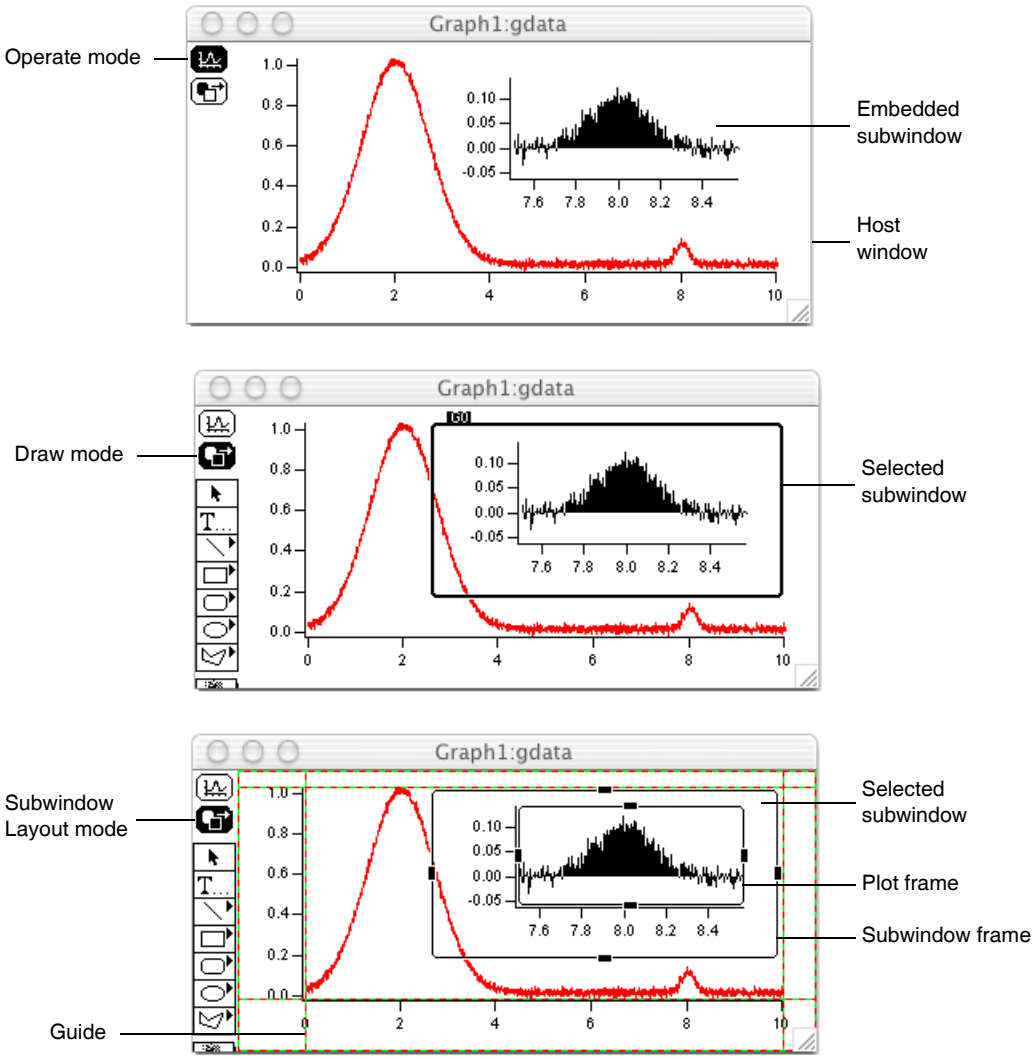
Although you can create graphs like this by careful positioning of free axes, it is much easier to accomplish using embedding. In the next example, the two graphs are subwindows embedded in a host panel:



This example is derived from the CWT demo experiment which you can find in the Analysis section of your Examples folder.

Subwindow Terminology

When a window is inserted into another window it is said to be *embedded*. In some configurations (see **Restrictions** on page III-87), an embedded window does not support the same functionality that it has as a standalone window. It is then called a *presentation-only* object. For example, when a table is embedded in a panel, it has scroll bars and data entry features just like a standalone table. But when a table is embedded in a graph or in a page layout, it is a presentation-only object with no scroll bars or other user interface elements.



Restrictions

The following table summarizes the rules for allowed host and embedded subwindow configurations.

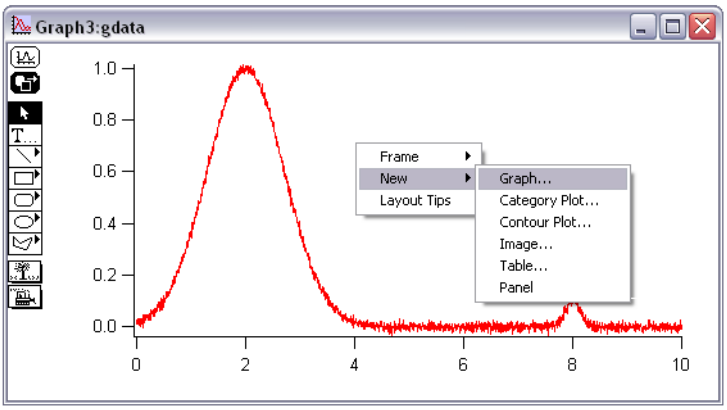
Subwindow	Host: Graph	Host: Table	Host: Panel	Host: Layout
Graph	Yes	No	Yes	Yes
Table	Yes*	No	Yes	Yes*
Panel	Yes†	No	Yes	No
Layout	No	No	No	No
Notebook	No	No	Yes	No

* Tables embedded in a graphs or layouts are presentation-only objects. They do not support editing of data.

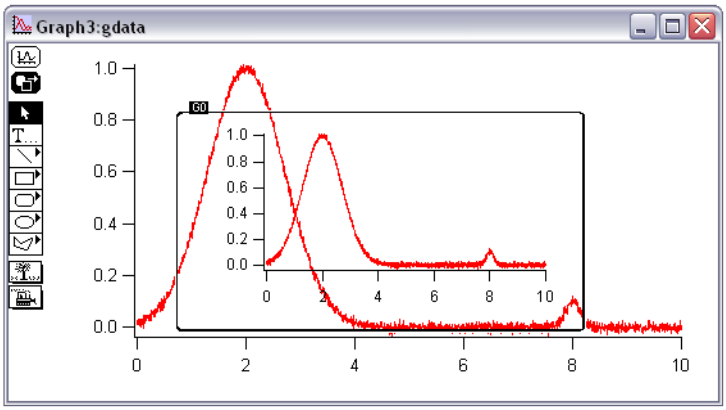
† Panels can be embedded in base graphs only.

Creating Subwindows

You can create subwindows either from the command line (see **Subwindow Command Concepts** on page III-95) or interactively using contextual menus. To add a subwindow interactively, add tools to the target window (Show Tools menu item), click the lower icon to enter Drawing mode, and then right-click (*Windows*) or Control-click (*Macintosh*) in the interior of the window and choose the desired type of subwindow from the New menu:



You will be presented with the standard dialog for creating a new window but the result will be a subwindow:



You can position the subwindow by clicking on its heavy frame to enter Subwindow-Layout mode (see **Subwindow Layout Mode and Guides** on page III-90). Finally, click the top icon of the tools to adjust the graphs using Operate (Normal) mode.

Positioning and Guides

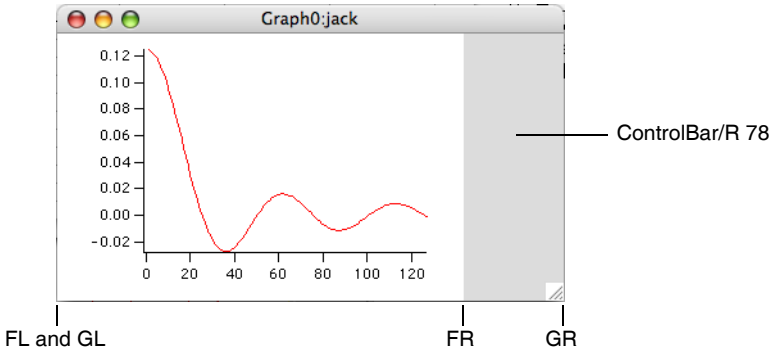
Subwindows may be positioned in their hosts in a wide variety of ways. You can specify the position of a subwindow numerically using either absolute (fixed distance) or relative modes. You can also attach key locations in a subwindow to named guides.

Guides are horizontal or vertical reference locations defined by the immediate host of a subwindow and may be either fixed (built-in) or moveable (user-defined). Built-in guides represent fixed locations of the host such as its frame or the interior plot area of graphs. Built-in guides can not be moved except by moving the object to which the guide refers.

All host windows have built-in guides named *FL*, *FT*, *FR*, and *FB* for Frame Left, Frame Top, Frame Right, and Frame Bottom. Graphs also have the corresponding *PL*, *PR*, *PT*, and *PB* for the interior plot area. In addition, base graphs (top level host graph windows) have built-in guides *GL*, *GR*, *GT*, and *GB* for the Graph area.

	Left	Right	Top	Bottom
Host Window Frame	FL	FR	FT	FB
Host Graph Rectangle	GL	GR	GT	GB
Inner Graph Plot Rectangle	PL	PR	PT	PB

The graph area is the total area of the graph window excluding the areas occupied by the tool palette and the cursor information panel. The frame area is the total area of the graph window excluding the areas occupied by the tool palette, the cursor information panel and the control bar.



User-defined guides may be based on built-in or other user-defined guides. They may be defined as being either a fixed distance from a guide or a relative distance between two guides.

Reference points of a subwindow that may be attached to guides include the outer left, right, top and bottom for all subwindow types and, for graphs only, the interior plot area.

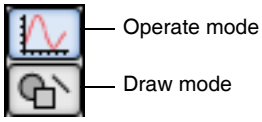
Guides are especially useful when creating stacked graphs. By attaching the plot left (PL) location on each graph to a user-defined guide, all left axes will be lined up and will move in unison when you drag the guide around. This is illustrated in **Layout Mode and Guide Tutorial** on page III-91.

Frames

You may specify a frame style for each subwindow. Frames, if any, are drawn inside the rectangle that defines the location of the subwindow and the normal content is then inset by the frame thickness. Frames may also be specified for base graph and panel windows. This is handy when you want to include a frame when you export or print a graph. You can adjust the frame for a window or subwindow using a contextual menu (right-click (*Windows*) or Control-click (*Macintosh*)).

Subwindow User-Interface Concepts

Each host window has two main modes corresponding to the top two icons in the window's toolbar. Choose Show Tools from the Graph or Panel menu to show the toolbar in which clicking the top icon selects Operate (Normal) mode and clicking the second icon selects Drawing mode.



When using subwindows, there is a third mode: Subwindow Layout (see **Subwindow Layout Mode and Guides** on page III-90).

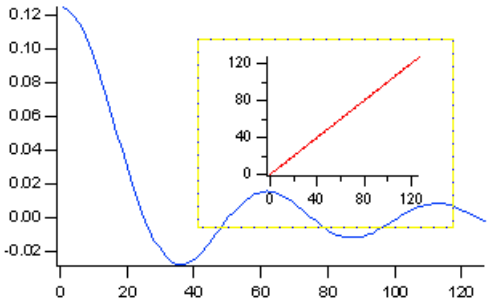
When not using subwindows, a particular window is the *target window* — the default window for command-line commands that do not explicitly specify a window. The addition of subwindows leads to the analogous concept of the *active subwindow*.

You make a subwindow the active subwindow by clicking it. In Operate mode the active subwindow is indicated by a yellow and black border. In Drawing mode it is indicated by a heavy black border with the name of the subwindow shown in the upper left corner.

Panel subwindows are exceptions in that clicking them in Operate mode does not make them the active subwindow. You must click them while in Drawing mode.

As an example, execute the following:

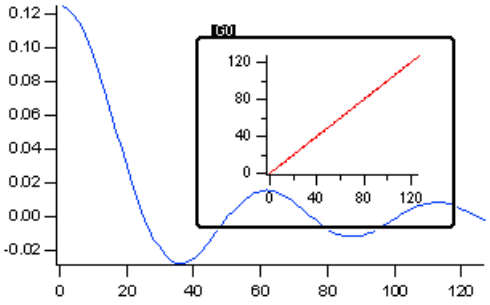
```
Make/O jack=sin(x/8)/x,sam=x
Display jack
Display/W=(0.5,0.14,0.9,0.7)/HOST=# sam
```



Notice the yellow and black border around the newly created subwindow:.

This indicates that it is the active subwindow. Now double click on the curve in the host window (but not within the subwindow border). Change the color of the trace `jack` to blue and notice that the subwindow is no longer active. Now move the mouse over the subwindow and notice that the cursor changes to the usual shapes corresponding to the parts of the graph that it is hovering over. Drag out a selection rectangle in the plot area of the subwindow and notice that the Marquee pop-up menu is available for use on the subwindow and that the subwindow has been activated. Depending on your actions in a window, Igor activates subwindows as appropriate and generally you do not have to be aware of which subwindow is active.

Choose Show Tools from the Graph menu and notice that the tools are provided by the main window. Tools and the cursor information panel are hosted by the base window but may apply to subwindows.



Click the drawing icon in the tool bar.

Notice the subwindow, which had been indicated as active using the yellow/black border, now has a heavy frame.

You are now in a mode where you can use drawing tools on the subwindow. To draw in the main window, click outside the subwindow to make the host window active.

When in Operate mode, the main menu bar always references the base window. In order to make changes to a subwindow of a different type, you can use a context click to access a menu specific to the subwindow. In Drawing mode, the main menu references the active subwindow. For example, if the base window is a graph with an embedded table, the menu bar contains a Graph menu when in Operate mode. However, when the embedded table is selected in Drawing mode, the main menu bar contains a Table menu.

When in Drawing mode, you can right-click (*Windows*) or Control-click (*Macintosh*) to get a pop-up menu from which you can choose frame styles and insert new subwindows or delete the active subwindow. Deleting a subwindow is not undoable.

The info box (see **Info Box and Cursors** on page II-286) in a graph targets the active subgraph. You can not simultaneously view or move cursors in two different subgraphs.

Subwindow Layout Mode and Guides

To layout one or more subwindows in a host window, enter Drawing mode, click the selector (arrow) tool and click in a subwindow. A heavy frame will be drawn with the name of the subwindow in the upper left. Now click on the frame to enter subwindow layout mode. In this mode, the subwindow is drawn with a light frame with handles in the middle of each side. In addition, built-in and user guides are drawn as dashed lines.

A subwindow can be moved by dragging its frame and resized using the handles. If a handle is positioned near a guide, it will snap in place and attach itself to the guide. However, if one or more handles are attached to guides and then the subwindow is moved using the frame, all attachments will be deleted.

A graph subwindow is drawn using two frames. The inner frame represents the plot area of the graph. Its handles can be attached to guides to allow easy alignment of multiple graph subwindows.

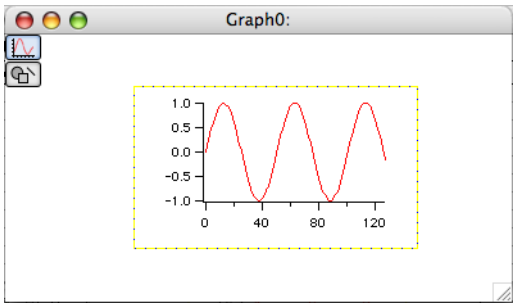
You can create user-defined guides by pressing Alt (*Windows*) or Option (*Macintosh*) and then click-dragging an existing guide. By default, the new guide will be a fixed distance from its parent. You can convert the new guide to relative mode (where the guide is specified as a fraction of the distance between two guides) by right-clicking (*Windows*) or Control-click (*Macintosh*) on the new guide and then choosing a partner guide from the "make relative to" list of other guides. You can also use the right-click menu to convert a relative guide to fixed or to delete a guide, if it is not in use.

Layout Mode and Guide Tutorial

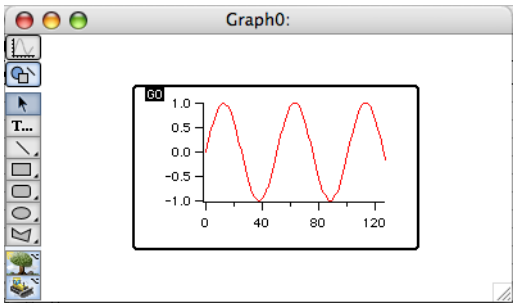
In a new experiment, execute these commands:

```
Make/O jack=sin(x/8),sam=cos(x/8)
Display
Display/HOST=# jack
ShowTools
```

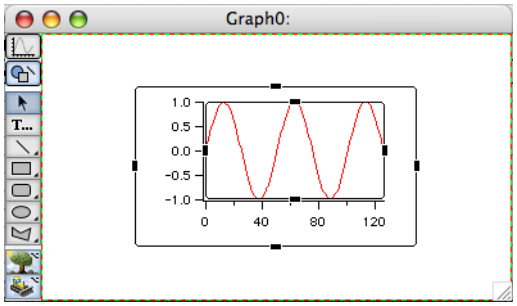
The first `Display` command created an empty graph and the second inserted a subgraph. We used the command line just to get going quickly. The graph is in Operate mode and it looks like this:



Click the lower icon in the tool bar to enter Drawing mode and notice the subwindow is drawn with a black frame with the name of the subwindow (`G0`):

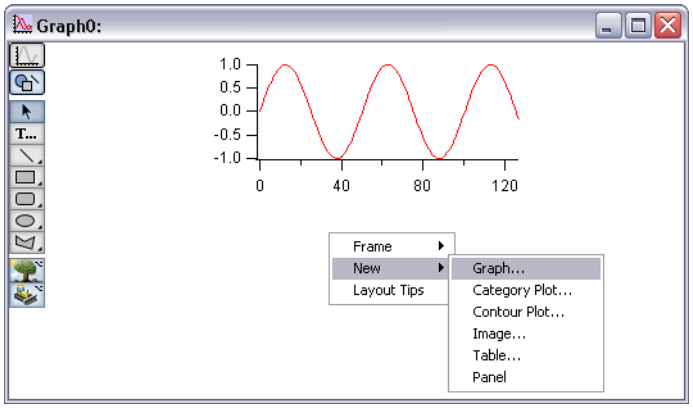


Use the arrow tool to click on the black frame around the subgraph. You are now in Subwindow Layout mode as indicated by the two rectangles with handles on each edge of the graph.



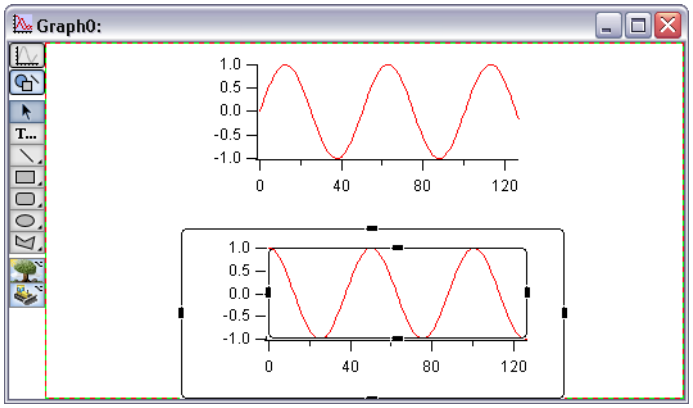
Position the mouse over the outer rectangle until the cursor changes to a four-headed arrow. Drag the subwindow up as high as it will go and then drag the bottom handle up to just above the halfway point so that the subgraph is in the upper half of the window.

Click outside the subwindow to leave Subwindow Layout mode and then click again to select the main (empty) graph as the active subwindow. Right-click (*Windows*) or Control-click (*Macintosh*) below the subgraph and choose the New→Graph menu item:



Pick `sam` as the Y wave in the resulting dialog and click Do It. This creates a new subwindow and makes it active. Click on the heavy frame to enter Subwindow Layout mode for the new subgraph and position it in the lower half of the window.

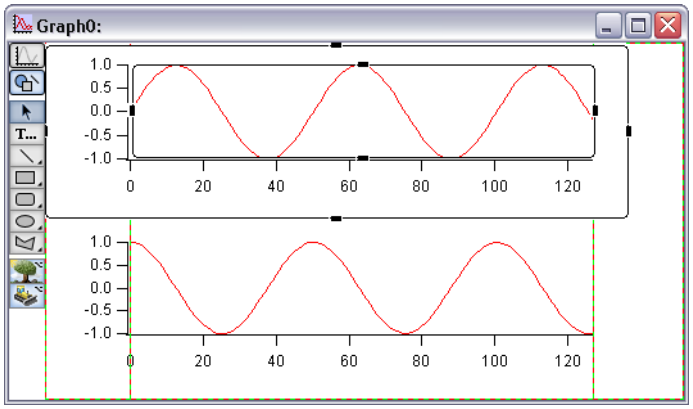
While still in Subwindow Layout mode for the second graph, notice the red/green dashed lines around the periphery. These are fixed guides and are properties of the base window. Hold down Alt (*Windows*) or Option (*Macintosh*) and move the mouse over the left hand dashed line. When you notice the cursor changing to a two headed arrow, click and drag to the right about 3 cm to create a user-defined guide. Use the same technique to create another user-defined guide based on the right edge also inset by about 3 cm:



Move the mouse over the new guides and notice the cursor changes to a two headed arrow indicating they can be moved.

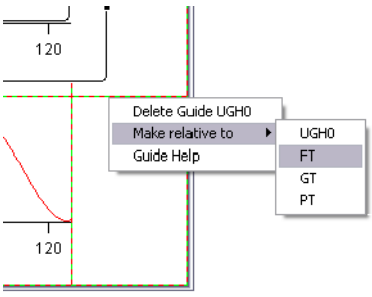
While still in Subwindow Layout mode for the second graph, click in the black handle centered on the left axis and drag the handle over the position of the left user guide. Notice that it snaps into place when it is near. Release the mouse button and use the same technique to connect the right edge of the interior plot area to the right user guide.

Now place the top subgraph in Subwindow Layout mode and connect its left and right plot area handles to the user guides:

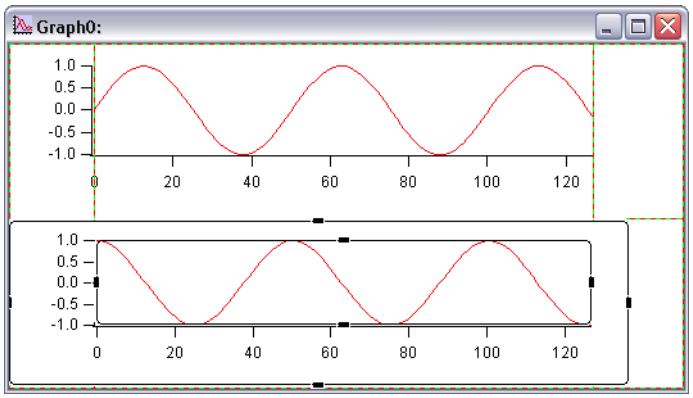


While still in Subwindow Layout mode, drag the user guides around and notice that both graphs follow.

The two guides we created are of a type that is a fixed distance from another (the frame left (FL) and frame right (FR) in this case). We will now create a relative guide. Hold down Alt (*Windows*) or Option (*Macintosh*) and move the mouse over the bottom dashed line near the window frame. When you notice the cursor changing to a two headed arrow, click and drag up to about the middle of the graph to create another user-defined guide. Position the mouse over the new guide, right-click (*Windows*) or Control-click (*Macintosh*), and choose Make Relative to→FT from the menu.



Now, as you resize the window, the guide will remain at the same relative distance between the bottom (FB) and the top (FT). Use the handles to attach the bottom of the top graph to the new guide and then put the bottom graph into Subwindow Layout mode and attach its top to the guide:



Graph Control Bars and Subpanels

Prior to Igor Pro 5, you could use the arrow tool in a graph to drag down a control bar from the top of the window. You would then use this area for buttons. Now you can drag out control areas from all four sides of a graph window.You do this by entering Drawing mode and clicking just inside an outer edge of the window. Subpanels are automatically created and anchored to appropriate guides. If you want to create an Igor Pro 4-compatible control area with no subpanel, you can either delete the automatically generated subpanel or you can use the ControlBar command.

To adjust the size of a control bar, be sure the subpanel is not currently active to avoid putting it into layout mode by a click near the frame. Unlike most other draggable objects in Igor, the cursor does not change shape to indicate it is over the control area margins.

Page Layouts and Subwindows

Subwindows in Page Layouts can be a bit confusing because you use two different modes to position conventional graph and table objects versus subwindows. In the normal mode for a page layout (top icon in the toolbar), you can adjust the positioning of conventional layout objects but not subwindows because they are in operate mode.

In operate mode, subwindow graphs act just like conventional graph windows and allow marquee expansion, double clicking on graph elements to bring up dialogs, dragging textboxes and axes and other normal graph behavior. But to adjust the position of subwindows, you have to enter Subwindow Layout (see **Subwindow Layout Mode and Guides** on page III-90).

Although a bit confusing, the use of subwindows in Page Layouts is very useful because the Page Layout is then self-contained and need not refer to other windows. An additional feature of subwindows is the ability to see more detail when you expand the layout above 100%. Conventional layout objects are simply expanded images originally taken at 100% while subwindows are actually drawn at the increased resolution.

Here are a few reasons to use the conventional window or object layout method rather than subwindows:

- You can use cursors and buttons in a full graph but not in a subgraph.
- You can place the same graph in multiple layouts.
- You can have a graph window be a different (and more convenient) size than the layout object.

In a page layout, you can insert a graph subwindow or table by first using the marquee tool to specify the desired location and then using the pop-up menu available in the interior of the marquee to choose one of several subwindow types. If you need to reposition the new subwindow, you will need to enter Drawing mode and use the selector (arrow) tool of the drawing palette, not the layout arrow tool.

You can convert conventional layout graph and table objects to subwindows via the contextual menu for the object. In operate mode (select the top icon in the layout tool panel), Control-click or right-click anywhere on the layout object and choose Convert Graph/Table To Embedded. Portions of a graph that are not allowed in layouts, such as buttons and subpanels, will be lost in the conversion.

Similarly, you can convert a subgraph or subtable to a conventional window and layout object via the contextual menu. In operate mode, Control-click or right-click and choose Convert To Graph/Table and Object. In a graph you must click in an area free of traces or axes, such as in the graph margin, to get the correct popup menu.

Notebooks as Subwindows in Control Panels

You can create a notebook subwindow in a control panel using the NewNotebook operation. A notebook subwindow might be used to present status information to the user or to permit the user to enter multi-line text. Here is an example:

```
NewPanel /W=(150,50,654,684)
NewNotebook /F=1 /N=nb0 /HOST=# /W=(36,36,393,306)
Notebook # text="Hello World!\r"
```

The notebook subwindow can be plain text (/F=0) or formatted text (/F=1).

By default, the notebook ruler is hidden and the notebook status area width is set to zero when a notebook subwindow is created. You can change this using the Notebook operation.

To make it easier to use for text input or display, when a formatted text notebook subwindow is first created and when you resize the width of the subwindow, Igor automatically adjusts the Normal ruler's right indent so that all of the text governed by the Normal ruler fits in the subwindow. This adjustment is done for the Normal ruler only. Other rulers, including Normal+ (variations of Normal) rulers, are not adjusted.

You can programmatically insert text in the notebook using the **Notebook** operation.

If you create a window recreation macro for the control panel, by default the contents of the notebook subwindow are saved in the recreation macro. If you later run the macro to recreate the control panel, the notebook sub-

window's contents will be restored. This also applies to experiment recreation which automatically uses window recreation macros.

If you do not want the contents of the notebook subwindow to be preserved in the recreation macro, you must disable the autosave property, like this:

```
Notebook Panel0#nb0, autosave=0
```

When you create a window recreation macro while autosave is on, it will contain commands that look something like this:

```
Notebook kwTopWin, zdata="GaqDU%ejN7!Z)ts!+J\\.F^>EB"
Notebook kwTopWin, zdata= "jmRiCVsF?/]21,HG<k,\"@i1,&\\.F^>EB"
Notebook kwTopWin, zdataEnd=1
```

The Notebook zdata command sends to the notebook encoded binary data in an Igor-private format that represents the contents of the notebook when the recreation macro was created. In real life, there would be a number of zdata commands, one after the other, which cumulatively define the contents of the notebook. The notebook accumulates all of the zdata text. The zdataEnd command causes the notebook to decode the binary data and use it to restore the notebook's contents.

When you save an experiment containing a control panel, a window recreation macro is created for you by Igor and when you open the experiment, Igor runs the recreation macro to recreate the control panel. If autosave is off, after saving and reopening the experiment, the notebook will be empty. If autosave is on, the window recreation macro will include zdata and zdataEnd commands that restore the contents of the notebook subwindow.

The encoded binary data includes a checksum. If the Notebook zdata commands have been altered, the checksum will fail and you will receive an error when the Notebook zdataEnd command executes.

Subwindow Command Concepts

All operations that create window types that can be subwindows can take a /HOST=*hcSpec* flag in order to create a subwindow in a specific host. In addition, operations and functions that can modify or operate on a subwindow can affect a specific subwindow using the /W=*hcSpec* flag (for operations) or an *hcSpec* as a string parameter (for functions).

Subwindow Syntax

The Command Line syntax for identifying a subwindow for a command is summarized in this table.

Subwindow Specification	Location
baseName	Base host window
baseName#*sub1*	Absolute path from base host window
#*sub1*	Relative path from the active window or subwindow
#	Active window or subwindow
##	Host of active subwindow

The window "path" uses the # symbol as a separator between a window name and the name of a subwindow. If you have a panel subwindow named P0 inside a graph subwindow named G0 inside a panel named Panel0, the absolute path to the panel subwindow would be Panel0#G0#P0. The relative path from the main panel to the panel subwindow would be #G0#P0.

Subwindow Sizing

When /HOST is used in conjunction with **Display**, **NewPanel**, **NewWaterfall**, **NewImage** and **Edit** commands to create a subwindow, the values used with the window size /W=(*a*,*b*,*c*,*d*) flag can have one of two different meanings. If all the values are less than 1.0, then the values are taken to be fractional relative

to the host's frame. If any of the values are greater than 1.0, then they are taken to be fixed locations measured in points or, for panels, pixels relative to the top left corner of the host.

Guides may override the numeric positioning set by /W. All operations supporting /HOST may take the /FG=(*gleft*,*gtop*,*gright*,*gbottom*) flag where gleft-gbottom are the names of built-in or user-defined guides. FG stands for Frame Guide and this flag specifies that the outer frame of the subwindow is attached to the guides. A * character may be substituted for a name to indicate that the default value should be used.

The inner plot area of a graph subwindow may be attached to guides using the analogous PG flag. Thus a subgraph may need up to three specifications. For example:

```
Display/host=#/W=(0,10,400,200)/FG=(FL,*,FR,*)/PG=(PL,*,PR,*) sam
```

When the subwindow position is fully specified using guides, the /W flag is not needed but it is OK to include it anyway.

Subwindow Operations and Functions

Here are the main operations and functions that will be useful in dealing with subwindows. For full documentation, see Chapter V-1, **Igor Reference**.

ChildWindowList(*hostName*)

DefineGuide [/W= *winName*] *newGuideName* = {[*guideName1*, *val*
[, *guideName2*]]} [,…]

KillWindow *winNameStr*

MoveSubwindow [/W=*winName*] *key* = (*values*)[, *key* = (*values*)]…

RenameWindow *oldName*, *newName*

SetActiveSubwindow *subWinName*

Chapter
III-5

Exporting Graphics (Macintosh)

Overview

This chapter discusses exporting graphics from Igor graphs, page layouts and tables to another program on Macintosh. You can export graphics through the Clipboard by choosing Edit→Export Graphics, or through a file, by choosing File→Save Graphics.

Igor Pro supports a number of different graphics export formats. You can usually obtain very good results by choosing the appropriate format, which depends on the nature of your graphics, your printer and the characteristics of the program to which you are exporting.

Unfortunately, experimentation is sometimes required to find the best export format for your particular circumstances. This section provides the information you need to make an informed choice.

This table shows the available graphic export formats:

Export Format	Export Method	Notes
Quartz PDF	Clipboard, file	The standard format for OS X. Best format for general use. Generated via the operating system and consequently more capable than Igor PDF.
LowRes PDF	Clipboard, file	Mostly just a compatibility placeholder for the legacy Macintosh PICT format. May have specialized uses but generally should not be used. Use Quartz PDF instead.
Igor PDF	Clipboard, file	PDF generated by Igor's own code rather than by the OS. May have specialized uses but generally should not be used. Use Quartz PDF instead.
Bitmap PICT	Clipboard, file	Legacy Macintosh-specific vector format. Largely obsolete as of Mac OS X.
		Resolution is 72 dpi. Expanded export can be used with some programs to simulate higher resolution.
EPS (Encapsulated Postscript)	File only	Platform-independent except for the screen preview.
		Supports high resolution.
		Useful only when printing on a PostScript printer, creating a PDF file, or exporting to PostScript-savvy program (e.g., Adobe Illustrator, Tex).
PNG (Portable Network Graphics)	Clipboard, file	Platform-independent bitmap format.
		Uses lossless compression. Supports high resolution.
JPEG	Clipboard, file	Platform-independent bitmap format.
		Uses lossy compression. Supports high resolution.
TIFF	Clipboard, file	Platform-independent bitmap format.
		Supports high resolution but not compression.
QuickTime Formats	File only	Additional bitmap formats are added by QuickTime, if installed. QuickTime formats appear in lower half of Format menu in Save Graphics dialog.

Macintosh PICT Format

PICT is the legacy pre-OS X Macintosh graphics format. In Igor Pro 6.1, it has been replaced by the PDF format. In an emergency, when you are exporting to an older program that does not support PDF (such as Microsoft Office prior to 2008,) you can cause Igor to revert to the old QuickDraw graphics mode. See **Graphics Technology** on page III-421 for details.

PDF Format

PDF (Portable Document Format) is Adobe's platform-independent vector graphics format that has been adopted by Apple as the standard graphics format for OS X. This is the best format as long as your destination program supports it.

The Quartz PDF format is generated by the operating system and consequently is more capable than Igor's existing native PDF generator. For example, it does a better job of embedding fonts and fully supports imported pictures.

The Igor PDF format is generated by Igor's own code rather than by the OS. Due to the following limitations, it should not be used unless you need to export in CMYK color mode (See **Exporting Colors** on page III-102 for details.) Limitations are:

- If the window contains any drawings imported into Igor from other programs, they will be rendered in the PDF as opaque bitmap images.
- You will need to pay attention to fonts. See **Font Embedding** on page III-102 for details.

Encapsulated PostScript (EPS) Format

Encapsulated PostScript is a widely-used, platform-independent vector graphics format consisting of PostScript commands in plain text form. It usually gives the best quality, but it works only when printed to a PostScript printer or exported to a PostScript-savvy program such as Adobe Illustrator. You should use only PostScript fonts (e.g., Helvetica).

Because PostScript is very complex, few programs can display it on screen. Consequently EPS includes an optional screen preview. In most programs, you need to print a document containing the EPS in order to see what you really have, as opposed to the preview that you see on screen.

On Macintosh, the EPS screen preview format is standard PICT, stored in the resource fork of the EPS file. If you transfer a Macintosh EPS file to Windows, the screen preview will be lost because Windows does not support the resource fork. So you will see a plain box instead of the preview on the screen of the Windows program, but the EPS should print correctly.

On Windows, the screen preview format is TIFF and is embedded in the EPS file. If you transfer it to Macintosh, the preview may or may not be understood, depending on the program into which you are importing. Embedding the TIFF preview makes the Windows EPS file a binary file, which is not editable with a text editor.

Some poorly written applications are confused by the screen preview. They ignore the EPS rules and use the size of the preview image rather than the PostScript bounding box, resulting in improper recreation of the EPS graphic. If you get unsatisfactory results, try using Igor's Suppress Preview option. The resulting EPS will display as a plain box in most programs but will print correctly.

EPS files normally use the RGB encoding to represent color but you can also use CMYK. See **Exporting Colors** on page III-102 for details.

Igor Pro exports EPS files using PostScript language level 2. This allows much better fill patterns when printing and also allows Adobe Illustrator to properly import Igor's fill patterns. For backwards compatibility with old printers, you can force Igor to use level 1 by specifying /PLL=1 with the SavePICT operation.

If the graph or page layout that you are exporting as EPS contains a non-EPS picture imported from another program, Igor exports the picture as an image incorporated in the output EPS file.

Igor Pro 5.02 and later can embed TrueType fonts as outlines. See **Font Embedding** on page III-102 for details.

Platform-Independent Bitmap Formats

PNG (Portable Network Graphics) is a platform-independent bitmap format that uses lossless compression and supports high resolution. It is a superior alternative to JPEG or GIF. Although Igor can export and import PNG via the Clipboard, not all programs can paste PNG from the Clipboard.

JPEG is a lossy image format whose main virtue is that it is accepted by all web browsers. However, all modern web browsers support PNG so there is little reason to use JPEG. Although Igor can export and import JPEG via the Clipboard, not all programs can paste JPEGs.

TIFF is an Adobe format often used for digital photographs. Igor's implementation of TIFF export does not use compression. TIFF files normally use the RGB scheme to specify color but you can also use CMYK. See **Exporting Colors** on page III-102 for details. There is no particular reason to use TIFF over PNG unless you are exporting to a program that does not support PNG. Igor can export and import TIFF via the Clipboard and most OS X programs can import TIFF.

A number of additional bitmap file formats are supported through Apple's QuickTime. The formats added by QuickTime are listed in the bottom half of the Format pop-up menu in the Save Graphics dialog.

Choosing a Graphics Format

Because of the wide variety of types of graphics, destination programs, printer capabilities, operating system behaviors and user-priorities, it is not possible to give definitive guidance on choosing an export format. But here is an approach that will work in most situations.

If the destination will accept PDF, then that is probably your best choice because of its high-quality vector graphics and platform-independence.

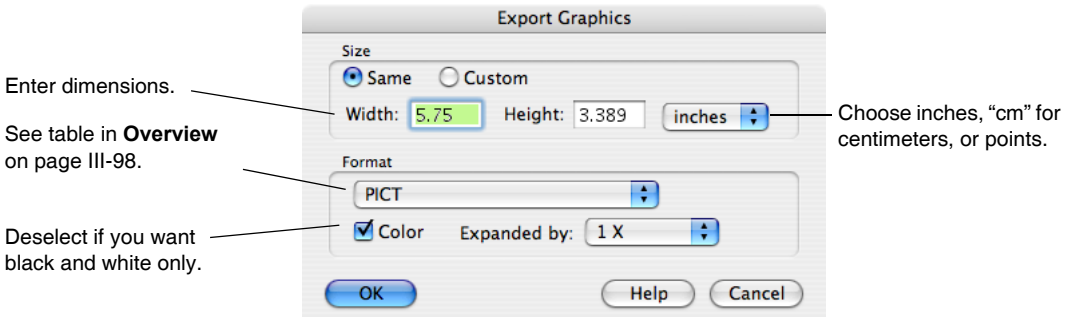
Encapsulated PostScript (EPS) is also a very high quality format and is the format of choice if you are:

- Exporting to a word processor for printing on a PostScript printer.
- Exporting to a word processor for creating a PDF file.
- Exporting to a PostScript-savvy drawing program such as Adobe Illustrator.

If EPS and PDF are not appropriate, your next choice would be a high-resolution bitmap. The PNG format is preferred because it is platform-independent and is compressed. If the application to which you are exporting does not support PNG, your next choice would be TIFF or JPEG.

Exporting Graphics Via the Clipboard

To export a graphic from the active graph, page layout or table window via the Clipboard, choose Edit→Export Graphics. This displays the Export Graphics dialog. For a graph, it looks like this:



When you click the OK button, Igor will copy the graph, page layout, or table to the Clipboard. You can then switch to another program and do a paste.

For a page layout or table, the dialog is the same except that you can't enter the width and height.

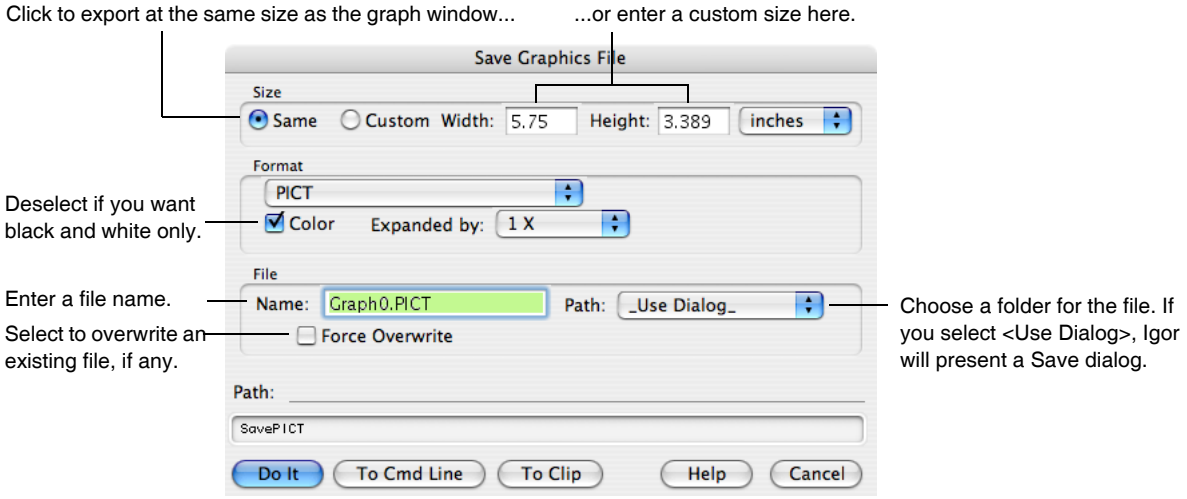
When a graph or page layout is active and in operate mode, choosing Edit→Copy copies to the Clipboard whatever format was last used in the Export Graphics dialog. (This is new as of Igor Pro 6.01. Previously the format was always a 1X standard vector PICT.) For a table, Edit→Copy copies the selected numbers to the Clipboard and does not copy graphics.

When a page layout has an object selected or when the marquee is active, choosing Edit→Copy copies an Igor object in a format used internally by Igor along with a 1x standard vector PICT and does not use the format from the Export Graphics dialog

Igor can export PNG images to the Clipboard and can then paste them back in. On the Macintosh, the Clipboard type is `'PNGf'` but because there is no standard for PNG on the Clipboard it is therefore unlikely that other programs can import them except as files.

Exporting Graphics Via a File

To export a graphic from the active graph, page layout or table window via a file, choose File→Save Graphics. The Save Graphics File dialog for a graph looks like this:



The controls in the Format area of the dialog change to reflect options appropriate to each export format.

When you click the Do It button, Igor will write the graphic to a file. You can then switch to another program and import the file.

If you select <Use Dialog> from the Path list, Igor will present a Save dialog from which you can choose a folder in which to save the file.

The controls in the Format area of the dialog change to reflect options appropriate to each export format.

Exporting a Graphic File for Transfer to a Windows Computer

The best method for transferring Igor graphics to a Windows computer is to transfer the entire Igor experiment file, open it in Igor for Windows, and export the graphic via one of the Windows-compatible methods available in Igor for Windows.

If your graph or layout contains embedded pictures in PDF or PICT format, you will need to convert them to the cross-platform format, PNG, because PDFs and Macintosh PICTs are not displayed in Igor for Windows. If any embedded pictures are in JPEG, TIFF, or EPS formats, these will work without conversion. See Chapter III-15, **Platform-Related Issues**, especially the section **Picture Compatibility** on page III-395.

If you don't have a copy of Igor for Windows available, you have these choices:

1. Export PDF if the destination program can accept it.
2. Export an EPS file. This works only if the Windows program can import EPS files, and requires that it be printed on a PostScript printer.
   Furthermore, the PICT screen preview that Igor puts into a Macintosh EPS file is not displayable on Windows. Use the Suppress Preview checkbox to eliminate the preview. This will render the graphic unviewable on-screen, but it will still print (on a PostScript printer) just fine.
3. Export a PNG.

Exporting a Section of a Layout

To export a section of a page layout, use the marquee tool to identify the section and then choose Export Graphics (Edit menu) or Save Graphics (File menu). If you don't use the marquee, Igor exports the area of the layout that is in use plus a small margin.

Exporting Colors

The PDF, EPS and TIFF graphics formats normally use the RGB scheme to specify color. Some publications require the use of CMYK instead of RGB, although the best results are obtained if the publisher does the RGB to CMYK conversion using the actual characteristics of the output device. For those publications that insist on CMYK, you can use the SavePICT /C=2 flag

Exporting and Printing Graphs of Large Data Sets

When exporting or printing an image plot, Igor normally uses a fast algorithm but has to resort to a much slower, memory-intensive algorithm if the image contains holes (the displayed matrix wave contains NaNs) or if it is displayed on a nonuniform grid (you specified X and Y coordinate waves when creating the plot).

The best method of exporting a graph for placement in another program is normally a vector format such as HiRes PICT or EPS. However, graphs containing large images or very large waveforms may take a long time to export or print. You may be able to solve this by using a high-resolution bitmap format for such graphs. This has the added advantage of working well with non-PostScript printers.

Graphs in Page Layouts

If you discover that a page layout is taking much too long to print, you can print graph objects in the layout layer of the layout using a high-resolution bitmap rather than the usual vector method. Use this only in an emergency when a printer driver has bugs that affect normal operations and when printing graphs with very large numbers of data points. There are drawbacks to the bitmap method. A large amount of memory will be needed and patterns will be too small to be useful. Also, the quality of lines (especially dashed lines) may be degraded.

To force Igor to print graph objects in a page layout using the bitmap method, execute the following on the command line:

```
Variable/G root:V_PrintUsingBitmap= 1
```

You may want to set this variable to zero after printing a problem layout so as not to affect other layouts in the same experiment.

Font Embedding

You can embed TrueType fonts in EPS files and in PDF files. This means you can print EPS or PDF files on systems lacking the equivalent PostScript fonts. This also helps for publications that require embedded fonts.

Font embedding is done automatically for the Quartz PDF format and you do not need to bother with this section unless you are using EPS or Igor PDF formats.

There are three levels of font embedding: No embedding, embed only nonstandard fonts, and embed all fonts. For most purposes, embed only nonstandard fonts is the best choice.

Igor embeds TrueType fonts as synthetic PostScript Type 3 fonts derived from the TrueType font outlines. Only the actual characters used are included in the fonts and only single byte fonts are supported.

You should not use font embedding if you plan on exporting to a drawing program such as Adobe Illustrator and wish to edit the text in that program.

Not all fonts and font styles on your system can be embedded. Some fonts may not allow embedding and others may not be TrueType or may give errors. Be sure to test your EPS files on a local printer or by import-

ing into Adobe Illustrator before sending them to your publisher. You can test your PDF files with Adobe Reader. You can also use the "TrueType Outlines.pxp" example experiment to validate fonts for embedding. You will find this experiment file in your Igor Pro Folder in the "Examples:Testing & Misc:" folder.

For EPS, the "embed only nonstandard fonts" method determines if a font is nonstandard by attempting to look up the font name in the TTPSFNames table described in **PostScript Font Names (OS X)** on page III-103 after doing any font substitution using the TTtoPS table. In addition, if a nonplain font style name is the same as the plain font name, then embedding is done. This means that standard PostScript fonts that do not come in italic versions (such as Symbol), will be embedded for the italic case but not for the plain case.

For PDF, "embed only nonstandard fonts" embeds fonts other than the basic fonts guaranteed by the PDF specification to be built-in to any PDF reader. Those fonts are Helvetica and TImes in plain, bold, italic and bold-italic forms as well as Symbol and Zapf Dingbats only in plain style. If embedding is not used or if a font can not be embedded, fonts other than those just listed will be rendered as Helvetica and will not give the desired results

PostScript Font Names (OS X)

When generating PostScript, Igor needs to generate proper PostScript font names. This presents problems under Macintosh OS X. Igor also needs to be able to substitute PostScript fonts for non-PostScript fonts.

If you use only the basic fonts in the following table or if you use **Font Embedding** on page III-102, then you do not have to read any further. Igor has built-in PostScript font names for these as well as a built-in translation table that you use to specify standard system TrueType fonts but get proper PostScript fonts when exporting. The built-in name translations are:

TrueType Name	PostScript Name
Helvetica	Helvetica
Arial	Helvetica
Helvetica-Narrow	Helvetica-Narrow
Arial Narrow	Helvetica-Narrow
Palatino	Palatino
Book Antiqua	Palatino
Bookman	Bookman
Bookman Old Style	Bookman
Avant Garde	AvantGarde
Century Gothic	AvantGarde
New Century Schlbk	NewCenturySchlbk
Century Schoolbook	NewCenturySchlbk
Courier	Courier
Courier New	Courier
Zapf Chancery	ZapfChancery
Monotype Corsiva	ZapfChancery
Zapf Dingbats	ZapfDingbats
Monotype Sorts	ZapfDingbats
Symbol	Symbol

TrueType Name	PostScript Name
Times	Times
Times New Roman	Times

If you want to use fonts that are not in this table then you need to customize Igor's table as follows.

1. Open the experiment containing the graphic you wish to export and then create an EPS file. Igor will print in the history warnings about any fonts that are not in the table. The first time an EPS is generated from a given experiment, Igor creates a data folder containing a pair of text waves containing the names from the built-in table.
2. Using the Data Browser (Data Menu), navigate to "root:Packages:PSFontInfo:". You will see two waves named TTtoPS and TTPSFNames. Double click the icons to open the waves in a table. You will need to edit one or both of these tables.
3. Edit the TTtoPS wave to add a row at the bottom of the table that provides the screen font name on the left and the base PostScript font name on the right. If they are the same, you don't need to do this step.
4. Edit the TTPSFNames wave to add a row at the bottom of the table that provides the base PostScript font name in column 0, the normal font name in column 1, the bold name in column 2, the italic name in column 3 and the bold-italic name in column 4.

If you don't know the proper PostScript font names for the font you wish to use, you may be able to find this information in a file named 5090.fontnamelist.pdf on Adobe's web site.

This technique adds names only to the current experiment. If you want all experiments to have access to a set of names, you can create (or edit) a tab-delimited text file named UserFontNames.txt in "Igor Pro User Files/Igor Extensions" (see **Igor Pro User Files** on page II-46 for details). It must have the same structure as the TTPSFNames wave just described but should contain only the new fonts you are adding. Your screen font name should match the PostScript base name. If it doesn't, you may need to add an entry in the TTtoPS translation wave described above.

Igor gets the TrueType to PostScript translation table from the operating system if it is available. When you add a new font to your system, the installation program may update the system translation table. Because of this, the TTtoPS wave may contain more or different entries than described above. If you install a new font, you can force Igor to update the TTtoPS wave in a given experiment by deleting or renaming the wave and then writing out a dummy EPS file.

In addition to the UserFontNames.txt in the Igor Extensions folder, Igor also looks for a file named PSFontNames.txt. This file, if present, contains additional font names provided by WaveMetrics to extend Igor's built-in table. Although you can use this table rather than UserFontNames.txt you probably should not since your changes will be lost the next time a new version of Igor Pro is released.

Chapter
III-6

Exporting Graphics (Windows)

Overview

This chapter discusses exporting graphics from Igor graphs, page layouts and tables to another program on Windows. You can export graphics through the Clipboard by choosing Edit→Export Graphics, or through a file, by choosing File→Save Graphics.

Igor Pro supports a number of different graphics export formats. You can usually obtain very good results by choosing the appropriate format, which depends on the nature of your graphics, your printer and the characteristics of the program to which you are exporting.

Unfortunately, experimentation is sometimes required to find the best export format for your particular circumstances. This section provides the information you need to make an informed choice.

This table shows the available graphic export formats:

Export Format	Export Method	Notes
EMF (Enhanced Metafile)	Clipboard, file	Windows-specific vector format.
BMP (Bitmap)	Clipboard, file	Windows-specific bitmap format.
		Does not use compression.
PDF	Clipboard, file	Platform-independent and high quality.
EPS (Encapsulated Postscript)	File only	Platform-independent except for the screen preview.
		Supports high resolution.
		Useful only when printing on PostScript printer, creating a PDF file or exporting to PostScript-savvy program (e.g., Adobe Illustrator, Tex).
PNG (Portable Network Graphics)	Clipboard, file	Platform-independent bitmap format.
		Uses lossless compression. Supports high resolution.
JPEG	Clipboard, file	Platform-independent bitmap format.
		Uses lossy compression. Supports high resolution.
TIFF	Clipboard, file	Platform-independent bitmap format.
		Supports high resolution but not compression.
QuickTime Formats	File only	Additional formats are added by QuickTime, if installed. QuickTime formats appear in lower half of Format menu in Save Graphics dialog.

Metafile Formats

The metafile formats are Windows vector graphics formats that support drawing commands for the individual objects such as lines, rectangles and text that make up a picture. Drawing programs can decompose a metafile into its component parts to allow editing the individual objects. Most word processing programs treat a metafile as a black box and call the operating system to display or print it.

WMF is an obsolete format and is not produced by Igor when running in the default advanced graphics mode. In an emergency, when you are exporting to an older program that does not support EMF, you can cause Igor to revert to the old graphics mode. See **Graphics Technology** on page III-421 for details.

Enhanced Metafile (EMF) is the primary Windows-native graphics format. EMF is easy to use because nearly all Windows programs can import it and because it can be copied to the Clipboard as well as written to a file. Some programs, notably some older versions of Microsoft Office, require that you choose Paste Special rather than Paste to paste an EMF from the Clipboard.

Although drawing programs can decompose an EMF into its component parts to allow editing the individual objects, they often get it wrong due to the complexity of the metafile format. The default advanced

graphics mode introduced in Igor Pro 6.1 may be especially stressful and you may find it necessary to revert to the old graphics mode. See **Graphics Technology** on page III-421 for details.

BMP Format

BMP is a Windows bitmap format. It is accepted by a wide variety of programs but requires a lot of memory and disk space because it is not compressed. A BMP is also known as a DIB (device-independent bitmap).

If the program to which you are exporting supports PNG then PNG is a better choice.

PDF Format

PDF (Portable Document Format) is Adobe's platform-independent vector graphics format. However, not all programs can import PDF. In fact, although Igor can export PDF, it can not itself import PDF.

If a window contains drawings imported into Igor from other programs, they will be rendered in the PDF as opaque bitmap images.

PDF files normally use the RGB encoding to represent color but you can also use CMYK. See **Exporting Colors** on page III-110 for details.

PDF export can (and should) embed TrueType fonts. See **Font Embedding** on page III-110 for details.

Encapsulated PostScript (EPS) Format

Encapsulated PostScript is a widely-used, platform-independent vector graphics format consisting of PostScript commands in plain text form. It usually gives the best quality, but it works only when printed to a PostScript printer or exported to a PostScript-savvy program such as Adobe Illustrator. You should use only PostScript fonts (e.g., Helvetica).

Because PostScript is very complex, few programs can display it on screen. Consequently EPS includes an optional screen preview. In most programs, you need to print a document containing the EPS in order to see what you really have, as opposed to the preview that you see on screen.

On Macintosh, the EPS screen preview format is standard PICT, stored in the resource fork of the EPS file. If you transfer a Macintosh EPS file to Windows, the screen preview will be lost because Windows does not support the resource fork. So you will see a plain box instead of the preview on the screen of the Windows program, but the EPS should print correctly.

On Windows, the screen preview format is TIFF and is embedded in the EPS file. If you transfer it to Macintosh, the preview may or may not be understood, depending on the program into which you are importing. Embedding the TIFF preview makes the Windows EPS file a binary file, which is not editable with a text editor.

Some poorly written applications are confused by the screen preview. They ignore the EPS rules and use the size of the preview image rather than the PostScript bounding box, resulting in improper recreation of the EPS graphic. If you get unsatisfactory results, try using Igor's Suppress Preview option. The resulting EPS will display as a plain box in most programs but will print correctly.

EPS files normally use the RGB encoding to represent color but you can also use CMYK. See **Exporting Colors** on page III-110 for details.

Igor Pro exports EPS files using PostScript language level 2. This allows much better fill patterns when printing and also allows Adobe Illustrator to properly import Igor's fill patterns. For backwards compatibility with old printers, you can force Igor to use level 1 by specifying /PLL=1 with the SavePICT operation.

If the graph or page layout that you are exporting as EPS contains a non-EPS picture imported from another program, Igor exports the picture as an image incorporated in the output EPS file.

Igor Pro 5.02 and later can embed TrueType fonts as outlines. See **Font Embedding** on page III-110 for details.

Platform-Independent Bitmap Formats

PNG (Portable Network Graphics) is a platform-independent bitmap format. It uses lossless compression and supports high resolution. It is a superior alternative to JPEG or GIF. Although Igor can export and import PNG via the Clipboard, not all programs can paste PNG from the Clipboard.

JPEG is a lossy format whose main virtue is that it is accepted by all web browsers. However, all modern web browsers support PNG so there is little reason to use JPEG. Although Igor can export and import JPEG via the Clipboard, not all programs can paste JPEGs.

TIFF is an Adobe format often used for digital photographs. Igor's implementation of TIFF export does not use compression. TIFF files normally use the RGB scheme to specify color but you can also use CMYK. See **Exporting Colors** on page III-110 for details. There is no particular reason to use TIFF over PNG unless you are exporting to a program that does not support PNG. Igor can export and import TIFF via the Clipboard and most OS X programs can import TIFF.

A number of additional bitmap file formats are supported through Apple's QuickTime, if it is installed on your system. The formats added by QuickTime are listed in the bottom half of the Format pop-up menu in the Save Graphics dialog.

Choosing a Graphics Format

Because of the wide variety of types of graphics, destination programs, printer capabilities, operating system behaviors and user-priorities, it is not possible to give definitive guidance on choosing an export format. But here is an approach that will work in most situations.

If the destination will accept PDF, then that is probably your best choice because of its high-quality vector graphics and platform-independence.

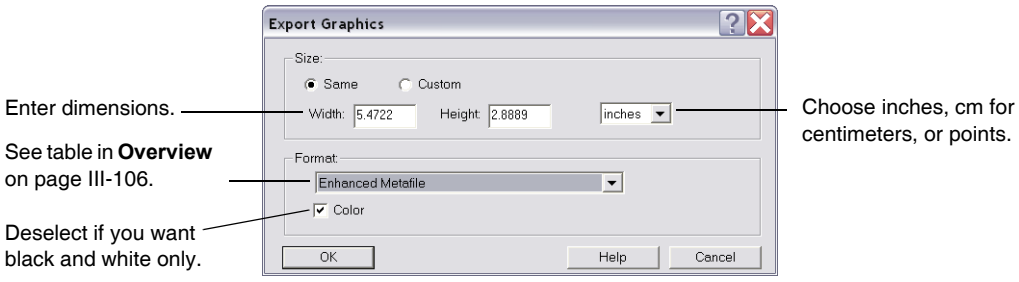
Encapsulated PostScript (EPS) is also a very high quality format and is the format of choice if you are:

- Exporting to a word processor for printing on a PostScript printer.
- Exporting to a word processor for creating a PDF file.
- Exporting to a PostScript-savvy drawing program such as Adobe Illustrator.

If EPS and PDF are not appropriate, your next choice would be a high-resolution bitmap. The PNG format is preferred because it is platform-independent and is compressed. If the application to which you are exporting does not support PNG, your next choice would be TIFF or JPEG.

Exporting Graphics Via the Clipboard

To export a graphic from the active graph, page layout or table window via the Clipboard, choose Edit→Export Graphics. This displays the Export Graphics dialog. For a graph, it looks like this:



When you click the OK button, Igor will copy the graph, page layout or table to the Clipboard. You can then switch to another program and do a paste.

From the Format pop-up menu, you can choose Windows Metafile, Enhanced Metafile, Bitmap, PostScript Enhanced Metafile and PNG Image.

If Bitmap or PNG Image is chosen, then you can select a resolution.

For a page layout or table, the dialog is the same except that you can't enter the width and height. Windows Metafile Format (WMF) is not available for page layouts.

To paste an enhanced metafile, some versions of Microsoft Office require that you choose Paste Special instead of Paste. If you just choose Paste, the enhanced metafile is converted into a less capable Windows metafile.

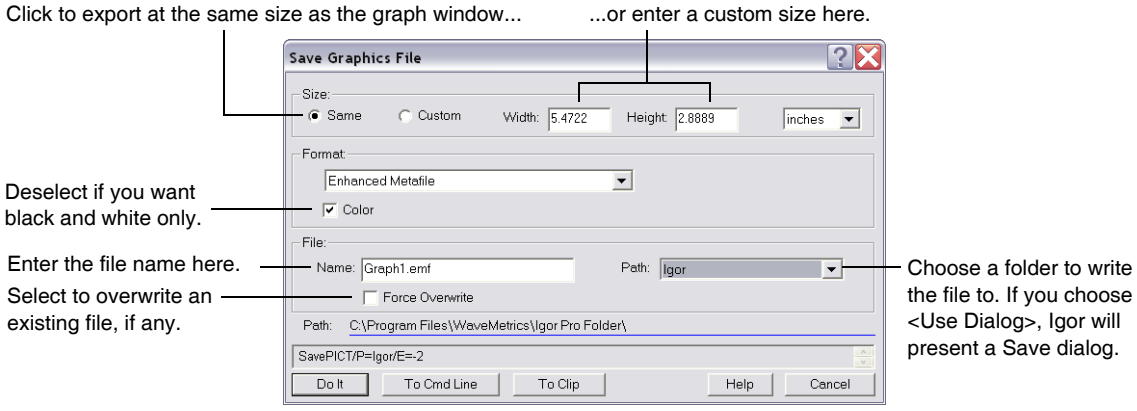
When a graph or page layout is active and in operate mode, choosing Edit→Copy copies to the Clipboard whatever format was last used in the Export Graphics dialog. (This is new as of Igor Pro 6.01. Previously the format was always an enhanced metafile.) For a table, Edit→Copy copies the selected numbers to the Clipboard and does not copy graphics.

When a page layout has an object selected or when the marquee is active, choosing Edit→Copy copies an Igor object in a format used internally by Igor along with an enhanced metafile and does not use the format from the Export Graphics dialog.

Igor can export PNG images to the Clipboard and can then paste them back in. Igor exports PNG images under Windows by registering the format with the name "PNG" which is supported by some Microsoft products. Igor can also export PDF, JPEG and TIFF formats to the Clipboard (registered as "PDF", "JPEG" and "TIFF") but few if any programs can import these

Exporting Graphics Via a File

To export a graphic from the active graph, page layout or table window via a file, choose File→Save Graphics. The Save Graphics dialog for a graph looks like this:



The controls in the Format area of the dialog change to reflect options appropriate to each export format.

When you click the Do It button, Igor will write a graphics file. You can then switch to another program and import the file.

If you select _Use Dialog_ from the Path pop-up menu, Igor will present a Save dialog from which you can choose a folder in which to save the file.

You may also be able to create an EPS file using a PostScript printer driver. Igor's Save EPS File routine produces smaller files than the printer driver and should be used in most cases. However, if your graph or layout includes non-EPS graphics imported from other programs, you may get better results using the printer driver because Igor renders such imported graphics as images rather than object graphics.

Exporting a Section of a Layout

To export a section of a page layout, use the marquee tool to identify the section and then choose Edit→Export Graphics or File→Save Graphics. If you don't use the marquee, Igor exports the area of the layout that is in use plus a small margin.

Exporting Colors

The EPS and TIFF graphics formats normally use the RGB scheme to specify color. Some publications require the use of CMYK instead of RGB, although the best results are obtained if the publisher does the RGB to CMYK conversion using the actual characteristics of the output device. For those publications that insist on CMYK, you can use the SavePICT /C=2 flag.

Exporting and Printing Graphs of Large Data Sets

When exporting a graph as Encapsulated PostScript or printing a graph on a PostScript printer, Igor or the printer driver must convert the Windows graphics language into PostScript. Then the PostScript printer must convert the PostScript into dots on a page. When dealing with very large data sets the conversion to dots in the printer can be *very* slow.

When exporting or printing an image plot, Igor normally uses a fast algorithm but has to resort to a much slower, memory-intensive algorithm if the image contains holes (the displayed matrix wave contains NaNs) or if it is displayed on a nonuniform grid (you specified X and Y coordinate waves when creating the plot).

The best method of exporting a graph for placement in another program is normally a vector format such as EMF or EPS. However, graphs containing large images or very large waveforms may take a long time to export or print. You may be able to solve this by using a high-resolution bitmap format for such graphs. This has the added advantage of working well with non-PostScript printers.

Graphs in Page Layouts

If you discover that a page layout is taking much too long to print, you can print graph objects in the layout layer of the layout using a high-resolution bitmap rather than the usual vector method. Use this only in an emergency when a printer driver has bugs that affect normal operations and when printing graphs with very large numbers of data points. There are drawbacks to the bitmap method. A large amount of memory will be needed and patterns will be too small to be useful. Also, the quality of lines (especially dashed lines) may be degraded.

To force Igor to print graph objects in a page layout using the bitmap method, execute the following on the command line:

```
Variable/G root:V_PrintUsingBitmap= 1
```

You may want to set this variable to zero after printing a problem layout so as not to affect other layouts in the same experiment.

Font Embedding

As of Igor Pro 5.02, you can now embed TrueType fonts in EPS files. This means you can print EPS files on systems lacking the equivalent PostScript fonts. This also helps for publications that require fonts to be embedded.

There are three levels of font embedding: No embedding, embed only nonstandard fonts, and embed all fonts. For most purposes, embed only nonstandard fonts is the best choice.

Igor embeds TrueType fonts as synthetic PostScript Type 3 fonts derived from the TrueType font outlines. Only the actual characters used are included in the fonts and only single byte fonts are supported.

You should not use font embedding if you plan on exporting to a drawing program such as Adobe Illustrator and wish to edit the text in that program.

Not all fonts and font styles on your system can be embedded. Some fonts may not allow embedding and others may not be TrueType or may give errors. Be sure to test your EPS files on a local printer or by importing into Adobe Illustrator before sending them to your publisher. You can also use the "TrueType Outlines.pxp" example experiment to validate fonts for embedding. You will find this experiment file in your Igor Pro Folder in the "Examples:Testing & Misc:" folder.

For EPS, the "embed only nonstandard fonts" method determines if a font is nonstandard by attempting to look up the font name in the TTPSFNames table described in **PostScript Font Names** on page III-111 after doing any font substitution using the TTtoPS table. In addition, if a nonplain font style name is the same as the plain font name, then embedding is done. This means that standard PostScript fonts that do not come in italic versions (such as Symbol), will be embedded for the italic case but not for the plain case.

For PDF, "embed only nonstandard fonts" embeds fonts other than the basic fonts guaranteed by the PDF specification to be built-in to any PDF reader. Those fonts are Helvetica and TImes in plain, bold, italic, and bold-italic styles as well as plain versions of Symbol and Zapf Dingbats. If embedding is not used or if a font can not be embedded, fonts other than those just listed will be rendered as Helvetica and will not give the desired results.

PostScript Font Names

When generating PostScript, Igor needs to generate proper PostScript font names. This presents problems under Windows. Igor also needs to be able to substitute PostScript fonts for non-PostScript fonts.

If you use only the basic fonts in the following table or if you use **Font Embedding** on page III-110, then you do not have to read any further. Igor has built-in PostScript font names for these as well as a built-in translation table that you use to specify standard system TrueType fonts but get proper PostScript fonts when exporting. The built-in name translations are:

TrueType Name	PostScript Name
Arial	Helvetica
Arial Narrow	Helvetica-Narrow
Book Antiqua	Palatino
Bookman Old Style	Bookman
Century Gothic	AvantGarde
Century Schoolbook	NewCenturySchlbk
Courier New	Courier
Monotype Corsiva	ZapfChancery
Monotype Sorts	ZapfDingbats
Symbol	Symbol
Times New Roman	Times

If you want to use fonts that are not in this table then you need to customize Igor's table as follows.

1. Open the experiment containing the graphic you wish to export and then create an EPS file. Igor will print in the history warnings about any fonts that are not in the table. The first time an EPS is generated from a given experiment, Igor creates a data folder containing a pair of text waves containing the names from the built-in table.
2. Using the Data Browser (Data Menu), navigate to root:Packages:PSFontInfo:. You will see two waves named TTtoPS and TTPSFNames. Double click the icons to open the waves in a table. You

will need to edit one or both of these tables.

3. Edit the TTtoPS wave to add a row at the bottom of the table that provides the screen font name on the left and the base PostScript font name on the right. If they are the same, you don't need to do this step.
4. Edit the TTPSFNames wave to add a row at the bottom of the table that provides the base PostScript font name in column 0, the normal font name in column 1, the bold name in column 2, the italic name in column 3 and the bold-italic name in column 4.

If you don't know the proper PostScript font names for the font you wish to use, you may be able to find this information in a file named 5090.fontnamelist.pdf on Adobe's web site.

This technique adds names only to the current experiment. If you want all experiments to have access to a set of names, you can create (or edit) a tab-delimited text file named UserFontNames.txt in "Igor Pro User Files\Igor Extensions" (see **Igor Pro User Files** on page II-46 for details). It must have the same structure as the TTPSFNames wave just described but should contain only the new fonts you are adding. Your screen font name should match the PostScript base name. If it doesn't, you may need to add an entry in the TTtoPS translation wave described above.

Igor gets the TrueType to PostScript translation table from the operating system if it is available. When you add a new font to your system, the installation program may update the system translation table. Because of this, the TTtoPS wave may contain more or different entries than described above. If you install a new font, you can force Igor to update the TTtoPS wave in a given experiment by deleting or renaming the wave and then writing out a dummy EPS file.

In addition to the UserFontNames.txt in the Igor Extensions folder, Igor also looks for a file named PSFont-Names.txt. This file, if present, contains additional font names provided by WaveMetrics to extend Igor's built-in table. Although you can use this table rather than UserFontNames.txt you probably should not since your changes will be lost the next time a new version of Igor Pro is released.

Chapter
III-7

Analysis

Overview

Igor Pro is a powerful data analysis environment. The power comes from a synergistic combination of

- An extensive set of basic built-in analysis operations
- A fast and flexible waveform arithmetic capability
- Immediate feedback from graphs and tables
- Extensibility through an interactive programming environment
- Extensibility through external code modules (XOPs and XFUNCs)

Analysis tasks in Igor range from simple experiments using no programming to extensive systems tailored for specific fields. Chapter I-2, **Guided Tour of Igor Pro**, shows examples of the former. WaveMetrics' "Peak Measurement" procedure package is an example of the latter.

This chapter presents some of the basic analysis operations and discusses the more common analyses that can be derived from the basic operations. The end of the chapter shows a number of examples of using Igor's programmability for "number crunching".

Discussion of Igor Pro's more specialized analytic capabilities is in chapters that follow.

See the WaveMetrics procedures, technical notes, and sample experiments that come with Igor Pro for more examples.

Analysis of Multidimensional Waves

Many of the analysis operations in Igor Pro operate on 1D (one-dimensional) data. However, Igor Pro does includes the following capabilities for analysis of multidimensional data:

- Multidimensional waveform arithmetic
- Matrix math operations
- Multidimensional Fast Fourier Transform
- 2D and 3D image processing operations
- 2D and 3D interpolation operations and functions

Some of these topics are discussed in Chapter II-6, **Multidimensional Waves** and in Chapter III-11, **Image Processing**. The present chapter focuses on analysis of 1D waves.

There are many analysis operations that are designed only for 1D data. Multidimensional waves will not appear in dialogs for these operations. If you invoke them on multidimensional waves from the command line or from an Igor procedure, Igor may treat the multidimensional waves as if they were 1D. For example, the Histogram operation will treat a 2D wave consisting of *n* rows and *m* columns as if it were a 1D wave with *n***m* rows. In some cases (e.g., WaveStats), the operation will be useful. In other cases, it will make no sense at all.

Waveform Versus XY Data

Igor is highly adapted for dealing with waveform data. In a waveform, data values are uniformly spaced in the X dimension. This is discussed under **Waveform Model of Data** on page II-77.

If your data is uniformly spaced, you can set the spacing using the SetScale operation. This is crucial because most of the built-in analysis operations and functions need to know this to work properly.

If your data is not uniformly spaced, you can represent it using an XY pair of waves. This is discussed under **XY Model of Data** on page II-78. Some of the analysis operations and functions in Igor can *not* handle XY pairs directly. To use these, you must either make a waveform representation of the XY pair or use Igor procedures that build on the built-in routines.

Converting XY Data to a Waveform

Sometimes the best way to analyze XY data is to make a uniformly-spaced waveform representation of it and analyze that instead. Most analysis operations are easier with waveform data. Other operations, such as the FFT, can be done *only* on waveform data. Often your XY data set is nearly uniformly-spaced so a waveform version of it is a very close approximation.

Often your XY data set is nearly uniformly-spaced so a waveform version of it is a very close approximation.

In fact, often XY data imported from other programs has an X wave that is completely unnecessary in Igor because the values in the X wave are actually a simple "series" (values that define a regular intervals, such as 2.2, 2.4, 2.6, 2.8, etc), in which case conversion to a waveform is a simple matter of assigning the correct X scaling to the Y data wave, using **SetScale** (or the Change Wave Scaling dialog):

```
SetScale/P x, xWave[0], xWave[1]-xWave[0], yWave
```

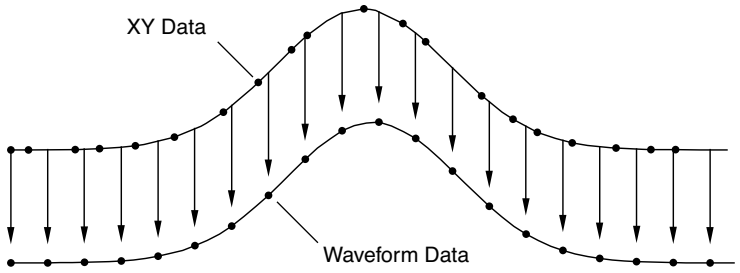
Now the X wave is superfluous and can be discarded:

```
KillWaves/Z xWave
```

The XY Pair to Waveform panel can be used to set the Y wave's X scaling when it detects that the X wave contains series data. The panel can be rather picky about what it considers a series, though, so you may need to use the SetScale command or dialog when the series is of low accuracy due to truncation or rounding in the original data set. See **Using the XY Pair to Waveform Panel** on page III-117.

If your X wave is not a series, then to create a waveform representation of XY data you need to use interpolation. To create a waveform representation of XY data you need to do interpolation. Interpolation creates a waveform from an XY pair by sampling the XY pair at uniform intervals. The diagram below shows how the XY pair defining the upper curve is interpolated to compute the uniformly-spaced waveform defining the lower curve.

Each arrow indicates an interpolated waveform value:



Igor provides three tools for doing this interpolation: The XY Pair to Waveform panel, the built-in **interp** function and the Interpolate external operation. To illustrate these tools we need some sample XY data. The following commands make sample data and display it in a graph:

```
Make/N=100 xData = .01*x + gnoise(.01)
Make/N=100 yData = 1.5 + 5*exp(-((xData-.5)/.1)^2)
Display yData vs xData
```

This creates a Gaussian shape. The x wave in our XY pair has some noise in it so the data is not uniformly spaced in the X dimension.

The x data goes roughly from 0 to 1.0 but, because our x data has some noise, it may not be monotonic. This means that, as we go from one point to the next, the x data usually increases but at some points may decrease. We can fix this by sorting the data.

```
Sort xData, xData, yData
```

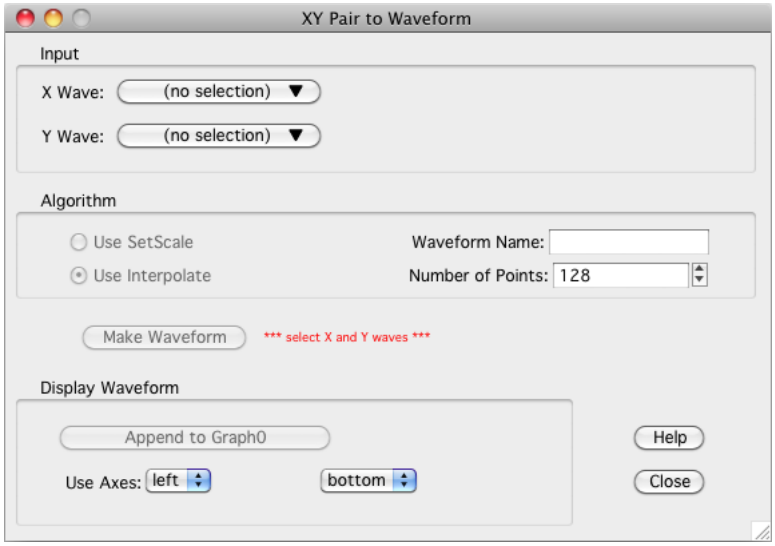
This command uses the xData wave as the sort key and sorts both xData and yData so that xData always increases as we go from one point to the next.

Using the XY Pair to Waveform Panel

The XY Pair to Waveform panel creates a waveform from XY data using the **SetScale** or Interpolate2 operations, based on an automatic analysis of the X wave's data.

The required steps are:

1. Select XY Pair to Waveform from Igor's Data→Packages submenu.
   The panel is displayed:



2. Select the X and Y waves (xData and yData) in the popup menus. When this example's xData wave is analyzed it is found to be "not regularly spaced (slope error avg= 0.52...)", which means that SetScale is not appropriate for converting yData into a waveform.
3. Use Interpolate is selected here, so you need a waveform name for the output. Enter any valid wave name.
4. Set the number of output points. Using a number roughly the same as the length of the input waves is a good first attempt. You can choose a larger number later if the fidelity to the original is insufficient. A good number depends on how uneven the X values are - use more points for more unevenness.
5. Click Make Waveform.
6. To compare the XY representation of the data with the waveform representation, append the waveform to a graph displaying the XY pair. Make that graph the top graph, then click the "Append to <Name of Graph>" button.
7. You can revise the Number of Points and click Make Waveform to overwrite the previously created waveform in-place.

Using the Interp Function

We can use the **interp** function (see page V-316) to create a waveform version of our Gaussian. The required steps are:

1. Make a new wave to contain the waveform representation.
2. Use the **SetScale** operation to define the range of X values in the waveform.
3. Use the interp function to set the data values of the waveform based on the XY data.

Here are the commands:

```
Duplicate yData, wData
SetScale/I x 0, 1, wData
wData = interp(x, xData, yData)
```

To compare the waveform representation to the XY representation, we append the waveform to the graph.

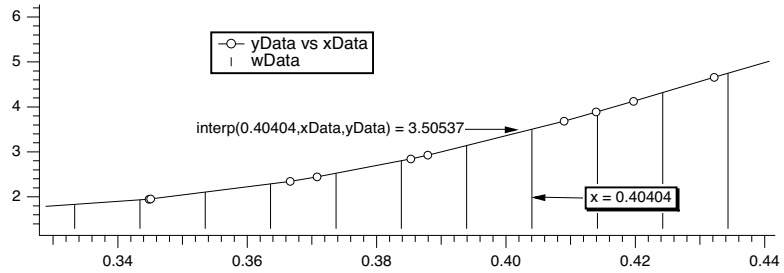
```
AppendToGraph wData
```

Let's take a closer look at what these commands are doing.

First, we cloned yData and created a new wave, wData. Since we used Duplicate, wData will have the same number of points as yData. We could have made a waveform with a different number of points. To do this, we would use the Make operation instead of Duplicate.

The SetScale operation sets the X scaling of the wData waveform. In this example, we are setting the X values of wData to go from 0 up to and including 1.0. This means that our waveform representation will contain 100 values at uniform intervals in the X dimension from 0 to 1.0.

The last step uses a waveform assignment to set the data values of wData. This assignment evaluates the right-hand expression once for each point in wData. For each evaluation, x takes on a different value from 0 to 1.0. The interp function returns the value of the curve yData versus xData at x. For instance, x=.40404 (point number 40 of wData) falls between two points in the XY curve. The interp function linearly interpolates between those values to estimate a data value of 3.50537:



We can wrap these calculations up into an Igor procedure that can create a waveform version of any XY pair.

```
Function XYToWave1(xWave, yWave, wWaveName, numPoints)
   Wave/D xWave                // X wave in the XY pair
   Wave/D yWave                // Y wave in the XY pair
   String wWaveName            // Name to use for new waveform wave
   Variable numPoints          // Number of points for waveform

   Make/O/N=(numPoints) $wWaveName  // Make waveform.
   Wave wWave= $wWaveName
   WaveStats/Q xWave                // Find range of x coords.
   SetScale/I x V_min, V_max, wWave // Set X scaling for wave.
   wWave = interp(x, xWave, yWave)  // Do the interpolation.
End
```

This function uses the **WaveStats** operation to find the X range of the XY pair. WaveStats creates the variables V_min and V_max (among others). See **Accessing Variables Used by Igor Operations** on page IV-103 for details.

The function makes the assumption that the input waves are already sorted. We left the sort step out because the sorting would be a side-effect and we prefer that procedures not have nonobvious side effects.

To use the WaveMetrics-supplied XYToWave1 function, include the "XY Pair To Waveform" procedure file. See **The Include Statement** on page IV-145 for instructions on including a procedure file.

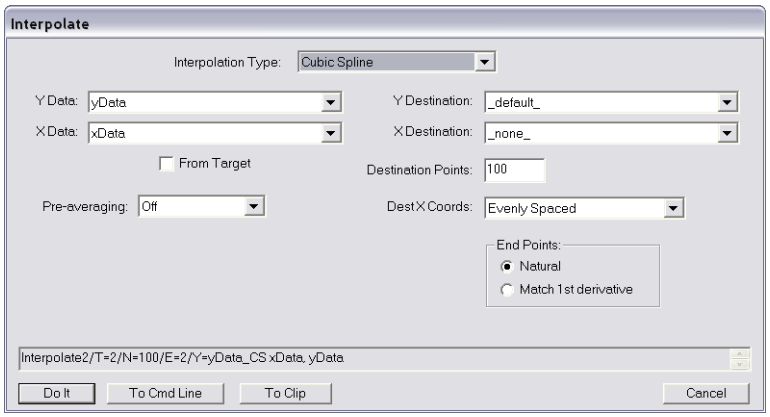
If you have blanks (NaNs) in your input data, the interp function will give you blanks in your output waveform as well. The Interpolate XOP, discussed in the next section, interpolates across gaps in data and does not produce gaps in the output.

Using the Interpolate External Operation

The Interpolate XOP provides not only linear but also cubic and smoothing spline interpolation. Furthermore, it does not require the input to be sorted and can automatically make the destination waveform and set its X scaling. It also has a dialog that makes it easy to use interactively.

The Interpolate XOP is automatically installed by the Igor installer in the "Igor Extensions" folder. It adds an Interpolate item to the Analysis menu.

To use it on our sample XY data, choose Interpolate and set up the dialog as shown below.



Choosing "_default_" as the "Y destination" auto-names the destination wave by appending "_CS" to the name of the input "Y data" wave. Choosing "_none_" as the "X destination" creates a waveform from the input XY pair rather than a new XY pair.

Here is a rewrite of the XYToWave1 function that uses the Interpolate XOP rather than the interp function.

```
Function XYToWave2(xWave, yWave, wWaveName, numPoints)
   Wave/D xWave                // x wave in the XY pair
   Wave/D yWave                // y wave in the XY pair
   String wWaveName            // name to use for new waveform wave
   Variable numPoints          // number of points for waveform

   Interpolate2/T=2/N=(numPoints)/E=2/Y=$wWaveName xWave, yWave
End
```

Gaps in the input data are ignored. Most often, this is what is desired but if you want to maintain gaps in the output data you will have to install them yourself. You can use the fact that the interp function maintains gaps to restore gaps in the Interpolate XOP output. For example:

```
yDest= yDest + 0*interp(x, xSource, ySource)
```

Only when the output of interp is a NaN is data in the destination wave affected. This works because NaN multiplied by anything (even zero) is NaN.

To use the WaveMetrics-supplied XYToWave2 function, include the "XY Pair To Waveform" procedure file. See **The Include Statement** on page IV-145 for instructions on including a procedure file.

The cubic spline algorithm used by the Interpolate XOP is derived from the cubic spline function in *Numerical Recipes in C* (see **References** on page III-151). The XOP also provides a smoothing spline based on Reinsch (1967).

Dealing with Missing Values

A missing value is represented in Igor by the value NaN which means "Not a Number". A missing value is also called a "blank", because it appears as a blank cell in a table.

When a NaN is combined arithmetically with any value, the result is NaN. To see this, execute the command:

```
Print 3+NaN, NaN/5, sin(NaN)
```

By definition, a NaN is not equal to anything. Consequently, the condition in this statement:

```
if (myValue == NaN)
```

is always false.

The workaround is to use the **numtype** function:

```
if (NumType(myValue) == 2)        // Is it a NaN?
```

See also **NaNs, INFs and Missing Values** on page II-100 for more about how NaN values.

Some routines deal with missing values by ignoring them. The **CurveFit** operation (see page V-85) is one example. Others may produce unexpected results in the presence of missing values. Examples are the **FFT** operation and the **area** and **mean** functions.

Here are some strategies for dealing with missing values.

Replace the Missing Values With Another Value

You can replace NaNs in a wave with this statement:

```
wave0 = NumType(wave0)==2 ? 0:wave0    // Replace NaNs with zero
```

If you're not familiar with the :? operator, see **Operators** on page IV-5.

For multi-dimensional waves you can replace NaNs using **MatrixOp**. For example:

```
Make/O/N=(3,3) matNaNTest = p + 10*q
Edit matNaNTest
matNaNTest[0][0] = NaN; matNaNTest[1][1] = NaN; matNaNTest[2][2] = NaN
MatrixOp/O matNaNTest=ReplaceNaNs(matNaNTest,0)    // Replace NaNs with 0
```

Remove the Missing Values

For 1D waves you can remove NaNs using **WaveTransform** `zapNaNs`. For example:

```
Make/N=5 NaNTest = p
Edit NaNTest
NaNTest[1] = NaN; NaNTest[4] = NaN
WaveTransform zapNaNs, NaNTest
```

There is no built-in operation to remove NaNs from an XY pair if the NaN appears in either the X or Y wave. You can do this, however, using the RemoveNaNsXY procedure in the "Remove Points" WaveMetrics procedure file which you can access through Help→Windows→WM Procedures Index.

There is no operation to remove NaNs from multi-dimensional waves as this would require removing the entire row and entire column where each NaN appeared.

Work Around Gaps in Data

Many analysis routines can work on a subrange of data. In many cases you can just avoid the regions of data that contain missing values. In other cases you can extract a subset of your data, work with it and then perhaps put the modified data back into the original wave.

Here is an example of extract-modify-replace (even though Smooth properly accounts for NaNs):

```
Make/N=100 data1= sin(P/8)+gnoise(.05); data1[50]= NaN
Display data1
Duplicate/R=[0,49] data1,tmpdata1      // start work on first set
Smooth 5,tmpdata1
data1[0,49]= tmpdata1[P]               // put modified data back
Duplicate/O/R=[51,] data1,tmpdata1     // start work on 2nd set
Smooth 5,tmpdata1
data1[51,]= tmpdata1[P-51]
KillWaves tmpdata1
```

Replace Missing Data with Interpolated Values

You can replace NaN data values prior to performing operations that do not take kindly to NaNs by replacing them with smoothed or interpolated values using the **Smooth** operation (page V-577), the **Loess** operation (page V-357), or the Interpolate XOP.

Replace Missing Data Using the Interpolate XOP

By using the same number of points for the destination as you have source points, you can replace NaNs without modifying the other data.

If you have waveform data, simply duplicate your data and perform linear interpolation using the same number of points as your data. For example, assuming 100 data points:

```
Duplicate data1,data1a
Interpolate/T=1/N=100/Y=data1a data1
```

If you have XY data, the XOP has the ability to include the input x values in the output X wave. For example:

```
duplicate data1, yData1, xData1
xData1 = x
Display yData1 vs xData1
Interpolate2/T=1/N=100/I/Y=yData1a/X=xData1a xData1,yData1
```

If, after performing an operation on your data, you wish to put the modified data back in the source wave while maintaining the original missing values you can use a wave assignment similar to this:

```
yData1 = (numtype(yData1) == 0) ? yData1 : yData1a
```

This technique can also be applied using interpolated results generated by the **Smooth** operation (page V-577) or the **Loess** operation (page V-357).

Replace Missing Data Using Median Smoothing

You can use the Smooth dialog to replace each NaN with the median of surrounding values.

Select the Median smoothing algorithm, select "NaNs" from the Replace popup, and choose "Median" for the "with:" radio button. Enter the number of surrounding points used to compute the median (an odd number is best).

You can choose to overwrite the NaNs or create a new waveform with the result. The Smooth dialog produces commands like this:

```
Duplicate/O data1,data1_smth;DelayUpdate
Smooth/M=(NaN) 5, data1_smth
```

Interpolation

Igor Pro has a number of interpolation tools that are designed for different applications. We summarize these in the table below.

Data	Operation/Function	Interpolation Method
1D waves	wave assignment, e.g., `val=wave(x)`	Linear
1D waves	**Smooth**	Running median, average, binomial, Savitsky-Golay
1D XY waves	**interp**()	Linear
1D single or XY waves	Interpolate XOP	Linear, cubic spline, smoothing spline
1D or 2D single or XY	**Loess**	Locally-weighted regression
Triplet XYZ waves	**ImageInterpolate**	Voronoi

Data	Operation/Function	Interpolation Method
1D X, Y, Z waves	Data→Packages→XYZ to Matrix	Voronoi
1D X, Y, Z waves	**Loess**	Locally-weighted regression
2D waves	**ImageInterpolate**	Bilinear, splines, Kriging, Voronoi
2D waves	**Interp2D**()	Bilinear
2D waves (points on the surface of a sphere)	**SphericalInterpolate**	Voronoi
3D waves	**Interp3D**(), **Interp3DPath**, **ImageTransform extractSurface**	Trilinear
3D scatter data	**Interpolate3D**	Barycentric

All the interpolation methods in this table consist of two common steps. The first step involves the identification of data points that are nearest to the interpolation location and the second step is the computation of the interpolated value using the neighboring values and their relative proximity. You can find the specific details in the documentation of the individual operation or function.

Differentiation and Integration

The **Differentiate** operation (see page V-114) and **Integrate** operation (see page V-309) provide a number of algorithms for operation on one-dimensional waveform and XY data. These operations can either replace the original data or create a new wave with the results. The easiest way to use these operations is via dialogs available from the Analysis menu. These handy dialogs even provide for graphing the results.

For most applications, trapezoidal integration and central differences differentiation are appropriate methods. See the reference section for details. However, when operating on XY data, the different algorithms have different requirements for the number of points in the X wave. If your X wave does not show up in the dialog, try choosing a different algorithm or click the help button to see what the requirements are.

When operating on waveform data, X scaling is taken into account (although this can be turned off using the /P flag). You can use the **SetScale** operation (see page V-564) to define the X scaling of your Y data wave.

Although these operations work along just one dimension, they can be targeted to operate along rows or columns of a matrix (or even higher dimensions) using the /DIM flag.

The Integrate operation replaces or creates a wave with the numerical integral. For finding the area under a curve, see **Areas and Means** on page III-122.

Areas and Means

You can compute the area and mean value of a wave in several ways using Igor.

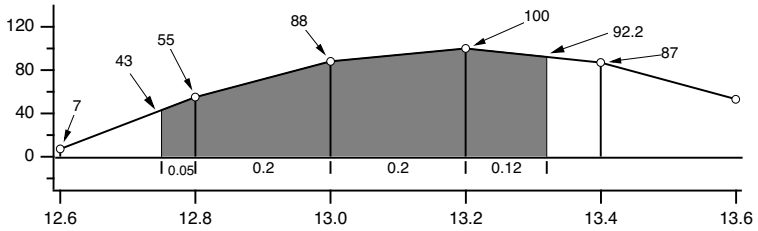
Perhaps the simplest way to compute a mean value is with the Wave Stats dialog in the Analysis menu. The dialog is pictured under **Wave Statistics** on page III-124. You select the wave, type in the X range (or use the current cursor positions), click Do It, and Igor prints several statistical results to the history area. Among them is V_avg, which is the average, or mean value. This is the same value that is returned by the **mean** function (see page V-388), which is faster because it doesn't compute any other statistics. The mean function will return NaN if any data within the specified range is NaN. The WaveStats operation, on the other hand, ignores such missing values.

WaveStats and the mean function use the same method for computing the mean: find the waveform values within the given X range, sum them together, and divide by the number of values. The X range serves only to select the values to combine; the range is rounded to the nearest point numbers.

If you consider your data to describe discrete values, such as a count of events, then you should use either WaveStats or the mean function to compute the average value. You can most easily compute the total number of events, which is an area of sorts, using the **sum** function (see page V-682). It can also be done easily by multiplying the WaveStats outputs V_avg and V_npnts.

If your data is a sampled representation of a continuous process such as a sampled audio signal, you should use the faverage function to compute the mean, and the area function to compute the area. These two functions use the same linear interpolation scheme as does trapezoidal integration to estimate the waveform values between data points. The X range is *not* rounded to the nearest point; partial X intervals are included in the calculation through this linear interpolation.

The diagram below shows the calculations each function performs for the data shown. The two values 43 and 92.2 are linear interpolation estimates.



Comparison of area, faverage and mean functions over interval (12.75,13.32)

`WaveStats/R=(12.75,13.32) wave`		
`V_avg`	= (55+88+100+87)/4 = 82.5	
`mean(wave,12.75,13.32)`	= (55+88+100+87)/4 = 82.5	
`area(wave,12.75,13.32)`	= 0.05 · (43+55) / 2	first trapezoid
	+ 0.20 · (55+88) / 2	second trapezoid
	+ 0.20 · (88+100) / 2	third trapezoid
	+ 0.12 · (100+92.2) / 2	fourth trapezoid
	= 47.082	
`faverage(wave,12.75,13.32)`	= area(wave,12.75,13.32) / (13.32-12.75)	
	= 47.082/0.57 = 82.6	

Note that only the area function is affected by the X scaling of the wave. faverage eliminates the effect of X scaling by dividing the area by the same X range that area multiplied by.

One problem with these functions is that they can not be used if the given range of data has missing values (NaNs). See **Dealing with Missing Values** on page III-119 for details.

X Ranges and the Mean, faverage, and area Functions

The X range input for the mean, faverage and area functions are optional. Thus, to include the entire wave you don't have to specify the range:

```
Make/N=10 wave=2; Edit wave.xy    // X ranges from 0 to 9
Print area(wave)                  // entire X range, and no more
18
```

Sometimes, in programming, it is not convenient to determine whether a range is beyond the ends of a wave. Fortunately, these functions also accept X ranges that go beyond the ends of the wave.

```
Print area(wave, 0, 9)            // entire X range, and no more
18
```

You can use expressions that evaluate to a range beyond the ends of the wave:

```
Print leftx(wave),rightx(wave)
0  10
Print area(wave,leftx(wave),rightx(wave))      // entire X range, and more
18
```

or even an X range of ±∞:

```
Print area(wave, -Inf, Inf)  // entire X range of the universe
18
```

Finding the Mean of Segments of a Wave

Under **Analysis Programming** on page III-142 is a function that finds the mean of segments of a wave where you specify the length of the segments. It creates a new wave to contain the means for each segment.

Area for XY Data

To compute the area of a region of data contained in an XY pair of waves, use the **areaXY** function (see page V-29). There is also an XY version of the faverage function; see **faverageXY** on page V-152.

Technical Note 018, "Area and Integration" discusses XY area computations in greater detail and provides routines for cubic spline area and integration, area of exponential data and integration of spectroscopic or chromatographic peaks.

Wave Statistics

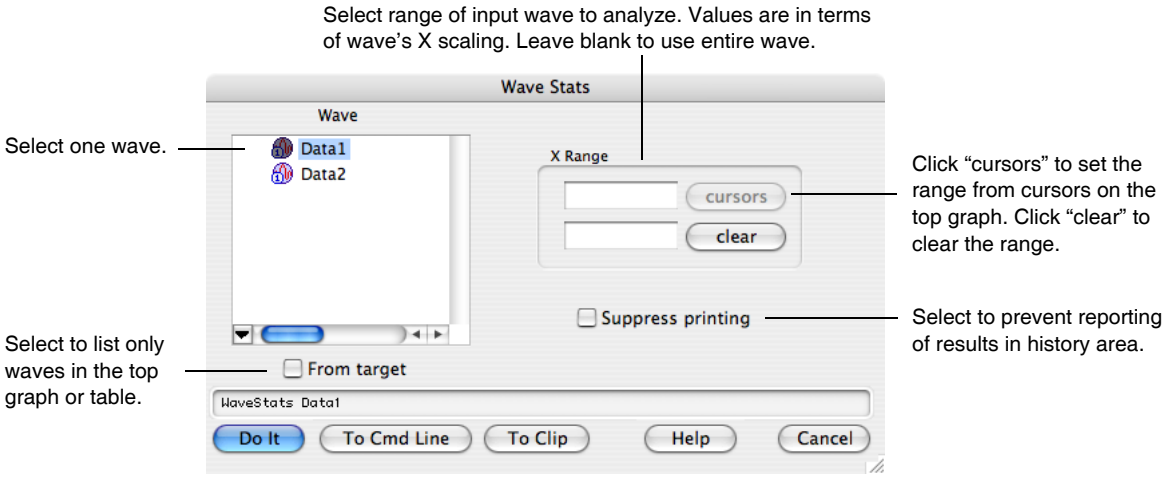
The **WaveStats** operation (see page V-729) computes various descriptive statistics relating to a wave and prints them in the history area of the command window. It also stores the statistics in a series of special variables or in a wave so you can access them from a procedure.

The statistics printed and the corresponding special variables are:

Variable	Meaning
V_npnts	Number of points in range, not including points whose value is NaN or INF.
V_numNaNs	Number of NaNs in range.
V_numINFs	Number of INFs in range.
V_avg	Average of data values.
V_sdev	Standard deviation of data values, $\sigma = \sqrt{\frac{1}{V\_npnts-1}\sum (Y_i - V\_avg)^2}$ ("Variance" is V_sdev^2.)
V_rms	RMS (Root Mean Square) of Y values $= \sqrt{\left(\frac{1}{V\_npnts}\sum Y_i^2\right)}$
V_adev	Average deviation $= \frac{1}{V\_npnts}\sum_{i=0}^{V\_npnts-1} \lvert x_i - \bar{x}\rvert$

Variable	Meaning
V_skew	$\text{Skewness} = \frac{1}{V\_npnts}\sum_{i=0}^{V\_npnts-1}\left[\frac{x_i-\bar{x}}{\sigma}\right]^3$
V_kurt	$\text{Kurtosis} = \frac{1}{V\_npnts}\sum_{i=0}^{V\_npnts-1}\left[\frac{x_i-\bar{x}}{\sigma}\right]^4 - 3$
V_minloc	X location of minimum data value.
V_min	Minimum data value.
V_maxloc	X location of maximum data value.
V_max	maximum data value.
V_minRowLoc	Row containing minimum Z value (2D or higher waves).
V_minColLoc	Column containing minimum Z value (2D or higher waves).
V_maxColLoc	Column containing maximum Z value (2D or higher waves).
V_maxRowLoc	Row containing maximum Z value (2D or higher waves).
V_minLayerLoc	Layer containing minimum Z value (3D or higher waves).
V_maxLayerLoc	Layer containing maximum Z value (3D or higher waves).
V_minChunkLoc	Chunk containing minimum Z value (4D waves only).
V_maxChunkLoc	Chunk containing maximum Z value (4D waves only).
V_startRow	First wave point. Zero if you do not use /R.
V_endRow	Last wave point. Last point if you do not use /R.

To use the WaveStats operation, choose Wave Stats from the Analysis menu.



The WaveStats dialog expects that the range you specify, if any, be in terms of the X values of the source wave. You set this using the Change Wave Scaling dialog or **SetScale** operation (see page V-564). The WaveStats operation can use a point range, too (see page V-729).

The "Suppress printing" option is normally used when you call WaveStats from an Igor procedure. The procedure uses the special variables set by WaveStats.

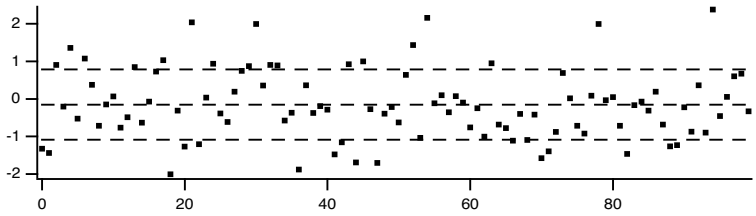
Igor ignores NaNs and INFs in computing the average, standard deviation, RMS, minimum and maximum. NaNs result from computations that have no defined mathematical meaning. They can also be used to represent missing values. INFs result from mathematical operations that have no finite value.

Following is a macro that illustrates the use of WaveStats. The macro shows the average and standard deviation of a source wave, assumed to be displayed in the top graph. It draws lines to indicate the average and standard deviation.

```
#pragma rtGlobals=1     // Use modern global access method.

Function ShowAvgStdDev(source)
   Wave source                                          // source waveform

   Variable left=leftx(source),right=rightx(source)   // source X range
   WaveStats/Q source
   SetDrawLayer/K ProgFront
   SetDrawEnv xcoord=bottom,ycoord=left,dash= 7
   DrawLine left, V_avg, right, V_avg                  // show average
   SetDrawEnv xcoord=bottom,ycoord=left,dash= 7
   DrawLine left, V_avg+V_sdev, right, V_avg+V_sdev   // show +std dev
   SetDrawEnv xcoord=bottom,ycoord=left,dash= 7
   DrawLine left, V_avg-V_sdev, right, V_avg-V_sdev   // show -std dev
   SetDrawLayer UserFront
End
```

You could try this function using the following commands.

```
Make/N=100 wave0 = gnoise(1)
Display wave0; ModifyGraph mode(wave0)=2, lsize(wave0)=3
ShowAvgStdDev(wave0)
```



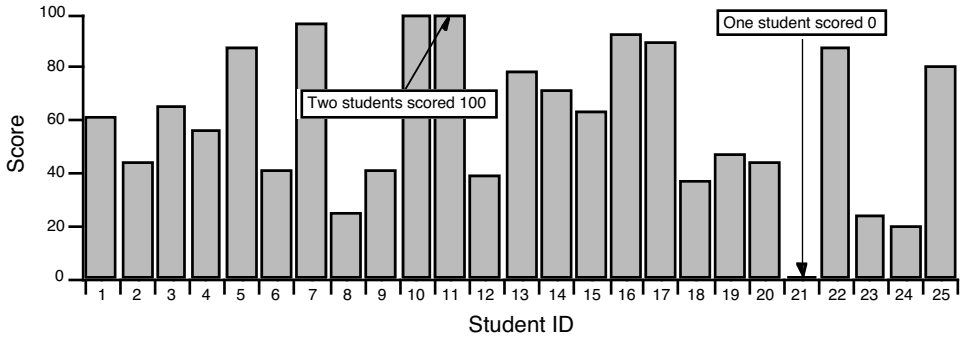
When you use WaveStats with a complex wave, you can choose to compute the same statistics as above for the real, imaginary, magnitude and phase of the wave. By default WaveStats only computes the statistics for the real part of the wave. When computing the statistics for other components, the operation stores the results in a multidimensional wave M_WaveStats.

If you are working with large amounts of data and you are concerned about computation speed you might be able to take advantage of the /M flag that limits the calculation to the first order moments.

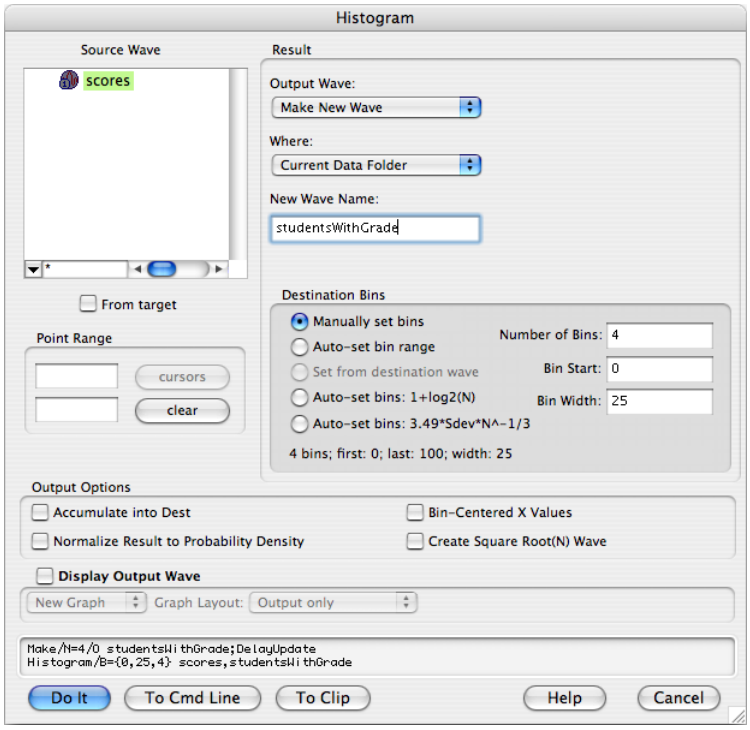
If you are working with 2D or 3D waves and you want to compute the statistics for a domain of an arbitrary shape you should use the **ImageStats** operation (see page V-287) with an ROI wave.

Histograms

A histogram totals the number of input values that fall within each of a number of value ranges (or "bins") usually of equal extent. For example, a histogram is useful for counting how many data values fall in each range of 0-10, 10-20, 20-30, etc. This calculation is often made to show how students performed on a test:



The usual use for a histogram in this case is to figure out how many students fall into certain numerical ranges, usually the ranges associated with grades A, B, C, and D. Suppose the teacher decides to divide the 0-100 range into 4 equal parts, one per grade. The **Histogram** operation (see page V-245) can be used to show how many students get each grade by counting how many students fall in each of the 4 ranges. Let's use the Histogram dialog and enter the obvious values:



The Histogram operation analyzes the source wave (`scores`), and puts the histogram result into a destination wave (`studentsWithGrade`).

Note: The Histogram operation does not produce a "bar chart". For information on how to make a bar chart, see **Bars** on page II-252, or Chapter II-13, **Category Plots**. Also see **Graphing Histogram Results** on page III-129.

The first of the "bins" has been set manually to start at 0 and to count values up to (but not including) 25. We expect the four bins to span the range of 0–100. The Histogram dialog created the needed `studentsWithGrade` destination wave:

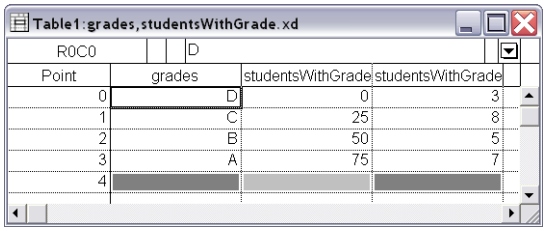
```
Make/N=4/D/O studentsWithGrade; DelayUpdate
```

and then used it in the Histogram operation:

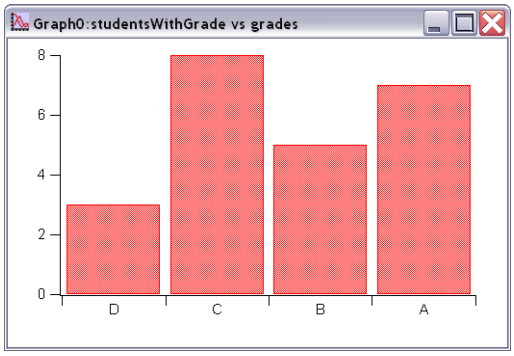
```
Histogram/B={0,25,4} scores,studentsWithGrade
```

Let's create a text wave of `grades` to plot `studentsWithGrade` versus a grade letter in a category plot:

```
Make/O/T grades= {"D", "C", "B", "A"}
Display studentsWIthGrade vs grades
SetAxis/A/E=1 left
```



Point	grades	studentsWithGrade	studentsWithGrade
0	D	0	3
1	C	25	8
2	B	50	5
3	A	75	7
4			



Everything *looks* good in the category plot. Let's double-check that all the students made it into the bins:

```
•print sum(studentswithgrade)
  23
```

There are two missing students. They are ones who scored 100 on the test. The four bins we defined are actually:

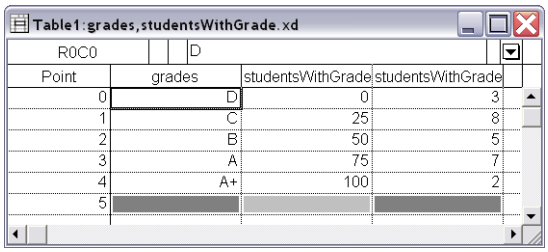
Bin 1:	0 - 24.99999
Bin 2:	25 - 49.99999
Bin 3:	50 - 74.99999
Bin 4:	75 - 99.99999

The problem is that the test scores actually encompass 101 values, not 100. To include the perfect scores in the last bin, we could add a small number such as 0.001 to the bin width:

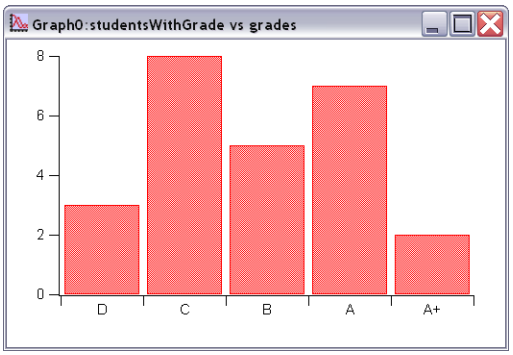
Bin 1:	0 - 25.00999
Bin 2:	25.001 - 49.00199
Bin 3:	50.002 - 74.00299
Bin 4:	75.003 - 100.0399

The students who scored 25, 50 or 75 would be moved down one grade, however. Perhaps the best solution is to add another bin for perfect scores:

```
Make/O/T grades= {"D", "C", "B", "A", "A+"}
Histogram/B={0,25,5} scores,studentsWithGrade
```



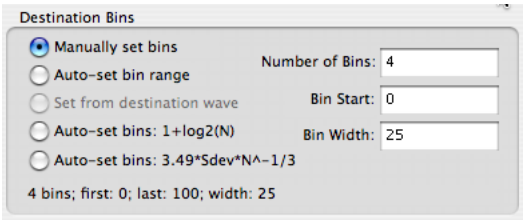
Point	grades	studentsWithGrade	studentsWithGrade
0	D	0	3
1	C	25	8
2	B	50	5
3	A	75	7
4	A+	100	2
5			



This example was intended to point out the care needed when choosing the histogram binning. Our example used "manual binning". The Histogram operation actually sets the binning in five ways:

Bin Mode	What It Does
Manual bins	Sets number of points and X scaling of the destination (output) wave based on parameters that you explicitly specify.
Auto-set bins	Sets X scaling of destination wave to cover the range of values in the **source** wave. Does not change the number of points (bins) in the destination wave. Thus, you must set the number of destination wave points before computing the histogram. When using the Histogram Dialog, if you select Make New Wave or Auto from the Output Wave menu, the dialog must be told how many points the new wave should have. It displays the Number of Bins box to let you specify the number.
Set bins from destination wave	Does not change the X scaling or the number of points in the destination wave. Thus, you need to set the X scaling and number of points of the destination wave before computing the histogram. When using the Histogram Dialog, the Set from destination wave radio button is only available if you choose Select Existing Wave from the Output Wave menu.
Auto-set bins: 1+log2(N)	Examines the input data and sets the number of bins based on the number of input data points. Sets the bin range the same as if Auto-set bin range were selected.
Auto-set bins: 3.49*Sdev*N^-1/3	Examines the input data and sets the number of bins based on the number of input data points and the standard deviation of the data. Sets the bin range the same as if Auto-set bin range were selected.

These five options correspond to the radio buttons in the Histogram dialog:



Note: The option "Set from destination wave" is not available because the dialog Output Wave menu is set to Make New Wave. To use Set from destination wave you must choose Existing Wave in the Output Wave menu.

Histogram Caveats

You must create the destination wave, using the Make operation or the Histogram dialog, before computing the histogram. Depending on how you want to set the histogram's bins, you may need to set the X scaling of the destination wave also, using the **SetScale** operation (see page V-564).

The Histogram operation does not distinguish between 1D waves and multidimensional waves. If you use a multidimensional wave as the source wave, it will be analyzed as if the wave were one dimensional. This may still be useful- you will get a histogram showing counts of the data values from the source wave as they fall into bins.

If you would like to perform a histogram of 2D or 3D image data, you may want to use the **ImageHistogram** operation (see page V-261), which supports specific features that apply to images only.

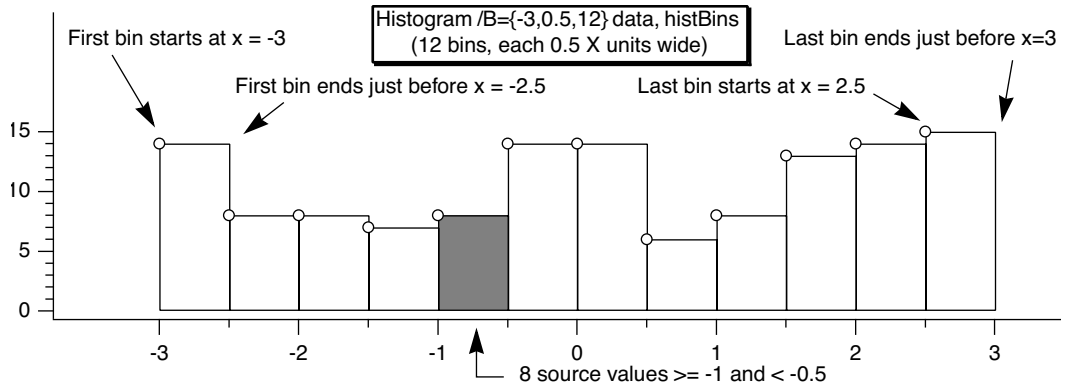
Graphing Histogram Results

Our example above displayed the histogram results as a category plot because the bins corresponded to text values. Often histogram bins are displayed on a numeric axis. In this case you need to know how Igor displays a histogram result.

For example, this histBins destination wave has 12 points (bins), the first bin starting at -3, and each bin is 0.5 wide. The X scaling is shown in the table:

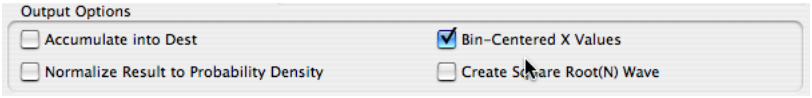
Point	histBins.x	histBins.d
0	-3	14
1	-2.5	8
2	-2	8
3	-1.5	7
4	-1	8
5	-0.5	14
6	0	14
7	0.5	6
8	1	8
9	1.5	13
10	2	14
11	2.5	15
12		

When histBins is graphed in both bars and markers modes, it looks like this:



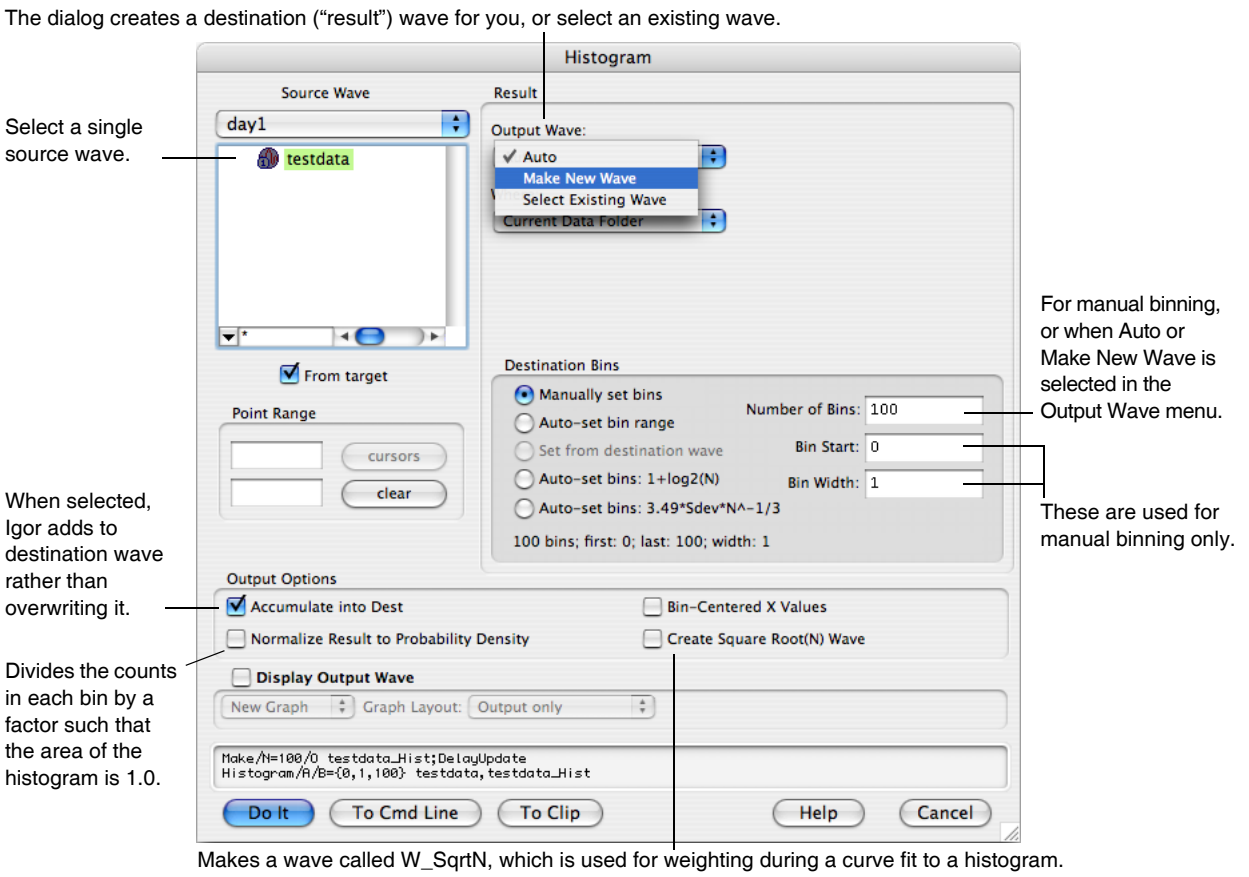
Note that the markers are positioned at the start of the bars. You can offset the marker trace by half the bin width if you want them to appear in the center of the bin.

Alternatively, you can make a second histogram using the Bin-Centered X Values option:



Histogram Dialog

To use the Histogram operation, choose Histogram from the Analysis menu.



To use the “Manually set bins” or “Set from destination wave” bin modes, you need to decide the range of data values in the source wave that you want the histogram to cover. You can do this visually by graphing the source wave or you can use the WaveStats operation to find the exact minimum and maximum source values.

The dialog requires that you enter the starting bin value and the bin width. If you know the starting bin value and the ending bin value then you need to do some arithmetic to calculate the bin width. One way to do this is to click the To Cmd Line button and edit the Histogram command in Igor’s command line. For example, if you want 12 bins from -3 to +3, you could execute

```
Histogram/B={-3, (3 - (-3))/12, 12} test, hist
```

A line of text at the bottom of the Destination Bins box tells you the first and last values, as well as the width and number of bins. This information can help with trial-and-error settings.

If you use the “Manually set bins” or any of the “Auto-set” modes, Igor will set the X units of the destination wave to be the same as the Y units of the source wave.

If you enable the Accumulate checkbox, Histogram does not clear the destination wave. Use this to accumulate results from several histograms in one destination. If you want to do this, don’t use the “Auto-set bins” option since it makes no sense to change bins in mid-stream. Instead, use the “Set from destination wave” mode. To use the Accumulate option, the destination wave must be double- or single-precision and real.

The “Bin-Centered X Values” and “Create Square Root(N) Wave” options are useful for curve fitting to a histogram. If you do not use Bin-Centered X Values, any X position parameter in your fit function will be shifted by half a bin width. The Square Root(N) Wave creates a wave that estimates the standard deviation of the histogram data; this is based on the fact that counting data have a Poisson distribution. The wave created by this option does not try to do anything special with bins having zero counts, so if you use the

square root(N) wave to weight a curve fit, these zero-count bins will be excluded from the fit. You may need to replace the zeroes with some appropriate value.

The binning modes were added in Igor Pro. In earlier versions of Igor, the accumulate option had *two* effects:

- Did not clear the destination wave.
- Effectively used the "Set bins from destination wave" mode.

To maintain backward compatibility, the Histogram operation still behaves this way if the accumulate ("/A") flag is used and no bin ("/B") flag is used. This dialog always generates a bin flag. Thus, the accumulate flag just forces accumulation and has no effect on the binning.

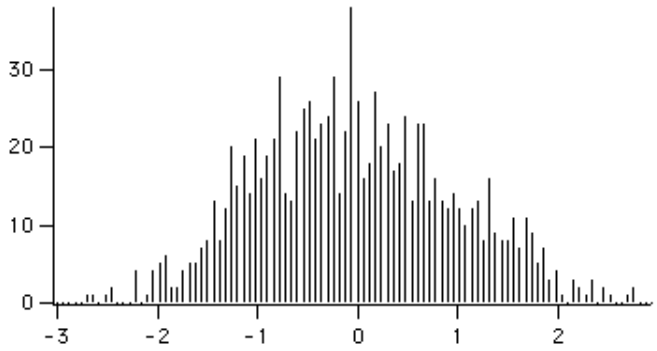
You can use the Histogram operation on multidimensional waves but they are treated as though the data belonged to a single 1D wave. If you are working with 2D or 3D waves you may prefer to use the **Image-Histogram** operation (see page V-261), which computes the histogram of one layer at a time.

Histogram Example

The following commands illustrate a simple test of the histogram operation.

```
Make/N=1024 noise = gnoise(1)     // make raw data
Make hist                         // make destination for histogram
Histogram/B={-3, (3 - -3)/100, 100} noise, hist // do histogram
Display hist; Modify mode(hist)=1
```

These commands produce the following graph.



Curve Fitting to a Histogram

By default, a histogram wave has the X scaling set such that an X value gives the value at the left end of a bin. Usually if you are going to fit a curve to a histogram, you want centered X values. You can change the X scaling to centered values with this command:

```
setscale/P x  leftx(hist)+deltax(hist)/2, deltax(hist),hist
```

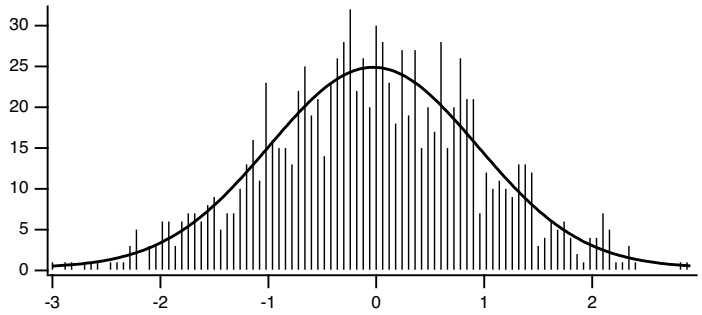
In this command, `hist` is the name of a wave containing a histogram. Substitute whatever name your wave has.

It is easier to simply use the option to have the Histogram operation produce output with bin-centered X values. Using the example from the previous section, add the /C flag:

```
Histogram/C/B={-3, (3 - -3)/100, 100} noise, hist  // do histogram
```

Because the values in the source wave are Gaussian deviates generated by the gnoise function, the histogram should have the familiar Gaussian bell-shape. You can estimate the characteristics of the population the samples were taken from by fitting a Gaussian curve to the data. First try fitting a Gaussian curve to the example histogram:

```
CurveFit gauss hist /D     // curve fit to histogram
```



The solution from the curve fit was (if you try this example on your computer, the results will be somewhat different because the random noise added by gnoise will be different):

```
y0    = -0.19106 ± 0.78
A     = 24.618 ± 0.871
x0    = -0.037059 ± 0.031
width = 1.4361 ± 0.0748
```

Note that the peak position (x0) is shifted approximately half a bin below zero. Since the gnoise() function produces random numbers with mean of zero, we would expect x0 to be close to zero. The shifted value of x0 is a result of Igor's way of storing the X values for histogram bins. Setting the X value to the left edge is good for displaying a bar chart, but bad for curve fitting.

One solution to the problem is simply to correct the curve fit result after the fact:

```
W_coef[2] += 0.03    // add half a bin width to the peak position
```

This solution has many drawbacks, including the fact that the fit curve on the graph is still wrong because it was calculated using the uncorrected value of x0.

Another solution is to use the bin-centered X value option before doing the curve fit, as was done above:

```
Histogram/C/B={-3, (3 - -3)/100, 100} noise, hist  // do histogram
CurveFit gauss hist /D           // curve fit to histogram
```

The result of the curve fit is closer to what we expect:

```
y0    = -0.19106 ± 0.78
A     = 24.618 ± 0.871
x0    = -0.0070593 ± 0.031
width = 1.4361 ± 0.0748
```

But this shifts the trace showing the histogram by half a bin on the graph. If the trace is displayed using markers or dots, this may be what is desired, but if you have used bars, the display is incorrect.

Another possibility is to make an X wave to go with the histogram data. This X wave would contain X values shifted by half a bin. Use this X wave as input to the curve fit, but don't use it on the graph:

```
Duplicate hist, hist_x
hist_x = x + deltax(hist)/2
CurveFit gauss  hist /X=hist_x/D
```

Use this method to graph the original histogram wave without modifying the X scaling, so a graph using bars is correct. It also gives a curve fit that uses the center X values, giving the correct x0. You could also use the Histogram operation twice, once with the /C flag to get bin-centered X values, and once without to get the shifted X scaling appropriate for bars. Both methods have the drawback of creating an extra wave that you must track.

There is one last refinement to curve fitting to a histogram. Since the histogram represents counts, the values in a histogram should have uncertainties described by a Poisson distribution. The standard deviation of a Poisson distribution is equal to the square root of the mean, which implies that the estimated error of a histogram bin depends on the magnitude of the value. This, in turn, implies that the errors are not constant and a curve fit will give a biased solution.

The correct solution is to use a weighting wave; use the /N flag with the Histogram operation to get the appropriate wave. This example makes a new data set using gnoise to make gaussian-distributed values, makes a histogram with bin-centered X values and the appropriate weighting wave, and then does two curve fits, one without weighting and one with:

```
Make/N=1024 gdata=gnoise(1)
Make/N=20/O gdata_Hist
Histogram/C/N/B=4 gdata,gdata_Hist
Display gdata_Hist
ModifyGraph mode=3,marker=8
CurveFit gauss  gdata_Hist /D
CurveFit gauss  gdata_Hist /W=W_SqrtN /I=1 /D
```

Note the "`/W=W_SqrtN /I=1`" addition to the second CurveFit command; this adds the weighting using the weighting wave created by the Histogram operation. Also, /B=4 was used to have the Histogram operation set the number of bins and bin range appropriately for the input data.

The results from the unweighted fit:

```
y0   =-3.3383 ± 2.98
A    =133.26 ± 3.51
x0   =0.024088 ± 0.0252
width=1.5079 ± 0.0578
```

And from the weighted fit:

```
y0   =0.33925 ± 0.804
A    =135.21 ± 5.25
x0   =0.0038416 ± 0.031
width=1.3604 ± 0.0405
```

Computing a Histogram with Logarithmic Bins

Analysis Programming on page III-142 describes how you can compute a histogram with logarithmic bins. The built-in Histogram operation can not do this.

Computing an "Integrating" Histogram

In a histogram, each bin of the destination wave contains a count of the number of occurrences of values in the source that fell within the bounds of the bin. In an *integrating* histogram, instead of *counting* the occurrences of a value within the bin, we *add* the value itself to the bin. When we're done, the destination wave contains the sum of all values in the source which fell within the bounds of the bin.

Igor comes with an example experiment called "Integrating Histogram" that illustrates how to do this with a user function. This experiment is in the "Examples:Analysis" folder.

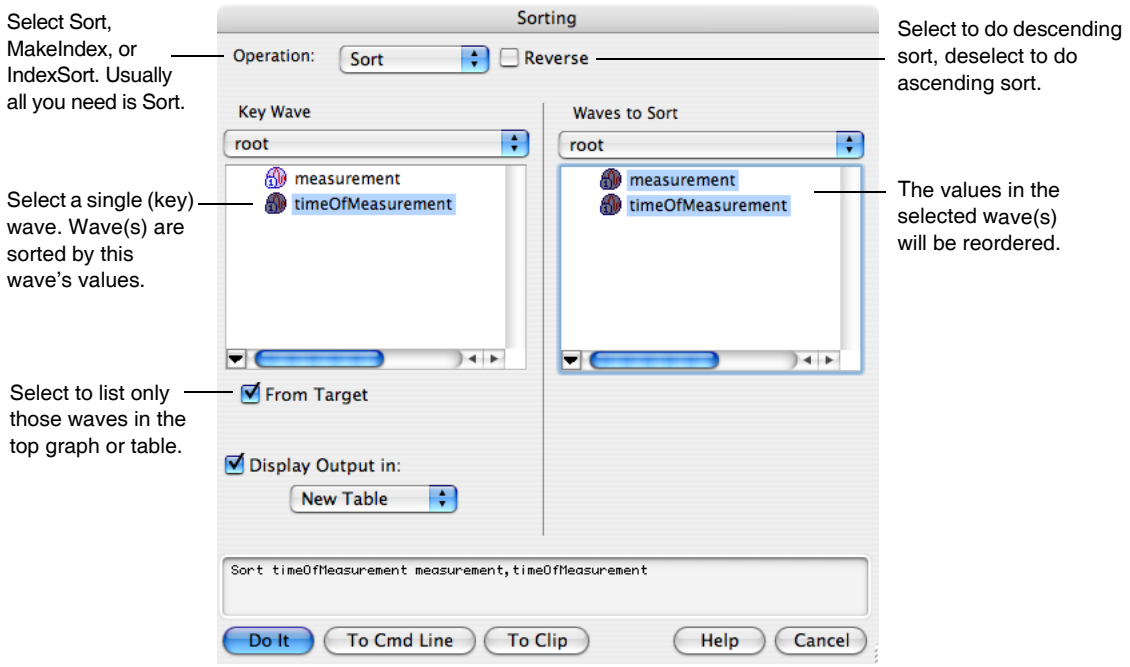
Sorting

The **Sort** operation (see page V-581) sorts one or more 1D numeric or text waves in ascending or descending order.

The Sort operation is often used to prepare a wave or an XY pair for subsequent analysis. For example, the interp function assumes that the X input wave is monotonic.

There are other sorting-related operations: MakeIndex and IndexSort. These are used in rare cases and are described the section **MakeIndex and IndexSort Operations** on page III-137. Also see the **SortList** operation (page V-581).

To use the Sort operation, choose Sort from the Analysis menu.



The sort key wave controls the reordering of points. However, the key wave itself is not reordered unless it is also selected as a destination wave in the "Waves to Sort" list.

The number of points in the destination wave or waves must be the same as in the key wave. When you select a wave from the dialog's Key Wave list, Igor shows only waves with the same number of points in the Waves to Sort list.

The key wave can be a numeric or text wave, but it must not be complex. The destination wave or waves can be text, real or complex except for the MakeIndex operation in which case the destination must be text or real.

The number of destination waves is limited by the 400 character limit in Igor's command buffer. To sort a very large number of waves, use several Sort commands in succession, being careful not to sort the key wave until the very last.

Simple Sorting

In the simplest case, you would select a single wave as both the source and the destination. Then, Sort would merely sort that wave.

If you want to sort an XY pair such that the X wave is in order, you would select the X wave as the source and both the X and Y waves as the destination.

Sorting to Find the Median Value

The following user-defined function illustrates a simple use of the Sort operation to find the median value of a wave.

```
Function/D Median(w, x1, x2)  // Returns median value of wave w
   Wave w
   Variable x1, x2            // range of interest

   Variable result

   Duplicate/R=(x1,x2) w, tempMedianWave  // Make a clone of wave
   Sort tempMedianWave, tempMedianWave    // Sort clone
   SetScale/P x 0,1,tempMedianWave
   result = tempMedianWave((numpnts(tempMedianWave)-1)/2)
   KillWaves tempMedianWave
```

```
   return result
End
```

We used the name tempMedianWave rather than just temp to minimize the chance of a conflict. It is possible that a procedure in the chain of procedures leading to Median has created a wave named temp.

It is easier and faster to use the **StatsMedian** operation (page V-640) to find the median value in a wave.

Multiple Sort Keys

If the key wave has two or more identical values, you may want to use a secondary source to determine the order of the corresponding points in the destination. This requires using multiple sort keys. The Sorting dialog does not provide a way to specify multiple sort keys but the Sort operation does. Here is an example demonstrating the difference between sorting by single and by multiple keys. Notice that the sorted wave (`tdest`) is a text wave, and the sort keys are text (`tsrc`) and numeric (`nw1`):

```
Make/O/T tsrc={"hello","there","hello","there"}
Duplicate/O tsrc,tdest
Make nw1= {3,5,2,1}
tdest= tsrc + " " + num2str(nw1)
Edit tsrc,nw1,tdest
```

Point	tsrc	nw1	tdest
0	hello	3	hello 3
1	there	5	there 5
2	hello	2	hello 2
3	there	1	there 1
4			

Single-key text sort:

```
Sort tsrc,tdest                 // nw1 not used
```

Point	tsrc	nw1	tdest
0	hello	3	hello 3
1	there	5	hello 2
2	hello	2	there 1
3	there	1	there 5
4			

Execute this to scramble tdest again:

```
tdest= tsrc + " " + num2str(nw1)
```

Execute this to see a two key sort (nw1 breaks ties):

```
Sort {tsrc,nw1},tdest
```

Point	tsrc	nw1	tdest
0	hello	3	hello 2
1	there	5	hello 3
2	hello	2	there 1
3	there	1	there 5
4			

The reason that “hello 3” sorts after “hello 2” is because nw1[0] = 3 is greater than nw1[2] = 2.

You can sort by more than two keys by specifying more than two waves inside the braces.

Sorting Text

You can sort text waves with the Sort operation or the Sorting dialog. See **Multiple Sort Keys** on page III-136 for an example.

By default, text sorting is case-insensitive; “hello” sorts equally with “HELLO”. You can make the sorting case-sensitive by adding the /C flag to the generated command. If you use the Sorting dialog, use the To Cmd Line button, and type /C after Sort in the generated command.

MakeIndex and IndexSort Operations

The MakeIndex and IndexSort operations are infrequently used. You will normally use the Sort operation.

Applications of MakeIndex and IndexSort include:

- Sorting large quantities of data
- Sorting individual waves from a group one at a time
- Accessing data in sorted order without actually rearranging the data
- Restoring data to the original ordering

The MakeIndex operation creates a set of index numbers. IndexSort can then use the index numbers to rearrange data into sorted order. Together they act just like the Sort operation but with an extra wave and an extra step.

The advantage is that once you have the index wave you can quickly sort data from a given set of waves at any time. For example, if you have hundreds of waves you can not use the normal sort operation on a single command line. Also, when writing procedures it is more convenient to loop through a set of waves one at a time than to try to generate a single command line with multiple waves. This is particularly true when not all waves from a given set will fit into memory at one time.

You can also use the index values to access data in sorted order without using the IndexSort operation. For example, if you have data and index waves named wave1 and wave1index, you can access the data in sorted order on the right hand side of a wave assignment like so:

```
wave1[wave1index[p]]
```

If you create an index wave, you can undo a sort and restore data to the original order. To do this, simply use the Sort operation with the index wave as the source.

To understand the MakeIndex operation, consider that the following commands

```
Duplicate data1,data1index
MakeIndex data1,data1index
```

are identical in effect to

```
Duplicate data1,data1index
data1index= P
Sort data1,data1index
```

Like the Sort operation, the MakeIndex operation can handle multiple sort keys.

Decimation

If you have a large data set it may be convenient to deal with a smaller but representative number of points. In particular, if you have a graph with hundreds of thousands of points, it probably takes a long time to draw or print the graph. You can probably do without many of the data points without altering the graph much. Decimation is one way to accomplish this.

There are at least two ways to decimate data:

1. Keep only every nth data value. For example, keep the first value, discard 9, keep the next, discard 9 more, etc. We call this **Decimation by Omission** (see page III-137).
2. Replace every nth data value with the result of some calculation such as averaging or filtering. We call this **Decimation by Smoothing** (see page III-138).

Decimation by Omission

To decimate by omission, create the smaller output wave and use a simple assignment statement (see **Waveform Arithmetic and Assignments** on page II-93) to set their values. For example, If you are decimating by a factor of 10 (omitting 9 out of every 10 values), create an output wave with 1/10th as many points as the input wave.

For example, make a 1000 point test input waveform:

```
Make/O/N=1000 wave0
SetScale x 0, 5, wave0
wave0 = sin(x) + gnoise(.1)
```

Now, make a 100 point waveform to contain the result of the decimation:

```
Make/O/N=100 decimated
SetScale x 0, 5, decimated    // preserve the x range
decimated = wave0[p*10]       // for(p=0;p<100;p+=1) decimated[p]= wave0[p*10]
```

Decimation by omission can be obtained more easily using the Resample operation and dialog by using an interpolation factor of 1 and a decimation factor of (in this case) 10, and a filter length of 1.

```
Duplicate/O wave0, wave0Resampled
Resample/DOWN=10/N=1 wave0Resampled
```

Note: Before Igor 6.01, the minimum filter length was 3. For backwards compatibility with Igor 6.0 you must instead use /N=3 and a window type whose first and last values are 0 and whose middle value is 1. Most windows do this, including the default Hanning and the Bartlett windows; /WINF=None doesn't. The result is the same as if /N=1 were specified with Igor 6.01 or later.

```
Resample/DOWN=10/N=3 wave0Resampled  // Works with Igor 6.0 and 6.01 or later
```

Decimation by Smoothing

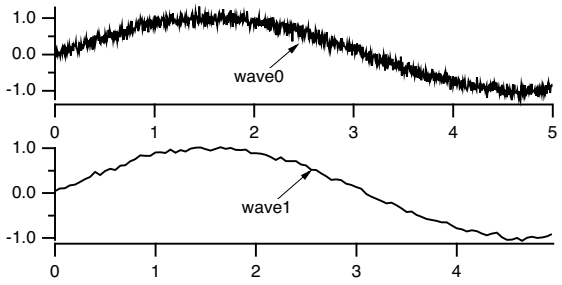
While decimation by omission completely discards some of the data, decimation by smoothing combines all of the data into the decimated result. The smoothing can take many forms: from simple averaging to various kinds of lowpass digital filtering.

The simplest form of smoothing is averaging (sometimes called "boxcar" smoothing). You can decimate by averaging some number of points in your original data set. If you have 1000 points, you can create a 100 point representation by averaging every set of 10 points down to one point. For example, make a 1000 point test waveform:

```
Make/O/N=1000 wave0
SetScale x 0, 5, wave0
wave0 = sin(x) + gnoise(.1)
```

Now, make a 100 point waveform to contain the result of the decimation:

```
Make/O/N=100 wave1
SetScale x 0, 5, wave1
wave1 = mean(wave0, x, x+9*deltax(wave0))
```



Notice that the output wave, wave1, has one tenth as many points as the input wave.

The averaging is done by the waveform assignment

```
wave1 = mean(wave0, x, x+9*deltax(wave0))
```

This evaluates the right-hand expression 100 times, once for each point in wave1. The symbol "x" returns the X value of wave1 at the point being evaluated. The right-hand expression returns the average value of wave0 over the segment that corresponds to the point in wave1 being evaluated.

It is essential that the X values of the output wave span the same range as the X values of the input range. In this simple example, the SetScale commands satisfy this requirement.

There is a WaveMetrics-supplied macro for decimation. To use it, include the "Decimation" file. **The Include Statement** on page IV-145 describes how to include a procedure file.

Results similar to the example above can be obtained more easily using the **Resample** operation (page V-525) and dialog.

Resample is based on a general sample rate conversion algorithm that optionally interpolates, low-pass filters, and then optionally decimates the data by omission. The lowpass filter can be set to "None" which averages an odd number of values centered around the retained data points. So decimation by a factor of 10 would involve averaging 11 values centered around every 10th point.

The decimation by averaging above can be changed to be 11 values centered around the retained data point instead 10 values from the beginning of the retained data point this way:

```
Make/O/N=100 wave1Centered
SetScale x 0, 5, wave1Centered
wave1Centered = mean(wave0, x-5*deltax(wave0), x+5*deltax(wave0))
```

Each decimated result (each average) is formed from different values than wave1 used, but it isn't any less valid as a representation of the original data.

Using the Resample operation:

```
Duplicate/O wave0, wave2
Resample/DOWN=10/WINF=None/N=11 wave2       // no /UP means no interpolation
```

gives nearly identical results to the wave1Centered = mean(…) computation, the exceptions being only the initial and final values, which are simple end-effect variations.

The /WINF and /N flags of Resample define simple low-pass filtering options for a variety of decimation-by-smoothing choices. The default /WINF=Hanning window gives a smoother result than /WINF=None. See the **WindowFunction** operation (page V-738) for more about these window options.

Miscellaneous Operations

WaveTransform

When working with large amounts of data (many waves or multiple large waves), it is frequently useful to replace various wave assignments with wave operations which execute significantly faster. The **WaveTransform** operation (see page V-732) is designed to help in these situations. For example, to flip the data in a 1D wave you can execute the following code:

```
Function flipWave(inWave)
   wave inWave

   Variable num=numPnts(inWave)
   Variable n2=num/2
   Variable i,tmp
   num-=1
   Variable j
   for(i=0;i<n2;i+=1)
      tmp=inWave[i]
      j=num-i
      inWave[i]=inWave[j]
      inWave[j]=tmp
   endfor
End
```

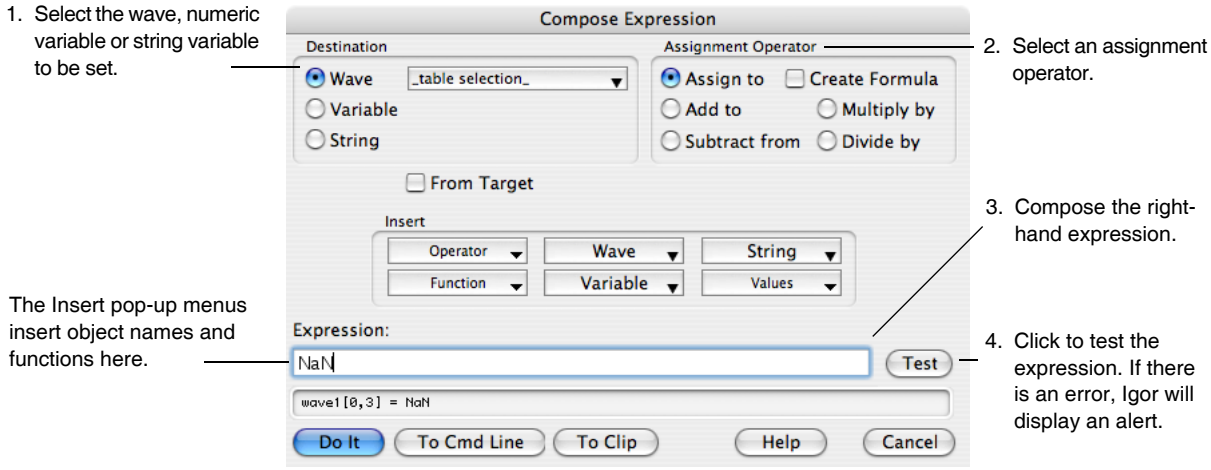
You can obtain the same result (about 250 times faster) using the command

```
WaveTransform/O flip waveName
```

In addition to "flip", WaveTransform can also fill a wave with point index or the inverse point index, shift data points, normalize, convert to complex-conjugate, compute the squared magnitude or the phase, etc.

Compose Expression Dialog

The Compose Expression item in the Analysis menu brings up the Compose Expression dialog.



This dialog generates a command that sets the value of a wave, variable or string based on a numeric or string expression created by pointing and clicking. Any command that you can generate using the dialog could also be typed directly into the command line.

The command that you generate with the Compose Expression dialog consists of three parts: the destination, the assignment operator and the expression. The command resembles an equation and is of the form:

```
<destination> <assignment-operator> <expression>
```

For example:

```
wave1 = K0 + wave2              // a wave assignment command
K0 += 1.5 * K1                  // a variable assigment command
str1 = "Today is" + date()      // a string assignment command
```

Table Selection Item

The Destination Wave pop-up menu contains a "_table selection_" item. When you choose "_table selection_", Igor assigns the expression to whatever is selected in the table. This could be an entire wave or several entire waves, or it could be a subset of one or more waves.

To use this feature, start by selecting in a table the numeric wave or waves to which you want to assign a value. Next, choose Compose Expression from the Analysis menu. Choose "_table selection_" in the Destination Wave pop-up menu. Next, enter the expression that you want to assign to the waves. Notice the command that Igor has created which is displayed in the command box toward the bottom of the dialog. If you have selected a subset of a wave, Igor will generate a command for that part of the wave only. Finally, click Do It to execute the command.

Create Formula Checkbox

The Create Formula checkbox in the Compose Expression dialog generates a command using the := operator rather than the = operator. The := operator establishes a dependency such that, if a wave or variable on the right hand side of the assignment statement changes, Igor will reassign values to the destination (left hand side). We call the right hand side a formula. Chapter IV-9, **Dependencies**, provides details on dependencies and formulas.

Matrix Math Operations

There are three basic methods for performing matrix calculations: normal wave expressions, the matrix*XXX* operations, and the **MatrixOp** operation.

Normal Wave Expressions

You can add matrices to other matrices and scalars using normal wave expressions. You can also multiply matrices by scalars. For example:

```
Make matA={{1,2,3},{4,5,6}}, matB= {{7,8,9},{10,11,12}}
matA= matA+0.01*matB
```

gives new values for

```
matA ={{1.07,2.08,3.09},{4.1,5.11,6.12}}
```

matrix*XXX* Operations

A number of matrix operations are implemented in Igor; most have names starting with the word "matrix". For example, you can multiply a string of matrices (and column and row vectors) using the **MatrixMultiply** operation (page V-376). This operation. The /T flag allows you to specify that a given matrix's data should be transposed before being used in the multiplication.

Many of Igor's matrix operations use the LAPACK library. To learn more about LAPACK see:

LAPACK Users' Guide, 3rd ed., SIAM Publications, Philadelphia, 1999.

or the LAPACK web site:

```
http://www.netlib.org/lapack/lug/lapack_lug.html
```

Unless noted otherwise, LAPACK routines support real or complex, IEEE single and double precision matrix waves. Most matrix operations create the variable V_flag and set it to zero if the operation is successful. If the flag is set to a negative number it indicates that one of the parameters passed to the LAPACK routines is invalid. If the flag value is positive it usually indicates that one of the rows/columns of the input matrix caused the problem.

MatrixOp Operation

The **MatrixOp** operation (page V-377) improves the execution efficiency and simplifies the syntax of matrix expressions. For example, the expression

```
MatrixOp matA = (matD - matB x matC) x matD
```

is equivalent to matrix multiplications and subtraction following standard precedence rules.

Matrix Commands

Here are the matrix math operations and functions. For full documentation, see Chapter V-1, **Igor Reference**.

General:

```
MatrixConvolve coefMatrix, dataMatrix
MatrixCorr [flags] waveA [, waveB]
MatrixDet(matrixA)
MatrixDot(waveA, waveB)
MatrixFilter [flags] Method  dataMatrix
MatrixMultiply matrixA[/T], matrixB[/T] [, additional matrices]
MatrixOp [/O] destwave = matrixExpression
MatrixRank(matrixA [, maxConditionNumber])
MatrixTrace(matrixA)
MatrixTranspose [/H] matrix
```

EigenValues, eigenvectors and decompositions:

```
MatrixEigenV [flags] matrixWave
```

```
MatrixInverse [flags] srcWave
MatrixLUD matrixA
MatrixSchur [/Z] srcMatrix
MatrixSVD matrixA
```

Linear equations and least squares:

```
MatrixGaussJ matrixA, vectorsB
MatrixLinearSolve [flags] matrixA  matrixB
MatrixLLS [flags] matrixA  matrixB
MatrixLUBkSub matrtixL, matrixU, index, vectorB
MatrixSolve method, matrixA, vectorB
MatrixSVBkSub matrixU, vectorW, matrixV, vectorB
```

Macintosh and LAPACK Library

Any matrix operation that uses the LAPACK library will use Apple's veclib implementation if it is available. On computers that include Velocity Engine, single-precision matrix operations may use the Velocity Engine. It is possible to disable use of veclib using the **SetIgorOption** operation (see page V-562):

```
SetIgorOption UseVeclib=[1 or 0 or ?]
```

Analysis Programming

This section contains data analysis programming examples. There are many more examples in the WaveMetrics Procedures, Igor Technical Notes, and Sample Experiments folders.

Passing Waves to User Functions and Macros

As you look through various examples you will notice two different ways to pass a wave to a function: using a Wave parameter or using a String parameter.

Using a Wave Parameter	Using a String Parameter
`Function Test1(w)` `  Wave w`	`Function Test2(wn)` `  String wn`
Usable in functions, not in macros.	Usable in functions and macros.
w is a "formal" name. Use it just as if it were the name of an actual wave.	Use the $ operator to convert from a string to wave name.

The string method is used in macros and in user functions for passing the name of a wave that the function is to create or for passing the base name of a family of waves. The wave parameter method is used in user functions when the wave will always exist before the function is called. For details, see **Accessing Waves in Functions** on page IV-65.

Returning Created Waves from User Functions

A function can return only a number or string, not a wave. But functions can return the name (or better, the full path) of the created wave(s). See also **Accessing Waves in Functions** on page IV-65.

A function that creates a single wave can be defined as a string function that returns the path to the created wave. The calling routine uses that string to refer to the wave:

```
Function CallingFunction()
   String pathToWave= fCreateANoiseWave(5)
   WAVE w = $pathToWave
   WaveStats w
   Print NameOfWave(w)        // Prints (only) the name of the created wave
End
```

```
Function/S fCreateANoiseWave(noiseValue)
   Variable noiseValue

   Make/O theNoiseWave= gnoise(noiseValue)   // The same name, or pass a name
   return GetWavesDataFolder(theNoiseWave,2) // String is full path to wave
End
```

To return paths to more than one wave, use **Pass-By-Reference** on page IV-45:

```
Function CallingFunction()
   String pw1, pw2
   pbrCreateTwoNoiseWaves(5,3,pw1,pw2)
   WAVE w1 = $pw1
   WAVE w2 = $pw2
End

Function/S pbrCreateTwoNoiseWaves(noise1,noise2,path1,path2)
   Variable noise1, noise2 // Inputs
   String &path1, &path2   // Outputs (pass-by-reference ala Fortran)

   Make/O noiseWave1= gnoise(noise1)
   path1= GetWavesDataFolder(noiseWave1,2)

   Make/O noiseWave2= gnoise(noise2)
   path1= GetWavesDataFolder(noiseWave2,2)

   return 0                      // Or some other useful value
End
```

Returning Created Waves from Macros

A Macro or Proc can not return any value and functions can return only a number or string, so how can you write one that passes back to the calling routine a wave created in the subroutine?

For a macro, you'll need to take advantage of the fact that waves are global objects with names. Pass to the subroutine a name parameter to be used to create the wave. The calling routine then knows what the name of the created wave is because it supplied the name:

```
Macro CallingRoutine()
   String name="noiseWave"
   CreateANoiseWave(name,5)
   WaveStats $name
End

Proc CreateANoiseWave(name,noiseValue)
   String name                   // Create a wave with this name
   Variable noiseValue           // with this much noise

   Make/O $name= gnoise(noiseValue)
End
```

If CreateANoiseWave needs to create more than one wave, pass more than one name parameter. This technique can also be used by functions as in the **WavesAverage Example** on page III-146.

A method that works (but is bug-prone) is to just document the name of the wave created by the called routine and use that name in the calling routines. The problem with that is when you change the name of the created wave you need to update each calling routine.

Writing Functions that Process Waves

The user function is a powerful, general-purpose analysis tool. You can do practically any kind of analysis. However, complex analyses require programming skill and patience.

It is useful to think about an analysis function in terms of its input parameters, its return value and any side effects it may have. By return value, we mean the value that the function directly returns. For example, a function might return a mean or an area or some other characteristic of the input. By side effects, we mean

changes that the function makes to any objects. For example, a function might change the values in a wave or create a new wave.

This table shows some of the common types of analysis functions.

Input Parameters	Return Value	Side Effects	Example Function
A source wave	A number	None	WaveArea
A source wave	Not used	The source wave is modified	RemoveOutliers
A source wave	String	A new destination wave is created	LogRatio
A source wave and a destination wave	Not used	The destination wave is modified	
The base name of a family of waves	A number	None	WavesMax
The base name of a family of waves	String	One or more new waves are created	WavesAverage

It is also possible to write an analysis function that has both a meaningful return value and side effects.

The following example functions are intended to show you the general form for some common analysis function types. We have tried to make the examples useful while keeping them simple.

WaveSum Example

Input: Source wave
Return value: Number
Side effects: None

```
// WaveSum(w)
// Returns the sum of the entire wave, just like Igor's sum function.
Function WaveSum(w)
   Wave w

   Variable i, n=numpnts(w), total=0
   for(i=0;i<n;i+=1)
      total += w[i]
   endfor

   return total
End
```

To use this, you would execute something like

```
Print "The sum of wave0 is:", WaveSum(wave0)
```

RemoveOutliers Example

Input: Source wave
Return value: Number
Side effects: Source wave is modified

Often a user function used for number-crunching needs to loop through each point in an input wave. The following example illustrates this.

```
// RemoveOutliers(theWave, minVal, maxVal)
// Removes all points in the wave below minVal or above maxVal.
// Returns the number of points removed.
Function RemoveOutliers(theWave, minVal, maxVal)
   Wave theWave
   Variable minVal, maxVal
```

```
   Variable i, numPoints, numOutliers
   Variable val
   numOutliers = 0
   numPoints = numpnts(theWave)        // number of times to loop

   for (i = 0; i < numPoints; i += 1)
      val = theWave[i]
      if ((val < minVal) || (val > maxVal))  // is this an outlier?
         numOutliers += 1
      else                                   // if not an outlier
         theWave[i - numOutliers] = val      // copy to input wave
      endif
   endfor

   // Truncate the wave
   DeletePoints numPoints-numOutliers, numOutliers, theWave
   return numOutliers
End
```

To test this function, try the following commands.

```
Make/O/N=10 wave0= gnoise(1); Edit wave0
Print RemoveOutliers(wave0, -1, 1), "points removed"
```

RemoveOutliers uses the for loop to iterate through each point in the input wave. It uses the built-in numpnts function to find the number of iterations required and the local variable p as the loop index. This is a very common practice.

The line "`if ((val < minVal) || (val > maxVal))`" decides whether a particular point is an outlier. `||` is the logical OR operator. It operates on the logical expressions "`(val < minVal)`" and "`(val > maxVal)`". This is discussed in detail under **Bitwise and Logical Operators** on page IV-33.

To use the WaveMetrics-supplied RemoveOutliers function, include the Remove Points.ipf procedure file:

```
#include <Remove Points>
```

See **The Include Statement** on page IV-145 for instructions on including a procedure file.

LogRatio Example

Input: Source waves
Return value: String
Side effects: Destination wave created

```
// LogRatio(source1, source2)
// Creates a new wave that is the log of the ratio of input waves.
// Returns full path to destination wave as a string.
Function/S LogRatio(source1, source2)
   Wave source1, source2

   String destName = NameOfWave(source1) + "_" + NameOfWave(source2)
   destName = CleanupName(destName,1)      // obey name length limitation

   Duplicate/O source1, $destName
   WAVE dest = $destName
   dest = log(source1/source2)
   return GetWavesDataFolder(dest,2)       // string is full path to wave
End
```

To use this in a macro, you would execute something like

```
Display $LogRatio(wave0, wave1)
```

To use this in a function, you would execute something like

```
String pathToWave= LogRatio(wave0, wave1)
Display $pathToWave
```

The "Wave dest = $destName" line creates a local wave reference which refers to the wave whose name is in the destName string variable. This is necessary because "$destName = <expression>" is not allowed in user functions.

This simple function illustrates two commonly used techniques.

The first technique is the algorithmic derivation of a destination wave name based on a source wave name. The algorithm used here is to concatenate the two wave names. This could result in a name longer than the 31 character wave name limit, which is corrected here by the **CleanupName** function (see page V-50). A better function would check for this and use an alternate name if necessary. Another common algorithm is to derive the destination wave name by appending a suffix to the source wave name.

The second technique is the use of Duplicate/O to generate a destination wave. Using Duplicate guarantees that the destination wave has the same number of points, precision, and scaling as the source wave. Using /O (overwrite) prevents an error if the destination wave already exists.

WavesMax Example

Input: Base name
Return value: Number
Side effects: None

```
// WavesMax(baseName)
// Returns the maximum value in all waves whose names start with
//  the specified base name.
Function WavesMax(baseName)
   String baseName

   String wn         // contains the name of a particular wave
   String wl         // contains a list of wave names
   Variable theMax
   Variable index=0

   // get list of waves whose names start with baseName
   wl = WaveList(baseName+"*", ";", "")

   theMax = -INF
   do
      wn = StringFromList(index, wl, ";") // get next wave
      if (strlen(wn) == 0)                // no more names in list?
         break                            // break out of loop
      endif
      WaveStats/Q $wn                  // WaveStats finds max value
      theMax = max(V_max, theMax)      // and puts it in V_max
      index += 1
   while (1)      // do unconditional loop

   return theMax
End
```

This function illustrates the common technique of iterating through a list of waves.

WavesAverage Example

Input: Base name
Return value: String
Side effects: Creates destination wave

```
// WavesAverage(baseName, destName)
// Produces a new wave, each point of which contains the average of the
//  corresponding points of a number of source waves.
// All waves whose name starts with the specified base name are source waves.
// This function assumes that all waves that start with the base name have
//  the same number of points and that there is at least one such wave.
```

```
// Returns full path to destination wave as a string.
Function/S WavesAverage(baseName, destName)
   String baseName      // name for source wave
   String destName      // name for destination wave

   String wn      // contains the name of a particular wave
   String wl      // contains a list of wave names
   Variable index=0

   // get list of waves whose names start with baseName
   wl = WaveList(baseName+"*", ";", "")

   // Make destination wave based on the first source wave
   wn = StringFromList(0, wl)
   Duplicate/O $wn, $destName

   WAVE dest = $destName   // create wave reference for destination
   dest = 0

   do
      wn = StringFromList(index, wl)      // get next wave
      if (strlen(wn) == 0)    // no more names in list?
         break                // break out of loop
      endif
      WAVE source = $wn       // create wave reference for source
      dest += source          // add source to dest
      index += 1
   while (1)   // do unconditional loop

   dest /= index           // divide by number of waves
   return GetWavesDataFolder(dest,2)// string is full path to wave
End
```

The name of the destination wave is passed in as a parameter. A wave with that name is created using Duplicate. We iterate through the list of waves using the **StringFromList** operation (see page V-675). We need to use WAVE references for both the source waves and the destination wave because of the limitations on the use of the $ operator in a function.

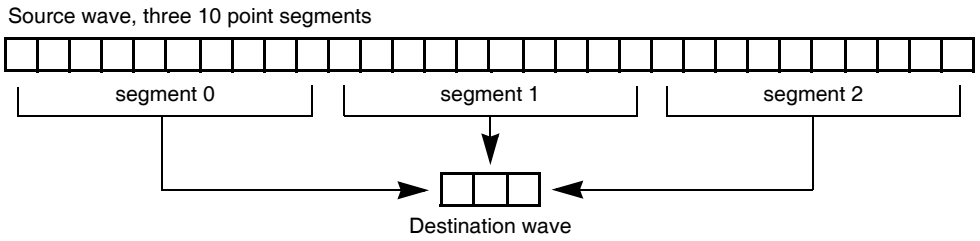
Finding the Mean of Segments of a Wave

An Igor user who considers each of his waves to consist of a number of segments with some number of points in each segment asked us how he could find the mean of each of these segments. We wrote the FindSegmentMeans function to do this.

```
Menu "Macros"
 "Find Segment Means", FindSegmentMeans()
End

Function FindSegmentMeans()
   String source           // name of wave that we want to analyze
   Variable n              // number of points in each segment
   Prompt source, "Source wave", popup WaveList("*", ";", "")
   Prompt n, "Number of points in each segment"
   DoPrompt "Find Segment Means", source,n

   SegmentMeans($source, n)
End

Function/S SegmentMeans(source, n)
   Wave source
   Variable n
```

```
   String dest                    // name of destination wave
   Variable segment, numSegments
   Variable startX, endX, lastX

   dest = NameOfWave(source)+"_m"  // derive name of dest from source
   numSegments = trunc(numpnts(source) / n)
   if (numSegments < 1)
      DoAlert 0, "Destination must have at least one point"
      return ""
   endif
   Make/O/N=(numSegments) $dest
   WAVE destw = $dest
   lastX = pnt2x(source, numpnts(source)-1)

   for (segment = 0; segment < numSegments; segment += 1)
      startX = pnt2x(source, segment*n)      // start X for segment
      endX = pnt2x(source, (segment+1)*n - 1)// end X for segment
      // this handles case where numpnts(source)/n is not an integer
      endX = min(endX, lastX)
      destw[segment] = mean(source, startX, endX)
   endfor
   return GetWavesDataFolder(destw,2)// string is full path to wave
End
```

This diagram illustrates a source wave with three ten-point segments and a destination wave that will contain the mean of each of the source segments. The macro makes the destination wave.



To test FindSegmentMeans, try the following commands.

```
Make/N=100 wave0=p+1; Edit wave0
FindSegmentMeans(wave0,10)
Append wave0_m
```

The loop index is the variable “segment”. It is the segment number that we are currently working on, and also the number of the point in the destination wave to set.

Using the segment variable, we can compute the range of points in the source wave to work on for the current iteration: segment*n up to (segment+1)*n - 1. Since the mean function takes arguments in terms of a wave’s X values, we use the pnt2x function to convert from a point number to an X value.

We wrote this in three parts: two functions and a menu definition. The FindSegmentMeans function uses Prompt statements along with DoPrompt to make a simple dialog for entering the parameters for the function. Users of previous versions of Igor will recognize that this capability used to be available only in macros.

The SegmentMeans function does the actual work. The two functions are partitioned this way so that you can call SegmentMeans in another function or on the command line without having to use the dialog.

Finally, the menu definition makes an entry in the Macros menu so that it is convenient to invoke the dialog.

If it is guaranteed that the number of points in the source wave is an integral multiple of the number of points in a segment, then the function can be speeded up and simplified by using a waveform assignment statement in place of the loop. Here is the statement.

```
destw = mean(source, pnt2x(source,p*n), pnt2x(source,(p+1)*n-1))
```

The variable p, which Igor automatically increments as it evaluates successive points in the destination wave, takes on the role of the segment variable used in the loop. Also, the startX, endX and lastX variables are no longer needed.

Using the example shown in the diagram, p would take on the values 0, 1 and 2 as Igor worked on the destination wave. n would have the value 10.

Computing a Logarithmic Histogram

The built-in Histogram operation always uses bins of equal width. Here is some code that uses logarithmic bins.

This code is split into two user functions, LogHist and DoLogHist, using the same organization that we used for the previous example. The function LogHist calls DoLogHist to do the low-level, point-by-point computations. LogHist supplies a graphical user interface using Prompt and DoPrompt, while the DoLogHist function can be called from a user function without invoking a dialog. The menu definition provides a convenient way to invoke the LogHist function.

LogHist produces two destination waves - an XY pair. The X wave holds the coordinates of the start of the bins and the Y wave holds the counts. The waves are named using the name of the source wave with "_hx" and "_hy" suffixes.

```
#pragma rtGlobals=1        // Use modern global access method.

Menu "Analysis"
    "Log Histogram...", LogHist()
End

// DoLogHist(sw, dwX, dwY, startX, logDeltaX)
// Creates the logarithmic histogram of the source wave by summing the
// appropriate numbers into the destination y wave.
// sw is the source wave.
// dwX is the destination x wave.
// dwY is the destination y wave.
// startX, logDeltaX are explained below in LogHist().
Function DoLogHist(sw, dwX, dwY, startX, logDeltaX)
    Wave sw, dwX, dwY
    Variable startX, logDeltaX
    Variable pt, pp, dpnts, spnts

    // first find bin edges and put them in dwX
    dpnts = numpnts(dwX)
    for (pt = 0; pt < dpnts; pt += 1)
        dwX[pt]= 0^(pt*logDeltaX+startX)  // this value is 10^startX when p == 0
    endfor

// now find which bin of dwY each Y value in sw belongs in and increment it.
    spnts = numpnts(sw)
    for (pt = 0; pt < spnts; pt += 1)
        pp = (log(sw[pt]) - startX) / logDeltaX
        if( pp == limit(pp,0,dpnts) )     // unless it is out of range or NaN
            dwY[pp] += 1
        endif
    endfor
End

// LogHist(sourceWave, numDecades, startDecade, binsPerDecade)
// Creates XY pair of waves that represent the logarithmic histogram of the
// source wave. If the source wave is named "data" then the output waves will
// be named "data_hx" and "data_hy".
// The product of numDecades and binsPerDecade specifies the number of bins in
// the histogram.
// startDecade specifies the X coordinate of the left edge of first bin. The bin
// starts at 10^startDecade.
// binsPerDecade specifies the bin width. Values less than 1 result in bins that
// span multiple decades. For example, set binsPerDecade to 0.5 to create bins
// that span two decades.
// Example
// Make/N=100 test = 10^(1+abs(gnoise(3)))
// Display test; ModifyGraph log(left)=1, mode=8,msize=2
// LogHist("test",12,0,1)
```

```
Function LogHist()

    String sourceWave= StrVarOrDefault("root:Packages:LogHist:sourceWave","_demo_")
    Variable numDecades = NumVarOrDefault("root:Packages:LogHist:numDecades",10)
    // first bin at 0.0001
    Variable startDecade = NumVarOrDefault("root:Packages:LogHist:startDecade",-4)
    Variable binsPerDecade= NumVarOrDefault("root:Packages:LogHist:binsPerDecade",1)
    Prompt sourceWave, "Source wave", popup "_demo_;"+WaveList("*", ";", "")
    Prompt startDecade, "start decade (first bin starts at 10^startDecade)"
    Prompt numDecades, "Number of decades in destination waves"
    Prompt binsPerDecade, "bins per decade"
    DoPrompt "Log Histogram", sourceWave, numDecades, startDecade, binsPerDecade

    if( CmpStr(sourceWave,"_demo_") == 0 )
        sourceWave= "demoData"
        Make/O/N=100 $sourceWave=0.0001+10^(gnoise(2))       // about 10^-4 to 10^6
        CheckDisplayed/A $sourceWave
        if( V_Flag == 0 )
            Display $sourceWave; ModifyGraph log(left)=1, mode=8,msize=2
        endif
        startDecade=-4
        numDecades=10
        binsPerDecade=1
    endif
    if( binsPerDecade < 0 )
        binsPerDecade = 1
    endif
    // Save values for next attempt
    NewDataFolder/O root:Packages
    NewDataFolder/O root:Packages:LogHist
    String/G root:Packages:LogHist:sourceWave = sourceWave
    Variable/G root:Packages:LogHist:numDecades= numDecades
    Variable/G root:Packages:LogHist:startDecade= startDecade
    Variable/G root:Packages:LogHist:binsPerDecade= binsPerDecade
    String destXWave, destYWave
    // Concoct names for dest waves.
    // This does not work if sourceWave is a full or partial path requiring single
    // quotes (e.g., root:Data:'wave 0').
    Variable numBins= numDecades * binsPerDecade
    Variable logDeltaX=1/binsPerDecade      // Log delta X (1 gives 1 decade per bin)
    destXWave = sourceWave + "_hx"
    destYWave = sourceWave + "_hy"
    Make/O/N=(numBins+1) $destXWave=0, $destYWave=0
    DoLogHist($sourceWave, $destXWave, $destYWave, startDecade, logDeltaX)
    CheckDisplayed/A $destYWave
    if( V_Flag == 0 )
        Display $destYWave vs $destXWave
        AutoPositionWindow/E/M=1
        ModifyGraph mode=4, marker=19, log(bottom)=1
    endif
End
```

To test LogHist, choose “Log Histogram” from the Analysis menu, and choose “_demo_” from the Source wave pop-up menu in the resulting dialog.

To use the WaveMetrics-supplied logarithmic histogram procedures, include the “Log Histogram” procedure file. See **The Include Statement** on page IV-145 for instructions on including a procedure file.

Working with Mismatched Data

Occasionally, you may find yourself with several sets of data each sampled at a slightly different rate or covering a different range of the independent variable (usually time). If all you want to do is create a graph showing the relationship between the data sets then there is no problem.

However, if you want to subtract one from another or do other arithmetic operations then you will need to either:

- Create representations of the data that have matching X values. Although each case is unique, usually you will want to use the Interpolate XOP (see **Using the Interpolate External Operation** on page III-118) or the interp function (see **Using the Interp Function** on page III-117) to create data sets with common X values. You can also use the **Resample** operation (page V-525) to create a wave to match another.

- Properly set each wave's X scaling, and perform the waveform arithmetic using X scaling values and Igor's automatic linear interpolation. See **Mismatched Waves** on page II-99.

The WaveMetrics procedure file Wave Arithmetic Panel uses these techniques to perform a variety of operations on data in waves. You can access the panel by choosing Packages→Wave Arithmetic from the Analysis menu. This will open the procedure file and display the control panel. Click the help button in the panel to learn how to use it.

References

Press, W.H., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

Reinsch, Christian H., Smoothing by Spline Functions, *Numerische Mathematic, 10*, 177-183, 1967.

Chapter
III-8

Curve Fitting

Overview

Igor Pro's curve fitting capability is one of its strongest analysis features. Here are some of the highlights.

- Linear and general nonlinear curve fitting.
- Fit by ordinary least squares, or by least orthogonal distance for errors-in-variables models.
- Fit to implicit models.
- Built-in functions for common fits.
- Automatic initial guesses for built-in functions.
- Fitting to user-defined functions of any complexity.
- Fitting to functions of any number of independent variables, either gridded data or multicolumn data.
- Fitting to a sum of fit functions.
- Fitting to a subset of a waveform or XY pair.
- Produces estimates of error.
- Supports weighting.

The idea of curve fitting is to find a mathematical model that fits your data. We assume that you have theoretical reasons for picking a function of a certain form. The curve fit finds the specific coefficients which make that function match your data as closely as possible.

You cannot use curve fitting to find which of thousands of functions fit a data set.

People also use curve fitting to merely show a smooth curve through their data. This sometimes works but you should also consider using smoothing or interpolation, which are described in Chapter III-7, **Analysis**.

You can fit to three kinds of functions:

- Built-in functions.
- User-defined functions.
- External functions (XFUNCs).

The built-in fitting functions are line, polynomial, sine, exponential, double-exponential, Gaussian, Lorentzian, Hill equation, sigmoid, lognormal, Gauss2D (two-dimensional Gaussian peak) and Poly2D (two-dimensional polynomial).

You create a user-defined function by entering the function in the New Fit Function dialog. Very complicated functions may have to be entered in the Procedure window.

External functions, XFUNCs, are written in C or C++. To create an XFUNC, you need the optional "Igor External Operations Toolkit" and a C/C++ compiler. You don't need the toolkit to *use* an XFUNC that you get from WaveMetrics or from another user.

Curve fitting works with equations of the form $y = f(x_1, x_2, \ldots, x_n)$; although you can fit functions of any number of independent variables (the x_n's) most cases involve just one. For more details on multivariate fitting, see **Fitting to a Multivariate Function** on page III-180.

You can also fit to implicit functions; these have the form $f(x_1, x_2, \ldots, x_n) = 0$. See **Fitting Implicit Functions** on page III-210.

You can do curve fits with linear constraints (see **Fitting with Constraints** on page III-197).

Curve Fitting Terminology

Built-in fits are performed by the CurveFit operation. User-defined fits are performed by the FuncFit or FuncFitMD operation. We use the term "curve fit operation" to stand for CurveFit, FuncFit, or FuncFitMD, whichever is appropriate.

Fitting to an external function works the same as fitting to a user-defined function (with some caveats concerning the Curve Fitting dialog — see **Fitting to an External Function (XFUNC)** on page III-174).

If you use the Curve Fitting dialog, you don't really need to know much about the distinction between built-in and user-defined functions. You may need to know a bit about the distinction between external functions and other types. This will be discussed later.

We use the term "coefficients" for the numbers that the curve fit is to find. We use the term "parameters" to talk about the values that you pass to operations and functions.

Overview of Curve Fitting

In curve fitting we have raw data and a function with unknown coefficients. We want to find values for the coefficients such that the function matches the raw data as well as possible. The "best" values of the coefficients are the ones that minimize the value of Chi-square. Chi-square is defined as:

$$\sum_i \left(\frac{y - y_i}{\sigma_i}\right)^2$$

where y is a fitted value for a given point, y_i is the measured data value for the point and σ_i is an estimate of the standard deviation for y_i.

The simplest case is fitting to a straight line: $y = ax + b$. Suppose we have a theoretical reason to believe that our data should fall on a straight line. We want to find the coefficients a and b that best match our data.

For a straight line or polynomial function, we can find the best-fit coefficients in one step. This is noniterative curve fitting, which uses the singular value decomposition algorithm for polynomial fits.

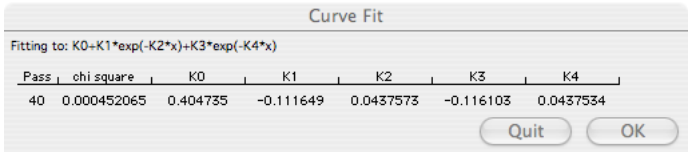
Iterative Fitting

For the other built-in fitting functions and for user-defined functions, the operation is iterative as the fit tries various values for the unknown coefficients. For each try, it computes chi-square searching for the coefficient values that yield the minimum value of chi-square.

The Levenberg-Marquardt algorithm is used to search for the coefficient values that minimize chi-square. This is a form of nonlinear, least-squares fitting.

As the fit proceeds and better values are found, the chi-square value decreases. The fit is finished when the rate at which chi-square decreases is small enough.

During an iterative curve fit, you will see the Curve Fit progress window. This shows you the function being fit, the updated values of the coefficients, the value of chi-square, and the number of passes.



Normally you will let the fit proceed until completion when the Quit button is disabled and the OK button is enabled. When you click OK, the results of the fit are written in the history area.

If the fit has gone far enough and you are satisfied, you can click the Quit button, which finishes the iteration currently under way and then puts the results in the history area as if the fit had completed on its own.

Sometimes you can see that the fit is not working, e.g., when chi-square is not decreasing or when some of the coefficients take on very large nonsense values. You can abort it by pressing Command-period (*Macintosh*) or Ctrl+Break (*Windows*), which discards the results of the fit. You will need to adjust the fitting coefficients and try again.

Initial Guesses

The Levenberg-Marquardt algorithm is used to search for the minimum value of chi-square. Chi-square defines a surface in a multidimensional error space. The search process involves starting with an initial guess at the coefficient values. Starting from the initial guesses, the fit searches for the minimum value by travelling down hill from the starting point on the chi-square surface.

We want to find the deepest valley in the chi-square surface. This is a point on the surface where the coefficient values of the fitting function minimize, in the least-squares sense, the difference between the experimental data and fit data. Some fitting functions may have only one valley. In this case, when the bottom of the valley is found, the best fit has been found. Some functions, however, may have multiple valleys, places where the fit is better than surrounding values, but it may not be the best fit possible.

When the fit finds the bottom of a valley it concludes that the fit is complete even though there may be a deeper valley elsewhere on the surface. Which valley is found first depends on the initial guesses.

For built-in fitting functions, you can automatically set the initial guesses. If this produces unsatisfactory results, you can try manual guesses. For fitting to user-defined functions you must supply manual guesses.

Termination Criteria

A curve fit will terminate after 40 passes in searching for the best fit, but will quit if 9 passes in a row produce no decrease in chi-square. This can happen if the initial guesses are so good that the fit starts at the minimum chi-square. It can also happen if the initial guesses are way off or if the function does not fit the data at all.

Unless you know a great deal about the fitting function and the data, it is unwise to assume that a solution is a good one. In almost all cases you will want to see a graph of the solution to compare the solution with the data. You may also want to look at a graph of the residuals, the differences between the fitted model and the data. Igor makes it easy to do both in most cases.

Errors in Curve Fitting

In certain cases you may encounter a situation in which it is not possible to decide where to go next in searching for the minimum chi-square. This results in a "singular matrix" error. This is discussed under **Singularities** on page III-230. **Curve Fitting Troubleshooting** on page III-231 can help you find the solution to the problem.

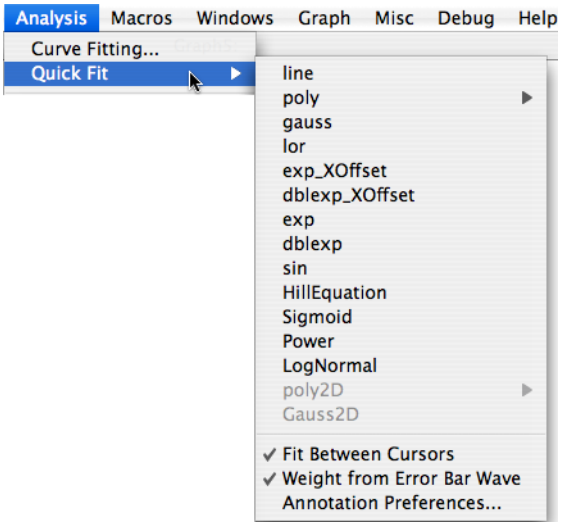
Data for Curve Fitting

You must have measured values of both the dependent variable (usually called "y") and the independent variables (usually called "x" especially if there is just one). These are sometimes called the "response variable" and "explanatory variables." You can do a curve fit to waveform data or to XY data. That is, you can fit data contained in a single wave, with the data values in the wave representing the Y data and the wave's X scaling representing equally-spaced X data. Or you can fit data from two (or more) waves in which the data values in one wave represent the Y values and the data values in another wave represent the X data. In this case, the data do not need to be equally spaced. In fact, the X data can be in random order.

You can read more about waveform and XY data in Chapter II-5, **Waves**.

Curve Fitting Using the Quick Fit Menu

The Quick Fit menu is the easiest, fastest way to do a curve fit.



The Quick Fit menu gives you quick access to curve fits using the built-in fitting functions. The data to be fit are determined by examining the top graph; if a single trace is found, the graphed data is fit to the selected fitting function. If the graph contains more than one trace a dialog is presented to allow you to select which trace should be fit.

The graph contextual menu also gives access to the Quick Fit menu. If you Control-click (*Macintosh*) or right-click (*Windows*) on a trace in a graph, you will see a Quick Fit item at the bottom of the resulting contextual menu. When you access the Quick Fit menu this way, it automatically fits to the trace you clicked on. This gives you a way to avoid the dialog that Quick Fit uses to select the correct trace when there is more than one trace on a graph.

When you use the Quick Fit menu, a command is generated to perform the fit and automatically add the model curve to the graph. By default, if the graph cursors are present, only the data between the cursors is fit. You can do the fit to the entire data set by selecting the Fit Between Cursors item in the Quick Fit menu in order to uncheck the item. When unchecked, fits are done disregarding the graph cursors.

If the trace you are fitting has error bars and the data for the error bars come from a wave, Quick Fit will use the wave as a weighting wave for the fit. Note that this assumes that your error bars represent one standard deviation. If your error wave represents more than one standard deviation, or if it represents a confidence interval, you should not use it for weighting. You can select the Weight from Error Bar Wave item to unmark it, preventing Igor from using the error bar wave for weighting.

By default, a report of curve fit results is printed to the history. If you select Textbox Preferences, the Curve Fit Textbox Preferences dialog is displayed. It allows you to specify that a textbox be added to your graph containing most of the information that is printed in the history. You can select various components of the information by selecting items in the Dialog.

In the screen capture above, the poly2D and Gauss2D fit functions are not available because the top graph does not contain a contour plot or image plot, in which case the fitting functions would be available.

For a discussion of the built-in fit functions, see **Notes on the Built-in Fit Functions** on page III-165.

Limitations of the Quick Fit Menu

The Quick Fit menu does not give you access to the full range of curve fitting options available to you. It does not give you access to user-defined fitting functions, automatic residual calculation, masking, or confidence interval analysis. A Quick Fit always uses automatic guesses; if the automatic guesses don't work, you must use the Curve Fitting dialog to enter manual guesses.

If your graph displays an image that uses auxiliary X and Y waves to set the image pixel sizes, Quick Fit will not be able to do the fit. This is because these waves for an image plot have an extra point that makes them unsuitable for fitting. A contour plot uses X and Y waves that set the centers of the data, and these can be used for fitting. Quick Fit will do the right thing with such a contour plot.

Using the Curve Fitting Dialog

If you want options that are not available via the Quick Fit menu, the next easiest way to do a fit is to choose Curve Fitting from the Analysis menu. This displays the Curve Fitting dialog, which presents an interface for selecting a fitting function and data waves, and for setting various curve fitting options. You can use the

dialog to enter initial guesses if necessary. The Curve Fitting dialog can also be used to create a new user-defined fitting function.

Most curve fits can be accomplished using the Curve Fitting dialog. If you need to do many fits using the same fit function fitting to numerous data sets you will probably want to write a procedure in Igor's programming language to do the job.

The facility for creating a user-defined fitting function using the Curve Fitting dialog will handle most common cases, but is probably not the best way to create very complex fitting function. In such cases, you will need to write a fitting function in a procedure window. This is described later under **User-Defined Fitting Function: Detailed Description** on page III-217.

Some very complicated user-defined fitting functions may not work well with the Curve Fitting dialog. In some cases, you may need to write the fitting function in the Procedure window, and then use the dialog to set up and execute the fit. In other cases it may be necessary to enter the operation manually using either a user procedure or by typing on the command line. These cases should be quite rare.

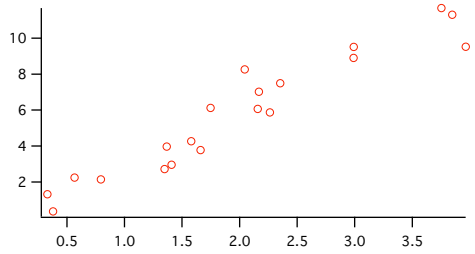
A Simple Case — Fitting to a Built-In Function: Line Fit

To get started, we will cover fitting to a simple built-in fit: a line fit. You may have a theoretical reason to believe that your data should be described by the function $y = ax + b$. You may simply have an empirical observation that the data appear to fall along a line and you now want to characterize this line. It's better if you have a theoretical justification, but we're not all that lucky.

The Curve Fitting dialog is organized into four tabs. Each tab contains controls for some aspect of the fitting operation. Simple fits to built-in functions using default options will require only the Function and Data tab.

We will go through the steps necessary to fit a line to data displayed in a graph. Other built-in functions work much the same way.

You might have data displayed in a graph like this:



Now you wish to find the best-fitting line for this data. The following commands will make a graph like this one, but with the random scatter arranged in a different random way. If you would like to perform the actions yourself as you read the manual, you can make the data shown here and the graph by typing these commands on the command line:

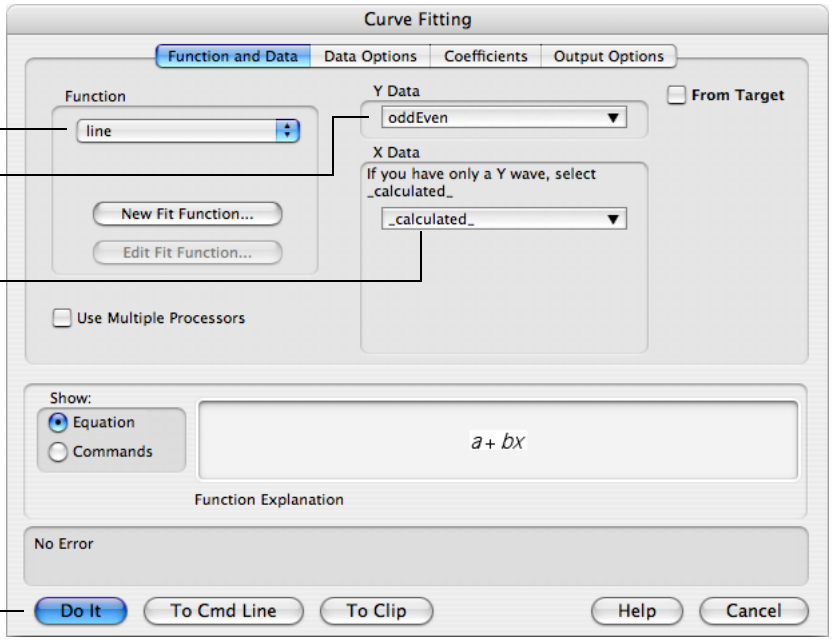
```
Make/N=20/D LineYData, LineXData
SetRandomSeed 0.5   // So the example always makes the same "random" numbers
LineXData = enoise(2)+2             // enoise makes random numbers
LineYData = LineXData*3+gnoise(1)   // so does gnoise
Display LineYData vs LineXData
ModifyGraph mode=3,marker=8
```

The first line makes two waves to receive our "data". The second line fills the X wave with uniformly-distributed random numbers in the range of zero to four. The third line fills the Y wave with data that falls on a line having a slope of three and passing through the origin, with some normally-distributed noise added (your graph will look somewhat different because the noise will have different values). The final two lines make the graph and set the display to markers mode with open circles as the marker.

Function and Data Tab

You display the Curve Fitting dialog by choosing Curve Fitting from the Analysis menu. If you have not used the dialog yet, it looks like this, with the Function and Data tab showing:

If you haven't used the dialog yet, it comes up with the **Function and Data** tab showing.



Step One: choose a fit function from the Function menu.

Step Two: choose data to fit from the Y Data menu.

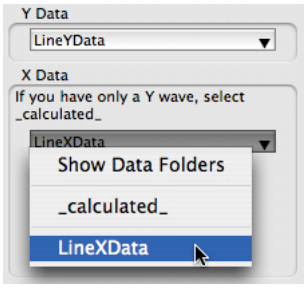
If you have an X wave (as in the example) choose it in the X Data menu.

Step Three: Click the Do It button.

The first step in doing a curve fit is to choose a fit function. We are doing a simple line fit, so pop up the Function menu and choose "line".

Select the Y data from the Y Data menu. If you have waveform data, be sure that the X data menu has "_Calculated_" selected.

If you have separate X and Y data waves, you must select the X wave in the X Data menu. Only waves having the same number of data points as the Y wave are shown in this menu. A mismatch in the number of points is usually the problem if you don't see your X wave in the menu.

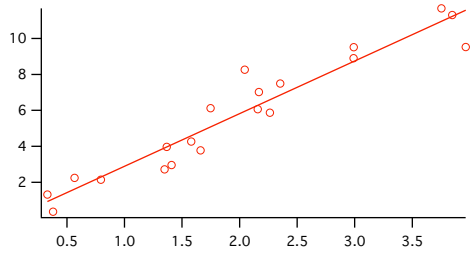


For the line fit example, we select LineYData from the Y Data menu, and LineX-Data from the X Data menu.

Note: The menu shown here is what you see in a pop-up Wave Browser when data folders are hidden. For most users, this is the preferred view. You can, however, select Show Data Folders to display a hierarchical view of the data folders in your experiment. Read about the Wave Browser widget in **Dialog Wave Browser** on page II-181.

If you have a large number of waves in your experiment, it may be easier if you select the From Target checkbox. When it is selected only waves from the top graph or table are shown in the Y and X wave menus, and an attempt is made to select wave pairs used by a trace on the graph.

At this point, everything is set up to do the fit. For this simple case it is not necessary to visit the other tabs in the dialog. When you click Do It, the fit proceeds. The line fit example graph winds up looking like this:



In addition to the model line shown on the graph, various kinds of information appears in the history window:

Command line generated by the dialog.

```
•CurveFit line LineYData /X=LineXData /D
   fit_LineYData= W_coef[0]+W_coef[1]*x
   W_coef={-0.037971,2.9298}
   V_chisq= 18.25; V_npnts= 20; V_numNaNs= 0; V_numINFs= 0;
   V_startRow= 0;V_endRow= 19;V_q= 1;V_Rab= -0.879789;
   V_Pr= 0.956769;V_r2= 0.915408;
   W_sigma={0.474,0.21}
   Coefficient values ± one standard deviation
      a   =-0.037971 ± 0.474
      b   =2.9298 ± 0.21
```

This line can be copied and used to reevaluate the model curve.

Fit coefficients as a wave.

Standard deviations of the Fit coefficients as a wave.

Coefficient values in a list using the names shown in the dialog.

Two Useful Additions: Holding a Coefficient and Generating Residuals

Well, you've done the fit and looked at the graph and you decide that you have reason to believe that the line should go through the origin. Because of the scatter in the measured Y values, the fit line misses the origin. The solution is to do the fit again, but with the Y intercept coefficient held at a value of zero.

You might also want to display the residuals as a visual check of the fit.

Bring up the dialog again. The dialog remembers the settings you used last time, so the line fit function is already chosen in the Function menu, and your data waves are selected in the Y Data and X Data menus.

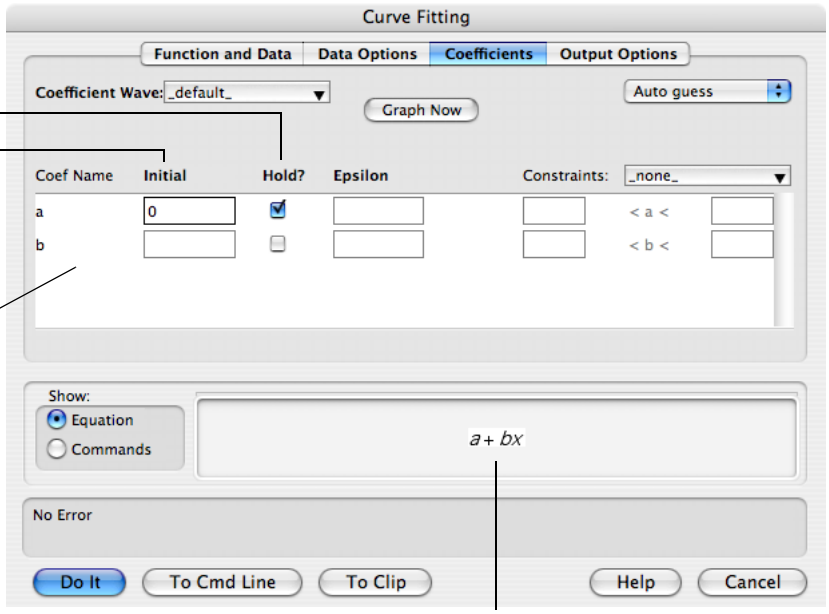
Select the Coefficient tab. Each of the coefficients has a row in the Coefficients list:

To hold a coefficient:

1. Select the checkbox under the "Hold?" label.

2. Set the coefficient value in the Initial Guess column. To make a line fit pass through the origin, set *a* to zero.

The Coefficients list has a row for each fit coefficient.

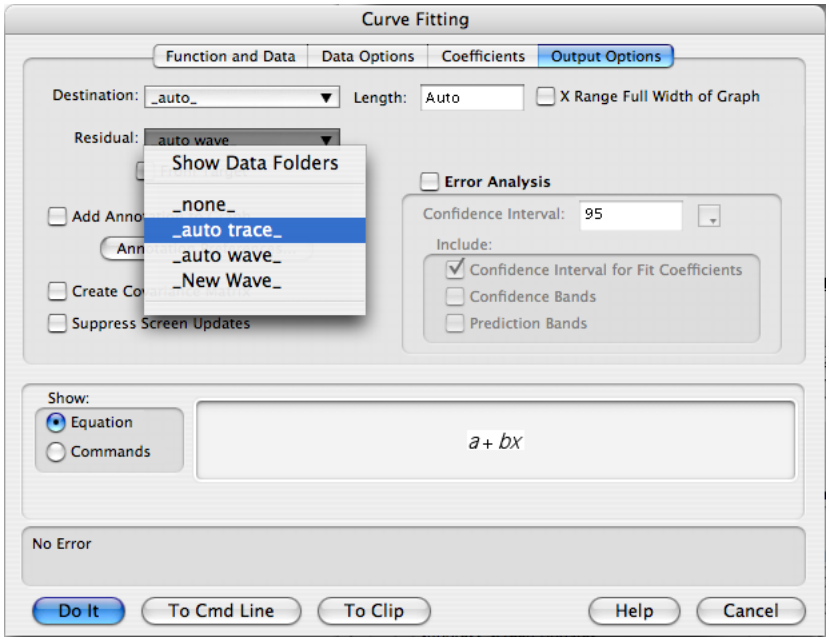


If you can't remember what the coefficients mean, check the equation display.

Click the checkbox in the column labelled "Hold?" to fix the value of that coefficient. To specify a coefficient value, fill in the corresponding box in the Initial Guess column. Until you select the Hold box the initial guess box is not available because built-in fits don't require initial guesses.

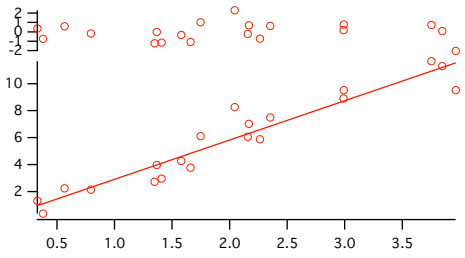
To fill in a value, click in the box. You can now type a value. When you have finished, press Enter (*Windows*) or Return (*Macintosh*) to exit editing mode for that box.

Now we want to calculate the fit residuals and add them to the graph. Click the Output Options tab and choose _auto trace_ from the Residual menu:



There are a number of options for the residual. We chose _auto trace_ to calculate the residual and add it to the graph. You may not always want the residuals added to your graph; choose _auto wave_ to automatically calculate the residuals but *not* display them on your graph. Both _auto trace_ and _auto wave_ create a wave with the same name as your Y wave with "Res_" prefixed to the name. Choosing _New Wave_ generates commands to make a new wave with your choice of name to fill with residuals. It is not added to your graph.

Now when we click Do It, the fit is recalculated with *a* held at zero so that the line passes through the origin. Residuals are calculated and added to the graph:



Note that the line on the graph doesn't cross the vertical axis at zero, because the horizontal axis doesn't extend to zero.

Holding *a* at zero, the result of the fit printed in the history is:

```
•K0 = 0;
•CurveFit/H="10" line LineYData /X=LineXData /D /R
   fit_LineYData= W_coef[0]+W_coef[1]*x
```

The /H flag shows that one or more coefficients are held.

```
Res_LineYData= LineYData[p] - (W_coef[0]+W_coef[1]*LineXData[p])
W_coef={0,2.915}
V_chisq= 18.2565; V_npnts= 20; V_numNaNs= 0; V_numINFs= 0;
V_startRow= 0; V_endRow= 19; V_q= 1; V_Rab= 0; V_Pr= 0.956769;
V_r2= 0.906186;
W_sigma={0,0.0971}
Coefficient values ± one standard deviation
   a  =  0 ± 0
   b  =  2.915 ± 0.0971
```

a is zero because it was held.

Automatic Guesses Didn't Work

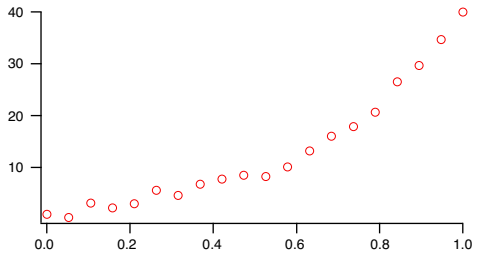
Most built-in fits will work just like the line fit. You simply choose a function from the Function menu, choose your data wave (or waves if you have both X and Y waves) and select output options on the Output Options tab. For built-in fits you don't need the Coefficients tab unless you want to hold a coefficient.

In a few cases, however, automatic guesses don't work. Then you must use the Coefficient tab to set your own initial guesses. One important case in which this is true is if you are trying to fit a growing exponential, $y = ae^{bx}$, where b is positive.

Here are commands to create an example for this section. Once again, you may wish to enter these commands on the command line to follow along:

```
make/n=20 RisingExponential
SetScale/I x 0,1,RisingExponential
RisingExponential = 2*exp(3*x)+gnoise(1)
Display RisingExponential
ModifyGraph mode=3,marker=8
```

These commands make a 20-point wave, set its X scaling to cover the range from 0 to 1, fill it with exponential values plus a bit of noise, and make a graph:

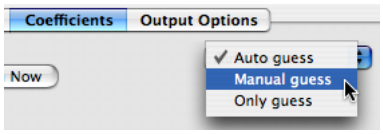


The first-cut trial to fitting an exponential function is to select exp from the Function menu and the RisingExponential wave in the Y Data menu (if you are continuing from the previous section, you may need to go to the Coefficients tab and un-hold the y0 coefficient, and to the Output Options tab and de-select _auto trace_ in the Residual menu). Automatic guesses assume that the exponential is well described by a negative coefficient in the exponential, so the fit doesn't work:

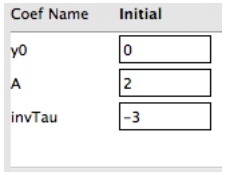
```
•CurveFit exp RisingExponential /D
  Fit converged properly
  fit_RisingExponential= W_coef[0]+W_coef[1]*exp(-W_coef[2]*x)
  W_coef={109.92,-114.59,0.3282}
  V_chisq= 432.163; V_npnts= 20; V_numNaNs= 0; V_numINFs= 0;
  W_sigma={256,254,0.86}
  Coefficient values ± one standard deviation
   y0 =  109.92 ± 2.56e+04
   A  =  -114.59 ± 2.54e+04
   K  =  0.3282 ± 86
```

Assumes a decaying exponential and doesn't fit the example data correctly.

The solution is to provide your own initial guesses. Click the Coefficients tab and choose Manual Guesses in the menu in the upper right:



The Initial Guesses column in the Coefficients list is now available for you to type your own initial guesses, including a *negative* value for invTau:



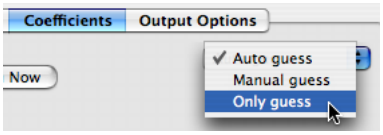
In response, Igor generates some extra commands for setting the initial guesses and this time the fit works correctly:

/G flag specifies manual guesses.

```
•K0 = 0;K1 = 2;K2 = -3;
•CurveFit/G exp RisingExponential /D
  Fit converged properly
  fit_RisingExponential= W_coef[0]+W_coef[1]*exp(-W_coef[2]*x)
  W_coef={-0.93965,2.3035,-2.8849}
  V_chisq= 14.4714; V_npnts= 20; V_numNaNs= 0; V_numINFs= 0;
  W_sigma={0.764,0.391,0.163}
  Coefficient values ± one standard deviation
   y0 =  -0.93965± 76.4
   A  =  2.3035±   39.1
   K  =  -2.8849±  16.3
```

Extra commands generated to set the manual initial guesses.

It may well be that finding a set of initial guesses from scratch is difficult. Automatic guesses might be a good starting point which will provide adequate initial guesses when modified. For this the dialog provides the Only Guess mode.



When this mode is selected, click Do It to create the automatic initial guesses, and then stop without trying to do the fit. Now, when you bring up the Curve Fitting dialog again, you can choose the coefficient wave created by the auto guess (W_coef if you chose _default_ in the Coefficient Wave menu). Choosing this wave will set the initial guesses to the automatic guess values. Now choose Manual Guesses and modify the initial guesses. The Graph Now button may help you find good initial guesses (see **Coefficients Tab for a User-Defined Function** on page III-173).

Notes on the Built-in Fit Functions

For the most part you will get good results using automatic guesses. A few require additional input beyond what is summarized in the preceding sections. This section contains notes on the fitting functions that give a bit more detail where it may be helpful.

gauss

Fits a Gaussian peak.

$$y_0 + A\exp\left[-\left(\frac{x-x_0}{width}\right)^2\right]$$

Note that the width parameter is sqrt(2) times the standard deviation of the peak. This is different from the w_i parameter in the Gauss function, which is simply the standard deviation.

lor

Fits a Lorentzian peak.

$$y_0 + \frac{A}{(x - x_0)^2 + B}$$

exp_XOffset

Fits a decaying exponential.

$$y_0 + A \exp\left(\frac{x - x_0}{\tau}\right)$$

In this equation, x_0 is a constant, not a fit coefficient. During generation of automatic guesses, x_0 will be set to the first X value in your fit data. This eliminates problems caused by floating-point roundoff.

You can set the value of x_0 using the /K flag with the CurveFit operation, but it is recommended that you accept the automatic value. Setting x_0 to a value far from the initial X value in your input data is guaranteed to cause problems.

Note: The fit coefficient τ is the inverse of the equivalent coefficient in the exp function. It is actually the decay constant, not the inverse decay constant.

Automatic guesses don't work for growing exponentials (negative τ). To fit a negative value of τ, use Manual Guess on the Coefficients tab, or CurveFit/G on the command line.

dblexp_XOffset

Fits a sum of two decaying exponentials.

$$y_0 + A_1 \exp\left(\frac{x - x_0}{\tau_1}\right) + A_2 \exp\left(\frac{x - x_0}{\tau_2}\right)$$

In this equation, x_0 is a constant, not a fit coefficient. During generation of automatic guesses, x_0 wall be set to the smallest X value in your fit data. This eliminates problems caused by floating-point roundoff.

You can set the value of x_0 using the /K flag with the CurveFit operation, but it is recommended that you accept the automatic value. Setting x_0 to a value far from the initial X value in your input data is guaranteed to cause problems.

Note: The fit coefficients τ_1 and τ_2 are the inverse of the equivalent coefficients in the dblexp function. They are actual decay constants, not inverse decay constants.

See the notes for **exp_XOffset** on page III-166 for growing exponentials. You will also need to use manual guesses if the amplitudes have opposite signs:

If the two decay constants (τ_1 and τ_2) are not quite distinct you may not get accurate results.

exp

Fits a decaying exponential. Similar to exp_XOffset, but not as robust. Included for backward compatibility; in new work you should use exp_Xoffset.

$y_0 + A\exp(-Bx)$

Note that offsetting your data in the X direction will cause changes in *A*. Use exp_XOffset for a result that is independent of X position.

Note: The fit coefficient *B* is the inverse decay constant.

Automatic guesses don't work for growing exponentials (negative *B*). To fit a negative value of *B*, use Manual Guess on the Coefficients tab, or CurveFit/G on the command line.

Floating-point arithmetic overflows will cause problems when fitting exponentials with large X offsets. This problem often arises when fitting decays in time as times are often large. The best solution is to use the exp_XOffset fit function. Otherwise, to fit such data, the X values must be offset back toward zero.

You could simply change your input X values, but it is usually best to work on a copy. Use the Duplicate command on the command line, or the Duplicate Waves item in the Data menu to copy your data.

For an XY pair, execute these commands on the command line (these commands assume that you have made a duplicate wave called `myXWave_copy`):

```
Variable xoffset = myXWave_copy[0]
myWave_copy[0] -= xoffset
```

Note that these command assume that you are fitting data from the beginning of the wave. If you are fitting subset, replace `[0]` with the point number of the first point you are fitting. If you are using graph cursors to select the points, substitute `[pcsr(A)]`. This assumes that the round cursor (cursor A) marks the beginning of the data.

If you are fitting to waveform data (you selected _calculated_ in the X Data menu) then you need to set the x0 part of the wave scaling to offset the data. If you are fitting the entire wave, simply use the Change Wave Scaling dialog from the Data menu to set the x0 part of the scaling to zero. If you are fitting a subset selected by graph cursors, it is easier to change the scaling on the command line:

```
SetScale/P x leftx(myWave_copy)-xcsr(A), deltax(myWave_copy), myWave_copy
```

This command assumes that you have used the round cursor (cursor A) to mark the beginning of the data.

Subtracting an X offset will change the amplitude coefficient in the fit. Often the only coefficient of interest is the decay constant (invTau) and the change in the amplitude can be ignored. If that is not the case, you can calculate the correct amplitude after the fit is done:

```
W_coef[1] = W_coef[1]*exp(W_coef[2]*xoffset)
```

If you are fitting waveform data, the value of xoffset would be `-leftx(myWave_copy)`.

dblexp

Fits a sum of decaying exponentials. Similar to dblexp_XOffset, but suffers from floating-point roundoff problems if the data do not start quite close to x=0. Included for backward compatibility; in new work you should use exp_Xoffset.

$y_0 + A_1\exp(-B_1x) + A_2\exp(-B_2x)$

Note that offsetting your data in the X direction will cause changes in A_1 and A_2. Use dblexp_XOffset for a result that is independent of X position.

Note: The fit coefficients B_1 and B_2 are inverse decay constants.

See the notes for exp for growing exponentials. You will also need to use manual guesses if the amplitudes have opposite signs:

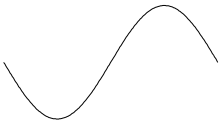
If the two decay constants (B_1 and B_2) are not quite distinct you may not get accurate results.

Fitting data with a large X offset will have the same sort of troubles as when fitting with exp. The best solution is to use the dblexp_XOffset fit function; you can also solve the problem using a procedure similar to the one outlined for **exp** on page III-166.

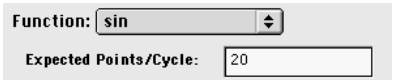
sin

Fits a sinusoid.

$$(y_0 + A\sin(fx + \phi))$$



phi is in radians. To convert to degrees, multiply by 180/PI.



A sinusoidal fit takes an additional parameter that sets the approximate frequency of the sinusoid. This is entered in terms of the approximate number of data points per cycle. When you choose `sin` from the Function menu, a box appears where you enter this number.

If you enter a number less than 6, the default value will be 7. It may be necessary to try various values to get good results. You may want to simply use manual guesses.

The nature of the sin function makes it impossible for a curve fit to distinguish phases that are different by 2π. It is probably easier to subtract 2nπ than to try to get the fit to fall in the desired range.

line

Fit a straight line through the data.

$$(a + bx)$$

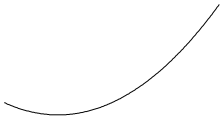
Never requires manual guesses.

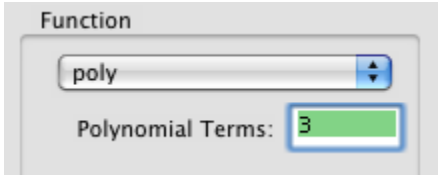
If you want to fit a line through the origin, in the Coefficients tab select the Hold box for coefficient *a* and set the Initial Guess value to zero.

poly n

Fits a polynomial with n terms, or order n-1.

$$(K_0 + K_1x + K_2x^2 + \ldots)$$





A polynomial fit takes an additional parameter that sets the number of polynomial terms. When you choose `poly` from the Function menu, a box appears where you enter this number.

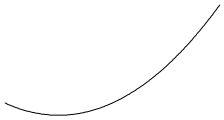
The minimum value is 3 corresponding to a quadratic polynomial.

A polynomial fit never requires manual guesses.

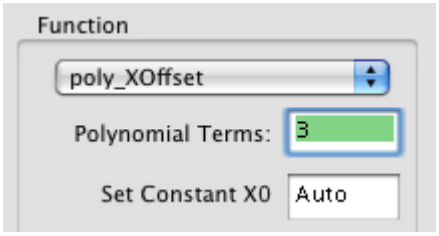
poly_XOffset n

Fits a polynomial with n terms, or order n-1. The constant x_0 is not an adjustable fit coefficient; it allows you to place the polynomial anywhere along the X axis. This would be particularly useful for situations in which the X values are large, for instance when dealing with date/time data.

$$K_1(x-x_0)+K_2(x-x_0)^2+\ldots)$$



The poly_XOffset fit function takes additional parameters that set the number of polynomial terms and the value of x_0. When you select poly_XOffset from the Function menu, boxes appear where you enter these numbers.



The minimum value for Polynomial Terms 3 corresponding to a quadratic polynomial.

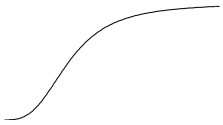
When Set Constant X0 is set to Auto, x_0 is set to the minimum X value found in the data you are fitting. You can set x_0 to any value you wish. Values far from the X values in your data set will cause numerical problems.

A polynomial fit never requires manual guesses.

HillEquation

Fits Hill's Equation, a sigmoidal function.

$$base+\frac{(max-base)}{1+\left(\frac{x_{1/2}}{x}\right)^{rate}}$$



The coefficient *base* sets the y value at small X, *max* sets the y value at large X, *rate* sets the rise rate and $x_{1/2}$ sets the X value at which Y is at (*base* + *max*)/2.

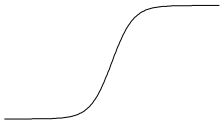
Note that X values must be greater than 0. Including a data point at $X \le 0$ will result in a singular matrix error and the message, "The fitting function returned NaN for at least one X value."

You can reverse the values of *base* and *max* to fit a falling sigmoid.

sigmoid

Fits a sigmoidal function with a different shape than Hill's equation.

$$base+\frac{max}{1+\exp\left(\frac{x_0-x}{rate}\right)}$$



The coefficient *base* sets the y value at small X, *base+max* sets the Y value at large X, x_0 sets the X value at which Y is at (*base* + *max*)/2 and *rate* sets the rise rate. Smaller *rate* causes a faster rise.

power

Fits a power law.

$(y_0 + Ax^{pow})$

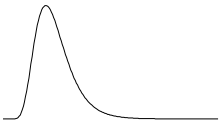
May be difficult to fit, requiring good initial guesses, especially for *pow* > 1 or *pow* close to zero.

Note that X values must be greater than 0. Including a data point at $X \le 0$ will result in a singular matrix error and the message, "The fitting function returned NaN for at least one X value."

lognormal

Fits a lognormal peak shape. This function is gaussian when plotted on a log X axis.

$$y_0 + A\exp\left[-\left(\frac{\ln(x/x_0)}{width}\right)^2\right]$$



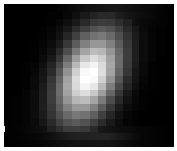
Coefficient y_0 sets the baseline, *A* sets the amplitude, x_0 sets the peak position in X and *width* sets the peak width.

Note that X values must be greater than 0. Including a data point at $X \le 0$ will cause a singular matrix error and the message, "The fitting function returned NaN for at least one X value."

gauss2D

Fits a Gaussian peak in two dimensions.

$$z_0 + A\exp\left[\frac{-1}{2(1-cor^2)}\left(\left(\frac{x-x_0}{xwidth}\right)^2 + \left(\frac{y-y_0}{ywidth}\right)^2 - \frac{2cor(x-x_0)(y-y_0)}{xwidth \cdot ywidth}\right)\right]$$



Coefficient *cor* is the cross-correlation term; it must be between -1 and 1 (the small illustration was done with *cor* equal to 0.5). A constraint automatically enforces this range. If you know that a value of zero for this term is appropriate, you can hold this coefficient. Holding *cor* at zero usually speeds up the fit quite a bit.

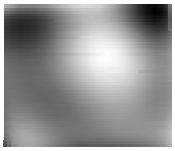
In contrast with the gauss fit function, xWidth and yWidth are standard deviations of the peak.

Note that the Gauss function lacks the cross-correlation parameter *cor*.

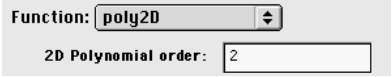
poly2D n

Fits a polynomial of order n in two dimensions.

$$(C_0 + C_1x + C_2y + C_3x^2 + C_4xy + C_5y^2 + \ldots)$$



A poly2D fit takes an additional parameter that specifies the order of the polynomial. When you choose poly2D from the Function menu, a box appears where you enter this number:



The minimum value is 1, corresponding to a first-order polynomial, a plane. The coefficient wave for poly2D has the constant term (C_0) in point zero, and following points contain groups of increasing order. There are two first-order terms, C_1*x and C_2*y, then three second-order terms, etc. The total number of terms is (N+1)(N+2)/2, where N is the order.

Poly2d never requires manual guesses.

Fitting to a User-Defined Function

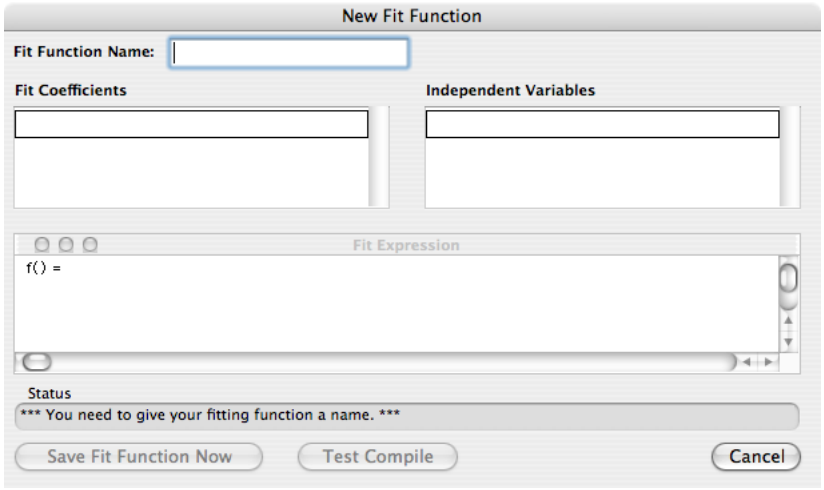
Fitting to a user-defined function is much like fitting to a built-in function, with two main differences:

- You must define the fitting function.
- You must supply initial guesses.

To illustrate the creation of a user-defined fit function, we will create a function to fit a log function: $y = C_1 + C_2 \ln(x)$.

Creating the Function

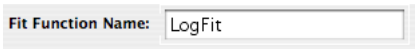
To create a user-defined fitting function, click the New Fit Function button in the Function and Data tab of the Curve Fitting Dialog. The New Fit Function dialog is displayed:



You must fill in a name for your function, fill in the Fit Coefficients list with names for the coefficients, fill in the Independent Variables list with names for independent variables, and then enter a fit expression in the Fit Expression window.

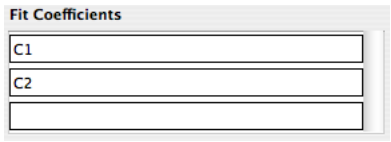
The function name must conform to nonliberal naming rules. That means it must start with a letter, contain only letters, numbers or underscore characters, and it must be 31 or fewer characters in length (see **Object Names** on page III-415). It must not be the same as the name of another object like a built-in function, user procedure, or wave name.

For the example log function, we enter "LogFit":

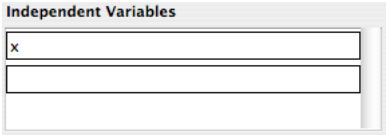


Press Tab to move to the first entry in the Fit Coefficient list. There is always one blank entry in the list where you can add a new coefficient name; since we haven't entered anything yet, there is only one blank entry.

Each time you enter a name, press Return (*Macintosh*) or Enter (*Windows*). Igor accepts that name and makes a new blank entry where you can enter the next name. We enter C1 and C2 as the names:



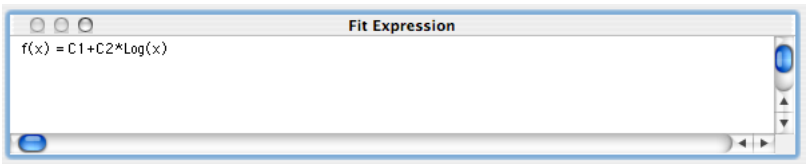
Press Tab to move to the blank entry in the Independent Variables list. Most fit functions will require just a single independent variable. We choose to name our independent variable x:



It is now time to enter the fit expression. You will notice that when you have entered a name in the Independent Variables list, some text is entered in the expression window. The return value of the fit function (the Y value in most cases) is marked with something like "f(x) = ". If you had entered "temperature" as the independent variable, it would say "f(temperature) = ".

This "f() = " text is required; otherwise the return value of the function will be unknown.

The fit expression is not an algebraic expression. It must be entered in the same form as a command on the command line. If you need help constructing a legal expression, you may wish to read **Assignment Statements** on page IV-4. The expression you need to type is simply the right-hand side of an assignment statement. The log expression in our example will look like this:



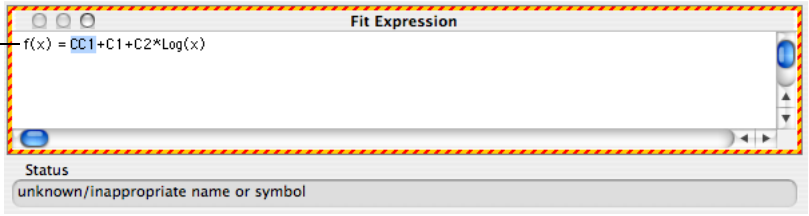
Multiplication requires an explicit *.

The dialog will check the fit expression for simple errors. For instance, it will not make the Save Fit Function Now button available if any of the coefficients or independent variables are missing from the expression.

The dialog cannot check for correct expression syntax. If all the easily-checked items are correct, the Save Fit Function Now and Test Compile buttons are made available. Clicking either of them will enter a new function in the Procedure window and attempt to compile procedures. If you click the Save Fit Function Now button and compilation is successful, you are returned to the Curve Fitting dialog with the new function chosen in the Function menu.

If compile errors occur, the compiler's error message is displayed in the status box, and the offending part of your expression is highlighted. A common error might be to misspell a coefficient name somewhere in your expression. For instance, if you had typed CC1 instead of C1 somewhere you might see something like this:

Note that C1 appears in the expression. Otherwise, the dialog would show that C1 is missing.



When everything is ready to go, click the Save Fit Function Now button to construct a function in the Procedure window. It includes comments in the function code that identify various kinds of information for the dialog. Our example function looks like this:

```
Function LogFit(w,x) : FitFunc
    WAVE w
    Variable x

    //CurveFitDialog/ These comments were created by the Curve Fitting dialog. Altering them will
    //CurveFitDialog/ make the function less convenient to work with in the Curve Fitting dialog.
    //CurveFitDialog/ Equation:
    //CurveFitDialog/ f(x) = C1+C2*log(x)
    //CurveFitDialog/ End of Equation
    //CurveFitDialog/ Independent Variables 1
    //CurveFitDialog/ x
    //CurveFitDialog/ Coefficients 2
```

```
    //CurveFitDialog/ w[0] = C1
    //CurveFitDialog/ w[1] = C2

    return w[0]+w[1]*log(x)
End
```

You shouldn't have to deal with the code in the Procedure window unless your function is so complex that the dialog simply can't handle it. You can look at **User-Defined Fitting Function: Detailed Description** on page III-217 for details on how to write a fitting function in the Procedure window.

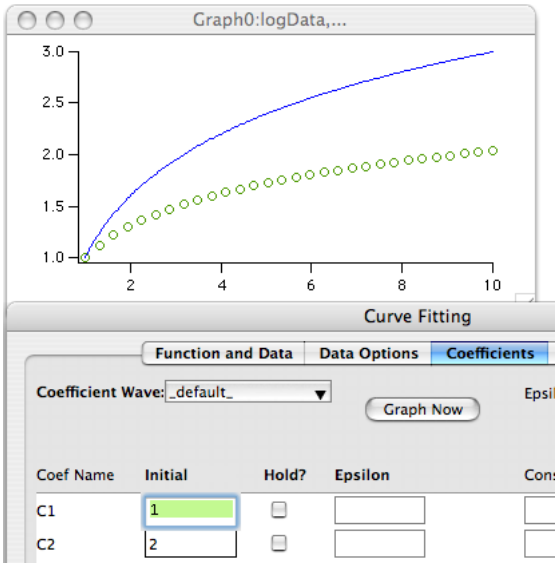
Having entered the fit expression correctly, click the Save Fit Function Now button, which returns you to the main Curve Fitting dialog. The Function menu will now have LogFit chosen as the fitting function.

Coefficients Tab for a User-Defined Function

To fit a user-defined function, you will need to enter initial guesses in the Coefficients tab.

Having created a user-defined fitting function (or simply having selected a preexisting one) you will find that the error message window at the bottom of the dialog now states: "You have selected a user-defined fit function so you must enter an initial guess for every fit coefficient. Go to the Coefficients Tab to do this."

When you have selected a user-defined fitting function, the Initial Guess column of the Coefficients List is available. You must enter a number in each row. Some functions may be difficult to fit; in such a case the initial guess may have to be pretty close to the final solution.



To help you find good initial guesses, the Coefficients tab includes the Graph Now button. This button will add a trace to the top graph showing your fitting function using the initial guesses you have entered.

You can change the values in the initial guess column and click the Graph Now button as many times as you wish. The trace will be updated with the changes each time.

The Graph Now button makes a wave using the name of your Y data wave prefixed with "fit_". The auto trace destination wave also uses a wave with the same name when you execute the fit.

On the Coefficients tab you have the option of selecting an "epsilon" wave. An epsilon wave contains one epsilon value for each point in your coefficients wave. By default the Epsilon menu is set to _none_ indicating that the epsilon values are set to the default values.

Each epsilon value is used to calculate partial derivatives with respect to the fit coefficients. The partial derivatives are used to determine the search direction for coefficients that give the smallest chi-square.

In most cases the epsilon values are not critical. However, you can supply your own if you have reason to believe that the default epsilon values are not providing acceptable partial derivatives (sometimes a singular matrix error can be avoided by using custom epsilon values). To specify epsilon values, choose the wave from the Epsilon menu. The values in the selected wave are entered in the Epsilon column in the Coefficients list. You can change the values by clicking and typing in the box you want to change. To specify epsilon values, type them into the boxes in the Epsilon column in the Coefficients list. If you select a wave from the Epsilon menu, the values in that wave will be entered in the list. If you choose _New Wave_, the dialog will generate commands to create an epsilon wave and fill it with the values in the Epsilon column.

Making a User-Defined Function Always Available

Note that, because the fitting function is created in the Procedure window, it is stored as part of the experiment file. That means that it will be available for fitting while you are working on the experiment in which it was created, but will not be available when you work on other experiment files.

You can make the fit function available whenever you start up Igor Pro. Make a new procedure window using the Procedure item under New in the Windows menu. Find the fit function in the Procedure window, select all the text from `Function` through `end` and choose Cut from the Edit menu. Paste the code into your new procedure window. Finally, choose Save Procedure Window from the File menu and save in"Igor Pro User Files/Igor Procedures" (see **Igor Pro User Files** on page II-46 for details). The next time you start Igor Pro you will find that the function is available in all your experiments.

Removing a User-Defined Fitting Function

To remove a user-defined fitting function that you don't want any more, choose Procedure Window from the Windows menu. Find the function in the Procedure window (you can use Find from the Edit menu and search for the name of your function). Select all of the function definition text from the word "`Function`" through the word "`End`" and delete the text.

If you have followed the directions in the section above for making the function always available, find the procedure file in "Igor Pro User Files/Igor Procedures", remove it from the folder, and then restart.

User-Defined Fitting Function Details

The New Fit Function dialog is the easiest way to enter a user-defined fit function. But if your fit expression is very long, or it requires multiple lines with local variables or conditionals, the dialog can be cumbersome. Certain special situations may call for a format that is not supported by the dialog.

For a complete discussion of user-defined fit function formats and the uses for different formats, see **User-Defined Fitting Function: Detailed Description** on page III-217.

Fitting to an External Function (XFUNC)

An external function, or XFUNC, is a function provided via an Igor extension or plug-in. A programmer uses the XOP Toolkit to build an XFUNC. You don't need the toolkit to use one. An XFUNC must be installed before it can be used. See **Igor Extensions** on page III-423.

An XFUNC can speed up curve fitting greatly if your fitting function requires a great deal of computation. The speed of fitting is usually dominated by other kinds of overhead and the effort of writing an XFUNC is not justified.

Fitting to an external function is just like fitting to a user-defined function, except that the Curve Fitting dialog has no way to find out how many fit coefficients are required. When you switch to the Coefficients tab, you will see an alert telling you of that fact. The solution to this problem is to select a coefficient wave with the correct number of points. You must create the wave before entering the Curve Fitting dialog.

When you select a coefficient wave the contents of the wave are used to build the Coefficients list. The wave values are entered in the Initial Guess column. If you change an initial guess, the dialog will generate the commands necessary to enter the new values in the wave.

The Coefficient Wave menu normally shows only those waves whose length is the same as the number of fit coefficients required by the fitting function. When you choose an XFUNC for fitting, the menu shows all waves. You have to know which one to select. We suggest using a wave name that identifies what the wave is for.

Igor doesn't know about coefficient names for an XFUNC. Coefficient names will be derived from the name of the coefficient wave you select. That is, if your coefficient wave is called "coefs", the coefficient names will be "coefs_0", "coefs_1", etc.

Of course, implementing your function in C or C++ is more time-consuming and requires both the XOP Toolkit from WaveMetrics, and a software development environment. See **Creating Igor Extensions** on page IV-181 for details on using the XOP Toolkit to create your own external function.

The Coefficient Wave

When you fit to a user-defined function, your initial guesses are transmitted to the curve fitting operation via a coefficient wave. The coefficients that result from the fit are output in a coefficient wave no matter what kind of function you select. For the most part, the Curve Fitting dialog hides this from you.

When you create a user-defined function, the dialog creates a function that takes a wave as the input containing the fit coefficients. But through special comments in the function code, the dialog gives names to each of the coefficients. A built-in function has names for the coefficients stored internally. Using these names, the dialog is able to hide from you some of the complexities of using a coefficient wave.

In the history printout following a curve fit, the coefficient values are reported both in the form of a wave assignment, using the actual coefficients wave, and as a list using the coefficient names. For instance, here is the printout from the example user-defined fit earlier (**Fitting to a User-Defined Function** on page III-171):

```
•FuncFit LogFit W_coef logData /D
  Fit converged properly
  fit_logData= LogFit(W_coef,x)
  W_coef={1.0041,0.99922}
  V_chisq= 0.00282525; V_npnts= 30; V_numNaNs= 0; V_numINFs= 0;
  W_sigma={0.00491,0.00679}
  Coefficient values ± one standard deviation
   C1 =  1.0041± 0.491
   C2 =  0.99922± 0.679
```

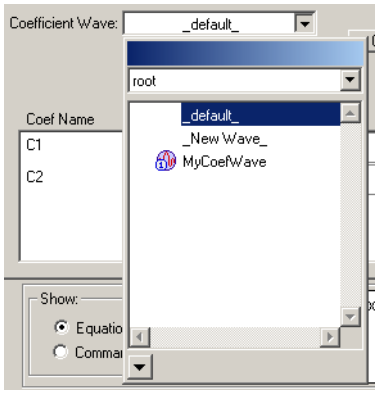
Fit Coefficient values as a wave assignment.

Fit Coefficient sigmas as a wave assignment.

Fit Coefficient values and sigmas in a list using the coefficient names.

The wave assignment version can be copied to the command line and executed, or it can be used as a command in a user procedure. The list version is easier to read.

You control how to handle the coefficients wave using the Coefficient Wave menu on the Coefficients tab (this shows a Wave Browser in Show Data Folders mode):



Default

When _default_ is chosen it creates a wave called W_coef. For built-in fits this wave is used only for output. For user-defined fits it is also input. The dialog generates commands to put your initial guesses into the wave before the fit starts.

Explicit Wave

The Coefficient Wave menu lists any wave whose length matches the number of fit coefficients. If you select one of these waves, it is used for input and output from any fit. In the illustration above, myCoefWave is an explicit coefficient wave.

When you choose a wave from the menu the data in the wave is used to fill in the Initial Guess column in the Coefficients list. This can be used as a convenient way to enter initial guesses. If you choose an explicit wave and then edit the initial guesses, the dialog generates commands to change the values in the selected coefficient wave before the fit starts. To avoid altering the contents of the wave, you can choose _default_ or _New Wave_. The initial guesses will be put into the new or default coefficients wave.

New Wave

A variation on the explicit coefficients wave is _New Wave_. This works just like an explicit wave except that the dialog generates commands to make the wave before the fit starts, so you don't have to remember to make it before entering the dialog. The wave is filled in with the values from the Initial Guess column.

The _New Wave_ option is convenient if you are doing a number of fits and you want to save the fit coefficients from each fit. If you use _default_ the results of a fit will overwrite the results from any previous fit.

Errors

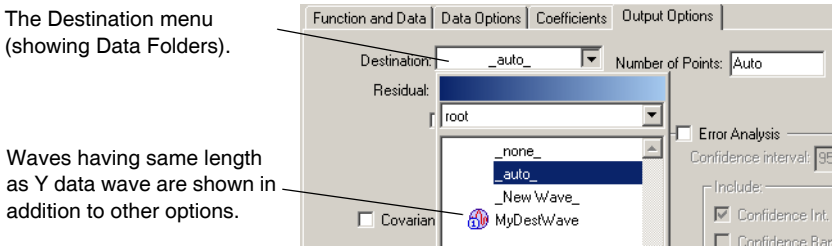
Estimates of fitting errors (the estimated standard deviation of the fit coefficients) are automatically put into a wave called W_sigma. There is no user choice for this output.

The Destination Wave

When performing a curve fit, it will calculate the model curve corresponding to the fit coefficients. As with most results, the model curve is stored as an array of numbers in a wave. This wave is the "destination wave".

The main purpose of the destination wave is to show what the fit function looks like with various coefficients during the fit operation and with the final fit coefficients when the operation is finished. You can choose no destination wave, an explicit destination wave or you can use the auto-trace feature.

You choose the destination wave option on the Output Options tab of the dialog:



No Destination

You would choose no destination wave if you don't want graphic feedback during the fitting process and you don't need to graphically compare your raw data to the fitting function. This might be the case if you are batch fitting a large number of data sets. Choose _none_ in the Destination menu.

Auto-Trace

In most cases, auto-trace is recommended; choose _auto_ from the Destination menu. When you choose this, it automatically creates a new wave with 200 points, sets its X scaling appropriately, and appends it to the top graph if the Y data wave is displayed in it. The new wave is generated by prepending "fit_" to the name of the Y data wave. If a wave of that name already exists, it is overwritten. If the name exceeds the 31 character maximum for a wave, the name is truncated.

If you want to fit more than one function to the same raw data using auto-trace, you should rename the auto-trace wave after the fit so that it will not be overwritten by a subsequent fit. You can rename it using the Rename item in the Data menu or by executing a Rename command directly from the command line. You may also want to save the W_coef and W_sigma waves using the same technique.

Usually the auto-trace wave is set up as a waveform even if you are fitting to XY data. The X scaling of the auto-trace wave is set to spread the 200 points evenly over the X range of the raw data. When preferences are on, the auto-trace wave appended to the graph has the preferred line style (color, size, etc.) *except* that the line mode is always set to "lines between points", which is best suited to showing the curve fitting results.

Evenly-spaced data are not well suited to displaying a curve on a logarithmic axis. If your data are displayed using a log axis, the fit will create an XY pair of waves. The X wave will be named by prepending "fitX_" to the name of the Y data wave. This X wave is filled with exponentially-spaced X values spread out over the X range of the fit data. Of course, if you subsequently change the axis to a linear axis, the point spacing will not look right.

With _auto_ chosen in the Destination menu, the dialog displays a box labelled Length:. Use this to change the number of points in the destination wave. You can set this to any number greater than 3. The more points, the smoother the curve (up to a point). More points will also take longer to draw so the fit will be slower.

Explicit Destination

You can specify an explicit destination wave rather than using auto-trace. Use this to easily calculate function values corresponding to your data values and aids in calculating the residuals from the fit. An explicit destination wave must have the same number of points as the Y data wave, so you should create it using the Duplicate operation. The Destination menu shows only waves that have the same number of points as the selected Y Data wave.

The explicit destination wave is not automatically appended to the top graph. Therefore, before executing the curve fit operation, you would normally execute commands like:

```
Duplicate/O yData, yDataFit
AppendToGraph yDataFit vs xData
```

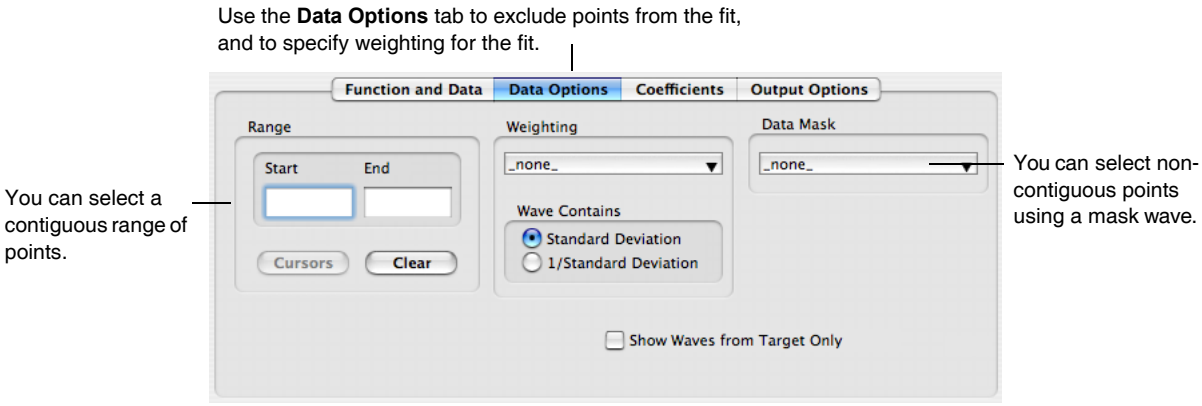
If you are fitting waveform data rather than XY data, you would omit "vs xData" from the AppendToGraph command.

New Wave

As a convenience, the Curve Fitting dialog can create a destination wave for you if you choose _New Wave_ from the Destination menu. It does this by generating a Duplicate command to duplicate your Y data wave and then uses it just like any other explicit destination wave. The new wave is not automatically appended to your graph, so you will have to do that yourself after the fit is completed.

Fitting a Subset of the Data

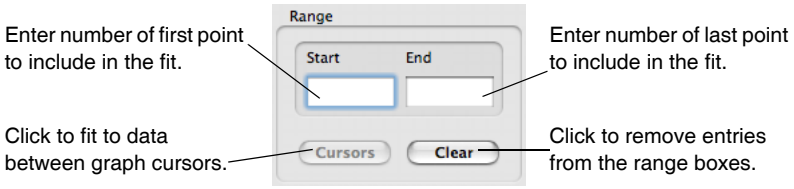
A common problem is that you don't want to include all of the points in your data set in a curve fit. There are two methods for restricting a fit to a subset of points. You will find these options on the Data Options tab of the Curve Fitting dialog:



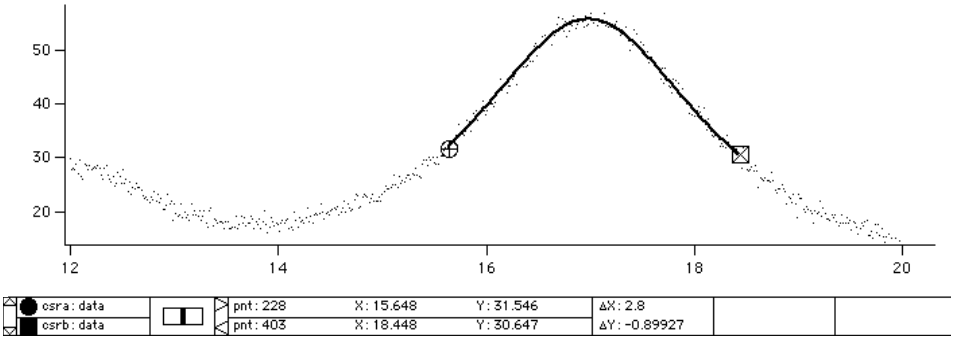
Selecting a Range to Fit

You can select a contiguous range of points in the Range box. The values that you use to specify the range for a curve fit are in terms of point or row numbers of the Y data wave. Note that if you are fitting to an XY pair of waves and your X values are in random order, you will not be selecting a contiguous range as it appears on a graph.

To simplify selecting the range, you can use graph cursors to select the start and end of the range. To use cursors, display your raw data in a graph. Display the info panel in the graph by selecting ShowInfo from the Graph menu. Drag the cursors onto your raw data. Then use the Cursors button in the Curve Fitting dialog to generate a command to fit the cursor range.



Here is what the graph will look like after the fit.

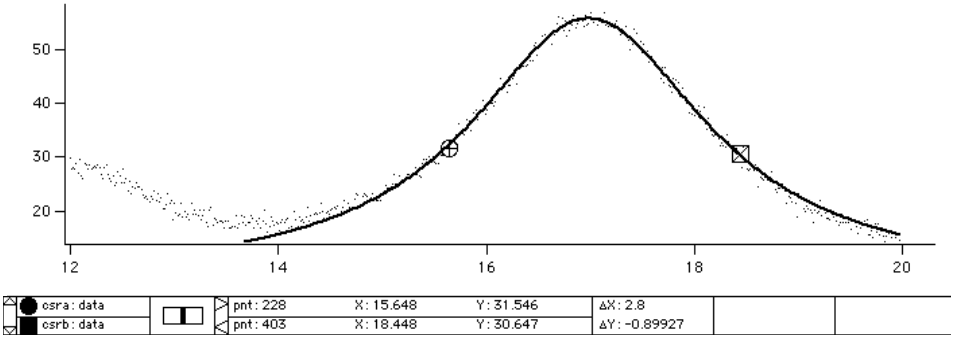


In this example, we used auto-trace for the destination. Notice that the trace appears over the selected range only. If we want to show the destination over a wider range, we need to change the destination wave's X scaling. These commands change the destination wave to show more points over a wider range:

```
Redimension/N=500 fit_data              // change to 500 points
SetScale x 13, 20, fit_data             // set domain from 13 to 20
fit_data= W_coef[0]+W_coef[1]/((x-W_coef[2])^2+W_coef[3])
```

The last line was copied from the history area, where it was displayed after the fit.

This produces the following graph.



If you use an explicit destination wave rather than auto-trace, it is helpful to set the destination wave to blanks (NaN) before performing the fit. As the fit progresses, it will store new values only in the destination wave range that corresponds to the range being fit. Also, it stores into the destination wave only at points where the source wave is not NaN or INF. If you don't preset the destination wave to blanks, you will wind up with a combination of new and old data in the destination wave.

These commands illustrate presetting the destination wave and then performing a curve fit to a range of an XY pair.

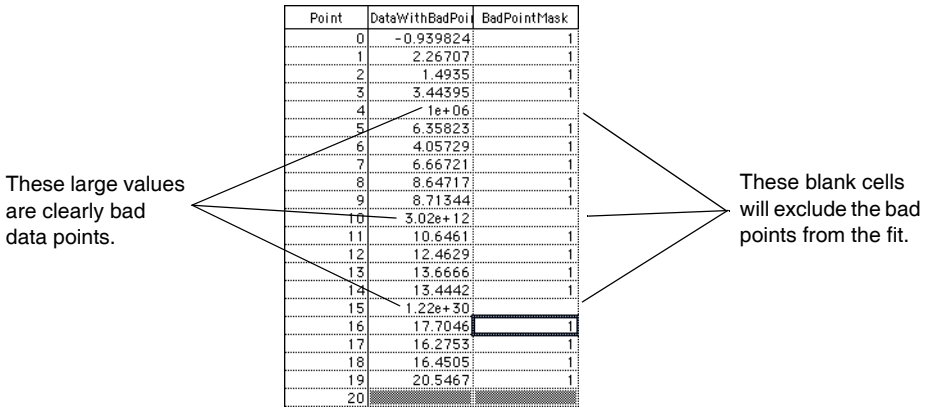
```
Duplicate/O yData, yDataFit          // make destination
yDataFit = NaN                       // preset to blank (NaN)
AppendToGraph yDataFit vs xData
CurveFit lor yData(xcsr(A),xcsr(B)) /D=yDataFit
```

Another way to make the fit curve cover a wider range is to select the checkbox labelled X Range Full Width of Graph. You will find the checkbox on the Output Options tab of the Curve Fitting dialog.

Using a Mask Wave

Sometimes the points you want to exclude are not contiguous. This might be the case if you are fitting to a data set with occasional bad points. Or, in spectroscopic data you may want to fit regions of baseline and exclude points that are part of a spectroscopic peak. You can achieve the desired result using a mask wave.

The mask wave must have the same number of points as the Y Data wave. You fill in the mask wave with a NaN (Not-a-Number, blank cell) or zero corresponding to data points to exclude and nonzero for data points you want to include. You must create the mask wave before bringing up the Curve Fitting dialog. Here is a table displaying a dataset with some bad data points and a wave to be used as a mask wave:



Point	DataWithBadPoi	BadPointMask
0	-0.939824	1
1	2.26707	1
2	1.4935	1
3	3.44395	1
4	1e+06	
5	6.35823	1
6	4.05729	1
7	6.66721	1
8	8.64717	1
9	8.71344	1
10	3.02e+12	
11	10.6461	1
12	12.4629	1
13	13.6666	1
14	13.4442	1
15	1.22e+30	
16	17.7046	1
17	16.2753	1
18	16.4505	1
19	20.5467	1
20		

Enter a NaN in a table by typing "NaN" and pressing Return or Enter. You can also use a wave assignment on the command line. In the case shown here, a suitable command to set point four would be

```
BadPointMask[4] = NaN
```

You can use a mask with NaN points to suppress display of the masked points in a graph if you select the mask wave as an f(z) wave in the Modify Trace Appearance dialog.

When you have a suitable mask wave, you choose it from the Data Mask menu on the Data Options tab.

Weighting

You may provide a weighting wave if you want to assign greater or lesser importance to certain data points. You would do so for one of two reasons:

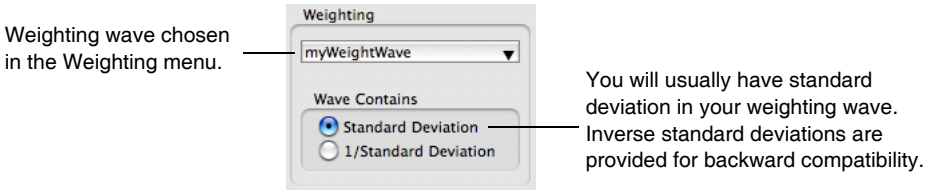
- To get a better, more accurate fit.
- To get more accurate error estimates for the fit coefficients.

The weighting wave is used in the calculation of chi-square. chi-square is defined as

$$\sum_i \left(\frac{y - y_i}{w_i}\right)^2$$

where y is a fitted value for a given point, y_i is the original data value for the point and w_i is the standard error for the point. The weighting wave provides the w_i values. The values in the weighting wave can be either $1/\sigma_i$ or simply σ_i, where σ_i is the standard deviation for each data value. If necessary, the weighting value is squared and the inverse taken before it is used to perform the weighting.

You specify the wave containing your weighting values by choosing it from the Weighting menu in the Data Options tab. In addition you must specify whether your wave has standard deviations or inverse standard deviations in it. You do this by selecting one of the buttons below the menu:



Usually you would use standard deviations. Inverse standard deviations are permitted for historical reasons.

There are several ways in which you might obtain values for σ_i. For example, you might have *a priori* knowledge about the measurement process. If your data points are average values derived from repeated measurements, then the appropriate weight value is the standard error. That is the standard deviation of the repeated measurements divided by $N^{1/2}$. This assumes that your measurement errors are normally distributed with zero mean.

If your data are the result of counting, such as a histogram or a multichannel detector, the appropriate weighting is $(\sqrt{y})$. This formula, however, makes infinite weighting for zero values, which isn't correct and will eliminate those points from the fit. It is common to substitute a value of 1 for the weights for zero points.

You can use a value of zero to completely exclude a given point from the fit process, but it is better to use a data mask wave for this purpose.

If you do not provide a weighting wave, then unity weights are used in the fit and the covariance matrix is normalized based on the assumption that the fit function is a good description of the data. The reported errors for the coefficients are calculated as the square root of the diagonal elements of the covariance matrix and therefore the normalization process will provide valid error estimates only if all the data points have roughly equal errors and if the fit function is, in fact, appropriate to the data.

If you do provide a weighting wave then the covariance matrix is not normalized and the accuracy of the reported coefficient errors rests on the accuracy of your weighting values. For this reason you should not use arbitrary values for the weights.

In some cases, it is desirable to use weighting but you know only proportional weights, not absolute measurement errors. In this case, you can use weighting and after the fit is done, calculate reduced chi-square. The reduced chi-square can be used to adjust the reported error estimates for the fit coefficients.

See **Guided Tour 3 - Histograms and Curve Fitting** on page I-55 for an example of curve fitting using weighting.

Fitting to a Multivariate Function

A multivariate function is a function having more than one independent variable. This might arise if you have measured data over some two-dimensional area. You might measure surface temperature at a variety of locations, resulting in temperature as a function of both X and Y. It might also arise if you are trying to find dependencies of a process output on the various inputs, perhaps initial concentrations of reagents, plus temperature and pressure. Such a case might have a large number of independent variables.

Fitting a multivariate function is pretty much like fitting to a function with a single independent variable. This discussion assumes that you have already read the instructions above for fitting a univariate function.

You can create a new multivariate user-defined function by clicking the New Fit Function button. In the Independent Variables list, you would enter more than one variable name. You can use as many independent variables as you wish (within generous limits set by the length of a line in the procedure window).

A univariate function usually is written as $y = f(x)$, and the Curve Fitting dialog reflects this in using "Y Data" and "X Data" to label the menus where you select the input data.

Multivariate data isn't so convenient. Functions intended to fit spatial data are often written as $z = f(x,y)$; volumetric data may be $g = f(x,y,z)$. Functions of a large number of independent variables are often written

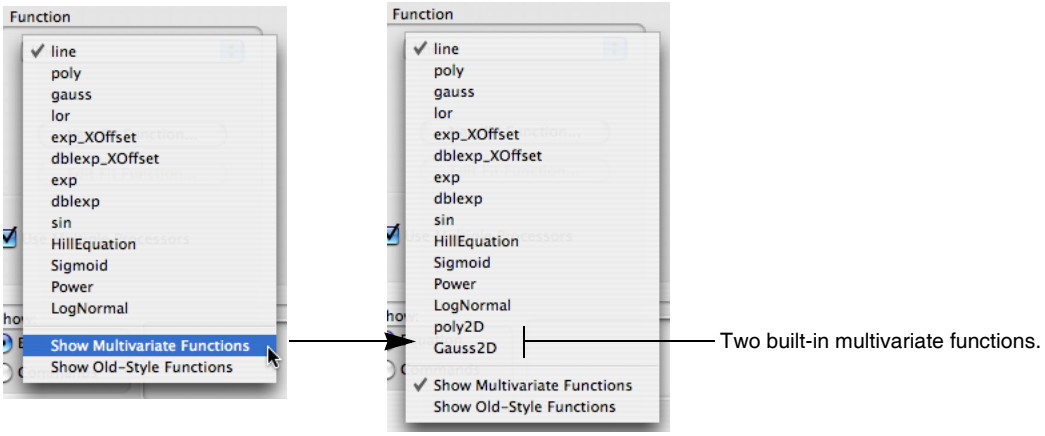
as $y = f(x_1,x_2,\ldots)$. To avoid confusion, we just keep the Y Data and X Data labels and use them to mean dependent variable and independent variables.

The principle difference between univariate and multivariate functions is in the selection of input data. If you have four or fewer independent variables, you can use a multidimensional wave to hold the Y values. This would be appropriate for data measured on a spatial grid, or any other data measured at regularly-spaced intervals in each of the independent variables. We refer to data in a multidimensional wave as "gridded data."

Alternately, you can use a 1D wave to hold the Y values. The independent variables can then be in N 1D waves, one wave for each independent variable, or a single N-column matrix wave. The X wave or waves must have the same number of rows as the Y wave.

Selecting a Multivariate Function

When the Curve Fitting dialog is first used, multivariate functions are not listed in the Function menu. The first thing you must do is to turn on the listing of multivariate functions. You do this by choosing Show Multivariate Functions from the Function menu:



The Show Multivariate Functions setting is saved in the preferences. Unless you turn it off again, you never need select it again.

Now you can select your multivariate function from the menu.

Selecting Fit Data for a Multivariate Function

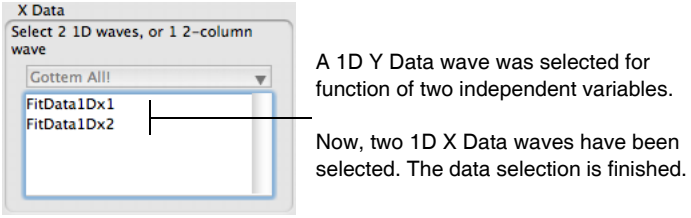
When you have selected a multivariate function, the Y Data menu is filled with 1D waves and any multidimensional waves that match the number of independent variables required by the fit function.

Selecting X Data for a 1D Y Data Wave

If your Y data are in a 1D wave, you must select an X wave for each independent variable. There is no way to store X scaling data in the Y wave for more than one independent variable, so there is no _calculated_ item.

With a 1D wave selected in the Y Data menu, the X Data menu lists both 1D waves and 2D waves with N columns for a function with N independent variables.

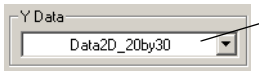
As you select X waves, the wave names are transferred to a list below the menu. When you have selected the right number of waves the X Data menu is disabled. The order in which you select the X waves is important. The first selected wave gives values for the first independent variable, etc.



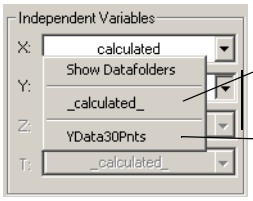
If you need to remove an X Data wave from the list, simply click the wave name and press Backspace (*Windows*) or Delete (*Macintosh*). To change the order of X Data waves, select one or more waves in the list and drag them into the proper order.

Selecting X Data for Gridded Y Data

When you select a multidimensional Y wave, the independent variable values can come from the dimension scaling of the Y wave or from 1D waves containing values for the associated dimensions of the Y wave. That is, if you have a 2D matrix Y wave, you could select a wave to give values for the X dimension and a wave to give values for the Y dimension. The Independent Variable menus list only waves that match the given dimension of the Y wave.



A matrix wave with 20 rows and 30 columns is chosen in the Y Data menu.



Choose _calculated_ if the Ydata for this dimension is to come from the wave's dimension scaling. (This menu has data folders hidden.)

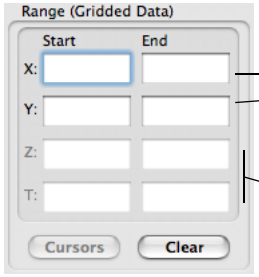
Only 1D waves matching the Y wave's dimension size are listed in these menus.

The Z and T menus are not used— the function has two independent variables and the Y wave has two dimensions.

Fitting a Subrange of the Data for a Multivariate Function

Selecting a subrange of data for a 1D Y wave is just like selecting a subrange for a univariate function. Simply enter point numbers in the Start and End range boxes in the Data Options tab.

If you are fitting gridded Y data, the Data Options tab displays eight boxes to set start and end ranges for each dimension of a multidimensional wave. Enter row, column, layer or chunk numbers in these boxes:



Set starting and ending.

row and column numbers in these boxes.

These boxes are unavailable because the selected fit function has only two independent variables, and a 2D Y wave is selected.

If your Y wave is a matrix wave displayed in a graph as an image, you can use the cursors to select a subset of the data. With the graph as the target window, clicking the Cursors button will enter text in the range boxes to do this.

Using cursors with a contour plot is not straightforward, and the dialog does not support it.

You can also select data subranges using a data mask wave (see **Using a Mask Wave** on page III-179). The data mask wave must have the same number of points and dimensions as the Y Data wave.

Model Results for Multivariate Fitting

As with fitting to a function of one independent variable, it will create waves containing the model output and residuals automatically. This is done if you choose _auto_ for the destination and _auto trace_ for the residual on the Output Options tab. There are some differences in detail, however.

For a univariate fit, it by default makes a smooth curve having 200 points to display the model fit. This action depends on being able to sensibly interpolate between successive values of the independent variable. Multicolumn independent variables, on the other hand, probably don't have successive values of all the independent variables in sequential order, so it is not possible to do this. Consequently, it calculates a model point for each point in the dependent variable data wave. If the data wave is displayed as a simple 1D trace in the top graph window, the fit results will be appended to the graph.

Residuals are always calculated on a point-for-point basis, so calculating residuals for a multicolumn multivariate fit is just like a univariate fit.

Displaying results of fitting to a multidimensional wave is more problematic. If the dependent variable has three or more dimensions, it is not easy to display the results. The model and residual waves will be created and the results calculated but not displayed. You can make a variety of 3D plots using Gizmo: just choose the appropriate plot type from the Windows→New→3D Plots menu.

Fits to a 2D matrix wave are displayed on the top graph if the Y Data wave is displayed there as either an image or a contour. The model results are plotted as a contour regardless of whether the data are displayed as an image or a contour. Model results contoured on top of data displayed as an image can be a very powerful visualization technique.

Residuals are displayed in the same manner as the data in a separate, automatically-created graph window. The window size will be the same as the window displaying the data.

Time Required to Update the Display

Because contours and images can take quite a while to redraw, the time to update the display at every iteration may cause fits to contour or image data to be very slow. To suppress the updates, click the Suppress Screen Updates checkbox on the Output Options tab.

Multivariate Fitting Examples

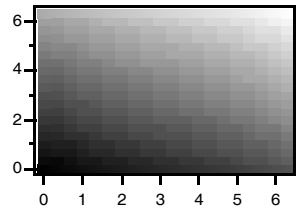
Here are two examples of fitting to a multivariate function — the first uses the built-in poly2D function to fit a plane to a gridded dataset to remove a planar trend from the data. The second defines a simplified 2D gaussian function and uses it to define the location of a peak in XY space using random XYZ data.

Example One — Remove Planar Trend Using Poly2D

Here is an example in which a matrix is filled with a two-dimensional sinusoid with a planar trend that overwhelms the sinusoid. The example shows how you might fit a plane to the data to remove the trend. First, make the data matrix, fill it with values, and display the matrix as an image:

```
Make/O/N=(20,20) MatrixWave
SetScale/I x 0,2*pi,MatrixWave
SetScale/I y 0,2*pi,MatrixWave
MatrixWave = sin(x) + sin(y) + 5*x + 10*y
Display;AppendImage MatrixWave
```

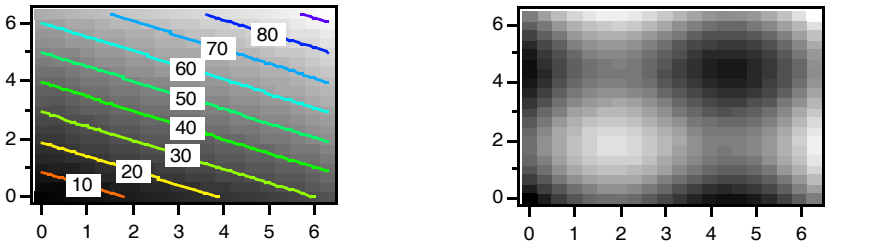
These commands make a graph displaying an image like the one that follows. Note the gradient from the lower left to the upper right:



We are ready to do the fit.

1. Choose Curve Fitting from the Analysis menu to bring up the Curve Fitting dialog.
2. If you have not already done so, choose Show Multivariate Functions from the Function menu.
3. Choose Poly2D from the Function menu.
4. Make sure the 2D Polynomial Order is set to 1.
5. Choose MatrixWave from the Y Data menu.
6. Click the Output Options tab.
7. Choose _auto trace_ from the Residual menu.

The result is the original graph with a contour plot showing the fit to the data, and a new graph of the residuals, showing the sinusoidal signal left over from the fit:

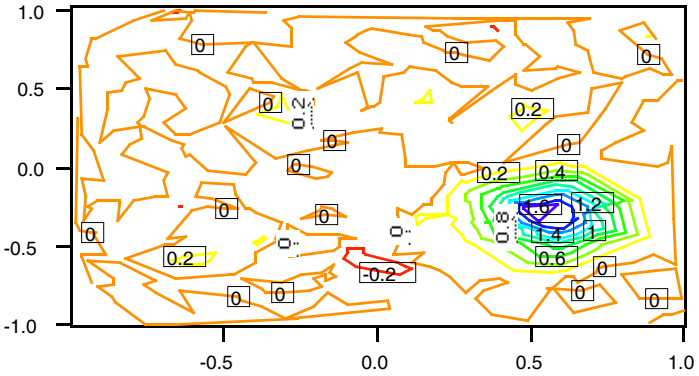


Similarly, you can use the ImageRemoveBackground operation, which provides a one-step operation to do the same fit. It has no dialog support, however.

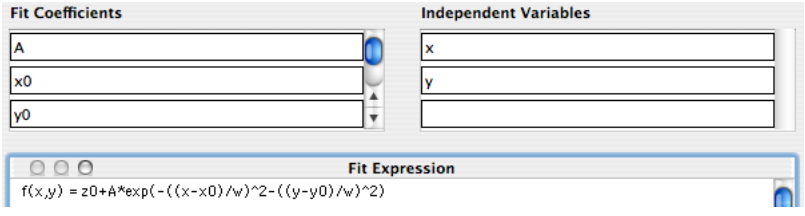
Example Two — User-Defined Simplified 2D Gaussian Fit

In this example, we have data defining a spot which we wish to fit with a 2D Gaussian to find the center of the spot. For some reason this data is in the form of XYZ triplets with random X and Y coordinates. These commands will generate the example data:

```
Make/n=300 SpotXData, SpotYData, SpotZData
SpotXData = enoise(1)
SpotYData = enoise(1)
// make a gaussian centered at {0.55, -0.3}
SpotZData = 2*exp(-((SpotXData-.55)/.2)^2 -((SpotYData+.3)/.2)^2)+gnoise(.1)
Display; AppendXYZContour SpotZData vs {SpotXData,SpotYData}
```



Now bring up the Curve Fitting dialog and click the New Fit Function button so that you can enter your user-defined fit function. We have reason to believe that the spot is circular so the gaussian can use the same width in the X and Y directions, and there is no need for the cross-correlation term. Thus, the new function has a z0 coefficient for the baseline offset (which doesn't show in the figure below), A for amplitude, x0 and y0 for the X and Y location and w for width. Here is what it looks like in the New Fit Function dialog:



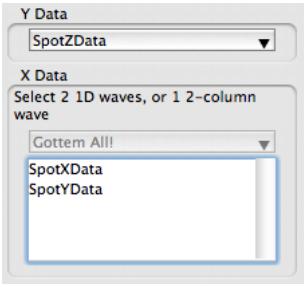
Click Save Fit Function Now to save the function in the Procedure window and return to the Curve Fitting dialog. The new function is selected in the Function menu automatically.

To perform the fit choose:

1. SpotZData in the Y Data menu.

2. SpotXData in the X Data menu.
3. SpotYData in the X Data menu.

At this point, the data selection area of the Function and Data tab looks like this:



Continuing,

4. Click the Coefficients tab (the error box at the bottom shows that we must enter initial guesses).
5. Enter initial guesses: we set z0 to 0, A to 2, x0 to 0.5, y0 to -0.3, and width to 0.5.
6. For our problem, residuals and destination aren't really important since we just want to know the coordinates of the spot center. We click Do It and get this in history:

```
FuncFit SimpleGaussian W_coef SpotZData /X={SpotXData,SpotYData} /D
  Fit converged properly
  fit_SpotZData= SimpleGaussian(W_coef,x,y)
  W_coef={0.0094545,2.0668,0.54885,-0.29678,0.19363}
  V_chisq= 2.55898; V_npnts= 300; V_numNaNs= 0; V_numINFs= 0;
  W_sigma={0.00569,0.0468,0.00383,0.00616,0.00428}
  Coefficient values ± one standard deviation
   z0  = 0.0094545 ± 0.00569
   A   = 2.0668 ± 0.0468
   x0  = 0.54885 ± 0.00383
   y0  = -0.29678 ± 0.00616
   w   = 0.19363 ± 0.00428
```

The output shows that the fit has determined that the center of the spot is {0.54885, -0.29678}.

Problems with the Curve Fitting Dialog

Occasionally you may find that things don't work the way you expect when using the Curve Fitting dialog. Common problems are:

- You can't find your user-defined function in the Function menu.

  This usually happens for one of two reasons: either your function is a multivariate function or it is an old-style function. The problem is solved by choosing Show Multivariate Functions or Show Old-Style Functions from the Function menu on the Function and Data tab.

  If you find that choosing Show Old-Style Functions makes your fit function appear, you may want to consider clicking the Edit Fit Function button, which makes the Edit Fit Function dialog appear. Part of the initialization for the dialog involves revising your fit function to make it conform to current standards. While you're there you can give your fit coefficients mnemonic names.

- You get a message that "Igor can't determine the number of coefficients…".

  This happens when you click the Coefficients tab when you are using an external function or a user-defined function that is so complicated that the dialog can't parse the function code to determine how many coefficients are required.

  The only way to get around this is to choose an explicit coefficient wave (**The Coefficient Wave** on page III-174). The dialog will then use the number of points in the coefficient wave to determine the number of coefficients.

Inputs and Outputs for Built-In Fits

There are a number of variables and waves that provide various kinds of input and output to a curve fit. Usually you will use the Curve Fitting dialog and the dialog will make it clear what you need, and detailed descriptions are available in various places in this chapter. For a quick introduction, here is a table that lists the waves and variables used for fitting to a built-in function.

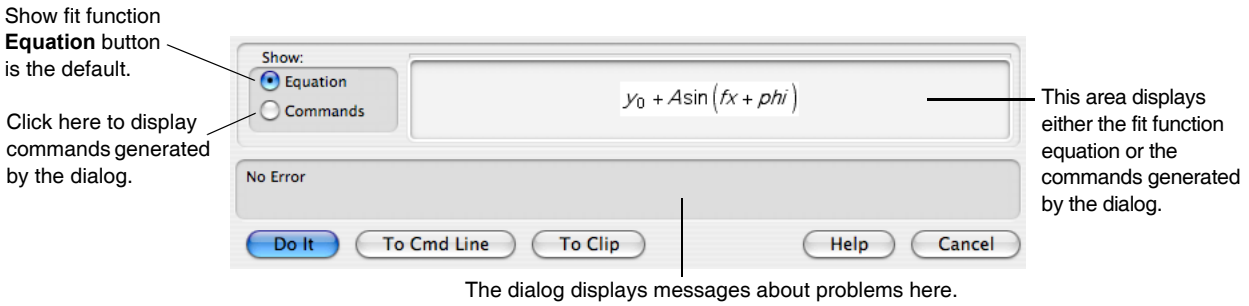
Wave or Variable	Type	What It Is Used For
Dependent variable data wave	Input	Contains measured values of the dependent variable of the curve to fit. Often referred to as "Y data".
Independent variable data wave	Input	Contains measured values of the independent variable of the curve to fit. Often referred to as "X data".
Destination wave	Optional output	For graphical feedback during and after the fit. The destination wave continually updates during the fit to show the fit function evaluated with the current coefficients.
Residual wave	Optional output	Difference between the data and the model.
Weighting wave	Optional input	Used to control how much individual Y data points contribute to the search for the output coefficients.
System variables K0, K1, K2 …	Input and output	*Built-in fit functions only.* Optionally takes initial guesses from the system variables and updates them at the end of the fit.
Coefficients wave By default, W_coef.	Input and Output	Takes initial guesses from the coefficients wave, updates it during the fit and leaves final coefficients in it. See the reference for CurveFit and FuncFit for additional options.
Epsilon wave	Optional input	*User-defined fit functions only.* Used by the curve fitting algorithm to calculate partial derivatives with respect to the coefficients.
W_sigma	Output	Creates this wave and stores the estimates of error for the coefficients in it.
V_<xxx>	Input	There are a number of special variables, such as V_FitOptions, that you can set to tweak the behavior of the curve fitting algorithms.
V_<xxx>	Output	Creates and sets a number of variables such as V_chisq and V_npnts. These contain various statistics found by the curve fit.
M_Covar	Output	Optionally creates a matrix wave containing the "covariance matrix". It can be used to generate advanced statistics.
Other waves	Optional input and output	User-supplied or automatically generated waves for displaying confidence and prediction bands, and for specifying constraints on coefficient values.

Detailed Description of the Curve Fitting Dialog Tabs

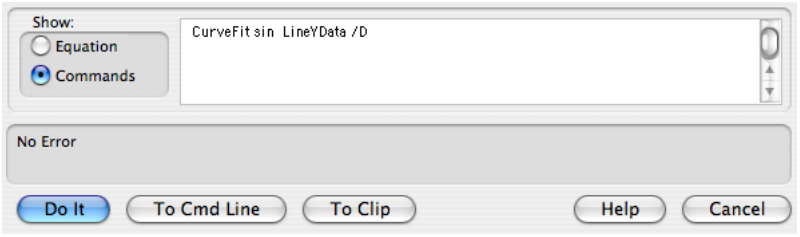
This section describes all the controls on every tab and on the main pane of the Curve Fitting dialog, in excruciating, nitty-gritty detail. Enjoy!

Global Controls

The controls at the bottom of the dialog are always available:



The display area with the **Commands** button turned on:



Do It: copies the generated commands to the command line and executes them.

To Cmd Line: copies the generated commands to the command line but does not execute them. You might use this button if you want to edit the command line to add an option not supported by the dialog.

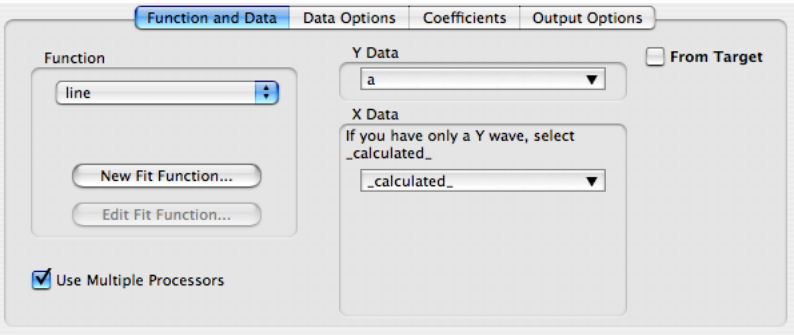
To Clip: copies the generated commands to the Clipboard. You can then paste the commands into a notebook or procedure window. Useful for writing procedures for curve fitting.

Help: takes you to an Igor Help window displaying general help for the dialog.

Cancel: leaves the dialog without doing anything.

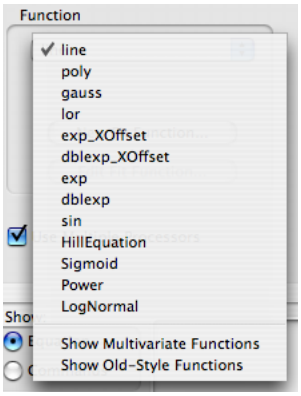
Function and Data Tab

The Function and Data tab has a variety of appearances depending on the function chosen in the Function menu. Here is what it looks like when the built-in gauss function is chosen:



The Data wave selection area as it appears for a univariate function.

Function menu: The main purpose is to choose a function to fit.



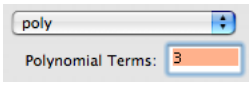
The menu also has two items that control what functions are displayed in the menu.

Choose Show Multivariate functions to include functions of more than one independent variable.

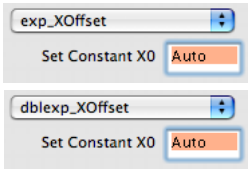
Choose Show Old-Style Functions to display functions that lack the FitFunc keyword. See **User-Defined Fitting Function: Detailed Description** on page III-217 for details on the FitFunc keyword. You may need to choose this item if you have fitting functions from a version of Igor Pro older than version 4.

Some of the fit functions require additional information that is collected by customized items that appear when you select the function.

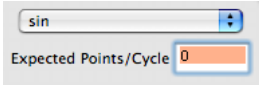
Polynomial Terms: appears when you choose the poly function to fit a polynomial. Note that the number of terms is one greater than the degree — set this to three for a quadratic polynomial.



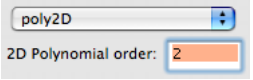
Set Constant X0: appears when you select the exp_XOffset or dblexp_XOffset function to fit an exponential or sum of two exponentials. X0 is a constant subtracted from the X values to move the X range for fitting closer to zero. This eliminates numerical problems that arise when evaluating exponentials over X ranges even moderately far from zero. Setting X0 to Auto causes it to be set to the minimum X value in your input data. Setting to a number overrides this default behavior.



Expected Points/Cycle: appears when you choose the sin function. Use it to set the approximate number of data points in one cycle of the sin function. Helps the automatic initial guess come up with good guesses. If it is set to less than four, Igor will use the default of seven.



2D Polynomial Order: appears when you choose the Poly2D function to fit a two-dimensional polynomial (a multivariate function — only appears in the menu when you have chosen Show Multivariate Functions). Sets the *order* of the polynomial, not the number of terms. Because a given order includes a term for X and a term for Y plus cross terms, the number of terms is (N+1)(N+2)/2 where N is the order.



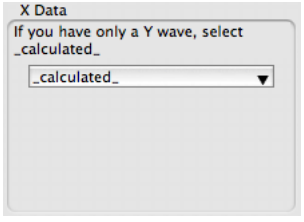
New Fit Function: click this button to bring up a dialog in which you can define your own fitting function.

Edit Fit Function: this button is available when you have chosen a user-defined function from the Function menu. Brings up a dialog in which you can edit your user-defined function.

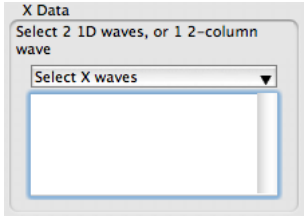
Y Data: select a wave containing the dependent variable data to fit. When a multivariate function is chosen in the Function menu, the Y Data menu shows1D waves and waves with dimensions matching the number of independent variables used by the function.

X Data: this area changes depending on the function and Y Data wave chosen:

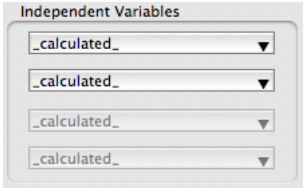
X Data area when a univariate function is selected.



A multivariate function and 1D Y wave are selected.



A multivariate function and N-Dimensional Y wave are



With a **univariate function**, just the X data menu is shown. Choose _calculated_ if you have just a Y wave; the X values will come from the Y wave's X scaling. The X Data menu shows only waves with the same number of points as the Y wave.

When a **multivariate function** and a **1D Y wave** are selected, it adds a list box below the X wave menu. You must select one X wave for each independent variable used by the fit function, or a single multicolumn wave with the correct number of columns. As you select X waves, they are added to the list. The order of waves in the list is important — it determines which is identified with each of the function's independent variables.

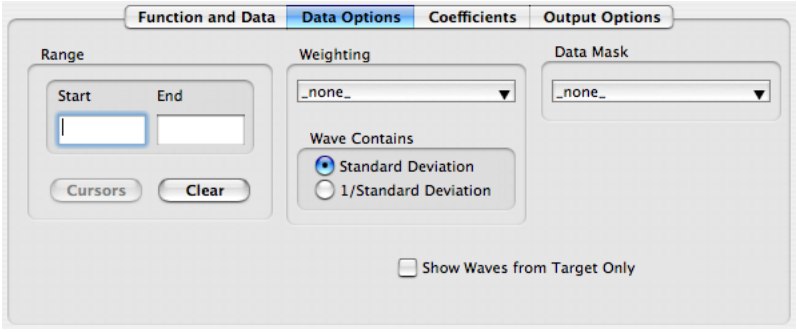
Remove waves from the list by highlighting a wave and pressing Delete (*Macintosh*) or Backspace (*Windows*).

With a **multivariate function** and a **multidimensional Y wave** selected, it displays four independent variable wave menus, one for each dimension that a wave can have. The picture above was taken with a 2D function selected, so two of the menus are available. Each menu displays waves of length matching the corresponding dimension of the Y Data wave. Choose "_calculated_" if the independent variable values will come from the Y wave's dimension scaling.

From Target: when From Target is selected, the Y Data and X Data menus show only waves that are displayed in the top table or graph. Makes it easier to find waves in the menus when you are working on an experiment with many waves. When you click the From Target checkbox, it will attempt to select appropriate waves for the X and Y data.

Use Multiple Processors: If you are running on a computer with multiple processors, this checkbox is on by default. In certain cases, it causes the curve fit code to break some computations into more than one thread that can run simultaneously. See **Curve Fitting with Multiple Processors** on page III-216.

Data Options Tab



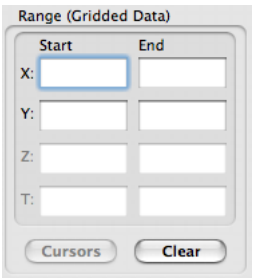
Range: enter point numbers for starting and ending points when fitting to a subset of the Y data.

Historical Note: these boxes used to require X values, they now require point numbers.

Cursors: available when the top window is a graph displaying the Y data and the graph has the graph cursors on the Y data trace. Click this button to enter text in the Start and End range boxes that will restrict fitting to the data between the graph cursors.

Note: If the data use both a Y wave and an X wave, and the X values are in random order, you won't get the expected result.

Clear: click this button to remove text from the Start and End range boxes.



The **range** box changes if you have selected a multivariate function and multidimensional Y wave. The dialog presents Start and End range boxes for each dimension of the Y wave.

Weighting: select a wave that contains weighting values. Only waves that match the Y wave in number of points and dimensions are shown in this menu. See **Weighting** on page III-179 for details.

Wave Contains: select Standard Deviation if the weighting wave contains values of standard deviation for each Y data point. A larger value decreases the influence of a point on the fit.

Select 1/Standard Deviation if your weighting wave contains values of the reciprocal of the standard deviation. A larger value increases the influence of the point on the fit.

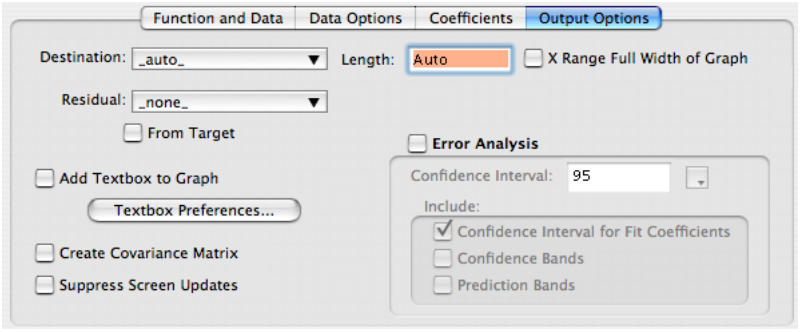
Data Mask: select a wave that contains ones and zeroes or NaN's indicating which Y Data points should be included in the fit. Only waves that match the Y wave in number of points and dimensions are shown in this menu. A one indicates a data point that should be included, a zero or NaN (Not a Number or blank in a table) indicates a point that should be excluded.

Coefficients Tab

The coefficients tab is quite complex. It is completely explained in the various sections on how to do a fit. See **Two Useful Additions: Holding a Coefficient and Generating Residuals** on page III-162, **Automatic Guesses Didn't Work** on page III-164, **Coefficients Tab for a User-Defined Function** on page III-173, and **The Coefficient Wave** on page III-174.

Output Options Tab

The output options tab has settings that control the reporting and display of fit results:



Destination: select a wave to receive model values from the fit. Updated on each iteration so you can follow the fit progress by the graph display. See **The Destination Wave** on page III-176 for details on the Destination menu, the Length box shown above and on the New Wave box that isn't shown above.

X Range Full Width of Graph: If you have restricted the range of the fit using graph cursors, the auto destination wave will cover only the range selected. Select this checkbox to make the auto destination cover the full width of the graph.

Residual: select a wave to receive calculated values of residuals, or the differences between the model and the data. See **Computing Residuals** on page III-191 for details on residuals and on the various selections you can make from this menu.

Error Analysis: selects various kinds of statistical error analysis. See **Confidence Bands and Coefficient Confidence Intervals** on page III-194 for details.

Add Textbox to Graph: when selected, a textbox with information about the fit will be added to the graph containing the Y data. Click the **Textbox Preferences** button to display a dialog in which you can select various bits of information to be included in the text box.

Create Covariance Matrix: when this is selected, the dialog generates the command to create a covariance matrix for the fit. See **Covariance Matrix** on page III-197 for details on the covariance matrix.

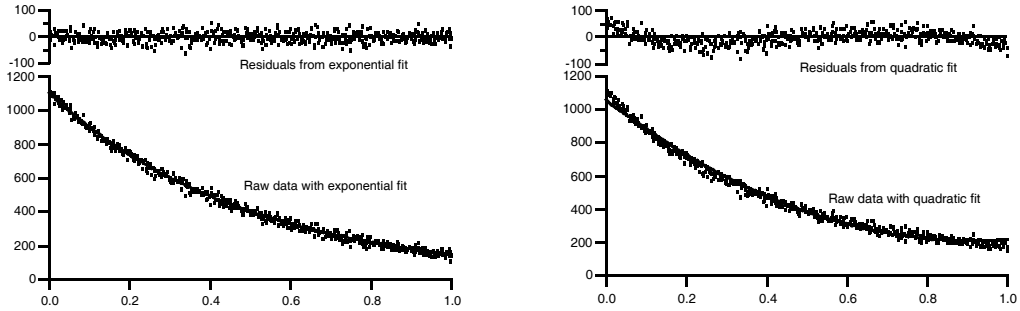
Suppress Screen Updates: when this is selected, graphs and tables are not updated while the fit progresses. This can greatly speed up the fitting process, especially if the fit involves a contour or image plot, but reduces the feedback you get during the fit.

Computing Residuals

A residual is what is left when you subtract your fitting function model from your raw data.

Ideally, your raw data is equal to some known function plus random noise. If you subtract the function from your data, what's left should be noise. If this is not the case, then the function doesn't properly fit your raw data.

The graphs below illustrate some exponential raw data fitted to an exponential function and to a quadratic (3 term polynomial). The residuals from the exponential fit are random whereas the residuals from the quadratic display a trend overlaid on the random scatter. This indicates that the quadratic is not a good fit for the data.



The easiest way to make a graph such as these is to let it proceed automatically using the Residual pop-up menu in the Output Options tab of the Curve Fitting dialog. The graphs above were made this way with some minor tweaks to improve the display.

The residuals are recalculated at every iteration of a fit. If the residuals are displayed on a graph, you can watch the residuals change as the fit proceeds.

In addition to providing an easy way to compute residuals and add the residual plot to a graph, it prints the wave assignment used to create the residuals into the history area as part of the curve fitting process. For instance, this is the result of a line fit to waveform data:

```
•CurveFit line LineYData /X=LineXData /D /R
  fit_LineYData= W_coef[0]+W_coef[1]*x
  Res_LineYData= LineYData - (W_coef[0]+W_coef[1]*LineXData)
  W_coef={0.73804,2.4492}
  V_chisq= 15.6414; V_npnts= 20; V_numNaNs= 0; V_numINFs= 0;
  V_q= 1; V_Rab= -0.337408; V_Pr= 0.934854;
  W_sigma={0.482,0.219}
  Coefficient values ± one standard deviation
   a = 0.73804 ± 48.2
   b = 2.4492 ± 21.9
```

In this case, a wave called "LineYData" was fit with a straight line model. _auto trace_ was selected for both the destination (/D flag) and the residual (/R flag), so the waves fit_LineYData and Res_LineYData were created. The third line above gives the equation for calculating the residuals. You can copy this line if you wish to recalculate the residuals at a later time. You can edit the line if you want to calculate the residuals in a different way. See **Calculating Residuals After the Fit** on page III-193.

Residuals Using Auto Trace

If you choose _auto trace_ from the Residual menu, it will automatically create a wave for residual values and, if the Y data are displayed in the top graph, it will append the residual wave to the graph.

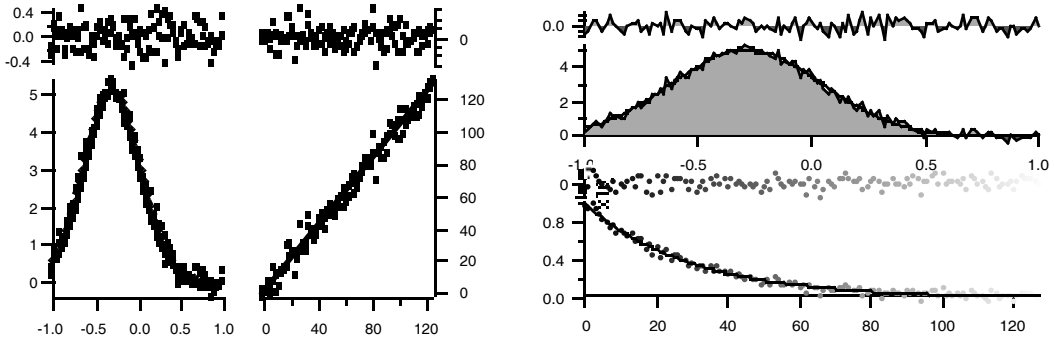
The automatically-created residual wave is named by prepending "Res_" to the name of the Y data wave. If the resulting name is too long, it will be truncated. If a wave with that name already exists, it will be reused, overwriting the values already in the wave. The residual wave is made with the same number of points as the data wave, with one residual value calculated for each data point.

If you fit to a subrange of the data, residual values are calculated only for the points within the subrange. Thus, you can use the auto-residual option with a piece-wise fit to build up the residuals by several curve fitting operations.

If you fit the same data again, the same residual wave will be used. Thus, to preserve a previous result, you must rename the residual wave. This can be done using the Rename item in the Data menu, or the Rename command on the command line.

Residuals are displayed on a stacked graph like those above. This is done by shortening the axis used to display the Y data, and positioning a new free axis above the axis used for the Y data. The residual plot occupies the space made by shortening the data axis. The axis that Igor creates is given a name derived from the axis used to display the Y data, either "Res_Left" or "Res_Right" if the Y data are displayed on the standard axes. If the data axis is a named free axis, the automatically-generated axis is given a name created by prepending "Res_" to the name of the data axis.

Igor tries hard to make the appended residual plot look nice by copying the formatting of the Y data trace and making the residual trace identical so that you can identify which residual trace goes with a given Y data trace. The axis formatting for the residual axis is copied from the vertical axis used for the data trace, and the residual axis is positioned directly above the data axis. Any other vertical axes that might be in the way are also shortened to make room. Here are two examples of graphs that have been modified by the auto-trace residuals:



It is likely that the automatic formatting won't be quite what you want, but it will probably be close enough that you can use the Modify Axis and Modify Trace Appearance dialogs to tweak the formatting. For details see Chapter II-12, **Graphs**, especially the sections **Creating Graphs with Multiple Axes** on page II-292 and **Creating Stacked Plots** on page II-293.

Removing the Residual Auto Trace

When an auto-trace residual plot is added to a graph, it modifies the axis used to plot the original Y data. If you remove the auto-trace residual from the graph, the residual axis is removed and in most cases the Y data axis is restored to its previous state.

In some complicated graphs the restoration of the data axis isn't done correctly. To restore the graph, double-click the Y data axis to bring up the Modify Axes dialog and select the Axis tab. You will find two settings labeled "Draw between … and … % of normal". Typically, the correct settings will be 0 and 100.

Residuals Using Auto Wave

Because the changes to the graph formatting are substantial, you may want the automatic residual wave created and filled in but not appended to the graph. To accomplish this, simply choose _Auto Wave_ from the Residual pop-up menu in the Curve Fitting dialog. Once the wave is made by the curve fitting process, you can append it to a graph, or use it in any other way you wish.

Residuals Using an Explicit Residual Wave

You can choose a wave from the Residual pop-up menu and that wave will be filled with residual values. In this case, it does not append the wave to the top graph. You can use this technique to make graphs with the formatting completely under your own control, or to use the residuals for some other purpose.

Only waves having the same number of points as the Y data wave are listed in the menu. If you don't want to let the dialog create the wave, you would first create a suitable wave by duplicating the Y data wave using the Duplicate Waves item in the Data menu, or using the Duplicate command:

```
Duplicate yData, residuals_poly3
```

It is often a good idea to set the wave to NaN, especially when fitting to a subrange of the data:

```
residuals_poly3=NaN
```

After the wave is duplicated, it would typically be appended to a graph or table before it is used in curve fitting.

Explicit Residual Wave Using New Wave

The easiest way to make an explicit residual wave is to let the Curve Fitting dialog do it for you. You do this by choosing _New Wave_ in the Residual menu. A box appears allowing you to enter a name for the new wave. The dialog will then generate the required commands to make the wave before the curve fit starts. The wave made this way will not be added to a graph. You will need to do that yourself after the fit is finished.

Calculating Residuals After the Fit

You may wish to avoid the overhead of calculating residuals during the fitting process. This section describes ways to calculate the residuals after a curve fit without depending on automatic wave creation.

Graphs similar to the ones at the top of this section can be made by appending a residuals wave using a free left axis. Then, under the Axis tab of the Modify Axes dialog, the distance for the free axis was set to zero and the axis was set to draw between 80 and 100% of normal. The normal left axis was set to draw between 0 and 70% and axis standoff was turned off for both left and bottom axes.

Here are representative commands used to accomplish this.

```
// Make sample data
Make/N=500 xData, yData
xData = x/500 + gnoise(1/1000)
yData = 100 + 1000*exp(-.005*x) + gnoise(20)

// Do exponential fit with auto-trace
CurveFit exp yData /X=xData /D
Rename fit_yData, fit_yData_exp

// Calculate exponential residuals using interpolation in
// the auto trace wave to get model values
Duplicate yData, residuals_exp
residuals_exp = yData - fit_yData_exp(xData)

// Do polynomial fit with auto-trace
CurveFit poly 3, yData /X=xData /D
Rename fit_yData fit_yData_poly3

// Find polynomial residuals
Duplicate yData, residuals_poly3
residuals_poly3 = yData - fit_yData_poly3(xData)
```

Calculating model values by interpolating in the auto trace wave may not be sufficiently accurate. Instead, you can calculate the exact value using the actual fitting expression. When the fit finishes, it prints the equation for the fit in the history. With a little bit of editing you can create a wave assignment for calculating residuals. Here are the assignments from the history for the above fits:

```
fit_yData= poly(W_coef,x)
fit_yData= W_coef[0]+W_coef[1]*exp(-W_coef[2]*x)
```

We can convert these into residuals calculations like this:

```
residuals_poly3 = yData - poly(W_coef,xData)
residuals_exp = yData - (W_coef[0]+W_coef[1]*exp(-W_coef[2]*xData))
```

Note that we replaced "x" with "xData" because we have tabulated x values. If we had been fitting equally spaced data then we would not have had a wave of tabulated x values and would have left the "x" alone.

This technique for calculating residuals can also be used if you create and use an explicit destination wave. In this case the residuals are simply the difference between the data and the destination wave. For example, we could have done the exp fit and residual calculations as follows:

```
Duplicate yData, yDataExpFit,residuals_exp
// explicit destination wave using /D=wave
CurveFit exp yData /X=xData /D=yDataExpFit
residuals_exp = yData - yDataExpFit
```

Estimates of Error

Igor automatically calculates the estimated error (standard deviation) for each of the coefficients in a curve fit. When you perform a curve fit, it creates a wave called W_sigma. Each point of W_sigma is set to the estimated error of the corresponding coefficients in the fit. The estimated errors are also indicated in the history area, along with the other results from the fit. If you don't provide a weighting wave, the sigma values are estimated from the residuals. This implicitly assumes that the errors are normally distributed with zero mean and constant variance and that the fit function is a good description of the data.

The coefficients and their sigma values are estimates (usually remarkably good estimates) of what you would get if you performed the same fit an infinite number of times on the same underlying data (but with different noise each time) and then calculated the mean and standard deviation for each coefficient.

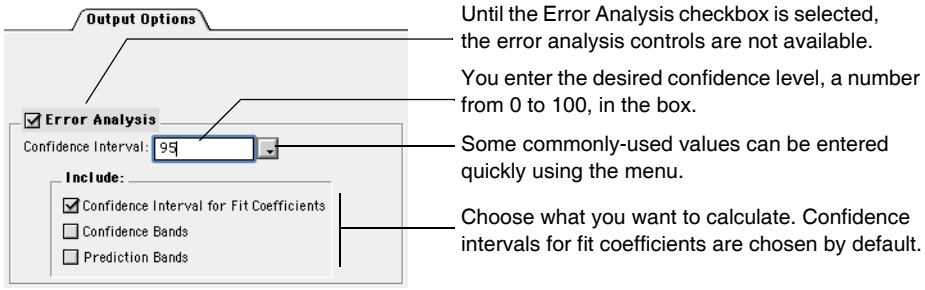
Confidence Bands and Coefficient Confidence Intervals

You can graphically display the uncertainty of a model fit to data by adding confidence bands or prediction bands to your graph. These are curves that show the region within which your model or measured data are expected to fall with a certain level of probability. A confidence band shows the region within which the model is expected to fall while a prediction band shows the region within which random samples from that model plus random errors are expected to fall.

You can also calculate a confidence interval for the fit coefficients. A confidence interval estimates the interval within which the real coefficient will fall with a certain probability.

Note: Confidence and prediction bands are not available for multivariate curve fits.

You control the display of confidence and prediction bands and the calculation of coefficient confidence intervals using the Error Analysis section of the Output Options tab of the Curve Fitting dialog:



Using the line fit example at the beginning of this chapter (see **A Simple Case — Fitting to a Built-In Function: Line Fit** on page III-160), we set the confidence level to 95% and selected all three error analysis options to generate this output and graph:

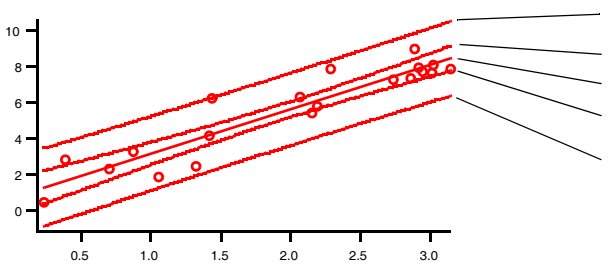
Dialog has added the /F with parameters to select error analysis options.

```
•CurveFit line LineYData /X=LineXData /D /F={0.950000, 7}
   fit_LineYData= W_coef[0]+W_coef[1]*x
```

```
W_coef={-0.037971,2.9298}
V_chisq= 18.25; V_npnts= 20; V_numNaNs= 0; V_numINFs= 0;
V_startRow= 0; V_endRow= 19; V_q= 1; V_Rab= -0.879789;
V_Pr= 0.956769;V_r2= 0.915408;
W_sigma={0.474,0.21}
Fit coefficient confidence intervals at 95.00% confidence level:
W_ParamConfidenceInterval={0.995,0.441,0.95}
Coefficient values ± 95.00% Confidence Interval
   a  =  -0.037971 ± 0.995
   b  =  2.9298 ± 0.441
```

When confidence intervals are available they are listed here instead of the standard deviation.

Coefficient confidence intervals are stored in the wave W_ParamConfidenceInterval. Note that the last point in the wave contains the confidence level used in the calculation.

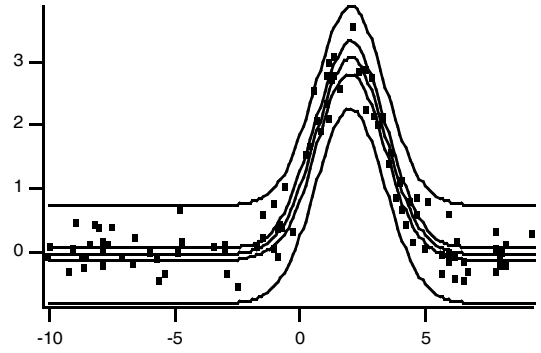


95% of measured points should fall within the prediction band.

If you could repeat the experiment numerous times, 95% of the time the fit line should fall within the confidence band.

You can do this with nonlinear functions also, but be aware that it is only an approximation for nonlinear functions:

```
Make/O/N=100 GDataX, GDataY                        // waves for data
GDataX = enoise(10)                                // Random X values
GDataY = 3*exp(-((GDataX-2)/2)^2) + gnoise(0.3)    // Gaussian plus noise
Display GDataY vs GDataX                           // graph the data
ModifyGraph mode=2,lsize=2                         // as dots
CurveFit Gauss GDataY /X=GDataX /D/F={.99, 3}
```



The dialog supports only automatically-generated waves for confidence bands. The CurveFit and FuncFit operations support several other options including an error bar-style display. See **CurveFit** on page V-85 for details.

Calculating Confidence Intervals After the Fit

You can use the values from the W_sigma wave to calculate a confidence interval for the fit coefficients after the fit is finished. Use the StudentT function to do this. The following information was printed into the history following a curve fit:

```
Fit converged properly
fit_junk= f(coeffs,x)
coeffs={4.3039,1.9014}
V_chisq= 101695; V_npnts= 128; V_numNaNs= 0; V_numINFs= 0;
W_sigma={4.99,0.0679}
```

To calculate the 95 per cent confidence interval for fit coefficients and deposit the values into another wave, you could execute the following lines:

```
Duplicate W_sigma, ConfInterval
ConfInterval = W_sigma[p]*StudentT(0.95, V_npnts-numpnts(coeffs))
```

Naturally, you could simply type "126" instead of "V_npnts-numpnts(coeffs)", but as written the line will work unaltered for any fit. When we did this following the fit in the example, these were the results:

```
ConfInterval = {9.86734,0.134469}
```

Clearly, coeffs[0] is not significantly different from zero.

Confidence Band Waves

New waves containing values required to display confidence and prediction bands are created by the curve fit if you have selected these options. The number of waves and the names depend on the type number and the style. For a contour band, there are two waves for the upper and a lower contour. Only one wave is required to display error bars. For details, see the **CurveFit** operation on page V-85.

Some Statistics

Calculation of the confidence and prediction bands involve some statistical assumptions. First, of course, the measurement errors are assumed to be normally distributed. Departures from normality usually have to be fairly substantial to cause real problems.

If you don't supply a weighting wave, the distribution of errors is estimated from the residuals. In making this estimate, the distribution is assumed to be not only normal, but also uniform with mean of zero. That is, the error distribution is centered on the model and the standard deviation of the errors is the same for all values of the independent variable. The assumption of zero mean requires that the model be correct; that is, it assumes that the measured data truly represent the model plus random normal errors.

Some data sets are not well characterized by the assumption that the errors are uniform. In that case, you should specify the error distribution by supplying a weighting wave (see **Weighting** on page III-179). If you do this, your error estimates are used for determining the uncertainties in the fit coefficients, and, therefore, also in calculating the confidence band.

The confidence band relies only on the model coefficients and the estimated uncertainties in the coefficients, and will always be calculated taking into account error estimates provided by a weighting wave. The prediction band, on the other hand, also depends on the distribution of measurement errors at each point. These errors are not taken into account, however, and only the uniform measurement error estimated from the residuals are used.

The calculation of the confidence and prediction bands is based on an estimate of the variance of a predicted model value:

$$V(\hat{Y}) = a^T C a$$
$$a = \delta F/\delta p|_x$$

Here, $\hat{Y}$ is the predicted value of the model at a given value of the independent variable X, $\boldsymbol{a}$ is the vector of partial derivatives of the model with respect to the coefficients evaluated at the given value of the independent variable, and $\boldsymbol{C}$ is the covariance matrix. Often you see the $\boldsymbol{a}^T\boldsymbol{C}\boldsymbol{a}$ term multiplied by σ^2, the sample variance, but this is included in the covariance matrix. The confidence interval and prediction interval are calculated as:

$$CI = t(n-p, 1-\alpha/2)[V(\hat{Y})]^{1/2} \text{ and } PI = t(n-p,1-\alpha/2)[\sigma^2 + V(\hat{Y})]^{1/2}.$$

The quantities calculated by these equations are the magnitudes of the intervals. These are the values used for error bars. These values are added to the model values ($\hat{Y}$) to generate the waves used to display the bands as contours. The function $t(n-p, 1-\alpha/2)$ is the point on a Student's t distribution having probability $1-\alpha/2$, and σ^2 is the sample variance. In the calculation of the prediction interval, the value used for σ^2 is the uniform value estimated from the residuals. This is not correct if you have supplied a weighting

wave with nonuniform values because there is no information on the correct values of the sample variance for arbitrary values of the independent variable. You can calculate the correct prediction interval using the **StudentT** function. You will need a value of the derivatives of your fitting function with respect to the fitting coefficients. You can either differentiate your function and write another function to provide derivatives, or you can use a numerical approximation. Igor uses a numerical approximation.

Confidence Bands and Nonlinear Functions

Strictly speaking, the discussion and equations above are only correct for functions that are linear in the fitting coefficients. In that case, the vector **a** is simply a vector of the basis functions. For a polynomial fit, that means 1, x, x^2, etc. When the fitting function is nonlinear, the equation results from an approximation that uses only the linear term of a Taylor expansion. Thus, the confidence bands and prediction bands are only approximate. It is impossible to say how bad the approximation is, as it depends heavily on the function.

Covariance Matrix

If you select the Create Covariance Matrix checkbox in the Output Options tab of the Curve Fitting dialog, it generates a covariance matrix for the curve fitting coefficients. This is available for all of the fits except the straight-line fit. Instead, a straight-line fit generates the special output variable V_rab giving the covariance between the slope and Y intercept.

By default (if you are using the Curve Fitting dialog) it generates a matrix wave having N rows and columns, where N is the number of coefficients. The name of the wave is M_Covar. This wave can be used in matrix operations. If you are using the CurveFit, FuncFit or FuncFitMD operations from the command line or in a user procedure, use the /M=2 flag to generate a matrix wave.

Originally, curve fits created one 1D wave for each fit coefficient. The waves taken all together made up the covariance matrix. For compatibility with previous versions, the /M=1 flag still produces multiple 1D waves with names W_Covar*n*. Please don't do this on purpose.

The diagonal elements of the matrix, M_Covar[i][i], are the variances for parameter i. The variance is the square of sigma, the standard deviation of the estimated error for that parameter.

The covariance matrix is described in detail in *Numerical Recipes in C*, page 531 and section 14.5. Also see the discussion under **Weighting** on page III-179.

Correlation Matrix

Use the following commands to calculate a correlation matrix from the covariance matrix produced during a curve fit:

```
Duplicate M_Covar, CorMat  // You can use any name instead of CorMat
CorMat = M_Covar[p][q]/sqrt(M_Covar[p][p]*M_Covar[q][q])
```

A correlation matrix is a normalized form of the covariance matrix. Each element shows the correlation between two fit coefficients as a number between -1 and 1. The correlation between two coefficients is perfect if the corresponding element is 1, it is a perfect inverse correlation if the element is -1, and there is no correlation if it is 0.

Curve fits in which an element of the correlation matrix is very close to 1 or -1 may signal "identifiability" problems. That is, the fit doesn't distinguish between two of the parameters very well, and so the fit isn't very well constrained. Sometimes a fit can be rewritten with new parameters that are combinations of the old ones to get around this problem.

Fitting with Constraints

It is sometimes desirable to restrict values of the coefficients to a fitting function. Sometimes fitting functions may allow coefficient values that, while fine mathematically, are not physically reasonable. At other times, some ranges of coefficient values may cause mathematical problems such as singularities in the func-

tion values, or function values that are so large that they overflow the computer representation. In such cases it is often desirable to apply constraints to keep the solution out of the problem areas. It could be that the final solution doesn't involve any active constraints, but the constraints prevent termination of the fit on an error caused by wandering into bad regions on the way to the solution.

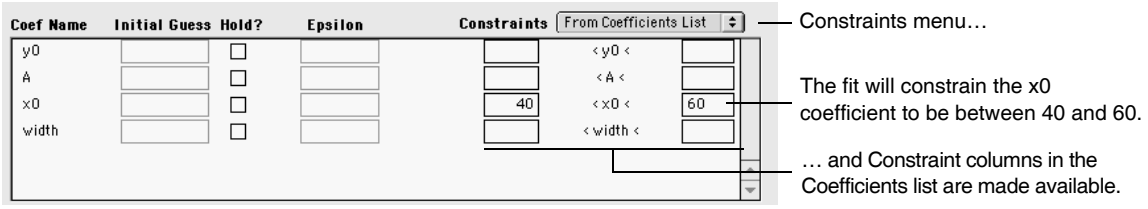
Curve fitting supports constraints on the values of any linear combination of the fit coefficients. The Curve Fitting dialog supports constraints on the value of individual coefficients.

The algorithm used to apply constraints is quite forgiving. Your initial guesses do not have to be within the constraint region (that is, initial guesses can violate the constraints). In most cases, it will simply move the parameters onto a boundary of the constraint region and proceed with the fit. Constraints can even be contradictory ("infeasible" in curve fitting jargon) so long as the violations aren't too severe, and the fit will simply "split the difference" to give you coefficients that are a compromise amongst the infeasible constraints.

Constraints are not available for the built-in line, poly and poly2D fit functions. To apply constraints to these fit functions you must create a user-defined fit function.

Constraints Using the Curve Fitting Dialog

The Coefficients Tab of the Curve Fitting dialog includes a menu to enable fitting with constraints. When you select the checkbox, the constraints section of the Coefficients list becomes available:



Filling in a value in the left column causes the dialog to generate a command to constrain the corresponding coefficient to values greater than the value you enter. Filling in a value in the right column constrains the corresponding coefficient to values less than the value you enter. A box that is left empty does not generate any constraint.

The figure above was made with the gauss function chosen. The following commands are generated by the dialog:

```
Make/O/T/N=2 T_Constraints
T_Constraints[0] = {"K2 > 40","K2 < 60"}
CurveFit gauss aa /D /C=T_Constraints
```

A Make command to make a text wave to contain constraint expressions.

A wave assignment to put constraint expressions into the wave.

/C parameter added to CurveFit command line to request a constrained fit.

More complicated constraints are possible, but cannot be entered in the Curve Fitting dialog. This requires that you make a constraints wave before you enter the dialog. Then choose the wave from the menu. See the following sections to learn how to construct the constraints wave.

Complex Constraints Using a Constraints Wave

You can constrain the values of linear combinations of coefficients, but the Curve Fitting dialog provides support only for simple constraints. You can construct an appropriate text wave with constraints before entering the Curve Fitting dialog. Select the wave from the Constraints menu in the Coefficients tab. You can also use the CurveFit or FuncFit commands on the command line with a constraints wave.

Each element of the text wave holds one constraint expression. Using a text wave makes it easy to edit the expressions in a table. Otherwise, you must use a command line like the second line in the example shown above.

Constraint Expressions

Constraint expressions can be arbitrarily complex, and can involve any or all of the fit coefficients. Each expression must have an inequality symbol ("<", "<=", ">", or ">="). In the expressions the symbol Kn (K0, K1, etc.) is used to represent the nth fitting coefficient. This is like the Kn system variables, but they are merely symbolic place holders in constraint expressions. Expressions can involve sums of any combination of the Kn's and factors that multiply or divide the Kn's. Factors may be arbitrarily complex, even nonlinear, as long as they do not involve any of the Kn's. The Kn's cannot be used in a call to a function, and cannot be involved in a nonlinear expression. Here are some legal constraint expressions:

```
K0 > 5
K1+K2 < numVar^2+2      // numVar is a global numeric variable
K0/5 < 2*K1
(numVar+3)*K3 > K1+K2/(numVar-2)
log(numVar)*K3 > 5      // nonlinear factor doesn't involve K3
```

These are not legal:

```
K0*K1 > 5          // K0*K1 is nonlinear
1/K1 < 4           // This is nonlinear: division by K1
ln(K0) < 1         // K0 not allowed as parameter to a function
```

When constraint expressions are parsed, the factors that multiply or divide the Kn's are extracted as literal strings and evaluated separately. Thus, if you have <expression>*K0 or K0/<expression>, <expression> must be executable on its own.

Note: You cannot use a text wave with constraint expressions for fitting from a threadsafe function. You must use the method described in **Constraint Matrix and Vector** on page III-200.

Equality Constraint

You may wish to constrain the value of a fit coefficient to be equal to a particular value. The constraint algorithm does not have a provision for equality constraints. One way to fake this is to use two constraints that require a coefficient to be both greater than and less than a value. For instance, "K1 > 5" and "K1 < 5" will require K1 to be equal to 5.

If it is a single parameter that is to be held equal to a value, this isn't the best method. You are much better off holding a parameter. In the Curve Fitting dialog, simply select the Hold box in the Coefficients list on the Coefficients tab and enter a value in the Initial Guess column. If you are using a command line to do the fit,

```
FuncFit/H="01"...
```

will hold K1 at a particular value. Note that you have to set that value before starting the fit.

Example Fit with Constraint

This example fits to a sum of two exponentials, while constraining the sum of the exponential amplitudes to be less than some limit that might be imposed by theoretical knowledge. The examples here are available in the Curve Fitting section of the Using Igor help file, where they can be conveniently executed directly from the help window. The example uses command lines because the constraint is too complicated to enter into the Curve Fitting dialog.

First, make the data and graph it:

```
Make/O/N=50 expData= 3*exp(-0.2*x) + 3*exp(-0.03*x) + gnoise(.1)
Display expData
ModifyGraph mode=3,marker=8
```

Do a fit without constraints:

```
CurveFit dblExp expData /D/R
```

The following command makes a text wave with a single element containing the string "K1 + K3 < 5" which implements a restriction on the sum of the individual exponential amplitudes.

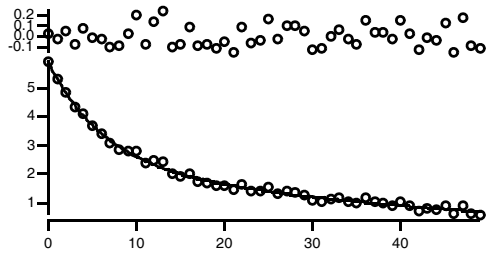
```
Make/O/T CTextWave={"K1 + K3 < 5"}
```

The wave is made using commands so that it could be written into this help file. It may be easier to use the Make Waves item from the Data menu to make the wave, and then display the wave in a table to edit the expressions. Make sure you make Text wave. Do not leave any blank lines in the wave.

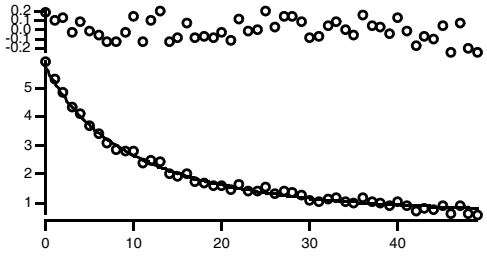
Now do the fit again with constraints:

```
CurveFit dblExp expData /D/R/C=CTextWave
```

In this case, the difference is slight; in the graph of the fit with constraints, notice that the fit line is slightly lower at the left end and slightly higher at the right end than in the standard curve fit, and that difference is reflected in the residual values at the ends:



Standard curve fit



Fit with constraints

The output from a curve fit with constraints includes these lines reporting on the fact that constraints were used, and that the constraint was active in the solution:

```
  --Curve fit with constraints--
     Active Constraint: Desired:  K1+K3<5  Achieved:  K1+K3=5
```

In most cases you will see a message similar to this one. If you have conflicting constraints, it is likely that one or more constraints will be violated. In that case, you will get a report of that fact. The following commands add two more constraints to the example. The new constraints require values for the individual amplitudes that sum to a number greater than 5, while still requiring that the sum be less than 5 (so these are "infeasible constraints"):

```
Make/O/T CTextWave={"K1 + K3 < 5", "K1 > 3.3", "K3 > 2.2"}
CurveFit dblExp expData /D/R/C=CTextWave
```

In most cases, you would have added the new constraints by editing the constraint wave in a table.

Point	CTextWave
0	K1 + K3 < 5
1	K1 > 3.3
2	K3 > 2.2
3	

The curve fit report shows that all three constraints were violated to achieve a solution:

```
--Curve fit with constraints--
   Constraint VIOLATED:  Desired:  K1+K3<5  Achieved:  K1+K3=5.32378
   Constraint VIOLATED:  Desired:  K1>3.3  Achieved:  K1=3.1956
   Constraint VIOLATED:  Desired:  K3>2.2  Achieved:  K3=2.12818
```

Constraint Matrix and Vector

When you do a constrained fit, it parses the constraint expressions and builds a matrix and a vector that describe the constraints.

Each constraint expression is parsed to form a simple expression like $C_0K_0 + C_1K_1 + \ldots \le D$, where the K_i's are the fit coefficients, and the C_i's and D are constants. The constraints can be expressed as the matrix operation $\mathbf{CK} \le \mathbf{D}$, where **C** is a matrix of constants, **K** is the fit coefficient vector, and **D** is a vector of limits. The matrix has dimensions N by M where N is the number of constraint expressions and M is the number of fit coefficients. In most cases, almost all the elements of the matrix are zeroes. In the previous example, the **C** matrix and **D** vector are:

C matrix					D vector
0	1	0	1	0	5
0	-1	0	0	0	-3.3
0	0	0	-1	0	-2.2

This is the form used internally by the curve-fitting code. If you wish, you can build a matrix wave and one-dimensional wave containing the *C* and *D* constants. Instead of using /C=*textwave* to request a fit with constraints, use `/C={matrix, 1Dwave}`.

In general, it is confusing and error-prone to create the right waves for a given set of constraints. However, it is not allowed to use the textwave method for curve fitting from a threadsafe function (see **Multiple Curve Fits Simultaneously** on page III-217). Fortunately, Igor can create the necessary waves when it parses the expressions in a text wave.

You can create waves containing the **C** matrix and **D** vector using the /C flag (note that this flag goes right after the CurveFit command name, not at the end). If you have executed the examples above, you can now execute the following commands to build the waves and display them in a table:

```
CurveFit/C dblExp expData /D/R/C=CTextWave
Edit M_FitConstraint, W_FitConstraint
```

The **C** matrix is named M_FitConstraint and the **D** vector is named W_FitConstraint (by convention, matrix names start with "M_", and 1D wave names start with "W_").

In addition to their usefulness for specifying constraints in a threadsafe function, you can use these waves later to check the constraints. The following commands multiply the generated fit coefficient wave (W_coefs) by the constraint matrix and appends the result to the table made previously:

```
MatrixMultiply M_FitConstraint, W_coef
AppendToTable M_Product
```

The result of the MatrixMultiply operation is the matrix wave M_Product. Note that the values in M_Product are all larger than the values in W_FitConstraint because the constraints were infeasible and resulted in constraint violations.

M_F	M_F	M_F	M_F	M_F	W_FitConstraint	M_product[][0]
0	1	2	3	4		0
0	1	0	1	0	5	4.99998
0	-1	0	0	0	-3.3	-3.05782
0	0	0	-1	0	-2.2	-1.94217

Constrained Curve Fit Pitfalls

Bad Values on the Constraint Boundary

The most likely problem with constrained curve fitting is a value on a constraint boundary that produces a singular matrix error during the curve fit. For instance, it would be reasonable in many applications to require a preexponential multiplier to be positive:

$y = K_0 + K_1 e^{K_2 x}$ and require K_1 to be positive.

It would be natural to write a constraint expression "$K_1 > 0$", but this can lead to problems. If some iteration tries a negative value of K_1, the constraints will set K_1 to be exactly zero. But when K1 is zero, the function has no dependence on K_2 and a singular matrix error results. The solution is to use a constraint that requires K_1 to be greater than some small positive number: "$K_1 > 0.1$", for instance. The value of the limit must be tailored to your application. If you expect the constraint to be inactive when the solution is found, and the fit reports that the constraint was active, you can decrease the limit and do the fit again.

Severe Constraint Conflict

Although the method used for applying the constraints can find a solution even when constraints conflict with each other (are infeasible), it is possible to have a conflict that is so bad that the method fails. This will result in a singular matrix error.

Constraint Region is a Poor Fit

It is possible that the region of coefficient space allowed by the constraints is such a bad fit to the data that a singular matrix error results.

Initial Guesses Far Outside the Constraint Region

Usually if the initial guesses for a fit lie outside the region allowed by the constraints, the fit coefficients will shift into the constraint region on the first iteration. It is possible, however, for initial guesses to be so far from the constraint region that the solution to the constraints fails. This will cause the usual singular matrix error.

Constraints Conflict with a Held Parameter

You cannot hold a parameter and apply a constraint to the same parameter. Thus, this is not allowed:

```
Make/T CWave="K1 > 5"
FuncFit/H="01" myFunc, myCoefs, myData /C=CWave
```

NaNs and INFs in Curve Fits

Curve fits ignore NaNs and INFs in the input data. This is a convenient way to eliminate individual data values from a fit. A better way is to use a data mask wave (see **Using a Mask Wave** on page III-179).

Special Variables for Curve Fitting

There are a number of special variables used for curve fitting input (to provide additional control of the fit) and for output (to provide additional statistics). Knowledgeable users can use the input variables to tweak the fitting process. However, this is usually not needed. Some output variables help users knowledgeable in statistics to evaluate the quality of a curve fit.

To use an input variable interactively, create it from the command line using the Variable operation before performing the fit.

Most of the output variables are automatically created by the CurveFit or FuncFit operations. Some, as indicated below, are not automatically created; you must create them yourself if you want the information they provide.

If you are fitting using a procedure, both the input and output variables can be local or global. It is best to make them local. See **Accessing Variables Used by Igor Operations** on page IV-103 for information on how to use local variables.

If you perform a curve fit interactively via the command line or via the Curve Fitting dialog, the variables will be global. If you use multiple data folders (described in Chapter II-8, **Data Folders**), you need to remember that input and output variables are searched for or created in the current data folder.

The following table lists all of the input and output special variables. Some variables are discussed in more detail in sections following the table.

Variable	I/O	Meaning
V_FitOptions	Input	Miscellaneous options for curve fit.
V_FitTol	Input	Normally, an iterative fit terminates when the fractional decrease of chi-square from one iteration to the next is less than 0.001. If you create a global variable named V_FitTol and set it to a value between 0.1 and 0.00001 then that value will be used as the termination tolerance. Values outside that range will have no effect.
V_tol	Input	(poly fit only) The "singular value threshold". See **Special Considerations for Polynomial Fits** on page III-230.

Variable	I/O	Meaning
V_chisq	Output	A measure of the goodness of fit. It has absolute meaning only if you've specified a weighting wave containing the reciprocal of the standard error for each data point.
V_q	Output	(line fit only) A measure of the believability of chi-square. Valid only if you specified a weighting wave.
V_siga, V_sigb	Output	(line fit only) The probable uncertainties of the intercept (K0 = a) and slope (K1 = b) coefficients for a straight-line fit (to y = a + bx).
V_Rab	Output	(line fit only) The coefficient of correlation between the uncertainty in a (the intercept, K0) and the uncertainty in b (the slope, K1).
V_Pr	Output	(line fit only) The linear correlation coefficient r (also called Pearson's r). Values of +1 or -1 indicate complete correlation while values near zero indicate no correlation.
V_r2	Output	(line fit only) The coefficient of determination, usually called simply "r-squared".
V_npnts	Output	The number of points that were fitted. If you specified a weighting wave then points whose weighting was zero are not included in this count. Also not included are points whose values are NaN or INF.
V_nterms	Output	The number of coefficients in the fit.
V_nheld	Output	The number of coefficients held constant during the fit.
V_numNaNs	Output	The number of NaN values in the fit data. NaNs ignored during a curve fit.
V_numINFs	Output	The number of INF values in the fit data. NaNs ignored during a curve fit.
V_FitError	Input/ Output	Used from a procedure to attempt to recover from errors during the fit.
V_FitQuitReason	Output	Provides additional information about why a nonlinear fit stopped iterating. You must create this variable; it is not automatically created.
V_FitIterStart	Output	Use of V_FitIterStart is obsolete; use all-at-once fit functions instead. See **All-At-Once Fitting Functions** on page III-222 for details. Set to 1 when an iteration starts. Identifies when the user-defined fitting function is called for the first time for a particular iteration. You must create this variable; it is not automatically created.
V_FitMaxIters	Input	Controls the maximum number of passes without convergence before stopping the fit. By default this is 40. You can set V_FitMaxIters to any value greater than 0. If V_FitMaxIters is less than 1 the default value of 40 is used.
V_FitNumIters	Output	Number of iterations. You must create this variable; it is not automatically created.
S_Info	Output	Keyword-value pairs giving certain kinds of information about the fit. You must create this variable; it is not automatically created.

V_FitOptions

There are a number of options that you can invoke for the fitting process by creating a variable named V_FitOptions and setting various bits in it. Set V_FitOptions to 1 to set Bit 0, to 2 to set Bit 1, etc.

Bit 0: Controls X Scaling of Auto-Trace Wave

If V_FitOptions exists and has bit 0 set (`Variable V_fitOptions=1`) and if the Y data wave is on the top graph then the X scaling of the auto-trace destination wave is set to match the appropriate x axis on the graph. This is useful when you want to extrapolate the curve outside the range of x data being fit.

A better way to do this is with the /X flag (not parameter- this flag goes immediately after the CurveFit or FuncFit operation and before the fit function name). See **CurveFit** for details.

Bit 1: Robust Fitting

You can get a form of robust fitting where the sum of the absolute deviations is minimized rather than the squares of the deviations, which tends to deemphasize outlier values. To do this, create V_FitOptions and set bit 1 (`Variable V_fitOptions=2`). **Warning 1**: No attempt to adjust the results returned for the estimated errors or for the correlation matrix has been made. You are on your own. **Warning 2**: Don't set this bit and then forget about it. **Warning 3**: Setting Bit 1 has no effect on line, poly or poly2D fits.

Bit 2: Suppresses Curve Fit Window

Normally, an iterative fit puts up an informative window while the fit is in progress. If you don't want this window to appear, create V_FitOptions and set bit 2 (`Variable V_fitOptions=4`). This may speed things up a bit if you are performing batch fitting on a large number of data sets.

Bit 3: Save Iterates

It is sometimes useful to know the path taken by a curve fit getting to a solution (or failing to). To save his information, create V_FitOptions and set bit 3 (`Variable V_FitOptions=8`). This creates a matrix wave called M_iterates, which contains the values of the fit coefficients at each iteration. The matrix has a row for each iteration and a column for each fit coefficient. The last column contains the value of chi square for each iteration.

V_chisq

V_chisq is a measure of the goodness of fit. It has absolute meaning only if you've specified a weighting wave. See the discussion in the section **Weighting** on page III-179.

V_q

V_q (straight-line fit only) is a measure of the believability of chi-square. It is valid only if you specified a weighting wave. It represents the quantity q which is computed as follows:

```
q = gammq((N-2)/2, chisq/2)
```

where gammq is the incomplete gamma function 1-P(a,x) and N is number of points. A q of 0.1 or higher indicates that the goodness of fit is believable. A q of 0.001 indicates that the goodness of fit may be believable. A q of less than 0.001 indicates systematic errors in your data or that you are fitting to the wrong function.

V_FitError and V_FitQuitReason

When an error occurs during a curve fit, it normally causes any running user-defined procedure to abort.

This makes it impossible for you to write a procedure that attempts to recover from errors. However, you can prevent an abort in the case of certain types of errors that arise from unpredictable mathematical circumstances. Do this creating a variable named V_FitError and setting it to zero before performing a fit. If

an error occurs during the fit, it will set bit 0 of V_FitError. Certain errors will also cause other bits to be set in V_FitError:

Error	Bit Set
Any error	0
Singular matrix	1
Out of memory	2
Function returned NaN or INF	3
Fit function requested stop	4
Reentrant curve fitting	5

Reentrant curve fitting means that somehow a second curve fit started execution when there was already one running. That could happen if your user-defined fit function tried to do a curve fit, or if a button action procedure that does a fit responded too soon to another click.

There is more than one reason for a fit to stop iterating without an error. To obtain more information about the reason that a nonlinear fit stopped iterating, create a variable named V_FitQuitReason. After the fit, V_FitQuitReason is zero if the fit terminated normally, 1 if the iteration limit was reached, 2 if the user stopped the fit, or 3 if the limit of passes without decreasing chi-square was reached.

Other types of errors, such as missing waves or too few data points for the fit, are likely to be programmer errors. V_FitError does not catch those errors, but you can still prevent an abort if you wish, using the special function **AbortOnRTE** and Igor's **try-catch-endtry** construct. Here is an example function that attempts to do a curve fit to a data set that may contain nothing but NaNs:

```
Function PreventCurveFitAbort()
   Make/O test = NaN
   try
      CurveFit/N/Q line, test; AbortOnRTE
   catch
      if (V_AbortCode == -4)
         Print "Error during curve fit:"
         Variable CFerror = GetRTError(1)    // 1 to clear the error
         Print GetErrMessage(CFerror)
      endif
   endtry
End
```

If you run this function, the output is:

```
  Error during curve fit:
  You must have at least as many data points as fit parameters.
```

No error alert is presented because of the call to **GetRTError**; the error is reported to the user of this code by printing the error message to the history window using **GetRTErrMessage**.

V_FitIterStart

V_FitIterStart provides a way for a user-defined function to know when the fitting routines are about to start a new iteration. The original, obsolete purpose of this is to allow for possible efficient computation of user-defined fit functions that involve convolution. Such functions should now use all-at-once fit functions. See **All-At-Once Fitting Functions** on page III-222 for details.

S_Info

If you create a string variable in a function that calls CurveFit or Funcfit, Igor will fill it with keyword-value pairs giving information about the fit:

Keyword	Information Following Keyword
DATE	The date of the fit.
TIME	The time of day of the fit.
FUNCTION	The name of the fitting function.
AUTODESTWAVE	If you used the /D parameter flag to request an autodestination wave, this keyword gives the name of the wave.
YDATA	The name of the Y data wave.
XDATA	A comma-separated list of X data waves, or "_calculated_" if there were no X waves. In most cases there is just one X wave.

Use **StringByKey** to get the information from the string. You should set keySepStr to "=" and listSepStr to ";".

Errors in Variables: Orthogonal Distance Regression

When you fit a model to data, it is usually assumed that all errors are in the dependent variable, and that independent variables are known perfectly (that is, X is set perfectly and Y is measured with error). This assumption is often not far from true, and as long as the errors in the dependent variable are much larger than those for the independent variable, it will not usually cause much difference to the curve fit.

When the errors are normally distributed with zero mean and constant variance, and the model is exact, then the standard least-squares fit gives the maximum-likelihood solution. This is the technique described earlier (see **Overview of Curve Fitting** on page III-157).

In some cases, however, the errors in both dependent and independent variables may be comparable. This situation has a variety of names including errors in variables, measurement error models or random regressor models. An example of a model that can result in similar errors in dependent and independent variables is fitting the track of an object along a surface; the variables involved would be measurements of cartesian coordinates of the object's location at various instants in time. Presumably the measurement errors would be similar because both involve spatial measurement.

Fitting such data using standard or ordinary least squares can lead to bias in the solution. To solve this problem, we offer Orthogonal Distance Regression (ODR). Rather than minimizing the sum of squared errors in the dependent variable, ODR minimizes the orthogonal distance from the data to the fitted curve by adjusting both the model coefficients and an adjustment to the values of the independent variable. This is also sometimes called "total least squares".

For ODR curve fitting, Igor Pro uses the freely available ODRPACK95. The CurveFit, FuncFit, and FuncFitMD operations can all do ODR fitting using the /ODR flag (see the documentation for the **CurveFit** operation on page V-85 for details on the /ODR flag, and the **Curve Fitting References** on page III-231 for information about the ODRPACK95 implementation of ODR fitting).

Weighting Waves for ODR Fitting

Just as with ordinary least-squares fitting, you can provide a weighting wave to indicate the expected magnitude of errors for ODR fitting. But in ODR fitting, there are errors in both the dependent and independent variables, so ODR fits accept weighting waves for both. You use the /XW flag to specify weighting waves for the independent variable.

If you do not supply a weighting wave, it is assumed the errors have a variance of 1.0. This may be acceptable if the errors in the dependent and independent variables truly have similar magnitudes. But if the dependent and independent variables are of very different magnitudes, the chances are good that the errors are also of very different magnitudes, and weighting is essential for a proper fit. Unlike the case for ordinary least squares, where there is only a single weighting wave, ODR fitting depends on both the magnitude of the weights as well as the relative magnitudes of the X and Y weights.

ODR Initial Guesses

An ordinary least squares fit that is linear in the coefficients can be solved directly. No initial guess is required. The built-in line, poly, and poly2D curve fit functions are linear in the coefficients and do not require initial guesses.

An ordinary least-squares fit to a function that is nonlinear in the coefficients is an iterative process that requires a starting point. That means that you must provide an initial guess for the fit coefficients. The accuracy required of your initial guess depends greatly on the function you are fitting and the quality of your data. The built-in fit functions also attempt to calculate a good set of initial guesses, but for user-defined fits you must supply your own initial guesses.

An ODR fit introduces a nonlinearity into the fitting equations that requires iterative fitting and initial guesses even for fit functions that have linear fit coefficients. In the case of line, poly, and poly2D fit functions, ODR fitting uses an ordinary least squares fit to get an initial guess. For nonlinear built-in fit functions, the same initial guesses are used regardless of fitting method.

Because the independent variable is adjusted during the fit, an initial guess is also required for the adjustments to the independent variable. The initial guess is transmitted via one or more waves (one for each independent variable) specified with the /XR flag. The X residual wave is also an output- see the **ODR Fit Results** on page III-207.

In the absence of the /XR flag, initial guesses for the adjustments to the independent variable values are set to zero. This is usually appropriate; in areas where the fitting function is largely vertical, you may need nonzero guesses to fit successfully. One example of such a situation would be the region near a singularity.

Holding Independent Variable Adjustments

In some cases you may have reason to believe that you know some input values of the independent variables are exact (or nearly so) and should not be adjusted. To specify which values should not be adjusted, you supply X hold waves, one for each independent variable, via the /XHLD flag. These waves should be filled with zeroes corresponding to values that should be adjusted, or ones for values that should be held.

This is similar to the /H flag to hold fit coefficients at a set value during fitting. However, in the case of ODR fitting and the independent variable values, holds are specified by a wave instead of a string of ones and zeroes. This was done because of the potential for huge numbers of ones and zeroes being required. To save memory, you can use a byte wave for the holds. In the Make Waves dialog, you select Byte 8 Bit from the Type menu. Use the /B flag with the **Make** operation on page V-366.

ODR Fit Results

An ordinary least-squares fit adjusts the fit coefficients and calculates model values for the dependent variable. You can optionally have the fit calculate the residuals — the differences between the model and the dependent variable data.

ODR fitting adjusts both the fit coefficients and the independent variable values when seeking the least orthogonal distance fit. In addition to the residuals in the dependent variable, it can calculate and return to you a wave or waves containing the residuals in the independent variables, as well as a wave containing the adjusted values of the independent variable.

Residuals in the independent variable are returned via waves specified by the /XR flag. Note that the contents of these waves are inputs for initial guesses at the adjustments to the independent variables, so you must be careful — in most cases you will want to set the waves to zero before fitting.

The adjusted independent variable values are placed into waves you specify via the /XD flag.

Note that if you ask for an auto-destination wave (/D flag; see**The Destination Wave** on page III-176) the result is a wave containing model values at a set of evenly-spaced values of the independent variables. This wave will also be generated in response to the /D flag for ODR fitting.

You can also specify a specific wave to receive the model values (/D=*wave*). The values are calculated at the values of the independent variables that you supply as input to the fit. In the case of ODR fitting, to make a graph of the model, the appropriate X wave would be the output from the /XD flag, not the input X values.

Constraints and ODR Fitting

When fitting with the ordinary least-squares method (/ODR=0) you can provide a text wave containing constraint expressions that will keep the fit coefficients within bounds. These expressions can be used to apply simple bound constraints (keeping the value of a fit coefficient greater than or less than some value) or to apply bounds on linear combinations of the fit coefficients (constrain *a*+*b*>1, for instance).

When fitting using ODR (/ODR=1 or more) only simple bound constraints are supported.

Error Estimates from ODR Fitting

In a curve fit, the output includes an estimate of the errors in the fit coefficients. These estimates are computed from the linearized quadratic approximation to the chi-square surface at the solution. For a linear fit (line, poly, and poly2D fit functions) done by ordinary least squares, the chi-square surface is actually quadratic and the estimates are exact if the measurement errors are normally distributed with zero mean and constant variance. If the fitting function is nonlinear in the fit coefficients, then the error estimates are an approximation. The quality of the approximation will depend on the nature of the nonlinearity.

In an ODR fit, even when the fitting function is linear in the coefficients, the fitting equations themselves introduce a nonlinearity. Consequently, the error estimates from an ODR fit are always an approximation. See the **Curve Fitting References** on page III-231 for detailed information.

ODR Fitting Examples

A simple example: a line fit with no weighting. If you run these commands, the call to SetRandomSeed will make your “measurement error” (provided by **gnoise** function on page V-228) the same as the example shown here:

```
SetRandomSeed 0.5      // so that the "random" data will always be the same...
Make/N=10 YLineData, YLineXData
YLineXData = p+gnoise(1)      // gnoise simulates error in X values
YLineData = p+gnoise(1)       // gnoise simulates error in Y values
// make a nice graph with errors bars showing standard deviation errors
Display YLineData vs YLineXData
ModifyGraph mode=3,marker=8
ErrorBars YLineData XY,const=1,const=1
```

Now we’re ready to perform a line fit to the data. First, a standard curve fit:

```
CurveFit line, YLineData/X=YLineXData/D
```

This command results in the following history report:

```
fit_YLineData= W_coef[0]+W_coef[1]*x
W_coef={1.3711,0.78289}
V_chisq= 15.413; V_npnts= 10; V_numNaNs= 0; V_numINFs= 0;
V_startRow= 0; V_endRow= 9; V_startCol= 0; V_endCol= 0;
V_startLayer= 0; V_endLayer= 0; V_startChunk= 0; V_endChunk= 0;
V_q= 1; V_Rab= -0.41378; V_Pr= 0.889708; W_sigma={0.727,0.142}
Coefficient values ± one standard deviation
   a   = 1.3711 ± 0.727
   b   = 0.78289 ± 0.142
```

Next, we will use /ODR=2 to request orthogonal distance fitting:

```
CurveFit/ODR=2 line, YLineData/X=YLineXData/D
```

which gives this result:

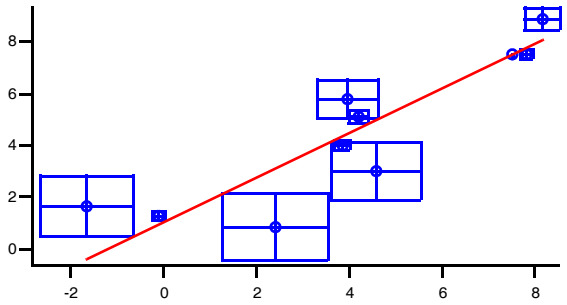
```
Fit converged properly
fit_YLineData= W_coef[0]+W_coef[1]*x
W_coef={1.0311,0.86618}
V_chisq= 9.18468; V_npnts= 10; V_numNaNs= 0; V_numINFs= 0;
V_startRow= 0; V_endRow= 9; V_startCol= 0; V_endCol= 0;
V_startLayer= 0; V_endLayer= 0; V_startChunk= 0; V_endChunk= 0;
V_q= 1; V_Rab= -0.41378; V_Pr= 0.889708; W_sigma={0.753,0.148}
Coefficient values ± one standard deviation
    a   = 1.0311 ± 0.753
    b   = 0.86618 ± 0.148
```

Add output of the X adjustments and Y residuals:

```
Duplicate/O YLineData, YLineDataXRes, YLineDataYRes
CurveFit/ODR=2 line, YLineData/X=YLineXData/D/XR=YLineDataXRes/R=YLineDataYRes
```

And a graph that uses error bars to show the residuals:

```
Display YLineData vs YLineXData
ModifyGraph mode=3,marker=8
AppendToGraph fit_YLineData
ModifyGraph rgb(YLineData)=(0,0,65535)
ErrorBars YLineData BOX,wave=(YLineDataXRes,YLineDataXRes),
wave=(YLineDataYRes,YLineDataYRes)
```



The boxes on this graph do not show error estimates, they show the residuals from the fit. That is, the differences between the data and the fit model. Because this is an ODR fit, there are residuals in both X and Y; error bars are the most convenient way to show this. Note that one corner of each box touches the model line.

In the next example, we do an exponential fit in which the Y values and errors are small compared to the X values and errors. The curve fit history report has been edited to include just the output of the solution. First, fake data and a graph:

```
SetRandomSeed 0.5   // so that the "random" data will always be the same…
Make/D/O/N=20 expYdata, expXdata
expYdata = 1e-6*exp(-p/2)+gnoise(1e-7)
expXdata = p+gnoise(1)
display expYdata vs expXdata
ModifyGraph mode=3,marker=8
```

A regular exponential fit:

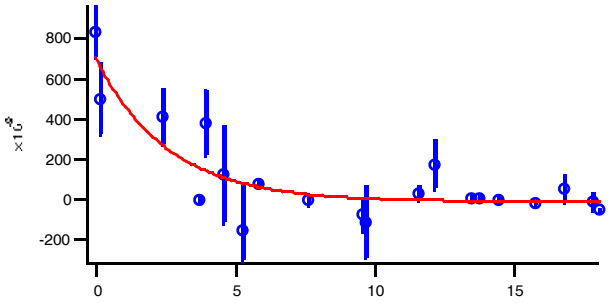
```
CurveFit exp, expYdata/X=expXdata/D

Coefficient values ± one standard deviation
    y0        =-1.0805e-08 ± 4.04e-08
    A         =7.0438e-07 ± 9.37e-08
    invTau    =0.38692 ± 0.116
```

An ODR fit with no weighting, with X and Y residuals:

```
Duplicate/O expYdata, expYdataResY, expYdataResX
expYdataResY=0
expYdataResX=0
CurveFit/ODR=2 exp, expYdata/X=expXdata/D/R=expYdataResY/XR=expYdataResX

Coefficient values ± one standard deviation
    y0     =-1.0541e-08 ± 4.03e-08
    A      =7.0443e-07 ± 9.37e-08
    invTau =0.38832 ± 0.116
```

And a graph:

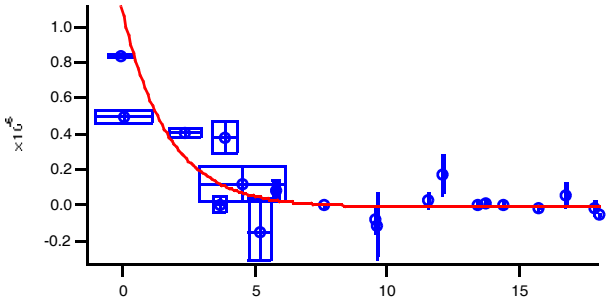
```
Display /W=(137,197,532,405) expYdata vs expXdata
AppendToGraph fit_expYdata
ModifyGraph mode(expYdata)=3
ModifyGraph marker(expYdata)=8
ModifyGraph lSize(expYdata)=2
ModifyGraph rgb(expYdata)=(0,0,65535)
ErrorBars expYdata
BOX,wave=(expYdataResX,expYdataResX),wave=(expYdataResY,expYdataResY)
```



Because the Y values are very small compared to the X values, and we didn't use weighting to reflect smaller errors in Y, the residual boxes are tall and skinny. If the vertical graph scale were the same as the horizontal scale, the boxes would be approximately square.

Now with appropriate weighting. It's easy to decide on the correct weighting since we added "measurement error" using gnoise():

```
Duplicate/O expYdata, expYdataWY
expYdataWY=1e-7
Duplicate/O expYdata, expYdataWX
expYdataWX=1
// Caution: Next command wrapped to fit on page.
CurveFit/ODR=2 exp, expYdata/X=expXdata/D/R=expYdataResY/XR =expYdataResX/W=expYdataWY
/XW=expYdataWX/I=1

Coefficient values ± one standard deviation
    y0     =-9.8498e-09 ± 3e-08
    A      =1.0859e-06 ± 5.39e-07
    invTau =0.57731 ± 0.248
```



Fitting Implicit Functions

Occasionally you may need to fit data to a model that doesn't have a form that can be expressed as y=f(x). An example would be fitting a circle or ellipse using an equation like

$$\frac{x^2}{a^2}+\frac{y^2}{b^2} = 1$$

Because this equation can't be expressed as y=f(x), you can't write a standard user-defined fitting function for it.

This problem is related to the errors in variables problem in which data has measurement errors in both the dependent and independent variables. You have two inputs, or independent variables (x and y), both with (probably) similar measurement errors. It differs from errors in variables fitting in that the function output is zero instead of being a dependent variable.

The ODRPACK95 package also supports fitting to implicit functions, which you can do with the /ODR=3 flag. You create a fitting function with multiple independent variables (in the case of the ellipse above, two for x and y). The fitting process will attempt to find x and y residuals and fit coefficients to minimize the distance from the data to the zero contour of the function.

It may be a good idea at this point to read the section about errors in variables fitting; see **Errors in Variables: Orthogonal Distance Regression** on page III-206; much of it applies also to implicit fits.

There are a few differences between regular fitting and implicit fitting. For implicit fits, there is no autodestination, and the report printed in the history does not include the wave assignment showing how to get values of the fitted model (doing that is actually not at all easy).

Because of the details of the way ODRPACK95 operates, when you do an implicit fit, the curve fit progress window does not update, and it appears that the fit takes just one iteration. That is because all the action takes place inside the call to ODRPACK95.

The fit function must be written to return zero when the function solution is found. So if you have an equation of the form $f(x_i) = 0$, you are ready to go; you simply create a fit function that implements $f(x_i)$. If it in the form $f(x_i)$ = constant, you must move the constant to the other side of the equation: $f(x_i)$-constant = 0. If it is a form like $f(x_i) = g(x_i)$, you must write your fitting function to return $f(x_i) - g(x_i)$.

Example: Fit to an Ellipse

In this example we will show how to fit an equation like the one above. In the example the center of the ellipse will be allowed to be at nonzero x_0, y_0.

First, we must define a fitting function. The function includes special comments to get mnemonic names for the fit coefficients (see **Fit Function Dialog Adds Special Comments** on page III-220). To try the example, you will need to copy this function and paste it into the Procedure window:

```
Function FitEllipse(w,x,y) : FitFunc
   Wave w
   Variable x
   Variable y

   //CurveFitDialog/
   //CurveFitDialog/ Coefficients 4
   //CurveFitDialog/ w[0] = a
   //CurveFitDialog/ w[1] = b
   //CurveFitDialog/ w[2] = x0
   //CurveFitDialog/ w[3] = y0

   return ((x-w[2])/w[0])^2 + ((y-w[3])/w[1])^2 - 1
End
```

An implicit fit seeks adjustments to the input data and fit coefficients that cause the fit function to return zero. To implement the ellipse function above, it is necessary to subtract 1.0 to account for "= 1" on the right in the equation above.

The hard part of creating an example is creating fake data that falls on an ellipse. We will use the standard parametric equations to do the job (y = a*cos(theta), x=b*sin(theta)). So far, we will not add "measurement error" to the data so that you can see the ellipse clearly on a graph:

```
Make/N=20 theta,ellipseY,ellipseX
theta = 2*pi*p/20
ellipseY = 2*cos(theta)+2
ellipseX=3*sin(theta)+1
```

A graph of the data (you could use the Windows→New Graph menu, and then the Modify Trace Appearance dialog):

```
Display ellipseY vs ellipseX
ModifyGraph mode=3,marker=8
ModifyGraph width={perUnit,72,bottom},height={perUnit,72,left}
```

The last command line sets the width and height modes of the graph so that the ellipse is shown in its true aspect ratio. Now add some "measurement error" to the data:

```
SetRandomSeed 0.5       // so that the "random" data will always be the same…
ellipseY += gnoise(.3)
ellipseX += gnoise(.3)
```

Now you can see why we didn't do that before — it's a pretty lousy ellipse!

Now, finally, do the fit. That requires making a coefficient wave and filling it with the initial guesses, and making a pair of waves to receive estimated values of X and Y at the fit:

```
Duplicate ellipseY, ellipseYFit, ellipseXFit
Make/D/O ellipseCoefs={3,2,1,2}                    // a, b, x0, y0
FuncFit/ODR=3 FitEllipse, ellipseCoefs /X={ellipseX, ellipseY}
/XD=ellipseXFit,ellipseYFit}
```

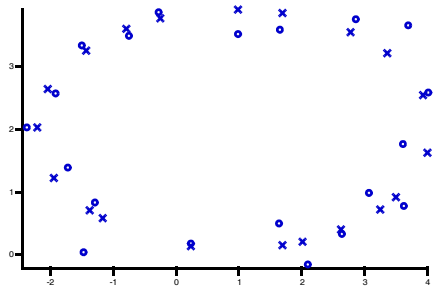
The call to the FuncFit operation has no Y wave specified (it would ordinarily go right after the coefficient wave, ellipseCoefs) because this is an implicit fit.

The results:

```
Fit converged properly
  ellipseCoefs={3.1398,1.9045,0.92088,1.9971}
  V_chisq= 1.74088; V_npnts= 20; V_numNaNs= 0; V_numINFs= 0;
  W_sigma={0.158,0.118,0.128,0.0906}
  Coefficient values ± one standard deviation
    a   =3.1398 ± 0.158
    b   =1.9045 ± 0.118
    x0  =0.92088 ± 0.128
    y0  =1.9971 ± 0.0906
```

And add the destination waves (the ones specified with the /XD flag) to the graph:

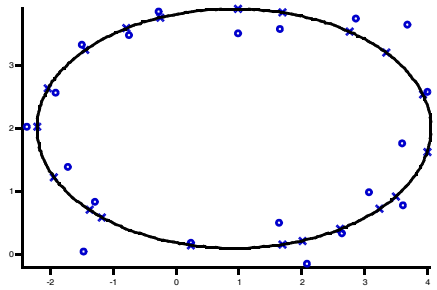
```
AppendToGraph ellipseYFit vs ellipseXFit
ModifyGraph mode=3,marker(ellipseYFit)=1
ModifyGraph rgb=(1,4,52428)
```



It is difficult to compute the model representing the solution, as it requires finding roots of the implicit function. A quick way to add a smooth model curve to a graph is to fill a matrix with values of the fit function at a range of X and Y and then add a contour plot of the matrix to your graph. Then modify the contour plot to show only the zero contour.

Here are commands to add a contour of the example function to the graph above:

```
Make/N=(100,100) elllipseContour
SetScale/I x -3,4.5,elllipseContour
SetScale/I y -.2, 5, elllipseContour
elllipseContour = FitEllipse(ellipseCoefs, x, y)
AppendMatrixContour elllipseContour
ModifyContour elllipseContour labels=0,autoLevels={*,*,0},moreLevels=0,moreLevels={0}
ModifyContour elllipseContour rgbLines=(0,0,0)
```



Fitting Sums of Fit Functions

Sometimes the appropriate model is a combination, typically a sum, of simpler models. It might be a sum of exponential decay functions, or a sum of several different peaks at various locations. Peak fitting can include a baseline function implemented as a separate user-defined fit function. With standard curve fitting, you have to write a user-defined function that implements the sum.

Instead, you can use an alternate syntax for the FuncFit operation to fit to a list of fit functions that are summed automatically. The results for each fit function are returned via separate coefficient waves, making it easy to keep the results of each term in the sum separate. The syntax is described in the **FuncFit** operation on page V-193.

The sum-of-fit-functions feature can mix any kind of fit function — built-in, or any of the variants described in **User-Defined Fitting Function: Detailed Description** on page III-217. However, even if you use only built-in fit functions you must provide initial guesses for each of the fit functions because the fit cannot discern how the various features of the data are partitioned amongst the list of fit functions.

Linear Dependency: A Major Issue

When you fit a list of fit functions you must be careful not to introduce linear dependencies between the coefficients of the list of functions. The most likely such dependency will arise from a constant Y offset used with all of the built-in fit functions, and commonly included in user functions. This Y offset accounts for any offset error that is common when making real-world measurements. For instance, the equation for the built-in exp function is:

$$y_0 + A\exp(-Bx)$$

If you sum two terms, you get two y_0's (primes used for the second copy of the function):

$$y_0 + A\exp(-Bx) + y_0' + A'\exp(-B'x)$$

As long as *B* and *B'* are distinct, the two exponential terms will not be a problem. But y_0 and y'_0 are linearly dependent — they cannot be distinguished mathematically. If you increase one and decrease the other the same amount, the result is exactly the same. This is sure to cause a singular matrix error when you try to do the fit.

The solution is to hold one of the y_0 coefficients. Because each fit function in the list has its own coefficient wave, and you can specify a hold string for each function, this is easy (see **Example: Summed Exponentials** on page III-214).

There are many ways to introduce linear dependence. They are not always so easy to spot!

Constraints Applied to Sums of Fit Functions

The sums of fit functions feature does not include a keyword for specifying constraints on a function-by-function basis. But you can use the normal constraint specifications - you just have to translate the the constraint expressions to take account of the list of functions. (If you are interested in using constraints, but don't know about the syntax for constraints, see **Fitting with Constraints** on page III-197.)

Constraint expressions use K*n* to designate fit coefficients where *n* is simply the sequential number of the fit coefficient within the list of coefficients. n starts with zero. In order to reference fit coefficients for a fit function in a list of summed fit functions, you must account for all the fit coefficients in all the fit functions in the list before the fit function you are trying to constrain. For example, if you have this:

```
FuncFit {{exp,expTerm1},{exp,expTerm2,hold="1"},{exp,expTerm3,hold="1"}} ...
```

in order to constrain coefficients of the second exponential term, you must know first that the built-in exp fit function has three fit coefficients. Here is the equation of the exp fit function:

```
f(x) = y0 + A*exp(-x*invTau)
```

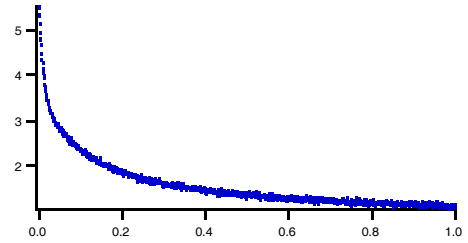
The vertical offset (y0) of the first summed exp term is represented in a constraint expression as "K0", the amplitude as "K1", and invTau as "K2". Constraint expressions for the second exp term continue the count starting with "K3" for y0 (but see the section **Linear Dependency: A Major Issue** on page III-213 and note that you are not allowed to hold and constrain a fit coefficient simultaneously), and use "K4" for the amplitude and "K5" for invTau.

Example: Summed Exponentials

This example fits a sum of three exponentials using the built-in exp fit function. For real work, we recommend the exp_XOffset function; it handles data that's not at X=0 better, and the decay constant fit coefficient is actually the decay constant. The exp fit function gives you the inverse of the decay constant.

First, make some fake data and graph it:

```
Make/D/N=1000 expSumData
SetScale/I x 0,1,expSumData
expSumData = 1 + exp(-x/0.5) + 1.5*exp(-x/.1) + 2*exp(-x/.01)+gnoise(.03)
Display expSumData
ModifyGraph mode=2,rgb=(1,4,52428)
```



The fake data was purposely made with a Y offset of 1.0 in order to illustrate how to handle the vertical offset terms in the fit function.

Next, we need to make a coefficient wave for each of the fit functions. In spite of the linear dependence between the vertical offset in each copy of the function, you must include all the fit coefficients in the coefficient waves. Otherwise when the function is evaluated the fit won't have all the needed information.

```
Make/D/O expTerm1 = {1, 1, 2}
Make/D/O expTerm2 = {0, 1.5, 10}
Make/D/O expTerm3 = {0, 2, 100}
```

Each of these lines makes one coefficient wave with three elements. The first element is y_0, the second is amplitude, and the third is the inverse of the decay constant. Since the data is fake, we have a pretty good idea of the initial guesses!

To reflect the baseline offset of 1.0, the y_0 coefficient for *only* the first exponential coefficient wave was set to 1. If y_0 were set to 1.0 for all three, the offset would be 3.0.

Now we can do the fit. A FuncFit command with a list of fit functions and coefficient waves can be pretty long:

```
FuncFit {{exp, expTerm1},{exp, expTerm2, hold="1"},{exp, expTerm3, hold="1"}}
expSumData/D
```

The entire list of functions is enclosed in braces and each fit function specification in the list is enclosed in braces as well. At a minimum you must provide a fit function and a coefficient wave for each function in the list, as was done here for the first exponential term.

The specification for each function can also contain various keywords; the second and third terms here contain the *hold* keyword in order to include a hold string. The string used will cause the fit to hold the y_0 coefficient for the second and third terms at zero. That prevents problems caused by linear dependence between the three fit functions. Since the coefficient waves for the second and third terms set their respective y_0 to zero, this puts all the vertical offset into the one coefficient for the first term.

In this example the hold strings are literal quoted strings. They can be any string expression but see below for restrictions if you use a function list contained in a string:

```
String holdStr = "1"
FuncFit {{exp,expTerm1},{exp,expTerm2,hold=holdStr},
            {exp,expTerm3,hold="1"}} expSumData/D     // All on one line
```

The history report for a sum of fit functions is enhanced to show all the functions:

```
Fit converged properly
fit_expSumData= Sum of Functions(,x)
expTerm1={1.027,1.045,2.2288}
expTerm2={0,1.4446,10.416}
expTerm3={0,2.0156,102.28}
V_chisq= 0.864844; V_npnts= 1000; V_numNaNs= 0; V_numINFs= 0;
V_startRow= 0; V_endRow= 999; V_startCol= 0; V_endCol= 0;
V_startLayer= 0; V_endLayer= 0; V_startChunk= 0; V_endChunk= 0;
W_sigma={0.0184,0.0484,0.185,0,0.052,0.495,0,0.0239,2.34}
For function 1: exp:
Coefficient values ± one standard deviation
   y0       = 1.027 ± 0.0184
   A        = 1.045 ± 0.0484
   invTau   = 2.2288 ± 0.185
For function 2: exp:
Coefficient values ± one standard deviation
   y0       = 0 ± 0
   A        = 1.4446 ± 0.052
   invTau   = 10.416 ± 0.495
For function 3: exp:
Coefficient values ± one standard deviation
   y0       = 0 ± 0
   A        = 2.0156 ± 0.0239
   invTau   = 102.28 ± 2.34
```

Example: Function List in a String

The list of functions in the FuncFit command in the first example is moderately long, but it could be much longer. If you wanted to sum 20 peak functions and a baseline function, the list could easily exceed the limit of 400 characters in a command line.

Fortunately, you can build the function specification in a string variable and use the string keyword. Doing this on the command line for the first example looks like this:

```
String myFunctions="{exp, expTerm1}"
myFunctions+="{exp, expTerm2, hold=\"1\"}"
myFunctions+="{exp, expTerm3, hold=\"1\"}"
FuncFit {string = myFunctions} expSumData/D
```

These commands build the list of functions one function specification at a time. The second and third lines use the += assignment operator to add additional functions to the list. Each function specification includes its pair of braces.

Notice the treatment of the hold strings — in order to include quotation marks in a quoted string expression, you must escape the quotation marks. Otherwise the command line parser thinks that the first quote around the hold string is the closing quote.

The function list string is parsed at run-time outside the context in which FuncFit is running. Consequently, you cannot reference local variables in a user-defined function. The hold string may be either a quoted literal string, as shown here, or it can be a reference to a global variable, including the full data folder path:

```
String/G root:myDataFolder:holdStr="1"
String myFunctions="{exp, expTerm1}"
myFunctions+="{exp, expTerm2, hold=root:myDataFolder:holdStr}"
myFunctions+="{exp, expTerm3, hold=root:myDataFolder:holdStr}"
FuncFit {string = myFunctions} expSumData/D
```

Curve Fitting with Multiple Processors

If you are lucky enough to be using a computer with multiple processors, you no doubt want to take advantage of them. Because curve fitting can be a time-consuming, computation-intensive process, curve fitting will take advantage of multiple processors in two ways.

First, you can split the computations required for a curve fit into multiple threads to get the benefit of multiple processors, even on a single curve fit. This has been done for built-in fit functions and for user-defined fitting functions in the basic form (see **User-Defined Fitting Function: Detailed Description** on page III-217).

Second, the CurveFit, FuncFit, and FuncFitMD operations are "thread safe". That means that you can do more than one curve fit simultaneously, with computations being done on multiple processors simultaneously. This requires that you write a function using the threaded programming techniques (see **ThreadSafe Functions and Multitasking** on page IV-288). This is not a task for a beginning programmer. See the example experiment "MultipleFitsInThreads.pxp" in the File→Example Experiments→Curve Fitting menu.

Multithreaded Curve Fits

To take advantage of multiple processors for curve fit computations, use the /NTHR flag. This flag specifies how many threads to use in computing the model values and derivatives of the fit function. Setting /NTHR=0 uses the "best" number of threads; /NTHR=*n* uses exactly *n* threads. Usually *n* should be equal to the number of processors.

In the Curve Fit dialog you will find a checkbox, Use Multiple Processors. It is selected by default if your computer appears to have multiple processors, and not selected by default if your computer has just one processor. Selecting the checkbox adds /NTHR=0 to the generated command, using the best number of threads. Deselecting it sets /NTHR=1 to use just one processor no matter how many you have.

There is significant overhead when running multiple concurrent threads. The benefit you get from multiple threads will depend on the computation time spent actually computing the model and derivatives. This means that large problems (many data points) will benefit more than small problems. Fit functions that take longer to compute will benefit more than simple fit functions, and user-defined fits benefit more than built-in fits because built-in fits run more efficient code.

Because of the efficiency of the built-in fit function computation, we find that multiple processors do not help fits to built-in functions. In fact, the overhead may actually hurt performance for smaller problems. Consequently, the default for /NTHR=0 for built-in fits is to use just one thread. On a multiprocessor computer this means that only one processor will be used.

User-defined fit functions, because they are implemented in Igor Pro's programming language, run more slowly. Our testing indicates that user-defined fit functions may benefit from running on more than one processor, especially for problems having more than a few hundred data points. Very complex user-defined functions that execute more slowly will get more benefit for smaller problems. When you use /NTHR=0, the fit uses as many threads as your machine has processors.

So far, only built-in fits and basic user-defined fit functions are multithreaded.

Multiple Curve Fits Simultaneously

Another way to take advantage of multiple processors for curve fitting is to run the curve fit from threaded Igor code. If you do this, you can use multiple threads to run multiple curve fits simultaneously, or you can run a single curve fit as a background process.

Potentially, programming multiple curve fits in a threaded user-defined function could result in better performance on a multiprocessor computer because the fits could be running simultaneously. Because all of the curve fit computations will be simultaneous, unlike the situation with a single fit using /NTHR=*n*, the performance gain could be greater than for single fits with multiple processors. Only testing can tell you for sure.

Note that any user-defined fit function used with curve fitting from threaded procedures must be a threadsafe function (see **ThreadSafe Functions** on page IV-83 for details).

For an example of multiple curve fits in threaded user-defined functions, see the example experiment "MultipleFitsInThreads.pxp". You will find it under the File→Example Experiments→Curve Fitting menu.

Constraints and ThreadSafe Functions

The usual way to specify constraints to a curve fit is via expressions in a text wave (see **Fitting with Constraints** on page III-197). As part of the process of parsing these expressions and getting ready to use them in a curve fit involves evaluating part of the expressions. That, in turn, requires sending them to Igor's command-line interpreter, in a process very similar to the way the **Execute** operation works. Unfortunately, this is not threadsafe.

Instead, you must use the method described in **Constraint Matrix and Vector** on page III-200. Unfortunately, it is hard to understand, inconvenient to set up, and easy to make mistakes. The best way to do it is to set up your constraint expressions using the normal text wave method (see **Constraint Expressions** on page III-199) and use the /C flag with a trial fit. Igor will generate the matrix and vector required.

In most cases, the basic expressions will not change from one fit to another, just the limits of the constraints will change. If that is the case, you can use the matrix provided by Igor, and alter the numbers in the vector to change the constraint limits.

User-Defined Fitting Function: Detailed Description

When you use the New Fit Function dialog to create a user-defined function, the dialog uses the information you enter to create code for your function in the Procedure window. Using the New Fit Function dialog is the easiest way to create a user-defined fitting function, but it is possible also to write the function directly in the Procedure window.

Certain kinds of complexities will *require* that you write the function yourself. It may be that the easiest way to create such a function is to create a skeleton of the function using the dialog, and then modify it by editing in the procedure window.

This section describes the format of user-defined fitting functions so that you can understand the output of the New Fit Function dialog, and so that you can write one yourself.

You can use a variety of formats for user-defined fit functions tailored to different situations, and involving varying degrees of complexity to write and use. The following section describes the simplest format, which is also the format created by the New Fit Function dialog.

Discussion of User-Defined Fitting Function Formats

You can use three formats for user-defined fitting functions: the Basic format discussed above, the All-At-Once format, and Structure Fit Functions, which use a structure as the only input parameter. Additionally, Structure Fit Functions come in basic and all-at-once variants.

Each of these formats address particular situations. The Basic format was the original format; it returns just one model value at a time. The All-At-Once format addresses problems in which the operations involved naturally calculate all the model values at once. Such problems are ones that involve operations like convo-

lution, integration, or FFT. Structure Fit Functions use a structure as the only function parameter, allowing arbitrary information to be transmitted to the function during fitting. This makes it very flexible, but also makes it necessary that FuncFit be called from a user-defined function.

Basic Fit Function	All-At-Once Function	Structure Function
Can be selected, created, and edited within the Curve Fitting dialog.	Can be selected, but *not* created or edited, within the Curve Fitting dialog.	Cannot be used from the Curve Fitting dialog.
With appropriate comments, mnemonic coefficient names.	No mnemonic coefficient names.	Must be used with FuncFit called from a user-defined function.
Straight-forward programming: one X value, one return value.	Programming requires a good understanding of wave assignment; there are some issues that can be difficult to avoid.	Hardest to program: requires both an understanding of structures and writing a driver function that calls FuncFit.
Not an efficient way to write a fit function that uses convolution, integration, FFT, or any operation that uses all the data values in a single operation.	Most efficient for problems involving operations like convolution, integration, or FFT. Often much faster than the Basic format, even for problems that don't require it.	Very flexible: any arbitrary information can be transmitted to the fit function. More information about the fit progress transmitted via the structure.
See **Format of a Basic Fitting Function** on page III-218.	See **All-At-Once Fitting Functions** on page III-222.	See **Structure Fit Functions** on page III-226.

Format of a Basic Fitting Function

A basic user-defined fitting function has the following form:

```
Function F(w, x) : FitFunc
   WAVE w; Variable x

   <body of function>
   <return statement>
End
```

You can choose a more descriptive name for your function.

The function must have exactly two parameters. The first parameter is the coefficients wave, conventionally called w. The second parameter is the independent variable, conventionally called x. If your function has this form, it will be recognized as a curve fitting function and will allow you to use it with the FuncFit operation.

The FitFunc keyword marks the function as being intended for curve fitting. Functions with the FitFunc keyword that have the correct format are included in the Function menu in the Curve Fitting dialog.

The FitFunc keyword is not required. The FuncFit operation will allow any function that has a wave and a variable as the parameters. In the Curve Fitting dialog you can choose Show Old-Style Functions from the Function menu to display a function that lacks the FitFunc keyword, but you may also see functions that just happen to match the correct format but aren't fitting functions.

Note that the function does not know anything about curve fitting. All it knows is how to compute a return value from its input parameters. The function is called during a curve fit when it needs the value for a certain X and a certain set of fit coefficients.

Here is an example of a user-defined function to fit a log function. This might be useful since log is not one of the functions provided as a built-in fit function.

```
Function LogFit(w,x) : FitFunc
   WAVE w
   Variable x
```

```
   return w[0]+w[1]*log(x)
End
```

In this example, two fit coefficients are used. Note that the first is in the zero element of the coefficient wave. You cannot leave out an index of the coefficient wave- an unused element of the wave will result in a singular matrix error.

Intermediate Results for Very Long Expressions

The body of the function is usually fairly simple but can be arbitrarily complex. If necessary, you can use local variables to build up the result, piece-by-piece. You can also call other functions or operations and use loops and conditionals.

If your function has many terms, you might find it convenient to use a local variable to store intermediate results rather than trying to put the entire function on one line. For example, instead of:

```
return w[0] + w[1]*x + w[2]*x^2
```

you could write:

```
Variable val          // local variable to accumulate result value
val = w[0]
val += w[1]*x
val += w[2]*x^2
return val
```

Conditionals

Flow control statements including `if` statements are allowed in fit functions. You could use this to fit a piece-wise function, or to control the return value in the case of a singularity in the function.

Here is an example of a function that fits two lines to different sections of the data. It uses one of the parameters to decide where to switch from one line to the other:

```
Function PieceWiseLineFit(w,x) : FitFunc
   WAVE w
   Variable x

   variable result
   if (x < w[4])
      result = w[0]+w[1]*x
   else
      result = w[2]+w[3]*x
   endif
   return result
End
```

This function can be entered into the New Fit Function dialog. Here is what the dialog looked like when we created the function above:

Note: there's a line that you can't see at the top of the code.

Fit Function Dialog Adds Special Comments

In the example above of a piece-wise linear fit function, the New Fit Function dialog uses coefficient names instead of indexing a coefficient wave, but there isn't any way to name coefficients in a fit function. The New Fit Function dialog adds special comments to a fit function that contain extra information. For instance, the PieceWiseLineFit function as created by the dialog looks like this:

```
Function PieceWiseLineFit(w,x) : FitFunc
    WAVE w
    Variable x

    //CurveFitDialog/ These comments were created by the Curve Fitting dialog. Alteri
    //CurveFitDialog/ make the function less convenient to work with in the Curve Fit
    //CurveFitDialog/ Equation:
    //CurveFitDialog/ variable result
    //CurveFitDialog/ if (x < breakX)
    //CurveFitDialog/  result = a1+b1*x
    //CurveFitDialog/ else
    //CurveFitDialog/  result = a2+b2*x
    //CurveFitDialog/ endif
    //CurveFitDialog/ f(x) = result
    //CurveFitDialog/ End of Equation
    //CurveFitDialog/ Independent Variables 1
    //CurveFitDialog/ x
    //CurveFitDialog/ Coefficients 5
    //CurveFitDialog/ w[0] = a1
    //CurveFitDialog/ w[1] = b1
    //CurveFitDialog/ w[2] = a2
    //CurveFitDialog/ w[3] = b2
    //CurveFitDialog/ w[4] = breakX

    variable result
    if (x < w[4])
     result = w[0]+w[1]*x
    else
     result = w[2]+w[3]*x
    endif
    return result
End
```

Special comments give the Curve Fitting dialog extra information about the fit function.

The function code as it appears in the text window of the New Fit Function dialog.

Independent variable name (or names, for a multivariate function).

Coefficient names.

This prefix in the comment identifies the comment as belonging to the curve fitting dialog.

The actual function code.

If you click the Edit Fit Function button, the function code is analyzed to determine the number of coefficients. If the comments that name the coefficients are present, the dialog uses those names. If they are not present, the coefficient wave name is used with the index number appended to it as the coefficient name.

Having mnemonic names for the fit coefficients is very helpful when you look at the curve fit report in the history window. The minimum set of comments required to have names appear in the dialog and in the history is a lead-in comment line, plus the Coefficients comment lines. For instance, the following version of the function above will allow the Curve Fitting dialog and history report to use coefficient names:

```
Function PieceWiseLineFit(w,x) : FitFunc
   WAVE w
   Variable x

   //CurveFitDialog/
   //CurveFitDialog/ Coefficients 5
   //CurveFitDialog/ w[0] = a1
   //CurveFitDialog/ w[1] = b1
   //CurveFitDialog/ w[2] = a2
   //CurveFitDialog/ w[3] = b2
   //CurveFitDialog/ w[4] = breakX

   variable result
   if (x < w[4])
    result = w[0]+w[1]*x
   else
    result = w[2]+w[3]*x
   endif
   return result
End
```

The blank comment before the line with the number of coefficients on it is required- the parser that looks at these comments needs one lead-in line to throw away. That line can contain anything as long as it includes the lead-in "`//CurveFitDialog/`".

Functions that the Fit Function Dialog Doesn't Handle Well

In the example functions it is quite clear by looking at the function code how many fit coefficients a function requires, because the coefficient wave is indexed with a literal number. The number of coefficients is simply one more than the largest index used in the function.

Occasionally a fit function uses constructions other than a literal number for indexing the coefficient wave. This will make it impossible for the Curve Fitting dialog to figure out how many coefficients are required. In this case, the Coefficients tab can't be constructed until you specify how many coefficients are needed. You do this by choosing a coefficient wave having the right number of points from the Coefficient Wave menu.

You cannot edit such a function by clicking the Edit Fit Function button. You must write and edit the function in the Procedure window.

Here is an example function that can fit an arbitrary number of Gaussian peaks. It uses the length of the coefficient wave to determine how many peaks are to be fit. Consequently, it uses a variable (`cfi`) rather than a literal number to access the coefficients:

```
Function FitManyGaussian(w, x) : FitFunc
   WAVE w
   Variable x

   Variable returnValue = w[0]
   Variable i
   Variable numPeaks = floor((numpnts(w)-1)/3)
   Variable cfi

   for (i = 0; i < numPeaks; i += 1)
      cfi = 3*i+1
      returnValue += w[cfi]*exp(-((x-w[cfi+1])/w[cfi+2])^2)
   endfor
   return returnValue
End
```

The first coefficient is a baseline offset.

Each peak takes three coefficients: amplitude, x position and width.

Calculate index of amplitude for this peak.

Expression of a single Gaussian peak.

Loop over the peaks, calculating them one at a time.

Each peak is added to the result.

Format of a Multivariate Fitting Function

A multivariate fitting function has the same form as a univariate function, but has more than one independent variable:

```
Function F(w, x1, x2, ...) : FitFunc
   WAVE w;
   Variable x1
   Variable x2
   Variable ...

   <body of function>
   <return statement>
End
```

A function to fit a planar trend to a data set could look like this:

```
Function Plane(w, x1, x2) : FitFunc
   WAVE w
   Variable x1, x2

   return w[0] + w[1]*x1 + w[2]*x2
End
```

There is no limit on the number of independent variables, with the exception that the entire Function declaration line must fit within a single command line of 400 characters. Thus, if you use a three-character func-

tion name, a one-character name for the coefficients wave parameter, and two-character names for the independent variable parameters, you could write a fitting function with 128 independent variables.

The New Fit Function dialog will add the same comments to a multivariate fit function as it does to a basic fit function. The `Plane()` function above might look like this (we have truncated the first two special comment lines to make them fit):

```
Function Plane(w,x1,x2) : FitFunc
   WAVE w
   Variable x1
   Variable x2

   //CurveFitDialog/ These comments were created by the Curve...
   //CurveFitDialog/ make the function less convenient to work...
   //CurveFitDialog/ Equation:
   //CurveFitDialog/ f(x1,x2) = A + B*x1 + C*x2
   //CurveFitDialog/ End of Equation
   //CurveFitDialog/ Independent Variables 2
   //CurveFitDialog/ x1
   //CurveFitDialog/ x2
   //CurveFitDialog/ Coefficients 3
   //CurveFitDialog/ w[0] = A
   //CurveFitDialog/ w[1] = B
   //CurveFitDialog/ w[2] = C

   return w[0] + w[1]*x1 + w[2]*x2
End
```

All-At-Once Fitting Functions

The scheme of calculating one Y value at a time doesn't work well for some fitting functions. This is true of functions that involve a convolution such as might arise if you are trying to fit a theoretical signal convolved with an instrument response. Fitting to a solution to a differential equation might be another example.

For this case, you can create an "all at once" fit function. Such a function provides you with an X and Y wave. The X wave is input to the function; it contains all the X values for which your function must calculate Y values. The Y wave is for output — you put all your Y values into the Y wave.

Because an all-at-once function is called only once for a given set of fit coefficients, it will be called many fewer times than a basic fit function. Because of the saving in function-call overhead, all-at-once functions can be faster even for problems that don't require an all-at-once function.

Here is the format of an all-at-once fit function:

```
Function myFitFunc(pw, yw, xw) : FitFunc
   WAVE pw, yw, xw

   yw = <expression involving pw and xw>
End
```

Note that there is no return statement because the function result is put into the wave yw. Even if you include a return statement the return value is ignored during a curve fit.

The X wave contains all the X values for your fit, whether you provided an X wave to the curve fit or not. If you did not provide an X wave, xw simply contains equally-spaced values derived from your Y wave's X scaling.

There are some restrictions on all-at-once fitting functions:

1) You can't create or edit an all-at-once function using the Curve Fitting dialog. You must create it by editing in the Procedure window. All-at-once functions are, however, listed in the Function menu in the Curve Fitting dialog.
2) There is no such thing as an "old-style" all-at-once function. It must have the FitFunc keyword.
3) You don't get mnemonic coefficient names.

Here is an example that fits an exponential decay using an all-at-once function. This example is silly — there is no reason to make this an all-at-once function. It is simply an example showing how a real function works without the computational complexities. Here it is, as an all-at-once function:

```
Function allatonce(pw, yw, xw) : FitFunc
   WAVE pw, yw, xw

   // a wave assignment does the work
   yw = pw[0] + pw[1]*exp(-xw/pw[2])
End
```

This is the same function written as a standard user fitting function:

```
Function notallatonce(pw, x) : FitFunc
   WAVE pw
   Variable x

   return pw[0] + pw[1]*exp(-x/pw[2])
End
```

In the all-at-once function, the argument of exp() includes xw, a wave. The basic format uses the input parameter x, which is a variable containing a single value. The use of xw in the all-at-once version of the function uses the implied point number feature of wave assignments (see **Waveform Arithmetic and Assignments** on page II-93).

There are a couple of things to watch out for when creating an all-at-once fitting function:

1. You must not change the yw wave. That is, don't use Make or Redimension operations on yw. This will get you into trouble:
   ```
   Function allatonce(pw, yw, xw) : FitFunc
      WAVE pw, yw, xw

      Redimension/N=2000 yw                  // BAD!
      yw = pw[0] + pw[1]*exp(-xw/pw[2])
   End
   ```
2. You may not get the same number of points in yw and xw as you have in the waves you provide as the input data. If you fit to a restricted range, if there are NaNs in the input data, or if you use a mask wave to select a subset of the input data, you will get a wave with a reduced number of points. Your fitting function must be written to handle that situation or you must not use those features.
3. It is tricky (but not impossible) to write an all-at-once fitting function that works correctly with the auto-destination feature (that is, _auto_ in the Destination menu, or /D by itself in a FuncFit command).
4. The xw and yw waves are *not* your data waves. They will be destroyed when fitting is finished.

The example above uses xw as an argument to the exp function, and it uses the special features of a wave assignment statement to satisfy point 2.

The next example fits a convolution of a Gaussian peak with an exponential decay. It fits a baseline offset, amplitude and width of the Gaussian peak and the decay constant for the exponential. This might model an instrument with an exponentially decaying impulse response to recover the width of an impulsive signal.

```
Function convfunc(pw, yw, xw) : FitFunc
   WAVE pw, yw, xw

   // pw[0] = gaussian baseline offset
   // pw[1] = gaussian amplitude
   // pw[2] = gaussian position
   // pw[3] = gaussian width
   // pw[4] = exponential decay constant of instrument response

   // Make a wave to contain an exponential with decay constant pw[4]. The
   // wave needs enough points to allow any reasonable decay constant to
   // get really close to zero. The scaling is made symmetric about zero to
   // avoid an X offset from Convolve/A
```

```
   Variable dT = deltax(yw)
   make/D/O/N=201 expwave           // long enough to allow decay to zero
   setscale/P x -dT*100,dT,expwave

   // fill expwave with exponential decay
   expwave = (x>=0)*pw[4]*dT*exp(-pw[4]*x)

   // Normalize exponential so that convolution doesn't change
   // the amplitude of the result
   Variable sumexp
   sumexp = sum(expwave, -inf,inf)
   expwave /= sumexp

   // Put a Gaussian peak into the output wave
   yw = pw[0]+pw[1]*exp(-((x-pw[2])/pw[3])^2)
   // and convolve with the exponential; NOTE /A
   convolve/A expwave, yw
End
```

Some things to be aware of with regard to this function:

1) A wave is created inside the function to store the exponential decay. Making a wave can be a time-consuming operation; it is less convenient for the user of the function, but can save computation time if you make a suitable wave ahead of time and then simply reference it inside the function.
2) The wave containing the exponential decay is passed to the convolve operation. The use of the /A flag prevents the convolve operation from changing the length of the output wave yw. You may wish to read the section on **Convolution** on page III-249.
3) The output wave yw is used as a parameter to the convolve operation. Because the convolve operation assumes that the data are evenly-space, this use of yw means that the function *does not satisfy* points 2) or 3) above. If you use input data to the fit that has missing points or unevenly-spaced X values, this function *will fail*.

You may also find the section **Waveform Arithmetic and Assignments** on page II-93 helpful.

Here is a much more complicated version of the fitting function that solves these problems, and also is more robust in terms of accuracy. The comments in the code explain how the function works.

Note that this function makes at least two different waves using the Make operation, and that we have used Make/D to make the waves double-precision. This can be crucial.

```
Function convfunc(pw, yw, xw) : FitFunc
   WAVE pw, yw, xw

   // pw[0] = gaussian baseline offset
   // pw[1] = gaussian amplitude
   // pw[2] = gaussian position
   // pw[3] = gaussian width
   // pw[4] = exponential decay constant of instrument response (this
   //       parameter is actually the inverse of the time constant, in
   //       order to be just like Igor's built-in exp fit function, which
   //       was written in the days when a floating-point divide took much
   //       longer than a multiply).

   // Make a wave to contain an exponential with decay constant pw[4]. The
   // wave needs enough points to allow any reasonable decay constant to
   // get really close to zero. The scaling is made symmetric about zero to
   // avoid an X offset from Convolve/A

   // resolutionFactor sets the degree to which the exponential will be
   // over-sampled with regard to the problems parameters. Increasing this
   // number increases the number of time constants included in the calculation.
   // It also decreases the point spacing relative to the problem's time
   // constants. Increasing will also increase the time required to compute.
```

```
   Variable resolutionFactor = 10

   // dt contains information on important time constants. We wish to set
   // the point spacing for model calculations much smaller than exponential
   // time constant or gaussian width.

   Variable dT = min(1/(resolutionFactor*pw[4]), pw[3]/resolutionFactor)

   // Calculate suitable number points for the exponential. Length of
   // exponential wave is 10 time constants; doubled so exponential can start
   // in the middle; +1 to make it odd so exponential starts at t=0, and t=0 is
   // exactly the middle point. That is better for the convolution.

   Variable nExpWavePnts = round(resolutionFactor/(pw[4]*dT))*2 + 1

   Make/D/O/N=(nExpWavePnts) expwave

   // In this version of the function, we make a y output wave ourselves, so
   // that we can control the resolution and accuracy of the calculation. It
   // also will allow us to use a wave assignment later to solve the problem
   // of variable X spacing or missing points.

   Variable nYPnts = max(resolutionFactor*numpnts(yw), nExpWavePnts)
   Make/D/O/N=(nYPnts) yWave

   // This wave scaling is set such that the exponential will
   // start at the middle of the wave
   setscale/P x -dT*(nExpWavePnts/2),dT,expwave

   // Set the wave scaling of the intermediate output wave to have the resolution
   // calculated above, and to start at the first X value.
   setscale/P x xw[0],dT, yWave

   // fill expwave with exponential decay
   expwave = (x>=0)*pw[4]*dT*exp(-pw[4]*x)

   // Normalize exponential so that convolution doesn't change
   // the amplitude of the result
   Variable sumexp
   sumexp = sum(expwave, -inf,inf)
   expwave /= sumexp

   // Put a Gaussian peak into the intermediate output wave. We use our own wave
   // (yWave) because the convolution requires a wave with even spacing in X,
   // whereas we may get X values input that are not evenly spaced.
   yWave = pw[0]+pw[1]*exp(-((x-pw[2])/pw[3])^2)

   // and convolve with the exponential; NOTE /A
   convolve/A expwave, yWave

   // Move appropriate values corresponding to input X data into the output Y
   // wave. We use a wave assignment involving the input X wave. This will
   // extract the appropriate values from the intermediate wave, interpolating
   // values where the intermediate wave doesn't have a value precisely at an X
   // value that is required by the input X wave. This wave assignment also
   // solves the problem with auto-destination: The function can be called with
   // an X wave that sets any X spacing, so it doesn't matter what X values
   // are required.
   yw = yWave(xw[p])
End
```

None of the computations involve the wave yw. That was done so that the computations could be done at finer resolution than the resolution of the fitted data. By making a separate wave, it is not necessary to modify yw.

The actual return values are picked out of yWave using the special X-scale-based indexing available for one-dimensional waves in a wave assignment.

Structure Fit Functions

Sometimes you may need to transmit extra information to a fitting function. This might include constants that need to be set to reflect conditions in a run, but should not be fit; or a wave containing a look-up table or a measured standard sample that is needed for comparison. Perhaps your fit function is very complex and needs some sort of book-keeping data.

If you use a basic fit function or an all-at-once function, such information must be transmitted via global variables and waves. That, in turn, requires that this global information be looked up inside the fit function. The global data requirement makes it difficult to be flexible: the fit function is tied to certain global variables and waves that must be present for the function to work. In addition to adding complexity and difficulty to management of global information, it adds a possible time-consuming operation: looking up the global information requires examining a list of waves or variables, and comparing the names to the desired name.

Structure fit functions are ideal for such problems because they take a single parameter that is a structure of your own design. The first few members of the structure must conform to certain requirements, but the rest is up to you. You can include any kind of data in the structure. An added bonus is that you can include members in the structure that identify for the fit function which fit coefficient is being perturbed when calculating numerical derivatives, that signal when the fit function is being called to fill in the auto-destination wave, and identify when a new fit iteration is starting. You can pass back a flag to have FuncFit abandon fitting.

To use a structure fit function, you must do three things:

1) Define a structure containing certain standard items at the top.
2) Write a fitting function that uses that structure as its only parameter.
3) Write a wrapper function for FuncFit that creates an instance of your structure, initializes it and invokes FuncFit with the /STRC parameter flag.

You should familiarize yourself with the use of structures before attempting to write a structure fit function (see **Structures in Functions** on page IV-78).

Structure fit functions come in basic and all-at-once variants; the difference is determined by the members at the beginning of the structure. The format for the structure for a basic structure fit function is:

```
Structure myBasicFitStruct
   Wave coefw
   Variable x
   …
EndStructure
```

The name of the structure can be anything you like that conforms to Igor's naming rules; all that is required is that the first two fields be a wave and a variable. By convention we name the wave coefw and the variable x to match the use of those members. These members of the structure are equivalent to wave and variable parameters required of a basic fit function.

You may wish to use an all-at-once structure fit function for the same reason you might use a regular all-at-once fit function. The same concerns apply to all-at-once structure fit functions; you should read and understand **All-At-Once Fitting Functions** on page III-222 before attempting to write an all-at-once structure fit function.

The structure for an all-at-once structure fit function is:

```
Structure myAllAtOnceFitStruct
   Wave coefw
   Wave yw
   Wave xw
   …
EndStructure
```

The first three members of the structure are equivalent to the pw, yw, and xw parameters required in a regular all-at-once fit function.

A simple example of fitting with a structure fit function follows. More examples are available in the File menu: select appropriate examples from Examples→Curve Fitting.

Basic Structure Fit Function Example

As an example of a basic structure fit function, we will write a fit function that fits an exponential decay using an X offset to compensate for numerical problems that can occur when the X range of an exponential function is small compared to the X values. The X offset must not be a fit coefficient because it is not mathematically distinguishable from the decay amplitude. We will write a structure that carries this constant as a custom member of a structure. This function will duplicate the built-in exp_XOffset function (see **Notes on the Built-in Fit Functions** on page III-165).

First, we create fake data by executing these commands:

```
Make/D/O/N=100 expData,expDataX
expDataX = enoise(0.5)+100.5
expData = 1.5+2*exp(-(expDataX-100)/0.2) + gnoise(.05)
Display expData vs expDataX
ModifyGraph mode=3,marker=8
```

Now the code. Copy this code and paste it into your Procedure window:

```
// The structure definition
Structure expFitStruct
   Wave coefw           // required coefficient wave
   Variable x           // required X value input
   Variable x0          // constant
EndStructure

// The fitting function
Function fitExpUsingStruct(s) : FitFunc
   Struct expFitStruct &s

   return s.coefw[0] + s.coefw[1]*exp(-(s.x-s.x0)/s.coefw[2])
End

// The driver function that calls FuncFit:
Function expStructFitDriver(pw, yw, xw, xOff)
   Wave pw     // coefficient wave- pre-load it with initial guess
   Wave yw
   Wave xw
   Variable xOff
   Variable doODR

   // An instance of the structure. We initialize the x0 constant only,
   // Igor (FuncFit) will initialize coefw and x as required.
   STRUCT expFitStruct fs
   fs.x0 = xOff   // set the value of the X offset in the structure

   FuncFit fitExpUsingStruct, pw, yw /X=xw /D /STRC=fs

   // no history report for structure fit functions. We print our own
   // simple report here:
   print pw
   Wave W_sigma
   print W_sigma
End
```

Finally, make a coefficient wave loaded with initial guesses and invoke our driver function:

```
Make/D/O expStructCoefs = {1.5, 2, .2}
expStructFitDriver(expStructCoefs, expData, expDataX, 100)
```

This is a very simple example, intended to show only the most basic aspects of fitting with a structure fit function. An advanced programmer could add a control panel user interface, plus code to automatically calculate initial guesses and provide a default value of the x0 constant.

The WMFitInfoStruct Structure

In addition to the required structure members, you can include a WMFitInfoStruct structure member immediately after the required members. The WMFitInfoStruct structure, if present, will be filled in by FuncFit with information about the progress of fitting, and includes a member allowing you to stop fitting if your fit function detects a problem.

Adding a WMFitInfoStruct member to the structure in the example above:

```
Structure expFitStruct
   Wave coefw                    // Required coefficient wave.
   Variable x                    // Required X value input.
   STRUCT WMFitInfoStruct fi     // Optional WMFitInfoStruct.
   Variable x0                   // Constant.
EndStructure
```

And the members of the WMFitInfoStruct:

WMFitInfoStruct Structure Members

Member	Description
char IterStarted	Nonzero on the first call of an iteration.
char DoingDestWave	Nonzero when called to evaluate the autodestination wave.
char StopNow	Fit function sets this to nonzero to indicate that a problem has occurred and fitting should stop.
Int32 IterNumber	Number of iterations completed.
Int32 ParamPerturbed	Index of the fit coefficient being perturbed for the calculation of numerical derivatives. Set to -1 when evaluating a solution point with no perturbed coefficients.

The IterStarted and ParamPerturbed members may be useful in some obscure cases to short-cut lengthy computations. The DoingDestWave member may be useful in an all-at-once structure fit function.

Multivariate Structure Fit Functions

To fit multivariate functions (those having more than one dimension or independent variable) you simply use an array for the X member of the structure. For instance, for a basic 2D structure fit function:

```
Structure My2DFitStruct
   Wave coefw
   Variable x[2]
   …
EndStructure
```

Or a 2D all-at-once structure fit function:

```
Structure My2DAllAtOnceFitStruct
   Wave coefw
   Wave yw
   Wave xw[2]
   …
EndStructure
```

Fitting Using Commands

A few curve fitting features are not completely supported by the Curve Fitting dialog, such as constraints involving combinations of fit coefficients, or user-defined fit functions involving more complex constructions. Also, you might sometimes want to batch-fit to a number of data sets without interacting with the dialog for each data set. These circumstances will require that you use command lines to do fits.

The easiest way to do this is to use the Curve Fitting dialog to generate command lines that do almost what you want. If you simply want to add a more complex feature, such as a more complicated constraint expression, click the To Cmd Line button, then edit the commands generated by the dialog to add the features you want.

If you are writing a user procedure that does curve fitting, you can click the To Clip button to copy the commands generated by the dialog. Then paste the commands into the Procedure window. Edit them as needed by your application.

Curve fitting is done by three operations — **CurveFit**, **FuncFit**, and **FuncFitMD**. You will find details on these operations in Chapter V-1, **Igor Reference**.

Batch Fitting

If you are doing batch fitting, you probably want to call a curve fitting operation inside a loop. In that case, you probably don't want a history report for every fit- it could possible make a very large amount of text in the history. You probably don't want the progress window during the fits- it slows down the fit and will make a flashing window as it appears and disappears. Finally, you probably don't want graphs and tables updating during the fits, as this can slow down computation considerably.

Here is an example function that will do all of this, plus it checks for an error during the fit. If the use of the $ operator is unfamiliar you will want to consult **Accessing Waves in Functions** on page IV-65**.**

```
Function FitExpToListOfWaves(theList)
   String theList

   Variable i=0
   string aWaveName = ""
   Variable V_fitOptions = 4     // suppress progress window
   Variable V_FitError = 0       // prevent abort on error
   do
      aWaveName = StringFromList(i, theList)
      WAVE/Z aWave = $aWaveName
      if (!WaveExists(aWave))
         break
      endif

      // /N suppresses screen updates during fitting
      // /Q suppresses history output during fitting
      CurveFit/N/Q exp aWave /D/R
      WAVE W_coef

      // save the coefficients
      Duplicate/O W_coef $("cf_"+aWaveName)
      // save errors
      Duplicate/O W_sigma, $("sig_"+aWaveName)
      if (V_FitError != 0)
         // Mark the results as being bad
         WAVE w = $("cf_"+aWaveName)
         w = NaN
         WAVE w = $("sig_"+aWaveName)
         w = NaN
         WAVE w = $("fit_"+aWaveName)
         w = NaN
         WAVE w = $("Res_"+aWaveName)
```

```
            w = NaN
            V_FitError = 0
        endif
        i += 1
    while(1)
End
```

Curve Fitting Examples

The Igor Pro Folder includes a number of example experiments that demonstrate the capabilities of curve fitting. These examples cover fitting with constraints, multivariate fitting, multipeak fitting, global fitting, fitting a line between cursors, and fitting to a user-defined function. All of these experiments can be found in Igor Pro Folder:Examples:Curve Fitting.

Singularities

You may occasionally run across a situation where you see a "singular matrix" error. This means that the system of equations being solved to perform the fit has no unique solution. This generally happens when the fitted curve contains degeneracies, such as if all Y values are equal.

In a fit to a user-defined function, a singular matrix results if one or more of the coefficients has no effect on the function's return value. Your coefficients wave must have the exact same number of points as the number of coefficients that you actually use in your function or else you must hold constant unused coefficients.

Certain functions may have combinations of coefficients that result in one or more of the coefficients having no effect on the fit. Consider the Gaussian function: $K_0 + K_1 \exp((x - K_2)/K_3)^2$

If K_1 is set to zero, then the following exponential has no effect on the function value. The fit will report which coefficients have no effect. In this example, it will report that K_2 and K_3 have no effect on the fit. However, as this example shows, it is often not the reported coefficients that are at fault.

Special Considerations for Polynomial Fits

Polynomial fits use the singular value decomposition technique. If you encounter singular values and some of the coefficients of the fit have been zeroed, you are probably asking for more terms than can be supported by your data. You should use the smallest number of terms that gives a "reasonable" fit. If your data does not support higher-order terms then you can actually get a poorer fit by including them.

If you really think your data should fit with the number of terms you specified, you can try adjusting the singular value threshold. You do this by creating a special variable called V_tol and setting it to a value smaller than the default value of 1e-10. You might try 1e-15.

Another way to run into trouble during polynomial fitting is to use a range of X values that are very much offset from zero. If you suspect this may be the cause of problems you should be able to get good results by temporarily offsetting your x values, performing the fit and then restoring the original x values.

Errors Due to X Values with Large Offsets

The single and double exponential fits can be thrown off if you try to fit to a range of X values that are very much offset from zero. In general, any function, which, when extrapolated to zero, returns huge or infinite values, can create problems. The solution is to temporarily offset your x values, perform the fit and then restore the original x values. You may need to perform a bit of algebra to fix up the coefficients.

As an example consider a fit to an exponential where the x values range from 100 to 101. We temporarily offset the x values by 100, perform the fit and then restore the x values by adding 100. When we did the fit, rather than fitting to k0+k1*exp(-k2*x) we really did the fit to c0+c1*exp(-c2*(x-100)). A little rearrangement

and we have c0+c1*exp(-c2*x)*exp(c2*100). Comparing these expressions, we see that k0= c0, k1= c1*exp(c2*100) and k2= c2.

A better solution to the problem of fitting exponentials with large X offsets is to use the built-in exp_XOffset and dblexp_XOffset fit functions. These fit functions automatically incorporate the X shifting; see **Notes on the Built-in Fit Functions** on page III-165 for details.

The same problem can occur when fitting to high-degree polynomials. In this case, the algebra required to transform the solution coefficients back to unoffset X values is nontrivial. It would be better to simply redefine your problem in terms of offset X value.

Curve Fitting Troubleshooting

If you are getting unsatisfactory results from curve fitting you should try the following before giving up.

Make sure your data is valid. It should not be all one value. It should bear some resemblance to the function that you're trying to fit it to.

If the fit is iterative try different initial guesses.

If you are fitting to a user-defined function, check the following:

- Your coefficients wave must have exactly the same number of points as the number of coefficients that you actually use in your function unless you hold constant the unused coefficients.
- Your initial guesses should not be zero unless the expected range is near unity or you have specified an epsilon wave.
- Ensure that your function is working properly. Try plotting it over a representative domain.
- Examine your function to ensure all your coefficients are distinguishable. For example in the fragment (k0+k1)*x, k0 and k1 are indistinguishable. If this situation is detected, the history will contain the message: "Warning: These parameters may be linearly dependent:" followed by a line listing the two parameters that were detected as being indistinguishable.
- Because the derivatives for a user-defined fit function are calculated numerically, if the function depends only weakly on a coefficient, the derivatives may appear to be zero. The solution to that problem is to create an epsilon wave and set its values large enough to give a nonzero difference in the function output. The epsilon wave sets the perturbation to a coefficient that is applied to estimate derivatives.
- A variation the previous problem is a function that changes in a step-wise fashion, or is "noisy" because an approximation is used that is good to only a limited precision. Again, create an epsilon wave and set the values large enough to give nonzero differences that are of consistent sign.
- Verify that each of your coefficients has an effect on the function.
- Make sure that the optimal value of your coefficients is not infinity (it takes a long time to increment to infinity).
- Check to see if your function could possibly return NaN or INF for any value of the coefficients. You might be able to add constraints to prevent this from happening. You will see warnings if a singular matrix error resulted from NaN or INF values returned by the fitting function.

Curve Fitting References

An explanation of the Levenberg-Marquardt nonlinear least squares optimization can be found in Chapter 14.4 of:

Press, W.H., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

The algorithm used for applying constraints is given in:

Shrager, Richard, Quadratic Programming for Nonlinear Regression, *Communications of the ACM*, *15*, 41-45, 1972.

The method is described in gory mathematical detail in:

Shrager, Richard, Nonlinear Regression With Linear Constraints: An Extension of the Magnified Diagonal Method, *Journal of the Association for Computing Machinery*, *17*, 446-452, 1970.

References for the ODRPACK95 package used for orthogonal distance regression:

Boggs, P.T., R.H. Byrd, and R.B. Schnabel, A Stable and Efficient Algorithm for Nonlinear Orthogonal Distance Regression, *SIAM Journal of Scientific and Statistical Computing*, *8*, 052-1078, 1987.

Boggs, P.T., R.H. Byrd, J.R. Donaldson, and R.B. Schnabel, Algorithm 676 - ODRPACK: Software for Weighted Orthogonal Distance Regression, *ACM Transactions on Mathematical Software*, *15*, 348-364, 1989

Boggs, P.T., J.R. Donaldson, R.B. Schnabel and C.H. Spiegelman, A Computational Examination of Orthogonal Distance Regression, *Journal of Econometrics*, *38*, 69-201, 1988.

An exhaustive, but difficult to read source for nonlinear curve fitting:

Seber, G.A.F, and C.J. Wild, *Nonlinear Regression*, John Wiley & Sons, 1989.

A discussion of the assumptions and approximations involved in calculating confidence bands for nonlinear functions can be found in the beginning sections of Chapter 5.

General books on curve fitting and statistics:

Draper, N., and H. Smith, *Applied Regression Analysis*, John Wiley & Sons, 1966.

Box, G.E.P., W.G. Hunter, and J.S. Hunter, *Statistics for Experimenters*, John Wiley & Sons, 1978.

Chapter
III-9

Signal Processing

Overview

Analysis tasks in Igor range from simple experiments using no programming to extensive systems tailored for specific fields. Chapter I-2, **Guided Tour of Igor Pro**, shows examples of the former. WaveMetrics' "Peak Measurement" technical note is an example of the latter.

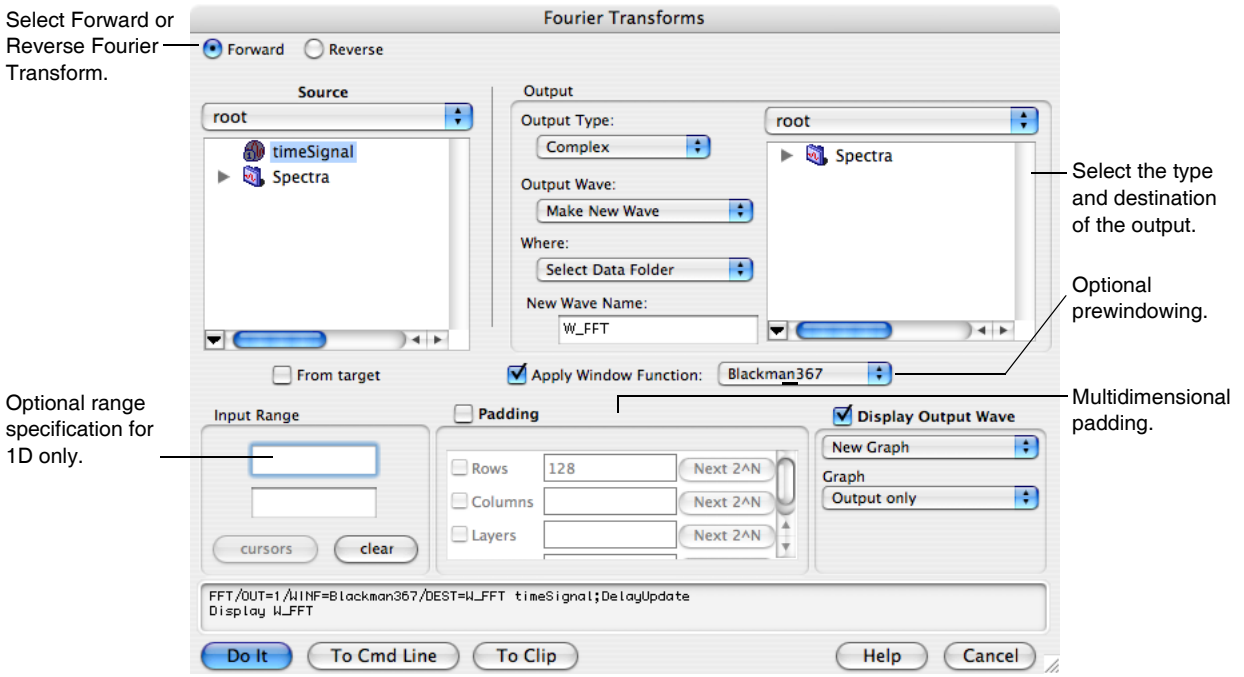
The Signal Processing chapter covers basic analysis operations with emphasis on signal transformations.

Fourier Transforms

Igor uses the Fast Fourier Transform (FFT) algorithm to compute a Discrete Fourier Transform (DFT). The FFT is usually called from an Igor procedure as one step in a larger process, such as finding the magnitude and phase of a signal. Igor's FFT uses a prime factor decomposition multidimensional algorithm. Prime factor decomposition allows the algorithm to work on nearly any number of data points. Previous versions of Igor were restricted to a power-of-two number of data points.

This section concentrates on the one-dimensional FFT. See **Multidimensional Fourier Transform** on page II-114 for information on multidimensional aspects of the FFT.

You can perform a Fourier transform on a wave by choosing Analysis→Fourier Transforms. This brings up the Fourier Transforms dialog:



Select the type of transform by clicking the Forward or Reverse radio button. Select the wave that you want to transform from the Wave list. If you enable the From Target box under the Wave list, only appropriate waves in the target window will appear in the list.

Why Some Waves Aren't Listed

What do we mean by "appropriate" waves?

The data can be either real or complex. If the data are real, the number of data points must be even. This is an artificial limitation that was introduced in order to guarantee that the inverse transform of a forward-transformed wave is equal to the original wave. For multidimensional data, only the number of rows must be even. You can work around some of the restrictions of the inverse FFT with the command line.

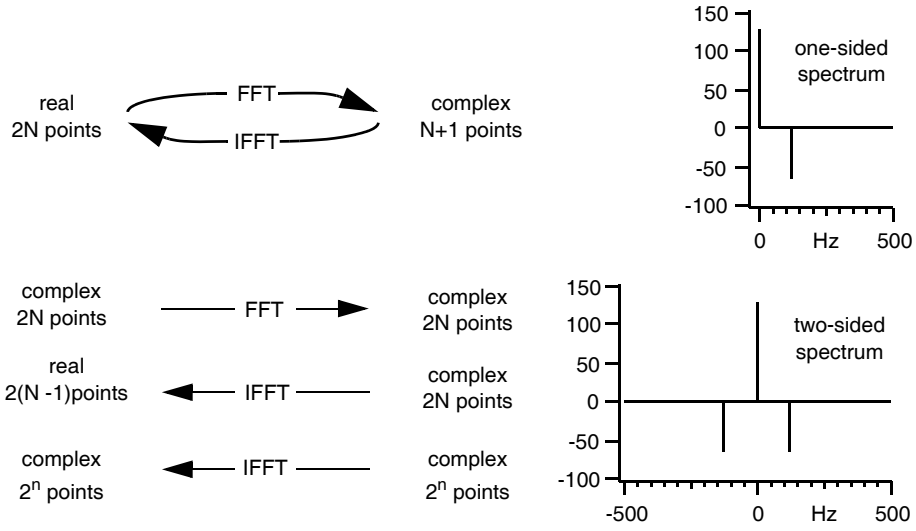
The inverse FFT requires complex data. There are no restrictions on the number of data points. However, for historic and compatibility reasons, certain values for the number of points are treated differently as described in the next sections.

Changes in Wave Type and Number of Points

If the wave is 1D real wave of N points, the FFT operation results in a complex wave consisting of N/2+1 points containing the "one-sided spectrum". The negative spectrum is not computed, since it is identical for real input waves.

If the wave is complex (even if the imaginary part is zero), its data type and number of points are unchanged by the forward FFT. The FFT result is a "two-sided spectrum", which contains both the positive *and* the negative frequency spectra, which are different if the imaginary part of the complex input data is nonzero.

The diagram below shows the two-sided spectrum of data containing a zero imaginary component. This example data has length 2^n, but your data can be any length.



Magic Number of Points and the IFFT

When performing the inverse FFT, the result may be either real or complex. Because versions prior to Igor Pro 3.0 only allowed an integral power of two (2^n) to be forward transformed, it could use the number of points to identify what kind of result to create for the inverse transform. To ensure compatibility, later versions continue to treat certain numbers of points as "magical."

If the number of points in the wave is an integral power of two (2^n), then the wave resulting from the IFFT is complex. If the number of points in the wave is one greater than an integral power of two (2^n+1), then the wave resulting from the IFFT is real and twice as long (2^{n+1}).

If the number of points is not one of the two magic values, then the result from the inverse transform is real unless the complex result is selected in the above dialog.

Changes in X Scaling and Units

The FFT operation changes the X scaling of the transformed wave. If the X-units of the transformed wave are time (s), frequency (Hz), length (m), or reciprocal length (m^{-1}), then the resulting wave units are set to the respective conjugate units. Other units are currently ignored. The X scaling's X_0 value is altered depending on whether the wave is real or complex, but dx is always set the same:

$$\Delta x_{FFT} = \frac{1}{N \cdot \Delta x_{original}} \qquad \textit{where, } N \equiv \textit{original length of wave}$$

If the original wave is real, then after the FFT its minimum X value (X_0) is zero and its maximum X value is:

$$x_{N/2} = \frac{N}{2} \cdot \Delta x_{FFT} = \frac{N}{2} \cdot \frac{1}{N \cdot \Delta x_{original}}$$

$$= \frac{1}{2 \cdot \Delta x_{original}}$$

$$= \textit{Nyquist Frequency}$$

If the original wave is complex, then after the FFT its maximum X value is $X_{N/2}$ - dX_{FFT}, its minimum X value is $-X_{N/2}$, and the X value at point N/2 is zero.

The IFFT operation reverses the change in X scaling caused by the FFT operation except that the X value of point 0 will always be zero.

FFT Amplitude Scaling

Various programs take different approaches to scaling the amplitude of an FFTed waveform. Different scaling methods are appropriate for different analyses and there is no general agreement on how this is done. Igor uses the method described in *Numerical Recipes in C* (see **References** on page III-264) which differs from many other references in this regard.

The DFT equation computed by the FFT for a complex $wave_{orig}$ with N points is:

$$wave_{FFT}[n] = \sum_{k=0}^{N-1} wave_{orig}[k] \cdot e^{2\pi i \cdot kn/N}, \text{ where } i = \sqrt{-1}$$

$wave_{orig}$ and $wave_{FFT}$ refer to the same wave before and after the FFT operation.

The IDFT equation computed by the IFFT for a complex $wave_{FFT}$ with N points is:

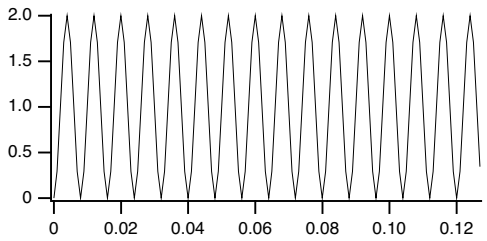
$$wave_{IFT}[n] = \frac{1}{N} \cdot \sum_{k=0}^{N-1} wave_{FFT}[k] \cdot e^{-2\pi i \cdot kn/N}, \text{ where } i = \sqrt{-1}$$

To scale $wave_{FFT}$ to give the same results you would expect from the continuous Fourier Transform, you must divide the spectral values by N, the number of points in $wave_{orig}$.

However, for the FFT of a real wave, only the positive spectrum (containing spectra for positive frequencies) is computed. This means that to compare the Fourier and FFT amplitudes, you must account for the identical negative spectra (spectra for negative frequencies) by doubling the positive spectra (but not $wave_{FFT}[0]$, which has no negative spectral value).

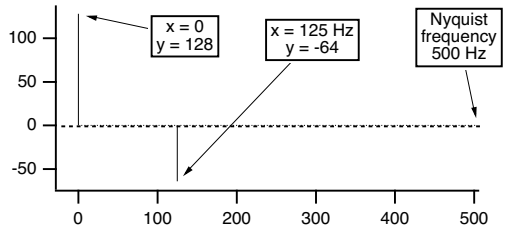
For example, here we compute the one-sided spectrum of a real wave, and compare it to the expected Fourier Transform values:

```
Make/N=128 wave0
SetScale/P x 0,1e-3,"s",wave0 // dx=1ms,Nyquist frequency is 500Hz
wave0= 1 - cos(2*Pi*125*x)    // signal frequency is 125Hz, amp. is -1
Display wave0;ModifyGraph zero(left)=3
```




```
FFT wave0
```

Igor computes the “one-sided” spectrum and updates the graph:



The Fourier Transform would predict a zero-frequency ("DC") result of 1, which is what we get when we divide the FFT value of 128 by the number of input values which is also 128. In general, the Fourier Transform value at zero frequency is:

$$Fourier\ Transform\ Amplitude(0) = \frac{1}{N} \cdot real(r2polar(wave_{FFT}(0)))$$

The Fourier Transform would predict a spectral peak at -125Hz of amplitude (-0.5 + i0), and an identical peak in the positive spectrum at +125Hz. The sum of those expected peaks would be (-1+0·i).

(This example is contrived to keep the imaginary part 0; the real part is negative because the input signal contains -cos(…) instead of + cos(…).)

Igor computed only the positive spectrum peak, so we double it to account for the negative frequency peak twin. Dividing the doubled peak of -128 by the number of input values results in (-1+i0), which agrees with the Fourier Transform prediction. In general, the Fourier Transform value at a nonzero frequency f is:

$$Fourier\ Transform\ Amplitude(f) = \frac{2}{N} \cdot real(r2polar(wave_{FFT}(f)))$$

The only exception to this is the Nyquist frequency value (the last value in the one-sided FFT result), whose value in the one-sided transform is the same as in the two-sided transform (because, unlike all the other frequency values, the two-sided transform computes only one Nyquist frequency value). Therefore:

$$Fourier\ Transform\ Amplitude(f_{Nyquist}) = \frac{1}{N} \cdot real(r2polar(wave_{FFT}(f_{Nyquist})))$$

The frequency resolution $dX_{FFT} = 1/(N_{original} \cdot dx_{original})$, or 1/(128*1e-3) = 7.8125 Hz. This can be verified by executing:

```
Print deltax(wave0)
```

Which prints into the history area:

```
  7.8125
```

You should be aware that if the input signal *is not* a multiple of the frequency resolution (our example *was* a multiple of 7.8125 Hz) that the energy in the signal will be divided among the two closest frequencies in the FFT result; this is different behavior than the continuous Fourier Transform exhibits.

Phase Polarity

There are two different definitions of the Fourier transform regarding the phase of the result. Igor uses a method that differs in sign from many other references. This is mainly of interest if you are comparing the result of an FFT in Igor to an FFT in another program. You can convert from one method to the other as follows:

```
FFT wave0;wave0=conj(wave0)    // negate the phase angle by changing
                               // the sign of the imaginary component.
```

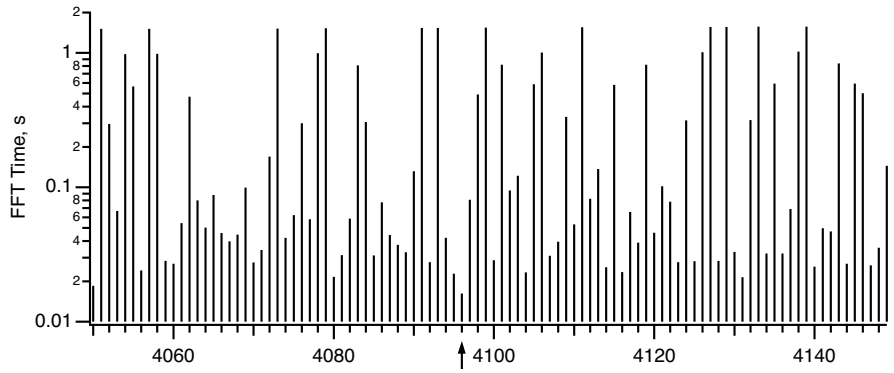
Effect of FFT and IFFT on Graphs

Igor displays complex waves in Lines between points mode by default. But, as demonstrated above, if you perform an FFT on a wave that is displayed in a graph and the display mode for that wave is lines between points, then Igor changes its display mode to Sticks to zero. Also, if you perform an IFFT on a wave that is

displayed in a graph and the display mode for that wave is Sticks to zero then Igor changes its display mode to Lines between points.

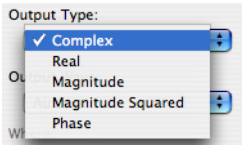
Effect of the Number of Points on the Speed of the FFT

Although the prime factor FFT algorithm does not require that the number of points be a power of two, the speed of the FFT can degrade dramatically when the number of points can not be factored into small prime numbers. The following graph shows the speed of the FFT on a complex vector of varying number of points. Note that the time (speed) axis is log. The results are from a Power Mac 9500/120.



The arrow is at N=4096, a power of two. For that number of points, the FFT time was less than 0.02 seconds while other nearby values exceed one second. The moral of the story is that you should avoid numbers of points that have large prime factors (4078 takes a long time- it has prime factors 2039 and 2). You should endeavor to use a number with small prime factors (4080 is reasonably fast — it has prime factors 2*2*2*2*3*5*17). For best performance, the number of points should be a power of 2, like 4096.

Finding Magnitude and Phase



The FFT operation can create a complex, real, magnitude, magnitude squared, or phase result directly when you choose the desired Output Type.

If you choose to use the complex wave result of the FFT operation you can compute the magnitude and phase using the **WaveTransform** operation (see page V-732) (with keywords magnitude, magsqr, and phase), or with various procedures from the WaveMetrics Procedures folder (described in the next section).

If you want to unwrap the phase wave (to eliminate the phase jumps that occur between ±180 degrees), use the Unwrap operation or the Unwrap Waves dialog in the Data menu. See **Unwrap** on page V-714. In two dimensions you can use **ImageUnwrapPhase** operation (see page V-302).

Magnitude and Phase Using WaveMetrics Procedures

For backward compatibility you can compute FFT magnitude and phase using the WaveMetrics-provided procedures in the "WaveMetrics Procedures:Analysis:DSP (Fourier Etc)" folder.

You can access them using Igor's "#include" mechanism. See **The Include Statement** on page IV-145 for instructions on including a procedure file.

The WM Procedures Index help file, which you can access from the Windows→Help Windows menu, is a good way to find out what routines are available and how to access them.

FTMagPhase Functions

The FTMagPhase functions provide an easy interface to the FFT operation. FTMagPhase has the following features:

- Automatic display of the results.
- Original data is untouched.

- Can display magnitude in decibels.
- Optional phase display in degrees or radians.
- Optional 1D phase unwrapping.
- Resolution enhancement.
- Supports non-power-of-two data with optional windowing.

Use `#include <FTMagPhase>` in your procedure file to access these functions.

FTMagPhaseThreshold Functions

The FTMagPhaseThreshold functions are the same as the FTMagPhase procedures, but with an extra feature:

- Phase values for low-amplitude signals may be ignored.

Use `#include <FTMagPhaseThreshold>` in your procedure file to access these functions.

DFTMagPhase Functions

The DFTMagPhase functions are similar to the FTMagPhase procedures, except that the slower Discrete Fourier Transform is used to perform the calculations:

- User-selectable frequency start and end.
- User-selectable number of frequency bands.

The procedures also include the DFTAtOneFrequency procedure, which computes the amplitude and phase at a single user-selectable frequency.

Use `#include <DFTMagPhase>` in your procedure file to access these functions.

CmplxToMagPhase Functions

The CmplxToMagPhase functions convert a complex wave, presumably the result of an FFT, into separate magnitude and phase waves. It has many of the features of FTMagPhase, but doesn't do the FFT.

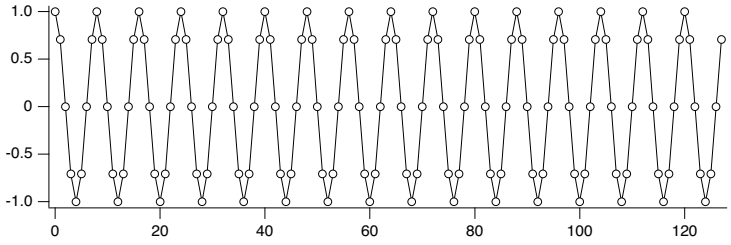
Use `#include <CmplxToMagPhase>` in your procedure file to access these functions.

Spectral Windowing

The FFT computation makes an assumption that the input data repeats over and over. This is important if the initial value and final value of your data are not the same. A simple example of the consequences of this repeating data assumption follows.

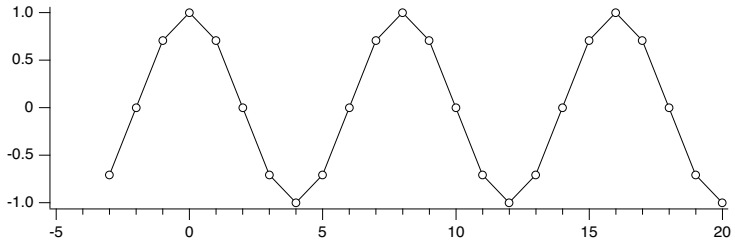
Suppose that your data is a sampled cosine wave containing 16 complete cycles:

```
Make/O/N=128 cosWave=cos(2*pi*p*16/128)
Display cosWave
ModifyGraph mode=4,marker=8
```



Notice that if you copied the last several points of cosWave to the front, they would match up perfectly with the first several points of cosWave. In fact, let's do that with the **Rotate** operation (see page V-532):

```
Rotate 3,cosWave        // wrap last three values to front of wave
SetAxis bottom,-5,20    // look more closely there
```

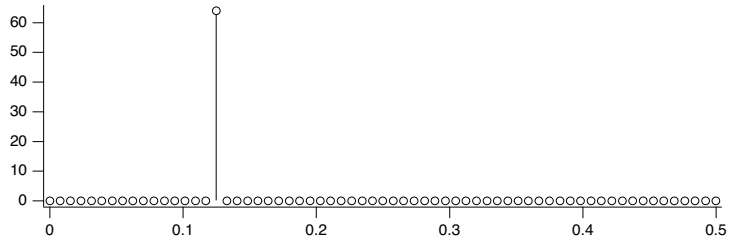


The rotated points appear at x=-3, -2, and -1. This indicates that there is no discontinuity as far as the FFT is concerned.

Because of the absence of discontinuity, the FFT magnitude result matches the ideal expectation:

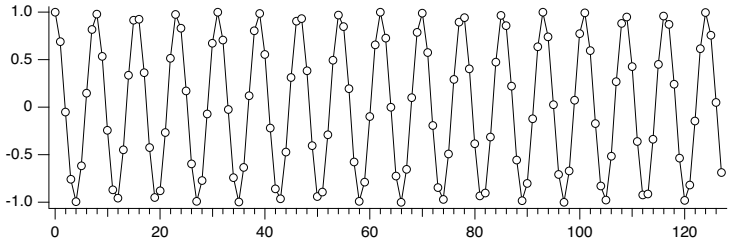
ideal FFT amplitude = cosine amplitude * number of points/2 = 1 * 128 / 2 = 64

```
FFT/Out=3/Dest=cosWave cosWave
SetAxis/A
```



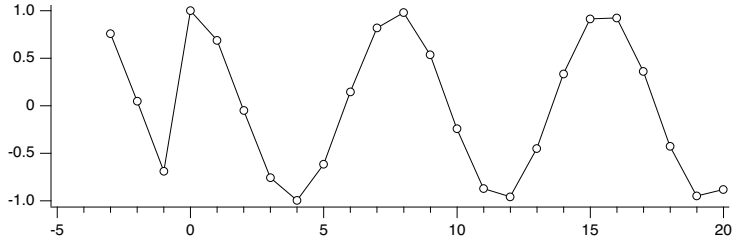
Notice that all other FFT magnitudes are zero. Now let us change the data so that there are 16.5 cosine cycles:

```
Make/O/N=128 cosWave=cos(2*pi*p*16.5/128)
```



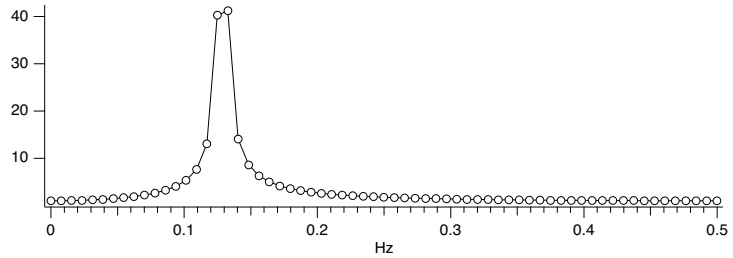
When we rotate this data as before, you can see what the FFT will perceive to be a discontinuity between the point 127 and point 0 of the unrotated data. In this next graph, the original point 127 has been rotated to x= -1 and point 0 is still at x=0.

```
Rotate 3,cosWave
SetAxis bottom,-5,20
```



When the FFT of this data is computed, the discontinuity causes “leakage” of the main cosine peak into surrounding magnitude values.

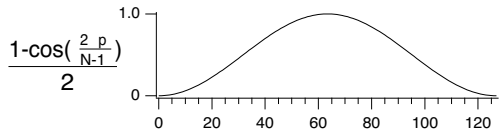
```
FFT/Out=3/Dest=cosWave cosWave
SetAxis/A
```



How does all this relate to spectral windowing? Spectral windowing reduces this leakage and gives more accurate FFT results. Specifically, windowing reduces the number of adjacent FFT values affected by leakage. A typical window accomplishes this by smoothly attenuating both ends of the data towards zero.

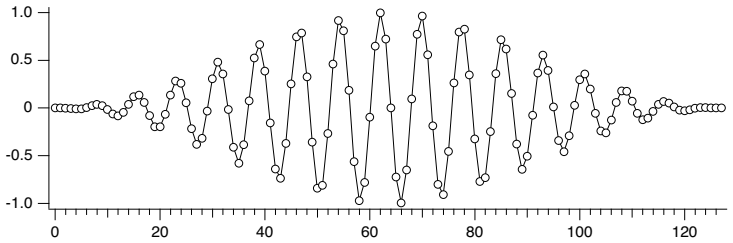
Hanning Window

Windowing the data *before* the FFT is computed can reduce the leakage demonstrated above. The Hanning window is a simple raised cosine function defined by the equation:



Let us apply the Hanning window to the 16.5 cycle cosine wave data:

```
Make/O/N=128 cosWave=cos(2*pi*p*16.5/128)
Hanning cosWave
```



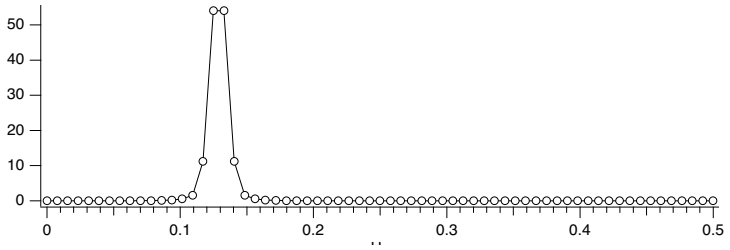
By smoothing the ends of the wave to zero, there is no discontinuity when wrapping around the ends.

In applying a window to the data, energy is lost. Depending on your application you may want to scale the output to account for coherent or incoherent gain. The coherent gain is sometimes expressed in terms of amplitude factor and it is equal to the sum of the coefficients of the window function over the interval. The incoherent gain is a power factor defined as the sum of the squares of the same coefficients. In the case that we are considering the correction factor is just the reciprocal of the coherent gain of the Hanning window

$$\text{coherent gain} \equiv \int_0^1 \left[\frac{1-\cos(2\pi x/N)}{2}\right] dx \;=\; 0.5$$

so we can multiply the FFT amplitudes by 2 to correct for them:

```
cosWave *= 2                    // account for coherent gain
FFT /Out=3/Dest=cosWave cosWave
```



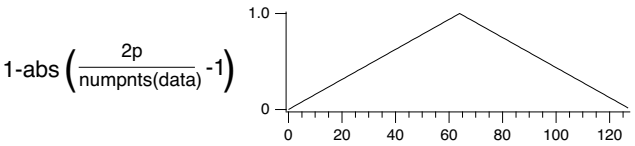
Note that frequency values in the neighborhood of the peak are less affected by the leakage, and that the amplitude is closer to the ideal of 64.

Other Windows

The Hanning window is not the ultimate window. Other windows that suppress more leakage tend to broaden the peaks. The FFT and WindowFunction operations have the following built-in windows: Hanning, Hamming, Bartlett, Blackman, Cosa(x), KaiserBessel, Parzen, Riemann, and Poisson.

You can create other windows by writing a user-defined function or by executing a simple wave assignment statement such as this one which applies a triangle window:

```
data *= 1-abs(2*p/numpnts(data)-1)
```



Use point indexing to avoid X scaling complications. You can determine the effect a window has by applying it to a perfect cosine wave, preferably a cosine wave at 1/4 of the sampling frequency (half the Nyquist frequency).

Other windows are provided in the WaveMetrics-supplied "DSP Window Functions" procedure file.

Multidimensional Windowing

When performing FFTs on images, artifacts are often produced because of the sharp boundaries of the image. As is the case for 1D waves, windowing of the image can help yield better results from the FFT.

To window images, you will need to use the ImageWindow operation, which implements the Hanning, Hamming, Bartlett, Blackman, and Kaiser windowing filters. See the **ImageWindow** operation on page V-304 for further details. For a windowing example, see **Correlations** on page III-306.

Power Spectra

Periodogram

The periodogram of a signal s(t) is an estimate of the power spectrum given by

$$P(f) = \frac{|F(f)|^2}{N},$$

where F(f) is the Fourier transform of s(t) computed by a Discreet Fourier Transform (DFT) and N is the normalization (usually the number of data points). You can compute the periodogram using the FFT but it is easier to use the DSPPeriodogram operation, which has the same built-in window functions but you can also select your own normalization to suppress the DC term or to have the results expressed in dB [20log10(P/P0)].

DSPPeriodogram can also compute the cross-power spectrum, which is the product of the Fourier transform of the first signal with the complex conjugate of the Fourier transform of the second signal:

$P(f) = \frac{F(f)G^*(f)}{N}$ where F(*f*) and G(*f*) are the DFTs of the two waves.

Power Spectral Density Functions

The PowerSpectralDensity routine supplied in the "Power Spectral Density" procedure file computes Power Spectral Density by averaging power spectra of segments of the input data. This is an early procedure file that does not take advantage of the new built-in features of the FFT or DSPPeriodogram operations. The procedure is still supported for backwards compatibility.

The PowerSpectralDensity functions take a long data wave on input and calculate the power spectral density function. These procedures have the following features:

- Automatic display of the results.
- Original data is untouched.
- Pop-up list of windowing functions.
- User setable segment length.

Use `#include <Power Spectral Density>` in your procedure file to access these functions. See **The Include Statement** on page IV-145 for instructions on including a procedure file.

PSD Demo Experiment

The PSD Demo experiment (in the Examples:Analysis: folder) uses the PowerSpectralDensity procedure and explains how it works in great detail, including justification for the scaling applied to the result.

Hilbert Transform

The Hilbert transform of a function f(x) is defined by

$$F_H(x) = \frac{1}{\pi}\int_{-\infty}^{\infty}\frac{f(t)}{t-x}dt\,.$$

The integral is evaluated as a Cauchy principal value. For numerical computation it is customary to express the integral as the convolution

$$F_H(x) = \left(\frac{-1}{\pi x}\right) \otimes f(x)\,.$$

Noting that the Fourier transform of (-1/πx) is i*sgn(x), we can evaluate the Hilbert transform using the convolution theorem of Fourier transforms. The **HilbertTransform** operation (see page V-244) is a convenient shortcut. In the next example we compute the Hilbert transform of a cosine function that gives us a sine function:

```
Make/N=512 cosWave=cos(2*pi*x*20/512)
HilbertTransform/Dest=hCosWave cosWave
Display cosWave,hCosWave
ModifyGraph rgb(hCosWave)=(0,0,65535)
```

Time Frequency Analysis

When you compute the Fourier spectrum of a signal you dispose of all the phase information contained in the Fourier transform. You can find out which frequencies a signal contains but you do not know when these frequencies appear in the signal. For example, consider the signal

$$f(t) = \begin{cases} \sin(2\pi f_1 t) & 0 \le t < t_1 \\ \sin(2\pi f_2 t) & t_1 \le t < t_2 \end{cases}.$$

The spectral representation of f(t) remains essentially unchanged if we interchange the two frequencies f_1 and f_2. In other words, the Fourier spectrum is not the best analysis tool for signals whose spectra fluctuate

in time. One solution to this problem is the so-called "short time Fourier Transform", in which you can compute the Fourier spectra using a sliding temporal window. By adjusting the width of the window you can determine the time resolution of the resulting spectra.

Two alternative tools are the Wigner transform and the Continuous Wavelet Transform (CWT).

Wigner Transform

The Wigner transform (also known as the Wigner Distribution Function or WDF) maps a 1D time signal U(*t*) into a 2D time-frequency representation. Conceptually, the WDF is analogous to a musical score where the time axis is horizontal and the frequencies (notes) are plotted on a vertical axis. The WDF is defined by the equation

$$W(t,\nu) = \int_{-\infty}^{\infty} dx\, U(t+x/2)\, U^*(t-x/2)\, e^{-i2\pi x\nu}$$

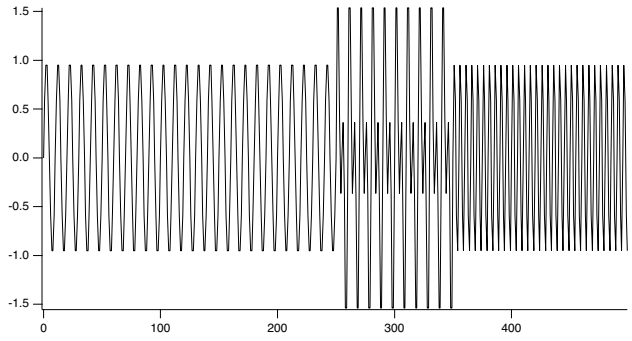
Note that the WDF W(*t*,ν) is real (this can be seen from the fact that it is a Fourier transform of an Hermitian quantity). The WDF is also a 2D Fourier transform of the Ambiguity function.

The localized spectrum can be derived from the WDF by integrating it over a finite area dtdn. Using Gaussian weight functions in both *t* and *n*, and choosing the minimum uncertainty condition dtdn=1, we obtain an estimate for the local spectrum

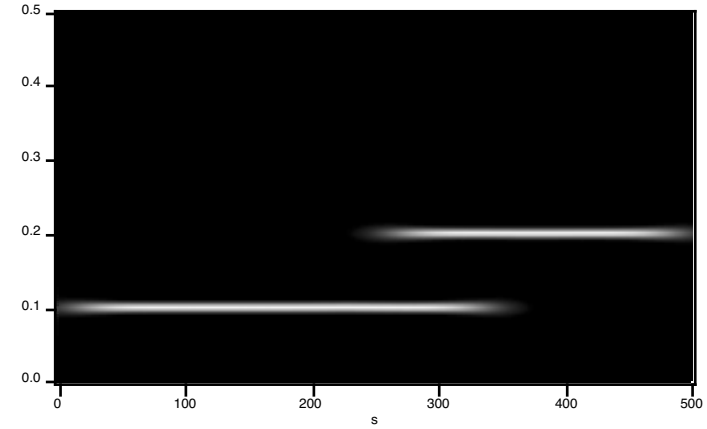
$$\hat{W}(t,\nu;\delta t) \propto \left| \int U(t') \exp\left[-2\pi \left(\frac{t-t'}{\delta t} \right)^2 \right] \exp(-i2\pi\nu t')\, dt' \right|^2$$

For an application of the **WignerTransform** operation (see page V-736), consider the two-frequency signal:

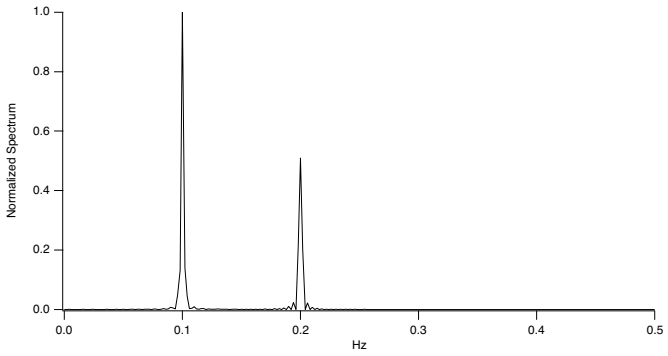
```
Make/N=500 signal
signal[0,350]=sin(2*pi*x*50/500)
signal[250,]+=sin(2*pi*x*100/500)
WignerTransform /Gaus=100 signal
DSPPeriodogram signal            // spectrum for comparison
```



The signal used in this example consists of two "pure" frequencies that have small amount of temporal overlap.



The temporal dependence is clearly seen in the Wigner transform. Note that the horizontal (time) transitions are not sharp. This is mostly due to the application of the minimum uncertainty relation dtdn=1 but it is also due to computational edge effects. By comparison, the spectrum of the signal while clearly showing the presence of two frequencies it provides no indication of the temporal variation of the signal's frequency content. Furthermore, the different power in the two frequencies may be attributed to either a different duration or a different amplitude.



Continuous Wavelet Transform

The Continuous Wavelet Transform (CWT) is a time-frequency representation of signals that graphically has a superficial similarity to the Wigner transform.

A wavelet transform is a convolution of a signal s(t) with a set of functions which are generated by translations and dilations of a main function. The main function is known as the mother wavelet and the translated or dilated functions are called wavelets. Mathematically, the CWT is given by

$$W(a,b) = \frac{1}{\sqrt{a}}\int s(t)\psi\left(\frac{t-b}{a}\right)dt\,.$$

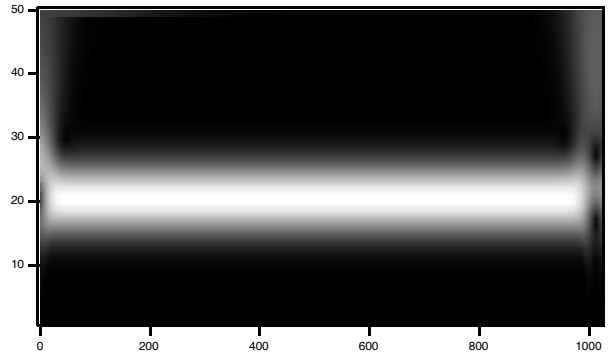
Here b is the time translation and a is the dilation of the wavelet.

From a computational point of view it is natural to use the FFT to compute the convolution which suggests that the results are dependent on proper sampling of s(t).

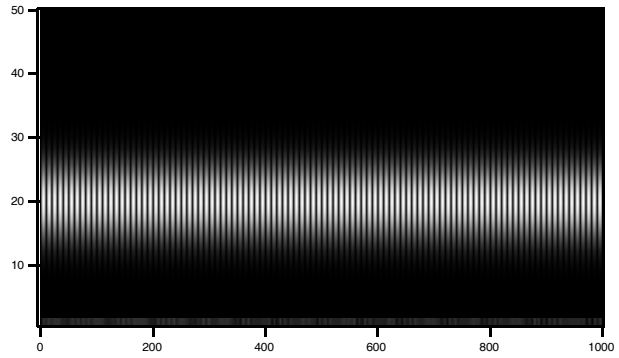
When the mother wavelet is complex, the CWT is also a complex valued function. Otherwise the CWT is real. The squared magnitude of the CWT $|W(a,b)|^2$ is equivalent to the power spectrum so that a typical display (image) of the CWT is a representation of the power spectrum as a function of time offset b. One should note however that the precise form of the CWT depends on the choice of mother wavelet y and therefore the extent of the equivalency between the squared magnitude of the CWT and the power spectrum is application dependent.

The **CWT** operation (see page V-97) is implemented using both the FFT and the discrete sum approach. You can use either one to get a representation of the effective wavelet using a delta function as an input. When comparing two CWT results you should always check that both use exactly the same definition of the wavelet function, same normalization and same computation method. For example,

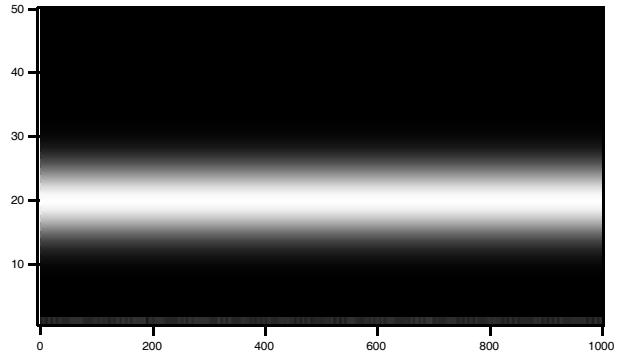
```
Make/N=1000 signal=sin(2*pi*x*50/1000)
CWT/OUT=4/SMP2=1/R2={1,1,40}/WBI1=Morlet/WPR1=5/FSCL signal
```




```
CWT /M=1/OUT=4/SMP2=1/R2={1,1,40}/WBI1=Morlet/FSCL /ENDM=2 signal
```



Using the complex Morlet wavelet in the direct sum method (/M=1) and displaying the squared magnitude we get



It is apparent that the last image has essentially the same results as the one generated using the FFT approach but in this case the edge effects are completely absent.

Discrete Wavelet Transform

The Discrete Wavelet Transform (DWT) has been supported in an IGOR XOP. As of IGOR Pro 5.0, the **DWT** operation (see page V-135) is available as a built-in command. To maintain backward compatibility and avoid conflicts with the XOP, the operation is named DWT.

The DWT is similar to the Fourier transform in that it is a decomposition of a signal in terms of a basis set of functions. In Fourier transforms the basis set consists of sines and cosines and the expansion has a single parameter. In wavelet transform the expansion has two parameters and the functions (wavelets) are generated from a single "mother" wavelet using dilation and offsets corresponding to the two parameters.

$$f(t) = \sum_a \sum_b c_{ab} \Psi_{ab}(t),$$

where the two-parameter expansion coefficients are given by

$$c_{ab} = \int f(t) \Psi_{ab}(t) dt$$

and the wavelets obey the condition

$$\Psi_{ab}(t) = 2^{\frac{a}{2}} \Psi(2^a t - b).$$

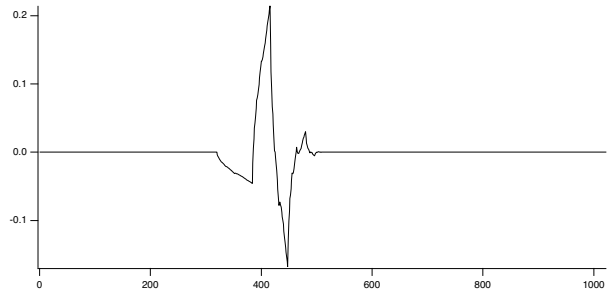
Here Ψ is the mother wavelet, a is the dilation parameter and b is the offset parameter.

The two parameter representation can complicate things quickly as one goes from 1D signal to higher dimensions. In addition, because the number of coefficients in each scale varies as a power of 2, the DWT of a 1D signal is not conveniently represented as a 2D image (as is the case with the CWT). It is therefore customary to "pack" the results of the transform so that they have the same dimensionality of the input. For example, if the input is a 1D wave of 128 (=27) points, there are 7-1=6 significant scales arranged as follows:

Scale	Storage Location
1	64-127
2	32-63
3	16-31
4	8-15
5	4-7
6	2-3

An interesting consequence of the definition of the DWT is that you can find out the shape of the wavelet by transforming a suitable form of a delta function. For example:

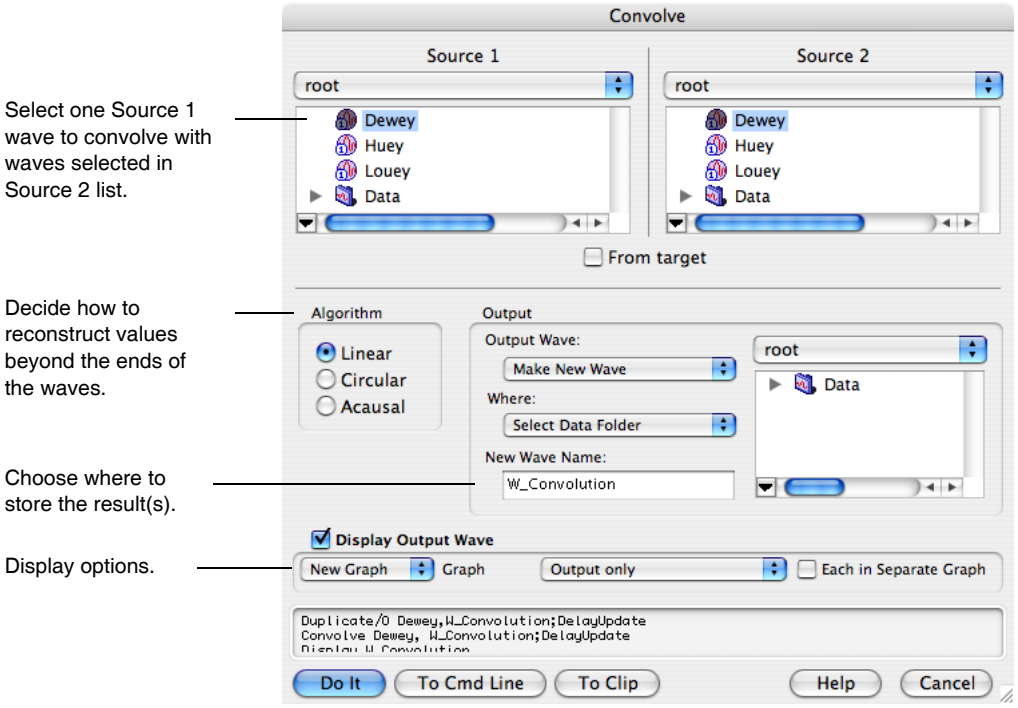
```
Make/N=1024 delta=0
delta[22]=1
DWT/I delta
Display W_DWT     // Daubechies 4 coefficient wavelet
```



Convolution

You can use convolution to compute the response of a linear system to an input signal. The linear system is defined by its impulse response. The convolution of the input signal and the impulse response is the output signal response. Convolution is also the time-domain equivalent of filtering in the frequency domain.

Igor implements general convolution with the **Convolve** operation (see page V-69). (Smoothing is also a form of convolution; see **Smoothing** on page III-255.) To use the Convolve operation, you can choose Analysis→Convolve.



The built-in Convolve operation computes the convolution of two waves named "source" and "destination" and overwrites the destination wave with the results. The operation can also convolve a single source wave with multiple destination waves (overwriting the corresponding destination wave with the results in each case). The Convolve dialog allows for more flexibility by preduplicating the second waves into new destination waves.

If the source wave is real-valued, each destination wave must be real-valued and if source wave is complex, each destination wave must be complex, too. Double and single precision waves may be freely intermixed; the calculations are performed in the higher precision.

Convolve combines neighboring points before and after the point being convolved, and at the ends of the waves not enough neighboring points exist. This is a general problem in any convolution operation; the smoothing operations use the End Effect pop-up to determine what to do. The Convolve dialog presents three algorithms in the Algorithm group to deal with these missing points.

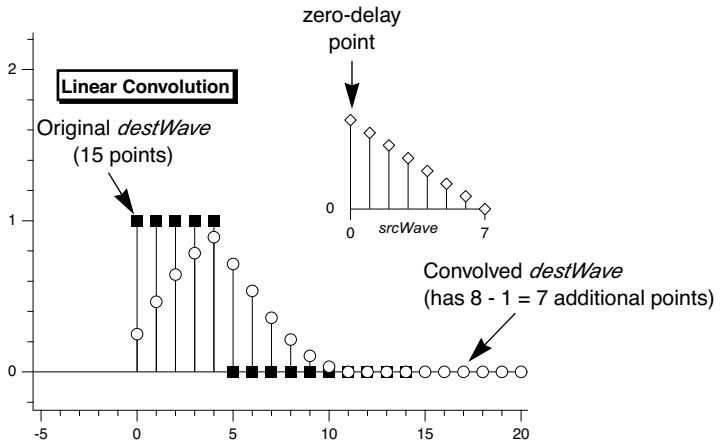
The Linear algorithm is similar to the Smooth operation's Zero end effect method; zeros are substituted for the values of missing neighboring points.

The Circular algorithm is similar to the Wrap end effect method; this algorithm is appropriate for data which is assumed to endlessly repeat.

The acausal algorithm is a special case of Linear which eliminates the time delay that Linear introduces.

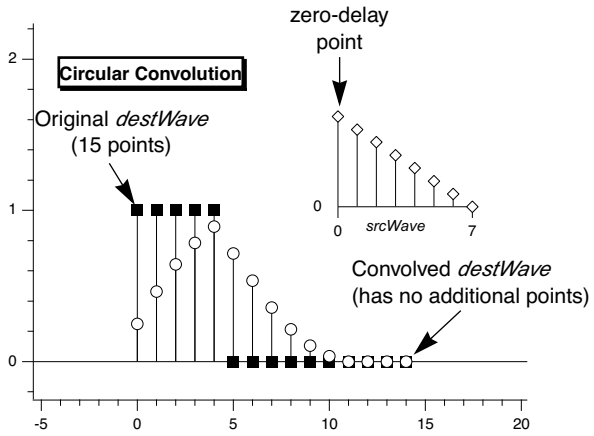
Depending on the algorithm chosen, the number of points in the destination waves may increase by the number of points in the source wave, less one. For linear and acausal convolution, the destination wave is first zero-padded by one less than the number of points in the source wave. This prevents the "wrap-

around" effect that occurs in circular convolution. The zero-padded points are removed after acausal convolution, and retained after linear convolution.

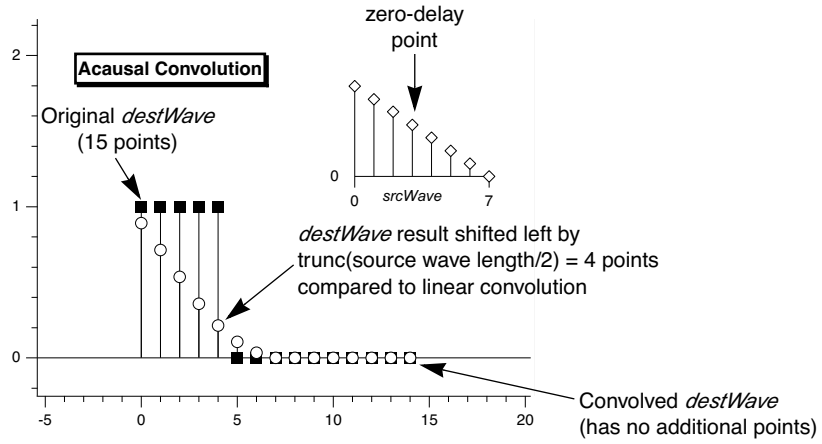


Use linear convolution when the source wave contains an impulse response (or filter coefficients) where the first point of *srcWave* corresponds to no delay ($t = 0$).

Use Circular convolution for the case where the data in the source wave and the destination waves are considered to endlessly repeat (or "wrap around" from the end back to the start), which means no zero padding is needed.

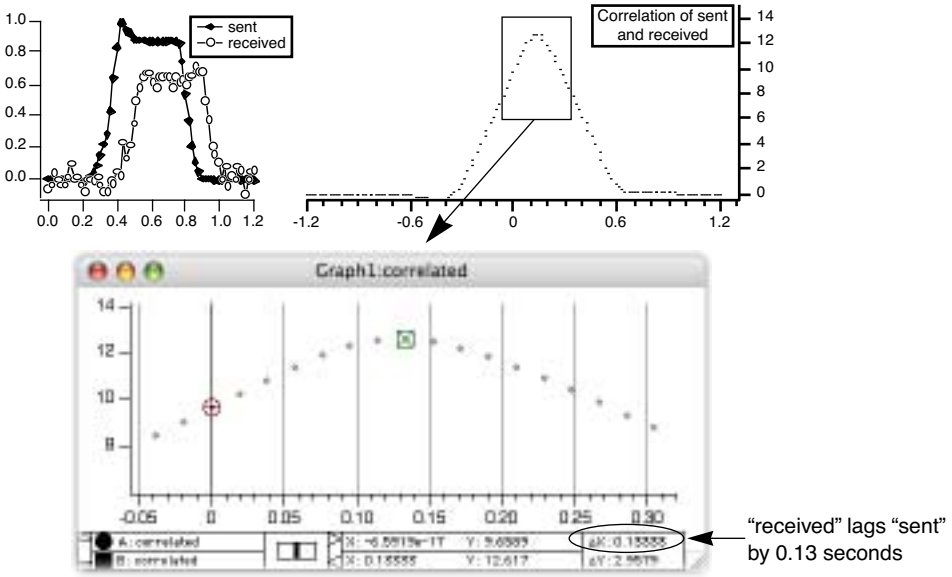


Use acausal convolution when the source wave contains an impulse response where the middle point of the source wave corresponds to no delay ($t = 0$).



Correlation

You can use correlation to compare the similarity of two sets of data. Correlation computes a measure of similarity of two input signals as they are shifted by one another. The correlation result reaches a maximum at the time when the two signals match best. If the two signals are identical, this maximum is reached at *t* = 0 (no delay). If the two signals have similar shapes but one is delayed in time and possibly has noise added to it then correlation is a good method to measure that delay.

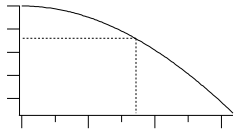


Igor implements correlation with the **Correlate** operation (see page V-73). The Correlate dialog in the Analysis menu works similarly to the Convolve dialog. The source wave may also be a destination wave, in which case afterward it will contain the "auto-correlation" of the wave. If the source and destination are different, this is called "cross-correlation".

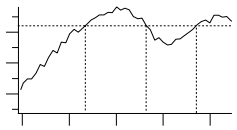
The same considerations about combining differing types of source and destination waves applies to correlation as to convolution. Correlation must also deal with end effects, and these are dealt with by the circular and linear correlation algorithm selections. See **Convolution** on page III-249.

Level Detection

Level detection is the process of locating the X coordinate at which your data passes through or reaches a given Y value. This is sometimes called "inverse interpolation". Stated another way, level detection answers the question: "given a Y level, what is the corresponding X value?" Igor provides two kinds of answers to that question.



One answer assumes your Y data is a list of unique Y values that increases or decreases monotonically. In this case there is only one X value that corresponds to a Y value. Since search position and direction don't matter, a binary search is most appropriate. For this kind of data, use the BinarySearch or BinarySearchInterp functions.



The other answer assumes that your Y data varies irregularly, as it would with acquired data. In this case, there may be multiple X values that cross the Y level; the X value returned depends on where the search starts and the search direction through the data. The FindLevel, FindLevels, EdgeStats, and PulseStats operations deal with this kind of data.

A related, but different question is "given a function y = f(x), find x where y is zero (or some other value)". This question is answered by the FindRoots operation. See **Finding Function Roots** on page III-283, and the **FindRoots** operation on page V-175.

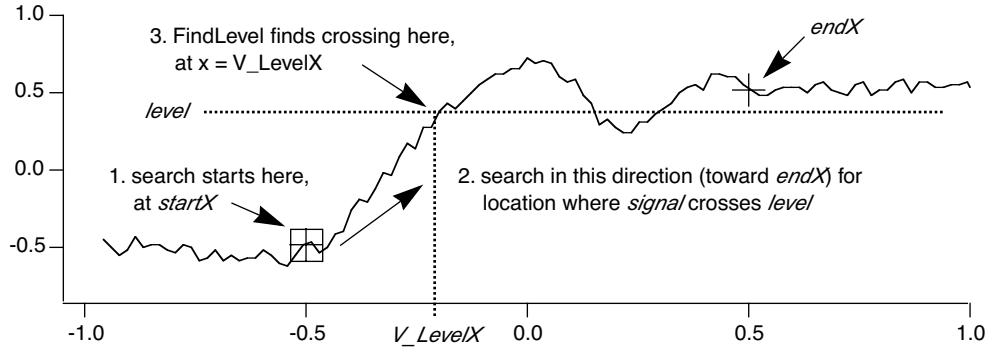
The following sections pertain to detecting level crossings in data that varies irregularly. The operations discussed are not designed to detect peaks; see **Peak Measurement** on page III-254.

Finding a Level in Waveform Data

You can use the **FindLevel** operation (see page V-170) to find a single level crossing, or the **FindLevels** operation (see page V-171) to find multiple level crossings in waveform data. Both of these operations can optionally smooth the waves they search to reduce the effects of noise. A subrange of the data can be searched, by either ascending or descending X values, depending on the *startX* and *endX* values you supply to the operation's /R flag.

FindLevel locates the first level crossing encountered in the search range, starting at *startX* and proceeding toward *endX* until a level crossing is found. The search is performed sequentially. The outputs of FindLevel are two special numeric variables: V_Flag and V_LevelX. V_Flag indicates the success or failure of the search (0 is success), and V_LevelX contains the X coordinate of the level crossing.

For example, given the following data:



the command:

```
FindLevel/R=(-0.5,0.5) signal,0.36
```

prints this level crossing information into the history area:

```
V_Flag= 0; V_LevelX= -0.200751;
```

Finding a Level in XY Data

You can find a level crossing in XY data by searching the Y wave and then figuring out where in the X wave that X value can be found. This requires that the values in the X wave be sorted in ascending or descending order. To ensure this, the command:

```
Sort xWave,xWave,yWave
```

sorts the waves so that the values in xWave are ascending, and the XY correspondence is preserved.

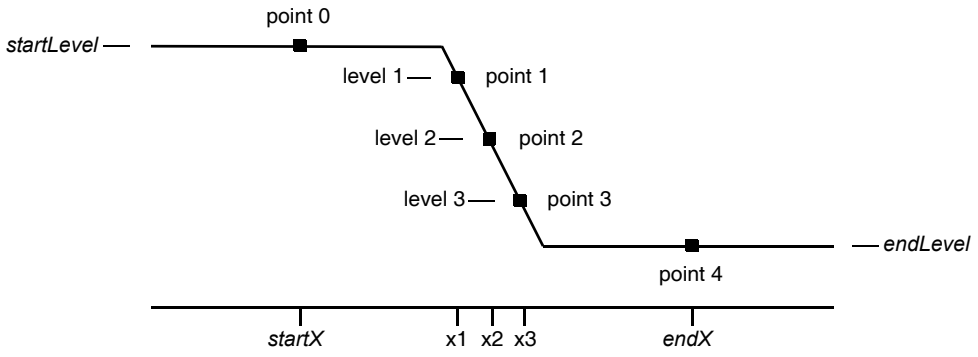
The following macros find the X location where a Y level is crossed within an X range, and store the result in the special variable V_LevelX:

```
Function FindLevelXY()

   String swy,swx                // strings contain the NAMES of waves
   Variable startX=-inf,endX=inf // startX,endX correspond to VALUEs in wx, not any X
scaling
   Variable level
   // Put up a dialog to get info from user
   Prompt swy,"Y Wave",popup WaveList("*",";","")
   Prompt swx,"X Wave",popup WaveList("*",";","")
   Prompt startX, "starting X value"
   Prompt endX, "ending X value"
   Prompt level, "level to find"
   DoPrompt "Find Level XY", swy,swx,startX, endX, level

   WAVE wx = $swx
   WAVE wy = $swy

   // Here's where the interesting stuff begins
   Variable startP,endP                  //compute point range covering startX,endX
   startP=BinarySearch(wx,startX)
   endP=BinarySearch(wx,endX)
   FindLevel/Q/R=[startP,endP] wy,level          // search Y wave, assume success
   Variable p1,m
   p1=x2pnt(wy,V_LevelX-deltaX(wy)/2)            //x2pnt rounds; circumvent it
   // Linearly interpolate between two points in wx
   // that bracket V_levelX in wy
   m=(V_LevelX-pnt2x(wy,p1))/(pnt2x(wy,p1+1)-pnt2x(wy,p1))  // slope
   V_LevelX=wx[p1] + m * (wx[p1+1] -wx[p1] )     //point-slope equation
End
```

This function does not handle a level crossing that isn't found; all that is missing is a test of V_Flag after searching the Y wave with FindLevel.

Edge Statistics

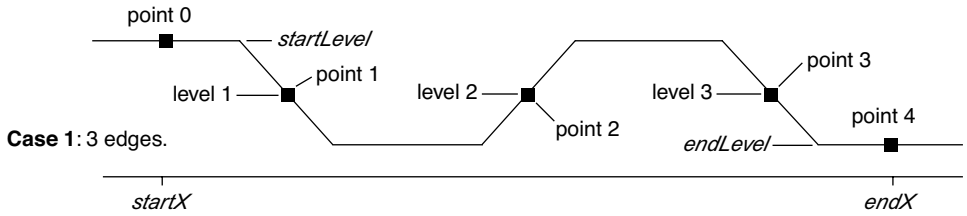
The **EdgeStats** operation (see page V-136) produces simple statistics (measurements, really) on a region of a wave that is expected to contain a single edge as shown below. If more than one edge exists, EdgeStats works on the first edge it finds. The edge statistics are stored in special variables which are described in the EdgeStats reference. The statistics are edge levels, X or point positions of various found "points", and the distances between them. These found points are actually the locations of level crossings, and are usually located between actual waveform points (they are interpolation locations).



EdgeStats is based on the same principles as FindLevel. EdgeStats does not work on an XY pair. See **Converting XY Data to a Waveform** on page III-116.

Pulse Statistics

The **PulseStats** operation (see page V-509) produces simple statistics (measurements) on a region of a wave that is expected to contain three edges as shown below. If more than three edges exist, PulseStats works on the first three edges it finds. PulseStats handles two other cases in which there are only one or two edges. The pulse statistics are stored in special variables which are described in the PulseStats reference.



PulseStats is based on the same principles as EdgeStats. PulseStats does not work on an XY pair. See **Converting XY Data to a Waveform** on page III-116.

Peak Measurement

The building block for peak measurement is the FindPeak operation. You can use it to build your own peak measurement procedures or you can use procedures provided by WaveMetrics.

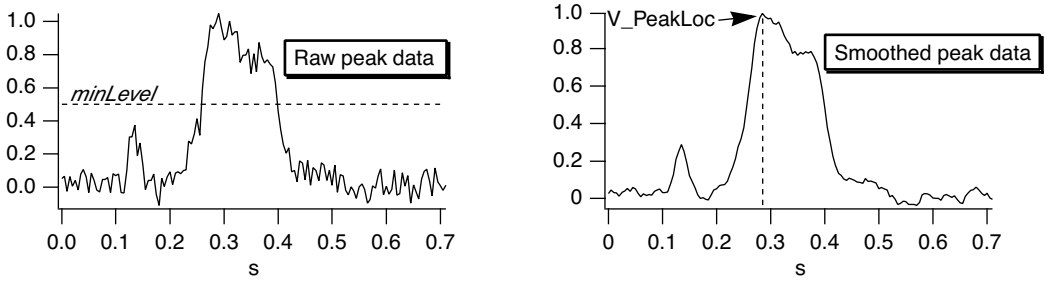
We have created several peak finding and peak fitting Technical Notes. They are described in a summary Igor Technical Note, TN020s-Choosing a Right One.ifn in the Technical Notes folder. There is also an example experiment, called Multi-peak Fit, that does fitting to multiple Gaussian, Lorentzian and Voigt peaks. Multi-peak Fit is less comprehensive but easier to use than Tech Note 20.

The **FindPeak** operation (see page V-173) searches a wave for a minimum or maximum by analyzing the smoothed first and second derivatives of the wave. The smoothing and differentiation is done on a copy of the input wave (so that the input wave is not modified). The peak maximum is detected at the smoothed first derivative zero-crossing, where the smoothed second derivative is negative. The position of the minimum or maximum is returned in the special variable V_PeakLoc. This and other special variables set by FindPeak are described in the operation reference.

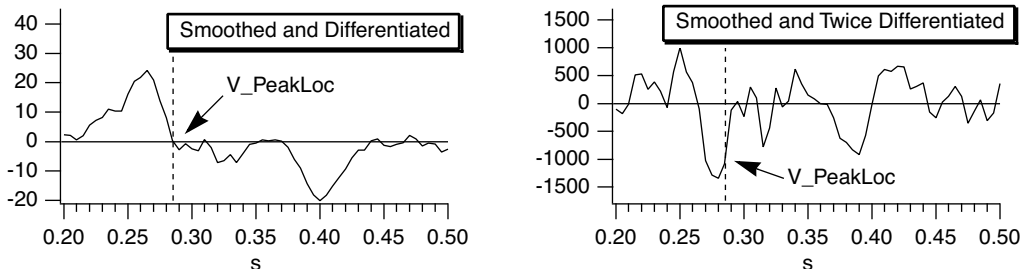
The following describes the process that FindPeak goes through when it executes a command like this:

```
FindPeak/M=0.5/B=5 peakData   // 5 point smoothing, min level = 0.5
```

The box smoothing is performed first:



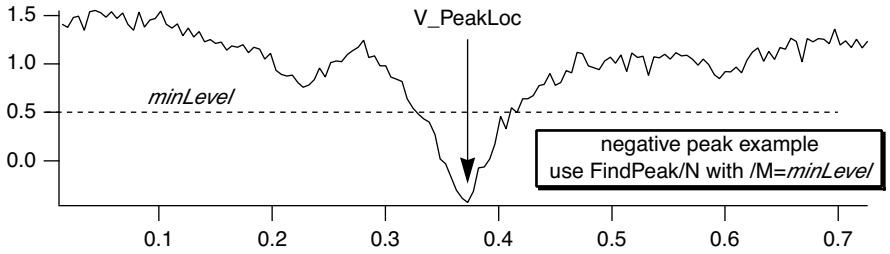
Then two central-difference differentiations are performed to find the first and second derivatives:



If you use the /M=*minLevel* flag, FindPeak ignores peaks that are lower than *minLevel* (i.e., the Y value of a found peak must exceed *minLevel*). The *minLevel* value is compared to the *smoothed* data, so peaks that appear to be large enough in the raw data may not be found if they are very near *minLevel*. If /N is also specified (search for minimum or "negative peak"), FindPeak ignores peaks whose amplitude is greater than *minLevel* (i.e., the Y value of a found peak will be *less* than *minLevel*). For negative peaks, the peak minimum is at the smoothed first derivative zero-crossing, where the smoothed second derivative is positive.

This command shows an example of finding a negative peak:

```
FindPeak/N/M=0.5/B=5 negPeakData    // 5 point smoothing, max level=0.5
```

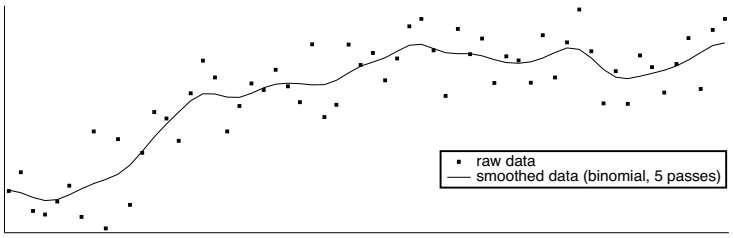


To find multiple peaks, write a procedure that calls FindPeak from within a loop. After a peak is found, restrict the range of the search with /R so that the just-found peak is excluded, and search again. Exit the loop when V_Flag indicates a peak wasn't found.

The FindPeak operation does not work on an XY pair. See **Converting XY Data to a Waveform** on page III-116.

Smoothing

Smoothing is a filtering operation used to reduce the variability of data. It is sometimes used to reduce noise.



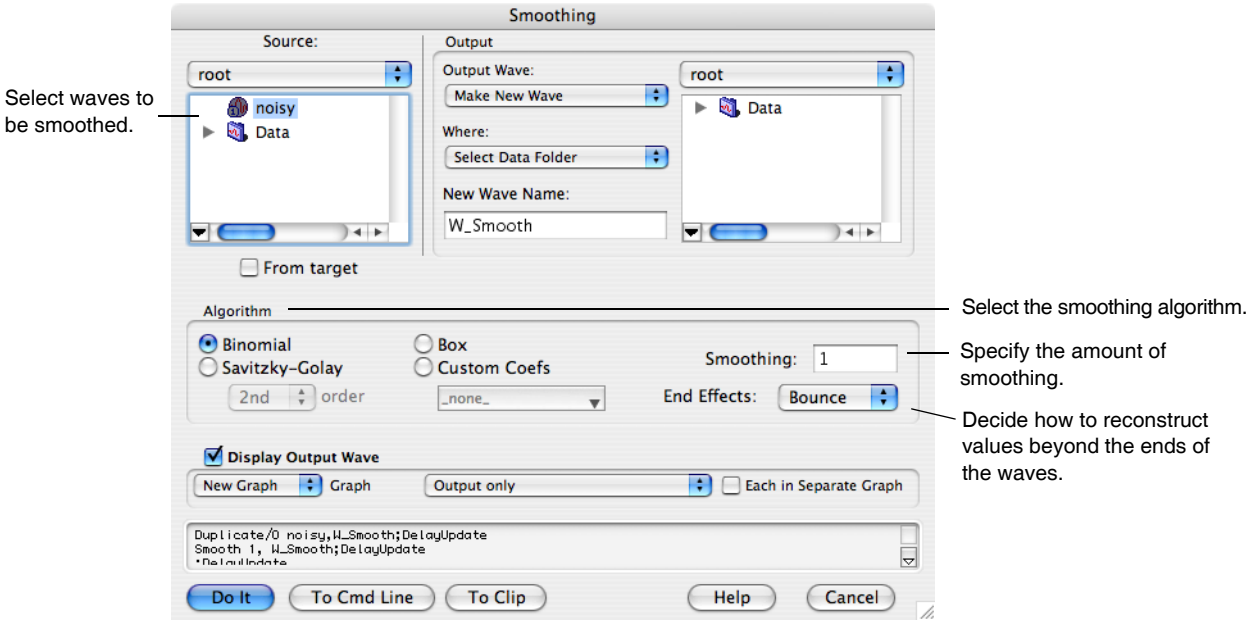
This section discusses smoothing 1-dimensional waveform data with the **Smooth**, **FilterFIR**, and **Loess** operations. Also see the **FilterIIR** and **Resample** operations.

Note: Smoothing XY data can be handled by the **Loess** operation and the Median.ipf procedure file (see **Median Smoothing** on page III-259).

Note: The **MatrixFilter**, **MatrixConvolve**, and **ImageFilter** operations smooth image and 3D data.

Igor has several built-in 1D smoothing algorithms. In addition, you can supply your own smoothing coefficients.

Choose Analysis→Smooth to see the Smoothing dialog:



Depending on the smoothing algorithm chosen, there may be additional parameters to specify in the dialog.

Built-in Smoothing Algorithms

Igor has numerous built-in smoothing algorithms for 1-dimensional waveforms, and one that works with the **XY Model of Data** on page II-78:

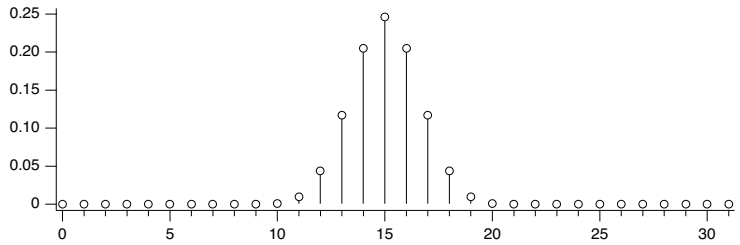
Algorithm	Operation	Data
Binomial	**Smooth**	1D waveform
Savitzky-Golay	**Smooth**/S	1D waveform
Box (Average)	**Smooth**/B	1D waveform
Custom Smoothing	**FilterFIR**	1D waveform
Median	**Smooth**/M	1D waveform
Percentile, Min, Max	**Smooth**/M/MPCT	1D waveform
Loess	**Loess**	1D waveform, XY 1D waves, false-color images*, matrix surfaces*, and multivariate data*.

* The Loess operation supports these data formats, but the Smooth dialog does not provide an interface to select them.

The first four algorithms precompute or apply one set of smoothing coefficients according to the smoothing parameters, and then replaces each data wave with the convolution of the wave with the coefficients.

You can determine what coefficients have been computed by smoothing a wave containing an impulse. For instance:

```
Make/O/N=32 wave0=0;wave0[15]=1;Smooth 5,wave0      // Smooth an impulse
Display wave0;ModifyGraph mode=8,marker=8           // Observe coefficients
```



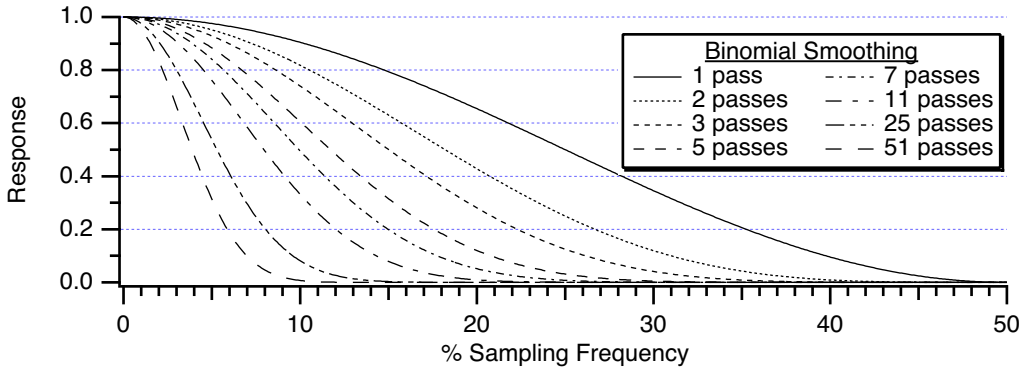
Compute the FFT of the coefficients with magnitude output to view the frequency response. See **Finding Magnitude and Phase** on page III-239.

The last two algorithms (the **Smooth**/M and **Loess** operations) are not based on creating a fixed set of smoothing coefficients and convolution, so this technique is not applicable.

Binomial Smoothing

The Binomial smoothing operation is a Gaussian filter. It convolves your data with normalized coefficients derived from Pascal's triangle at a level equal to the Smoothing parameter. The algorithm is derived from an article by Marchand and Marmet (1983).

This graph shows the frequency response of the binomial smoothing algorithm expressed as a percentage of the sampling frequency. For example, if your data is sampled at 1000 Hz and you use 5 passes, the signal at 200 Hz (20% of the sampling frequency) will be approximately 0.1.



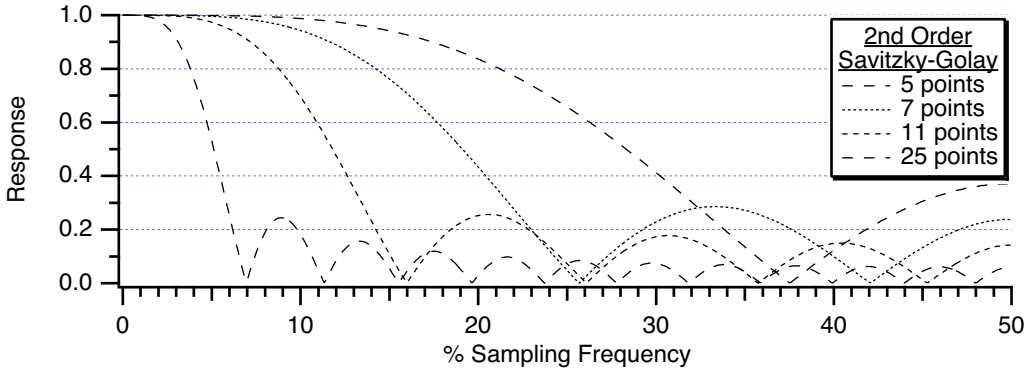
Savitzky-Golay Smoothing

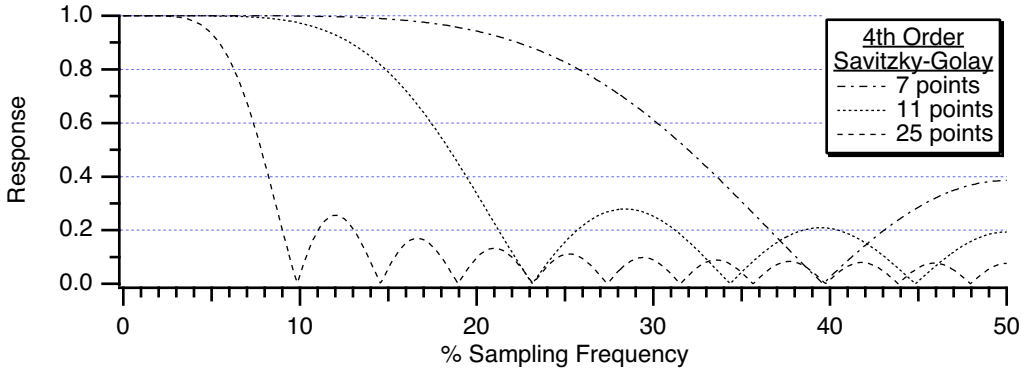
Savitzky-Golay smoothing uses a different set of precomputed coefficients popular in the field of chemistry. It is a type of Least Squares Polynomial smoothing. The amount of smoothing is controlled by two parameters: the polynomial order and the number of points used to compute each smoothed output value. This algorithm was first proposed by A. Savitzky and M.J.E. Golay in 1964. The coefficients were subsequently corrected by others in 1972 and 1978; Igor uses the corrected coefficients.

The maximum Points value is 32767; the minimum is either 5 (2nd order) or 7 (4th order). Note that 2nd and 3rd order coefficients are the same, so we list only the 2nd order choice. Similarly, 4th and 5th order coefficients are identical.

Even though Savitzky-Golay smoothing has been widely used, there are advantages to the binomial smoothing as described by Marchand and Marmet in their article.

The following graphs show the frequency response of the Savitzky-Golay algorithm for 2nd order and 4th order smoothing. The large responses in the higher frequencies show why binomial smoothing is often a better choice.

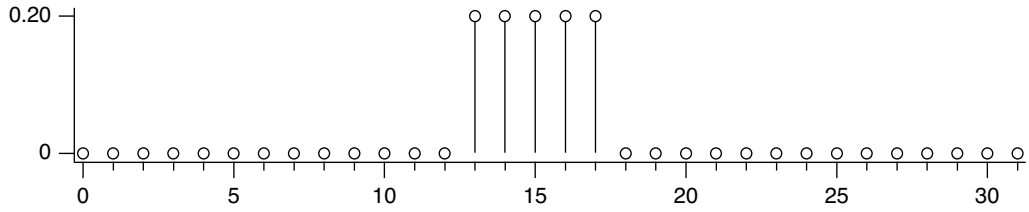




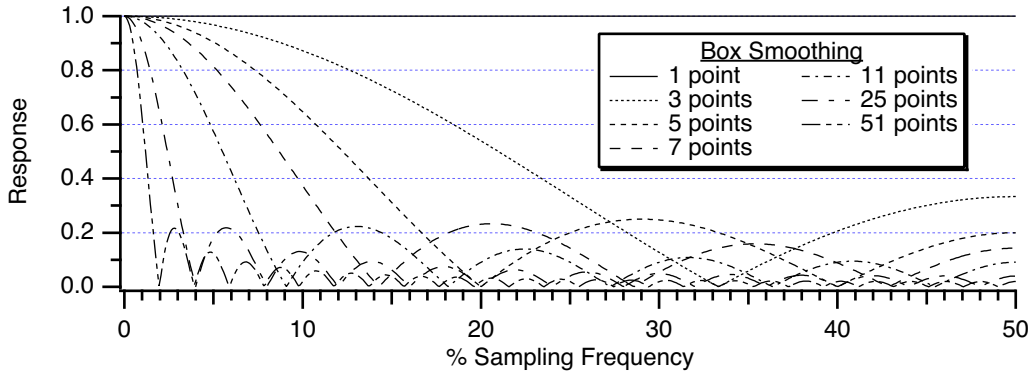
Box Smoothing

Box smoothing is similar to a moving average, except that an equal number of points before *and after* the smoothed value are averaged together with the smoothed value. The Points parameter is the total number of values averaged together. It must be an odd value, since it includes the points before, the center point, and the points after. For instance, a value of 5 averages two points before and after the center point, and the center point itself:

```
Make/O/N=32 wave0=0;wave0[15]=1;Smooth/B 5,wave0  //Smooth impulse
Display wave0;ModifyGraph mode=8,marker=8   // Observe coefficients
```



The following graph shows the frequency response of the box smoothing algorithm.



Median Smoothing

Median smoothing does not use convolution with a set of coefficients. Instead, for each point it computes the median of the values over the specified range of neighboring values centered about the point. NaN values in the waveform data are allowed and are excluded from the median calculations.

Note: For simple XY data median smoothing, include the Median.ipf procedure file:

```
#include <Median>
```

and use the Analysis→Packages→Median XY Smoothing menu item. Currently this procedure file does not handle NaNs in the data and only implements method 1 as described below.

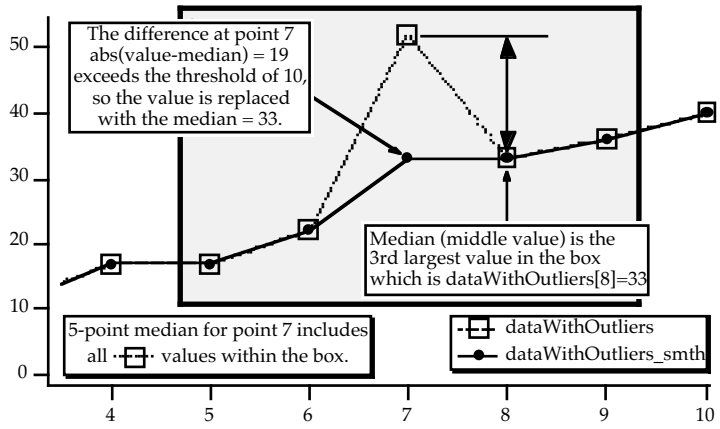
Note: For image (2D matrix) median smoothing, use the **MatrixFilter** or **ImageFilter** operation with the median method. ImageFilter can smooth 3D matrix data.

There are several ways to use median smoothing (Smooth/M) on 1D waveform data:

1) Replace all values with the median of neighboring values.
2) Replace each value with the median if the value itself is NaN. See **Replace Missing Data Using Median Smoothing** on page III-121.
3) Replace each value with the median if the value differs from the median by a the specified threshold amount.
4) Instead of replacing the value with the computed median, replace it with a specified number, including 0, NaN, +inf, or -inf.

Median smoothing can be used to replace “outliers” in data. Outliers are data that seem “out of line” from the other data. One measure of this “out of line” is excessive deviation from the median of neighboring values. The Threshold parameter defines what is considered “excessive deviation”.

```
// Example uses integer wave to simplify checking the results
Make/O/N=20/I dataWithOutliers= 4*p+gnoise(1.5)       // simple line with noise
dataWithOutliers[7] *=2                               // make an outlier at point 7
Display dataWithOutliers
Duplicate/O dataWithOutliers,dataWithOutliers_smth
Smooth/M=10 5, dataWithOutliers_smth                  // threshold=10, 5 point median
AppendToGraph  dataWithOutliers_smth
```



Percentile, Min, and Max Smoothing

Median smoothing is actually a specialization of Percentile smoothing, as are Min and Max.

Percentile smoothing returns the smallest value in the smoothing window that is greater than the smallest percentile % of the values:

Percentile	Type	Description
0	Min	The smoothed value is the minimum value in the smoothing window. 0 is the minimum value for percentile.
50	Median	The smoothed value is the median of the values in the smoothing window.
100	Max	The smoothed value is the maximum value in the smoothing window. 100 is the maximum value for percentile.

As an example, assume that percentile = 25, the number of points in the smoothing window is 7, and for one input point the values in the window after sorting are:

```
{0, 1, 8, 9, 10, 11, 30}
```

The 25th percentile is found by computing the rank R:

```
R = (percentile /100)*(num +1)
```

In this example, R evaluates to 2 so the second item in the sorted list, 1 in this example, is the percentile value for the input point.

The percentile algorithm uses an interpolated rank to compute the value of percentiles other than 0 and 100. See the **Smooth** operation for details.

Loess Smoothing

The Loess operation smooths data using locally-weighted regression smoothing. This algorithm is sometimes classified as a “nonparametric regression” procedure.

The regression can be constant, linear, or quadratic. A robust option that ignores outliers is available. In addition, for small data sets Loess can generate confidence intervals.

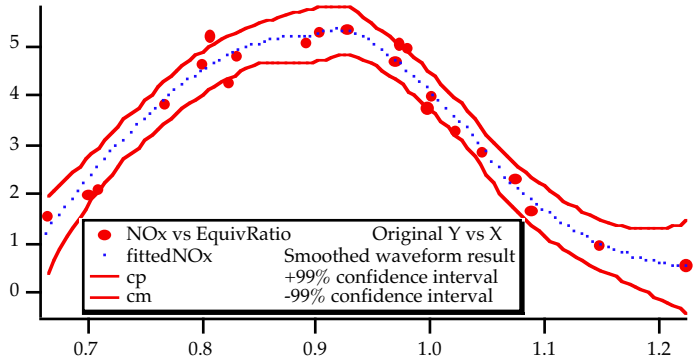
See the **Loess** operation on page V-357 help for a discussion of the basic and robust algorithms, examples, and references.

This implementation works with waveforms, XY pairs of waves, false-color images, matrix surfaces, and multivariate data (one dependent data wave with multiple independent variable data waves). Loess discards NaN input values.

The Smooth Dialog, however, provides an interface for only waveforms and XY pairs of waves (see **XY Model of Data** on page II-78), and does not provide an interface for confidence intervals or other less common options.

Here's an example from the Loess operation help of interpolating (smoothing) an XY pair and creating an interpolated 1D waveform (Y vs. X scaling). **Note**: the Make commands below are wrapped to fit the page:

```
// 3.  1-D Y vs X wave data interpolated to waveform (Y vs X scaling)
//     with 99% confidence interval outputs (cp and cm)
// NOx = f(EquivRatio)
// Y wave
Make/O/D NOx = {4.818, 2.849, 3.275, 4.691, 4.255, 5.064, 2.118, 4.602, 2.286, 0.97,
3.965, 5.344, 3.834, 1.99, 5.199, 5.283, 3.752, 0.537, 1.64, 5.055, 4.937, 1.561};

// X wave (Note that the X wave is not sorted)
Make/O/D EquivRatio = {0.831, 1.045, 1.021, 0.97, 0.825, 0.891, 0.71, 0.801,  1.074,
1.148, 1, 0.928, 0.767, 0.701, 0.807, 0.902,   0.997, 1.224, 1.089, 0.973, 0.98, 0.665};

// Graph the input data
Display NOx vs EquivRatio; ModifyGraph mode=3,marker=19

// Interpolate to dense waveform over X range
Make/O/D/N=100 fittedNOx
WaveStats/Q EquivRatio
SetScale/I x, V_Min, V_max, "", fittedNOx
Loess/CONF={0.99,cp,cm}/DEST=fittedNOx/DFCT/SMTH=(2/3) srcWave=NOx,factors={EquivRatio}

// Display the fit (smoothed results) and confidence intervals
AppendtoGraph fittedNOx, cp,cm
ModifyGraph rgb(fittedNOx)=(0,0,65535)
ModifyGraph mode(fittedNOx)=2,lsize(fittedNOx)=2
Legend
```



Note: Loess is memory intensive, especially when generating confidence intervals. Read the **Memory Details** section of the **Loess** operation (see page V-357) if you use confidence intervals.

Custom Smoothing Coefficients

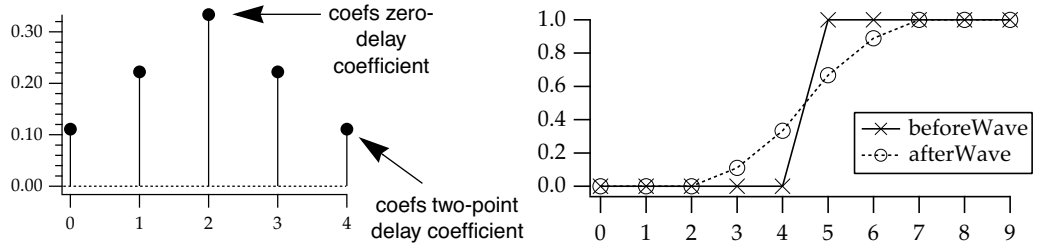
You can smooth data with your own set of smoothing coefficients by selecting the Custom Coefs algorithm. Use this option when you have low-pass filter (smoothing) coefficients created by another program or by the Igor Filter Design Laboratory.

Choose the wave that contains your coefficients from the pop-up menu that appears. Igor will convolve these coefficients with the input wave using the **FilterFIR** operation (see page V-162). You should use FilterFIR when convolving a short wave with a much longer one. Use the **Convolve** operation (see page V-69) when convolving two waves with similar number of points; it's faster.

All the values in the coefficients wave are used. FilterFIR presumes that the middle point of the coefficient wave corresponds to the delay = 0 point. This is usually the case when the coefficient wave contains the two-sided impulse response of a filter, which has an odd number of points. (For a coefficient wave with an even number of points, the "middle" point is `numpnts(coefs)/2-1`, but this introduces a usually unwanted delay in the smoothed data).

In the following example, the coefs wave smooths the data by a simple 7 point Bartlett (triangle) window (omitting the first and last Bartlett window values which are 0):

```
// This example shows a unit step signal smoothed
//   by a 7-point Bartlett window
Make/O/N=10 beforeWave = (p>=5)        // unit step at p == 5
Make/O coefs={1/3,2/3,1,2/3,1/3}       // 7 point Bartlett window
WaveStats/Q coefs
coefs/= V_Sum
Duplicate/O beforeWave,afterWave
FilterFIR/E=3/COEF=coefs afterWave
Display beforeWave,afterWave
```



End Effects

The first four smoothing algorithms compute the output value for a given point using the point's neighbors. Each algorithm combines an equal number of neighboring points before and after the point being smoothed. At the start or end of a wave some points will not have enough neighbors, so a method for fabricating neighbor values must be implemented.

You choose how to fabricate those values with the End Effect pop-up menu in the Smoothing dialog. In the descriptions that follow, i is a small positive integer, and wave[n] is the last value in the wave to be smoothed.

The Bounce method uses wave[i] in place of the missing wave[-i] values and wave[n-i] in place of the missing wave[n+i] values. This works best if the data is assumed to be symmetrical about both the start and the end of the wave. If you don't specify the end effect method, Bounce is used.

The Wrap method uses wave[n-i] in place of the missing wave[-i] values and vice-versa. This works best if the wave is assumed to endlessly repeat.

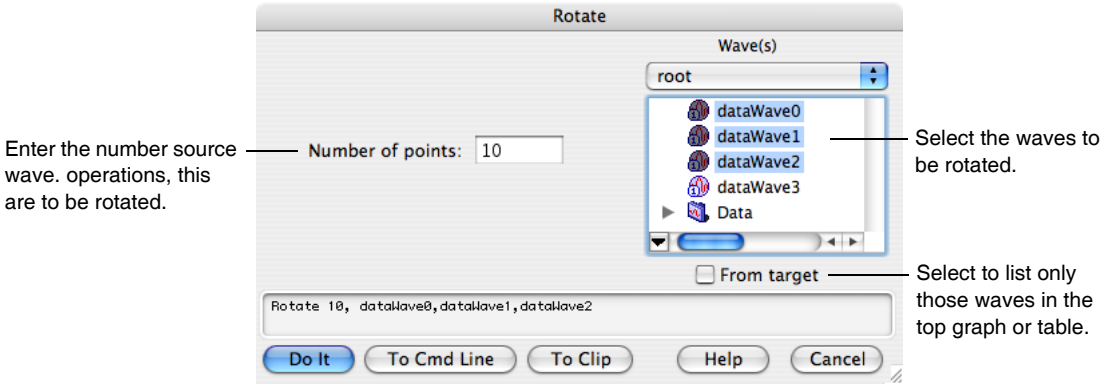
The Zero method uses 0 for any missing value. This works best if the wave starts and ends with zero.

The Repeat method uses wave[0] in place of the missing wave[-i] values and wave[n] in place of the missing wave[n+i] values. This works best for data representing a single event.

When in doubt, use Repeat.

Rotate Operation

The **Rotate** operation (see page V-532) rotates the data values of the selected waves by a specified number of points. Choose Data→Rotate Waves to display the Rotate dialog.



Think of the data values of a wave as a column of numbers. If the specified number of points is positive the points in the wave are rotated downward. If the specified number of points is negative the points in the wave are rotated upward. Values that are rotated off one end of the column wrap to the other end.

The rotate operation shifts the X scaling of the rotated wave so that, except for the points which wrap around, the X value of a given point is not changed by the rotation. To observe this, display the X scaling and data values of the wave in a table and notice the effect of Rotate on the X values.

This change in X scaling may or may not be what you want. It is usually not what you want if you are rotating an XY pair. In this case, you should undo the X scaling change using the SetScale operation:

```
SetScale/P x,0,1,"",waveName     // replace waveName with name of your wave
```

Also see the example of rotation in **Spectral Windowing** on page III-240.

Unwrap Operation

The **Unwrap** operation (see page V-714) scans through each specified wave trying to undo the effect of a modulus operation. For example, if you perform an FFT on a wave, the result is a complex wave in rectangular coordinates. You can create a real wave which contains the phase of the result of the FFT with the command:

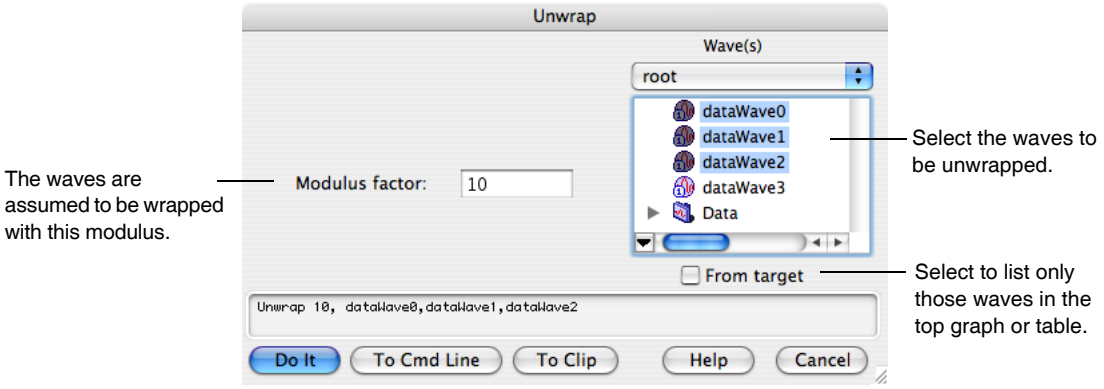
```
wave2 = imag(r2polar(wave1))
```

However the rectangular-to-polar conversion leaves the phase information modulo 2π. You can restore the continuous phase information with the command:

```
Unwrap 2*Pi, wave2
```

The Unwrap operation is designed for 1D waves only. Unwrapping 2D data is considerably more difficult. See the **ImageUnwrapPhase** operation on page V-302 for more information

The Unwrap dialog looks like this.



References

Cleveland, W.S., Robust locally weighted regression and smoothing scatterplots, *J. Am. Stat. Assoc.*, *74*, 829-836, 1977.

Marchand, P., and L. Marmet, Binomial smoothing filter: A way to avoid some pitfalls of least square polynomial smoothing, *Rev. Sci. Instrum.*, *54*, 1034-41, 1983.

Press, W.H., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

Savitzky, A., and M.J.E. Golay, Smoothing and differentiation of data by simplified least squares procedures, *Analytical Chemistry*, *36*, 1627–1639, 1964.

Wigner, E. P., On the quantum correction for thermo-dynamic equilibrium, *Physics Review*, *40*, 749-759, 1932.

Chapter III-10

Analysis of Functions

Operations that Work on Functions

Some Igor operations work on functions rather than data in waves. These operations take as input one or more functions that you define in the Procedure window. The result is some calculation based on function values produced when Igor evaluates your function.

Because the operations evaluate a function, they work on continuous data. That is, the functions are not restricted to data values that you provide from measurements. They can be evaluated at any input values. Of course, a computer works with discrete digital numbers, so even a "continuous" function is broken into discrete values. Usually these discrete values are so close together that they are continuous for practical purposes. Occasionally, however, the discrete nature of computer computations causes problems.

The following operations use functions as inputs:

- **IntegrateODE** computes numerical solutions to ordinary differential equations. The differential equations are defined as user functions. The **IntegrateODE** operation is described under **Solving Differential Equations** on page III-268.
- **FindRoots** computes solutions to f(x)=a, where a is a constant (often zero). The input x may represent a vector of x values. A special form of FindRoots computes roots of polynomials. The **FindRoots** operation is described in the section **Finding Function Roots** on page III-283.
- **Optimize** finds minima or maxima of a function, which may have one or more input variables. The **Optimize** operation is described in the section **Finding Minima and Maxima of Functions** on page III-289.
- **Integrate1D** integrates a function between two specified limits. Despite its name, it can also be used for integrating in two or more dimensions. See **Integrating a User Function** on page III-282.

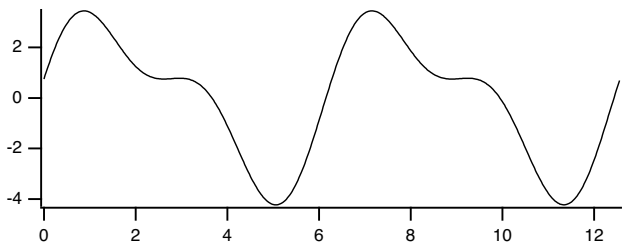
Function Plotting

Function plotting is very easy in Igor, assuming that you understand what a waveform is (see **Waveform Model of Data** on page II-77) and how X scaling works. Here are the steps to plot a function.

1. Decide how many data points you want to plot.
2. Make a wave with that many points.
3. Use the SetScale operation to set the wave's X scaling. This defines the domain over which you are going to plot the function.
4. Display the wave in a graph.
5. Execute a waveform assignment statement to set the data values of the wave.

Here is an example.

```
Make/O/N=500 wave0
SetScale/I x, 0, 4*PI, wave0     // plot function from x=0 to x=4π
Display wave0
wave0 = 3*sin(x) + 1.5*sin(2*x + PI/6)
```



To evaluate the function over a different domain, you need to reexecute the SetScale command with different parameters. This redefines "x" for the wave. Then you need to reexecute the waveform assignment statement. For example,

```
SetScale/I x, 0, 2*PI, wave0      // plot function from x=0 to x=2π
wave0 = 3*sin(x) + 1.5*sin(2*x + PI/6)
```

Reexecuting commands is easy, using the shortcuts shown in **History Area** on page II-22.

Using Dependencies

If you get tired of reexecuting the waveform assignment statement each time you change the domain, you can use a dependency to cause Igor to automatically reexecute it. To do this, use := instead of =.

```
wave0 := 3*sin(x) + 1.5*sin(2*x + PI/6)
```

See Chapter IV-9, **Dependencies**, for details.

You have made `wave0` depend on "X". The SetScale operation changes the meaning of "X" for the wave. Now when you do a SetScale on `wave0`, Igor will automatically reexecute the assignment.

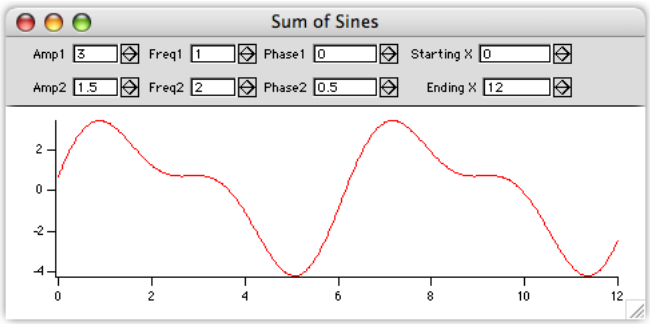
You can take this further by using global variables instead of literal numbers in the right-hand expression. For example:

```
Variable/G amp1=3, amp2=1.5, freq1=1, freq2=2, phase1=0, phase2=PI/6
wave0 := amp1*sin(freq1*x + phase1) + amp2*sin(freq2*x + phase2)
```

Now, wave0 depends on these global variables. If you change them, Igor will automatically reexecute the assignment.

Using Controls

For a slick presentation of function plotting, you can put controls in the graph to set the values of the global variables. When you change the value in the control, the global variable changes, which reexecutes the assignment. This changes the wave, which updates the graph. Here is what the graph would look like.



We've added two additional global variables and connected them to the Starting X and Ending X controls. This allows us to set the domain. These controls are both linked to an action procedure that does a SetScale on the wave.

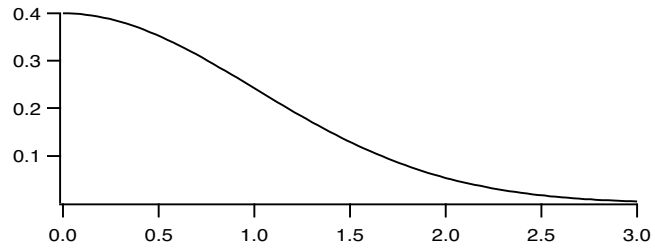
Controls are explained in detail in Chapter III-14, **Controls and Control Panels**.

Plotting a User-Defined Function

In the preceding example we used the built-in sin function in the right-hand expression. We can also use a user-defined function. Here is an example using a very simple function - the normal probability distribution function.

```
Function NormalProb(x)
   Variable x
```

```
   // the constant is 1/sqrt(2*pi) evaluated in double-precision
   return     0.398942280401433*exp(-(0.5*x^2))
End

Make/N=100 wave0; SetScale/I x, 0, 3, wave0; wave0 = NormalProb(x)
Display wave0
```



Note that, although we are using the NormalProb function to fill a wave, the NormalProb function itself has nothing to do with waves. It merely takes an input and returns a single output. We could also test the NormalProb function at a single point by executing

```
Print NormalProb(0)
```

This would print the output of the function in the history area.

It is the act of using the `NormalProb` function in a wave assignment statement that fills the wave with data values. As it executes the wave assignment, Igor calls the NormalProb function over and over again, 100 times in this case, passing it a different parameter each time and storing the output from the `NormalProb` function in successive points of the destination wave.

For more information on Wave Assignments, see **Waveform Arithmetic and Assignments** on page II-93. You may also find it helpful to read Chapter IV-1, **Working with Commands**.

WaveMetrics provides a procedure package that provides a convenient user interface to graph mathematical expressions. To use it, pull down the Analysis menu and choose Packages→Function Grapher. This will display a graph with controls to create and display a function. Click the Help button in the graph to learn how to use it.

Solving Differential Equations

Numerical solutions to initial-value problems involving ordinary differential equations can be calculated using the **IntegrateODE** operation (see page V-312). You provide a user-defined function that implements a system of differential equations. The solution to your differential equations are calculated by marching the solution forward (or backward) from the initial conditions in a series of steps or increments in the independent variable.

Terminology

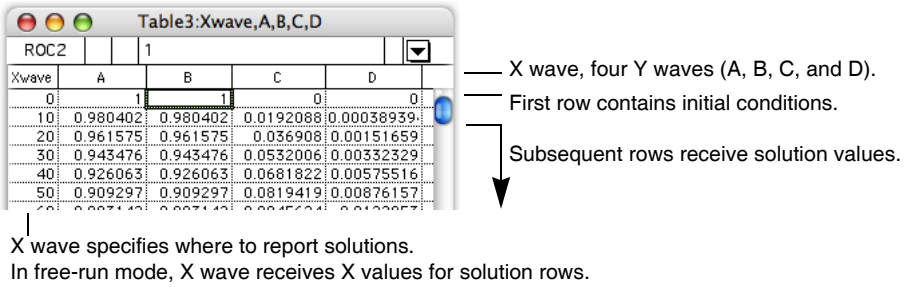
Referring to the independent variable and the dependent variables is very cumbersome, so we refer to these as X and Y[*i*]. Of course, X may represent distance or time or anything else.

A system of differential equations will be written in terms of derivatives of the Y[*i*]s, or dy[*i*]/dx.

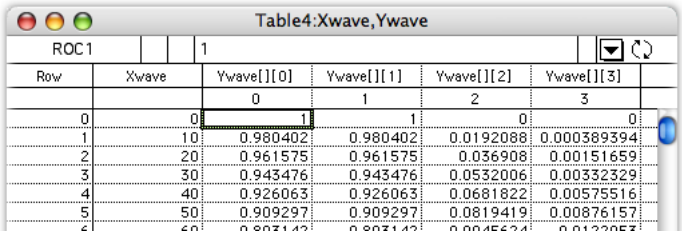
ODE Inputs

You provide to IntegrateODE a function to calculate the derivatives or right-hand-sides of your system of differential equations. You also provide a wave (or waves) to receive the solution. This solution wave will have a row for each output point you want, and a column or a wave for each equation in the system. So, for

a system of four equations (fourth-order system), if you provide an X wave to specify where you want values, you might have this situation:



You might also use a multicolumn Y wave:



Igor will calculate a solution value for each element of the Y wave (or waves). Before executing IntegrateODE, you must load the initial conditions (the initial Y[*i*] values) into the first row of the Y waves. Igor then calculates the solution starting from those values; the first solution value will be stored into the second element of the Y waves.

Note: If you are using the /R flag with IntegrateODE to start the integration at a point other than the beginning of the Y wave, the initial conditions must be in the first row specified by the /R flag. See **Stopping and Restarting IntegrateODE** on page III-280.

ODE Outputs

The algorithms Igor uses to integrate your ODE systems use adaptive step-size control. That is, the algorithms will advance the solution by the largest increment in X that will result in errors at least as small as you require. If the solution is changing rapidly, or the solution has some other difficulty, the step sizes may get very small.

IntegrateODE has two schemes for returning solution results to you: you can specify X values where you need solution values, or you can let the solution "free run".

In the first mode, results are returned to you at values of x corresponding to the X scaling of your Y waves, or at X values that you provide via an X wave or by providing X0 and deltaX and letting Igor calculate the X values. The actual calculation may require X increments smaller than those you ask for. Igor returns results only at the X values you ask for.

In free-run mode, IntegrateODE returns solution values for every step taken by the integration algorithm. In some cases, this may give you extremely small steps; you are now forewarned! Free-run mode returns to you not only the Y[*i*] values from the solution, but also values of x[*i*]. Free-run mode can be useful in that to some degree it will return results closely spaced when the solution is changing rapidly and with larger spacing when the solution is changing slowly.

The Derivative Function

You must provide a user-defined function to calculate derivatives corresponding to the equations you want to solve. All equations are solved as systems of first-order equations. Higher-order equations must be transformed to multiple first-order equations (an example is shown later).

The derivative function has this form:

```
Function D(pw, xx, yw, dydx)
   Wave pw           // parameter wave (input)
   Variable xx       // x value at which to calculate derivatives
   Wave yw           // wave containing y[i] (input)
   Wave dydx         // wave to receive dy[i]/dx (output)

   dydx[0] = <expression for one derivative>
   dydx[1] = <expression for next derivative>
   <etc.>

   return 0
End
```

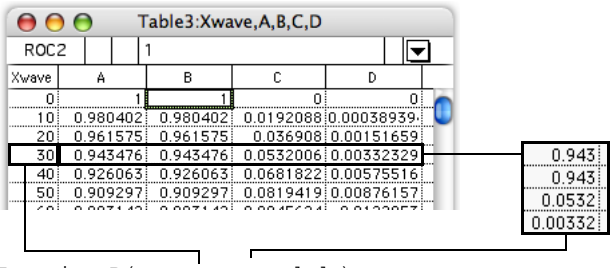
Note the return statement at the end of the function. The function result should normally be 0. If it is 1, IntegrateODE will stop. If the return statement is omitted, the function returns NaN which IntegrateODE treats the same as 0. But it is best to explicitly return 0.

Because the function may produce a large number of outputs, the outputs are returned via a wave in the parameter list.

The parameter wave is simply a wave containing possible adjustable constants. Using a wave for these makes it convenient to change the constants and try a new integration. It also will make it more convenient to do a curve fit to a differential equation. You must create the parameter wave before invoking IntegrateODE. The contents of the parameter wave are of no concern to IntegrateODE and are not touched. In fact, you can change the contents of the parameter wave inside your function and those changes will be permanent.

Other inputs are the value of x at which the derivatives are to be evaluated, and a wave (yw in this example) containing current values of the y[*i*]'s. The value of X is determined when it calls your function, and the waves yw and dydx are both created and passed to your function when Igor needs new values for the derivatives. Both the input Y wave and the output dydx wave have as many elements as the number of derivative equations in your system of ODEs.

The values in the yw wave correspond to a row of the table in the example above. That is:




```
Function D(pw, xx, yw, dydx)
    Wave pw     // parameter wave (input)
    Variable xx // x value at which to calculate derivatives
    Wave yw     // wave containing y[i] (input)
    Wave dydx   // wave to receive dy[i]/dx (output)

    dydx[0] = <expression for one derivative>
    dydx[1] = <expression for next derivative>
    <etc.>

    return 0
End
```

The wave yw contains the present value (or estimated value) of Y[*i*] at X =xx. You may need this value to calculate the derivatives.

Your derivative function is called many times during the course of a solution, and it will be called at values of X that do not correspond to X values in the final solution. The reason for this is two-fold: First, the solution method steps from one value of X to another using estimates of the derivatives at several intermediate X values. Second, the spacing between X values that you want may be larger than can be calculated accu-

rately, and Igor may need to find the solution at intermediate values. These intermediate values are not reported to you unless you call the **IntegrateODE** operation (see page V-312) in free-run mode.

Because the derivative function is called at intermediate X values, the yw wave is not the same wave as the Y wave (or waves) you create and pass to IntegrateODE. Note that one row of your Y wave (or one value from each Y wave) corresponds to the elements of the one-dimensional yw wave that is passed in to your derivative function. While the illustration implies that values from your Y wave are passed to the derivative function, in fact *the values in the yw wave passed into the derivative function correspond to whatever Y values the integrator needs at the moment*. The correspondence to your Y wave or waves is only conceptual.

You should be aware that, with the exception of the parameter wave (`pw` above) the waves are not waves that exist in your Igor experiment. Do not try to resize them with InsertPoints/DeletePoints and don't do anything to them with the Redimension operation. The yw wave is input-only; altering it will not change anything. The dydx wave is output-only; the only thing you should do with it is to assign appropriate derivative (right-hand-side) values.

Some examples are presented in the following sections.

A First-Order Equation

Let's say you want a numerical solution to a simple first-order differential equation:

$$\frac{dy}{dx} = -ay$$

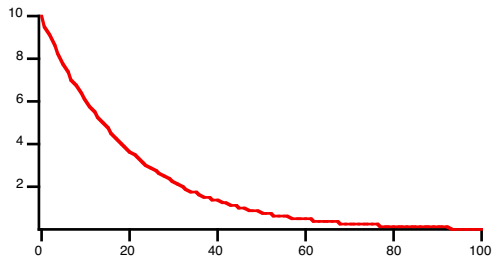
First you need to create a function that calculates the derivative. Enter the following in the procedure window:

```
Function FirstOrder(pw, xx, yw, dydx)
   Wave pw        // pw[0] contains the value of the constant a
   Variable xx    // not actually used in this example
   Wave yw        // has just one element- there is just one equation
   Wave dydx      // has just one element- there is just one equation

   // There's only one equation, so only one expression here.
   // The constant a in the equation is passed in pw[0]
   dydx[0] = -pw[0]*yw[0]

   return 0
End
```

Once the function is entered, execute the following commands:

```
Make/D/O/N=101 YY    // wave to receive results
YY[0] = 10           // initial condition- y0=10
Display YY           // make a graph
Make/D/O PP={0.05}   // set constant a to 0.05
IntegrateODE FirstOrder, PP, YY
```

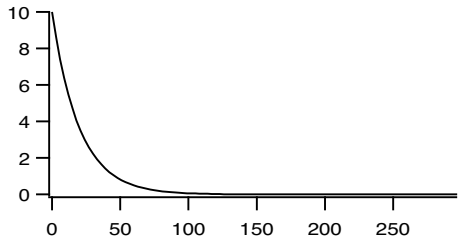
This results in the following graph with the expected exponential decay:



The IntegrateODE command shown in the example is the simplest you can use. It names the derivative function, `FirstOrder`, a parameter wave, `PP`, and a results wave, `YY`.

Because the command shown above does not explicitly set the X values, the output results are calculated according to the X scaling of the results wave YY. You can change the spacing of the X values by changing the X scaling of YY:

```
SetScale/P x 0,3,YY  // now the results will be at an x interval of 3
IntegrateODE FirstOrder, PP, YY
```



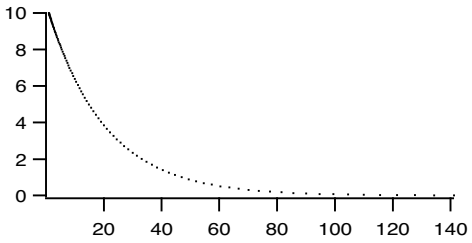
The same thing can be achieved by using your specified x0 and deltax with the /X flag:

```
IntegrateODE/X={0,3} FirstOrder, PP, YY
```

We presume that you have your own reasons for using the /X={X0, deltaX} form. Note that when you do this, it doesn't use the X scaling of your Y wave. If you graph the Y wave the values on the X axis may not match the X values used during the calculation.

Finally, you don't have to use a constant spacing in X if you provide an X wave. You might want to do this to get closely-spaced values only where the solution changes rapidly. For instance:

```
Make/D/O/N=101 XX    // same length as YY
XX = exp(p/20)       // X values get farther apart as X increases
Display YY vs XX     // make an XY graph
ModifyGraph mode=2   // plot with dots so you can see the points
IntegrateODE/X=XX FirstOrder, PP, YY
```



Note that throughout these examples the initial value of YY has remained at 10.

A System of Coupled First-Order Equations

While many interesting systems are described by simple (possibly nonlinear) first-order equations, more interesting behavior results from systems of coupled equations.

The next example comes from chemical kinetics. Suppose you mix two substances A and B together in a solution and they react to form intermediate phase C. Over time C transforms into final product D:

$$A + B \underset{k_2}{\overset{k_1}{\rightleftharpoons}} C \qquad C \xrightarrow[k_3]{} D$$

Here, k_1, k_2, and k_3 are rate constants for the reactions. The concentrations of the substances might be given by the following four coupled differential equations:

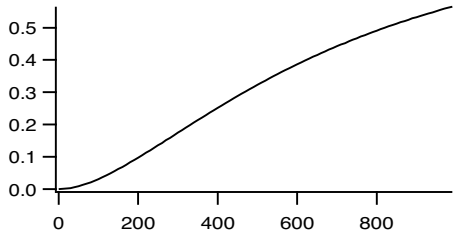
$$\frac{dA}{dt} = \frac{dB}{dt} = -k_1 \cdot A \cdot B + k_2 \cdot C \qquad \frac{dC}{dt} = k_1 \cdot A \cdot B - k_2 \cdot C - k_3 \cdot C \qquad \frac{dD}{dt} = k_3 \cdot C$$

To solve these equations, first we need a derivative function:

```
Function ChemKinetic(pw, tt, yw, dydt)
   Wave pw        // pw[0] = k1, pw[1] = k2, pw[2] = k3
   Variable tt    // time value at which to calculate derivatives
   Wave yw        // yw[0]-yw[3] containing concentrations of A,B,C,D
   Wave dydt      // wave to receive dA/dt, dB/dt etc. (output)
   dydt[0] = -pw[0]*yw[0]*yw[1] + pw[1]*yw[2]
   dydt[1] = dydt[0]  // first two equations are the same
   dydt[2] = pw[0]*yw[0]*yw[1] - pw[1]*yw[2] - pw[2]*yw[2]
   dydt[3] = pw[2]*yw[2]

   return 0
End
```

We think that it is easiest to keep track of the results using a single multicolumn Y wave. These commands make a four-column Y wave and use dimension labels to keep track of which column corresponds to which substance:

```
Make/D/O/N=(100,4) ChemKin
SetScale/P x 0,10,ChemKin  // calculate concentrations every 10 s
SetDimLabel 1,0,A,ChemKin  // set dimension labels to substance names
SetDimLabel 1,1,B,ChemKin  // this can be done in a table if you make
SetDimLabel 1,2,C,ChemKin  // the table using edit ChemKin.ld
SetDimLabel 1,3,D,ChemKin
ChemKin[0][%A] = 1      // initial conditions: concentration of A
ChemKin[0][%B] = 1      // and B is 1, C and D is 0
ChemKin[0][%C] = 0      // note indexing using dimension labels
ChemKin[0][%D] = 0
Make/D/O KK={0.002,0.0001,0.004}    // rate constants
Display ChemKin[][%D]   // graph concentration of the product
                        // Note graph made with subrange of wave
IntegrateODE/M=1 ChemKinetic, KK, ChemKin
```



Note that the waves yw and dydt in the derivative function have four elements corresponding to the four equations in the system of ODEs. At a given value of X (or t) yw[0] and dydt[0] correspond to the first equation, yw[1] and dydt[1] to the second, etc.

Note also that we have used the /M=1 flag to request the Bulirsch-Stoer integration method. For well-behaved systems, it is likely to be the fastest method, taking the largest steps in the solution.

Optimizing the Derivative Function

The Igor compiler does no optimization of your code. Because IntegrateODE may call your function thousands (or millions!) of times, efficient code can significantly reduce the time it takes to calculate the solution. For instance, the `ChemKinetic` example function above was written to parallel the chemical equations to make the example clearer. There are three terms that appear multiple times. As written, these terms are calculated again from scratch each time they are encountered. You can save some computation time by precalculating these terms as in the following example:

```
Function ChemKinetic2(pw, tt, yw, dydt)
   Wave pw        // pw[0] = k1, pw[1] = k2, pw[2] = k3
   Variable tt    // time value at which to calculate derivatives
```

```
   Wave yw        // yw[0]-yw[3] containing concentrations of A,B,C,D
   Wave dydt      // wave to receive dA/dt, dB/dt etc. (output)

   // Calculate common subexpressions
   Variable t1mt2 = pw[0]*yw[0]*yw[1] - pw[1]*yw[2]
   Variable t3 = pw[2]*yw[2]

   dydt[0] =  -t1mt2
   dydt[1] = dydt[0]          // first two equations are the same
   dydt[2] = t1mt2 - t3
   dydt[3] = t3

   return 0
End
```

These changes reduced the time to compute the solution by about 13 per cent. Your mileage may vary. Larger functions with subexpression repeated many times are prime candidates for this kind of optimization.

Note also that IntegrateODE updates the display every time 10 result values are calculated. Screen updates can be very time-consuming, so IntegrateODE provides the /U flag to control how often the screen is updated. For timing this example we used /U=1000000 which effectively turned off screen updating.

Higher Order Equations

Not all differential equations (in fact, not many) are expressed as systems of coupled first-order equations, but IntegrateODE can only handle such systems. Fortunately, it is always possible to make substitutions to transform an Nth-order differential equation into N first-order coupled equations.

You need one equation for each order. Here is the equation for a forced, damped harmonic oscillator (using y for the displacement rather than x to avoid confusion with the independent variable, which is t in this case):

$$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} + 2\lambda\frac{\mathrm{d}y}{\mathrm{d}t} + \omega^2 y = F(t)$$

If we define a new variable v (which happens to be velocity in this case):

$$v = \frac{\mathrm{d}y}{\mathrm{d}t}$$

Then

$$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = \frac{\mathrm{d}v}{\mathrm{d}t}$$

Substituting into the original equation gives us two coupled first-order equations:

$$\frac{\mathrm{d}v}{\mathrm{d}t} = -2\lambda v - \omega^2 y + F(t)$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = v$$

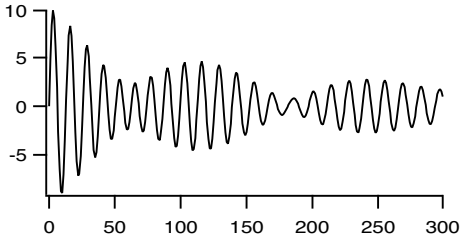
Of course, a real implementation of these equations will have to provide something for F(t). A derivative function to implement these equations might look like this:

```
Function Harmonic(pw, tt, yy, dydt)
   Wave pw     // pw[0]=damping, pw[1]=undamped frequency
               // pw[2]=Forcing amplitude, pw[3]=Forcing frequency
   Variable tt
```

```
   Wave yy     // yy[0] = velocity, yy[1] = displacement
   Wave dydt

   // simple sinusoidal forcing
   Variable Force = pw[2]*sin(pw[3]*tt)

   dydt[0] = -2*pw[0]*yy[0] - pw[1]*pw[1]*yy[1]+Force
   dydt[1] = yy[0]

   return 0
End
```

And the commands to integrate the equations:

```
Make/D/O/N=(300,2) HarmonicOsc
SetDimLabel 1,0,Velocity,HarmonicOsc
SetDimLabel 1,1,Displacement,HarmonicOsc
HarmonicOsc[0][%Velocity] = 5          // initial velocity
HarmonicOsc[0][%Displacement] = 0      // initial displacement
Make/D/O HarmPW={.01,.5,.1,.45}        // damping, freq, forcing amp and freq
Display HarmonicOsc[][%Displacement]
IntegrateODE Harmonic, HarmPW, HarmonicOsc
```



Free-Run Mode

Most of the examples shown so far use the Y wave scaling to set the X values where a solution is desired. In the section **A First-Order Equation** on page III-271 examples are also shown in which the /X flag is used to specify the sequence of X values, either by setting X0 and deltaX or by supplying a wave filled with X values.

These methods have the advantage that you have complete control over the X values where the solution is reported to you. They also are completely deterministic- you know before running IntegrateODE exactly how many points will be calculated and how big your waves need to be.

They also have the potential drawback that you may force IntegrateODE to use smaller X increments than required. If your ODE system is expensive to calculate, this may exact a considerable cost in computation time.

IntegrateODE also offers a “free-run” mode in which the solution is allowed to proceed using whatever X increments are required to achieve the requested accuracy limit. This mode has two possible advantages- it will use the minimum number of solution steps required and it may also produce a higher density of points in areas where the solution changes rapidly (but watch out for stiff systems, see page III-277).

Free-run mode has the disadvantage that in certain cases the solution may require miniscule steps to tip toe through difficult terrain, inundating you with huge numbers of points that you don’t really need. You also don’t know ahead of time how many points will be required to cover a certain range in X.

To illustrate the use of free-run mode, we will return to the example used in the section **A First-Order Equation** on page III-271. (Make sure the FirstOrder function is compiled in the procedure window.) Because we don’t know how many points will be produced, we will make the waves large:

```
Make/D/O/N=1000 FreeRunY      // wave to receive results
FreeRunY = NaN
FreeRunY[0] = 10              // initial condition- y0=10
```

Free-run mode requires that you supply an X wave. Unlike the previous use of an X wave, in free-run mode the X wave is filled by IntegrateODE with the X values at which solution values have been calculated. Like the Y waves, you must provide an initial value in the first row of the X wave. As before, it must have the same number of rows as the Y waves:

```
Make/O/D/N=1000 FreeRunX      // same length as YY
FreeRunX = NaN                // prevent display of extra points
FreeRunX[0] = 0               // initial value of X
```

In free-run mode, only the points that are required are altered. Thus, if you have some preexisting wave contents, they will be seen on a graph. We prevent the resulting confusion by filling the X wave with NaN's (Not a Number, or blanks). Igor graphs do not display points that have NaN values.

Make a graph:

```
Display FreeRunY vs FreeRunX     // make an XY graph
ModifyGraph mode=3, marker=19    // plot with dots to show the points
```

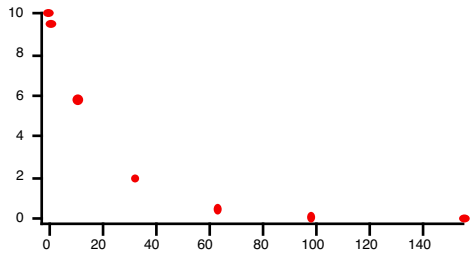
Make the parameter wave and set the value of the equation's lone coefficient:

```
Make/D/O PP={0.05}      // set constant a to 0.05
```

And finally do the integration in free-run mode. The /XRUN flag specifies a suggested first step size and the maximum X value. When the solution passes the maximum X value (100 in this case) or when your waves are filled, IntegrateODE will stop.

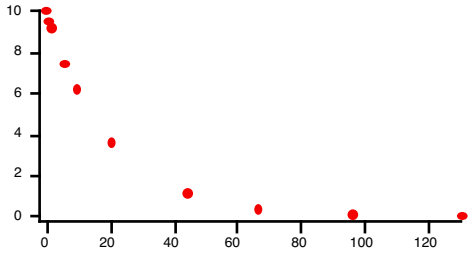
```
FreeRunX = NaN;FreeRunX[0] = 0
IntegrateODE/M=1/X=FreeRunX/XRUN={1,100} FirstOrder, PP, FreeRunY
```

In the earlier example, we (rather arbitrarily) chose 100 steps to make a reasonably smooth plot. In this case, it took 6 steps to cover the same X range, and the steps are closest together at the beginning where the exponential decay is most rapid:



Asking for more accuracy will cause smaller steps to be taken (9 when we executed the following command):

```
FreeRunX = NaN;FreeRunX[0] = 0
IntegrateODE/M=1/X=FreeRunX/XRUN={1,100}/E=1e-14 FirstOrder, PP, FreeRunY
```



After IntegrateODE has finished, you can use Redimension and the V_ODETotalSteps variable to adjust the size of the waves to just the points actually calculated:

```
Redimension/N=(V_ODETotalSteps+1) FreeRunY, FreeRunX
```

Note that we added 1 to V_ODETotalSteps to account for the initial value in row zero.

Stiff Systems

Some systems of differential equations involve components having very different time (or decay) constants. This can create what is called a "stiff" system; even though the short time constant decays rapidly and contributes negligibly to the solution after a very short time, ordinary solution methods (/M = 0, 1, and 2) are unstable because of the presence of the short time-constant component. IntegrateODE offers the Backward Differentiation Formula method (BDF, flag /M=3) to handle stiff systems.

A rather artificial example is the system (see "Numerical Recipes in C", edition 2, page 734; see **References** on page III-294)

```
du/dt = 998u + 1998v
dv/dt = -999u - 1999v
```

Here is the derivative function that implements this system:

```
Function StiffODE(pw, tt, yy, dydt)
   Wave pw     // not actually used because the coefficients
               // are hard-coded to give a stiff system
   Variable tt
   Wave yy
   Wave dydt

   dydt[0] = 998*yy[0] + 1998*yy[1]
   dydt[1] = -999*yy[0] - 1999*yy[1]

   return 0
End
```

Commands to set up the wave required and to make a suitable graph:

```
Make/D/O/N=(3000,2) StiffSystemY
Make/O/N=0 dummy              // dummy coefficient wave
StiffSystemY = 0
StiffSystemY[0][0] = 1        // initial condition for u component
make/D/O/N=3000 StiffSystemX
Display StiffSystemY[][0] vs StiffSystemX
AppendToGraph/L=vComponentAxis StiffSystemY[][1] vs StiffSystemX

// make a nice-looking graph with dots to show where the solution points are
ModifyGraph axisEnab(left)={0,0.48},axisEnab(vComponentAxis)={0.52,1}
DelayUpdate
ModifyGraph freePos(vComponentAxis)={0,kwFraction}
ModifyGraph mode=2,lsize=2,rgb=(0,0,65535)
```

These commands solve this system using the Bulirsch-Stoer method using free run mode to minimize the number of solution steps computed:

```
StiffSystemX = nan      // hide unused solution points
StiffSystemX[0] = 0     // initial X value
IntegrateODE/M=1/X=StiffSystemX/XRUN={1e-6, 2} StiffODE, dummy, StiffSystemY
Print "Required ",V_ODETotalSteps, " steps to solve using Bulirsch-Stoer"
```

which results in this message in the history area:

```
  Required   401 steps to solve using Bulirsch-Stoer
```

These commands solve this system using the BDF method:

```
StiffSystemX = nan   // hide unused solution points
StiffSystemX[0] = 0  // initial X value
IntegrateODE/M=3/X=StiffSystemX/XRUN={1e-6, 2} StiffODE, dummy, StiffSystemY
Print "Required ",V_ODETotalSteps, " steps to solve using BDF"
```

This results in this message in the history area:

```
  Required   133 steps to solve using BDF
```

I think you will agree that the difference between 401 steps and 133 is significant! Be aware, however, that the BDF method is not the most efficient for nonstiff problems.

Error Monitoring

To achieve the fastest possible solution to your differential equations, Igor uses algorithms with adaptive step sizing. As each step is calculated, an estimate of the truncation error is also calculated and compared to a criterion that you specify. If the error is too large, a smaller step size is used. If the error is small compared to what you asked for, a larger step size is used for the next step.

Igor monitors the errors by scaling the error by some (hopefully meaningful) number and comparing to an error level.

The Runge-Kutta and Bulirsch-Stoers methods (IntegrateODE flag /M=0 or /M=1) estimates the errors for each of your differential equations and the largest is used for the adjustments:

$$Max\left(\frac{Error_i}{Scale_i}\right) < eps$$

The Adams-Moulton and BDF methods (IntegrateODE flag /M=2 or /M=3) estimate the errors and use the root mean square of the error vector:

$$\left[\frac{1}{N}\sum\left(\frac{Error_i}{Scale_i}\right)^2\right]^{1/2} < eps\,.$$

Igor sets *eps* to 10^{-6} by default. If you want a different error level, use the /E=*eps* flag to set a different value of *eps*. Using the harmonic oscillator example, we now set a more relaxed error criterion than the default:

```
IntegrateODE/E=1e-3 Harmonic, HarmPW, HarmonicOsc
```

The error scaling can be composed of several parts, each optional:

$$Scale_i = h \cdot (C_i + y_i + dy_i/dx)$$

By default Igor uses constant scaling, setting h=1 and C_i=1, and does not use the y_i and dy_i/dx terms making $Scale_i$ = 1. In that case, *eps* represents an absolute error level: the error in the calculated values should be less than *eps*. An absolute error specification is often acceptable, but it may not be appropriate if the output values are of very different magnitudes.

You can provide your own customized values for C_i using the /S=*scaleWave* flag. You must first create a wave having one point for each differential equation. Fill it with your desired scaling values, and add the /S flag to the IntegrateODE operation:

```
Make/O/D errScale={1,5}
IntegrateODE/S=errScale Harmonic, HarmPW, HarmonicOsc
```

Typically, the constant values should be selected to be near the maximum values for each component of your system of equations.

Finally, you can control what Igor includes in $Scale_i$ using the /F=*errMethod* flag. The argument to /F is a bitwise value with a bit for each component of the equation above:

errMethod	What It Does
1	Add a constant C_i from *scaleWave* (or 1's if no *scaleWave*).
2	Add the current value of y_i's, the calculated result.
4	Add the current value of the derivatives, dy_i/dx.
8	Multiply by h, the current step size

Use *errMethod* = 2 if you want the errors to be a fraction of the current value of Y. That might be appropriate for solutions that asymptotically approach zero when you need smaller errors as the solution approaches zero.

The Scale numbers can never equal zero, and usually it isn't appropriate for $Scale_i$ to get very small. Thus, it isn't usually a good idea to use *errMethod* = 2 with solutions that pass through zero. A good way to avoid this problem can be to add the values of the derivatives (*errMethod* = (2+4)), or to add a small constant:

```
Make/D errScale=1e-6
IntegrateODE/S=errScale/F=(2+1) …
```

Finally, in some cases you need the much more stringent requirement that the errors be less than some global value. Since the solutions are the result of adding up myriad sequential solutions, any truncation error has the potential to add up catastrophically if the errors happen to be all of the same sign. If you are using Runge-Kutta and Bulirsch-Stoers methods (IntegrateODE flag /M=0 or /M=1), you can achieve global error limits by setting bit 3 of *errMethod* (/F=8) to multiply the error by the current step size (h in the equation above). If you are using Adams-Moulton and BDF methods (IntegrateODE flag /M=2 or /M=3) bit 3 does nothing; in that case, a conservative value of *eps* would be needed.

Caution: Higher accuracy will make the solvers use smaller steps, requiring more computation time. The trade-off for smaller step size is computation time. If you get too greedy, the step size can get so small that the X increments are smaller than the computer's digital resolution. If this happens Igor will stop the calculation and complain.

Solution Methods

Igor makes four solution methods available, Runge-Kutta-Fehlberg, Bulirsch-Stoers with Richardson extrapolation, Adams-Moulton and Backward Differentiation Formula.

Runge-Kutta-Fehlberg is a robust method capable of surviving solutions or derivatives that aren't smooth, or even have discontinuous derivatives. Bulirsch-Stoers, for well-behaved systems, will take larger steps than Runge-Kutta-Fehlberg, so it may be considerably faster. Step size for a given problem is larger so you get greater accuracy. Of course, if you ask for values closer together than the achievable step size, you get no advantage from this.

Details of these methods can be found in the second edition of *Numerical Recipes*(see **References** on page III-294).

The Adams-Moulton and Backward Differentiation Formula (BDF) methods are adapted from the CVODE package developed at Lawrence Livermore National Laboratory. In our very limited experience, for well-behaved nonstiff systems the Bulirsch-Stoers method is much more efficient than either the Runge-Kutta-Fehlberg method or the Adams-Moulton method, in that it requires significantly fewer steps for a given problem.

As shown above, stiff systems benefit greatly by the use of the BDF method. However, for nonstiff methods, it is not as efficient as the other methods.

See **IntegrateODE** on page V-312 for references on these methods.

Interrupting IntegrateODE

Numerical solutions to differential equations can require considerable computation and, therefore, time. If you find that a solution is taking too long you can abort the operation by pressing Command-period (*Mac-*

intosh) or by clicking the Abort button in the status bar (*Windows*). You may need to press and hold to make sure IntegrateODE notices.

When you abort an integration, IntegrateODE returns whatever results have been calculated. If those results are useful, you can restart the calculation from that point, using the last calculated result row as the initial conditions. Use the /R=(*startX*) flag to specify where you want to start.

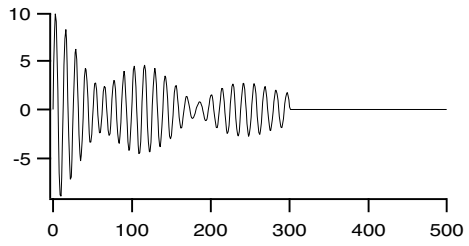
For Igor programmers, the V_ODEStepCompleted variable will be set to the last result. It is probably a good idea to restart a step or two before that:

```
IntegrateODE/R=(V_ODEStepCompleted-1) ...
```

Stopping and Restarting IntegrateODE

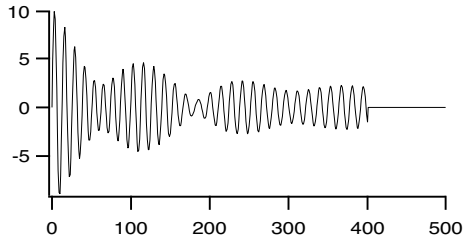
Any result can be used as initial conditions for a new solution. Thus, you can use the /R flag to calculate just a part of the solution, then finish later using the /R flag to pick up where you left off. For instance, using the harmonic oscillator example:

```
Make/D/O/N=(500,2) HarmonicOsc = 0
SetDimLabel 1,0,Velocity,HarmonicOsc
SetDimLabel 1,1,Displacement,HarmonicOsc
HarmonicOsc[0][%Velocity] = 5           // initial velocity
HarmonicOsc[0][%Displacement] = 0       // initial displacement
Make/D/O HarmPW={.01,.5,.1,.45}      // damping, freq, forcing amp and freq
Display HarmonicOsc[][%Displacement]

IntegrateODE/M=1/R=[,300] Harmonic, HarmPW, HarmonicOsc
```



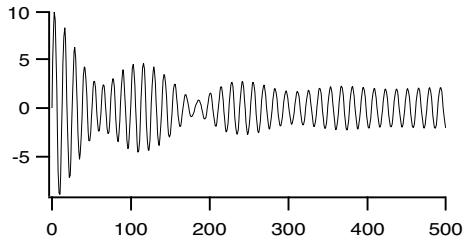
The calculation has been done for points 0-300. Note the comma in /R=[,300], which sets 300 as the end point, not the start point. Now you can restart at 300 and continue to the 400th point:

```
IntegrateODE/M=1/R=[300,400] Harmonic, HarmPW, HarmonicOsc
```



or finish the entire 500 points. Perhaps you need to start from an earlier point:

```
IntegrateODE/M=1/R=[350] Harmonic, HarmPW, HarmonicOsc
```



Stopping IntegrateODE on a Condition

Sometimes it is useful to be able to stop the calculation based on output values from the integration, rather than stopping when a certain value of the independent variable is reached. For instance, a common way to simulate a neuron firing is to solve the relevant system of equations until the output reaches a certain value. At that point, the solution should be stopped and the initial conditions reset to values appropriate to the triggered condition. Then the calculation can be re-started from that point.

The ability to stop and re-start the calculation is a general solution to the problem of discontinuities in the system you are solving. Integrate the system up to the point of the discontinuity, stop and re-start using a derivative function that reflects the system after the discontinuity.

There are two ways to stop the integration depending on the solution values.

The first way is to use the /STOP={*stopWave*, *mode*} flag, supplying a *stopWave* containing stopping conditions. *StopWave* must have one column for each equation in your system. Each column can specify stopping on a value of the solution for the equation corresponding to the column, or stopping on a value of the derivative corresponding to that equation, or both. Each row has different significance:

Row	Meaning
0	Stop flag for solution value 0: Ignore condition on solution for this equation 1: Stop when solution value is greater than the value in row 1 -1: Stop when solution value is less than the value in row 1
1	Value of solution at which to stop
2	Stop flag for derivative 0: Ignore condition on derivative for this equation 1: Stop when derivative value is greater than the value in row 3 -1: Stop when derivative value is less than the value in row 3
3	Value of derivative at which to stop

In the chemical kinetics example above (see **A System of Coupled First-Order Equations** on page III-272) the system has four equations so you need a stop wave with four columns. This wave:

Row	ChemKin_Stop[][0]	ChemKin_Stop[][1]	ChemKin_Stop[][2]	ChemKin_Stop[][3]
0	0	0	-1	1
1	0	0	0.15	0.4
2	0	0	-1	0
3	0	0	0	0

will stop the integration when the concentration of species C (column 2) is less than 0.15, or when the concentration of species D (column 3) is greater than 0.4, or when the derivative of the concentration of species C is less than zero.

When you have multiple stopping criteria, as in this example, you can specify either OR stopping mode or AND stopping mode using the mode parameter of the /STOP flag. If *mode* is 0, OR mode is applied - any of the conditions with a non-zero flag will stop the integration. If *mode* is 1, AND mode is applied - all conditions with a non-zero flag must be satisfied in order for the integration to be stopped.

The second way to stop the integration is by returning a value of 1 from the derivative function. You can apply any condition you like in the function so it is possible to make much more complex stopping conditions this way than using the /STOP flag. However, the derivative function is called for a many intermediate points during a single step, some of which aren't necessarily even on the eventual solution trajectory. That means that you could be applying your stopping criterion to values that are not meaningful to the final solution. That may be particularly true at a time when the internal step size is contracting - the derivative func-

tion may be called for points beyond the eventual solution point as the solver tries a step size that doesn't succeed.

Integrating a User Function

You can use the Integrate1D function to numerically integrate a user function. For example, if you want to evaluate the integral

$$I(a,b) = \int_a^b \exp[-x^3 \sin(2\pi/x^2)]dx,$$

you need to start by defining the user function

```
Function userFunc(v)
   Variable v

   return exp(-v^3*sin(2*pi/v^2))
End
```

The Integrate1D function supports three integration methods: Trapezoidal, Romberg and Gaussian Quadrature.

```
Printf "%.10f\r" Integrate1D(userfunc,0.1,0.5,0)      // default trapezoidal
0.3990547412
Printf "%.10f\r" Integrate1D(userfunc,0.1,0.5,1)      // Romberg
0.3996269165
Printf "%.10f\r" Integrate1D(userfunc,0.1,0.5,2,100)  // Gaussian Q.
0.3990546953
```

For comparison, you can also execute:

```
Make/O/N=1000 tmp
Setscale/I x,.1,.5,"" tmp
tmp=userfunc(x)
Printf "%.10f\r" area(tmp,-inf,inf)
0.3990545084
```

Integrate1D can also handle complex valued functions. For example, if you want to evaluate the integral

$$I(a,b) = \int_a^b \exp\{ix\sin x\}dx,$$

a possible user function could be

```
Function/C cUserFunc(v)
   Variable v
   Variable/C arg=cmplx(0,v*sin(v))
   return exp(arg)
End
```

Note that the user function is declared with a /C flag and that Integrate1D must be assigned to a complex number in order for it to accept a complex user function.

```
Variable/C complexResult=Integrate1D(cUserFunc,0.1,0.2,1)
print complexResult
(0.0999693,0.00232272)
```

Also note that if you just try to print the result without using a complex variable as shown above, you need to use the /C flag with print:

```
Print/C Integrate1D(cUserFunc,0.1,0.2,1)
```

in order to force the function to integrate a complex valued expression.

You can also evaluate multidimensional integrals with the help of Integrate1D. The trick is in recognizing the fact that the user function can itself return a 1D integral of another function which in turn can return a 1D integral of a third function and so on. Here is an example of integrating a 2D function: $f(x,y) = 2x + 3y + xy$.

```
Function do2dIntegration(xmin,xmax,ymin,ymax)
   Variable xmin,xmax,ymin,ymax
   Variable/G globalXmin=xmin
   Variable/G globalXmax=xmax
   Variable/G globalY
   return Integrate1D(userFunction2,ymin,ymax,1)
End

Function userFunction1(inX)
   Variable inX
   NVAR globalY=globalY
   return (3*inX+2*globalY+inX*globalY)
End

Function userFunction2(inY)
   Variable inY
   NVAR globalY=globalY
   globalY=inY
   NVAR globalXmin=globalXmin
   NVAR globalXmax=globalXmax
   return Integrate1D(userFunction1,globalXmin,globalXmax,1)
End
```

Finding Function Roots

Igor has the ability to find roots or zeros of a nonlinear function, a system of nonlinear functions, or of a polynomial with real coefficients. You do this on the command line with the **FindRoots** operation (see page V-175).

Here we discuss how the operation works, and give some examples. The discussion falls naturally into three sections: polynomial roots, roots of 1D nonlinear functions, and roots of systems of multidimensional nonlinear functions.

Igor's FindRoots operation finds function zeroes. Naturally, you can find other solutions as well. If you have a function f(x) and you want to find the X values that result in f(x) = 1, you would find roots of the function g(x) = f(x)-1. The FindRoots operation provides the /Z flag to make this more convenient. See **FindRoots** on page V-175 for more information.

Another related problem is to find places in a curve defined by data points where the data pass through zero (or another value). In this case, you don't have an analytical expression of the function. For this, use either the **FindLevel** operation (see page V-170) or the **FindLevels** operation (see page V-171); applications of these operations are discussed under **Level Detection** on page III-252.

Roots of Polynomials with Real Coefficients

The FindRoots operation can find all the complex roots of a polynomial with real coefficients. As an example, we will find the roots of

$x^4 - 3.75x^2 - 1.25x + 1.5$

We just happen to know that this polynomial can be factored as (x+1)(x-2)(x+1.5)(x-0.5) so we already know what the roots are. But let's use Igor to do the work.

First, we need to make a wave with the polynomial coefficients. The wave must have N+1 points, where N is the degree of the polynomial. Point zero is the coefficient for the constant term, the last point is the coefficient for the highest-order term:

```
Make/D/O PolyCoefs = {1.5, -1.25, -3.75, 0, 1}
```

Point	wave0
0	1.5
1	-1.25
2	-3.75
3	0
4	1
5	

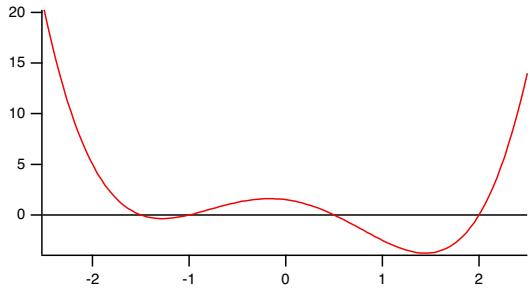
Alternatively, you might make this wave by typing in a table. Choose New Table from the Windows menu and type in the upper-left cell.

Change the name of the wave by pressing Control (*Macintosh*) or Ctrl (*Windows*) and clicking in the title cell of the wave, where you see "wave0" in the illustration above. Then type your new name in the resulting dialog.

This wave can be used with the **poly** function (see page V-486) to generate polynomial values. For instance:

```
Make/D/O PWave                  // a wave with 128 points
SetScale/I x -2.5,2.5,PWave     // give it an X range of (-2.5, 2.5)
PWave = Poly(PolyCoefs, x)      // fill it with polynomial values

Display PWave                   // and make a graph of it
ModifyGraph zero(left)=1        // add a zero line to show the roots
```

These commands make the following graph:



Now use FindRoots to find the roots:

```
FindRoots/P=PolyCoefs   // roots are now in W_polyRoots
edit W_polyRoots        // display the roots in a table
```

The table shows that Igor has found the roots that we expected:

Point	W_polyRoots.d.r	W_polyRoots.d.i
0	0.5	0
1	-1	0
2	-1.5	0
3	2	0
4		

Note that the imaginary part of the roots are zero, because this polynomial was constructed from real factors. In general, this won't be the case:

```
Make/D/O PolyCoefs2={1,2,3,4}
FindRoots/P=PolyCoefs2
```

Point	W_polyRoots.d.r	W_polyRoots.d.i
0	-0.0720852	0.638327
1	-0.0720852	-0.638327
2	-0.60583	0
3		

The FindRoots operation uses the Jenkins-Traub algorithm for finding roots of polynomials:

Jenkins, M.A., "Algorithm 493, Zeros of a Real Polynomial", *ACM Transactions on Mathematical Software*, *1*, 178-189, 1975, used by permission of ACM (1998).

Roots of a 1D Nonlinear Function

Unlike the case with polynomials, there is no general method for finding all the roots of a nonlinear function. Igor searches for a root of the function using Brent's method, and, depending on circumstances will find one or two roots in one shot.

You must write a user-defined function to define the function whose roots you want to find. Igor will call your function with values of X in the process of searching for a root. The format of the function is as follows:

```
Function myFunc(w,x)
   Wave w
   Variable x

   return <an arithmetic expression>
End
```

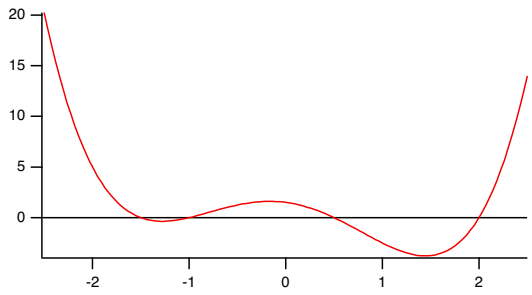
The wave *w* is a coefficient wave — it specifies constant coefficients that you may need to include in the function. It provides a convenient way to alter the coefficients so that you can find roots of members of a function family without having to edit your function code every time. Igor will not alter the values in w.

As an example we will find roots of the function y = a+b*sinc(c*(x-x0)). Here is a user-defined function to implement this:

```
Function mySinc(w, x)
   Wave w
   Variable x

   return w[0]+w[1]*sinc(w[2]*(x-w[3]))
End
```

Enter this code into the Procedure window (display the Procedure window by choosing Procedure Window from the Procedure Windows submenu of the Windows menu), then close the Procedure window. You can make a graph of the function:

```
Make/D/O SincCoefs={0, 1, 2, .5}     // a sinc function offset by 0.5
Make/D/O SincWave                    // a wave with 128 points
SetScale/I x -10,10,SincWave         // give it an X range of (-2, 2)
SincWave = mySinc(SincCoefs, x)      // fill it with function values

Display SincWave                     // and make a graph of it
ModifyGraph zero(left)=1             // add a zero line to show the roots
ModifyGraph minor(bottom)=1          // add minor ticks to the graph
```



Now we're ready to find roots. The algorithm for finding roots requires that the roots first be bracketed, that is, you need to know two X values that are on either side of the root (that is, that give Y values of opposite sign). Making a graph of the function as we did here is a good way to figure out bracketing values. For instance, inspection of the graph above shows that there is a root between x=1 and x=3. The FindRoots command line to find a root in this range is

```
FindRoots/L=1/H=3 mySinc, SincCoefs
```

The /L flag sets the lower bracket and the /H flag sets the upper bracket. Igor then finds the root between these values, and prints the results in the history:

```
Possible root found at 2.0708
Y value there is -1.46076e-09
```

Igor reports to you both the root (in this case 2.0708) and the Y value at the root, which should be very close to zero (in this case it is -1.46×10^{-9}). Some pathological functions can fool Igor into thinking it has found a root when it hasn't. The Y value at the supposed root should identify if this has happened.

The bracketing values don't actually have to be at points of opposite sign. If the bracket encloses an extreme point, Igor will find it and then find two roots. Thus, you might use this command:

```
FindRoots/L=0/H=5 mySinc, SincCoefs
```

The Y values at X=0 and X=5 are both positive. Igor finds the minimum point at about X=2.7 and then uses that with the original bracketing values as the starting points for finding two roots. The result, printed in the history:

```
Looking for two roots...

Results for first root:
Possible root found at 2.0708
Y value there is -2.99484e-11

Results for second root:
Possible root found at 3.64159
Y value there is 3.43031e-11
```

Finally, it isn't always necessary to provide bracketing values. If the /L and /H flags are absent, Igor assigns them the values 0 and 1. In this case, there is no root between X=0 and X=1, and there is no extreme point. So Igor searches outward in a series of expanding jumps looking for values that will bracket a root (that is, for X values having Y values of opposite sign). Thus, in this case, the following simple command works:

```
FindRoots mySinc, SincCoefs
```

Igor finds the first of the roots we found previously:

```
Possible root found at 2.0708
Y value there is 2.748e-13
```

You may have noticed by now that FindRoots is reporting Y values that are merely small instead of zero. It isn't usually possible to find exact roots with a computer. And asking for very high accuracy requires more iterations of the search algorithm. If function evaluation is time-consuming and you don't need much accuracy, you may not want to find the root with high accuracy. Consequently, you can use the /T flag to alter the acceptable accuracy:

```
FindRoots/T=1e-3 mySinc, SincCoefs     // low accuracy
Possible root found at 2.07088
   Y value there is -5.5659e-05
Print/D V_root, V_YatRoot
   2.07088376061435  -5.56589998578327e-05

FindRoots/T=1e-15 mySinc, SincCoefs    // high accuracy
Possible root found at 2.0708
   Y value there is 0
Print/D V_root, V_YatRoot
   2.0707963267949  0
```

This also illustrates another point: the results of FindRoots are stored in variables. We used these variables in this case to print the results to higher accuracy than the six digits used by the report printed by FindRoots.

Roots of a System of Multidimensional Nonlinear Functions

Finding roots of a system of multidimensional nonlinear functions works very similarly to finding roots of a 1D nonlinear function. You provide user-defined functions that define your functions. These functions have nearly the same form as a 1D function, but they have a parameter for each independent variable. For instance, if you are going to find roots of a pair of 2D functions, the functions will look like this:

```
Function myFunc1(w, x1, x2)
   Wave w
   Variable x1, x2

   return <an arithmetic expression>
End

Function myFunc2(w, x1, x2)
   Wave w
   Variable x1, x2

   return <an arithmetic expression>
End
```

These function look just like the 1D function mySinc we wrote in the previous section, but they have two input X variables, one for each dimension. The number of functions must match the number of dimensions.

We will use the functions f1= w[0]*sin((x-3)/w[1])*cos(y/w[2]) and f2 = w[0]*cos(x/w[1])*tan((y+5)/w[2]). Enter this code into your procedure window:

```
Function myf1(w, xx, yy)
   Wave w
   Variable xx,yy

   return w[0]*sin(xx/w[1])*cos(yy/w[2])
End

Function myf2(w, xx, yy)
   Wave w
   Variable xx,yy

   return w[0]*cos(xx/w[1])*tan(yy/w[2])
End
```

Before starting, let's make a contour plot to see what we're up against. Here are some commands to make a convenient one:

```
Make/D/O params2D={1,5,4}           // nice set of parameters for both f1 and f2
Make/D/O/N=(50,50) f1Wave, f2Wave   // matrix waves for contouring
SetScale/I x -20,20,f1Wave, f2Wave// nice range of X values for these functions
SetScale/I y -20,20,f1Wave, f2Wave  // and Y values
f1Wave = myf1(params2D, x, y)       // fill f1Wave with values from f1(x,y)
f2Wave = myf2(params2D, x, y)       // fill f2Wave with values from f2(x,y)
Display /W=(5,42,399,396)           // graph window for contour plot
AppendMatrixContour f1Wave
AppendMatrixContour f2Wave
ModifyContour f2Wave labels=0    // suppress contour labels to reduce clutter
ModifyContour f1Wave labels=0
ModifyContour f1Wave rgbLines=(65535,0,0)    // make f1 red
ModifyContour f2Wave rgbLines=(0,0,65535)    // make f2 blue
ModifyGraph lsize('f2Wave=0')=2              // make zero contours heavy
ModifyGraph lsize('f1Wave=0')=2
```

Places where the zero contours for the two functions cross are the roots we are looking for. In the contour plot you can see several, for instance the points (0,0) and (7.8, 6.4) are approximate roots.

The algorithm that searches for roots needs a starting point. You can specify this in the FindRoots command with the /X flag, or if you don't use /X, Igor will start by default at the origin, $X_n = 0$. You must also specify both functions and a coefficient wave for each function. In this case we will use the same coefficient wave

for each. The functions and coefficient waves are specified in pairs. Since we are looking for roots of two 2D functions, we have two function-wave pairs:

```
FindRoots myf1,params2D, myf2,params2D
```

Igor finds a root at the origin, and prints the results. The X,Y coordinates of the root are stored in the wave W_Root:

```
   Root found after 4 function evaluations.
W_Root={0,0}
   Function values at root:
W_YatRoot={0,0}
```

The wave W_YatRoot holds the values of each of the functions evaluated at the root.

If that's not the root you want to find, use /X to specify a different starting point:

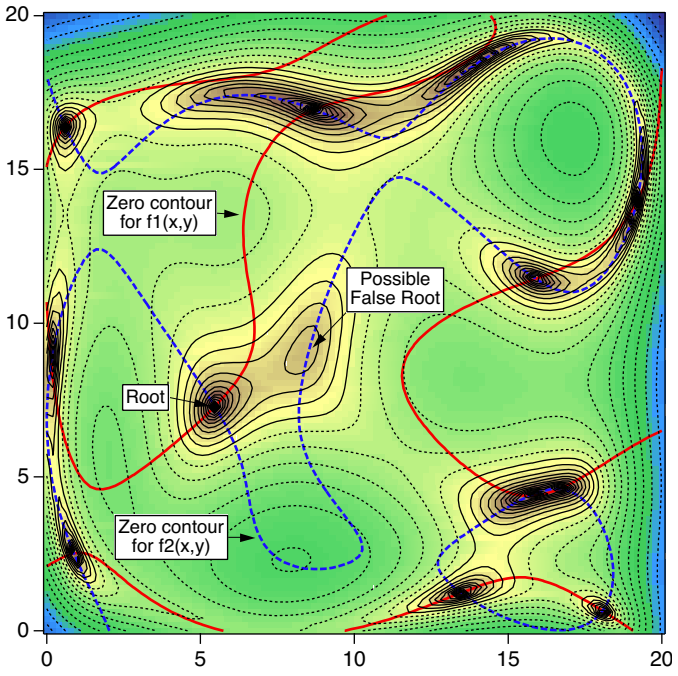
```
   FindRoots/X={7.7,6.3} myf1,params2D, myf2,params2D
   Root found after 47 function evaluations.
W_Root={-1.10261e-14,12.5664}
   Function values at root:
W_YatRoot={2.20522e-15,-1.89882e-15}
```

Caveats for Multidimensional Root Finding

Finding roots of multidimensional nonlinear functions is not straightforward. There is no general, foolproof way to do it. The method Igor uses is to search for minima in the sum of the squares of the functions. Since the squared values must be positive, the only places where this sum can be zero is at points where all the functions are zero at the same time. That point is a root, and it is also a minimum in the summed squares of the functions.

To find the zero points, Igor searches for local minima by travelling downhill from the starting point. Unfortunately, a local minimum doesn't have to be a root, it just has to be someplace where the sum of squares of the functions is less than surrounding points.

The adjacent graph shows how this can happen.



The two heavy lines are the zero contours for two functions (they happen to be fifth-order 2D polynomials). Where these zero contours cross are the roots for the system of the two functions.

The thin lines are contours of $f1(x,y)^2+f2(x,y)^2$, with dotted lines for high values; minima are surrounded by thin, solid contours. You can see that every intersection between the heavy zero contours is surrounded by thin contours showing that these are minima in the sum of the squared functions. One such point is labeled "Root".

There is at least one point, labelled "False Root", where there is a minimum but the zero contours don't cross. That is not a root, but FindRoots may find it anyway. For instance, a real root:

```
•findroots /x={3,6} MyPoly2d, nn1coefs, MyPoly2d, nn2coefs
   Root found after 11 function evaluations.
W_Root={5.4623,7.28975}
   Function values at root:
W_YatRoot={-4.15845e-13,1.08297e-12}
```

This point is the point marked "Root". However:

```
•findroots/x={9,10} MyPoly2d, nn1coefs, MyPoly2d, nn2coefs
   Root found after 52 function evaluations.
W_Root={8.38701,9.10129}
   Function values at root:
W_YatRoot={0.0686792,0.0129881}
```

You can see from the values in W_YatRoot that this is not a root. This point is marked "False root" on the figure above.

The polynomials used in this example have too many coefficients to be conveniently shown here. To see this example and others in action, try out the demo experiment. It is called "MD Root Finder Demo" and you will find it in your Igor Pro folder, in the Examples:Analysis: folder.

Finding Minima and Maxima of Functions

Igor has the ability to find extreme points (maxima or minima) of a nonlinear function. You do this on the command line with the **Optimize** operation (see page V-466).

Here we discuss how the operation works, and give some examples. The discussion falls naturally into two sections: extrema of 1D nonlinear functions and extrema of multidimensional nonlinear functions.

Another related problem is to find peaks or troughs in a curve defined by data points. In this case, you don't have an analytical expression of the function. To do this with one dimensional data, use the **FindPeak** operation (see page V-173).

Extreme Points of a 1D Nonlinear Function

The Optimize operation finds local maxima or minima of functions. That is, if a function has some X value where the nearby Y values are all higher than at that X value, it is deemed to be a minimum. Finding the point where a functions value is lower or higher than any other point anywhere is a much more difficult problem that is not addressed by the Optimize operation.

You must write a user-defined function to define the function for which the extreme points are calculated. Igor will call your function with values of X in the process of searching for a root. The format of the function is as follows:

```
Function myFunc(w,x)
   Wave w
   Variable x

   return f(x)          // an expression...
End
```

The wave w is a coefficient wave — it specifies constant coefficients that you may need to include in the function. It provides a convenient way to alter the coefficients so that you can find extreme points of members of a function family without having to edit your function code every time. Igor will not alter the values in *w*.

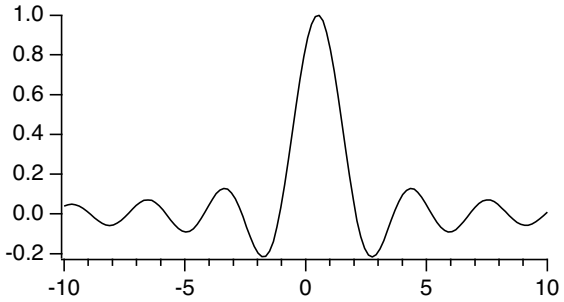
Although the coefficient wave must be present in the Function declaration, it does not have to be referenced in the function body. This may save computation time, arriving at the solution faster. You will have to create a dummy wave to list in the FindRoots command.

As an example we will find extreme points of the equation y = a+b*sinc(c*(x-x0)). A suitable user-defined function might look like this (Note that this is the same function used as an example in the section **Roots of a 1D Nonlinear Function** on page III-285. If you have just completed that example, you may already have the function and graph ready to go):

```
Function mySinc(w, x)
   Wave w
   Variable x

   return w[0]+w[1]*sinc(w[2]*(x-w[3]))
End
```

Copy this code into the Procedure window (display the Procedure window by choosing Procedure Window from the Procedure Windows submenu of the Windows menu), then close the window. You can make a graph of the function:

```
Make/D/O SincCoefs={0, 1, 2, .5} // a sinc function offset by 0.5
Make/D/O SincWave                // a wave with 128 points
SetScale/I x -10,10,SincWave     // give it an X range of [-10, 10]
SincWave = mySinc(SincCoefs, x)  // fill it with function values

Display SincWave                 // and make a graph of it
ModifyGraph minor(bottom)=1      // add minor ticks to the graph
```



Now we're ready to find extreme points. The algorithm for finding extreme points requires that the extreme points first be bracketed, that is, you need to know two X values that are on either side of the extreme point (that is, two points that have a lower or higher point between). Making a graph of the function as we did here is a good way to figure out bracketing values. For instance, inspection of the graph above shows that there is a minimum between x=1 and x=4. The Optimize command line to find a minimum in this range is

```
Optimize/L=1/H=4 mySinc, SincCoefs
```

The /L flag sets the lower bracket and the /H flag sets the upper bracket. Igor then finds the minimum between these values, and prints the results in the history:

```
Optimize probably found a minimum. Optimize stopped because
the Optimize operation found a minimum within the specified tolerance.
Current best solution: 2.7467
Function value at solution: -0.217234
 13 iterations, 14 function calls
V_minloc = 2.7467, V_min = -0.217234, V_OptNumIters = 13, V_OptNumFunctionCalls = 14
```

Igor reports to you both the X value that minimizes the function (in this case 2.7467) and the Y value at the minimum.

The bracketing values don't necessarily have to bracket the solution. Igor first tries to find the desired extremum between the bracketing values. If it fails, the bracketing interval is expanded searching for a suitable bracketing interval. If you don't use /L and /H, Igor sets the bracketing interval to [0,1]. In the case of the mySinc function, that doesn't include a minimum. Here is what happens in that case:

```
Optimize mySinc, SincCoefs

Optimize probably found a minimum. Optimize stopped because
the Optimize operation found a minimum within the specified tolerance.
Current best solution: 2.74671
Function value at solution: -0.217234
 16 iterations, 47 function calls
V_minloc = 2.74671, V_min = -0.217234, V_OptNumIters = 16, V_OptNumFunctionCalls = 47
```

Note that Igor found the same minimum that it found before.

The mySinc function makes it easy to find bracketing values because of the oscillatory nature of the function. Other functions may be more difficult if they contain just one extreme point, or if they have local extreme points but are unbounded elsewhere. Even in an easy case like mySinc, you can't be sure which extreme point Igor will find, so it is always better to supply a good bracket if you possibly can.

You may wish to find maximum points instead of minima. Use the /A flag to specify this:

```
Optimize/A/L=0/H=2 mySinc, SincCoefs

Optimize probably found a maximum. Optimize stopped because
the Optimize operation found a maximum within the specified tolerance.
Current best solution: 0.499999
Function value at solution: 1
 16 iterations, 17 function calls
V_maxloc = 0.499999, V_max = 1, V_OptNumIters = 16, V_OptNumFunctionCalls = 17
```

The results of the Optimize operation are stored in variables. Note that the report that Optimize prints in the history includes only six digits of the values. You can print the results to greater precision using the printf operation and the variables:

```
Printf "The max is at %.15g. The Y value there is %.15g\r", V_maxloc, V_max
```

yields this in the history:

```
The max is at 0.499999480759779. The Y value there is 0.99999999999982
```

Extrema of Multidimensional Nonlinear Functions

Finding extreme points of multidimensional nonlinear functions works very similarly to finding extreme points of a 1D nonlinear function. You provide a user-defined function having almost the same format as for 1D functions. A 2D function will look like this:

```
Function myFunc1(w, x1, x2)
   Wave w
   Variable x1, x2

   return f1(x1, x2)       // an expression...
End
```

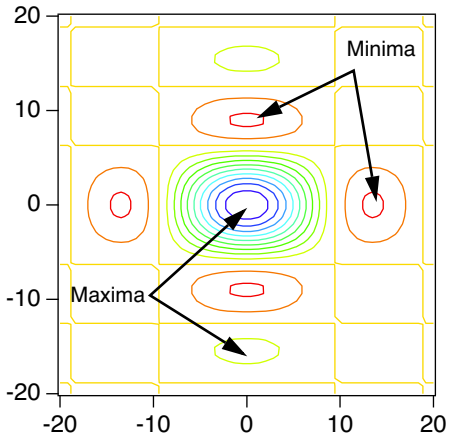
This function looks just like the 1D function mySinc we wrote in the previous section, but it has two input X variables, one for each dimension.

We will make a 2D function based on the sinc function. Copy this code into your procedure window:

```
Function Sinc2D(w, xx, yy)
   Wave w
   Variable xx,yy

   return w[0]*sinc(xx/w[1])*sinc(yy/w[2])
End
```

Before starting, let's make a contour plot to see what we're up against. Here are some commands to make a convenient one:

```
Make/D/O params2D={1,3,2}         // nice set of parameters
Make/D/O/N=(50,50) Sinc2DWave     // matrix wave for contouring
SetScale/I x -20,20,Sinc2DWave    // nice range of X values for these functions
SetScale/I y -20,20,Sinc2DWave    // and Y values
Sinc2DWave = Sinc2D(params2D, x, y) // fill f1Wave with values from f1(x,y)
Display /W=(5,42,399,396)         // graph window for contour plot
AppendMatrixContour Sinc2DWave
ModifyContour Sinc2DWave labels=0// suppress contour labels to reduce clutter
```



The algorithm that searches for extreme points needs a starting guess which you provide using the /X flag. Here is a command to find a minimum point:

```
Optimize/X={1,5} Sinc2D,params2D
```

This command results in the following report in the history:

```
Optimize probably found a minimum. Optimize stopped because
the gradient is nearly zero.
 Current best solution:
W_Extremum={-0.000147116,8.98677}
 Function gradient:
W_OptGradient={-4.65661e-08,1.11923e-08}
 Function value at solution: -0.217234
 11 iterations, 36 function calls
 V_min = -0.217234, V_OptTermCode = 1, V_OptNumIters = 11, V_OptNumFunctionCalls = 36
```

Note that the minimum is reported via a wave called W_extremum. Had you specified a wave using the /X flag, that wave would have been used instead. Another wave, W_OptGradient, receives the function gradient at the solution point.

There are quite a few options available to modify the workings of the Optimize operation. See **Optimize** operation on page V-466 for details, including references to reading material.

Stopping Tolerances

The Optimize operation stops refining the solution when certain criteria are met. The most desirable result is that it stops because the function gradient is very close to zero, since that is (almost) diagnostic of an extreme point. The algorithm also will take very small steps near an extreme point, so this is also a stopping criterion. You can set the values for the stopping criteria using the /T={*gradTol*, *stepTol*} flag.

Optimize stops when these conditions are met:

$$\max_{1 \le i \le n} \left\{ |g_i| \cdot \frac{max(|x_i|, typX_i)}{max(|f|, funcSize)} \right\} \le gradTol$$

or

$$\max_{1 \le i \le n} \left\{ \frac{|\Delta x_i|}{max(x_i, typX_i)} \right\} \le stepTol$$

Note that these conditions use values of the gradient (g_i) and step size (Δx_i) that are scaled by a measure of the magnitude of values encountered in the problem. In these equations, x_i is the value of a component of the solution and f is the value of the function at the solution; $typX_i$ is a "typical" value of the X component that is set by you using the /R flag and f is a typical function value magnitude which you set using the /Y flag. The values of $typX_i$ and f are one if the /R and /F flags are not present.

The default values for *gradTol* and *stepTol* are {8.53618×10^{-6}, 7.28664×10^{-11}}. These are the values recommended by Dennis and Schnabel (see the references in **Optimize** operation on page V-466) for well-behaved functions when the function values have full double precision resolution. These values are $(6.022 \times 10^{-16})^{1/3}$ and $(6.022 \times 10^{-16})^{2/3}$ as suggested by Dennis and Schnabel (see the references in **Optimize** operation on page V-466), where 6.022×10^{-16} is the smallest double precision floating point number that, when added to 1, is different from 1. Usually the default is pretty good.

Due to floating point truncation errors, it is possible to set *gradTol* and *stepTol* to values that can never be achieved. In that case you may get a message about "no solution was found that is better than the last iteration".

Problems with Multidimensional Optimization

Finding minima of multidimensional functions is by no means foolproof. The methods used by the Optimize operation are "globally convergent" which means that under suitable circumstances Optimize will be able to find some extreme point from just about any starting guess.

If the gradient of your function is zero, or very nearly so at the starting guess, Optimize has no information on which way to go to find an extreme point. Note that the Sinc2D function has a maximum exactly at (0,0). Here is what happens if you try to find a minimum starting at the origin:

```
Optimize/X={0,0} Sinc2D,params2D

==== The Optimize operation failed to find a minimum. ====
Optimize stopped because
The function gradient at your starting guess is too near zero, suggesting that
it is a critical point.
A different starting guess usually solves this problem.
 Current best solution:
W_Extremum={0,0}
 Function gradient:
W_OptGradient={0,0}
 Function value at solution: 1
 0 iterations, 3 function calls
 V_min = 1, V_OptTermCode = 6, V_OptNumIters = 0, V_OptNumFunctionCalls = 3
```

In this example the function gradient is zero at the origin because there is a function maximum there. A gradient of zero could also be a minimum or a saddle point.

The algorithms used by the Optimize operation assume that your function is smooth, that is, that the first and second derivatives are continuous. Optimize may work with functions that violate this assumption, but it is not guaranteed.

Although Optimize tends to look downhill to the nearest minimum (or uphill to the nearest maximum), it is not guaranteed to find any particular minimum, especially if your starting guess is near a point where the gradient is small. Sometimes using a different method (/M=(*stepMethod*, *hessianMethod*) will result in a different answer. You can limit the maximum step size to keep progress more or less local (/S=*maxStep*). If you set the maximum step size too small, however, Optimize may stop early because the maximum step size is exceeded too many times. Here is an example using the Sinc2D function. If the starting guess is near the origin, the gradient is small and the solution shoots off into the hinterlands (only a portion of the history report is shown):

```
Optimize/X={1,1} Sinc2D,params2D

W_Extremum={-61.1192,298.438}
 Function gradient:
W_OptGradient={-1.04095e-08,-1.19703e-08}
```

Use /S to limit the step size, and find a minimum nearer to the starting guess:

```
Optimize/X={1,1}/S=10 Sinc2D,params2D

W_Extremum={-0.00014911,8.9868}
 Function gradient:
W_OptGradient={9.31323e-09,5.92775e-08}
```

But if the maximum step is too small, it doesn't work:

```
Optimize/X={1,1}/S=1 Sinc2D,params2D

==== The Optimize operation failed to find a minimum. ====
Optimize stopped because
the maximum step size was exceeded in five consecutive iterations.
This can happen if the function is unbounded (there is no minimum),
or the function approaches the minimum asymptotically. It may also be that the
maximum step size (1) is too small.
```

Another way to get a solution near by is to set the initial trust region to a small value. This works if you select double dogleg or More Hebdon as the step selection method. It does not apply to the default line search method. Here is an example (note that the double dogleg method is selected using /M={1,0}):

```
Optimize/X={1,1}/M={1,0}/F=1 Sinc2D,params2D

W_Extremum={-0.000619834,8.9872}
 Function gradient:
W_OptGradient={1.09896e-07,2.9017e-08}
```

References

The IntegrateODE operation is based on routines in *Numerical Recipes*, and are used by permission:

Press, W.H., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

The Adams-Moulton and BDF methods are based on the CVODE package developed at Lawrence Livermore National Laboratory:

Cohen, Scott D., and Alan C. Hindmarsh, *CVODE User Guide*, LLNL Report UCRL-MA-118618, September 1994.

The CVODE package was derived in part from the VODE package. The parts used in Igor are described in this paper:

Brown, P.N., G. D. Byrne, and A. C. Hindmarsh, VODE, a Variable-Coefficient ODE Solver, *SIAM J. Sci. Stat. Comput.*, *10*, 1038-1051, 1989.

The Optimize operation uses Brent's method for univariate functions. *Numerical Recipes* has an excellent discussion in section 10.2 of this method (but we didn't use their code).

For multivariate functions Optimize uses code based on Dennis and Schnabel. To truly understand what Optimize does, read their book:

Dennis, J. E., Jr., and Robert B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Methods*, 378 pp., Society for Industrial and Applied Mathematics, Philadelphia, 1996.

The FindRoots operation uses the Jenkins-Traub algorithm for finding roots of polynomials:

Jenkins, M.A., "Algorithm 493, Zeros of a Real Polynomial", *ACM Transactions on Mathematical Software*, *1*, 178-189, 1975..

Chapter III-11

Image Processing

Overview

Image processing is a broad term describing most operations that you can apply to image data which may be in the form of a 2D, 3D or 4D waves. Image processing may sometimes provide the appropriate analysis tools even if the data have nothing to do with imaging. In Chapter II-15, **Image Plots**, we described operations relating to the display of images. Here we concentrate on transformations, analysis operations and special utility tools that are available for working with images.

You can use the IP Tutorial experiment (inside the Learning Aids folder in your Igor Pro folder) in parallel with this chapter. The experiment contains in addition to some introductory material, the sample images and most of the commands that appear in this chapter. To execute the commands you can select them in the Image Processing help file and press Control-Enter.

For a listing of all image analysis operations, see **Image Analysis** on page V-3.

Image Transforms

The two basic classes of image transforms are color transforms and grayscale/value transforms. Color transforms involve conversion of color information from one color space to another, conversions from color images to grayscale, and representing grayscale images with false color. Grayscale value transforms include, for example, pixel level mapping, mathematical and morphological operations.

Color Transforms

There are many standard file formats for color images. When a color image is stored as a 2D wave it either has an associated or implied colormap and the RGB value of every pixel is obtained by mapping values in the 2D wave into the colormap.

When the image is a 3D wave, each image plane corresponds to an individual red, green, or blue color component. If the image wave is of type unsigned byte (/B/U), values in each plane are in the range [0,255]. Otherwise, the range of values is [0,65535].

There are two other types of 3D image waves. The first consists of 4 layers corresponding to RGBA where the 'A' represents the alpha (transparency) channel. The second contains more than three planes in which case the planes are grayscale images that can be displayed using the command:

```
ModifyImage imageName plane=n
```

Multiple color images can be stored in a single 4D wave where each chunk corresponds to a separate RGB image.

You can find most of the tools for converting between different types of images in the **ImageTransform** operation. For example, you can convert a 2D image wave that has a colormap to a 3D RGB image wave. Here we create a 3-layer 3D wave named M_RGBOut from the 2D image named 'Red Rock' using RGB values from the colormap wave named 'Red RockCMap':

```
ImageTransform /C='Red RockCMap' cmap2rgb 'Red Rock'
NewImage M_RGBOut          // Resulting 3D wave is M_RGBOut
```

Note: The images in the IP Tutorial experiment are not stored in the root data folder, so many of the commands in the tutorial experiment include data folder paths. Here the data folder paths have been removed for easier reading. If you want to execute the commands you see here, use the commands in the IP Tutorial help window. See Chapter II-8, **Data Folders**, for more information about data folders.

In many situations it is necessary to dispose of color information and convert the image into grayscale. This usually happens when the original color image is to be processed or analyzed using grayscale operations. Here is an example using the RGB image which we have just generated:

```
ImageTransform rgb2gray M_RGBOut
NewImage M_RGB2Gray           // Display the grayscale image
```

The conversion to gray is based on the YIQ standard where the gray output wave corresponds to the Y channel:

```
gray=0.299*red+0.587*green+0.114*blue.
```

If you wish to use a different set of transformation constants say $\{c_i\}$, you can perform the conversion on the command line:

```
gray2DWave=c1*image[p][q][0]+c2*image[p][q][1]+c3*image[p][q][2]
```

For large images this operation may be slow. A more efficient approach is:

```
Make/O/N=3 scaleWave={c1,c2,c3}
ImageTransform/D=scaleWave scalePlanes image // Creates M_ScaledPlanes
ImageTransform sumPlanes M_ScaledPlanes
```

In some applications it is desirable to extract information from the color of regions in the image. We therefore convert the image from RGB to the HSL color space and then perform operations on the first plane (hue) of the resulting 3D wave. In the following example we convert the RGB image wave peppers into HSL, extract the hue plane and produce a binary image in which the red hues are nonzero.

```
ImageTransform /U rgb2hsl peppers// Note the /U for unsigned short result
MatrixOP/O RedPlane=greater(5000,M_RGB2HSL[][][0])+greater(M_RGB2HSL[][][0],60000)
NewImage RedPlane          // Here white corresponds to red hues in the source
```

As you can see, the resulting image is binary, with white pixels corresponding to regions where the original image was predominantly red. The binary image can be used to discriminate between red and other hue regions. The second command line above converts hue values that range from 0 to 65535 to 1 if the color is in the "reddish" range, or zero if it is outside that range. The selection of values below 5000 is due to the fact that red hues appear on both sides of 0° (or 360°) of the hue circle.

Hue based image segmentation is also supported through the **ImageTransform** operation (see page V-290) using the hslSegment, matchPlanes or selectColor keywords. The same operation also supports color space conversions from RGB to CIE XYZ (D65 based) and from XYZ to RGB. See also **Handling Color** on page III-323 and **HSL Segmentation** on page III-318.

Grayscale or Value Transforms

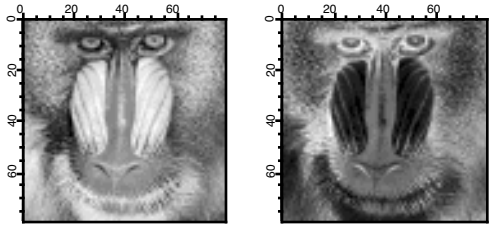
This class of transforms applies only to 2D waves or to individual layers of higher dimensional waves. They are called "grayscale" because 2D waves by themselves do not contain color information. We divide grayscale transforms into level mappings and mathematical operations.

Explicit Lookup Tables

Here is an example of using an explicit lookup table (LUT) to create the negative of an image the hard way:

```
Make/B/U/O/N=256 negativeLookup=255-x     // Create the lookup table
Duplicate/O baboon negativeBaboon
negativeBaboon=negativeLookup[negativeBaboon]// The lookup transformation
```

```
NewImage baboon
NewImage negativeBaboon
```



In this example the negativeBaboon image is a derived wave displayed with standard linear LUT. You can also obtain the same result using the original baboon image but displaying it with a negative LUT:

```
NewImage baboon
Make/N=256 negativeLookup=1-x/255   // Negative slope LUT from 1 to 0
ModifyImage baboon lookup=negativeLookup
```

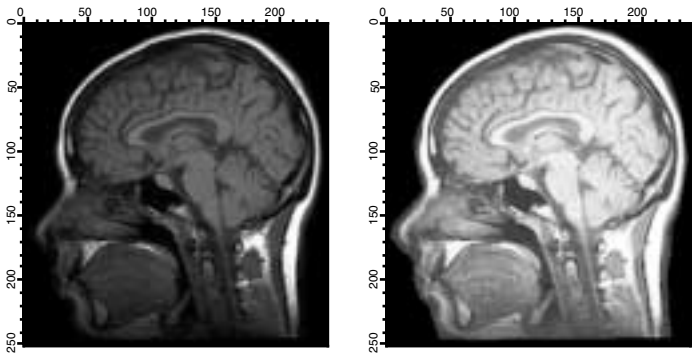
If you are willing to modify the original data you can execute:

```
ImageTransform invert baboon
```

Histogram Equalization

Histogram equalization maps the values of a grayscale image so that the resulting values utilize the entire available range of intensities:

```
NewImage MRI
ImageHistModification MRI
NewImage M_ImageHistEq
```



Adaptive Histogram Equalization

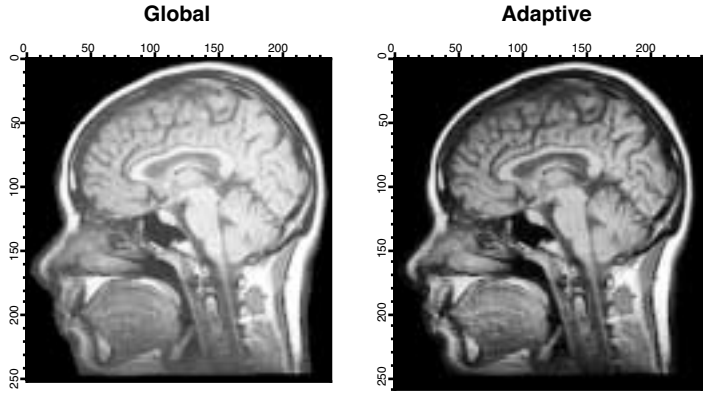
The **ImageHistModification** operation calculates a lookup table based on the cumulative histogram of the whole source image. The lookup table is then applied the output image. In cases where there are significant spatial variations in the histogram, a more local approach may be needed, i.e., perform the histogram equalization independently for different parts of the image and then combine the regional results by matching them across region boundaries. This is commonly referred to as "Adaptive Histogram Equalization".

```
ImageHistModification MRI
Duplicate/O M_ImageHistEq, globalHist
NewImage globalHist
ImageTransform/N={2,7} padImage MRI // To make the image divisible
ImageHistModification/A/C=10/H=2/V=2 M_paddedImage
NewImage M_ImageHistEq
```

The original image is 238 by 253 pixels. Because the number of rows and columns must be divisible by the number of equalization intervals, we first padded the image using the **ImageTransform padImage** operation (see page V-294). The result is an image that is 240 by 260. If you do not find the resulting adaptive his-

togram sufficiently different from the global histogram equalization, you can increase the number of vertical and horizontal regions that are processed:

```
ImageHistModification/A/C=100/H=20/V=20 M_paddedImage
```



You can now compare the global and adaptive histogram results. Note that the adaptive histogram performed better (increased contrast) over most of the image. The increase in the clipping value (/C flag) gave rise to a minor artifact around the boundary of the head.

Threshold

The threshold operation is an important member of the level mapping class. It converts a grayscale image into a binary image. A binary image in Igor is usually stored as a wave of type unsigned byte. While this may appear to be wasteful, it has advantages in terms of both speed and in allowing you to use some bits of each byte for other purposes (e.g., bits can be turned on or off for binary masking). The threshold operation, in addition to producing the binary thresholded image, can also provide a correlation value which is a measure of the threshold quality.

You can use the **ImageThreshold** operation (see page V-289) either by providing a specific threshold value or by allowing the operation to determine the threshold value for you. There are five methods for automatic threshold determination:

Iterated: Iteration over threshold levels to maximize correlation with the original image.

Bimodal: Attempts to fit a bimodal distribution to the image histogram. The threshold level is chosen between the two modal peaks.

Adaptive: Calculates a threshold for every pixel based on the last 8 pixels on the same scan line. It usually gives rise to drag lines in the direction of the scan lines. You can compensate for this artifact as we show in an example below.

Fuzzy Entropy: Considers the image as a fuzzy set of background and object pixels where every pixel may belong to a set with some probability. The algorithm obtains a threshold value by minimizing the fuzziness which is calculated using Shannon's Entropy function.

Fuzzy Means: Minimizes a fuzziness measure that is based on the product of the probability that the pixel belongs in the object and the probability that the pixel belongs to the background.

Each of the thresholding methods has its advantages and disadvantages. It is sometimes useful to try all the methods before you decide which method applies best to a particular class of images. The following example illustrates the different thresholding methods for an image of light gray blobs on a dark gray background (the "blobs" image in the IP Tutorial).

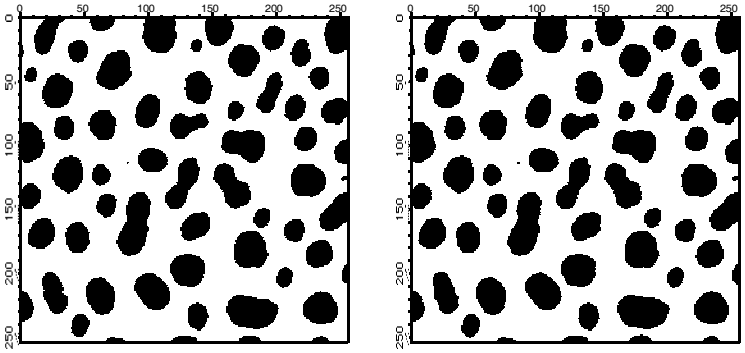
Threshold Examples

Here is a comparison of Igor's built-in threshold methods:

```
ImageThreshold/Q/T=128 blobs        // manual threshold at 128
Rename M_ImageThresh UserDefined
NewImage UserDefined
```

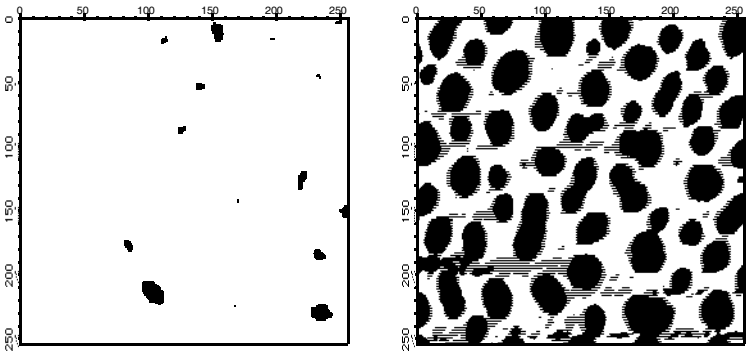
```
ImageThreshold/Q/M=1 blobs          // iterated method
Rename M_ImageThresh iterated
NewImage iterated
```

User Defined **Iterated**



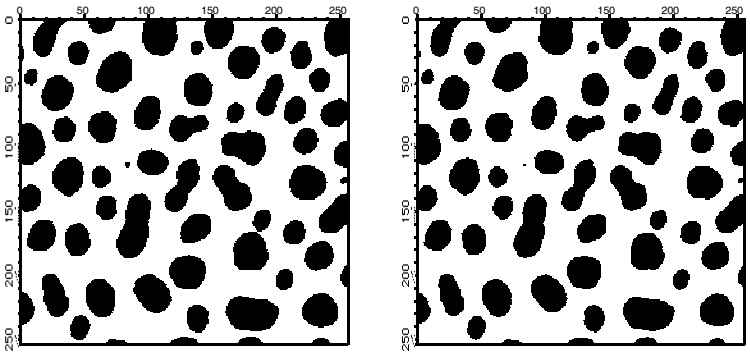
```
ImageThreshold/Q/M=2 blobs          // bimodal method
Rename M_ImageThresh bimodal
NewImage bimodal
ImageThreshold/Q/I/M=3 blobs        // adaptive method
Rename M_ImageThresh adaptive
NewImage adaptive
```

Bimodal **Adaptive**



```
ImageThreshold/Q/M=4 blobs          // fuzzy-entropy method
Rename M_ImageThresh fuzzyE
NewImage fuzzyE
ImageThreshold/Q/M=5 blobs          // fuzzy-M method
Rename M_ImageThresh fuzzyM
NewImage fuzzyM
```

fuzzyE **fuzzyM**



In the this example, you can add the /C flag to each ImageThreshold operation and remove the /Q flag to get some feedback about the quality of the threshold (the correlation coefficient will be printed to the history). It is easy to determine visually that in this case the adaptive and the bimodal algorithms performed rather poorly. Note that you can improve the results of the adaptive algorithm by running the adaptive threshold also on the transpose of the image (so that the operation becomes column based) and then combining the two outputs with a binary AND.

Spatial Transforms

Spatial transforms describe a class of operations that change the position of the data within the wave. These include the operations **ImageTransform** (with multiple keywords), **MatrixTranspose**, **ImageRotate** and **ImageRegistration**.

Rotating Images

You can rotate images using the **ImageRotate** operation (see page V-280). There are two issues that are worth noting in connection with image rotations where the rotation angle is not a multiple of 90 degrees. First, the image size is always increased to accommodate all source pixels in the rotated image (no clipping is done). The second issue is that rotated pixels are calculated using bilinear interpolation so the result of N consecutive rotations by 360/N degrees will not, in general, equal the original image. In cases of multiple rotations you should consider keeping a copy of the original image as the same source for all rotations.

Image Registration

In many situations one has two or more images of the same object where the differences between the images have to do with acquisition times, dissimilar acquisition hardware or changes in the shape of the object between exposures. To facilitate comparison between such images it is necessary to register them, i.e., to adjust them so that they match each other. The **ImageRegistration** operation (see page V-275) modifies a test image to match a reference image when the key features are not too different. The algorithm is capable of subpixel resolution but it does not handle very large offsets or large rotations. The algorithm is based on an iterative processing that proceeds from coarse to fine detail. The optimization is performed using a modified Levenberg-Marquardt algorithm and results in an affine transformation for the relative rotation and translation with optional isometric scaling and contrast adjustment. The algorithm is most effective with square images where the center of rotation is not far from the center of the image.

ImageRegistration is based on an algorithm described by Thévenaz and Unser.

Mathematical Transforms

This class of transforms includes standard wave assignments, interpolation and sampling, Fourier, Wavelet, Hough, and Hartley transforms, convolution filters, edge detectors and morphological operators.

Standard Wave Operations

Grayscale image waves are regular 2D Igor waves that can be processed using normal Igor wave assignments (**Waveform Arithmetic and Assignments** on page II-93 and **Multidimensional Wave Assignment** on page II-111). For example, you can perform simple linear operations:

```
Duplicate/O root:images:blobs sample
Redimension/S sample       // create a single precision sample
sample=10+5*sample         // linear operation
NewImage sample            // keep this image displayed
```

Note that the display of the image is invariant for this linear operation.

Nonlinear operations are just as easy:

```
sample=sample^3-sample     // not particularly useful
```

You can add noise and change the background using simple wave assignment:

```
sample=root:images:blobs   // rest to original
sample+=gnoise(20)+x+2*y   // add Gaussian noise and background plane
```

As we have shown in several examples above, it is frequently necessary to create a binary image from your data. For instance, if you want an image that is set to 255 for all pixels in the image wave sample that are between the values of 50 and 250, and set to 0 otherwise, you can use the following one line wave assignment:

```
MatrixOP/O sample=255*greater(sample,50)*greater(250,sample)
```

More Efficient Wave Operations

There are several operations in this category that are designed to improve performance of certain image calculations. For example, you can obtain one plane (layer) from a multiplane image using a wave assignment like:

```
Make/N=(512,512) newImagePlane
newImagePlane[][]=root:Images:peppers[p][q][0]
```

Alternatively, you can execute:

```
ImageTransform/P=0 getPlane root:Images:peppers
```

or

```
MatrixOp/O outWave=root:Images:peppers[][][0]
```

ImageTransform and MatrixOp are much faster for this size of image than the simple wave assignment. See **General Utilities: ImageTransform Operation** on page III-325 and **MatrixOp**.

Interpolation and Sampling

You can use the **ImageInterpolate** operation (see page V-264) as both an interpolation and sampling tool. In the following example we create an interpolated image from a portion of the MRI image. The resulting image is sampled at four times the original resolution horizontally and twice vertically.

```
NewImage root:images:MRI
ImageInterpolate /S={70,0.25,170,70,0.5,120} bilinear root:images:MRI
NewImage M_InterpolatedImage
```

As the keyword suggests, the interpolation is bilinear. You can use the same operation to sample the image. In the following example we reduce the image size by a factor of 4:

```
NewImage root:images:MRI               // display for comparison
ImageInterpolate /f={0.5,0.5} bilinear root:images:MRI
NewImage M_InterpolatedImage           // the sampled image
```

Note that in reducing the size of certain images, it may be useful to apply a blurring operation first (e.g., MatrixFilter gauss). This becomes important when the image contains thin (smaller than sample size) horizontal or vertical lines.

If the bilinear interpolation does not satisfy your requirements you can use spline interpolations of degrees 2-5. Here is a comparison between the bilinear and spline interpolation of degree 5 used to scale an image:

```
ImageInterpolate /f={1.5,1.5} bilinear MRI
Rename M_InterpolatedImage Bilinear
NewImage Bilinear
ImageInterpolate /f={1.5,1.5}/D=5 spline MRI
NewImage M_InterpolatedImage
```

Fast Fourier Transform

There are many books on the application of Fourier transforms in imaging so we will only discuss some of the technical aspects of using the **FFT** operation (see page V-156) in Igor.

It is important to keep in mind is that for historical reasons, the default FFT operation *overwrites* and *modifies* the image wave. As of Igor Pro 5 you can also specify a destination wave in the FFT operation and your source wave will be preserved. The second issue that you need to remember is that the transformed wave is converted into a complex data type and the number of points in the wave is also changed to accommodate this

conversion. The third issue is that when performing the FFT operation on a real wave the result is a one-sided spectrum, i.e., you have to obtain the rest of the spectrum by reflecting and complex-conjugating the result.

A typical application of the FFT in image processing involves transforming a real wave of 2N rows by M columns. The complex result of the FFT is (N+1) rows by M columns. If the original image wave has wave scaling of dx and dy, the new wave scaling is set to 1/(N*dx) and 1/(M*dy) respectively.

The following examples illustrate a number of typical applications of the FFT in imaging.

Calculating Convolutions

To calculate convolutions using the FFT it is necessary that the source wave and the convolution kernel wave have the same dimensions (see **MatrixOp** convolve for an alternative). Consider, for example, smoothing noise via convolution with a Gaussian:

```
// Create and display a noisy image.
Duplicate /O root:images:MRI mri    // an unsigned byte image.
Redimension/s mri                   // convert to single precision.
mri+=gnoise(10)                     // add noise.
NewImage mri
ModifyImage mri ctab= {*,*,Rainbow,0}// show the noise using false color.

// Create the filter wave.
Duplicate/O mri gResponse     // just so that we have the same size wave.
SetScale/I x -1,1,"" gResponse
SetScale/I y -1,1,"" gResponse

// Change the width of the Gaussian below to set the amount of smoothing.
gResponse=exp(-(x^2+y^2)/0.001)

// Calculate the convolution.
Duplicate/O mri processedMri
FFT processedMri                    // Transform the source
FFT gResponse                       // Transform the kernel
processedMri*=gResponse    // (complex) multiplication in frequency space
IFFT processedMri

// Swap the IFFT to properly center the result.
ImageTransform swap processedMri
Newimage processedM
ModifyImage processedMri ctab= {*,*,Rainbow,0}
```

In practice one can perform the convolution with fewer instructions. The example above has a number of commands that are designed to make it clearer. Also note that we used the **SetScale** operation (see page V-564) to create the Gaussian filter. This was done to make sure that the Gaussian was created at the center of the filter image, a choice that is compatible with the **ImageTransform swap** operation (see page V-298). This example is also not ideal because one can take advantage of the properties of the Gaussian (the Fourier transform of a Gaussian is also Gaussian) and perform the convolution as follows:

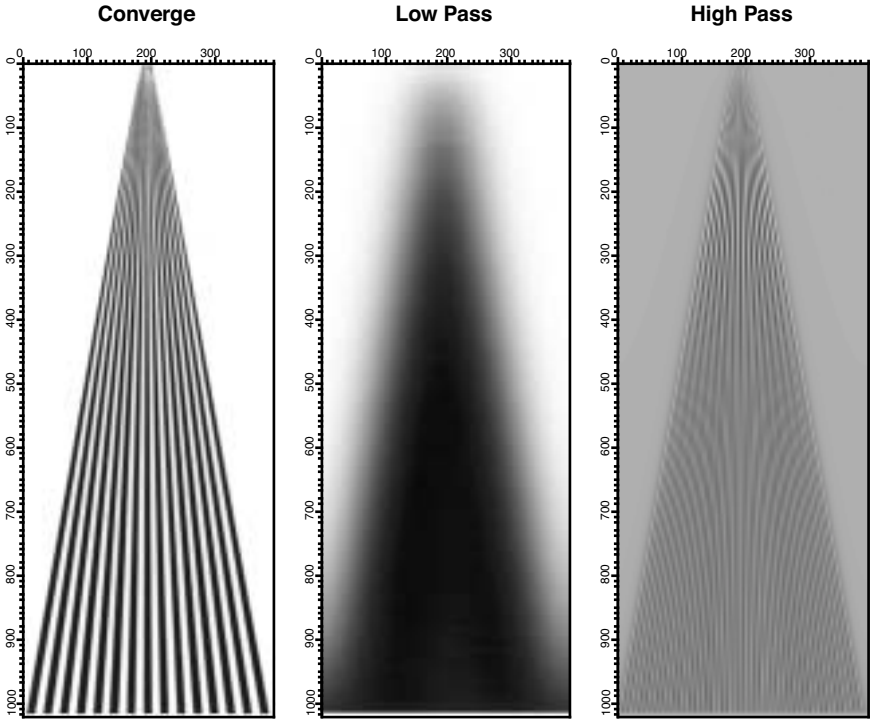
```
// Calculate the convolution.
Duplicate/O mri shortWay
FFT shortWay
shortWay*=cmplx(exp(-(x^2+y^2)/0.01),0)
IFFT shortWay
Newimage shortWay
ModifyImage shortWay ctab={*,*,Rainbow,0}
```

Spatial Frequency Filtering

The concept behind spatial frequency filtering is to transform the data into spatial frequency space. Once in frequency domain we can modify the spatial frequency distribution of the image and then inverse-transform to obtain the modified image.

Here is an example of low and high pass filtering. The converge image consists of wide black lines converging to a single point. If you draw a horizontal line profile anywhere below the middle of the image you will get a series of 15 rectangles which will give rise to a broad range of spatial frequencies in the horizontal direction.

```
// Prepare for FFT; we need SP or DP wave.
Duplicate/O root:images:converge converge
Redimension /s converge
FFT converge
Duplicate/O converge lowPass     // new complex wave in freq. domain
lowPass=lowPass*cmplx(exp(-(p)^2/5),0)
IFFT lowPass
NewImage lowPass                 // nonoptimal lowpass

Duplicate/O converge hiPass
hiPass=hiPass*cmplx(1-1/(1+(p-20)^2/2000),0)
IFFT hiPass
NewImage hiPass                  // nonoptimal highpass
```



We arbitrarily chose the Gaussian form for the low-pass filter. In practical applications it is usually important to select an exact "cutoff" frequency and at the same time choose a filter that is sufficiently smooth so that it does not give rise to undesirable filtering artifacts such as ringing, etc. The high-pass filter that we used above is almost a notch filter that rejects low frequencies. Both filters are essentially one-dimensional filters.

Calculating Derivatives

Using the derivative property of Fourier transform, you can calculate, for example, the x-derivative of an image in the following way:

```
Duplicate/O root:images:mri xDerivative   // retain the original.
Redimension/S xDerivative
FFT xDerivative
xDerivative*=cmplx(0,p)    // neglecting 2pi factor & wave scaling.
IFFT xDerivative
NewImage xDerivative
```

Although this approach may not be appealing in all applications, its advantages are apparent when you need to calculate higher order derivatives. Also note that this approach does not take into account any wave scaling that may be associated with the rows or the columns.

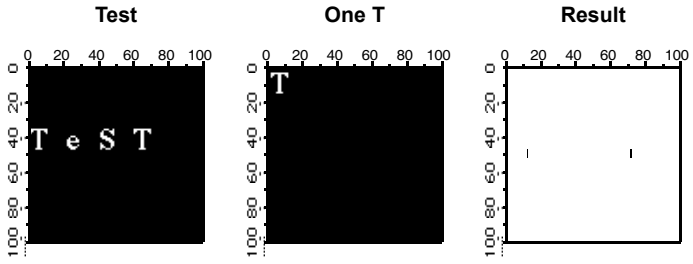
Calculating Integrals or Sums

Another useful property of the Fourier transform is that the transform values along the axes correspond to integrals of the image. There is usually no advantage in using the FFT for this purpose. However, if the FFT is calculated anyway for some other purpose, one can make use of this property. A typical situation where this is useful is in calculating correlation coefficient (normalized cross-correlation).

Correlations

The FFT can be used to locate objects of a particular size and shape in a given image. The following example is rather simple in that the test object has the same scale and rotation angle as the ones found in the image.

```
// Test image contains the word Test.
NewImage test                    // We will be looking for the two T's
Duplicate/O root:images:oneT oneT// the object we are looking for
NewImage oneT

Duplicate/O test testf           // because the FFT overwrites
FFT testf
Duplicate/O oneT oneTf
FFT oneTf
testf*=oneTf                     // not a "proper" correlation
IFFT testf
ImageThreshold/O/T=1.25e6 testf  // remove noise (due to overlap with other
characters
NewImage testf          // the results are the correlation spots for the T's
```



When using the FFT it is sometimes necessary to operate on the source image with one of the built-in window functions so that pixel values go smoothly to zero as you approach image boundaries. The **ImageWindow** operation (see page V-304) supports the Hanning, Hamming, Bartlett, Blackman, and Kaiser windows. Normally the **ImageWindow** operation (see page V-304) works directly on an image as in the following example:

```
// The redimension is required for the FFT operation anyway, so you
// might as well perform it here and reduce the quantization of the
// results in the ImageWindow operation.
Redimension/s blobs
ImageWindow /p=0.03 kaiser blobs
NewImage M_WindowedImage
```

To see what the window function looks like:

```
Redimension/S blobs                 // SP or DP waves are necessary
ImageWindow/i/p=0.01 kaiser blobs   // just creates the window data
NewImage M_WindowedImage      // you can also make a surface plot from this.
```

Wavelet Transform

The wavelet transform is used primarily for smoothing, noise reduction and lossy compression. In all cases the procedure we follow is first to transform the image, then perform some operation on the transformed wave and finally calculate the inverse transform.

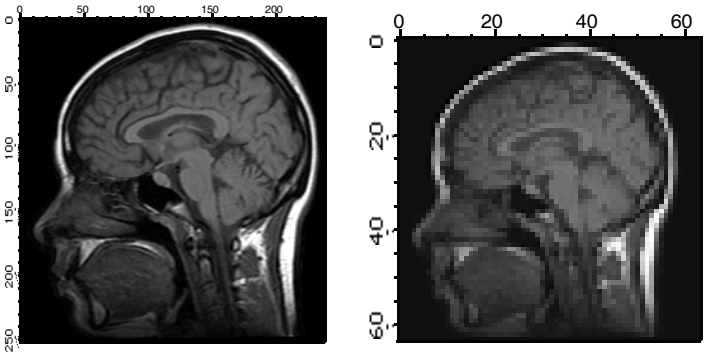
The next example illustrates a wavelet compression procedure. Start by calculating the wavelet transform of the image. Your choice of wavelet and coefficients can significantly affect compression quality. The compressed image is the part of the wave that corresponds to the low order coefficients in the transform (similar to low pass filtering in 2D Fourier transform). In this example we use the **ImageInterpolate** operation (see page V-264) to create a wave from a 64x64 portion of the transform.

```
DWT /N=4/P=1/T=1 root:images:MRI,wvl_MRI     // Wavelet transform
// reduce size by a factor of 16
Imageinterpolate/s={0,1,64,0,1,64} bilinear wvl_MRI
```

To reconstruct the image and evaluate compression quality, inverse-transform the compressed image and display the result:

```
DWT /I/N=4/P=0/T=1 M_InterpolatedImage,iwvl_compressed
NewImage iwvl_compressed
```

Original **Compressed**

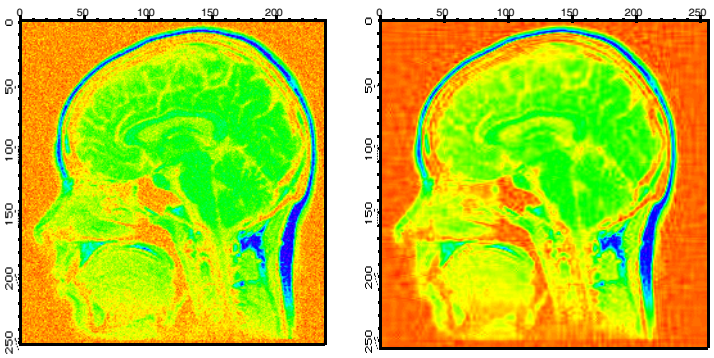


The reconstructed image exhibits a number of compression-related artifacts, but it is worth noting that unlike an FFT based low-pass filter, the advantage of the wavelet transform is that the image contains a fair amount of high spatial frequency content. The factor of 16 mentioned above is not entirely accurate because the original image was stored as a one byte per pixel while the compressed image consists of floating point values (so the true compression ratio is only 4).

To illustrate the application of the wavelet transform to denoising, we start by adding Gaussian distributed noise with standard deviation 10 to the MRI image:

```
Redimension/S Mri                          // SP so we can add bipolar noise
Mri+=gnoise(10)                            // Gaussian noise added
NewImage Mri
ModifyImage Mri ctab={*,*,Rainbow,0}// false color for better discrimination.
DWT/D/N=20/P=1/T=1/V=0.5 Mri,dMri          // increase /V for more denoising
NewImage dMri                              // display denoised image
ModifyImage dMri ctab={*,*,Rainbow,0}
```

MRI **dMRI**



Hough Transform

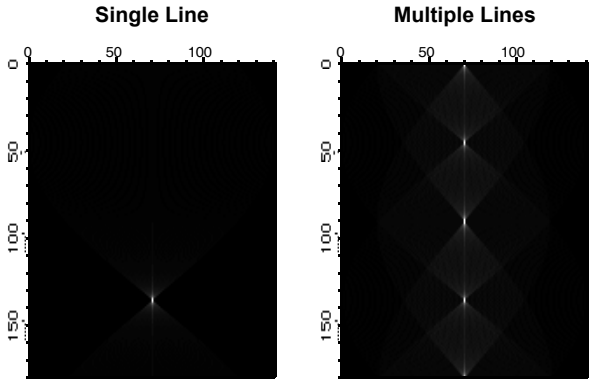
The Hough Transform is a mapping algorithm in which lines in image space map to single points in the transform space. It is most often used for line detection. Specifically, each point in the image space maps to a sinusoidal curve in the transform space. If pixels in the image lie along a line, the sinusoidal curves associated with these pixels all intersect at a single point in the transform space. By counting the number of sinusoids intersecting at each point in the transform space, lines can be detected. Here is an example of an image that consists of one line.

```
Make/O/B/U/N=(100,100) lineImage
lineImage=(p==q ? 255:0)            // single line at 45 degrees
Newimage lineimage
ImageTransform hough lineImage
Newimage M_Hough
```

The Hough transform of a family of lines:

```
lineImage=((p==100-q)|(p==q)|(p==50)|(q==50)) ? 255:0
ImageTransform Hough lineImage
```

The last image shows a series of bright pixels in the center. The first and last points correspond to lines at 0 and 180 degrees. The second point from the top corresponds to the line at 45 degrees and so on.



Fast Hartley Transform

Hartley transform is similar to the Fourier transform except that it uses only real values. The transform is based on the *cas* kernel defined by:

$$cas(vx) = \cos(vx) + \sin(vx).$$

The discrete Hartley transform is given by

$$H(u,v) = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)\left\{\cos\left(\left[2\pi\left(\frac{ux}{M}-\frac{vy}{N}\right)\right]\right)+\sin\left[2\pi\left(\frac{ux}{M}-\frac{vy}{N}\right)\right]\right\}$$

The Hartley transform has two interesting mathematical properties. First, the inverse transform is identical to the forward transform, and second, the power spectrum is given by the expression:

$$P(f) = \frac{[H(f)]^2 + [H(-f)]^2}{2}$$

The implementation of the Fast Hartley Transform is part of the **ImageTransform** operation (see page V-290). It requires that the source wave is an image whose dimensions are a power of 2.

```
ImageTransform /N={18,3}/O padImage Mri// make the image 256^2
ImageTransform fht mri
NewImage M_Hartley
```

Convolution Filters

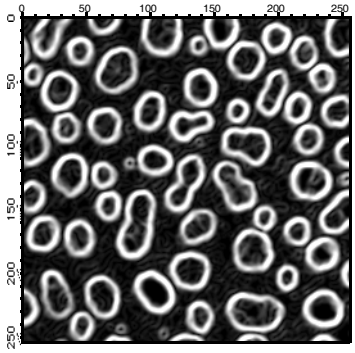
Convolution operators usually refer to a class of 2D kernels that are convolved with an image to produce a desirable effect (simple linear filtering). In some cases it is more efficient to perform convolutions using the FFT (similar to the convolution example above), i.e., transform both the image and the filter waves, multiply the transforms in the frequency domain and then compute the inverse transformation using the IFFT. The FFT approach is more efficient for convolution with kernels that are greater than 13x13 pixels. However, there is a very large number of useful kernels which play an important role in image processing that are 3x3 or 5x5 in size. Because these kernels are so small, it is fairly efficient to implement the corresponding linear filter as direct convolution without using the FFT.

In the following example we implement a low-pass filter with equal spatial frequency response along both axes.

```
Make /O/N=(9,9) sKernel       // first create the convolution kernel
SetScale/I x -4.5,4.5,"", sKernel
SetScale/I y -4.5,4.5,"", sKernel

// Equivalent to rect(2*fx)*rect(2*fy) in the spatial frequency domain.
sKernel=sinc(x/2)*sinc(y/2)

// Remember: MatrixConvolve executes in place; save your image first!
Duplicate/O root:images:MRI mri
Redimension/S mri                          // to avoid integer truncation
MatrixConvolve sKernel mri
NewImage mri
ModifyImage mri ctab= {*,*,Rainbow,0}    // just to see it better
```

The next example illustrates how to perform edge detection using a built-in convolution filter in the **ImageFilter** operation (see page V-258):

```
Duplicate/O root:images:blobs blobs
ImageFilter findEdges blobs
NewImage blobs
```



Other notable examples of image filters are Gauss, Median and Sharpen. You can also apply the same operation to 3D waves. The filters Gauss3D, avg3D, point3D, min3D, max3D and median3Dare the extensions of their 2D counterparts to 3x3x3 voxel neighborhoods. Note that the last three filters are not true *convolution* filters.

Edge Detectors

In many applications it is necessary to detect edges or boundaries of objects that appear in images. The edge detection consists of creating a binary image from a grayscale image where the pixels in the binary image are turned off or on depending on whether they belong to region boundaries or not. In other words, the detected edges are described by an image, not a vector (1D wave). If you need to obtain a wave describing boundaries of regions, you might want to use the **ImageAnalyzeParticles** operation (see page V-252).

Igor supports eight built-in edge detectors (methods) that vary in performance depending on the source image. Some methods require that you provide several parameters which tend to have a significant effect on the quality of the result. In the following examples we illustrate the importance of these choices.

```
// Create and display a simple artificial edge image.
Make/B/U/N=(100,100) edgeImage
edgeImage=(p<50? 50:5)
Newimage edgeImage

// Try a simple Sobel detector using iterated threshold detection.
ImageEdgeDetection/M=1/N Sobel, edgeImage
NewImage M_ImageEdges
ModifyImage M_ImageEdges explicit=0 // to see binary image in color
ModifyImage M_ImageEdges ctab= {*,*,Rainbow,0}
```

This result (the red line) is pretty much what we would expect. Here are other examples that work similarly well:

```
ImageEdgeDetection/M=1/N Kirsch, edgeImage   // same output wave
```

or

```
ImageEdgeDetection/M=1/N Roberts, edgeImage  // same output wave
```

The innocent looking /M=1 flag implies that the operation uses an iterative automatic thresholding. This appears to work well in the examples above, but it fails completely when using the Frei filter:

```
ImageEdgeDetection/M=1/N Frei, edgeImage
```

On the other hand, the bimodal fit thresholding works much better here:

```
ImageEdgeDetection/M=2/N Frei, root:edgeImage
```

The performance of this filter improves dramatically if you add a little noise to the image:

```
edgeImage+=gnoise(1)
ImageEdgeDetection/M=1/N/S=1 Canny, edgeImage
```

Using More Exotic Edge Detectors

The more exotic edge detectors consist of multistep operations that usually involve smoothing and differentiation. Here is an example that illustrates the effect of smoothing:

```
Duplicate/O root:images:blobs blobs
ImageEdgeDetection/M=1/N/S=1 Canny,blobs
Duplicate/O M_ImageEdges smooth1
ImageEdgeDetection/M=1/N/S=2 Canny,blobs
Duplicate/O M_ImageEdges smooth2
ImageEdgeDetection/M=1/N/S=3 Canny,blobs
Duplicate/O M_ImageEdges smooth3
NewImage smooth1
ModifyImage smooth1 explicit=0
NewImage smooth2
ModifyImage smooth2 explicit=0
NewImage smooth3
ModifyImage smooth3 explicit=0
```

As you can see, the third image (smooth3) is indeed much cleaner than the first or the second, however, that result is obtained at the cost of loosing some of the small blobs. The following commands will draw a circle around one of the blobs that is missing in the third image:

```
DoWindow/F Graph0
SetDrawLayer UserFront
SetDrawEnv linefgc= (65280,0,0),fillpat= 0
DrawOval 0.29,0.41,0.35,0.48
DoWindow/F Graph1
SetDrawLayer UserFront
SetDrawEnv linefgc= (65280,0,0),fillpat= 0
```

```
DrawOval 0.29,0.41,0.35,0.48
DoWindow/F Graph2
SetDrawLayer UserFront
SetDrawEnv linefgc= (65280,0,0),fillpat= 0
DrawOval 0.29,0.41,0.35,0.48
```

It is instructive to make a similar set of images using the Marr and Shen detectors.

```
// Note: This will take considerably longer time to execute!
Duplicate/O root:images:blobs blobs
ImageEdgeDetection/M=1/N/S=1 Marr,blobs
Duplicate/O M_ImageEdges smooth1
ImageEdgeDetection/M=1/N/S=2 Marr,blobs
Duplicate/O M_ImageEdges smooth2
ImageEdgeDetection/M=1/N/S=3 Marr,blobs
Duplicate/O M_ImageEdges smooth3
NewImage smooth1
ModifyImage smooth1 explicit=0
NewImage smooth2
ModifyImage smooth2 explicit=0
NewImage smooth3
ModifyImage smooth3 explicit=0
SetDrawLayer UserFront
SetDrawEnv linefgc= (65280,0,0),fillpat= 0
DrawOval 0.29,0.41,0.35,0.48
```

The three images of the calculated edges demonstrate the reduction of noise with the increase in the size of the convolution kernel. It's also worth noting that the blob that disappeared when we used the Canny detector is clearly visible using the Marr detector.

In the following example we use the Shen-Castan detector with various smoothing factors. Note that this edge detection algorithm does not use the standard thresholding (you have to specify the threshold using the /F flag).

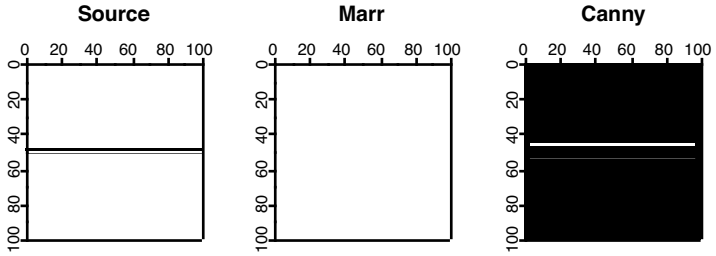
```
Duplicate/O root:images:blobs blobs
ImageEdgeDetection/N/S=0.5 shen,blobs
Duplicate/O M_ImageEdges smooth1
ImageEdgeDetection/N/S=0.75 shen,blobs
Duplicate/O M_ImageEdges smooth2
ImageEdgeDetection/N/S=0.95 shen,blobs
Duplicate/O M_ImageEdges smooth3
NewImage smooth1
ModifyImage smooth1 explicit=0
NewImage smooth2
ModifyImage smooth2 explicit=0
NewImage smooth3
ModifyImage smooth3 explicit=0
SetDrawLayer UserFront
SetDrawEnv linefgc=(65280,0,0),fillpat=0
DrawOval 0.29,0.41,0.35,0.48
```

As you can see in this example, the Shen detector produces a thin, though sometimes broken, boundary. The noise reduction is a trade-off with edge quality.

One of the problems of edge detectors that employ smoothing is that they usually introduce errors when there are two edges that are relatively close to each other. In the following example we construct an artificial image that illustrates this point:

```
Make/B/U/O/N=(100,100) sampleEdge=0
sampleEdge[][49]=255
sampleEdge[][51]=255
NewImage sampleEdge
ImageEdgeDetection/N/S=1 Marr, sampleEdge
Duplicate/O M_ImageEdges s2
```

```
NewImage s2
ImageEdgeDetection/M=1/S=3 Canny, sampleEdge
Duplicate/O M_ImageEdges s3
NewImage s3
```

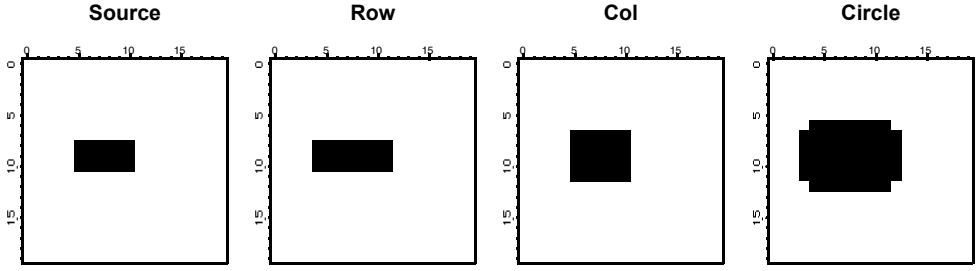


Note that the Marr detector completely misses the edge with the smoothing setting set to 1. Also, the position of the edge moves away from the true edge with increased smoothing in the Canny detector.

Morphological Operations

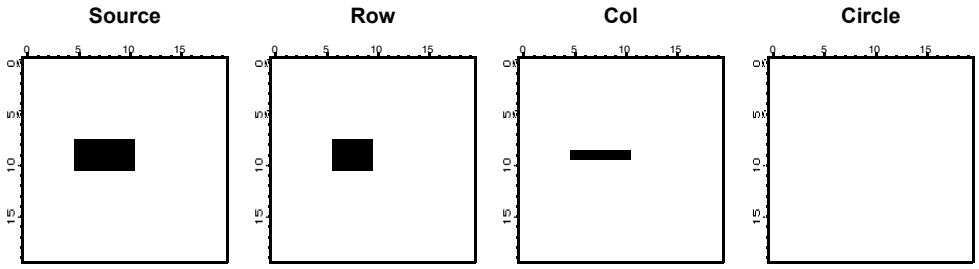
Morphological operators are tools that affect the shape and boundaries of regions in the image. Starting with dilation and erosion, the typical morphological operation involves an image and a structure element. The structure element is normally much smaller in size than the image. Dilation consists of reflecting the structure element about its origin and using it in a manner similar to a convolution mask. This can be seen in the next example:

```
Make/B/U/N=(20,20) source=0
source[5,10][8,10]=255                  // source is a filled rectangle
NewImage source
Imagemorphology /E=2 BinaryDilation source// dilation with 1x3 element
Duplicate M_ImageMorph row
NewImage row                        // display the result of dialation
Imagemorphology /E=3 BinaryDilation source// dilation by 3x1 column
Duplicate M_ImageMorph col
NewImage col                              // display column dilation
Imagemorphology /E=5 BinaryDilation source// dilation by a circle
NewImage M_ImageMorph                     // display circle dilation
```



The result of erosion is the set of pixels x, y such that when the structure element is translated by that amount it is still contained within the set.

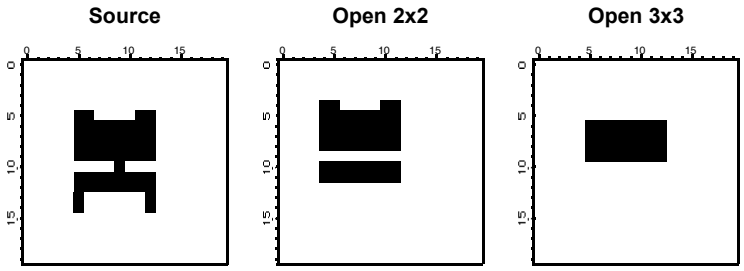
```
Make/B/U/N=(20,20) source=0
source[5,10][8,10]=255                  // source is a filled rectangle
NewImage source
Imagemorphology /E=2 BinaryErosion source// erosion with 1x3 element
Duplicate M_ImageMorph row
NewImage row                            // display the result of erosion
Imagemorphology /E=3 BinaryErosion source// erosion by 3x1 column
Duplicate M_ImageMorph col
NewImage col                            // display column erosion
Imagemorphology /E=5 BinaryErosion source// erosion by a circle
NewImage M_ImageMorph                   // display circle erosion
```



We note first that erosion by a circle erased all source pixels. We get this result because the circle structure element is a 5x5 "circle" and there is no x, y offset such that the circle is completely inside the source. The row and the col images show erosion predominantly in one direction. Again, try to imagine the 1x3 structure element (in the case of the row) sliding over the source pixels to produce the erosion.

The next pair of morphological operations are the opening and closing. Functionally, opening corresponds to an erosion of the source image by some structure element (say E), and then dilating the result using the same structure element E again. In general opening has a smoothing effect that eliminates small (narrow) protrusions as we show in the next example:

```
Make/B/U/N=(20,20) source=0
source[5,12][5,14] = 255
source[6,11][13,14] = 0
source[5,8][10,10] = 0
source[10,12][10,10] = 0
source[7,10][5,5] = 0
NewImage source
ImageMorphology /E=1 opening source // open using 2x2 structure element
Duplicate M_ImageMorph OpenE1
NewImage OpenE1
ImageMorphology /E=4 opening source // open using a 3x3 structure element
NewImage M_ImageMorph
```

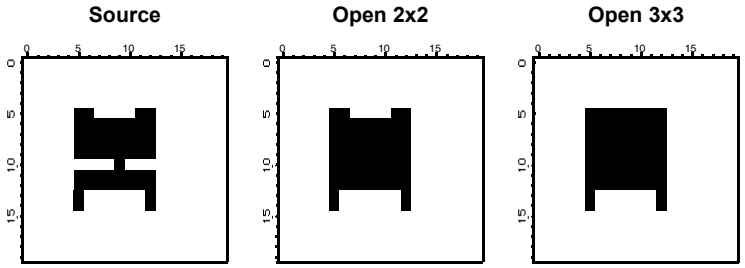


As you can see, the 2x2 structure element removed the thin connection between the top and the bottom regions as well as the two protrusions at the bottom. On the other hand, the two protrusions at the top were large enough to survive the 2x2 structure element. The third image shows the result of the 3x3 structure element which was large enough to eliminate all the protrusions but also the bottom region as well.

The closing operation corresponds to a dilation of the source image followed by an erosion using the same structure element.

```
Make/B/U/N=(20,20) source=0
source[5,12][5,14] = 255
source[6,11][13,14] = 0
source[5,8][10,10] = 0
source[10,12][10,10] = 0
source[7,10][5,5] = 0
NewImage source
ImageMorphology /E=4 closing source // close using 3x3 structure element
Duplicate M_ImageMorph CloseE4
NewImage CloseE4
```

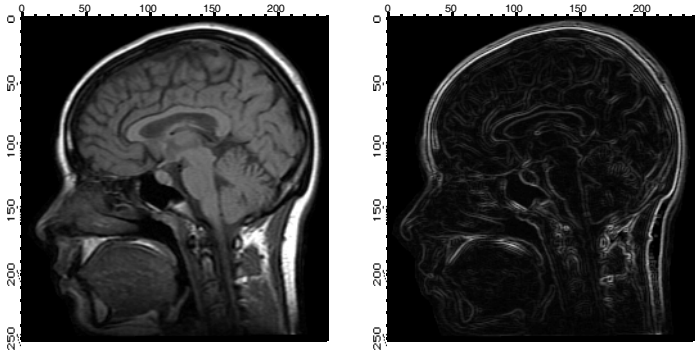
```
ImageMorphology /E=5 closing source // close using 5x5 structure element
NewImage M_ImageMorph
```

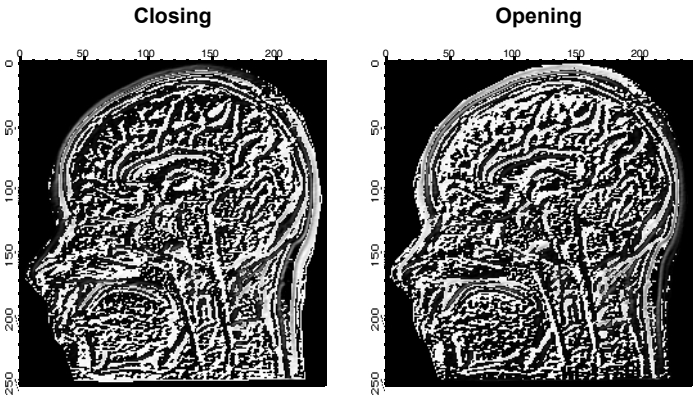


The center image above corresponds to a closing using a 3x3 structure element which appears to be large enough to close the gap between the top and bottom regions but not sufficiently large to fill the gaps between the top and bottom protrusions. The image on the right was created with a 5x5 “circle” structure element, which was evidently large enough to close the gap between the protrusions at the top but not at the bottom.

There are various definitions for the Top Hat morphological operation. Igor’s Top Hat calculates the difference between an eroded image and a dilated image. Other interpretations include calculating the difference between the image itself and its closing or opening. In the following example we illustrate some of these variations.

```
duplicate root:images:mri source
ImageMorphology /E=1 tophat source  // close using 2x2 structure element
Duplicate M_ImageMorph tophat
NewImage tophat
ImageMorphology /E=1 closing source // close using 3x3 structure element
Duplicate M_ImageMorph closing
closing-=source
NewImage closing
ImageMorphology /E=1 opening source // close using 3x3 structure element
Duplicate M_ImageMorph opening
opening=source-opening
NewImage opening
```

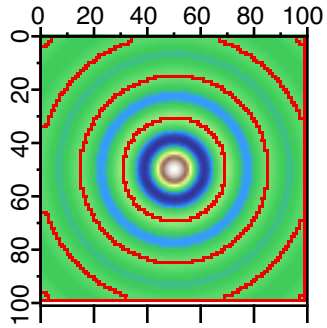




As you can see from the four images, the built-in Top Hat implementation enhances the boundaries (contours) of regions in the image whereas the opening or closing tophats enhance small grayscale variations.

The watershed operation locates the boundaries of watershed regions as we show below:

```
Make/O/N=(100,100) sample
sample=sinc(sqrt((x-50)^2+(y-50)^2)/2.5)// looks like concentric circles.
ImageTransform/O convert2Gray sample
NewImage sample
ModifyImage sample ctab= {*,*,Terrain,0}// color for better discrimination
ImageMorphology /N/L watershed sample
AppendImage M_ImageMorph
ModifyImage M_ImageMorph explicit=1, eval={0,65000,0,0}
```



Note that omitting the /L flag in the watershed operation may result in spurious watershed lines as the algorithm follows 4-connectivity instead of 8.

Image Analysis

The distinction between image processing and image analysis is rather fine. The pure analysis operations are: ImageStats, line profile, histogram, hsl segmentation and particle analysis.

ImageStats

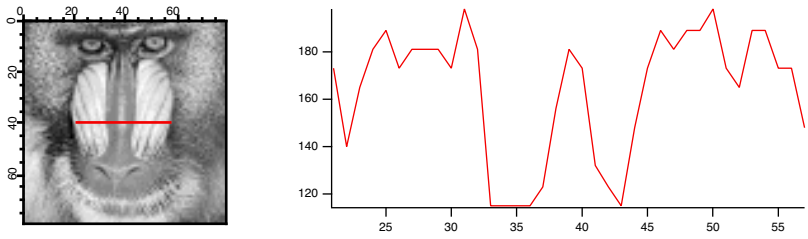
You can obtain global statistics on a wave using the standard **WaveStats** operation (see page V-729). The **ImageStats** operation (see page V-287) works specifically with 2D and 3D waves. The operation can define a completely arbitrary ROI using a standard ROI wave (see **Working with ROI** on page III-322). A special flag /M=1, speeds up the operation when you only want to know the minimum, maximum and average values in the ROI region, skipping over the additional computation time required to evaluate higher moments. This operation was designed to work in user defined adaptive algorithms.

ImageStats can also operate on a specific plane of a 3D wave using the /P flag.

ImageLineProfile

The **ImageLineProfile** operation (see page V-268) is somewhat of a misnomer as it samples the image along a path consisting of an arbitrary number of line segments. To use the operation you first need to create the description of the path using a pair of waves. Here is a simple example:

```
NewImage root:images:baboon    // Display the image that we want to profile
// Create the pair of waves representing a straight line path.
Make/O/N=2 xPoints={21,57}, yPoints={40,40}
AppendToGraph/T yPoints vs xPoints  // display the path on the image
// Calculate the profile.
ImageLineProfile xwave=xPoints, ywave=yPoints, srcwave=root:images:baboon
Display W_ImageLineProfile vs W_LineProfileX // display the profile
```



You can create a more complex path consisting of an arbitrary number of points. In this case you may want to take advantage of the W_LineProfileX and W_LineProfileY waves that the operation creates and plot the profile as a 3D path plot (see "Path Plots" in the Visualization help file). See also the IP Tutorial experiment for more elaborate examples.

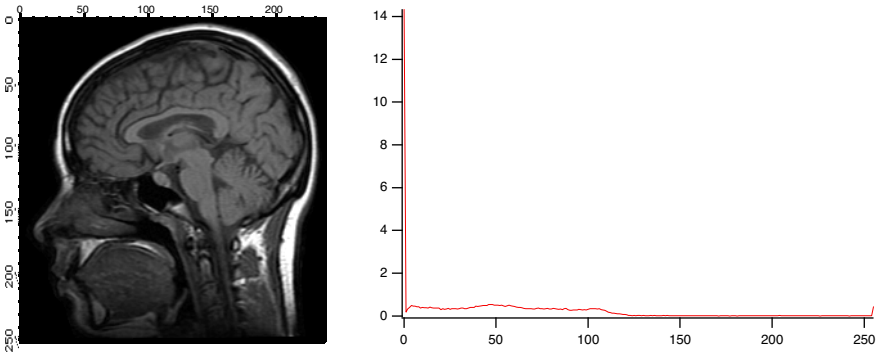
Note: If you are working with 3D waves with more than 3 layers, you can use `ImageLineProfile/P=`*`plane`* to specify the plane for which the profile is computed.

If you are using the line profile to extract a sequential array of data (a row or column) from the wave it is more efficient (about a factor of 3.5 in speed) to extract the data using ImageTransform getRow or getCol.

Histograms

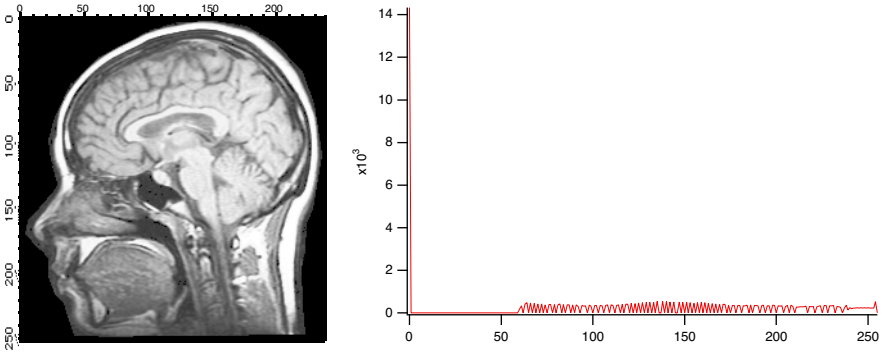
The histograms is a very important tool in image analysis. For example, the simplest approach to automating the detection of the background in an image is to calculate the histogram and to choose the pixel value which occurs with the highest frequency. Histograms are also very important in determining threshold values and in enhancing image contrast. Here are some examples using image histograms:

```
NewImage root:images:mri
ImageHistogram root:images:mri
Duplicate W_ImageHist origMriHist
Display /W=(201.6,45.2,411,223.4) origMriHist
```



It is obvious from the histogram that the image is rather dark and that the background is most likely zero. The small counts for pixels above 125 suggests that the image is a good candidate for histogram equalization.

```
ImageHistModification root:images:mri
ImageHistogram M_ImageHistEq
NewImage M_ImageHistEq
Display /W=(201.6,45.2,411,223.4) W_ImageHist
```



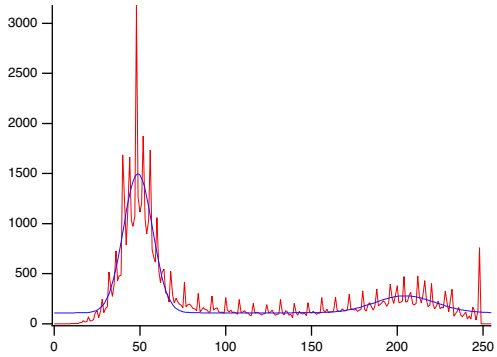
Comparing the two histograms two features stand out: first, there is no change in the dark background because it is only one level (0). Second, the rest of the image which was mostly between the values of 0 and 120 has now been stretched to the range 57-255.

The next example illustrates how you can use the histogram information to determine a threshold value.

```
NewImage root:images:blobs
ImageHistogram root:images:blobs
Display /W=(201.6,45.2,411,223.4) W_ImageHist
```

The resulting histogram is clearly bimodal. Let's fit it to a pair of Gaussians:

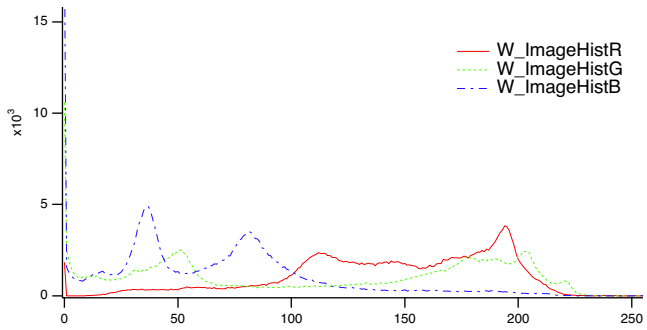
```
// Guess coefficient wave based on the histogram.
Make/O/N=6 coeff={0,3000,50,10,500,210,20}
Funcfit/Q twoGaussians,coeff,W_ImageHist /D
ModifyGraph rgb(fit_W_ImageHist)=(0,0,65000)
```



The curve shown in the graph is the functional fit of the sum of two Gaussians. You can now choose, by visual inspection, an x-value between the two Gaussians — probably somewhere in the range of 100-150. In fact, if you test the same image using the built-in thresholding operations that we have discussed above, you will see that the iterated algorithm chooses the value 125, fuzzy entropy chooses 109, etc.

Histograms of RGB or HSL images result in a separate histogram for each color channel:

```
ImageHistogram root:images:peppers
Display W_ImageHistR,W_ImageHistG,W_ImageHistB
ModifyGraph rgb(W_ImageHistG)=(0,65000,0),rgb(W_ImageHistB)=(0,0,65000)
```



Histograms of 3D waves containing more than 3 layers can be computed by specifying the layer with the /P flag. For example,

```
Make/N=(10,20,30) ddd=gnoise(5)
ImageHistogram/P=10 ddd
Display W_ImageHist
```

Unwrapping Phase

Unwrapping phase in two dimensions is more complicated than in one dimension because the operation's results must be independent of the unwrapping path. The path independence means that any path integral over a closed contour in the unwrapped domain must vanish. In many situations there are points in the domain around which closed contour path integrals do not vanish. Such points are called "residues". The residues are positive if a counter-clockwise path integral is positive. When unwrapping phase in two dimensions, the residues are typically ±1. This suggests that whenever two opposing residues are connected by a line (known as a "branch cut"), any contour integral whose path does not cross the branch cut will vanish. When a positive and negative residues are side by side they combine to a "dipole" which may be removed because a path integral around the dipole also vanishes. It follows that unwrapping can be performed using paths that either do not encircle unbalanced residues or paths that do not cross branch cuts.

The **ImageUnwrapPhase** operation (see page V-302) performs 2D phase unwrapping using either a fast method that ignores possible residues or a slower method which locates residues and attempts to find paths around them. The fast method uses direct integration of the differential phases. It can lead to incorrect results if there are residues in the domain. The slow method first identifies all residues, draws them into an internal bitmap adding branch cuts and then applying repeatedly the algorithm used in ImageSeedFill to obtain the paths around the residues and branch cuts until all pixels have been processed. Sometimes the distribution of residues and branch cuts is such that the domain of the data is covered by several regions, each of which is completely bounded by branch cuts or the data boundary. In this case, the phase is computed independently in each individual region with an offset that is based on the first processed pixel in that region. Note that when you use ImageUnwrapPhase using a method that computes the residues, the operation creates the variables V_numResidues and V_numRegions. You can also obtain a copy of the internal bitmap which could be useful for analyzing the results.

The ImageUnwrapPhase Demo in the Examples:Analysis folder provides a detailed example illustrating different types of residues, branch cuts and resulting unwrapped phase.

HSL Segmentation

When you work with color images you have two analogs to grayscale thresholding. The first is simple thresholding of the luminance of the image. To do this you need to convert the image from RGB to HSL and then perform the thresholding on the luminance plane. The second equivalent of thresholding is HSL segmentation, where the image is subdivided into regions of HSL values that fall within a certain range. In the following example we segment the peppers image to locate regions corresponding to red peppers:

```
NewImage root:images:peppers
ImageTransform/H={330,50}/L={0,255}/S={0,255} root:images:peppers
NewImage M_HueSegment
```

Note that we used /H={330,50}. The apparent flip of the limits is allowed in the case of hue values to cover the single range from hue angle 330 degrees to hue angle 50 degrees.

There are two additional approaches for color segmentation that should be mentioned here. You can use ImageTransform matchPlanes to segment an image for pixels that satisfy prescribed value ranges in all planes. This operation has the advantage that it can be applied to images in any color space. Another segmentation operation is ImageTransform selectColor which is based on RGB color space and a user provided tolerance value. The same concept can be applied with ImageSeedFill to get the effect of a "magic wand" selection.

Particle Analysis

Typical particle analysis consists of three steps. First you need to preprocess the image. This may include noise removal or reduction, possible background adjustments (see **ImageRemoveBackground** operation on page V-279) and thresholding. Once you obtain a binary image, your second step is to invoke the **ImageAnalyzeParticles** operation (see page V-252). The third and final step is making some sense of all the data produced by the ImageAnalyzeParticles operation or "post-processing".

Issues related to the preprocessing have been discussed elsewhere in this chapter. We will assume that we are starting with a preprocessed, clean, binary image which contains some particles.

```
NewImage root:images:blobs                // display the original image

// Step 1:create binary image.
// Note the /I flag to invert the output wave so that particles are marked by
zero.
ImageThreshold/I/Q/M=1 root:images:blobs  // Note the /I flag!!

// Step 2:Here we are invoking the operation in quiet mode, specifying particles
// of size equal or greater than 2 pixels. We are also asking for particles
moment
// Information, boundary waves and a particle masking wave.
ImageAnalyzeParticles /Q/A=2/E/W/M=2 stats M_ImageThresh

// Step 3:post processing choices
// Display the detected boundaries on top of the particles
AppendToGraph/T W_BoundaryY vs W_BoundaryX

// If you browse the numerical data:
Edit W_SpotX,W_SpotY,W_circularity,W_rectangularity,W_ImageObjPerimeter
AppendToTable W_xmin,W_xmax,W_ymin,W_ymax,M_Moments,M_RawMoments
```

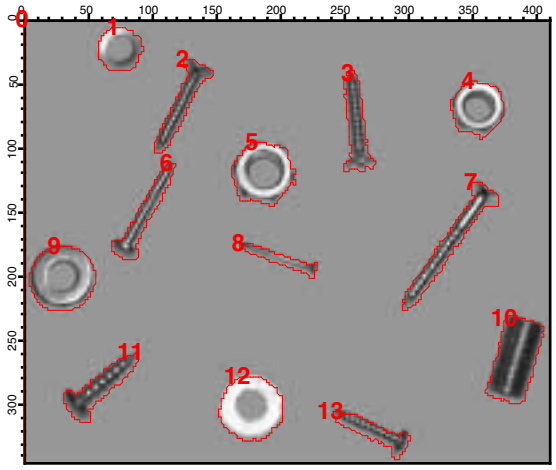
Note that particles that intersect the boundary of the image may give rise to inaccuracies in particle statistics. It is therefore useful sometimes to remove these particles before performing the analysis.

The raw values generated by ImageAnalyzeParticles operation can be used for further processing.

The following example illustrates slightly different pre and post-processing.

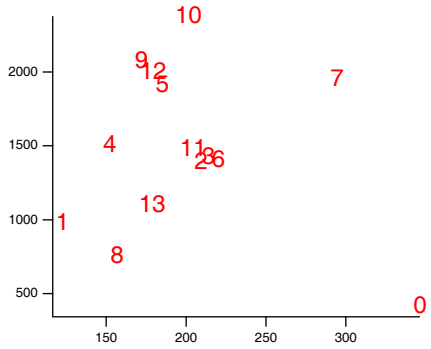
```
NewImage screws
// Here we have a synthetic background so the conversion to binary is easy.
screws=screws==163 ? 255:0
ImageMorphology /O/I=2/E=1 binarydilation screws
ImageMorphology /O/I=2/E=1 erosion screws

// Now the particle analysis operation with the option to fill the holes.
ImageAnalyzeParticles/E/W/Q/M=3/A=5/F stats, screws

NewImage root:images:screws
AutoPositionWindow/E $WinName(0,1)
// Show the detected boundaries.
AppendToGraph/T W_BoundaryY vs W_BoundaryX
AppendToGraph/T W_SpotY vs W_SpotX
Duplicate/O w_spotx w_index
w_index=p
```

```
ModifyGraph mode(W_SpotY)=3
ModifyGraph textMarker(W_SpotY)={w_index,"default",1,0,5,0.00,0.00}
ModifyGraph msize(W_SpotY)=6
```



Now for some shape classification in which we plot particle area versus perimeter:

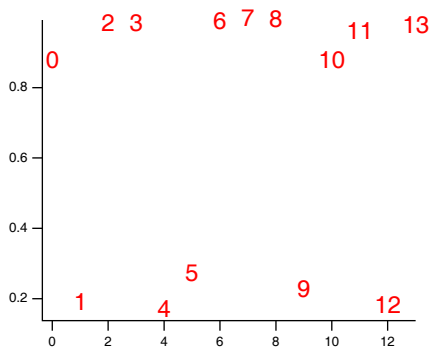
```
Display/W=(23.4,299.6,297,511.4) W_ImageObjArea vs W_ImageObjPerimeter
ModifyGraph
mode=3,textMarker(W_ImageObjArea)={w_index,"default",0,0,5,0.00,0.00}
ModifyGraph msize=6
```



The classification diagram we just created uses two parameters (area and perimeter) that are very sensitive to image noise. We can see that there are two basic classes that can be associated with the roundness of the boundaries but it is difficult to accept the classification of particle 9.

In the following we compute another classification based on the eccentricity of the objects:

```
Make/O/N=(DimSize(M_Moments,0)) ecc
ecc=sqrt(1-M_Moments[p][3]^2/M_Moments[p][2]^2)

Display /W=(23.4,299.6,297,511.4) ecc
ModifyGraph mode=3,textMarker(ecc)={w_index,"default",0,0,5,0.00,0.00}
ModifyGraph msize=6
```



The second classification produces a distinct separation of the screws from the washers and nuts. It also illustrates the importance of selecting the best classification parameters.

You can use the ImageAnalyzeParticles operation also for the purpose of creating masks for particular particles. For example, to create a mask for particle 9 in the example above:

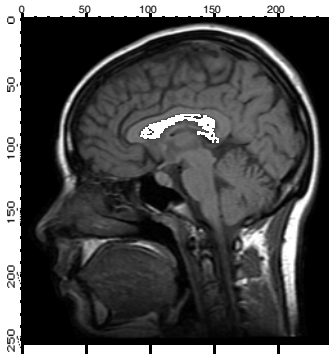
```
ImageAnalyzeParticles /L=(w_spotX[9],w_spotY[9]) mark screws
NewImage M_ParticleMarker
```

You can use this feature of the operation to color different classes of objects using an overlay.

Seed Fill

In some situations you may need to define segments of the image based on a contiguous region of pixels whose values fall within a certain range. The **ImageSeedFill** operation (see page V-283) helps you do just that.

```
NewImage mri
ImageSeedFill/B=64 seedX=132,seedY=77,min=52,max=65,target=255,srcWave=mri
AppendImage M_SeedFill
ModifyImage M_SeedFill explicit=1, eval={255,65535,65535,65535}
```



Here we have used the /B flag to create an overlay image but it can also be used to create an ROI wave for use in further processing. This example represents the simplest use of the operation. In some situations the criteria for a pixel's inclusion in the filled region are not so sharp and the operation may work better if you use the adaptive or fuzzy algorithms. For example (**Note**: the command is wrapped over two lines):

```
ImageSeedFill/B=64/c seedX=144,seedY=83,min=60,max=150,target=255,
srcWave=mri,adaptive=3
```

Note that the min and max values have been relaxed but the adaptive parameter provides alternative continuity criterion.

Other Tools

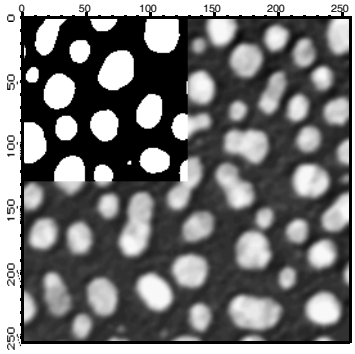
Igor provides a number of utility operations that help you manage and manipulate image data.

Working with ROI

Many of the image processing operations support a region of interest (ROI). The region of interest is the portion of the image that we want to affect by an operation. Igor supports a completely general ROI, specified as a binary wave (unsigned byte) that has the same dimensions as the image. Set the ROI wave to zero for all pixels within the region of interest and to any other value outside the region of interest. Note that in some situations it is useful to set the ROI pixels to a nonzero value (e.g., if you use the wave as a multiplicative factor in a mathematical operation). You can use ImageTransform with the invert keyword to quickly convert between the two options.

The easiest way to create an ROI wave is directly from the command line. For example, an ROI wave that covers a quarter of the blobs image may be generated as follows:

```
Duplicate root:images:blobs myROI
Redimension/B/U myROI
myROI=64                        // arbitrary nonzero value
myROI[0,128][0,127]=0           // the actual region of interest

// Example of usage:
ImageThreshold/Q/M=1/R=myROI root:images:blobs
NewImage M_ImageThresh
```



If you want to define the ROI using graphical drawing tools you need to open the tools and set the drawing layer to progFront. This can be done with the following instructions:

```
SetDrawLayer progFront
ShowTools/A rect          // selects the rectangle drawing tool first
```

Generating ROI Masks

You can now define the ROI as the area inside all the closed shapes that you draw. When you complete drawing the ROI you need to execute the commands:

```
HideTools/A               // Drawing tools are not needed any more.
// M_ImageThresh is the top image in this example.
ImageGenerateROIMask M_ImageThresh
// The ROI wave has been created; To see it,
NewImage M_ROIMask
```

The Image Processing procedures provide a utility for creating an ROI wave by drawing on a displayed image.

Converting Boundary to a Mask

A third way of generating an ROI mask is using the **ImageBoundaryToMask** operation (see page V-256). This operation takes a pair of waves (y versus x) that contain pixel values and scan-converts them into a mask. When you invoke the operation you also have to specify the rectangular width and height of the output mask.

```
// Create a circle.
Make/N=100 ddx,ddy
ddx=50*(1-sin(2*pi*x/100))
ddy=50*(1-cos(2*pi*x/100))
```

```
ImageBoundaryToMask width=100,height=100,xwave=ddx,ywave=ddy
// The result is an image not a curve!
NewImage M_ROIMask
```

Note that the resulting binary wave has the values 0 and 255, which you may need to invert before using them in certain operations.

In many situations the operation ImageBoundaryToMask is followed by ImageSeedFill in order to convert the mask to a filled region. You can obtain the desired mask in one step using the keywords seedX and seedY in ImageBoundaryToMask but you must make sure that the mask created by the boundary waves is a closed domain.

```
ImageBoundaryToMask width=100,height=100,xwave=ddx,ywave=ddy,
seedX=50,seedY=50
ModifyImage M_ROIMask explicit=0
```

Marquee Procedures

A fourth way to create an ROI mask is using the Marquee2Mask procedures. To use this in your own experiment you will have to add the following line to your procedure window:

```
#include <Marquee2Mask>
```

You can now create the ROI mask by selecting one or more rectangular marquees (drag the mouse) in the image. After you select each marquee click inside the marquee and choose MarqueeToMask or Append-MarqueeToMask.

Subimage Selection

You can use an ROI to apply various image processing operations to selected portions of an image. The ROI is a very useful tool especially when the region of interest is either not contiguous or not rectangular. When the region of interest is rectangular, you can usually improve performance by creating a new subimage which consists entirely of the ROI. If you know the coordinates and dimensions of the ROI it is simplest to use the `Duplicate/R` operation. If you want to make an interactive selection you can use the marquee together with CopyImageSubset marquee procedure (after making a marquee selection in the image, click inside the marquee and choose CopyImageSubset).

Handling Color

Most of the image operations are designed to work on grayscale images. If you need to perform an operation on a color image certain aspects become a bit more complicated. In the next example we illustrate how you might sharpen a color image.

```
NewImage root:images:rose
ImageTransform rgb2hsl root:images:rose   // first convert to hsl
ImageTransform/P=2 getPlane M_RGB2HSL
ImageFilter Sharpen M_ImagePlane    // you can also use sharpenmore
ImageTransform /D=M_ImagePlane /P=2 setPlane M_RGB2HSL
ImageTransform hsl2rgb M_RGB2HSL
NewImage M_HSL2RGB
```

Background Removal

There are many approaches to removing the effect of a nonuniform background from an image. If the non uniformity is additive, it is sometimes useful to fit a polynomial to various points which you associate with the background and then subtract the resulting polynomial surface from the whole image. If the nonuniformity is multiplicative, you need to generate an image corresponding to the polynomial surface and use it to scale the original image.

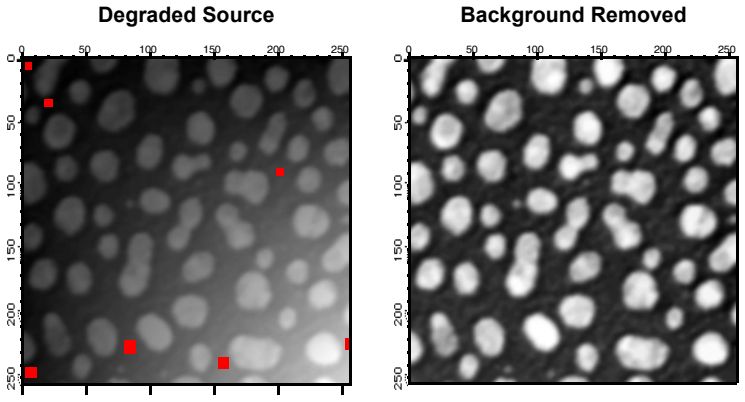
Additive Background

```
Duplicate/O root:images:blobs addBlobs
Redimension/S addBlobs              // convert to single precision
```

```
addBlobs+=0.01*x*y                    // add a tilted plane
NewImage addBlobs
```

To use the **ImageRemoveBackground** operation (see page V-279), we need an ROI mask designating regions in the image that represent the background. You can create one using one of the ROI construction methods that we discussed above. For the purposes of this example, we choose the ROI that consists of the 7 rectangles shown in the Degraded Source image below.

```
// Show the ROI background selection.
AppendImage root:images:addMask
ModifyImage addMask explicit=1, eval={1,65000,0,0}

// Create a corrected image and display it.
ImageRemoveBackground /R=root:images:addMask /w/P=2 addBlobs
NewImage M_RemovedBackground
```



If the source image contains relatively small particles on a nonuniform background, you may remove the background (for the purpose of particle analysis) by iterating grayscale erosion until the particles are all gone. You are then left with a fairly good representation of the background that can be subtracted from the original image.

Multiplicative Background

This case is much more complicated because the removal of the background requires division of the image by the calculated background (it is assumed here that the system producing the image has an overall gamma of 1). The first complication has to do with the possible presence of zeros in the calculated background. The second complication is that the calculations give us the additional freedom to choose one constant factor to scale the resulting image. There are many approaches for correcting a multiplicative background. The following example shows how an image can be corrected if we assume that the peak values (identified by the ROI mask) would all have the same value in the absence of a background.

```
Duplicate/O root:images:blobs mulBlobs
Redimension/S mulBlobs           // convert to single precision
mulBlobs*=(1+0.005*x*y)
NewImage mulBlobs

// Show us the ROI foreground selection; you can use histogram
// equalization to find fit regions in the dark area.
AppendImage root:images:multMask
ModifyImage multMask explicit=1, eval={1,65000,0,0}

ImageRemoveBackground /R=root:images:multMask/F/w/P=2 mulBlobs
// Normalize the fit.
WaveStats/Q/M=1 M_RemovedBackground

// Renormalize the fit--we can use that one free factor.
M_RemovedBackground=(M_RemovedBackground-V_min)/(V_max-V_min)
// Remove zeros by replacing with average value.
WaveStats/Q/M=1 M_RemovedBackground
```

```
MatrixOP/O M_RemovedBackground=M_RemovedBackground+V_avg*equal(M_RemovedBackground,0)
MatrixOP/O mulBlobs=mulBlobs/M_RemovedBackground   // scaled image.
```

In the example above we have manually created the ROI masks that were needed for the fit. You can automate this process (and actually improve performance) by subdividing the image into a number of smaller rectangles and selecting in each one the highest (or lowest) pixel values. An example of such procedure is provided in connection with the ImageStats operation above.

General Utilities: ImageTransform Operation

As we have seen above, the **ImageTransform** operation (see page V-290) provides a number of image utilities. As a rule, if you are unable to find an appropriate image operation check the options available under ImageTransform. Here are some examples:

When working with RGB or HSL images it is frequently necessary to access one plane at a time. For example, the green plane of the peppers image can be obtained as follows:

```
NewImage root:images:peppers        // display original
Duplicate/O root:images:peppers peppers
ImageTransform /P=1 getPlane peppers
NewImage M_ImagePlane               // display green plane in grayscale
```

The complementary operation can insert a plane into a 3D wave. For example, suppose you wanted to modify the green plane of the peppers image:

```
DoWindow/K WM_temp
ImageHistModification/o M_ImagePlane
ImageTransform /p=1 /D=M_ImagePlane setPlane peppers
NewImage peppers                    // display the processed image
```

Some operations are restricted to waves of particular dimensions. For example, if you want to use the Adaptive histogram equalization, the number of horizontal and vertical partitions is restricted by the requirement that the image be an exact multiple of the dimensions of the subregion. The ImageTransform operation provides three image padding options: If you specify a negative number to the changed rows or columns, the corresponding rows and columns are removed from the image. If the numbers are positive, rows and columns are added. By default the added rows and columns contain exactly the same pixel values as the last row and column in the image. If you specify the /W flag the operation duplicates the relevant portion of the image into the new rows and columns. Here are some examples:

```
Duplicate/o root:images:baboon baboon
NewImage baboon
ImageTransform/N={-20,-10} padImage baboon
Rename M_PaddedImage, cropped
NewImage cropped
ImageTransform/N={40,40} padImage baboon
Rename M_PaddedImage, padLastVals
NewImage padLastVals
ImageTransform/W/N={100,100} padImage baboon
NewImage M_PaddedImage
```

Another utility operation is the conversion of any 2D wave into a normalized (0-255) 8-bit image wave. This is accomplished with the ImageTransform operation using the keyword convert2gray. Here is an example:

```
// Create some numerical data
Make/O/N=(50,80) numericalWave=x*sin(x/10)*y*exp(y/100)
ImageTransform convert2gray numericalWave
NewImage M_Image2Gray
```

The conversion to an 8-bit image is required for certain operation. It is also useful sometimes when you want to reduce the size of your image waves.

References

Ghiglia, Dennis C., and Mark D. Pritt, *Two Dimensional Phase Unwrapping — Theory, Algorithms and Software*, John Wiley & Sons, 1998.

Gonzalez, Rafael C., and Richard E. Woods, *Digital Image Processing*, 3rd ed., Addison-Wesley, 1992.

Pratt, William K., *Digital Image Processing*, 2nd ed., John Wiley & Sons, 1991.

Thévenaz, P., and M. Unser, A Pyramid Approach to Subpixel Registration Based on Intensity, *IEEE Transactions on Image Processing*, *7*, 27-41, 1998.

Chapter
III-12

Statistics

Overview

This chapter describes operations and functions for statistical analysis together with some general guidelines for their use. This is not a statistics tutorial; for that you can consult one of the references at the end of this chapter or the references listed in the documentation of a particular operation or function. The material below assumes that you are familiar with techniques and methods of statistical analysis.

Grouping by Functionality

Prior to Igor Pro 6.0 there were few built-in operations and functions for statistical analysis. They included: Binomial, erf, erfc, Sort, StudentA, StudentT and WaveStats. Additional functionality was provided by the (now obsolete) StatFuncs XOP. As of Igor Pro 6.0, with few exceptions, new statistical analysis operations and functions are named with the prefix "Stats" and belong to a new Statistics class in the help browser. Naming exceptions include the random "noise" generation functions that have traditionally been named based on the distribution they represent.

There are six natural groups of Statistics operations and functions. They include: test operations, noise functions, probability distribution functions (PDFs), cumulative distribution functions (CDFs), inverse CDFs and general purpose operations and functions.

Statistical Test Operations

Test operations analyze the input data to examine the validity of a specific hypothesis. The common test involves a computation of some numeric value (also known as "test statistic") which is usually compared with a critical value in order to determine if you should accept or reject the test hypothesis (H_0). Most tests compute a critical value for the given significance *alpha* which has the default value 0.05 or a user-provided value via the /ALPH flag. Some tests directly compute the P value which you can compare to the desired significance value.

Critical values have been traditionally published in tables for various significance levels and tails of distributions. They are by far the most difficult technical aspect in implementing statistical tests. The critical values are usually obtained from the inverse of the CDF for the particular distribution, i.e., from solving *cdf(criticalValue)*=1-*alpha*, where *alpha* is the significance. In some distributions (e.g., Friedman's) the calculation of the CDF is so computationally intensive that it is impractical (using desktop computers in 2006) to compute for very large parameters. Igor's tests provide whenever possible exact critical values as well as the common relevant approximations.

Comparison of critical values with published table values can sometimes be interesting as there does not appear to be a standard for determining the published critical value when the CDF takes a finite number of discreet values (step-like). In this case the CDF attains the value (1-*alpha*) in a vertical transition so one could use the X value for the vertical transition as a critical value or the X value of the subsequent vertical transition. Some tables reflect a "conservative" approach and print the X value of subsequent transitions.

Statistical test operations can print their results to the history window and save them in a wave in the current data folder. Result waves have a fixed name associated with the operation. Elements in the wave are designated by dimension labels. You can use the /T flag to display the results of the operation in a table with dimension labels. The argument for this flag determines what happens when you kill the table. You can use/Q in all test operations to prevent printing information in the history window and you can use the /Z flag to make sure that the operations do not report errors except by setting the V_Flag variable to -1.

Statistical test operations tend to include several variations of the named test. You can usually choose to execute one or more variations by specifying the appropriate flags. The following table can be used as a guide for identifying the operation associated with a given test name.

Statistical Test Operations by Name

Test Name	Where to find it
Angular Distance	**StatsAngularDistanceTest**
Bartlett's test for variances	**StatsVariancesTest**
BootStrap	**StatsResample**
Brown and Forsythe	**StatsANOVA1Test**
Chi-squared test for means	**StatsChiTest**
Cochran's test	**StatsCochranTest**
Dunn-Holland-Wolfe	**StatsNPMCTest**
Dunnette multicomparison test	**StatsDunnettTest**, **StatsLinearRegression**
Fisher's Exact Test	**StatsContingencyTable**
Fixed Effect Model	**StatsANOVA1Test**
Friedman test on randomized block	**StatsFriedmanTest**
F-test on two distributions	**StatsFTest**
Hodges-Ajne (Batschelet)	**StatsHodgesAjneTest**
Hartigan test for unimodality	**StatsDIPTest**
Hotelling	**StatsCircularTwoSampleTest**, **StatsCircularMeans**
Jackknife	**StatsResample**
Jarque-Bera Test	**StatsJBTest**
Kolmogorov-Smirnov	**StatsKSTest**
Kruskal-Wallis	**StatsKWTest**
Kuiper Test	**StatsCircularMoments**
Levene's test for variances	**StatsVariancesTest**
Linear Correlation Test	**StatsLinearCorrelationTest**
Linear Order Statistic	**StatsCircularMoments**
Mann-Kendall	**StatsKendallTauTest**
Moore test	**StatsCircularTwoSampleTest**, **StatsCircularMeans**
Nonparametric multiple contrasts	**StatsNPMCTest**
Nonparametric angular-angular correlation	**StatsCircularCorrelationTest**
Nonparametric second order circular analysis	**StatsCircularMeans**
Nonparametric serial randomness (nominal)	**StatsNPNominalSRTest**
Parametric angular-angular correlation	**StatsCircularCorrelationTest**
Parametric angular-Linear correlation	**StatsCircularCorrelationTest**
Parametric second order circular analysis	**StatsCircularMeans**
Parametric serial randomness test	**StatsSRTest**
Rayleigh	**StatsCircularMoments**
Repeated Measures	**StatsANOVA2RMTest**

Test Name	Where to find it
Scheffe equality of means	**StatsScheffeTest**
Spearman	**StatsRankCorrelationTest**
Student-Newman-Keuls	**StatsNPMCTest**
Tukey Test	**StatsTukeyTest StatsLinearRegression**, **StatsMultiCorrelationTest**, **StatsNPMCTest**
Two-Factor ANOVA	**StatsANOVA2NRTest**
T-test	**StatsTTest**
Watson's nonparametric two-sample U2	**StatsWatsonUSquaredTest**, **StatsCircularTwoSampleTest**
Watson-Williams	**StatsWatsonWilliamsTest**
Weighted-rank correlation test	**StatsWRCorrelationTest**
Wheeler-Watson nonparametric test	**StatsWheelerWatsonTest**
Wilcoxon-Mann-Whitney two-sample	**StatsWilcoxonRankTest**
Wilcoxon signed rank	**StatsWilcoxonRankTest**

Statistical Test Operations by Data Format

The following tables group statistical operations and functions according to the format of the input data.

Data Type: Single wave.

Analysis Method	Comments
StatsChiTest	Compares with known binned values
StatsCircularMoments	WaveStats for circular data
StatsKendallTauTest	Similar to Spearman's correlation
StatsMedian	Returns the median
StatsNPNominalSRTest	Nonparametric serial randomness test
StatsQuantiles	Computes quantiles and more
StatsResample	Bootstrap analysis
StatsSRTest	Serial randomenss test
StatsTrimmedMean	Returns the trimmed mean
StatsTTest	Compares with known mean
Sort	Reorders the data
WaveStats	Basic statistical description
StatsJBTest	Jarque-Bera test for normality
StatsKSTest	Limited scope test for normality
StatsDIPTest	Hartigan test for unimodality

Data Type: Two waves.

Analysis Method	Comments
StatsChiTest	Statistic for comparing two distributions
StatsCochranTest	Randomized block or repeated measures test
StatsCircularTwoSampleTest	Second order analysis of angles
StatsDunnettTest	Compares multiple groups to a control
StatsFTest	Computes ratio of variances
StatsFriedmanTest	Nonparametric ANOVA
StatsKendallTauTest	Similar to Spearman's correlation
StatsTTest	Compares the means of two distributions
StatsANOVA1Test	One-way analysis of variances
StatsLinearRegression	Linear regression analysis
StatsLinearCorrelationTest	Linear correlation coefficient and its error
StatsRankCorrelationTest	Computes Spearman's rank correlation
StatsVariancesTest	Compares variances of waves
StatsWilcoxonRankTest	Two-sample or signed rank test
StatsWatsonUSquaredTest	Compares two populations of circular data
StatsWatsonWilliamsTest	Compares mean values of angular distributions
StatsWheelerWatsonTest	Compares two angular distributions

Data Type: Multiple waves or multidimensional waves.

Analysis Method	Comments
StatsANOVA1Test	One-way analysis of variances
StatsANOVA2Test	Two-factor analysis of variances
StatsANOVA2RMTest	Two-factor repeated measure ANOVA
StatsCochranTest	Randomized block or repeated measures test
StatsContingencyTable	Contingency table analysis
StatsDunnettTest	Comparing multiple groups to a control
StatsFriedmanTest	Nonparametric ANOVA
StatsNPMCTest	Nonparametric multiple comparison tests
StatsScheffeTest	Tests equality of means
StatsTukeyTest	Multiple comparisons based on means
StatsWatsonWilliamsTest	Compares mean values of angular distributions
StatsWheelerWatsonTest	Compares two angular distributions

Statistical Test Operations for Angular/Circular Data

StatsAngularDistanceTest	**StatsHodgesAjneTest**
StatsCircularMoments	**StatsWatsonUSquaredTest**
StatsCircularMeans	**StatsWatsonWilliamsTest**
StatsCircularTwoSampleTest	**StatsWheelerWatsonTest**
StatsCircularCorrelationTest	

Statistical Test Operations: Nonparametric Tests

Operation	Comments
StatsAngularDistanceTest	
StatsFriedmanTest	
StatsCircularTwoSampleTest	Parametric or nonparametric
StatsCircularCorrelationTest	Parametric or nonparameteric
StatsCircularMeans	Parametric or nonparameteric
StatsHodgesAjneTest	
StatsKendallTauTest	
StatsKWTest	
StatsNPMCTest	
StatsNPNominalSRTest	
StatsRankCorrelationTest	
StatsWatsonUSquaredTest	
StatsWheelerWatsonTest	
StatsWilcoxonRankTest	

Noise Functions

The following functions return numbers from a pseudo-random distribution of the specified shapes and parameters. Except for enoise and gnoise where you have an option to select a random number generator, the remaining noise functions use a Mersenne Twister algorithm for the initial uniform pseudo-random distribution. Note that whenever you need repeatable results you should use SetRandomSeed prior to executing any of the noise functions.

The following noise generation functions are available:

binomialNoise	**logNormalNoise**
enoise	**lorentzianNoise**
expnoise	**poissonNoise**
gammaNoise	**StatsPowerNoise**
gnoise	**StatsVonMisesNoise**
hyperGNoise	**wnoise**

Cumulative Distribution Functions

A cumulative distribution function (CDF) is the integral of its respective probability distribution function (PDF). CDFs are usually well behaved functions with values in the range [0,1]. CDFs are important in computing critical values, P values and power of statistical tests.

Many CDFs are computed directly from closed form expressions. Others can be difficult to compute because they involve evaluating a very large number of states, e.g., Friedman or USquared distributions. In these cases you have the following options:

1. Use a built-in table that consists of exact, precomputed values.
2. Compute an approximate CDF based on the prevailing approximation method or using a Monte-Carlo approach.
3. Compute the exact CDF.

Built-in tables are ideal if they cover the range of the parameters that you need. Monte-Carlo methods can be tricky in the sense that repeated application may return small variations in values. Computing the exact CDF may be desirable, but it is often impractical. In most situations the range of parameters that is practical to compute on a desktop machine is already covered in the built-in tables. Larger parameters not have been considered because they take days to compute or because they require 64 bit processors. In addition, most of the approximations tend to improve with increasing size of the parameters.

The functions to calculate values from CDFs are as follows:

StatsBetaCDF	**StatsHyperGCDF**	**StatsQCDF**
StatsBinomialCDF	**StatsKuiperCDF**	**StatsRayleighCDF**
StatsCauchyCDF	**StatsLogisticCDF**	**StatsRectangularCDF**
StatsChiCDF	**StatsLogNormalCDF**	**StatsRunsCDF**
StatsCMSSDCDF	**StatsMaxwellCDF**	**StatsSpearmanRhoCDF**
StatsDExpCDF	**StatsInvMooreCDF**	**StatsStudentCDF**
StatsErlangCDF	**StatsNBinomialCDF**	**StatsTopDownCDF**
StatsEValueCDF	**StatsNCFCDF**	**StatsTriangularCDF**
StatsExpCDF	**StatsNCTCDF**	**StatsUSquaredCDF**
StatsFCDF	**StatsNormalCDF**	**StatsVonMisesCDF**
StatsFriedmanCDF	**StatsParetoCDF**	**StatsQCDF**
StatsGammaCDF	**StatsPoissonCDF**	**StatsWaldCDF**
StatsGeometricCDF	**StatsPowerCDF**	**StatsWeibullCDF**

Probability Distribution Functions

Probability distribution functions (PDF) are sometimes known as probability densities. In the case of continuous distributions, the area under the curve of the PDF for each interval equals the probability for the random variable to fall within that interval. The PDFs are useful in calculating event probabilities, characteristic functions and moments of a distribution.

The functions to calculate values from PDFs are as follows:

StatsBetaPDF	**StatsGammaPDF**	**StatsParetoPDF**
StatsBinomialPDF	**StatsGeometricPDF**	**StatsPoissonPDF**
StatsCauchyPDF	**StatsHyperGPDF**	**StatsPowerPDF**
StatsChiPDF	**StatsLogNormalPDF**	**StatsRayleighPDF**

StatsDExpPDF	**StatsMaxwellPDF**	**StatsRectangularPDF**
StatsErlangPDF	**StatsBinomialPDF**	**StatsStudentPDF**
StatsErrorPDF	**StatsNCChiPDF**	**StatsTriangularPDF**
StatsEValuePDF	**StatsNCFPDF**	**StatsVonMisesPDF**
StatsExpPDF	**StatsNCTPDF**	**StatsWaldPDF**
StatsFPDF	**StatsNormalPDF**	**StatsWeibullPDF**

Inverse Cumulative Distribution Functions

The inverse cumulative distribution functions return the values at which their respective CDFs attain a given level. This value is typically used as a critical test value. There are very few functions for which the inverse CDF can be written in closed form. In most situations the inverse is computed iteratively from the CDF.

The functions to calculate values from inverse CDFs are as follows:

StatsInvBetaCDF	**StatsInvKuiperCDF**	**StatsInvQpCDF**
StatsInvBinomialCDF	**StatsInvLogisticCDF**	**StatsInvRayleighCDF**
StatsInvCauchyCDF	**StatsInvLogNormalCDF**	**StatsInvRectangularCDF**
StatsInvChiCDF	**StatsInvMaxwellCDF**	**StatsInvSpearmanCDF**
StatsInvCMSSDCDF	**StatsInvMooreCDF**	**StatsInvStudentCDF**
StatsInvDExpCDF	**StatsInvNBinomialCDF**	**StatsInvTopDownCDF**
StatsInvEValueCDF	**StatsInvNCFCDF**	**StatsInvTriangularCDF**
StatsInvExpCDF	**StatsInvNormalCDF**	**StatsInvUSquaredCDF**
StatsInvFCDF	**StatsInvParetoCDF**	**StatsInvVonMisesCDF**
StatsInvFriedmanCDF	**StatsInvPoissonCDF**	**StatsInvWeibullCDF**
StatsInvGammaCDF	**StatsInvPowerCDF**	
StatsInvGeometricCDF	**StatsInvQCDF**	

General purpose operations and functions

This group includes operations and functions that existed before IGOR Pro 6.0 and some general purpose operations and functions that do not belong to the main groups listed.

binomial	**Sort**	**StatsTrimmedMean**
binomialln	**StatsCircularMoments**	**StudentA**
erf	**StatsCorrelation**	**StudentT**
erfc	**StatsMedian**	**WaveStats**
inverseErf	**StatsQuantiles**	**StatsPermute**
inverseErfc	**StatsResample**	

Hazard and Survival functions

Igor does not provide built-in functions to calculate the Survival or Hazard functions. They can be calculated easily from the **Probability Distribution Functions** on page III-333 and **Cumulative Distribution Functions** on page III-333.

In the following, the cumulative distribution functions are denoted by $F(x)$ and the probability distribution functions are denoted by $p(x)$.

The Survival Function $S(x)$ is given by

$$S(x) = 1 - F(x)$$

The Hazard function $h(x)$ is given by

$$h(x) = \frac{p(x)}{S(x)} = \frac{p(x)}{1 - F(x)}.$$

The cumulative hazard function $H(x)$ is

$$H(x) = \int_{-\infty}^{x} h(u)du$$

$$H(x) = -ln\left[1 - F(x)\right]$$

Inverse Survival Function $Z(a)$ is

$$Z(\alpha) = G(1 - \alpha),$$

where $G()$ is the inverse CDF (see **Inverse Cumulative Distribution Functions** on page III-334).

Procedures

Several procedure files are provided to extend the built-in statistics capability described in this chapter. Some of these procedure files provide user interfaces to the built-in statistics functionality. Others extend the functionality.

In the Analysis menu you will find a Statistics item that brings up a submenu. Selecting any item in the submenu will cause all the statistics-related procedure files to be loaded, making them ready to use. Alternatively, you can load all the statistics procedures by adding the following include statement to the top of your procedure window:

```
#include <AllStatsProcedures>
```

Functionality provided by the statistics procedure files includes the 1D statistics Report package for automatic analysis of single 1D waves, and the ANOVA Power Calculations Panel, as well as functions to create specialized graphs:

statsAutoCorrPlot()	statsPlotLag()	statsPlotHistogram()
statsBoxPlot()	statsProbPlot()	

Convenience functions:

WM_2MeanConfidenceIntervals()	WM_MCPointOnRegressionLines()
WM_2MeanConfidenceIntervals2()	WM_MeanConfidenceInterval()

WM_2MeanConfidenceIntervals()
WM_BernoulliCdf()
WM_BinomialPdf()
WM_CIforPooledMean()
WM_CompareCorrelations()
WM_EstimateMinDetectableDiff()
WM_EstimateReqSampleSize()
WM_EstimateReqSampleSize2()
WM_EstimateSampleSizeForDif()
WM_GetANOVA1Power()
WM_GetGeometricAverage()
WM_GetHarmonicMean()
WM_GetPooledMean()
WM_GetPooledVariance()
WM_MCPointOnRegressionLines()
WM_OneTailStudentA()
WM_OneTailStudentT()
WM_PlotBiHistogram()
WM_RankForTies()
WM_RankLetterGradesWithTies()
WM_RegressionInversePrediction()
WM_SSEstimatorFunc()
WM_SSEstimatorFunc2()
WM_SSEstimatorFunc3()
WM_VarianceConfidenceInterval()
WM_WilcoxonPairedRanks()
WM_StatsKaplanMeier()

Obsolete XOP

Much of the statistics functionality prior to Igor Pro 6 was provided by the StatFuncs XOP. This XOP is now obsolete; all functionality provided by the XOP is available in the form of built-in functions and operations. The XOP is still available for backward compatibility, but you should move your work to the new built-in equivalents.

This table specifies how to map the old functions into new ones.

Old	New
pnoise()	**poissonNoise**()
gammanoise()	**gammaNoise**()
statTTest()	**StatsTTest**
statFTest()	**StatsFTest**
statsChiTest()	**StatsChiTest**
StatsPearsonTest()	**StatsCorrelation**

References

Ajne, B., A simple test for uniformity of a circular distribution, *Biometrica*, *55*, 343-354, 1968.

Bradley, J.V., *Distribution-Free Statistical Tests*, Prentice Hall, Englewood Cliffs, New Jersey, 1968.

Cheung, Y.K., and J.H. Klotz, The Mann Whitney Wilcoxon distribution using linked lists, *Statistica Sinica*, *7*, 805-813, 1997.

Copenhaver, M.D., and B.S. Holland, Multiple comparisons of simple effects in the two-way analysis of variance with fixed effects, *Journal of Statistical Computation and Simulation*, *30*, 1-15, 1988.

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

Fisher, N.I., *Statistical Analysis of Circular Data*, 295pp., Cambridge University Press, New York, 1995.

Iman, R.L., and W.J. Conover, A measure of top-down correlation, *Technometrics*, *29*, 351-357, 1987.

Kendall, M.G., *Rank Correlation Methods*, 3rd ed., Griffin, London, 1962.

Klotz, J.H., *Computational Approach to Statistics*, <http://www.stat.wisc.edu/~klotz/Book.pdf>.

Moore, B.R., A modification of the Rayleigh test for vector data, *Biometrica*, *67*, 175-180, 1980.

Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

van de Wiel, M.A., and A. Di Bucchianico, Fast computation of the exact null distribution of Spearman's rho and Page's L statistic for samples with and without ties, *J. of Stat. Plan. and Inference*, *92*, 133-145, 2001.

Wallace, D.L., Simplified Beta-Approximation to the Kruskal-Wallis H Test, *J. Am. Stat. Assoc*., *54*, 225-230, 1959.

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

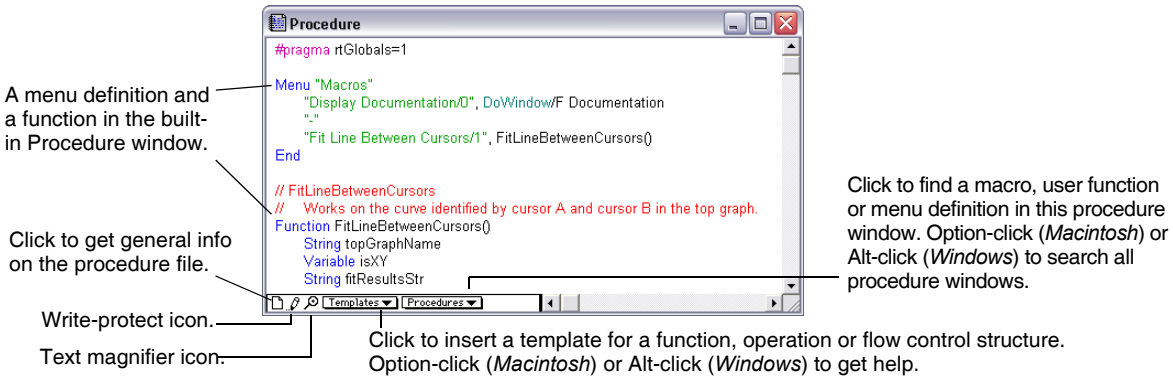
Chapter III-13

Procedure Windows

Overview

This chapter explains what procedure windows are, how they are created and organized, and how you work with them. It does not cover programming. See Chapter IV-2, **Programming Overview** for an introduction.

A procedure window is where Igor procedures are stored. Igor procedures are the macros, functions and menu definitions that you create or that Igor creates automatically for you.



The content of a procedure window is stored in a procedure file. In the case of a packed Igor experiment, the procedure file is packed into the experiment file.

Types of Procedure Files

There are four types of procedure files:

- The experiment procedure file, displayed in the built-in procedure window
- Auxiliary experiment procedure files, displayed in auxiliary procedure windows
- Shared procedure files, displayed in auxiliary procedure windows
- Global procedure files, displayed in auxiliary procedure windows

The built-in procedure window holds experiment-specific procedures of the currently open experiment. This is the only procedure window that beginning or casual Igor users may need.

All other procedure windows are called **auxiliary** to distinguish them from the built-in procedure window. You create an auxiliary procedure window using Windows→New→Procedure. You can then save it to a standalone file, using File→Save Procedure As, or allow Igor to save it as part of the current experiment.

An auxiliary experiment procedure file contains procedures that you want to use in a single experiment but want to keep separate from the built-in procedure window for organizational purposes. In a packed experiment it is saved as a packed file within the experiment file. In an unpacked experiment it is saved as a standalone file in the experiment folder.

A shared procedure file contains procedures that you want to use in more than one experiment but that you don't need in all experiments. It is always saved in a standalone file.

A global procedure file contains procedures that you might want to use in any experiment. It is always saved in a standalone file.

Working with the Built-in Procedure Window

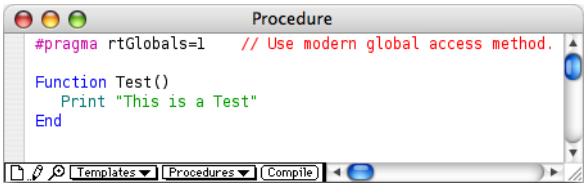
Procedures that are specific to the current experiment are usually stored in the built-in procedure window. Also, when Igor automatically generates procedures it stores them in the built-in procedure window.

To show the built-in procedure window, choose Procedure Window from the Procedure Windows submenu of the Windows menu or press Command-M (*Macintosh*) or Ctrl+M (*Windows*). To hide it, click the close button or press Command-W (*Macintosh*) or Ctrl+W (*Windows*).

To create a procedure, just type it into the procedure window.

The contents of the built-in procedure window are automatically stored when you save the current Igor experiment. For unpacked experiments the contents are stored in a file called "procedure" in the experiment folder. For packed experiments they are stored in the packed experiment file. When you open an experiment Igor loads its procedures back into the built-in procedure window.

Compiling the Procedures

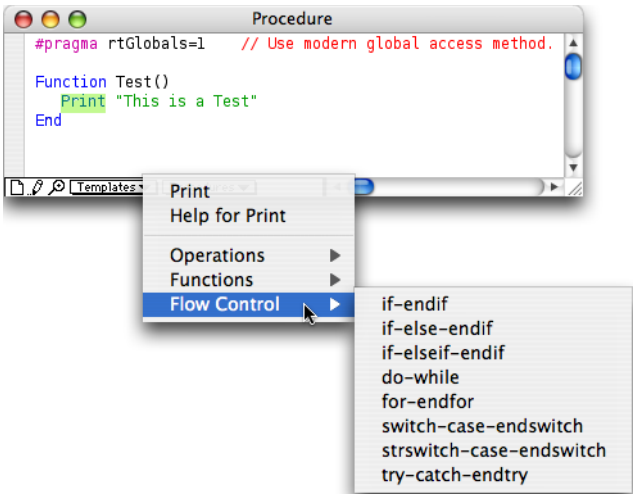


When you modify the text in the procedure window, you will notice that a Compile button appears at the bottom of the window.

Clicking the Compile button scans the procedure window looking for macros, functions and menu definitions. Igor compiles user-defined functions, generating low-level instructions that can be executed quickly.

Igor also compiles the code in the procedure window if you choose Compile from the Macros menu or if you activate any window other than a procedure or help window.

Templates Pop-Up Menu



The Templates pop-up menu lists all of the built-in and external operations and functions in alphabetical order and also lists the common flow control structures.

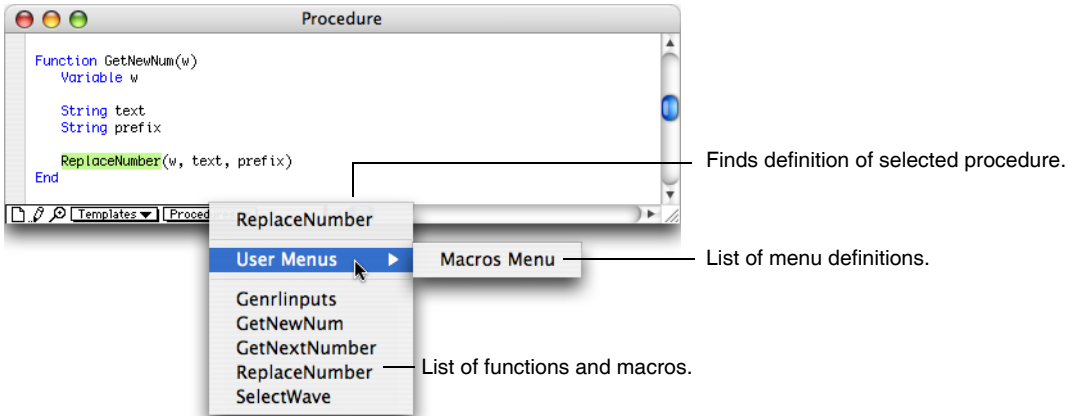
If you choose an item from the menu, Igor inserts the corresponding template in the procedure window.

If you select, click in, or click after a recognized operation, function or flow-control keyword in the procedure window, two additional items are listed at the top of the menu. The first item inserts a template and the second takes you to help.

If you select, click in, or click after a term for which Igor can provide a template, an item that inserts a template is added at the top of the menu. If Igor can provide help for the term, another item is added leading to the help. See **Procedure Window Shortcuts** on page III-354.

Procedures Pop-Up Menu

The Procedures pop-up menu provides a quick way for you to find procedures in the procedure window.



If you choose an item, Igor finds it. The menu normally lists just those procedures in the active window but if you press Option (*Macintosh*) or Alt (*Windows*), Igor will include procedures from all open procedure windows.

If you select, click in, or click after a recognized user procedure, an additional item is listed at the top of the menu which finds the selected procedure. Certain kinds of syntax errors prevent Igor from recognizing the selected procedure in which case, the menu will not include an item to find the procedure.

Magnifier Icon

You can magnify procedure text to make it more readable. See **Text Magnification** on page II-71 for details.

Write-Protect Icon

Procedure windows have a write-enable/write-protect icon which appears in the lower-left corner of the window and resembles a pencil. If you click this icon, Igor Pro will draw a line through the pencil, indicating that the procedure window is write-protected. The main purpose of this is to prevent accidental alteration of shared procedure files.

Igor opens included user procedure files for writing but turns on the write-protect icon so that you will get a warning if you attempt to change them. If you do want to change them, simply click the write-protect icon to turn protection off.

If a procedure file is opened for reading only, you will see a lock icon instead of the pencil icon. A file opened for read-only can not be modified.

WaveMetrics procedures, in the WaveMetrics Procedures folder, are assumed to be the same for all Igor users and should not be modified by users. Therefore, Igor Pro opens WaveMetrics procedures for reading only.

Creating Procedures

There are three ways to create procedures:

- Automatically by Igor
- Manually, when you type in a procedure window
- Semiautomatically, when you use various dialogs (e.g., the dialog that adds controls to a panel and the Curve Fitting dialog)

Igor offers to automatically create a **window recreation macro** when you close a target window. A window recreation macro is a procedure that can recreate a graph, table, page layout or control panel window. Igor always stores window recreation macros in the built-in procedure window. See **Saving a Window as a Recreation Macro** on page II-61 for details.

You can add procedures by merely typing them in a procedure window.

You can create user-defined controls in a graph or control panel. Each control has an optional **action procedure** that runs when the control is used. You can create a control and its corresponding action procedure using dialogs that you access through the Add Controls submenu in the Graph or Panel menus. These action procedures are initially stored in the built-in procedure window.

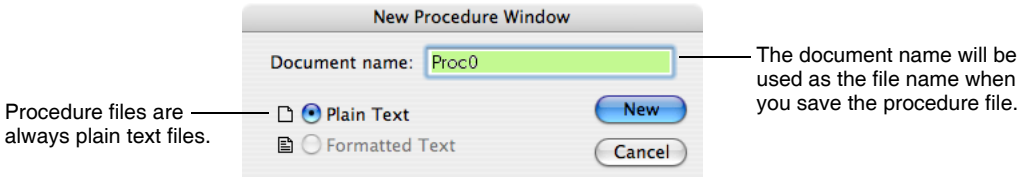
Creating New Procedure Files

You create a new procedure file if you want to write procedures to be used in more than one experiment.

Note: There is a risk in sharing procedure files among experiments. If you copy the experiment to another computer and forget to also copy the shared files, the experiment will not work on the other computer. See **References to Files and Folders** on page II-37 for further details.

If you do create a shared procedure file then you are responsible for copying the shared file when you copy an experiment that relies on it.

To create a new procedure file, choose Procedure from the New submenu of the Windows menu.



This creates a new procedure *window*. The procedure *file* is not created until you save the procedure window or save the experiment.

You can explicitly save the procedure window by choosing File→Save Procedure As or by closing it and choosing to save it in the resulting dialog. This saves the file as an auxiliary procedure file, separate from the experiment.

If you don't save the procedure window explicitly, Igor will save it as part of the current experiment the next time you save the experiment.

Opening an Auxiliary Procedure File

You can open a procedure file using the File→Open File→Procedure menu item.

When you open a procedure file, Igor displays it in a new procedure window. The procedures in the window can be used in the current experiment. When you save the current experiment, Igor will save a *reference* to the shared procedure file in the experiment file. When you later open the experiment, Igor will reopen the procedure file.

For commonly used auxiliary files, it is better to use the include statement than to explicitly open the files. See **Including a Procedure File** on page III-346.

Showing Procedure Windows

We usually hide procedure windows when we are not doing programming. To show the built-in procedure window, choose Procedure Window from the Procedure Windows submenu of the Windows menu or press Command-M (*Macintosh*) or Ctrl+M (*Windows*). To show auxiliary procedure windows, use the Windows→Procedure Windows menu item.

If you have more than one procedure window, you can cycle to the next procedure window by pressing Command-Option-M (*Macintosh*) or Ctrl+Alt+M (*Windows*). Pressing Command-Shift-Option-M (*Macintosh*) or Ctrl+Shift+Alt+M (*Windows*) hides the active procedure window and shows the next one.

You can also show a procedure window by choosing a menu item added by that window while pressing Option (*Macintosh*) or Alt (*Windows*). This feature works only if the top window is a procedure window.

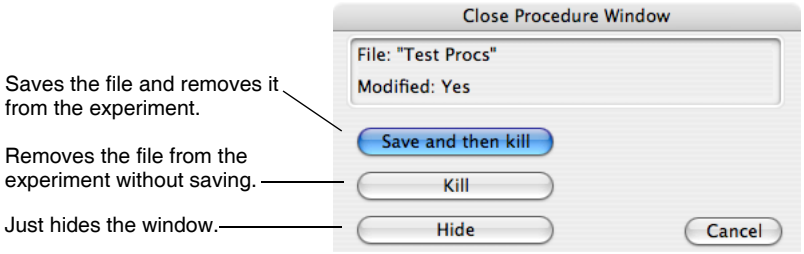
You can show all procedure windows and hide all procedure windows using the Windows→Show and Windows→Hide submenus.

Hiding and Killing Procedure Windows

The built-in procedure window always exists as part of the current experiment. You can hide it by clicking in the close button, pressing Command-W (*Macintosh*) or Ctrl+W (*Windows*) or by choosing Hide from the Windows menu. You can not kill it.

Auxiliary procedure files can be opened (added to the experiment), hidden and killed (removed from the experiment). This leads to a difference in behavior between auxiliary procedure windows and the built-in procedure window.

When you click the close button of an auxiliary procedure file, Igor presents the Close Procedure Window dialog to find out what you want to do.



If you just want to hide the window, you can press Shift while clicking the close button. This skips the dialog and just hide the window.

Killing a procedure window closes the window and removes it from the current experiment but does not delete or otherwise affect the procedure file with which the window was associated. If you have made changes or the procedure was not previously saved, you will be presented with the choice of saving the file before killing the procedure.

The Close item of the Windows menu and the equivalent, Command-W (*Macintosh*) or Ctrl+W (*Windows*), behave the same as the close button, as indicated in these tables.

Macintosh:

Actions	Modifier Key	Result
Click close button, choose Close, press Command-W		Displays dialog
Click close button, choose Close, press Command-W	Shift	Hides window
Click close button, choose Close, press Command-W	Option	Displays dialog

Windows:

Actions	Modifier Key	Result
Click close button, choose Close, press Ctrl+W		Displays dialog
Click close button, choose Close, press Ctrl+W	Shift	Hides window
Click close button, choose Close, press Ctrl+W	Alt	Displays dialog

Macintosh: When the Close Procedure Window dialog is showing, you can press Option to make the Kill button the default. The Kill button will become bold while the "Save and then kill" button will become normal. You can then press Return or Enter to kill the window. Similarly, press Shift to make the Hide button the default button.

Shared Procedure Files

You may develop procedures that you want to use in several but not all of your experiments. You can facilitate this by creating a shared procedure file. This is a procedure file that you save in its own file, separate from any experiment. Such a file can be opened from any experiment.

There are two ways to open a shared procedure file from an experiment: by explicitly opening it, using the File→Open File submenu or by adding an include statement to your experiment procedure window. The include method is preferred and is described in detail under **Including a Procedure File** on page III-346.

When Igor encounters an include statement, it searches for the included file in "Igor Pro Folder/User Procedures" and in "Igor Pro User Files/User Procedures" (see **Igor Pro User Files** on page II-46 for details). The Igor Pro User Files folder is the recommended place for storing user files. You can locate it by choosing Help→Show Igor Pro User Files.

You can store your shared procedure procedure file directly in "Igor Pro User Files/User Procedures" or you can store it elsewhere and put an alias (*Macintosh*) or shortcut (*Windows*) for it in "Igor Pro User Files/User Procedures". If you have many shared procedure files you can put them all in your own folder and put an alias/shortcut for the folder in "Igor Pro User Files/User Procedures".

When you explicitly open a procedure file using the Open File submenu, you are adding it to the current experiment. When you save the experiment, Igor saves a *reference* to the procedure file in the experiment file. When you close the experiment, Igor closes the procedure file. When you later reopen the experiment, Igor reopens the procedure file.

When you use an include statement, the included file is not considered part of the experiment but is still referenced by the experiment. Igor automatically opens the included file when it hits the include statement during procedure compilation.

Note: There is a risk in sharing procedure files among experiments. If you copy the experiment to another computer and forget to also copy the shared files, the experiment will not work on the other computer. See **References to Files and Folders** on page II-37 for more explanation.

Saving Shared Procedure Files

If you modify a shared procedure file, Igor saves it when you save the experiment that is sharing it. However, you might want to save the procedure file without saving the experiment. For this, choose File→Save Procedure.

Global Procedure Files

Global procedure files contain procedures that you want to be available in *all* experiments. They differ from other procedure files in that Igor opens them automatically and never closes them.

When Igor starts running, it searches "Igor Pro Folder/Igor Procedures" and "Igor Pro User Files/Igor Procedures" (see**Igor Pro User Files** on page II-46 for details), as well as files and folders referenced by aliases or shortcuts. Igor opens any procedure file that it finds during this search as a global procedure file.

You should save your global procedure files in "Igor Pro User Files/Igor Procedures". You can locate this folder by choosing Help→Show Igor Pro User Files.

Igor opens global procedure files with write-protection on since they presumably contain procedures that you have already debugged and which you don't want to inadvertently modify. If you *do* want to modify a global procedure file, click the write-protect icon (pencil in lower-left corner of the window).

When you create a new experiment or open an existing one, Igor normally closes any open procedure files, but it leaves global procedure files open. You can explicitly close a global procedure window at any time and then you can manually reopen it. Igor will not automatically reopen it until the next time Igor is launched.

Although its procedures can be used by the current experiment, a global procedure file is not part of the current experiment. Therefore, Igor does not save a global procedure file or a reference to a global procedure file inside an experiment file.

Note: There is a risk in using global procedure files. If you copy an experiment that relies on a global procedure file to another computer and forget to also copy the global procedure file, the experiment will not work on the other computer.

Saving Global Procedure Files

If you modify a global procedure file, Igor will save it when you save the current experiment even though the global procedure file is not part of the current experiment. However, you might want to save the procedure file without saving the experiment. For this, use the File→Save Procedure menu item.

Including a Procedure File

You can put an include statement in any procedure file. An include statement automatically opens another procedure file. This is the recommended way of accessing files that contain utility routines which you may want to use in several experiments. Using an include statement is preferable to opening a procedure file explicitly because it doesn't rely on the exact location of the file in the file system hierarchy.

Here is a typical include statement:

```
#include <MatrixToXYZ>
```

This automatically opens the MatrixToXYZ.ipf file supplied by WaveMetrics in "Igor Pro Folder/WaveMetrics Procedures/Data Manipulation". The angle brackets tell Igor to search the "Igor Pro Folder/WaveMetrics Procedures" hierarchy.

To see what WaveMetrics procedure files are available, choose Help→Help Windows→WM Procedures Index.

You can include your own utility procedure files by using double-quotes instead of the angle-brackets shown above:

```
#include "Your Procedure File"
```

The double-quotes tell Igor to search the "Igor Pro Folder/User Procedures" and "Igor Pro User Files/User Procedures" hierarchies (see **Igor Pro User Files** on page II-46 for details) for the named file. Igor searches those folders and subfolders and files or folders referenced by aliases/shortcuts in those folders.

These are the two main variations on the include statement. For details on less frequently used variations, see **The Include Statement** on page IV-145.

Included procedure files are not considered part of the experiment but are automatically opened by Igor when it compiles the experiment's procedures.

To prevent accidental alteration of an included procedure file, Igor opens it either write-protected (User Procedures) or read-only (WaveMetrics Procedures). See **Write-Protect Icon** on page III-342.

A #include statement must omit the file's ".ipf" extension, if it has one:

```
#include <Strings as Lists>      // RIGHT

#include <Strings as Lists.ipf>  // WRONG
```

See **Cross-Platform Procedure Compatibility** on page III-404 for details.

Creating Packages

A package is a set of procedure files, help files and other support files that add significant functionality to Igor.

Igor comes pre-configured with numerous WaveMetrics packages accessed through the Data→Packages, Analysis→Packages, Misc→Packages, Windows→New→Packages and Graph→Packages submenus as well as others.

Intermediate to advanced programmers can create their own packages. See **Packages** on page IV-222 for details.

Invisible Procedure Files

If you create a package of Igor procedures to be used by regular Igor users (as opposed to programmers), you may want to hide the procedures to reduce clutter or to eliminate the possibility that they might inadvertently change them. You can do this by making the procedure files invisible.

Invisible procedure files are omitted from Igor's Procedure Windows submenu which appears in the Windows menu. This keeps them out of the way of regular users.

There are three ways to make a procedure file invisible. In order of difficulty they are:

- Using the "#pragma hide" compiler directive
- Using an independent module
- Using the operating-system-supplied file visibility property

Invisible Procedure Windows Using #pragma hide

In Igor Pro 6.1 or later, you can make a procedure file invisible by inserting this compiler directive in the file:

#pragma hide=1

This prevents the procedure window from being listed in the Windows→Procedures submenu but only if the window is opened for read-only or is write-protected using Igor's write-protect icon (see **Write-Protect Icon** on page III-342). Since Igor automatically turns write-protect on for #included files, this method is easy to use with them.

Procedures windows that include this compiler directive and which are read-only or write-protected become invisible on the next compile.

You can make these windows visible during development by executing:

```
SetIgorOption IndependentModuleDev=1
```

and return them to invisible by executing:

```
SetIgorOption IndependentModuleDev=0
```

You must force a compile for this to take effect.

Invisible Procedure Windows Using Independent Modules

You can also make a set of procedure files invisible by making them an independent module. The independent module technique is more difficult to implement but has additional advantages. For details, see **The IndependentModule Pragma** on page IV-43.

Invisible Procedure Files Using The Files Visibility Property

This section discusses making procedure files invisible by setting the operating-system-supplied file "visible" property.

Note: This is an old technique that is no longer recommended. It may not be supported in future versions of Igor.

When Igor opens a procedure file, it asks the operating system if the file is invisible (*Macintosh*) or hidden (*Windows*). We will use the term "invisible" to mean invisible on Macintosh and hidden on Windows.

If the file is invisible, Igor makes the file inaccessible to the user. Igor checks the invisible property only when it opens the file. It does not pay attention to whether the property is changed while the file is open.

You create Igor procedures using normal visible procedure files, typically all in a folder or hierarchy of folders. When it comes time to ship to the end user, you set the files to be invisible. If you set a file to be invisible, you should also make it read-only.

You can use the **SetFileFolderInfo** operation (see page V-557) to set the visibility and read-only properties of a file:

```
SetFileFolderInfo /INV=1 /RO=1 "<path to file>"
```

The file will be invisible in Igor the next time you open it, typically by opening an experiment or using a #include statement.

The file will still be visible in the Macintosh Finder until you restart the Finder.

On Windows, merely setting the hidden property is not sufficient to actually hide the file. It is actually hidden only if the Hide Files of These Types radio button in the View Options dialog is turned on. You can access this dialog by opening a folder in the Windows desktop and choosing View→Options from the folder's menu bar. Although the hidden property in Windows does not guarantee that the file will be hidden in the Windows desktop, it does guarantee that it will be hidden from within Igor.

After the files are set to be invisible and read-only, if you want to edit them in Igor, you must close them (typically by closing the open experiment), set the files to be visible and read/write again, and then open them again.

Igor's behavior is changed in the following ways for a procedure file set to invisible:

1. The window will not appear in the Windows→Procedure Windows menu.
2. Procedures in the window will not appear in the Procedure pop-up menu at the bottom of all procedure windows.
3. Procedures in the window will not appear in the contextual pop-up menu in other procedure windows (Control-click on *Macintosh*, right-click on *Windows*).
4. If the user presses Option (*Macintosh*) or Alt (*Windows*) while choosing the name of a procedure from a menu, Igor will do nothing rather than its normal behavior of displaying the procedure window.
5. When cycling through procedure windows using Command-Shift-Option-M (*Macintosh*) or Ctrl+Shift+Alt+M (*Windows*), Igor will omit the procedure window.
6. The Button Control dialog, Pop-Up Menu Control dialog, and other control dialogs will not allow you to edit procedures in the invisible file.
7. The Edit Procedure and Debug buttons will not appear in the Macro Execute Error dialog.
8. If an error occurs in a function in the invisible file and the Debug On Error flag (Procedure menu) is on, the debugger will act as if Debug On Error were off.
9. The debugger won't allow you to step into procedures in the invisible file.
10. The **ProcedureText** function (see page V-506) and **DisplayProcedure** operation (see page V-118) will act as if procedures in the invisible file don't exist. (The **MacroList** and **FunctionList** functions will, however work as usual.)
11. The GetIgorProcedure and SetIgorProcedure XOPSupport routines in the XOP Toolkit will act as if procedures in the invisible file don't exist. (The GetIgorProcedureList function will, however work as usual.)

Inserting Text

On occasion, you may want to copy text from one procedure file to another. The Insert File item in the Edit menu makes this easy. With the procedure window active, choose Insert File. This will display a dialog in which you can find the file and will then insert its contents into the procedure window.

Adopting a Procedure File

Adoption is a way for you to copy a procedure file into the current experiment and break the connection to its original file. The reason for doing this is to make the experiment self-contained so that, if you transfer it to another computer or send it to a colleague, all of the files needed to recreate the experiment will be stored in the experiment itself.

To adopt a file, choose Adopt Window from the File menu. This item will be available only if the active window is a notebook or procedure file that is stored separate from the current experiment and the current experiment has been saved to disk.

If the current experiment is stored in packed form then, when you adopt a file, Igor does a save-as to a temporary file. When you subsequently save the experiment, the contents of the temporary file are stored in the packed experiment file.

If the current experiment is stored in unpacked form then, when you adopt a file, Igor does a save-as to the experiment's home folder. When you subsequently save the experiment, Igor updates the experiment's recreation procedures to open the new file in the home folder instead of the original file. If you adopt a file in an unpacked experiment and then you do not save the experiment, the new file will still exist in the home folder but the experiment's recreation procedures will still refer to the original file. Thus, you should normally save the experiment soon after adopting a file.

Adoption does not cause the original file to be deleted. You can delete it from the desktop if you want.

To "unadopt" a procedure file, choose Save Procedure File As from the File menu.

It is possible to do adopt multiple files at one time. For details see **Adopt All** on page II-38.

Auto-Compiling

If you modify a procedure window and then activate a nonprocedure window other than a help window, Igor automatically compiles the procedures. If you have a lot of procedures and compiling takes a long time, you may want to turn auto-compiling off.

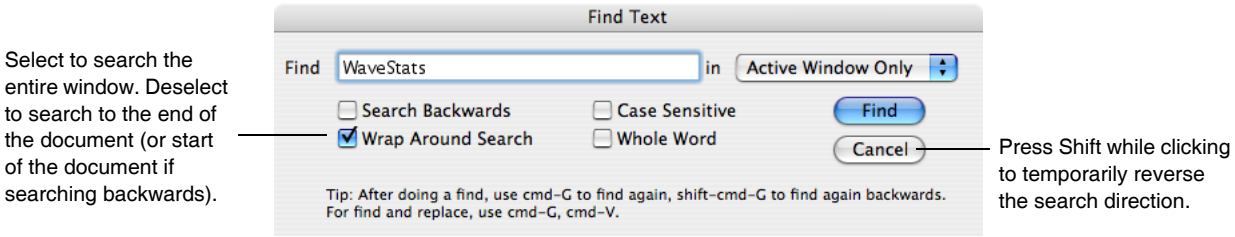
You can do this by deselecting the Auto-compile item in the Macros menu. This item appears only when the procedures need to be compiled (you have modified a procedure file or opened a new one). If you deselect this item, Igor will not auto-compile and compilation will be done only when you click the Compile button or choose Compile from the Macros menu.

Debugging Procedures

Igor includes a symbolic debugger. This is described in **The Debugger** on page IV-184.

Finding Text

You can access the Find Text dialog via the Edit menu or by pressing Command-F (*Macintosh*) or Ctrl+F (*Windows*).



You can search for the next occurrence of a string by selecting the string and pressing Command-Control-H (*Macintosh*) or Ctrl+H (*Windows*) (Find Selection in the Edit menu).

After doing a find, you can search for the same text again by pressing Command-G (*Macintosh*) or Ctrl+G (*Windows*) (Find Same in the Edit menu). You can search for the same text but in the reverse direction by pressing Command-Shift-G (*Macintosh*) or Shift+Ctrl+G (*Windows*).

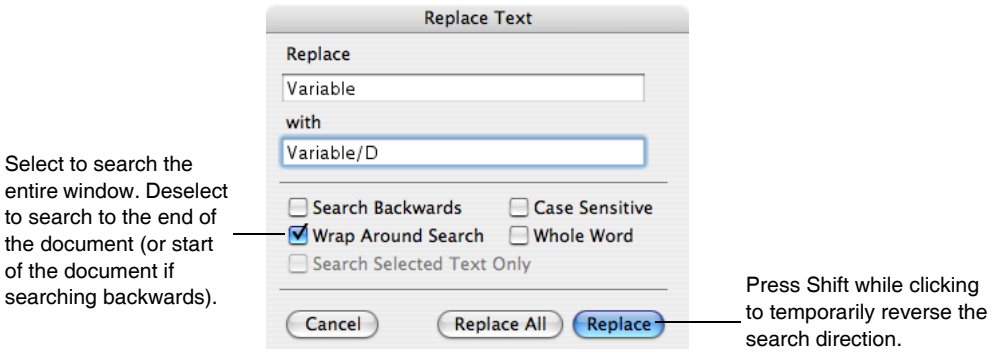
These keyboard shortcuts can be handy. For example, imagine that you are looking at the definition of a function and you want to see where it is used. You can double-click the name of the function to select it and then press Command-Control-H (*Macintosh*) or Ctrl+H (*Windows*) to search for the next occurrence. Then, you can press Command-G (*Macintosh*) or Ctrl+G (*Windows*) to find it again or Command-Shift-G (*Macintosh*) or Ctrl+Shift+G (*Windows*) to go back to the previous occurrence.

You can also perform a Find on multiple help, procedure and notebook windows at one time. See **Finding Text in Multiple Windows** on page II-69.

The Procedures pop-up menu at the bottom of the procedure window provides other ways to find the definition of a procedure in the same or in another procedure file. If you click the pop-up menu, you get a list of all procedures in the active procedure window. If you Option-click (*Macintosh*) or Alt click (*Windows*) on the pop-up menu, you get a list of all procedures in all procedure windows. If you select the name of a procedure and then click the pop-up menu, the first item in the pop-up menu will take you directly to the definition of that procedure, no matter what procedure file it is in.

Replacing Text

You can access the Replace Text dialog via the Edit menu or by pressing Command-R (*Macintosh*) or Ctrl+R (*Windows*).



The Search Selected Text Only option is handy for limiting the replacement to a particular procedure.

Replacing text is not undoable. Save the file before doing a mass replace so you can revert-to-saved if necessary.

Another method for searching and replacing consists of (*Macintosh*) using Command-F (Find) followed by a series of Command-V (Paste) and Command-G (Find Same) or on *Windows*, Ctrl+F followed by a series of Ctrl+V and Ctrl+G.

Printing Procedure Text

Each procedure file has its own page setup record which you can set using the Page Setup item of the File menu. You can use preferences (see **Procedure Window Preferences** on page III-352) to set the page setup record for new procedure windows.

As of Igor Pro 6.20, to make Igor start up faster, each opened help file and procedure file uses a copy of the same page setup record copied from the preferences for plain text notebooks. This change was made to cope with recent Macintosh HP drivers that take a very long time to create a new page setup record. This means that page setup records stored in help and procedure files loaded during launch are ignored.

You can print all or part of a procedure file.

Macintosh: To print all of the file, click so that no text is selected and then use the Print Procedure File item in the File menu. To print part of it, select the text to be printed and then use the Print Procedure Selection item in the File menu.

Windows: Choose File→Print Procedure File and use the Selection radio button in the Print dialog.

Indentation

We use indentation to indicate the structure of a procedure. This is described in **Indentation Conventions** on page IV-22.

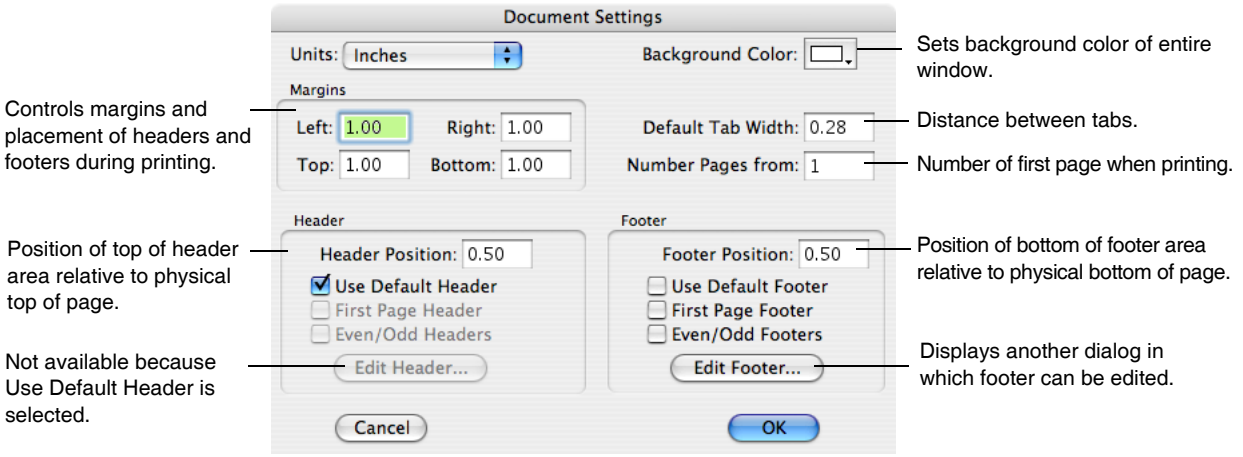
To make it easy to use the indentation conventions, Igor maintains indentation when you press Return or Enter in a procedure window. It automatically inserts enough tabs in the new line to have the same indentation as the previous line.

To indent more, as when going into the body of a loop, press Return or Enter and then Tab. To indent less, as when leaving the body of a loop, press Return or Enter and then Delete. When you don't want to change the level of indentation, just press Return.

Included in the Edit menu for Procedure windows, is the Adjust Indentation item, which adjusts indentation of all selected lines of text to match Igor standards. The Edit menu also contains Indent Left and Indent Right commands that add or remove indentation for all selected lines.

Document Settings

The Document Settings dialog controls settings that affect the procedure window as a whole. You can summon it via the Procedure menu.



Note: On *Windows* there is no way for Igor to store settings for an auxiliary procedure file, which is stored as a plain text file. When you open a procedure file or an experiment containing a procedure file on Windows, Igor uses preferences to set the procedure file's text format (text font, size, style). Thus, text format changes that you make to a procedure file are lost on Windows unless you capture them as your preferred format. The settings for the main procedure window are stored in the experiment file.

Syntax Coloring

The procedure editor colorizes comments, literal strings, flow control, and other function syntax elements. Colors of various elements can be adjusted using the following commands:

Command	Effect
`SetIgorOption colorize,doColorize=<1 or 0>`	Turn all colorize on or off
`SetIgorOption colorize,OpsColorized=<1 or 0>`	Turn operation keyword colorization on or off
`SetIgorOption colorize,BIFuncsColorized=<1 or 0>`	Turn function keyword colorization on or off
`SetIgorOption colorize,keywordColor=(r,g,b)`	Set color for language keywords
`SetIgorOption colorize,commentColor=(r,g,b)`	Set color for comments
`SetIgorOption colorize,stringColor=(r,g,b)`	Set color for strings
`SetIgorOption colorize,operationColor=(r,g,b)`	Set color for operation keywords
`SetIgorOption colorize,functionColor=(r,g,b)`	Set color for built-in function keywords
`SetIgorOption colorize,poundColor=(r,g,b)`	Set color for #keywords such as #pragma

The settings last only for the length of the Igor session. See **SetIgorOption** operation on page V-562.

Values for *r*, *g*, and *b* range from 0 to 65535.

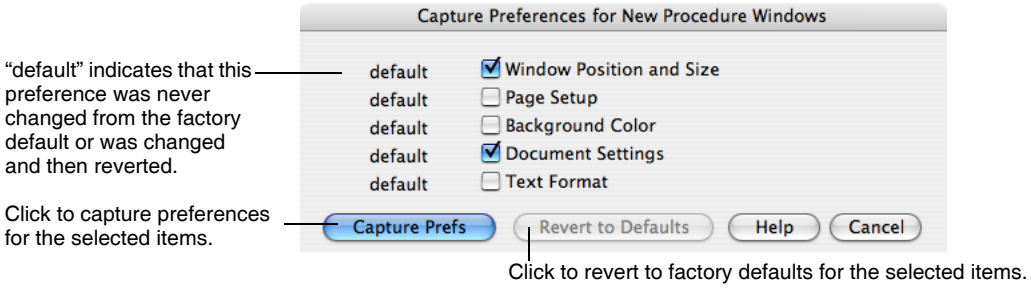
Text Character Settings

You can specify the font, text size, style and color using items in the Procedure menu. Since procedure windows are always plain text windows (as opposed to notebooks, which can be formatted text) these text settings are the same for all characters in the window. On Macintosh, text format settings are stored in the resource fork of the procedure file. On Windows, where there is no resource fork, text format settings are not stored and revert to the preferred settings when a procedure window is opened.

Procedure Window Preferences

The procedure window preferences affect the creation of *new* procedure windows. This includes the creation of auxiliary procedure windows and the initialization of the built-in procedure window that occurs when you create a new experiment.

To set procedure preferences, set the attributes of any procedure window and then use the Capture Procedure Prefs item in the Procedure menu.



To determine the current preference settings, you must create a new procedure window and examine its settings.

Procedure windows each have their own Page Setup values. New procedure windows will have their own copy of the captured (or default) Page Setup values.

Preferences are stored in the Igor Preferences file. See Chapter III-17, **Preferences**, for further information on preferences.

Double and Triple-Clicking

Double-clicking a word conventionally selects the entire word. Igor extends this idea a bit. In addition to supporting double-clicking, if you triple-click in a line of text, it selects the entire line. If you drag after triple-clicking, it extends the selection an entire line at a time.

Matching Characters

Igor includes a handy feature to help you check parenthesized expressions. If you double-click a parenthesis, Igor tries to find a matching parenthesis on the same line of text. If it succeeds, it selects all of the text between the parentheses. If it fails, it beeps. Consider the command

```
wave1 = exp(log(x))
```

If you double-clicked on the first parenthesis, it would select "`log(x)`". If you double-clicked on the last parenthesis, it would beep because there is no matching parenthesis.

If you double-click in-between adjacent parentheses Igor considers this a click on the outside parenthesis.

Igor does matching if you double-click the following characters:

Left and right parentheses	(xxx)
Left and right brackets	[xxx]
Left and right braces	{xxx}
Plain single quotes	'xxx'
Plain double quotes	"xxx"
Typographic single quotes	‘xxx’*
Typographic double quotes	“xxx”*

* No matching is done on two-byte typographic quotes, which are used in Asian fonts.

Code Comments

The Edit menu for procedures contains two items, Commentize and Decommentize, to help you edit and debug your procedure code when you want to comment out large blocks of code, and later, to remove these comments. Commentize inserts comment symbol at the start of each selected line of text. Decommentize deletes any comment symbols found at beginning of each selected line of text.

UTF-16 Files

You can open UTF-16 (two-byte Unicode) text files as procedure windows. Igor does not recognize non-ASCII characters, but does ignore the byte-order mark at the start of the file (BOM) and null bytes contained in UTF-16 text files. If you open a UTF-16 file and then save it from Igor, it will be saved as plain ASCII, not UTF-16, and some information may be lost.

Procedure Window Shortcuts

To view text window keyboard navigation shortcuts, see **Text Window Navigation** on page II-68.

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To show the built-in procedure window	Press Command-M.	Press Ctrl+M.
To hide the active procedure file and cycle to the next	Press Command-Shift-Option-M.	Press Ctrl+Shift+Alt+M.
To cycle through the open procedure windows	Press Command-Option-M.	Press Ctrl+Alt+M.
To get a contextual menu of commonly-used actions	Press Control and click in the body of the procedure window.	Right-click the body of the procedure window.
To execute commands in a procedure window	Select the commands or click in the line containing the commands and press Control-Return or Control-Enter.	Select the commands or click in the line containing the commands and press Ctrl+Enter.
To insert a template	Type or select the name of an operation or function and press Shift-Help.	Type or select the name of an operation or function and press Ctrl+F1.
	Control-click the name of an operation or function.	Right-click the name of an operation or function.
To get help for an operation or function	Type or select the name of an operation or function and press Shift-Option-Help.	Type or select the name of an operation or function and press Ctrl+Alt+F1.
	Control-click the name of an operation or function.	Right-click the name of an operation or function.
To find a procedure in the active procedure window	Click the Procedures pop-up menu at bottom of the procedure window.	Click the Procedures pop-up menu at bottom of the procedure window.
To find a procedure in any procedure window	Press Option and click the Procedures pop-up menu at bottom of any procedure window.	Press Alt and click the Procedures pop-up menu at bottom of any procedure window.
To find the definition of a procedure when you have selected an invocation of it	Click the Procedures pop-up menu at bottom of the procedure window or press Shift-Option-Help.	Click the Procedures pop-up menu at bottom of the procedure window or press Ctrl+Alt+F1.
To find the same text again	Press Command-G.	Press Ctrl+G.
To find again but in the reverse direction	Press Command-Shift-G.	Press Ctrl+Shift+G.
To find the selected text	Press Command-Control-H.	Press Ctrl+H.
To find the selected text but in the reverse direction	Press Command-Control-Shift-H.	Press Ctrl+Shift+H.

Action	Shortcut (*Macintosh*)	Shortcut (*Windows*)
To find a user-defined menu's procedure	Open any procedure window and press Option while selecting a user-defined menu item.	Open any procedure window and press Alt while selecting the user-defined menu item.
To select a word	Double-click.	Double-click.
To select an entire line	Triple-click.	Triple-click.

Chapter
III-14

Controls and Control Panels

Overview

We use the term *controls* for a number of user-programmable objects that can be employed by Igor programmers to create a graphical user interface for Igor users. We call them *controls* even though some of the objects only display values. The term *widgets* is sometimes used by other application programs, especially on non-Macintosh systems.

Here is a summary of the types of controls available.

Control Type	Control Description
Button	Calls a procedure that the programmer has written.
Chart	Emulates a mechanical chart recorder. Charts can be used to monitor data acquisition processes or to examine a long data record. Programming a chart is quite involved.
CheckBox	Sets an off/on value for use by the programmer's procedures.
CustomControl	Custom control type. Completely specified and modified by the programmer.
GroupBox	An organizational element. Groups controls with a box or line.
ListBox	Lists items for viewing or selecting.
PopupMenu	Used by the user to choose a value for use by the programmer's procedures.
SetVariable	Sets and displays a numeric or string global variable. The user can set the variable by clicking or typing. For numeric variables, the control can include up/down buttons for incrementing/decrementing the value stored in the variable.
Slider	Duplicates the behavior of a mechanical slider. Selects either discrete or continuous values.
TabControl	Selects between groups of controls in complex panels.
TitleBox	An organizational element. Provides explanatory text or message.
ValDisplay	Presents a readout of a numeric expression which usually references a global variable. The readout can be in the form of numeric text or a thermometer bar or both.

The programmer can specify a procedure to be called when the user clicks on or types into a control. This is called the control's *action procedure*. For example, the action procedure for a button may interrogate values in pop-up menu, checkbox, and SetVariable controls and then perform some action.

Control panels are simple windows that contain these controls. These windows have no other purpose. You can also place controls in graph windows and in panel panes embedded into graphs. Controls are not available in any other window type such as tables, notebooks, or layouts. When used in graphs, controls are not considered part of the *presentation* and thus are **not** included when a graph is printed or exported.

Nonprogrammers will want to skim only the Modes of Operation and Using Controls sections, and skip the remainder of the chapter. Igor programmers should study the entire chapter.

Modes of Operation

With respect to controls, there are two modes of operation: one mode to use the control and another to modify it. To see this, choose Show Tools from the Graph or Panel menu. Two icons will appear in the top-left corner window. When the top icon is selected, you are able to use the controls. When the next icon is selected, the draw tool palette appears below the second icon. To modify the control, select the arrow tool from the draw tool palette.

When the top icon is selected or when the icons are hidden, you are in the *use* or *operate* mode. You can momentarily switch to the *modify* or *draw* mode by pressing Command-Option (*Macintosh*) or Ctrl+Alt (*Windows*). Use this to drag or resize a control as well as to double-click it. Double-clicking

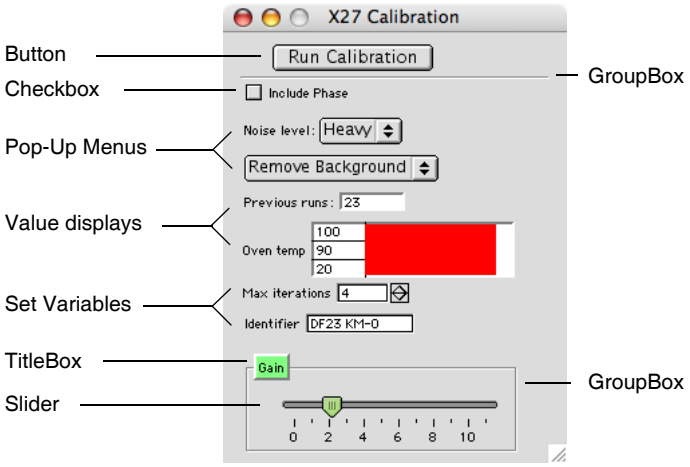
with the Command-Option (*Macintosh*) or Ctrl+Alt (*Windows*) pressed brings up a dialog that you use to modify the control.

You can also switch to modify mode by choosing an item from the Select Control submenu of the Graph or Panel menu.

Important:To enable the Add Controls submenu in the Graph and Panel menus, you must be in modify mode; either by clicking the second icon or by pressing Command-Option (*Macintosh*) or the Ctrl+Alt (*Windows*) while choosing the Add Controls submenu.

Using Controls

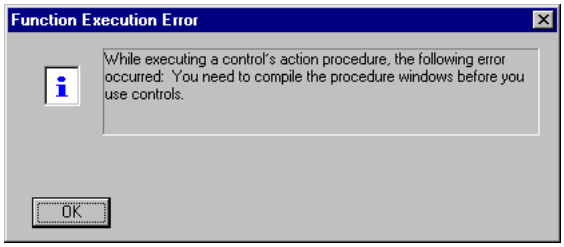
The following panel window illustrates most of the control types.



Buttons

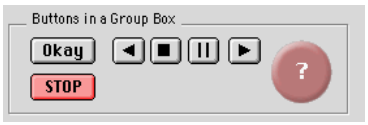
When you click a button, it runs whatever procedure the programmer may have specified.

If nothing happens when you click a button, then there is no procedure assigned to the button. If the procedure window(s) haven't been compiled, clicking a button that has an assigned procedure will produce this dialog:



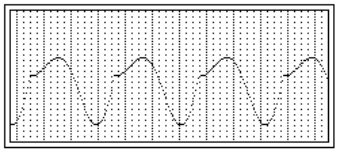
You should choose Compile from the Macros menu to correct this situation. If no error occurs then the button will now be functional.

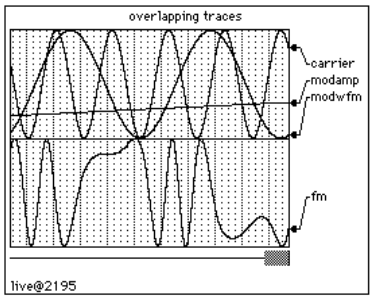
Buttons usually have a rounded appearance, but a programmer can assign a custom picture so that the button can have nearly any appearance.



Charts

Chart controls can be used to emulate a mechanical chart recorder that writes on paper with moving pens as the paper scrolls by under the pens. Charts can be used to monitor data acquisition processes or to examine a long data record.





Note: Although programming a chart is quite involved, using a chart is actually very easy. However, since most users will never use a chart control, their use is described in **Charts and FIFOs** on page III-369.

Checkboxes

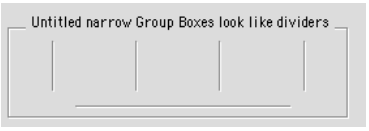
Clicking a checkbox changes its selected state and may run a procedure if the programmer specified one. A checkbox may be connected to a global variable. Checkboxes can be configured to look and behave like radio buttons.

CustomControl

CustomControls are used to create completely new types of controls that are custom-made by the programmer. You can define and control the appearance and all aspects of a custom control's behavior. See **CustomControl** on page III-371 for examples.

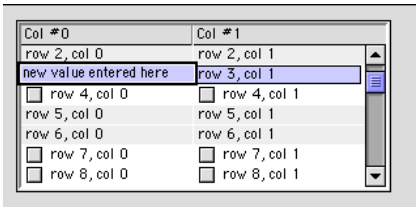
GroupBox

GroupBox controls are organizational or decorative elements. They are used to graphically group sets of controls. They may either draw a box or a separator line and can have optional titles.



ListBox

ListBox controls can present a single or multiple column list of items for viewing or selection. ListBoxes can be configured for a variety of selection modes. Items in the list can be made editable and can be configured as checkboxes.



Pop-Up Menus

These controls come in two forms: one where the current item is shown in the pop-up menu box

and another where there is no current item and a title is shown in the box.

The first form is usually used to choose one of many items while the second is used to run one of many commands.

Pop-up menus can also be configured to act like Igor's color, line style, pattern, or marker pop-up menus; these always show the current item.

Set Variable

Set Variable controls also can take on a number of forms and can display numeric values. Unlike Value Display controls that display the value of an expression, Set Variable controls are connected to individual global variables and can be used to set or change those variables in addition to reading out their current value. Set Variable controls can also be used with global string variables to display or set short one line strings. Set Variable controls are automatically updated whenever their associated variables are changed.

When connected to a numeric variable, these controls can optionally have up or down arrows that increment or decrement the current value of the variable by an amount specified by the programmer. Also, the programmer can set upper and lower limits for the numeric readouts.

New values for both numeric and string variables can be entered by directly typing into the control. If you click the control once you will see a thick border form around the current value.

You can then edit the readout text using the standard techniques including Cut, Copy, and Paste. If you want to discard changes you have made, press Escape. To accept changes, press Return, Enter, or Tab or click anywhere outside of the control. Tab enters the current value and also takes you to the next control if any. Shift-Tab is similar but takes you to the previous control if any.

If the control is connected to a numeric variable and the text you have entered can not be converted to a number then a beep will be emitted when you try to enter the value and no change will be made to the value of the variable. If the value you are trying to enter exceeds the limits set by the programmer then your value will be replaced by the nearest limit.

When a numeric control is selected for editing, the Up and Down Arrow keys on the keyboard act like the up and down buttons on the control.

Changing a value in a Set Variable control may run a procedure if the programmer has specified one.

Set Variable Controls and Data Folders

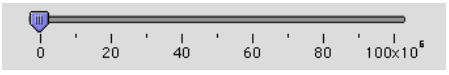
Note: If you are not using Data Folders (described in Chapter II-8, **Data Folders**), you can skip this section.

Set Variable controls remember the data folder in which the variable exists, and continue to function properly when the current data folder is different than the controlled variable. See **Set Variable** on page III-362.

The system variables (K0 through K19) belong to no particular data folder (they are available from any data folder), and there is only *one* copy of these variables. If you create a SetVariable controlling K0 while the current data folder is "aFolder", and another SetVariable controlling K0 while the current data folder is "bFolder", *they are actually controlling the same K0.*

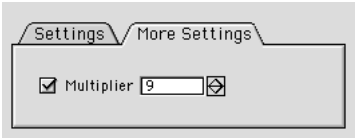
Sliders

Slider controls can be used to graphically select either discrete or continuous values. When used to select discrete values, a slider is similar to a pop-up menu or a set of radio buttons. Sliders can be live, updating a variable or running a procedure as the user drags the slider, or they can be configured to wait until the user finishes before performing any action.



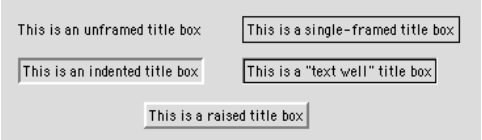
TabControl

TabControls are used to create complex panels containing many more controls than would otherwise fit. When the user clicks on a tab, the programmers procedure runs and hides the previous set of controls while showing the new set.



TitleBox

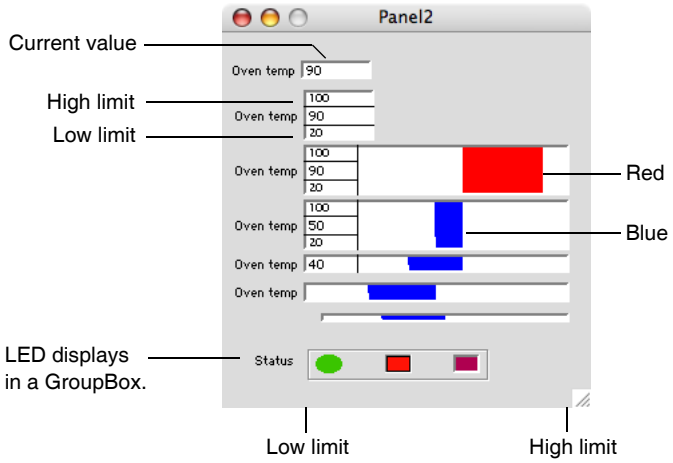
TitleBox controls are mainly decorative elements. They are used provide explanatory text in a control panel. They may also be used to display textual results. The text can be unchanging, or can be the contents of a global string variable. In either case, the user can't inadvertently change the text.



Value Displays

These can take on a number of forms ranging from a simple numeric readout to a thermometer bar. Regardless of the form, value displays are just readouts. There is no interaction with the user. They display the current value of whatever expression the programmer specified. Often this will be just the value of a numeric variable, but it can be any numeric expression including calls to user-defined functions and external functions.

Here is a sampling of the forms that Value Display controls can assume.

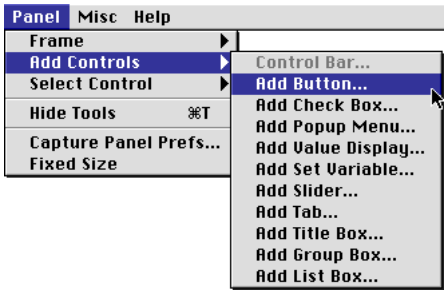


When a thermometer bar is shown, the left edge of the thermometer region represents a low limit set by the programmer while the right edge represents a high limit. The low and high limits appear in some of the above examples. The bar is drawn from a nominal value set by the programmer and will be red if the current value exceeds the nominal value and will be blue if it is less than the nominal value. In the above examples the nominal value is 60. There is no numeric indication of the nominal value. If the nominal value is less than the low limit then the bar will grow from the left to the right. If the nominal value is greater than the high limit then the bar will grow from the right to the left.

If you carefully observe a thermometer bar that is connected to an expression whose value is slowly changing with time you will see that the bar is drawn in a zig-zag fashion. This provides a much finer resolution than if the bar were to be extended or contracted by an entire column of screen pixels at once.

Creating Controls

The ease of creating the various controls varies widely. Anyone capable of writing a simple macro can create Buttons and Checkboxes, but creating Charts and CustomControls requires more expertise. Most controls can be created and modified using dialogs. You will find these dialogs under the Add Controls submenu in the Graph or Panel main menu.



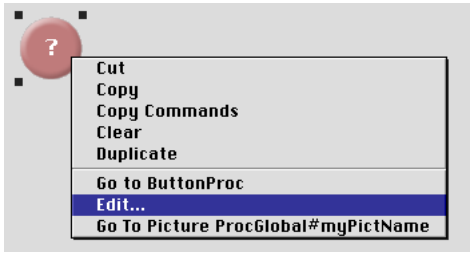
The Add Controls and Select Control menus are disabled until the arrow tool in the toolbar is enabled. (You use the toolbar's arrow tool to position and resize controls.) To do this, choose Show Tools from the Graph or Panel main menu and then click the second icon from the top (in the graph or panel's tool bar).

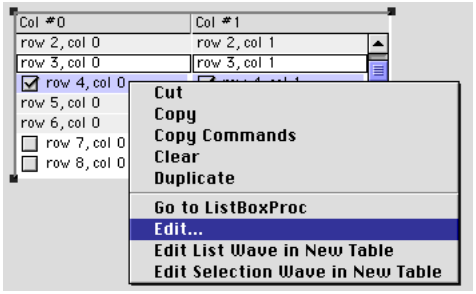
Note: You can also temporarily use the arrow tool without the toolbar showing by pressing Command-Option (*Macintosh*) or Ctrl+Alt (*Windows*). Holding down these keys while selecting from the Graph or Panel menu, you can also choose from the normally-disabled Add Controls and Select Control menus.

When you click a control with the arrow tool, small handles are drawn that allow you to resize the control. Note that some controls can not be resized in this way and some can only be resized in one dimension. You will know this when you try to resize a control and it doesn't budge. You can also use the arrow tool to reposition a control. You can select a control by name with the Select Control submenu in the Graph or Panel menu.

With the arrow tool, you can double-click most controls to get a dialog that modifies or duplicates the control. Charts and CustomControls do not have dialog support.

When you right-click (*Windows*) or Control-click (*Macintosh*) a control, you get a contextual menu that varies depending on the type of control.





You can select multiple controls, mix selections with draw objects and perform operations such as move, cut, copy, paste, delete with undo and align. You can't group buttons or use Send to Back as you can with Igor's draw objects.

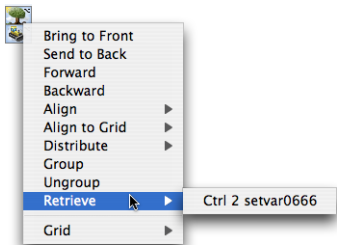
In panels, when you do a Select All it includes all controls and draw objects, but in the case of graphs, only draw objects are selected. This is because draw objects in graphs are used for presentation graphics whereas in panels they are used to construct the user interface.

If you want to copy controls from one window to another, simply use the Edit menu to copy and paste.

Note: If user controls are copied to the Clipboard, then the command and control names are also copied as text. This is handy when modifying controls that have no dialog support.

Hold down Option (*Macintosh*) or Alt (*Windows*) while choosing the Copy command to copy the complete set of procedures that create the copied controls.

If you copy a control from the extreme right side or bottom of a window, it may not be visible when you paste it into a smaller window. Use the smaller window's Retrieve submenu in the Mover menu to make it visible.



General Command Syntax

All of the control commands use the following general syntax:

```
ControlOperation Name [,keyword[=value] [,keyword[=value]]…]
```

Name is the control's name; it must be unique to the window containing the control. If *Name* is not already in use then a new control will be created. If a control with the same name already exists then that control will be modified, so that multiple commands using the same name result in only one control. This is useful for creating controls that require many keywords.

All keywords are optional. Not all controls accept all keywords, and some controls accept a keyword but do not actually use the value(s). The value for a keyword with one control can have a different form than the value for the same keyword used with other controls; the "value" keyword is a prime example of this. See the specific control operation documentation in Chapter V-1, **Igor Reference** for details.

Some controls utilize a format keyword to set a format string. The format string can be any printf style format that expects a single numeric value. Think of the output as being the result of the following command:

```
printf formatString , value_being_displayed
```

See the **printf** operation on page V-499 for a discussion of printf format strings. The maximum length of the format string is 63. The format is used only for controls that display numeric values and only for the principal value within a control (e.g., not used for the limit values for the ValDisplay control)

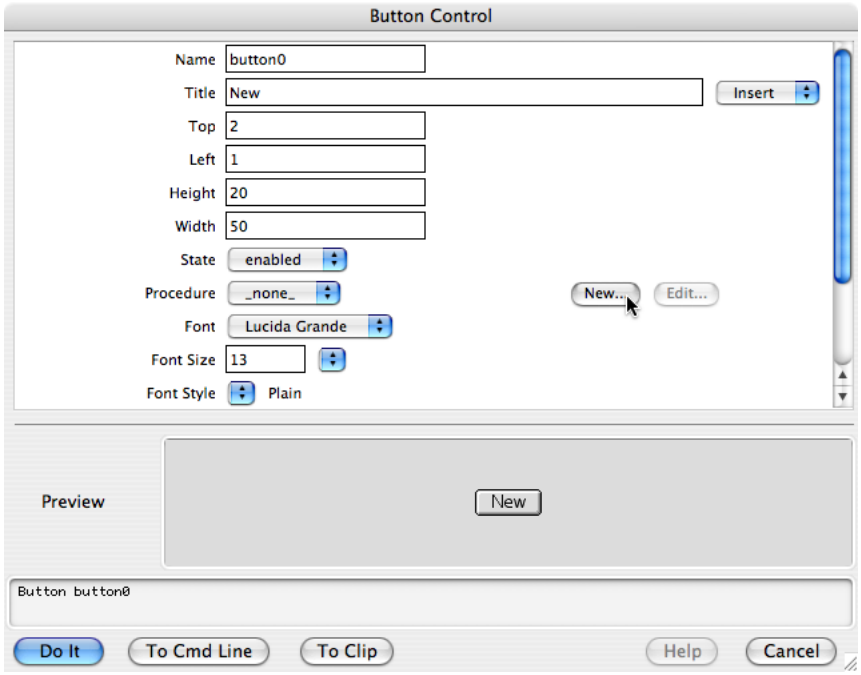
All of the clickable controls can optionally call a user-defined function usually when the user *releases* the mouse button. We use the term *action procedure* for such a function or macro. Each control passes one or more parameters to the action procedure. The dialogs for each control can create a blank user function with the correct parameters.

Read the following section on Buttons for general techniques that apply to all controls. You should also refer to Chapter V-1, **Igor Reference**, for each of the control-related operations and functions.

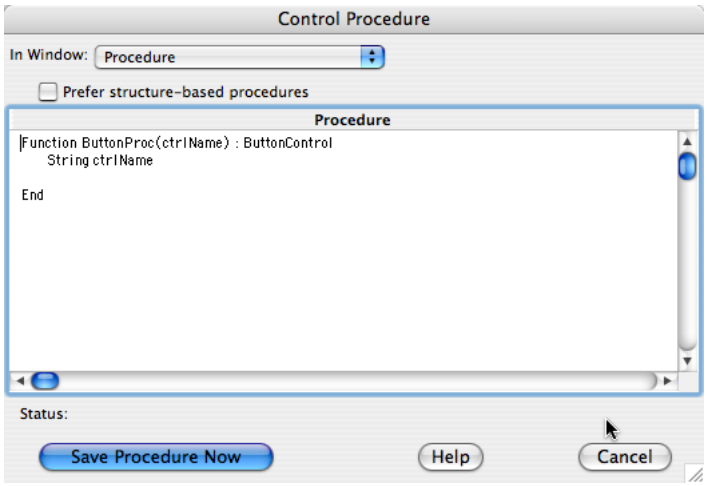
Button

The **Button** operation (page V-38) creates or modifies a rounded-edge or custom button with the title text centered in the button. The default font depends on the operating system, but you can change the font, font size, text color and use annotation-like escape codes (see **About Text Escape Codes** on page III-44). The amount of text does not change the button size, which you can set to what you want. However, the size of a custom button is determined only by the size of its Proc picture.

Here we create a simple button that will just emit a beep when pressed. Start by choosing the Add Button menu item in the Graph or Panel menu to get the Button Control dialog:



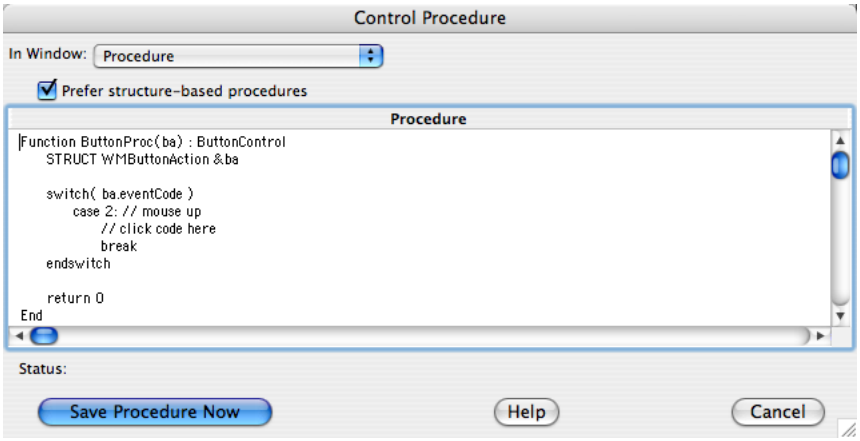
Clicking the procedure's New button brings up a dialog containing a procedure template that you can edit, rename, and save:



The controls can work with procedures using two formats: the "classic" procedure format used in Igor 3 and 4, and the "structure-based" format introduced in Igor 5.

Selecting the "Prefer structure-based procedures" checkbox creates new procedure templates using the structure-based format.

Tip: If you haven't edited a template (or if you delete the template), selecting or deselecting the checkbox will switch the template between the two formats:



Click Help to get help about the type of control the procedure works with. In this example, clicking Help would show the help file for the Button operation, which lists the details about the WMButtonAction structure in the Details section.

All we did in the above dialogs was click the New Procedure button, change the function name from `ButtonProc` to `MyBeepProc` and add the Beep command.

The fact that you can create the action procedure for a control in a dialog may lead you to believe that the procedure is stored with the button. This is not true. The procedure is actually stored in a procedure window. This way you can use the same action procedure for several controls. The parameters which are passed to a given procedure can be used to differentiate the individual controls.

As you can see from the above example, the user defined action procedure that you will need to write for buttons must have the following form:

```
Function ButtonProc(ctrlName) : ButtonControl
   String ctrlName

End
```

Replace `ButtonProc` with a descriptive name and fill in the body of the function. The `ButtonControl` subtype at the end of the function declaration line is optional but highly recommended. If it is present then the function will show up in the pop-up list of available procedures in the Button Control dialog. The other controls have similar subtypes.

It is legal for the action procedure to be written as a Macro (or Proc) rather than as a Function, but Functions are much faster than Macros.

You can use an additional kind of action procedure whose input parameter is a control-specific data structure. The Control Procedure dialog allows you to create either kind of action procedure. For more details, see **Control Structures** on page III-384, the **Button** operation (page V-38), and **Using Structures with Windows and Controls** on page IV-82.

Button Example

Here is how to make a button whose title alternates between Start and Stop.

Enter the following in the Procedure window:

```
Function StartStopButton(ctrlName) : ButtonControl
   String ctrlName

   if( cmpstr(ctrlName,"bStart") == 0 )
      Button $ctrlName,title="Stop",rename=bStop
      MyStartProc()  // or whatever you want when start is pressed
   else
      Button $ctrlName,title="Start",rename=bStart
```

```
      MyStopProc()    // or whatever you want when stop is pressed
   endif
End
```

Additionally, you will also need to create the functions `MyStartProc()` and `MyStopProc()` to actually do something when the button is clicked.

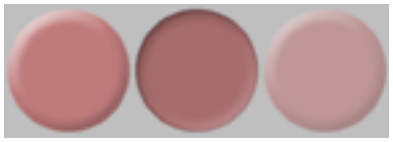
Then create the button in a graph or panel window with:

```
Button bStart,size={50,20},proc=StartStopButton,title="Start"
```

Custom Button Example

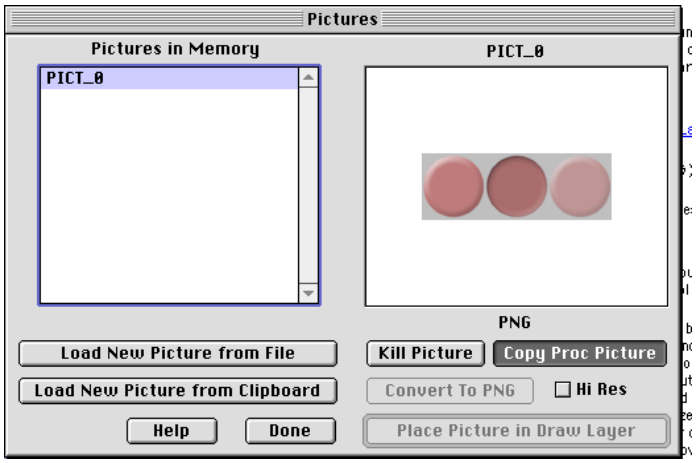
You can create custom buttons by following these steps:

1) Using a graphics-editing program, create a picture that shows the button in it's normal ("relaxed") state, then in the pressed-in state, and then in the disabled state. Each portion of the picture should be the same size:

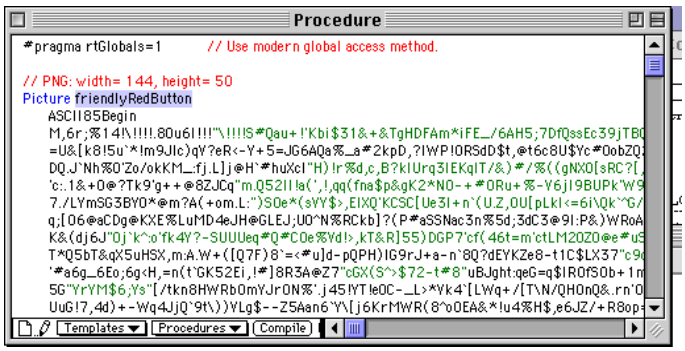


If the button blends into the background it will look better if the buttons are created on the background you will use in the panel. Igor looks at the pixels in the upper left corner, and if they are a light neutral color, Igor will omit those pixels when the button is drawn.

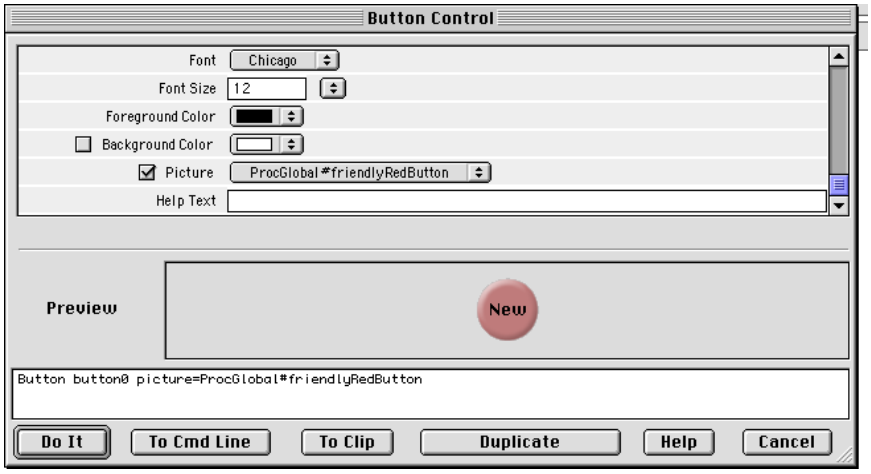
2) Copy the picture to the Clipboard.
3) Switch to Igor and open the Pictures dialog, and click Load New Picture from Clipboard.



4) Click Copy Proc Picture to create Proc Picture text on the Clipboard. Click Done.
5) Select a procedure window, paste the text, and give a suitable name to the picture:



6) Choose this picture in the Button Control dialog:

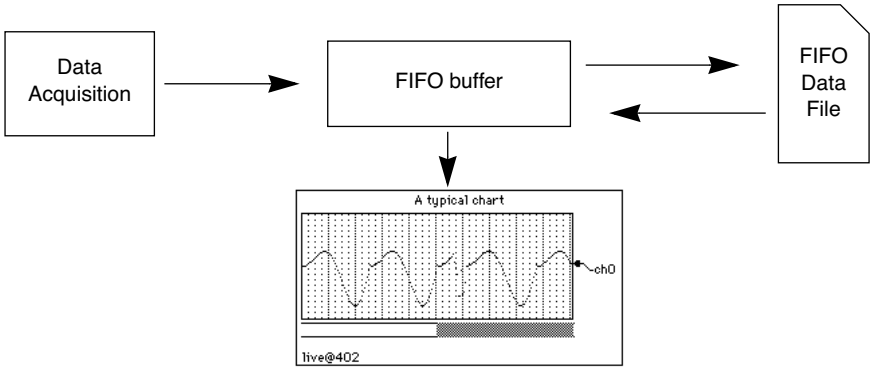


Charts and FIFOs

For further details see **FIFOs and Charts** on page IV-276.

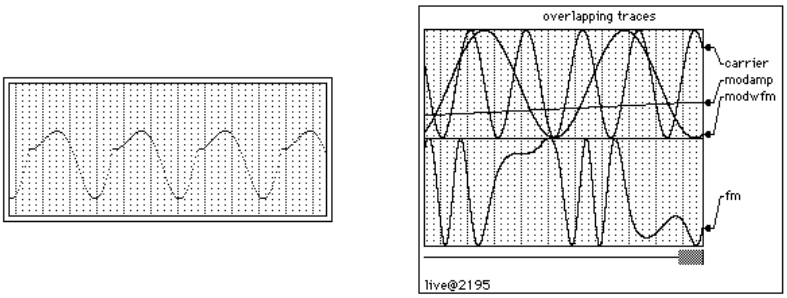
Chart controls can be used to emulate a mechanical chart recorder that writes on paper with moving pens as the paper scrolls by under the pens. Charts can be used to monitor data acquisition processes or to examine a long data record. Although programming a chart is quite involved, using a chart is very easy.

However, users of the Chart control do need to know the basics of the data acquisition process:



The First-In-First-Out (FIFO) buffer is an invisible Igor component that buffers the data coming from data acquisition hardware and software and also writes the data to a file. The data that is streaming through the FIFO can be observed using a Chart control. When data acquisition is finished the process can be reversed with data coming back out of the file and into the FIFO where it can be reviewed using the Chart. The FIFO file is optional but if missing then all data pushed out the end of the FIFO will be lost.

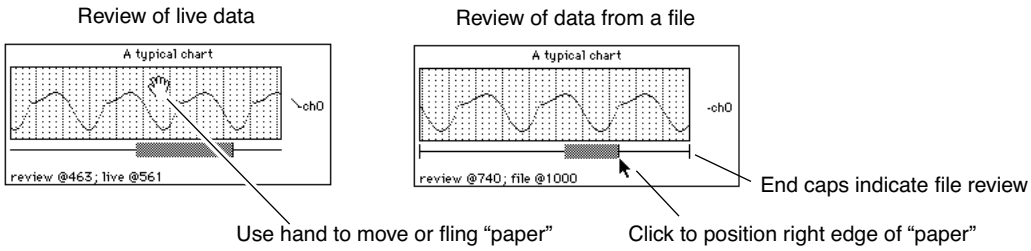
Chart controls can take on quite a number of forms from the simple to the sophisticated:



Charts can operate in two modes — live and review. When a chart is in live mode and data acquisition is in progress, the chart "paper" will scroll by from right to left under the influence of the acquisition process. When in review mode, you are in control of the chart. When you position the mouse over the chart area you will see that the cursor turns into a hand. You can move the chart paper right or left by dragging with the hand. If you give the paper a push it will continue scrolling until it hits the end. You can place the chart in review mode even as data acquisition is in progress by clicking in the paper with the hand cursor. To go back to live mode, give the paper a hard push to the left. When the paper hits the end then the chart will go to into live mode. You can also go back to live mode by clicking anywhere in the margins of the Chart.

Depending on the exact details of the data acquisition hardware and software you may run the risk of corrupting the data if you use review mode while acquisition is in progress. The person that created the hardware and software system you are using should have provided guidelines for the use of review mode during acquisition. In general, if the acquisition process is paced by hardware then it should be OK to use review mode.

In the above chart with lots of bells and whistles you may have noticed the line directly under the scrolling paper area. This line represents the current extent of data while the gray bar represents the data that is being shown in the chart. The right edge of the gray bar represents the right edge of the section of data being shown in the chart window. The above example is shown in live mode. Here are two examples shown in review mode:



While data acquisition is in progress, the horizontal line represents the extent of the data in the FIFO's memory. After acquisition is over then the line includes all of the data in the FIFO's output file (if any).

If you are in review mode while data acquisition is taking place you will notice that the gray bar will indicate the view area is moving even though the paper appears to be motionless. This is because the FIFO is moving out from under the chart. Eventually it will reach a position where the chart display can not be valid since the data it wants to display has been flushed off the end of the FIFO. When this happens the view area will go blank. Because it is very time-consuming for Igor to try to keep the chart updated in this situation your data acquisition rate may suffer.

CheckBox

The **CheckBox** operation (page V-46) creates or modifies a checkbox or a radio button. CheckBox controls automatically size themselves in both height and width. Checkboxes can optionally be connected to a global variable. For information on using checkboxes as radio buttons, see the example in the CheckBox reference section. You can use the font and fsize keywords to adjust the checkbox label.

The user-defined action procedure that you will need to write for CheckBoxes must have the following form:

```
Function CheckProc(ctrlName,checked) : CheckBoxControl
   String ctrlName
   Variable checked

End
```

The checked parameter will be set to the new checkbox value (0 or 1). Checkboxes do not usually need an action procedure since one can read the state of the checkbox with the **ControlInfo** operation (page V-63).

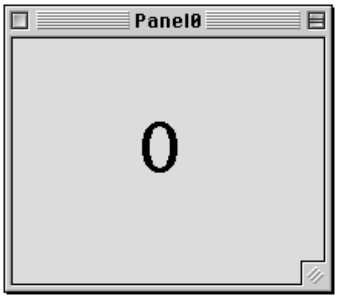
You can create custom checkboxes by following steps similar to those for custom Buttons (see **Custom Button Example** on page III-368), except the picture has six states side-by-side instead of three. The checkbox states are:

Image Order	Control State
Left	Deselected enabled.
	Deselected enabled and clicked down (about to be selected).
	Deselected disabled.
	Selected enabled.
	Selected enabled and clicked down (about to be deselected).
Right	Selected disabled.

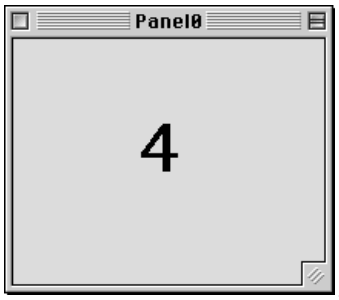
CustomControl

The CustomControl operation creates or modifies a custom control, all aspects of which are completely defined by the programmer. See the **CustomControl** operation on page V-93 for a complete description.

What you can create with a CustomControl can be fairly simple such as this counter that increments when you click on it.



Four clicks later:



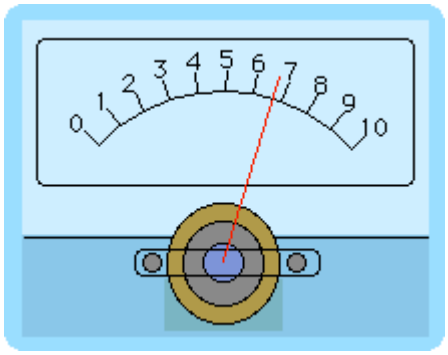
.

The following code implements the counter custom control using the `kCCE_frame` event. In the panel, click on the number to increment the counter; also try clicking and then dragging outside the control.

```
static constant kCCE_mouseup= 2
static constant kCCE_frame= 12

// PNG: width= 280, height= 49
Picture Numbers0to9
   ASCII85Begin
   M,6r;%14!\!!!!.8Ou6I!!!$:!!!!R#Qau+!00#^OT5@]&TgHDFAm*iFE_/6AH5;7DfQssEc39jTBQ
   =U"5QO:5u`*!m@2jnj"La,mA^'a?hQ[Z.[.,Kgd(1o5*(PSO8oS[GX%3u'11dTl)fII/"f-?Jq*no#
   Qb>Y+UBKXKHQpQ&qYW88I,Ctm(`:C^]$4<ePf>Y(L\U!R2N7CEAn![N1I+[hTtr.VepqSG4R-;/+$3
   IJE.V(>s0B@E@"n"ET+@5J9n_E:qeR_8:Fl?m1=DM;mu.AEj!)]K4CUuCa4T=W)#(SE>uH[A4\;IG/
   e]FqJ4u,2`*p=N5sc@qLD5bH89>gIBdF-1i6SF28oH@"3c2m)bDr&,UB$]i]/0bA.=qbR2#\-D9E?O
   2>3D>`($p(Kn)F8aF@)LYiXn[h2K):5@^kF?94)j*1Xtq1U2oFZmY.te?0G)EQ%5,RVT-c)DVa+%mP
   %+bS*_hN$hC*8uCJuIWqTHJR.U?32`_B)(g_8e#*YXa>=faEdJsF]6iJlrQ@QAX7huJUmXj8:PBTb2
   Y:DYf*Sci'Q"3_;@RDQA:A/([2sO8r$hW)\B$XBGASJ:6OpC+GL<FjVfeNm20U<l<9J%cndX3'HP+k
   R.IV?U>ns*_;Zt[]6G6"Rb-*'Nm-E8]LXXXo7Ub>A**7Bm5cS*">HbQ&_RhmUe]$iu@T?Cci:e-_`k
   sE+H.GRSMT(9to;IZuH`T4%Yt<jF$+W?Yh6Q*_`C4sGig=L@DKoT%.H=#e_H"QEeeBVNTWBSMYr3dj
   O=T%d&4kT9#cWPHS>kAG;3=or2(IK*IBF$^qK,+m0NSDK_!+e0#3fAI>HfKa<sk0641u\W@r+Y:$.i
```

```
    i$grCPR#&6,;+>nTs_IKS6XcYR)A$fJiC6Z_d2S!$R>_ZH+[<p:JI0ub]\BhE(0RP@((KTRTGo;#SY
    LT^9;D7X#km%UV20?$RS"FZoIF!(`FY-iL?n$%#o;-Wj(\PaBS6ZRQe@:kC>%ULrhTWLNM=n@fUbRp
    SKkLe\kJ)Sd]u7!?pRJk-!XL[/MZX'"n4?a?JIKO0k'KUm1IZ+roB=:Bq'$&E<#$Krp%p,E"4sI>[-
    0F#^ff5SN':2fO)LNC?L4(2ga=!aLm8)tVbGAM?L`l^=$D_YP7Z(sOFs)BL5er5G95p3?m%hM^lSr'
    *E^O@8=u6hL`L$mPcq!Bl-iHuGA6hiip%`cFjl9>W?'E-&5T%Y.]i2A@1i%p8XJ5[khb:&"JXYSC\r
    10Ss8<Ye;S^"Nc0%-DFouAiPQ9OemnR!"sHH$JKt@!"d0E"'M(P%:`p'15_10`!<nVt"TALQ>PF8WL
    Z:#f!!!!j78?7R6=>B
    ASCII85End
End

Structure CC_CounterInfo
    Int32 theCount      // current frame of 10 frame sequence of numbers in
EndStructure

Function MyCC_CounterFunc(s)
    STRUCT WMCustomControlAction &s

    STRUCT CC_CounterInfo info

    if( s.eventCode==kCCE_frame )
        StructGet/S info,s.userdata
        s.curFrame= mod(info.theCount+(s.curFrame!=0),10)
    elseif( s.eventCode==kCCE_mouseup )
        StructGet/S info,s.userdata
        info.theCount= mod(info.theCount+1,10)
        StructPut/S info,s.userdata   // will be written out to control
    endif

    return 0
End

Window Panel0() : Panel
    PauseUpdate; Silent 1          // building window...
    NewPanel /W=(69,93,271,252)
    CustomControl cc2,pos={82,46},proc=MyCC_CounterFunc,picture=
{ProcGlobal#Numbers0to9,10}
EndMacro
```

You can create even more sophisticated controls, such as this voltage meter control.



The following code creates the voltage meter control. This example illustrates both draw and drawOSBM custom drawing along with several other events. Move the mouse over the surface of the meter to see how it responds.

```
static constant kCCE_mousemoved= 4
static constant kCCE_draw= 10
static constant kCCE_drawOSBM= 17

// PNG: width= 223, height= 180
Picture PanelMeterNoScale
    ASCII85Begin
    M,6r;%14!\!!!!.8Ou6I!!!#V!!!#+#Qau+!1Tp)iW&rY&TgHDFAm*iFE_/6AH5;7DfQssEc39jTBQ
    =U#(]?65u`*!mG0F:)b1m'nPpnh+J:S?W-P8e=<.L=]Rg..(`<SC?kdiP`^-.-n1-1"nC6)E%8T9@)
    rPq5JlUFnl#(^QpIATX4WfnXZ@%P$r:]i[\<HT3D#d/2%Xe=B,O0rbf1e2OqDRDC@(\\;D7jSgHeq>
    \(g_'S0C#?+**/o`kOVHs`5,FtA6/n>RlVFW&@%cMZP0oV)$VJ,\0)`oG?Vc]h#!dQbl7OlG:P1*FA
    XE-;($H_-Zi35FAXE-;($H_-ZfNp`qu=!9Y*7pjQc.fr?&0genMa4o8PI;D,Nr4fgui3g2rKUkP<5Q
    1_K3EELh7caK>f[W"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Z
    b@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'
```

```
    =t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+
    U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=q*,]R*nUT1h:1']eQWA9.FS:AD]ufh%imdAE,ZBjYKP.M+"[d\
    `n,=D.ML$j<gcJhH>A0'\*Bl@^eTKJ)PC0'\+m<nBE'0)0#]f7R;%cFpb:=3G==4ioMiFiK<"dW,",
    dkoPlB%r3(E`XPc)N@.U!Ttho2I0A9LJ%P6g"9,6U\5q=a_EhILrSA1;SE(X[+l5[l1+A\m&\V%C>O
    VD5)@lp2,'I!>NN,Jg%iM!C!Pl89fMH^iJ@kEM`RKLVs0p,MPBo1V>Wj,@hNH,lJ.1hdrsu4dW,#dR
    0Fb'5^-)'0l_R'pb")`QUE`(T\J*6D>:_7:=K$RdDke-i%^0%ZST7kcfXt=\oc^=omAjPpMJ."J?I=
    BMPBT9:cO[3HN!)4j3:C"r&ruS*5M6dWFqB:Su,GP^'QiihgbFl]oVW_k05Qof)GZh\&Q@fm+3&Oh&
    aqJkV#Z:R@q8Z8+O^7prdjk'<X.O^Vl_[3R$JES);g=^hPkMchR.j]X(IbK6*F#cN&=c)LJUNAQeI#
    B&o9`fCtCHDk6n+:^l+?e`*:bVT"s;2]YuCl0RDOjGQ)MHWB,EB3CO%Mhg"CNA:.(g9I+E*ChNe.nV
    &>lM;OK`,0iikucRjj'S1NPs'B?Br>S@JULjp"Wsu58CBQRpQ%XA5'<<bf40/fmbX1Od^a:b`gEgG6
    Q$6<<n"%GkBXVqb[V+(UsA:mYG-G^[Y$!g`_!@8i1%[RguG#n%J8;nB=>FH8n=JLQS:D$p`37/+4!S
    \L36g0SCYDd78Y&tHC*sho#S%1A:lq:M`RO^_`C#0q-LZ:.Eh:\pTV1l096P-Zmm_>-X5`--pB<0'#
    $&E5sATscV%:cf/kLLpZsQUcrKD,'E>^#\CbN$WRIl[,RH-MC,qj09@'+YMP=@G%J*>,PMl&TjoQ3g
    !m/e$:!Kqj$H*k@;]o,bqaQ^GS"5$9,TGG)MEIJZI!tk]eT6M5XbPXT2/SNC9p'.9D@IY%hK#*VmFc
    u/(3<7j?We3/?9<9.j[PI(ep2O^4+%"1I2P1r%V%E3^V%,Ej,oDD(g$F(eT*^pFi8]PlXN;QNVX,Am
    gJBN(j2C92n)MZR)=k>cd-oF=Z7S/e$%BrSM`RMf^&0g:X`O@_&r3di3^L1B2IcB1,\*OpPP+I]X25
    ,HoJ/\*ne\g5KT5-kqG1X@X_p4Rir]<B(FtpM:lU%0pai.J%U9o^Q/gS@>MpF\U>anO.8Augk^@DBY
    ..%c.!3Vq#:R3Bi\'nZhX8Qh0iP*T)6GKc!&Wm_S0s&GI%)mbpMe7Wr,p`4%G_\bg%,MJsr:gca80,
    8CN4E=jKRdh[h*Y(sL8pRm=U\$togFGpMY`k]W5(AF>.4_qYg2s-\tJ4%#[a]hi>m(KPp\7+i2qmVO
    o3i'!3W3n"[JhmfB`?.90d@pI^Er:Ja;%foOpS^9frco"t0f>\ZR%nQm.Aq)q;@<OujVG:2CE*1jD1
    -;2q\PC-l%^$H'All@!*>4*.3ui&$ZtL/kn1<:QFTMfgXpDV;'=t+U])Dg%&&%AlD@.Hc!!!!j78?7
    R6=>B
    ASCII85End
End

Function drawneedle(v)
    Variable v                    // volts

    if( v<0 )
        v= 0
    elseif( v>10 )
        v= 10
    endif

    // Note: constants are specific to panel meter image
    Variable theta= 2.39 - 1.67*v/10
    Variable x0= 110, y0= 131,len= 96
    Variable x= x0 + len*cos(theta)
    Variable y= y0 - len*sin(theta)

    SetDrawEnv linefgc= (65535,0,0)
    DrawLine x0,y0,x,y
End

Function drawscale(vmin,vmax,n)
    Variable vmin,vmax,n

    variable i
    Variable theta0= 2.39  // Note: constants are specific to panel meter image
    Variable dtheta= -1.67/n
    Variable x00= 110, y00= 131,len= 85,ticklen=10,labellen=15
    String s

    SetDrawEnv textxjust= 1,textyjust= 1,save
    for(i=0;i<=n;i+=1)
        Variable theta= theta0 + i*dtheta
        Variable x0= x00 + len*cos(theta)
        Variable y0= y00 - len*sin(theta)
        sprintf s,"%.2g",vmin+i*(vmax-vmin)/n
        DrawLine x0,y0,x00 + (len+ticklen)*cos(theta),y00- (en+ticklen)*sin(theta)
        DrawText x00 + (len+labellen)*cos(theta),y00 - (len+labellen)*sin(theta),s
        if( i!=n )
            DrawLine x0,y0,x00 + len*cos(theta+dtheta),y00 - len*sin(theta+dtheta)
        endif
    endfor
End

Structure CC_MeterInfo
    double voltage                    // voltage value (0-10)
    STRUCT Point lastMouse
EndStructure

Function MyCC_MeterFunc(s)
    STRUCT WMCustomControlAction &s

    STRUCT CC_MeterInfo info
```

```
    if( s.eventCode==kCCE_drawOSBM )
        drawscale(0,10,10)
    elseif( s.eventCode==kCCE_draw )
        StructGet/S info,s.userdata
        drawneedle(info.voltage)
    elseif( s.eventCode==kCCE_mousemoved )
        StructGet/S info,s.userdata
    // Beware: Next command is wrapped to fit on the page
        variable dist= sqrt((s.mouseLoc.h-info.lastMouse.h)^2+(s.mouseLoc.v-
info.lastMouse.v)^2)
        info.voltage= info.voltage + (dist-info.voltage)/10
        info.lastMouse= s.mouseLoc
        StructPut/S info,s.userdata    // will be written out to control
        s.needAction= 1                // want redraw
    endif

    return 0
End

Window Panel0() : Panel
    PauseUpdate; Silent 1          // building window...
    NewPanel /W=(150,50,450,250)
    CustomControl cc3,pos={10,10},proc=MyCC_MeterFunc
    // This should be after the setting of the proc
    CustomControl cc3,picture= {ProcGlobal#PanelMeterNoScale,1}
EndMacro
```

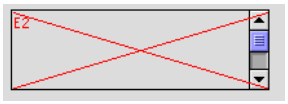
GroupBox

The GroupBox operation creates or modifies a listbox control. See the **GroupBox** operation on page V-238 for a complete description and examples.

ListBox

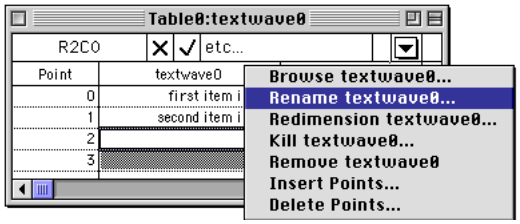
The ListBox operation creates or modifies a listbox control. See the **ListBox** operation on page V-336 for a complete description and examples.

The simplest listbox needs at least one text wave to contain the list items. Without the text wave, a listbox control has no list items.



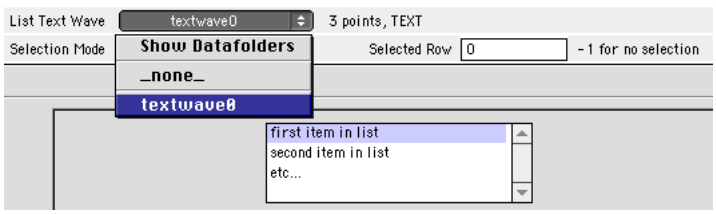
```
ListBox list0 size={200,60},mode=1
```

Create the text wave by opening a table and start typing a nonnumeric first list item (this is to make certain the wave is created as a text wave). You can rename the text wave by Control-clicking (*Macintosh*) or right-clicking (*Windows*) the name and choosing "Rename textwave0".

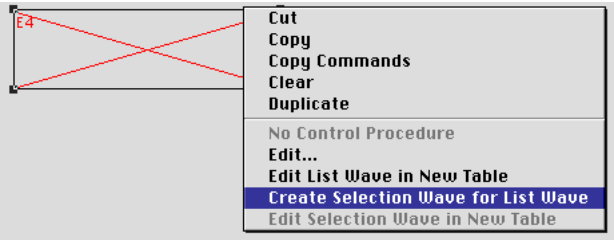


Note: If you want to create a text wave in a particular data folder, set the current data folder first (using the Data Browser in the Data menu).

Then select the wave you created in the ListBox Control dialog as the text wave for the list:



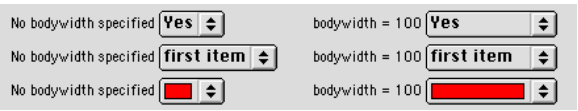
Right-clicking (*Windows*) or Control-clicking (*Macintosh*) a listbox shows a contextual menu for editing the list waves or action procedure, and to create a numeric selection wave (if the ListBox is a multiselection listbox):



PopupMenu

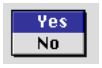
The **PopupMenu** operation (page V-490) creates or modifies a pop-up menu control. Pop-up menus provide a choice of colors, line styles, patterns, markers, or text items:

The control automatically sizes itself as a function of the title or the currently selected menu item. You can specify the bodyWidth keyword to force the body (nontitle portion) of the pop-up menu to be a fixed size. You might do this to get a set of pop-up menus of nicely aligned with equal width. The bodywidth keyword also affects the nontext pop-up menus.



The font and fsize keywords affect only the title of a pop-up menu — the pop-up menu itself uses standard system fonts.

Unlike color, line style, pattern, or marker pop-up menus, text pop-up menu controls can operate in two distinct modes as set by the mode keyword's value.



If the argument to the mode keyword is nonzero then it is considered to be the number of the menu item to be the initial current item *and* displays the current item in the pop-up menu box. This is the *selector mode*. There is often no need for an action procedure since the value of the current item can be read at any time using the **ControlInfo** operation (page V-63).

If mode is zero then the title appears inside the pop-up menu box (hence the name *title-in-box mode*). This mode is generally used to select a command for the action procedure to execute. The current item has no meaning except when the pop-up menu is activated and the selected item (command) is passed to the action procedure.

The menu that pops up when the control is clicked is determined by a string expression that you pass as the argument to the value keyword.

Create the color, line style, pattern or marker pop-up menus by setting the string expression to one of these fixed values: `"*COLORPOP*"`, `"*LINESTYLEPOP*"`, `"*MARKERPOP*"`, or `"*PATTERNPOP*"`.

For text pop-up menus, the string expression must evaluate to a list of items separated by semicolons. For example:

```
PopupMenu name value= "Item 1; Item 2; Item 3"
PopupMenu name value= "_none_;" + WaveList("*",";","")
```

It is possible to apply certain special effects to the menu items, such as disabling an item or marking an item with a check. See **Special Characters in Menu Item Strings** on page IV-114 for details.

It is important to note that the literal text of the string expression is stored with the control rather than the results of the evaluation of the expression. The expression is reevaluated every time the user clicks on the pop-up menu box.

The string expression evaluates as if it were typed on the Command line within the current data folder. Additionally, it is important to specify the data folder containing any string variables, numeric variables, or waves used in the string expression:

```
PopupMenu name value=#"func(root:folder:wave0, root:gVar)"
```

For this reason, the expression can not use numeric or string variables that are local to a procedure since such variables cease to exist when the procedure finishes execution.

To incorporate the value of local variables in the value expression use the **Execute** operation:

```
String str= "\"_none_;first;second;\"" // str contains quotes
Execute "PopupMenu name value=" + str
```

The reason that the evaluation of the menu expression takes place when the user clicks on the menu is to ensure that dynamic menus such as the above WaveList example will reflect the *current* conditions rather than the conditions that were in effect when the PopupMenu control was *created*.

Because of this, the pop-up menu does not automatically update if the value of the string expression changes. You can use the **ControlUpdate** operation (page V-67) to force the pop-up menu to update. Here is an example:

```
NewPanel/N=PanelX
String/G gPopupList="First;Second;Third"
PopupMenu oneOfThree value=gPopupList  // pop-up shows “First”
gPopupList="1;2;3"                     // pop-up is unchanged
ControlUpdate/W=PanelX oneOfThree      // pop-up shows “1”
```

If the value expression can not be evaluated at the time the command is compiled, you can defer the evaluation of the expression by enclosing the value this way:

```
PopupMenu name value= #"pathToNonExistentGlobalString"
```

(the `value=gPopupList` example requires that `gPopupList` exist during compilation.)

If a deferred expression has quotes in it, they need to be “escaped” with backslashes (for a description of this syntax, see **When Dependencies are Updated** on page IV-206):

```
PopupMenu name value= #"\"_none_;\"+UserFunc(\"foo\")"
```

The optional user defined action procedure is called after the user makes a selection from the popup menu. Popup menu procedures have the following form:

```
Function PopMenuProc(ctrlName,popNum,popStr) : PopupMenuControl
   String ctrlName
   Variable popNum
   String popStr

End
```

`popNum` will be the item number, *starting from one*, and `popStr` will be the text of the selected item. For the color pop-up menus the easiest way to determine the selected color is to use the **ControlInfo** operation (page V-63).

Another form of the action procedure uses structures. See **Using Structures with Windows and Controls** on page IV-82.

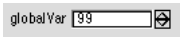
SetVariable

The **SetVariable** operation (page V-565) creates or modifies a SetVariable control. These controls are tied to numeric or string global variables (or even to one value in a numeric or text wave) and can be used to both view and set these values. When used with numeric variables, up/down arrows that the user can use to increment or decrement the variable will be drawn unless you set the increment value to zero (see the limits keyword).

You can set the width of the control but the height is determined from the font and font size. The width of the readout area is the width of the control less the width of the title and up/down arrows. However, you can use the bodyWidth keyword to specify a fixed width for the body (nontitle) portion of the control.

For example, executing the commands:

```
Variable/G globalVar=99
SetVariable setvar0 size={120,20},frame=1,font="Helvetica", value=globalVar
```

Results in the following SetVariable control: 

To associate a SetVariable control with a variable that is not in the current data folder at the time SetVariable runs, you must use a data folder path:

```
Variable/G root:Packages:ImagePack:globalVar=99
SetVariable setvar0 value=root:Packages:ImagePack:globalVar
```

Unlike PopupMenu controls, SetVariable controls remember the current data folder when the SetVariable command executes. Thus an equivalent set of commands is:

```
SetDataFolder root:Packages:ImagePack
Variable/G globalVar=99
SetVariable setvar0 value=globalVar
```

Also see **Set Variable Controls and Data Folders** on page III-362.

You can control the style of the numeric readout via the format keyword. For example, the string `"%.2d"` will display the value with 2 digits past the decimal point. You should not use the format string to include text in the readout because Igor has to read back the numeric value. You may be able to add suffixes to the readout but prefixes will not work. When used with string variables the format string is not used.

The user defined action procedure that you may need to write for SetVariables must have the following form:

```
Function SetVarProc(ctrlName,varNum,varStr,varName) : SetVariableControl
   String ctrlName
   Variable varNum
   String varStr
   String varName

End
```

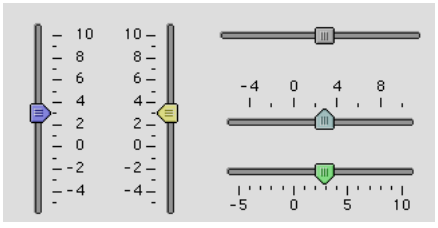
`varName` will be the name of the variable being used. If the variable is a string variable then `varStr` will contain its contents and `varNum` will be set to the results of an attempt to convert the string to a number. If the variable is numeric then `varNum` will contain its contents and `varStr` will be set to the results of a number to string conversion.

Note that when the user presses and holds in the up or down arrows then the value of the variable will be steadily changed by the increment value but your action procedure will not be called until the user releases the mouse button.

Another form of the action procedure uses structures. See **Using Structures with Windows and Controls** on page IV-82.

Slider

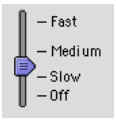
The **Slider** operation (page V-574) creates or modifies a slider control. The control can be used to set the value of a global variable or be used alone to indicate a value to be retrieved with the ControlInfo command. The value is changed by dragging the "thumb" part of the control. The slider can be drawn vertically or horizontally.



There are many options for labelling the numeric range such as setting the number of ticks.

You can also provide custom labels in two waves (one numeric and another providing the corresponding text label):

```
Make/O tickNumbers= {0,25,60,100}
Make/O/T tickLabels= {"Off","Slow","Medium","Fast"}
Slider speed,pos={86,28},size={74,73}
Slider speed,limits={0,100,0},value= 40
Slider speed,userTicks={tickNumbers,tickLabels}
```

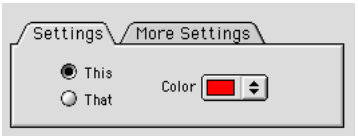


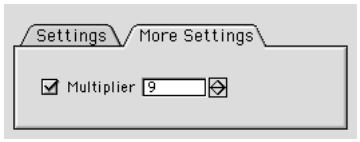
The action procedure for the slider can be used to apply the slider value to a process. The action procedure is called not only when the user moves the thumb, but also when the mouse button clicks down and up on the thumb, and when a procedure modifies the slider's controlled or controlling variable.

See the **Slider** operation on page V-574 for a complete description and more examples.

TabControl

The **TabControl** operation (page V-683) creates or modifies a TabControl control. Tabs are used to group controls into visible and hidden groups.





The tabs are numbered: the first tab is tab 0 and the second is tab 1, etc.

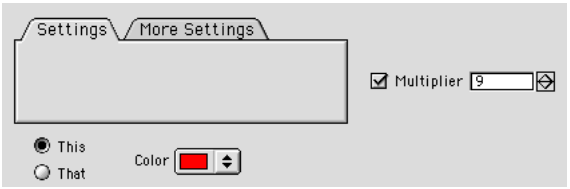
You add tabs to the control by providing additional tab titles:

```
TabControl tb, tabLabel(0)="Settings",tabLabel(1)="More Settings"
```

When you click on a tab, the control's action procedure receives the number of the clicked-on tab.

The showing and hiding of the controls are accomplished by user-written code in the tab control's action procedure. In this example, the "This", "That", and "Color" controls are shown when the "Settings" tab is clicked, and the "Multiplier" checkbox and SetVariable controls are hidden. When the "More Settings" tab is clicked, the action procedure makes opposite occur.

The simplest way to create a tabbed user interface is to create an over-sized panel with all the controls visible and not inside the tab control. Place controls in their approximate positions relative to one another:



By positioning the controls this way you can more easily modify each control until you are satisfied with them. Before you put them into the tab control, get a list of the nontab control names:

```
•Print ControlNameList("" ,"\r", "!tb")      // all but "tb"
thisCheck
thatCheck
colorPop
multCheck
multVar
```

and figure out which tab you want them to be visible in:

Tab 0: Settings	Tab 1: More Settings
thisCheck	multCheck
thatCheck	multVar
colorPop	

Write the action procedure for the tab control to show and hide the controls:

```
Function TabProc(ctrlName,tabNum) : TabControl
   String ctrlName
   Variable tabNum

   Variable isTab0= tabNum==0
   Variable isTab1= tabNum==1

   // note: disable=0 means "show", disable=1 means "hide"
   ModifyControl thisCheck disable= !isTab0  // hide if not Tab 0
   ModifyControl thatCheck disable= !isTab0  // hide if not Tab 0
   ModifyControl colorPop disable= !isTab0   // hide if not Tab 0

   ModifyControl multCheck disable= !isTab1  // hide if not Tab 1
   ModifyControl multVar disable= !isTab1    // hide if not Tab 1
   return 0
End
```

(A more elegant method, which will be important when you have many controls, is to systematically give the controls inside each tab a prefix or suffix that is unique to that tab, such as `tab0_thisCheck`, `tab0_thatCheck`, `tab1_multVar`, and use the ModifyControlList operation to show and hide the controls. See the **ModifyControlList** operation (page V-396) for an example.)

Then assign the action procedure to the tab control with the Tab Control dialog or a command like this:

```
TabControl tb proc=TabProc
```

Click on the tabs to see whether the showing and hiding is working correctly.

When it works correctly, click on one tab and then move the controls that belong inside into the tab area. Click on the next tab and them move that tab's controls into the tab area. You will need to use the operate mode to change the tab by clicking (and thus running the action procedure) and use the modify mode to move the controls. The "temporary selection" shortcut of pressing Command-Option (*Macintosh*) or Ctrl+Alt (*Windows*) is really handy here.

Save a recreation macro for the panel (Windows→Control→Window Control) to record the final control positions in a panel macro. Rewrite the macro as a function that initially creates the panel:

```
Function CreatePanel()
   DoWindow/K TabPanel                     // start over
   NewPanel/N=TabPanel/W=(596,59,874,175) as "Tab Demo Panel"
   TabControl tb,pos={15,19},size={250,80},proc=TabProc
   TabControl tb,tabLabel(0)="Settings"
   TabControl tb,tabLabel(1)="More Settings",value= 0
   CheckBox thisCheck,pos={53,52},size={39,14},title="This"
   CheckBox thisCheck,value= 1,mode=1
```

```
   CheckBox thatCheck,pos={53,72},size={39,14},title="That"
   CheckBox thatCheck,value= 0,mode=1
   PopupMenu colorPop,pos={126,60},size={82,20},title="Color"
   PopupMenu colorPop,mode=1,popColor= (65535,0,0)
   PopupMenu colorPop,value= #"\"*COLORPOP*\""
   CheckBox multCheck,pos={50,60},size={16,14},disable=1
   CheckBox multCheck,title="",value= 1
   SetVariable multVar,pos={69,60},size={120,15},disable=1
   SetVariable multVar,title="Multiplier",value=multiplier
End
```

See the **TabControl** operation on page V-683 for a complete description and examples.

TitleBox

The TitleBox operation creates or modifies a TitleBox control. The control's text can be unchanging, or can be the contents of a global string variable. See the **TitleBox** operation on page V-705 for a complete description and examples.

ValDisplay

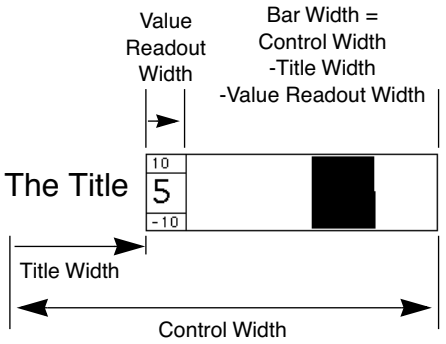
The **ValDisplay** operation (page V-715) creates or modifies a value display control. This is a very flexible and multifaceted control and can range from a simple numeric readout to a thermometer bar or a hybrid of both. A ValDisplay control is "connected" to a numeric expression that you provide as an argument to the value keyword. The display will be automatically updated whenever anything that the numeric expression depends on is changed.

ValDisplay controls evaluate their value expression in the context of the root data folder (see Chapter II-8, **Data Folders**, and **Programming with Data Folders** on page IV-148). To reference a data object that is not in the root, you must use a data folder path, such as "root:Folder1:var1".

Here are a few selected keywords extracted from the **ValDisplay** operation on page V-715:

```
size={width,height}
barmisc={lts, valwidth}
limits={low,high,base}
```

The appearance of the ValDisplay control depends primarily on the *valwidth* and size parameters and the width of the title. However, you can use the bodyWidth keyword to specify a fixed width for the body (non-title) portion of the control. Essentially, space for each element is allocated from left to right, with the title receiving first priority. If the control width hasn't all been used by the title, then the value readout width is the smaller of *valwidth* pixels of room or what is left. If the control width hasn't been used up, the bar is displayed in the remaining control width:



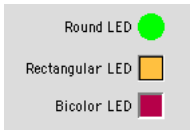
Here are the various major possible forms of ValDisplay controls. Some of these examples modify previous examples; check the names of the ValDisplay controls. For instance, the second bar-only example is a modification of the valdisp1 control created by the first bar-only example.

Numeric Readout Only

```
// default readout width (1000) is >= default control width (50)
ValDisplay valdisp0 value=K0
```

LED Display

```
// Create the three LED types.
ValDisplay led1,pos={67,17},size={75,20},title="Round LED"
ValDisplay led1,limits={-50,100,0},barmisc={0,0},mode=1
ValDisplay led1,bodyWidth= 20,value= #"K1",zeroColor=(0,65535,0)

ValDisplay led2,pos={38,48},size={104,20},title="Rectangular LED"
ValDisplay led2,frame=5,limits={0,100,0},barmisc={0,0},mode=2
ValDisplay led2,bodyWidth= 20,value= #"K2"
ValDisplay led2,zeroColor= (65535,49157,16385)

ValDisplay led3,pos={60,76},size={82,20},title="Bicolor LED"
ValDisplay led3,limits={-40,100,-100},barmisc={0,0},mode= 2
ValDisplay led3,bodyWidth= 20,value= #"K3"
```

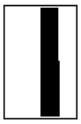


Bar Only

```
// readout width = 0
ValDisplay valdisp1,frame=1,barmisc={12,0},limits={-10,10,0},value=K0
K0= 5                // halfway from base of 0 to high limit of 10.
```

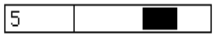
The nice thing about a bar-only ValDisplay is that you can make it 5 to 200 pixels tall whereas with a numeric readout, the height is set by the font sizes of the readout and printed limits.

```
// Set control height= 80
ValDisplay valdisp1, size={50,80}
```



Numeric Readout and Bar

```
// 0 < readout width (50) <  control width (150)
ValDisplay valdisp2 size={150,20},frame=1,limits={-10,10,0}
ValDisplay valdisp2 barmisc={0,50},value=K0  // no limits shown
```



Optional Limits

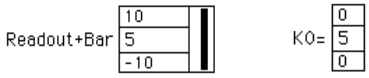
Whenever the numeric readout is visible, the optional limit values may be displayed too.

```
// Set limits font size to 10 points. Readout widths unchanged.
ValDisplay valdisp2 barmisc={10,50}
ValDisplay valdisp0 barmisc={10,1000}
```

Optional Title

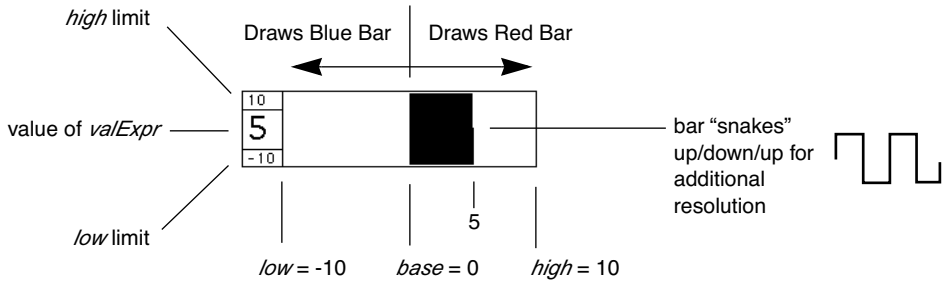
The control title steals horizontal space from the numeric readout and the bar, pushing them to the right. You may need to increase the control width to prevent them from disappearing.

```
// Add titles. Readout widths, control widths unchanged.
ValDisplay valdisp2 title="Readout+Bar"
ValDisplay valdisp0 title="K0="
```



The limits values *low*, *high*, and *base* and the value of *valExpr* control how the bar, if any, is drawn. The bar is drawn from a starting position corresponding to the *base* value to an ending position determined by the value of *valExpr*, *low* and *high*. *low* corresponds to the left side of the bar, and *high* corresponds to the right. The position that corresponds to the *base* value is linearly interpolated between *low* and *high*.

For example, with *low* = -10, *high*=10, and *base*= 0, a *valExpr* value of 5 will draw from the center of the bar area (0 is centered between -10 and 10) to the right, halfway from the center to the right of the bar area (5 is halfway from 0 to 10):



You can force the control to not draw bars with fractional parts by specifying mode=3.

Killing a Control

You can kill (delete) a control from within a procedure using the **KillControl** operation (page V-321). This might be useful in creating control panels that change their appearance depending on other settings.

You can interactively kill a control by selecting it with the arrow tool or the Select Control submenu and press Delete. Moving a control out of the window does not delete it; you just end up with a control that is offscreen.

Getting Information About a Control

You can use the **ControlInfo** operation (page V-63) to obtain information about a given control. This is useful to obtain the current state of a checkbox or the current setting of a pop-up menu.

Updating a Control

You can use the **ControlUpdate** operation (page V-67) to cause a given control to redraw with its current value. You can use this in a user-defined function or macro after changing the value or appearance of a control and to display the changes before the normal graph or panel update occurs.

Help Text for User-Defined Controls

You can easily add help Igor Tips (*Macintosh*) or context-sensitive/status line help text (*Windows*) for your controls. Each dialog has a button titled Igor Tips (*Macintosh*) or Edit Help (*Windows*) that leads to a subdi-

alog where you can edit the help text. You can also use the command line. You are limited to using a total of 255 characters for your help message. Here is an example:

```
Button button0 title="Beep", help={"This button beeps."}
```

This creates a button with a Macintosh Igor Tip, or under Windows, this text appears as context-sensitive help and in the status line when the mouse passes over the control. The help text that appears on the status line is limited to the first 127 characters of text or the text up to the first line break. The `help={"This button beeps."}` section sets the help text for the button. You can use the help keyword with the Checkbox, PopupMenu, ValDisplay, and SetVariable operations as well as the Button operation.

There is no way to specify help for the individual items in a user-defined pop-up menu.

Modifying a Control

The control operations create a new control if the name parameter doesn't match a control already in the window. The operations modify an existing control if the name does match a control in the window, but generate an error if the control kind doesn't match the operation.

For example, if a panel already has Button control named `button0`, you can modify, say, the disable state for that button with another `Button button0` command:

```
Button button0 disable=1              // hide
```

However, if you use a Checkbox instead of Button, you will get a "button0 is not a Checkbox" error.

You can use the **ModifyControl** operation (page V-395) and **ModifyControlList** operation (page V-396) to modify a control without needing to know what kind of control it is:

```
ModifyControl button0 disable=1      // hide
```

This is really handy when used in conjunction with tab controls.

Disabling and Hiding a Control

All controls support the keyword "disable=*d*" where *d* can be 0 (normal operation), 1 (hidden), or 2 (user input disabled). Charts and ValDisplays do not change appearance when disable=2 because they are read-only.

SetVariables also have the noedit keyword. This appears to be redundant with the disable=2 mode but there is one quasi-important difference: noedit still allows user input via the up or down arrows but disable=2 does not.

Background Color

The background color of Panel windows (see **Control Panels** on page III-389) and the area at the top of a graph as reserved by the **ControlBar** operation (page V-63) is a shade of gray chosen to match the standard Macintosh or Windows system look. This gray is used when the background color is the default pure white where the red, green and blue components are all 65535. Any other color, including not quite pure white, will be honored.

However some controls or portions of controls are drawn by the system and may look out of place if you choose a different background color. The cbRGB keyword is used to set the control background in Panels (**ModifyPanel** operation on page V-419) and Graphs (**ModifyGraph (colors)** operation on page V-415).

Most controls are drawn with a background color that matches the window background color giving the impression of transparency. A few controls are drawn with true transparent labels and no background color is needed.

For special purposes, you can specify a background color for an individual control using the labelBack keyword (see the reference descriptions of the individual controls for further details).

On the Macintosh prior to Igor Pro 4, the background color of controls was white. This matched the window color which was also white. The new technique is somewhat nonbackwards compatible and a few people may prefer the old white on white look. In many cases simply setting the window background to slightly off-white:

```
ModifyGraph cbRGB=(65534,65534,65534)
```

will do the job. In other cases, you may need to specify the background color of individual controls using the labelBack keyword.

On Windows, the background color of the controls themselves is dependent on the current Appearances Settings. Specifically, the background color of controls is set by the 3D Objects color in the Appearance Tab of the Display Properties control panel. You should be aware that any color you choose now may not result in pleasing esthetics when these settings are changed in the future.

On Windows if you do not explicitly set the background color, the standard Windows 3D Objects color is used. This works well with buttons and other standard Windows controls, even when the user changes the 3D Objects color later. You can change the background color to track the 3D Objects color by choosing the current 3D Objects color from the color pop-up palette, or by executing a command to set the background to maximum Macintosh white:

```
ModifyGraph cbRGB=(65535,65535,65535)
ModifyPanel cbRGB=(65535,65535,65535)
```

Control Structures

The action procedure for a control can also use a predefined, built-in structure as a parameter to the function. The control will use this more efficient method whenever the function properly matches the structure prototype for a control, otherwise it will use the "old-style" method.

An action procedure using a structure has the format:

```
Function newActionProcName(CB_Struct)
   STRUCT WMCheckboxAction &CB_Struct
   …
End
```

The names of the various control structures are listed in the next table. You should consult the reference information for each individual control to see what members each control structure contains.

Control Type	Structure Name
Button	**WMButtonAction**
CheckBox	**WMCheckboxAction**
CustomControl	**WMCustomControlAction**
ListBox	**WMListboxAction**
PopupMenu	**WMPopupAction**
SetVariable	**WMSetVariableAction**
Slider	**WMSliderAction**
TabControl	**WMTabControlAction**

Action functions should respond only to documented `eventCode` values. Other event codes may be added along with more fields. Although the return value is not currently used, action functions should always return zero.

The constants used to specify the size of structure `char` arrays are internal to Igor Pro and may change.

Control Structure Example

This example illustrates the extended event codes available with a Button control (as well as a application of user data). The function prints various text messages to the History area depending what actions you take while in the button area.

```
Function structureTest()
   NewPanel
   Button b0,proc= NewButtonProc
End

Structure MyButtonInfo
   Int32 mousedown
   Int32 isLeft
EndStructure

Function NewButtonProc(s)
   STRUCT WMButtonAction &s

   STRUCT MyButtonInfo bi
   Variable biChanged= 0

   StructGet/S bi,s.userdata
   if( s.eventCode==1 )
      bi.mousedown= 1
      bi.isLeft= s.mouseLoc.h < (s.ctrlRect.left+s.ctrlRect.right)/2
      biChanged= 1
   elseif( s.eventCode==2 || s.eventCode==3 )
      bi.mousedown= 0
      biChanged= 1
   elseif( s.eventCode==5 )
      print "Enter button"
   elseif( s.eventCode==6 )
      print "Leave button"
   endif

   if( s.eventCode==4 )              //  mousemoved
      if( bi.mousedown )
         if( bi.isLeft )
            printf "L"
         else
            printf "R"
         endif
      else
         printf "*"
      endif
   endif
   if( biChanged )
      StructPut/S bi,s.userdata     // written out to control
   endif

   return 0
End
```

Control Structure eventMod Field

The eventMod field appears in the built-in structure for each type of control. It is a bitfield defined as follows:

EventMod Bit	Meaning
Bit 0	Left mouse button is down.
Bit 1	Shift key is down.
Bit 2	Option (Macintosh) or Alt (Windows) is down.

EventMod Bit	Meaning
Bit 3	Command (Macintosh) or Ctrl (Windows) is down.
Bit 4	Contextual menu click: right-click or Control-click (Macintosh).

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Control Structure blockReentry Field

The blockReentry field appears in the built-in structure for each type of control. It allows you to prevent Igor from sending your control action procedure another event while you are servicing the first event. This is useful for action procedures that take a long time to service a click event. In such cases you typically do not want to service a second click until you finish servicing the first. This technique prevents an accidental double-click on a button from invoking a time-consuming procedure twice.

You tell Igor that you want to block further events until your action procedure returns by setting the blockReentry field to 1 when your action procedure is called:

```
Function ButtonProc(ba) : ButtonControl
   STRUCT WMButtonAction &ba

   // Tell Igor not to invoke ButtonProc again until this invocation is finished
   ba.blockReentry = 1
   . . .
```

Igor tests this field before invoking your action procedure while it is already running from a previous invocation. You do not need to test this field or reset it to 0 - just set it to 1 to block reentry.

Control Structure blockReentry Advanced Example

This example further illustrates the use of the blockReentry field. It is of interest only to those who want to experiment with this issue.

The ReentryDemoPanel procedure below creates a panel with two buttons. Each button prints a message in the history area when the action procedure receives the "mouse up" message, then pauses for two seconds, and then prints another message in the history before returning. The pause is a stand-in for a procedure that takes a long time.

The top button does not block reentry so, if you click it twice in quick succession, the action procedure is reentered and you get nested messages in the history area.

The bottom button does block reentry so, if you click it twice in quick succession, the action procedure is not reentered.

Because of architectural differences, reentry of the action procedure occurs on Macintosh but not on Windows so on Windows, both buttons behave the same.

```
Function ButtonProc(ba) : ButtonControl
   STRUCT WMButtonAction &ba

   switch( ba.eventCode )
      case 2: // mouse up
         // Block bottom button only
         ba.blockReentry= CmpStr(ba.ctrlName,"Block") == 0
         print "Start button ",ba.ctrlName
         Variable t0= ticks
         do
            DoUpdate
         while(ticks < (t0+120) )
         Print "Finish button",ba.ctrlName
         break
```

```
   endswitch

   return 0
End

Window ReentryDemoPanel() : Panel
   PauseUpdate; Silent 1// building window...
   NewPanel /K=1 /W=(322,55,622,255)
   Button NoBlock,pos={25,10},size={150,20},proc=ButtonProc,title="No Block Reentry"
   Button Block,pos={25,50},size={150,20},proc=ButtonProc,title="Block
Reentry"
End
```

User Data for Controls

You can store arbitrary data with a control using the userdata keyword. You can set user data for the following controls: Button, CheckBox, CustomControl, ListBox, PopupMenu, SetVariable, Slider, and TabControl.

Each control has a primary, unnamed user data that is used by default. You can also store an unlimited number of different user data strings by specifying a name for each different user data string. The name can be anything you desire as long as it is a legal Igor name.

You can retrieve information from the default user data using the **ControlInfo** operation (page V-63), which returns such information in the S_UserData string variable. To retrieve any named user data, you must use the **GetUserData** operation (page V-224).

Although there is no size limit to how much user data you can store, it does have to be generated as part of the recreation macro for the window when experiments are saved. Consequently, huge user data can slow down experiment saving and loading

User data is intended to replace or reduce the usage of global variables.

Control User Data Examples

A simple example of user data with a button:

```
NewPanel
Button b0,userdata="user data for button b0"
Print GetUserData("","b0","")
```

A more complex example using user data for buttons. Copy the following code into the Procedure window of a new experiment and run the `Panel()` macro. Then click on the buttons.

```
Structure mystruct
   Int32 nclicks
   double lastTime
EndStructure

Function ButtonProc(ctrlName) : ButtonControl
   String ctrlName

   STRUCT mystruct s1
   String s= GetUserData("", ctrlName,"")
   if( strlen(s) == 0 )
      print "first click"
   else
      StructGet/S s1,s
      // Warning: Next command is wrapped to fit on the page.
      printf "button %s clicked %d time(s), last click = %s\r",ctrlName, s1.nclicks,
Secs2Date(s1.lastTime, 1 )+" "+Secs2Time(s1.lastTime,1)
   endif
   s1.nclicks += 1
   s1.lastTime= datetime
   StructPut/S s1,s
   Button $ctrlName,userdata= s
End

Window Panel0() : Panel
   PauseUpdate; Silent 1            // building window...
   NewPanel /W=(150,50,493,133)
```

```
    SetDrawLayer UserBack
    Button b0,pos={12,8},size={50,20},proc=ButtonProc,title="Click"
    Button b1,pos={65,8},size={50,20},proc=ButtonProc,title="Click"
    Button b2,pos={119,8},size={50,20},proc=ButtonProc,title="Click"
    Button b3,pos={172,8},size={50,20},proc=ButtonProc,title="Click"
    Button b4,pos={226,8},size={50,20},proc=ButtonProc,title="Click"
EndMacro
```

Action Procedures for Multiple Controls

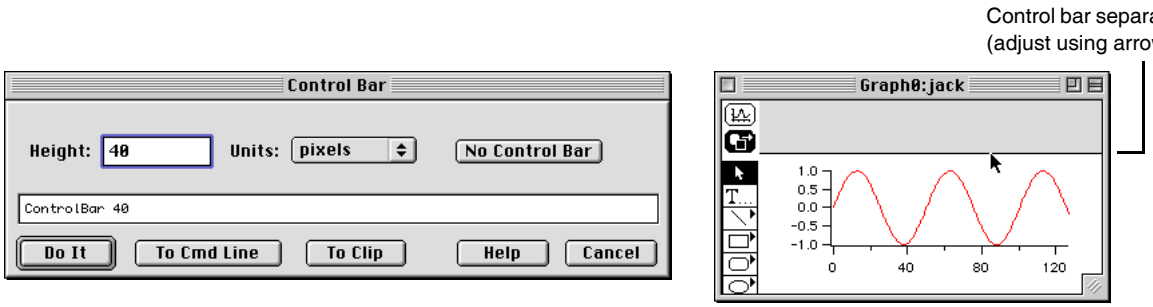
You can use the same action procedure for different controls of the same type (for all the Buttons in one window, for example). The name of the control is passed to the action procedure so that it can know which control was clicked. This is usually the name of the control *in the target/active window*, which is what most control operations assume.

When you call an action procedure from another procedure (rather than because a control was clicked), the name of the control may not be sufficient to identify the control; the window name may also be needed. This must be passed in through a global string. The window name can then be used to specify which window the named control is in by adding win=*winName* to the command, as in:

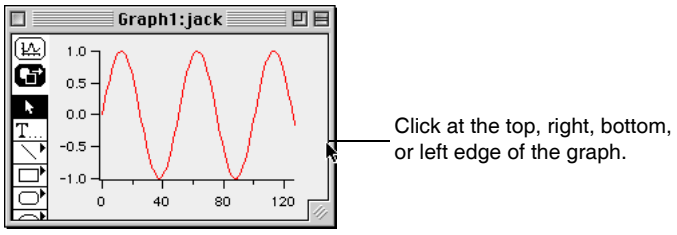
```
Button button0 win=$graphNameStr, title="Hello"
```

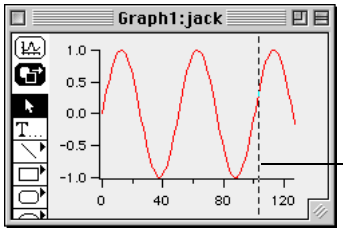
Controls in Graphs

Although controls can be placed anywhere in a graph, you can and should reserve an area just for controls at the edge of a graph window. Controls in graphs operate much more smoothly if they reside in these reserved areas. The **ControlBar** operation (page V-63) or the Control Bar dialog can be used to set the height of a nonembedded control area at the top of the graph.

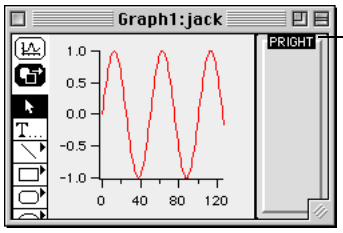


You can also embed a panel in the graph, which gives you more freedom on where to place the controls (see Chapter III-4, **Embedding and Subwindows** for further details and **Embedding into Control Panels** on page III-390 for a different approach). The simplest way to add a panel is to click near the edge of the graph and drag out a control area:

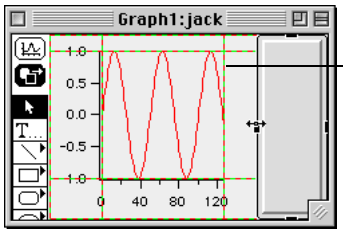




Drag the dashed line to define the inside edge of the embedded panel.



PRIGHT is the name of the resulting embedded panel subwindow. The label disappears in “operate” mode.



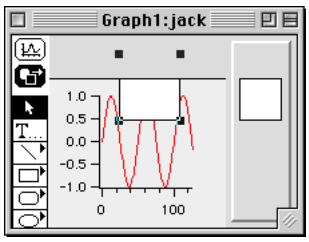
Adjust the position of the embedded window by clicking the subwindow frame and dragging its handles. The dashed lines represent the edges of the plot and graph areas, and the subwindow frame snap and attach to them.

The background color of a control area or embedded panel can be set by clicking the background to exit any subwindow layout mode, and then Control-clicking (*Macintosh*) or right-clicking (*Windows*) in the background and then selecting a color from the contextual menu’s pop-up color palette. See **Background Color** on page III-383 for further details.

The contextual menu adjusts the style of the frame around the panel.

You can use the same contextual menu to remove an embedded panel, leaving only the bare control area underneath. Remove the control area by dragging the inside edge back to the outside edge of the graph.

Drawing Limitations



The drawing tools can not be used in bare control areas of a graph. If you want to create a fancy set of controls with drawing tools, you will have to embed a panel subwindow into the graph.

Updating Problems

You may occasionally run into certain updating problems when you use controls in graphs. One class of update problems occurs when the action procedure for one control changes a variable used by a ValDisplay control in the same graph and also forces the graph to update while the action procedure is being executed. This short-circuits the normal chain of events and results in the ValDisplay not being updated.

You can force the ValDisplay to update using the **ControlUpdate** operation (page V-67). Another solution is to use a control panel instead of a graph.

The ControlUpdate operation can also solve problems in updating pop-up menus. This is described above under **PopupMenu** on page III-375.

Control Panels

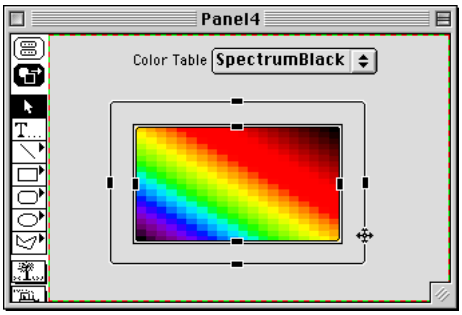
Control panels are windows designed to contain controls. The **NewPanel** operation (page V-440) is used to create a panel.

Drawing tools can be used in panel windows to decorate control panels. Only two drawing layers are provided and both are behind the controls layer. There is probably little reason to distinguish between the user and prog layers since users have no reason to draw in panels.

The Drawing tools can be used in panel windows to decorate control panels. Background color can be set by Control-clicking (*Macintosh*) or right-clicking (*Windows*) in the background and then selecting a color from the pop-up color palette. See **Background Color** on page III-383 for further details.

Embedding into Control Panels

You can embed a graph, table, notebook, or another panel into a control panel window (see Chapter III-4, **Embedding and Subwindows** for further details). You may find this preferable to putting control areas around a graph when you have many controls, or you may embed a graph in order to display an annotation, colorscale, or image plot. Use the contextual menu while in drawing mode to add an embedded window. Click on the frame of the embedded window to adjust the size and position.



You can use a notebook subwindow in a control panel to display status information or to accept lengthy user input. See **Notebooks as Subwindows in Control Panels** on page III-94 for details.

Exterior Subwindows

Exterior subwindows are panels that act like subwindows but live in their own windows attached to a host graph window. The host graph and its exterior subwindows move together and, in general, act as single window. Exterior subwindows have the advantage of not disturbing the host graph and, unlike normal subwindows, are not limited in size by the host graph.

Note: Exterior subwindows must be panels and the only host supported is a graph window.

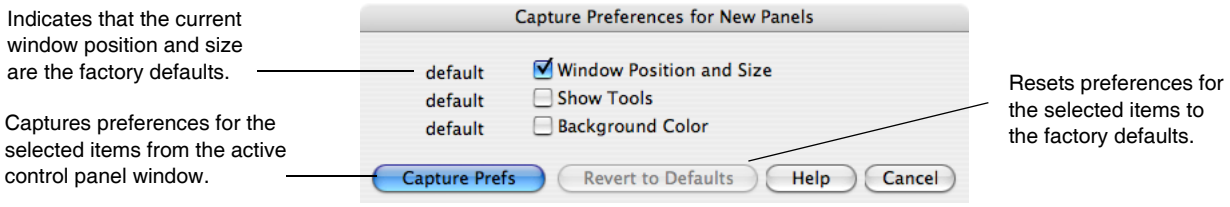
To create an exterior subwindow panel, use **NewPanel** with the /EXT flag in combination with /HOST.

Floating Panels

Floating control panels float above all other windows (except dialogs). To create a floating panel, use **NewPanel** with the /FLT flag.

Control Panel Preferences

Control panel preferences allow you to control what happens when you create a new control panel. To set preferences, create a panel and set it up to your taste. We call this your *prototype* panel. Then choose Capture Panel Prefs from the Panel menu.



Preferences are normally in effect only for *manual* operations, not for automatic operations from Igor procedures. This is discussed in more detail in Chapter III-17, **Preferences**.

When you initially install Igor, all preferences are set to the factory defaults. The dialog indicates which preferences you have changed.

The preferences affect the creation of new panels only.

Selecting the Show Tools category checkbox captures (or reverts) whether or not the drawing tools palette is initially shown or hidden when a new panel is created.

Controls Shortcuts

Action	Shortcut (*Macintosh*)	Shortcut (*Windows*)
To show or hide a panel's or graph's tool palette	Press Command-T.	Press Ctrl+T.
To move or resize a user-defined control without using the tool palette	Press Command-Option and click the control. With Command-Option still pressed, drag or resize it.	Press Ctrl+Alt and click the control. With Ctrl+Alt still pressed, drag or resize it.
To add a user-defined control without using the tool palette	Press Command-Option and choose from the Panel or Graph menu's Add Control submenu.	Press Ctrl+Alt and choose from the Panel or Graph menu's Add Control submenu.
To modify a user-defined control	Press Command-Option and double-click the control. This displays a dialog for modifying all aspects of the control. If the control is already selected, you don't need to press Command-Option.	Press Ctrl+Alt and double-click the control. This displays a dialog for modifying all aspects of the control. If the control is already selected, you don't need to press Ctrl+Alt.
To edit a user-defined control's action procedure	With the panel in modify mode (tools showing, second icon from the top selected) press the Control key and click the control. This displays a contextual menu with a "Go to <action procedure>" item.	With the panel in modify mode (tools showing, second icon from the top selected) right-click the control. This displays a contextual menu with a "Go to <action procedure>" item.
To create an embedded graph or table in the panel	With the panel in modify mode, press the Control key and click the panel background. Choose the subwindow type from the resulting contextual menu's "New" submenu.	With the panel in modify mode, right-click the panel background. Choose the subwindow type from the resulting contextual menu's "New" submenu.
To change an embedded window's border style	With the panel in modify mode, press the Control key and click the embedded window. Choose the border style from the resulting contextual menu's "Frame" and "Style" submenus.	With the panel in modify mode, right-click the embedded window. Choose the border style from the resulting contextual menu's "Frame" and "Style" submenus.
To remove an embedded window	With the panel in modify mode, press the Control key and click the embedded window. Choose the Delete from the resulting contextual menu.	With the panel in modify mode, right-click the embedded window. Choose the Delete from the resulting contextual menu.
To eliminate a control area at the edge of a graph	In modify mode or while pressing Command-Option, click the inside edge of the control area and drag it to the outside edge of the graph.	In modify mode or while pressing Ctrl+Alt, click the inside edge of the control area and drag it to the outside edge of the graph.
To nudge a user-defined control's position	Select the control and press arrow keys. Press Shift to nudge faster.	Select the control and press arrow keys. Press Shift to nudge faster.

Chapter
III-15

Platform-Related Issues

Platform-Related Issues

Igor Pro runs on Macintosh and Windows. This chapter contains information that is platform-specific and also information for people who use Igor on both platforms.

Windows-Specific Issues

On Windows, the name of the Igor program file must be “Igor.exe”, exactly. If you change the name, Igor extensions will not work because they will be unable to find Igor.

If you press Shift while launching Igor, Igor will skip loading extensions. (This feature is mainly of interest to the programmers at WaveMetrics who launch and quit Igor dozens of times a day during development.) If you want to do this on Windows, you should wait until you see Igor’s startup window before pressing the Shift key because Windows has its own interpretation for Shift during launch of a program.

If you save an experiment while a graph, table, layout, panel or notebook window is minimized, when you reopen that experiment, the window will again be minimized. This feature is not supported for any other kinds of windows, including the command window and all procedure windows. Instead, these other types of windows are reopened in their normal size.

Cross-Platform File Compatibility

Version 3.1 was the first version of Igor Pro that ran on Windows as well as Macintosh.

If you plan to use Igor on both platforms, it is a good idea to keep the same folder hierarchy for your Igor Pro files on both platforms. For example, if your Macintosh Igor files are in "hd:Igor Data Files: . . .", then it is best if you put your Windows Igor files in "C:\Igor Data Files\ . . .". Doing this will maximize the chances that Igor can find files referenced from Igor experiment files.

Experiment Files — Working with Earlier Versions

You may occasionally want to send an experiment file to a colleague who has an earlier version of Igor or open it on a computer with an earlier version of Igor.

If you use new features from a particular version of Igor Pro, you may get errors when you open an experiment file in an older version. Usually you can ignore or correct the errors and recover the file.

Crossing Platforms

When crossing from one platform to another, page setups are only partially translated. Igor tries to preserve the page orientation and margins.

When crossing platforms, Igor attempts to do font substitution where necessary. If Igor can not determine an appropriate font it will display the font substitution dialog where you can choose the font.

Platform-specific picture formats are displayed as gray boxes when you attempt to display them on the non-native platform. This includes the Macintosh PICT format and its variants when displayed on Windows and the Windows Metafile and Enhanced Metafile formats when displayed on Macintosh. The EPS, PNG, JPEG, and TIFF formats are platform-independent and are displayed on both platforms.

Transferring Files Using File Transfer Programs

Some transfer programs offer the option of translating file formats as they transfer the program from one computer to another. This translation usually consists of replacing each carriage return character with a carriage return/linefeed pair (Macintosh to Windows) or vice-versa (Windows to Macintosh). This is called a “text mode” transfer, as opposed to a “binary mode” transfer. This translation is appropriate for plain text files only. In Igor, plain text notebooks, procedure files, and Igor Text data files are plain text. All other files are not plain text and will be corrupted if you transfer in text mode. If you get flaky results after transferring a file, transfer it again making sure text mode is off.

If you have a problem opening a binary file after doing a transfer, compare the number of bytes in the file on both computers. If they are not the same, the transfer has corrupted the file.

File Name Extensions, File Types, and Creator Codes

On Windows, the file name extension indicates the nature of a file. When you double-click a file, Windows uses the extension to determine which program to launch. Mac OS 9 used the Mac-specific file type property to determine the nature of the file and the Mac-specific file creator code to determine which program to launch. Mac OS X still supports the file type and creator code properties but de-emphasizes them in favor of the file name extension. For this reason it is best to use the correct file name extension regardless of platform.

The file name extension and corresponding Macintosh file type for Igor Pro files are:

Extension	File Type	What's in the File
.pxp	IGsU	Packed experiment file
.pxt	IGsS	Packed experiment template (stationery)
.uxp	IGSU	Unpacked experiment file
.uxt	IGSS	Unpacked experiment template (stationery)
.ifn	WMT0	Igor formatted notebook (last character is zero)
.txt	TEXT	Igor plain notebook
.ihf	WMT0	Igor help file
.ibw	IGBW	Igor binary data file

The Macintosh creator code for Igor is 'IGR0' (last character is zero).

Experiments and Paths

An Igor experiment sometimes refers to wave, notebook, or procedure files that are stored separate from the experiment file itself. This is discussed under **References to Files and Folders** on page II-37. In this case, Igor creates a symbolic path that points to the folder containing the referenced file. It writes a **NewPath** command in the experiment file to recreate the symbolic path when the experiment is opened. When you move the experiment to another computer or to another platform, this path may not be valid. However, Igor goes to great lengths to find the folder, if possible.

Igor stores the path to the folder containing the file as a relative path, relative to the experiment file, if possible. This means that Igor will be able to find the folder, even on another computer, if the folder's location in the disk hierarchy is the same on both computers. You can minimize problems by using the same disk hierarchy on both computers.

If the folder is not on the same volume as the experiment file, then Igor can not use a relative path and must use an absolute path. Absolute paths cause problems because, although your disk hierarchy may be the same on both computers, often the name of the root volume will be different. For example, on the Macintosh your hard disk may be named "hd" while on Windows it may be named "C:". If Igor is unable to find a folder that is referenced by a full path, it looks for the folder in the same place in the hierarchy, but on the root volume containing the experiment file. If this fails, it looks for the folder in the same place in the hierarchy, but on the root volume containing the Igor application file. Also, if the path points inside the Igor Pro Folder, then Igor looks for the folder in the same place in the hierarchy, but inside the Igor Pro Folder containing the Igor application file.

If all of these methods fail, Igor displays a dialog asking you to locate the folder.

Picture Compatibility

Igor displays pictures in graphs, page layouts, control panels and notebooks. The pictures are stored in the Pictures collection (Misc→Pictures) and in notebooks. Graphs, page layouts and control panels reference pictures stored in the Pictures collection while notebooks store private copies of pictures.

This table shows the graphic formats that Igor can use to store pictures:

Format	How To Create	Notes
PICT	Paste or use Misc→Pictures	Macintosh only
PDF	Paste or use Misc→Pictures	Macintosh only
EMF (Enhanced Metafile)	Paste or use Misc→Pictures	Windows only
BMP (bitmap)	Use Misc→Pictures	Windows Only. BMP also called DIB (device-independent bitmap).
PNG (Portable Network Graphics)	Use Misc→Pictures	Cross-platform bitmap format
JPEG	Use Misc→Pictures	Cross-platform bitmap format
TIFF (Tagged Image File Format)	Use Misc→Pictures	Cross-platform bitmap format
EPS (Encapsulated PostScript)	Use Misc→Pictures	High resolution vector format. Requires PostScript printer. A screen preview is displayed on screen.

PICT was the standard format for Mac OS 9 graphics but has been supplanted by PDF on Mac OS X. EMF is the standard format for Windows graphics. PICT, PDF, EMF and BMP are platform-dependent and will display as gray boxes if you move the Igor experiment to the other platform. The other formats are platform-independent.

Although Igor does not display nonnative graphic formats, it does preserve them. For example, you can create an experiment on Macintosh and paste a Macintosh PDF into a page layout, graph, or notebook window. If you save the experiment and open it on Windows, the PDF will be displayed as a gray box. You can now paste a Windows metafile into the page layout, graph, or notebook window. If you save the experiment and open it on Macintosh, the Window metafile will be displayed as a gray box but the PDF will be displayed correctly. If you now save and open the experiment on Windows again, and the Windows metafile will be displayed correctly.

If you want PDF, PICT, EMF, or BMP/DIB pictures to be displayed correctly on both platforms, you must convert the pictures to PNG. To convert to PNG, use the Pictures dialog (Misc menu) for pictures in graphs and page layouts, or for pictures in notebooks, use the Special submenu in the Notebook menu.

Note: Converting a picture to PNG makes it a bitmap format and may degrade resolution. This is fine for graphics intended to be viewed on the screen but not for graphics intended to be printed at high resolution. You can convert to a high resolution PNG without losing much picture quality. However, this takes a lot of memory during the conversion and when the PNG is displayed. Because PNGs are compressed, they usually do not require excessive disk space when saved.

The ability to store pictures in JPEG, TIFF and EPS formats was added in Igor Pro 5. If you create an experiment with pictures in these formats, they will display as gray boxes if opened in older versions of Igor.

Prior to version 3.1, Igor stored graph and page layout pictures on the Macintosh in the experiment file's resource fork. When you transfer a file created by an Igor version older than 3.1 from the Macintosh to a PC, the resource fork is lost and Igor will display a placeholder rectangle in place of the picture.

Page Setup Compatibility

Page setup records store information regarding the size and orientation of the page. Page setups contain platform- and printer-dependent data. They are most important in regards to page layout windows but also affect printing of other kinds of windows.

On Macintosh, Igor stores page setups in experiment files, notebook files, and procedure files. In each experiment file, Igor stores a separate page setup for each page layout, notebook, and procedure window, and stores a single page setup for all graphs and single page setup for all tables.

On Windows, Igor stores page setups the same as on Macintosh except that it does not store page setups in plain notebook or procedure files. This is true whether the file is embedded in a packed experiment file or is a stand-alone file. When you open a plain notebook or procedure file on Windows, Igor creates a new page setup record by copying the preferred page setup record as set by the Capture Notebook Prefs or Capture Procedure Prefs dialogs.

When you transfer an experiment from one platform to another, page setup records are only partially preserved. Igor attempts to preserve the page orientation and margins.

Prior to version 3.1, Igor stored page setup records on the Macintosh in the experiment file's resource fork. When you transfer a file created by an Igor version older than 3.1 from from the Macintosh to a PC, the resource fork is lost and Igor has no way to know the orientation of the page setups. Therefore, when opening one of these old Macintosh experiment files on Windows, any page layouts in the experiment will use the page setup that you captured using the Capture Layout Prefs dialog or a default page setup if you have not captured a preferred page setup.

Pre-Carbon Page Setup Records

Carbon is a set of routines that Apple created for the transition from Macintosh OS 9 to OS X. The page setup record now used by the Macintosh operating system is called a Carbon page setup record. Prior to that the operating system used a different type of page setup record which we will call an "old Macintosh page setup record."

Experiments saved on Macintosh by pre-Carbon versions of Igor Pro (prior to version 4.05A) contain old page setup records. When you open one of these experiments under OS X, the old page setup record is passed to a system routine that is supposed to convert it to a Carbon page setup record. This usually works correctly but sometimes it corrupts the page setup record. The corruption usually appears as a nonsensical scaling value in the Page Setup dialog for the affected window. If the window is a page layout window, this results in a page that is grossly too small or too big. You can usually fix it by entering 100% for the scaling value in the page setup dialog.

Igor Pro 6 attempts to detect and fix this corruption if the scaling value is less than 50% or greater than 200%. It repairs the page setup record by replacing it with a new default page setup record and then sets the orientation of the new page setup record to match the orientation of the old record. All other properties, such as the scaling, of the old page setup record are lost.

If corruption occurs and the automatic repair is not successful, you can use the **SetIgorOption** operation (see page V-562) to control the old page setup record conversion. The command format is:

```
SetIgorOption RepairOldPageSetups = val
```

The parameter *val* is one of the following:

val	Effect
0	Never repair old page setup records.
1	Repair if the old page setup appears corrupted (default).
2	Always repair old page setup records.
3	Always present a page setup dialog.
4	Always repair and then present a page setup dialog.

The effect of SetIgorOption lasts only until you quit Igor.

File System Issues

This section discusses file system issues that you need to take into account if you use Igor on both Macintosh and Windows.

File and Folder Names

On Windows, the following characters are illegal in file and folder names: backslash (\), forward slash (/), colon (:), asterisk (*), question mark (?), double-quote ("), left angle bracket (<), right angle bracket (>), vertical bar (|). On Macintosh, the only illegal character is colon.

This means, for example, that you can not create a file with a name like "Data 1/23/98" on Windows. You can create a file with this name on Macintosh. If you write an Igor procedure that generates a file name like this, it will run on Macintosh but fail on Windows.

Therefore, if you are concerned about cross-platform compatibility, you must not use any of the Windows illegal characters in a file or folder name, even if you are running on Macintosh. Furthermore, you should not use "high ASCII" characters (see **Character Set Compatibility** on page III-401), because they don't translate across platforms. Also, don't use period except before a file name extension.

Prior to Igor Pro 6.1, the Macintosh version of Igor could not handle file or folder names exceeding 31 characters. Igor Pro 6.1 and later support long file names (up to 255 characters) on Macintosh as well as Windows. Most WaveMetrics XOPs now also support long file names on Macintosh. However XOP file names are still limited to 31 characters plus the ".xop" extension.

The following Igor Pro features will not work with long file names on Mac OS X because Igor calls Apple routines that do not support long file names for these features:

NewMovie
ImageSave when using QuickTime
ImageLoad when using QuickTime

File and folder names in Windows can theoretically be up to 255 characters in length. Because of some limitations in Windows and also in Igor, you will encounter errors if you use file names that long. However, both Igor and Windows are capable of dealing with file names up to about 250 characters in length. It is unlikely that you will approach this limit.

An exception to this is that Igor limits names of XOP files to 31 characters, plus the ".xop" extension. Igor will not recognize an XOP file with a longer name.

Paths in Windows are limited to 259 characters in length. Neither Windows nor Igor can deal with a path that exceeds this limit. For example, if you create a directory with a 250 character name and try to create a file with a 15 character name, neither Windows nor Igor will permit this.

This boils down to the following: Feel free to use long file and directory names, but expect to see errors if you use outrageously long names or if you have directories so deeply nested that paths approach the theoretical limit.

Path Separators

The Macintosh file system uses a colon to separate elements in a file path. For example:

```
hd:Igor Pro Folder:Examples:Sample Graphs:Contour Demo.pxp
```

The Windows file system uses a backslash to separate elements in a file path. For example:

```
C:\Igor Pro Folder\Examples\Sample Graphs:Contour Demo.pxp
```

Some Igor operations (e.g., LoadWave) allow you to enter file paths. Igor accepts Macintosh-style or Windows-style paths regardless of the platform on which you are running.

Note: Igor uses the backslash character as an escape character in literal strings. This can cause problems when using Windows-style paths.

For example, the following command creates a textbox with two lines of text. "\r" is an escape code inserts a carriage return character:

```
Textbox "This is line 1.\rThis is line 2."
```

Because Igor interprets a backslash as an escape character, the following command will not execute properly:

```
LoadWave "C:\Data Files\really good data.ibw"
```

Instead of loading a file named "really good data.ibw", Igor would try to load a file named "Data Files<CR>eally good data.ibw", where <CR> represents the carriage return character. This happens because Igor interprets "\r" in literal strings to mean carriage return.

To solve this problem, you must use "\\" instead of "\" in a file path. Igor will correctly execute the following:

```
LoadWave "C:\\Data Files\\really good data.ibw"
```

This works because Igor interprets "\\" as an escape sequence that means "insert a backslash character here".

Another solution to this problem is to use a Macintosh-style path, even on Windows:

```
LoadWave "C:Data Files:really good data.ibw"
```

Igor will convert the Macintosh-style path to a Windows-style path before using it. This avoids the backslash ambiguity.

For a complete list of escape sequences, see **Escape Characters in Strings** on page IV-13.

If you are writing procedures that need to extract sections of file paths or otherwise manipulate file paths, the **ParseFilePath** function on page V-472 may come in handy.

UNC Paths

"UNC" stands for "Universal Naming Convention". This is a Windows convention for identifying resources on a network. One type of network resource is a shared directory. Consequently, when running under Windows, in order to reference a network directory from an Igor command, you need to use a UNC path.

The format of a UNC path that points to a file in a folder on a shared server volume or directory is:

```
"\\server\share\directory\filename"
```

"server" is the name of the file server and "share" is the name of the top-level shared volume or directory on that server.

Because Igor treats a backslash as an escape character, in order to reference this from an Igor command, you would have to write:

```
"\\\\server\\share\\directory\\filename"
```

As described in the preceding section, you could also use Macintosh path syntax by using a colon in place of two backslashes. However, you can not do this for the "\\server\share" part of the path. Thus, using Macintosh syntax, you would write:

```
"\\\\server\\share:directory:filename"
```

Unix Paths

Unix paths use the forward slash character as a path separator. Igor does not recognize Unix paths. Use Macintosh paths instead.

FlushFileBuffers

On Windows, by default Igor calls the Windows FlushFileBuffers routine before closing a file. FlushFileBuffers flushes data cached in a hard drive's circuitry for the file being closed, ensuring that the data is written to disk. This guarantees the file's integrity in the event of a power failure.

When writing a lot of small files one right after another, for example when saving an unpacked experiment with a very large number of waves or data folders, calling FlushFileBuffers can adversely affect perfor-

mance. In this case, you can use SetIgorOption to disable calling FlushFileBuffers. This is rarely necessary and should be done only in the circumstances described in this paragraph.

To turn FlushFileBuffers off execute:

```
SetIgorOption UseFlushFileBuffers=0                // Turn off
```

To turn FlushFileBuffers on execute:

```
SetIgorOption UseFlushFileBuffers=1                // Turn on (default)
```

To query the state execute:

```
SetIgorOption UseFlushFileBuffers=?; Print V_Flag  // Query
```

This does nothing on Macintosh.

SetIgorOption is usually called from the command line. It can not be executed from a user-defined function. It's effect lasts until Igor quits.

Keyboard and Mouse Usage

This section describes how keyboard and mouse usage differs on Macintosh versus Windows. It is intended to help Igor users more easily adapt when switching platforms.

There are three main differences between Macintosh and Windows input mechanisms:

1. The Macintosh mouse may have one button and the Windows mouse has two.
2. The Macintosh keyboard has four modifier keys (Shift, Command, Option, Control) while the Windows keyboard has three (Shift, Ctrl, Alt).
3. The Macintosh keyboard has Return and an Enter keys while the Windows keyboard (usually) has two Enter keys.

For the most part, Igor maps between Macintosh and Windows input as follows:

Macintosh	Windows
Shift	Shift
Command	Ctrl
Option	Alt
Control	<not mapped>

Macintosh	Windows
Return	Enter
Enter	Enter
Control-click	Right-click

There are a few exceptions to this mapping. For example, Option-Tab enters a tab into a text wave in a table. Alt-Tab does not do this on Windows because Alt-Tab is reserved by the Windows operating system.

In notebooks, procedure windows and help windows, pressing Control-Return or Control-Enter executes the selected text or, if there is no selection, to execute the line of text containing the caret.

Command Window Input

This table compares command window mouse actions:

Action	Macintosh	Windows
Copy history selection to command line	Option-click	Alt+click
Copy history to command and start execution	Command-Option-click	Ctrl+Alt+click
Invoke contextual menu	Control-click	Right-click

You can quickly move the command window (or any built-in Igor window) to its preferred position. On Macintosh, press Option and click the zoom button. On Windows, press Alt and click the maximize button.

Other Input Issues

The accelerators (keyboard shortcuts) for Indent Left and Indent Right in the Edit menu are Command-[and Command-] on Macintosh but are Ctrl+Shift+L and Ctrl+Shift+L R on Windows. This is because Windows does not allow using Ctrl+[or Ctrl+] as an accelerator.

Here are some shortcuts related to the Igor help system:

Action	Macintosh	Windows
Invoke Igor Help Browser	Help	F1
Insert operation or function template	Shift-Help	Ctrl+F1
Go to help for operation or function	Shift-Option-Help	Ctrl+Alt+F1

Cross-Platform Text and Fonts

Character Set Compatibility

The Macintosh and Windows character encodings are the same for the common characters, such as upper and lower case letters, numbers and common punctuation. These are sometimes called the "low ASCII" characters because they have ASCII codes below 128 and appear in the first half of the ASCII character encoding table. The remaining characters, such as accented vowels, bullets and other symbols, are called "high ASCII" characters. The encoding for high ASCII characters is different on the Macintosh and Windows. For example, the character code for a bullet on the Macintosh is the character code for a yen symbol on Windows.

In most cases, Igor does an automatic translation to compensate for these encoding differences. For example, when Igor opens a formatted notebook file or a packed experiment file containing notebooks and procedures, Igor automatically translates. There are some characters for which no translation is possible because the characters are missing from one platform or the other. For example, the Macintosh has a "Š" character but Windows does not. For these characters, Igor does not do any translation. The character will appear incorrect on the nonoriginating platform but will appear correct if the file is moved back to the originating platform.

For some fonts, this translation causes a problem, because the fonts do not use the normal character encodings. Symbol font is an example. It has the same encoding on both platforms and therefore Igor does not do any translation.

When Igor opens a plain text notebook or a procedure file that is a stand-alone file (i.e., is not stored in a packed experiment file) it has no way to know which platform the file came from and therefore can not do character translation. This may result in funny characters appearing, such as a yen symbol where you expect a bullet symbol. If you use such characters in procedure files, this problem may cause errors when you compile or execute the procedures. The only solution is to avoid using high ASCII characters in plain text files.

Like the Symbol font, Asian fonts use the same character encoding on Macintosh and Windows and so Igor does not translate text governed by an Asian font. When opening an experiment using Asian text, it is

important that your preferred procedure file font be an Asian font. This is the factory default if you are running on an Asian version of the Macintosh or Windows operating system. This is important because Igor uses the preferred procedure file font in cases where it has no other way to determine which font should be used for the file, such as when you open a stand-alone procedure file on Macintosh or any procedure file (even packed into an experiment file) on Windows.

Text Styles

The outline and shadow text styles are available on Macintosh only. On Windows, these styles have no effect and Igor disables Outline and Shadow items in menus and Outline and Shadow checkboxes in dialogs.

Carriage Returns and Linefeeds

The character or character pattern that marks the end of a line of text in a plain text file is called the "line terminator". There are three common line terminators, carriage return (CR, ASCII 13, used on Macintosh), linefeed (LF, ASCII 10, used on Unix) and carriage return plus linefeed (CRLF, used on Windows).

When Igor Pro opens a text file (procedure file, plain text notebook or plain text data file), it accepts CR, LF or CRLF as the line terminator.

If you create a new procedure file or plain text notebook, Igor writes CR on Mac OS and CRLF on Windows. As of Igor Pro 5.04, if you open an existing plain text file, edit it and then save it, Igor preserves the original terminator as determined by examining the first line in the file.

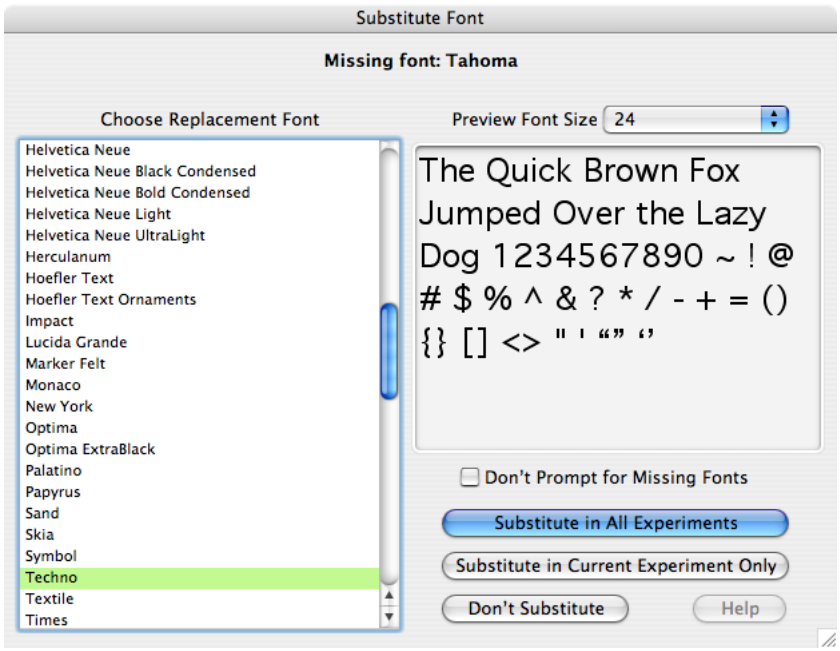
By default, the **FReadLine** operation treats CR, LF, or CRLF as terminators. Use this to write a procedure that can read lines from a text file without caring whether it is a Macintosh, Windows, or Unix file.

Font Substitution

When a font specified in a command or document is not installed Igor will apply font substitution to choose an installed font to use in place of the missing font. Dealing with these missing fonts often occurs when transferring a Windows-originated document to Macintosh or vice versa.

Igor employs two levels of font substitution: user-level editable substitution and built-in uneditable substitution.

The first level is an optional user-level font substitution facility that you will usually encounter for the first time when Igor displays the Substitute Font Dialog while opening an experiment or file. Use the dialog to choose a temporary or permanent replacement for the missing font:



Note: The Substitute Font Dialog appears when neither level of font substitution specifies a replacement for the missing font. You can prevent this dialog from appearing by selecting the Don't Prompt for Missing Fonts checkbox, which is also present in the Edit Font Substitutions Dialog.

The second level is Igor's built-in substitution table which substitutes between fonts normally installed on various Macintosh and Windows operating systems. For example, it substitutes Arial (a standard Windows font) for Geneva (a standard Macintosh font) if Geneva is not installed (which it usually isn't on a Windows computer).

If you prefer to replace Geneva with Verdana, you can use the Edit Font Substitution Dialog (accessed from the Misc menu) to edit the user-level substitutions, which take precedence over any built-in substitutions.

The user-level font substitution table is maintained in the "Igor Font Substitutions.txt" text file in Igor's preferences folder. The file format is:

```
<name of missing font to replace> = <name of font to use instead>
```

one entry per line. For example:

```
Palatino=New Times Roman
```

spaces or tabs are allowed around the equals sign.

When a missing font is replaced, Igor uses the name of the replacement font instead of the name of the font in the command.

The name of the missing font is *replaced* only in the sense that the altered or created object (window, control, etc.) uses and remembers only the name of the replacement font. Recreation macros (and experiment recreation procedures) use the name of the replacement font when the experiment is saved.

The command, however, is unaltered and still contains the name of the missing font.

Example

Suppose an opened experiment's recreation procedures contain commands like these:

```
Display                          // create a blank graph
ModifyGraph gFont="this font doesn't exist"
```

Suppose also that we haven't created an entry in the Edit Font Substitution dialog for a missing font named "this font doesn't exist".

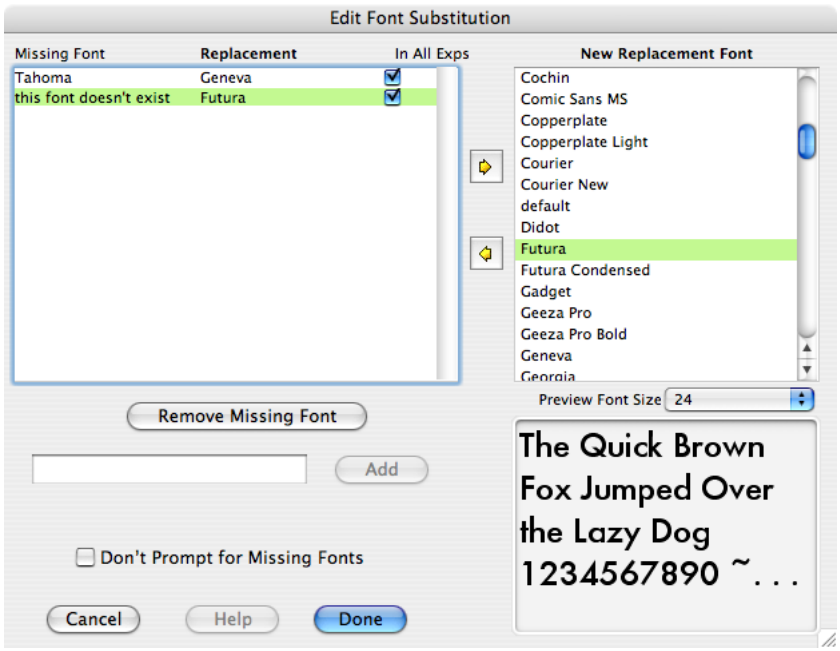
Because the `ModifyGraph gFont` command refers to a nonexistent font, the Substitute Font Dialog appears. Suppose we select a replacement font of Futura.

There are three consequences:

1) The recreation macro of the created graph looks like this:

```
Window Graph0() : Graph
   PauseUpdate; Silent 1      // building window...
   Display /W=(5,44,400,252)
   ModifyGraph gFont="Futura"
EndMacro
```

2) If *another* `ModifyGraph gFont="this font doesn't exist"` command is executed, the Substitute Font dialog does *not* appear, but silently converts the font to Futura.
3) Igor adds the font substitution to the Missing Font list you see in the Edit Font Substitution Dialog:



Cross-Platform Procedure Compatibility

Igor procedures are about 99.5% platform-independent. For the other 0.5%, you need to test which platform you are running on, using the **IgorInfo**(2) function, and act accordingly.

File Paths

As described under **Path Separators** on page III-398, the Macintosh uses colons to separate elements of a file path while Windows uses backslashes. So that you can write a single procedure that works on both platforms, Igor accepts paths with either colons or backslashes on either platform.

The use of backslashes is complicated by the fact that Igor uses the backslash character as an escape character in literal strings. This is also described in detail under **Path Separators** on page III-398. The simplest solution to this problem is to use a colon to separate path elements, even when you are running on Windows.

If you are writing procedures that need to extract sections of file paths or otherwise manipulate file paths, the **ParseFilePath** function on page V-472 may come in handy.

File Types and Extensions

On Mac OS 9, all files had a file type property. This property is a four letter code that is stored with the file by the Macintosh file system. For example, plain text files have the file type TEXT. Igor binary wave files have the file type IGBW. The file type property controls the icon displayed for the file and which programs can open the file.

In Mac OS X, the file type can be represented by the file type property or by the file name extension. Either or both can be used, with the extension preferred.

On Windows, the file type is indicated by the file name extension and files do not have file type properties.

For the most part, in Igor programming you do not need to worry about file types or extensions. However, there are some Igor operations and functions, such as Open and IndexedFile, which take a string parameter containing a file type or list of file types. For example, these commands present a dialog from which a user can choose plain text files or Igor formatted notebook files:

```
Variable fileRef
Open/R/T="TEXTWMT0" fileRef as ""
Close fileRef
```

If you use these operations and functions in procedures, then you need to understand how Igor maps file name extensions to file types.

Igor associates a file type with a file name extension. For example, Igor considers ".txt" files to be of type TEXT and considers ".ifn" files to be of type WMT0. The following table shows how Igor maps from extension to file type:

Windows Extension	Macintosh File Type	What's in the File
.uxp	IGSU	Unpacked experiment file
.uxt	IGSS	Unpacked experiment template (stationery)
.pxp	IGsU	Packed experiment file
.pxt	IGsS	Packed experiment template (stationery)
.ibw	IGBW	Binary wave file
.itx	IGTX	Igor Text file
.xop	IXOP	Igor extension file
.ibv	IGB-	Igor binary variable
.ibt	IGT-	Igor binary misc things
.txt	TEXT	Text file
.ifn	WMT0	Igor formatted notebook file
.ift	WMTS	Igor formatted notebook template (stationery)
.eps	EPSF	Encapsulated PostScript file
.rtf	RTF	Rich Text file
.tif	TIFF	TIFF file
.jpg	JPEG	JPEG file
.tga	TPIC	Targa file
.gif	GIFf	GIF file
.sgi	SGI	Silicon Graphics file
.png	.PNG	Portable Network Graphics file
.psd	8BPS	Photoshop file
.pct	PICT	Macintosh PICT file
.bmp	BMPp	Windows bitmap file
.*	????	Any type file

In addition to the standard Windows extensions listed above, Igor also recognizes files with the ".bwav" and ".awav" extensions. These are recognized as equivalent to ".ibw" and ".itx" for backward compatibility with old versions of Igor.

Igor also maps the following extensions to the associated file type. However, the file types listed in this table are fictitious. They are not file types that are really used on Macintosh but they still work, for example, in the Open operation.

Windows Extension	Fictitious Macintosh File Type	What's in the File
.ihf	IHLP	Igor help file (WMT0 files on Macintosh)
.ipf	IPRC	Procedure file (TEXT files on Macintosh)
.dat	.DAT	Data file
.csv	.CSV	Comma-separated file
.emf	.EMF	Enhanced metafile
.wmf	.WMF	Windows metafile
.bmp	.BMP	Bitmap file
.png	.PNG	PNG file
.htm	HTML	HTML file

Igor maps all extensions not listed above to the file type '????' which signifies "unknown file type". This includes files that have no extension.

The TextFile function will recognize only files of type TEXT. On Windows, a file must have the ".txt" extension to be of type TEXT.

The IndexedFile function requires that you supply a file type or a file extension as a parameter. Thus, the following commands will print the name of the first file in the Igor directory whose type is TEXT (extension is ".txt"):

```
Print IndexedFile(Igor, 0, "TEXT")
Print IndexedFile(Igor, 0, ".txt")
```

This will print the name of the second file of any type:

```
Print IndexedFile(Igor, 1, "????")
```

Points Versus Pixels

Most measurements of length in Igor are in terms of points. A point is roughly 1/72 of an inch.

A pixel is the area taken up by the smallest displayable dot on an output device such as a CRT or a printer. The physical width and height of a pixel depends on the device. CRTs typically display 60 to 120 pixels per inch (commonly called "dots per inch" or DPI). Printers typically display 150 to 2400 DPI.

The physical size of a screen pixel can vary from one CRT to another. The size of a pixel on a given CRT can often be adjusted by the user by twiddling knobs on the back of the CRT or using buttons on the front panel. Because of this, software programs don't know about or worry about the physical size of a pixel but instead deal with an assumed or "logical" size.

On the Macintosh, the logical width and height of a screen pixel is always 1/72 of an inch, which equates to a logical resolution of 72 DPI. The user can change the physical resolution but not the logical resolution. If a monitor's physical resolution is close to 72 DPI, there will be a pretty good match between the logical and physical sizes. On such monitors, if you make a graph 5 inches (360 points) wide, it will appear roughly 5 inches wide.

On Windows, the default logical width and height of a screen pixel is 1/96 of an inch which equates to a logical resolution of 96 DPI. The user can change both the physical resolution and the logical resolution using the Windows Display Properties control panel. It is common to have a significant discrepancy between the logical and physical sizes. In other words, if you make a graph 5 inches (360 points) wide, it may appear 6 or 7 inches wide. If you want the physical size of an object on the screen to match its logical size, you need to find an appropriate combination of physical and logical resolutions.

Window Position Coordinates

With one exception, Igor stores and interprets window position coordinates in units of points. For example the command

```
Display/W=(5, 42, 405, 242)
```

specifies the left, top, right, and bottom coordinates of the window in points relative to a reference point which is, roughly speaking, the top/left corner of the menu bar. Other Igor operations that use window position coordinates in points include Edit, Layout, NewNotebook, PlayMovie and MoveWindow.

The exception is the control panel window. To make it easier to create control panels that look the same on Macintosh and Windows, the NewPanel operation interprets its /W coordinates as pixels and operations that create or modify controls also interpret coordinates as pixels.

Most users do not need to worry about the exact meaning of these coordinates. However, for the benefit of programmers, here is a discussion of how Igor interprets them.

On the Macintosh, the reference point, (0, 0), is the top/left corner of the menu bar on the main screen. On Windows, the reference point is 20 points above the bottom/left corner of the main Igor menu bar. This difference is designed so that a particular set of coordinates will produce approximately the same effect on both platforms, so that experiments and procedures can be transported from one platform to another.

The coordinates specify the location of the content region, in Macintosh terminology, or the client area, in Windows terminology, of the window. They do not specify the location of the window frame or border.

On the Macintosh, a point is always interpreted to be one pixel.

On Windows, the correspondence between a point and a pixel can be controlled by the user, as described under **Points Versus Pixels** on page III-406. Since Igor stores window positions in units of points, if the user changes the number of pixels per point, the size of Igor windows in pixels will change. The exception is that control panels will not change because Igor interprets their coordinates as pixels.

Notebook Issues

On Macintosh, Igor stores the settings (font, size, style, etc.) for a plain text file in the file's resource fork. The data fork contains just the plain text. On Windows, text files have no resource fork. Therefore there is no way for Igor to store settings for a plain text file on Windows. When you open a plain text notebook or an experiment containing a plain text notebook on Windows, Igor uses preferences to set the notebook's text format. Thus, text format changes that you make to a plain text notebook are lost on Windows unless you capture them as your preferred text format.

PNG Pictures in Notebooks

If you want to display notebook pictures on both platforms, they must be in one of these cross-platform formats: PNG, JPEG, TIFF. Although EPS is a cross-platform format, the EPS screen preview on Macintosh is PICT, which is not cross-platform.

If you have pictures in PICT or EMF format and you want them to be viewable on both platforms, you should convert them to PNG. Igor uses PNGs for Igor help files which need to be platform-independent.

There are two ways to create a PNG picture in an Igor notebook. You can load it from a file using Misc→Pictures and then place it in a notebook or you can convert a picture that you have pasted into a notebook using Notebook→Special→Convert to PNG.

The Convert to PNG command in the Notebook→Special menu converts the selected picture or pictures into PNG. It skips selected pictures that are already PNG or that are foreign (not native to the platform on which you are running). You can determine the type of a picture in a notebook by clicking in it and looking at the notebook status line. If you select just one picture and do the conversion, Igor will let you undo it. However, if you select anything other than a single picture, the conversion will not be undoable. You can interrupt a conversion by pressing Command-period (*Macintosh*) or Ctrl+Break (*Windows*) and holding until Igor stops the conversion.

When you create a PNG file from within Igor, you can create it at screen resolution or at higher resolution. If you want crisp screen display when viewed at 100 percent magnification, use screen resolution when you create the PNG file. If you want better quality when viewed at higher magnifications and when printed, use higher than screen resolution when you create the PNG file. Using higher resolution requires a lot of memory during the conversion and when the picture is displayed but usually does not require excessive disk space when saved because PNGs are compressed. Once you have created a PNG file, you can load it into another program or into Igor using Misc→Pictures.

The next three paragraphs explain how Igor treats screen resolution PNGs differently from high resolution PNGs. You can skip this if you don't care about the technical details.

Normally, Igor stretches pictures if necessary to preserve the logical size of the pictures at different screen resolutions. However, PNG, being a bitmap type of picture, looks best when displayed without stretching. Therefore, Igor treats PNGs in notebooks specially. If the PNG was created at screen resolution, Igor displays the PNG without stretching. This gives a crisp, undistorted display when viewed at 100 percent magnification, but the physical size of the picture is not preserved. This is usually what you want for screen dumps and for graphics intended mostly for viewing on screen at 100% magnification as opposed to printing or viewing on screen at higher magnifications.

For example, suppose you create a 5 inch by 4 inch graph on the Macintosh and export that graph as a screen-resolution PNG. You paste the PNG into a notebook. On the Macintosh, the screen resolution is 72 dpi, so the PNG will be 72 dpi. Thus, the dimensions of the PNG in pixels is 360 by 288 (5*72 by 4*72). If you save the notebook and open it on Windows, Igor notices that the PNG is screen resolution and will therefore displays it using 360 by 288 pixels. This means that Igor does not need to stretch the PNG so it displays crisply. However, because Windows screen resolution is typically 96 dpi, the logical size of the PNG on Windows will be smaller than 5 inches by 4 inches by a factor of 72/96.

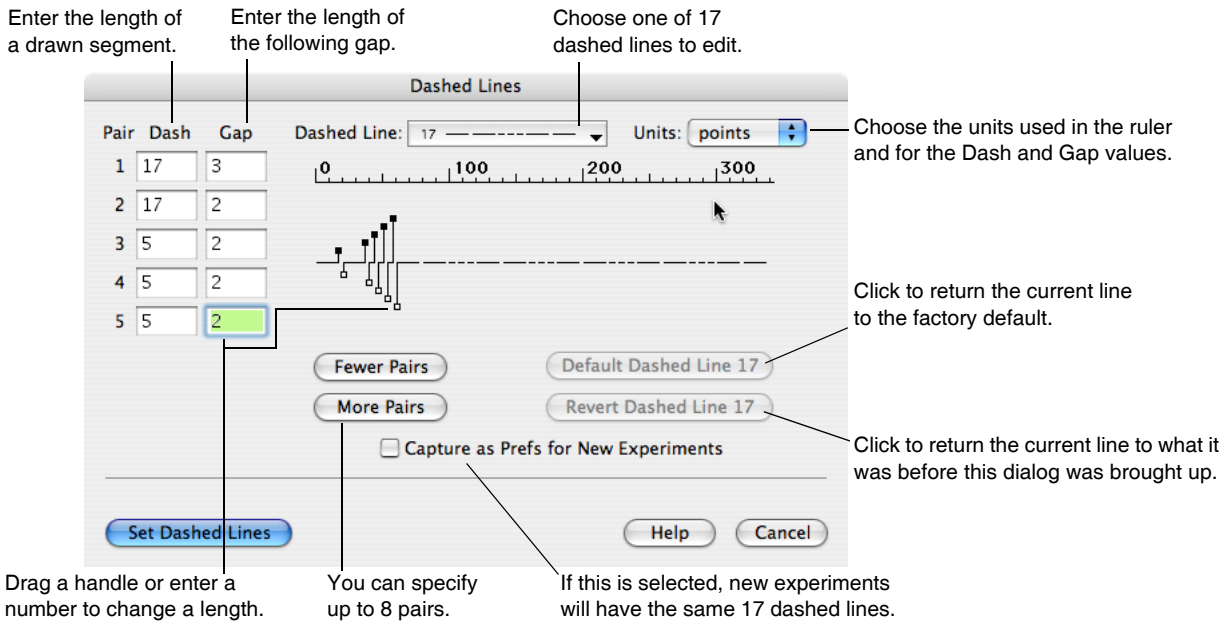
If instead of creating a screen-resolution PNG, you create a high-resolution PNG, Igor will stretch the PNG by a factor of 96/72 on Windows. This makes the logical size of the PNG the same on both platforms but it will be somewhat distorted on Windows because of the stretching.

Chapter
III-16

Miscellany

Dashed Lines

The dashed lines built into Igor can be edited with the Dashed Lines dialog (Misc menu).



You can edit one dashed line pattern at a time. The example shows the last dashed line (number 17) being edited. You can select another line with the Dashed Line pop-up menu. Dashed line 0 (the solid line) cannot be edited. If you need to create a custom dashed line pattern, we recommend that you modify the high numbered dashed lines, leaving the low number ones in their default state. This ensures that the low numbered patterns will be the same for everyone.

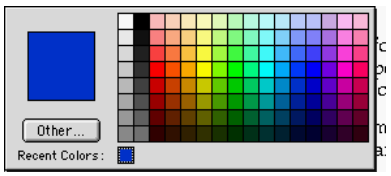
Dashed lines are described by pairs of dash and gap lengths. You can add or remove pairs with the More Pairs and Fewer Pairs buttons. The lengths can be adjusted by entering dash and gap values, or by dragging the dash (filled boxes) and gap (hollow boxes) handles.

You can also change dashed lines with the **SetDashPattern** operation (see page V-552).

Dashed lines are stored with the experiment, so each experiment may have different dashed lines. You can capture the current dashed lines as the preferred dashed lines for new experiments.

The Color Environment

Igor has a main color palette that contains colors that you can use for traces in graphs, text, axes, backgrounds and so on. The main color palette appears as a pop-up menu in a number of dialogs, such as the Modify Trace Appearance dialog. This section discusses this palette.



Igor also has color tables and color index waves you can select among when displaying contour plots and images. These are discussed in Chapter II-14, **Contour Plots**, and Chapter II-15, **Image Plots**.

You can select from colors presented in a color palette.

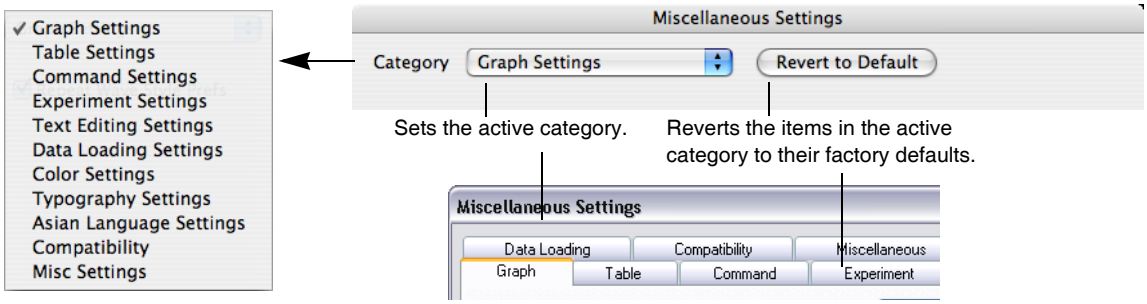
You can use the Other button to select colors that are not in the palette. As you use Igor, colors are added to the palette in the Recent Colors area.

On Macintosh only, the recent colors are remembered by Igor when it quits and restored when it restarts if you have selected the "Save and restore color palette's recent colors" checkbox in the Miscellaneous Settings dialog's Color Settings category.

If you run your monitor in 256 (8 bit) color mode, Igor has to make compromises in an attempt to provide reasonably accurate colors while still allowing other programs to display reasonably accurate colors. It is recommended that you set your monitor to the thousands of colors (16 bit) or millions of colors (32 bit) mode.

Miscellaneous Settings

You can customize some aspects of how Igor works using the Miscellaneous Settings dialog. The miscellaneous settings are grouped into categories.



Changes to miscellaneous settings take effect immediately when you click the Save Settings button. They affect the current experiment as well as new and old (reopened) experiments.

Graph Settings

The Repeat Wave Style Prefs in Graphs checkbox affects the way wave trace style preferences are applied when waves are appended to a graph. The factory default is selected. See **Graph Preferences** on page II-298.

When the Enable "Fling" Mode checkbox is checked, dragging in a graph with Option or Alt pressed and the mouse down will "fling" (scroll) the graph contents in the direction of the drag. To stop the motion, click in the graph while keeping the mouse still.

Table Settings

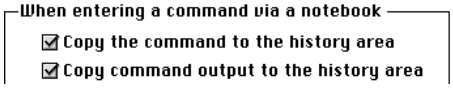
The Repeat Column Style Prefs in Tables checkbox affects the way column style preferences are applied when columns are appended to a table. The factory default is selected. See **Table Preferences** on page II-228.

Command Settings

The Limit Command History Text checkbox determines the number of lines of history text that Igor will retain in a given experiment. The factory default is deselected (no limit). Limiting history can save space on disk but more importantly can reduce clutter in the history area of the command window. It can also reduce the time it takes to open and save an experiment.

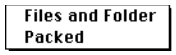
If you select the Limit Command History Text checkbox and enter the number of lines of history that you want to keep, Igor will automatically trim the history to that number of lines when you save the experiment. In addition, when Igor adds text to the history, it checks the number of lines. If it exceeds the requested maximum by 500 lines, Igor trims the history at that time, rather than waiting until you save the experiment. This is done so that if, for example, you run an experiment overnight, the history will not grow without bound.

You can enter and execute commands in a notebook as well as from the command window.



These settings let you determine whether Igor stores commands and output in the history area as well as in the notebook. See **Notebooks as Worksheets** on page III-5 for general information on using a notebook as a worksheet.

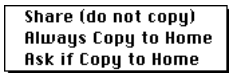
Experiment Settings



The Saved Experiment Format pop-up menu controls the way Igor experiments are saved. You can override this choice in the Save Experiment dialog. The choice here merely presets the dialog's Packed checkbox. The factory default is Packed. See **Saving Experiments** on page II-29 for details.

Unlike most spreadsheet and graphing programs, data can exist in Igor without being visible in a table. The "Make a new table for new experiments" checkbox controls whether you will get a new empty table when you choose New Experiment from the File menu. The default state is selected. This is convenient if you generally start working by entering data manually. If you do not enter data manually, you may find it more convenient to turn this checkbox off and create a table only when you need one.

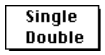
Data Loading Settings



The Loaded Igor Binary Data pop-up menu determines how Igor handles the loading of Igor Binary waves from disk files. This menu addresses a subtlety that caught many early Igor users off guard. Igor Binary data can be shared between experiments. Old versions of Igor defaulted to *sharing* loaded Igor Binary data, but many users thought that a loaded wave was another isolated *copy* of the data.

Use this pop-up menu to choose how Igor handles Igor Binary waves loaded from a disk file. Copy to Home makes an isolated copy of the wave in the experiment's home folder. If the experiment is saved in packed format, the isolated copy is stored in the experiment's packed file, which contains all the things usually stored in a home folder.

If you choose Ask if Copy to Home, you always get a chance to change your mind with a dialog. If you choose Share or Always Copy to Home, you can change your mind only if you use the Load Waves dialog; the Load Igor Binary dialog uses the setting you make here without asking for confirmation. The factory default is Ask if Copy to Home. See **Sharing Versus Copying Igor Binary Files** on page II-165 for details.



Default Data Precision presets the numerical precision checkboxes in the Make Waves and Load Waves dialogs. The factory default is Double. Note that this affects only the dialogs; if you use commands entered manually on the command line the default is single precision. See **Number Type and Precision** on page II-81 for details.

Color Settings (*Macintosh*)

Selecting the "Save and restore color palette's recent colors" checkbox restores the recent colors that were in the color palette when Igor quit. If deselected, Igor starts up with no recent colors. The factory default is deselected (don't restore colors).

The Color Palette menu sets the accuracy with which colors are drawn when you run in 256 color (8 bit) mode. It is recommended that you set your monitor to the thousands of colors (16 bit) or millions of colors (32 bit) mode.

Typography Settings (*Macintosh*)

This section discusses the way Igor draws text in graphs and page layouts.

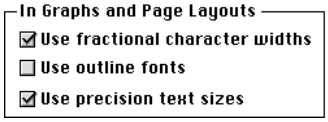
By default, Igor handles text in graphs and page layouts in a method designed to provide good readability of text on the screen and also good fidelity between text on the screen and text on the printed page. If you are satisfied with Igor's default performance in these regards, then you don't need to read the following material.

There is a trade-off to be made between readability of text on the screen and fidelity of text between the screen and the printed page. Unfortunately, there is no way to have maximum readability and maximum fidelity at the same time. You can control this trade-off via three checkboxes in the Typography section of the Miscellaneous Settings dialog.

Achieving text fidelity requires certain actions on the part of Igor, Apple's QuickDraw graphics software and the printer driver. Lack of fidelity stems from the fact that screen resolution is low while printer resolution is high. Because of roundoff and other effects, the width of a string on the screen does not necessarily

match the width of that same string when printed. The reasons for this are too complex to explain here. Apple's technical note TN-TEXT TE 21 gives some idea of the issues involved.

The "Use fractional character widths", "Use outline fonts", and "Use precision text sizes" checkboxes control aspects of how Igor draws text in graphs and page layouts.



The default and recommended state for the "Use fractional character widths" and "Use precision text sizes" checkboxes is on. Turning "Use outline fonts" on makes text in some fonts, such as Geneva and Monaco, difficult to read on screen. Because of this, the default state for this checkbox is off.

When the "Use fractional character widths" checkbox is on, Igor uses Apple's QuickDraw to keep track of the position of characters using higher precision arithmetic. This slightly distorts text on the screen but improves the fidelity with printed text.

When the "Use outline fonts" checkbox is on, Igor uses Apple's QuickDraw to create the outline versions of fonts (TrueType or PostScript) instead of the bitmap versions. This also distorts text on the screen but improves the fidelity with printed text. The "Use outline fonts" checkbox makes a big difference with some fonts, such as Geneva, Monaco and Courier, and makes little difference with other fonts, such as Helvetica, Palatino and Times. Results on your computer may be different because you have different versions of these fonts, different system software or different printer driver software.

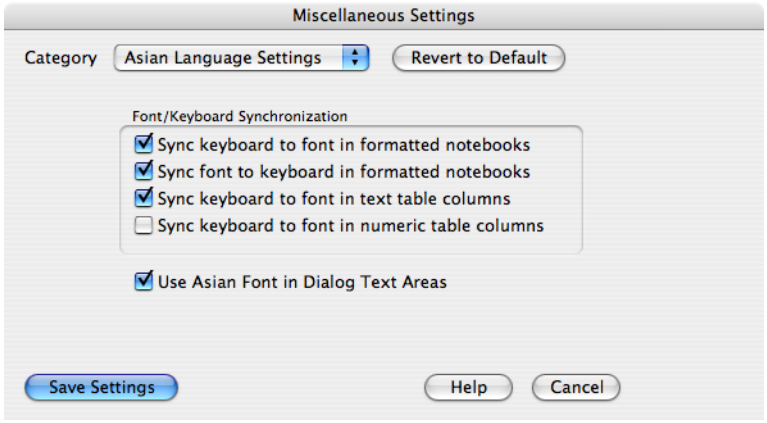
The "Use precision text sizes" checkbox to do a more precise calculation when converting screen text sizes to printer text sizes. The only reason to turn this feature off is to make this version of Igor behave like Igor Pro 2.0.

If all three checkboxes are deselected, you will get the same behavior as in Igor Pro 2.0. This produces good looking text but with less emphasis on text fidelity. Use it only for backward compatibility.

If you change the state of the "Use fractional character widths" or "Use outline fonts", Igor will automatically update all open graphs and page layouts. This is not needed if you change just the Use precision text sizes checkbox because this does not affect what is drawn on the screen.

Asian Language Settings (*Macintosh*)

The Font/Keyboard Synchronization settings control whether or not Igor will automatically match the keyboard input method and the active font. Synchronization applies to text documents and to tables.



The "Sync keyboard to font in text windows" checkbox determines whether Igor will set the keyboard script when you choose a font or click in a section of text in a particular font. If this is selected and you click in Asian text, Igor will activate the Asian keyboard method. If you click in Roman text, Igor will deactivate the Asian keyboard method. Igor chooses the Asian keyboard method if the font is Asian and the text at the insertion point in the document is Asian. Thus, if you mix Asian and Roman characters all in an Asian font, Igor will activate the appropriate keyboard for the text at the insertion point.

The "Sync font to keyboard in text windows checkbox" determines whether Igor will change the font automatically if you manually change the keyboard input method. If this is selected and you change the keyboard input method to Asian, Igor will automatically switch to an Asian font. If you change the keyboard input method to Roman, Igor will automatically switch to a Roman font. You may prefer to enter Roman text in an Asian font. If so then you should deselect this checkbox and manually change the font as needed.

The "Sync keyboard to font in table text columns" checkbox determines whether Igor will set the keyboard script when you click in a text column in a table. If this is on and you click in a column whose font is Asian, Igor will activate the Asian keyboard methods. If you click in a column whose font is Roman, Igor will deactivate the Asian keyboard method.

The "Sync keyboard to font in table numeric columns" checkbox determines whether Igor will set the keyboard script when you click in a numeric column in a table. We recommend that you leave this deselected and use Roman numbers in numeric columns because Igor currently does not understand non-Roman numbers.

The "Use Asian Font for Dialog Text Areas" checkbox controls the font that Igor will use for certain special dialog text entry areas, such as the text entry area in the Add Annotation and Browse Waves dialogs. Because they require advanced capabilities such as scrolling and undo, these areas are not standard Macintosh text boxes but rather are implemented using WaveMetrics' special routines. Normally these areas use the Geneva font. This prevents Asian users from entering Asian characters. When the Use Asian Font for Dialog Text Areas checkbox is selected, Igor will use an Asian font for these areas.

Compatibility Settings

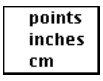
The "Native GUI Appearance for User-defined Controls" checkbox sets the default user-defined control appearance. Deselect it to use the Igor Pro 5 user-defined control appearance.

Deselecting the Floating Tool Palettes and Floating Graph Info Palettes checkboxes sets the tool and info palette locations to be internal to the associated window, as in Igor Pro 5, rather than using the default floating palettes. The settings will affect only new ShowInfo and ShowTools commands; current info and tool palette locations are not affected.

Misc Settings

The "Use short menu bar names" checkbox, when selected, shortened some menu names. If deselected, it uses long menu names. The factory default is deselected (long names). You may want to select this if you are running on a small screen or have many user-defined menus.

The "Maximum pop-up menu items" setting limits the number of menu items in some pop-up menus, such as the pop-up menus of waves, variables, string variables, and the Current Object pop-up menu in the Object Status dialog. Choosing a small value (the minimum is 50) prevents excessive start-up time for dialogs at the expense of listing all the items in the pop-up menus. We recommend that you set this to several hundred. Igor displays the spinning hand watch cursor when building large pop-up menus. You can type Command-period (*Macintosh*) or Ctrl+Break (*Windows*) to stop building the pop-up menu. Some dialogs have multiple pop-up menus that cause the cursor to reappear; you can type Command-period (*Macintosh*) or Ctrl+Break (*Windows*) again.



"Preferred units of length" controls the default units used in the Modify Graph, Print Graph, Export Graph, and Save Graphics dialogs. It does not affect page layouts or notebooks. These have their own preference settings for units (use the Capture Layout Preferences and Capture Notebook Preferences dialogs). This also does not affect the control dialogs (for adding buttons, checkboxes, etc.). These dialogs always use points as the default unit of measure.

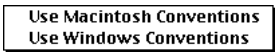
"Operations that overwrite or delete folders" determines what to do when a MoveFolder or CopyFolder command is about to overwrite a folder on disk or when a DeleteFolder command is about to delete a folder on disk.

The factory default setting is "Display dialog to ask user for permission". This means that Igor will ask for permission each time one of these operations is about to overwrite or delete a folder on disk. This setting is intended to reduce the chance of accidental or malicious deletion of files.

The other choices are "Always give permission" and "Always deny permission". "Always give permission" grants blanket permission to overwrite or delete folders. Choosing this increases the risk of accidental or malicious deletion of files so you should exercise caution.

(*Macintosh*) "Use Command-H for Find Selection (instead of Command-Control-H)" controls the keyboard shortcut for the "Find Selection" menu item in the Edit menu. The Find Selection keyboard shortcut defaults to Command-Control-H, allowing Command-H to hide Igor on Mac OS X, per Apple's recommendation. The Igor Pro 4 keyboard shortcut of Command-H can be restored by selecting this checkbox. When selected, hiding Igor on Mac OS X requires selecting "Hide Igor" with the mouse.

Keyboard Navigation affects the behavior of keys such as Page Down, Page Up, End and Home.



The Windows Menu Shows popup menu determines how target windows are identified in the submenus of the Windows menu. By default, just the window's title is displayed. You can choose to display the title and/or the name using this setting.

Help Settings (*Windows*)

These are controls governing the display of tool tips, the little windows that pop up automatically when you point at buttons and icons.

Object Names

Every Igor object has a name which you give to the object when you create it or which Igor automatically assigns. You use an object's name to refer to it in dialogs, from commands and from Igor procedures. The named objects are:

Waves	Data folders	Variables (numeric and string)
Windows	Symbolic paths	Pictures
Annotations	Controls	Rulers

In Igor Pro, the rules for naming waves and data folders are not as strict as the rules for naming all other objects, including string and numeric variables, which are required to have standard names. These sections describe the standard and liberal naming rules.

Standard Object Names

Here are the rules for standard object names:

- May be 1 to 31 characters in length.
- Must start with an alphabetic character (A-Z or a-z).
- May include alphabetic or numeric characters or the underscore character.
- Must not conflict with other names (of operations, functions, etc.).

All names in Igor are case insensitive. wave0 and WAVE0 refer to the same wave.

Characters other than letters and numbers, including spaces and periods, are not allowed. We put this restriction on names so that Igor can identify them unambiguously in commands, including waveform arithmetic expressions.

Historical Note: Prior to Igor Pro 3.0, wave names were limited to 18 characters.

Liberal Object Names

The rules for liberal names are the same as for standard names except that almost any character can be used in a liberal name. Liberal name rules are allowed for waves and data folders only.

If you are willing to expend extra effort when you use liberal names in commands and waveform arithmetic expressions, you can use wave and data folder names containing almost any character. If you create liberal names then you will need to enclose the names in single (not curly) quotation marks whenever they are used in commands or waveform arithmetic expressions. This is necessary to identify where the name ends. Liberal names have the same rules as standard names except you may use any character except control characters and the following:

```
"  '  :  ;
```

Here is an example of the creation and use of liberal names:

```
Make 'wave 0';    // 'wave 0' is a liberal name
'wave 0' = p
Display 'wave 0'
```

Note: Providing for liberal names requires extra effort and testing on the part of Igor programmers (see **Programming with Liberal Names** on page IV-147) so you may occasionally experience problems using liberal names with user-defined procedures.

Name Spaces

When you refer to an object by name, in a user function for example, each object must be referenced unambiguously. In general, an object must have a unique name so. Sometimes the object type can be inferred from the context, in which case the name can be the same as objects of other types. Objects whose names can be the same are said to be in different name spaces.

Data folders are in their own name space. Therefore the name of a data folder can be the same as the name of any other object, except for another data folder at the same level of the hierarchy.

Waves and variables (numeric and string) are in the same name space and so Igor will not let you create a wave and a variable in a single data folder with the same name.

An annotation is local to the window containing it. Its name must be unique only among annotations in the same window. The same applies for controls and rulers. Data folders, waves, variables, windows, symbolic paths and pictures are global objects, not associated with a particular window.

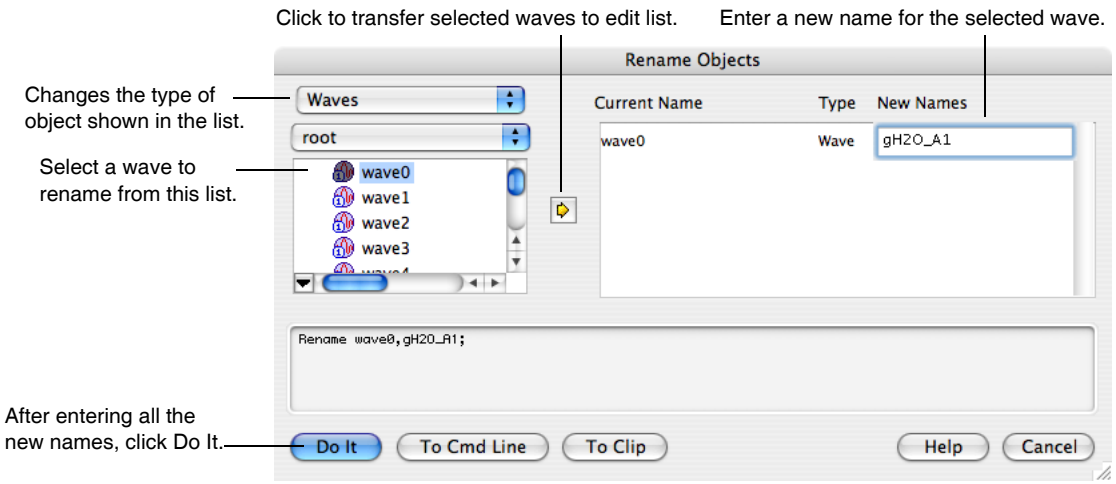
The names of global objects, except for data folders, are required to be distinct from the names of macros, functions (built-in, external or user-defined) and operations (built-in or external).

Here is a summary of the four global name spaces:

Name Space	Requirements
Data folders	Names must be distinct from other data folders at the same level of the hierarchy.
Waves, variables, windows	Names must be distinct from other waves, variables (numeric and string), windows.
Pictures	Names must be distinct from other pictures.
Symbolic paths	Names must be distinct from other symbolic paths.

Renaming Objects

You can use Misc→Rename Objects or Data→Rename to rename waves, variables, strings, symbolic paths, and pictures. Both of these invoke the Rename Objects dialog.



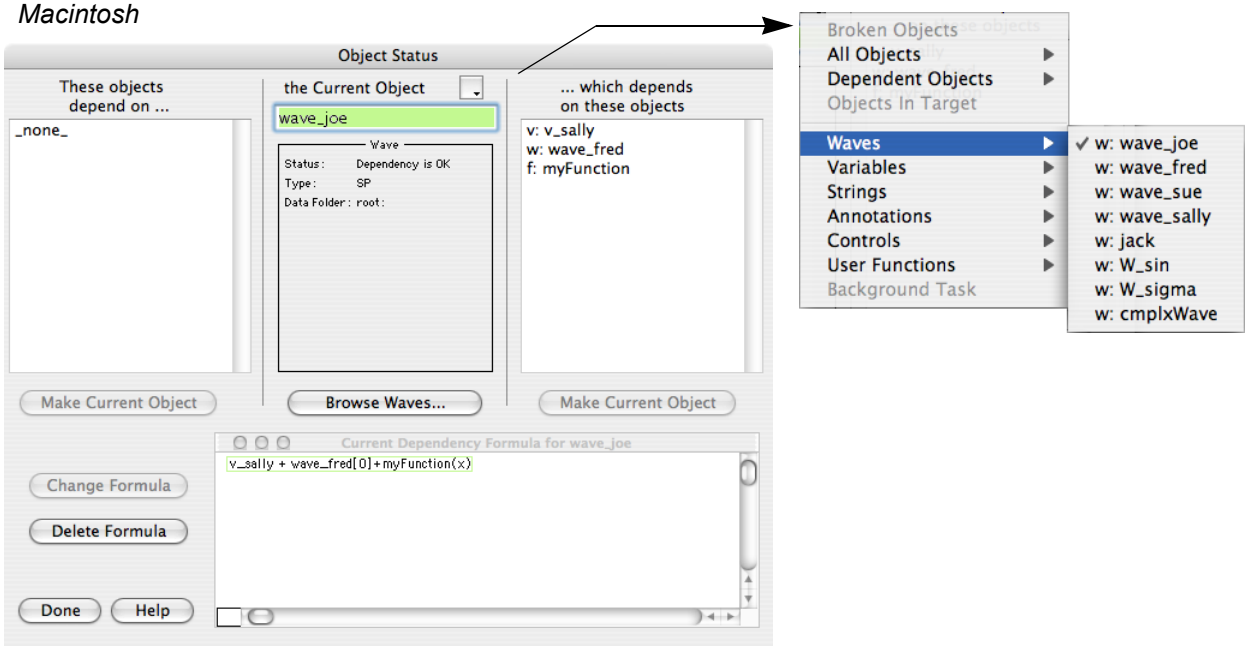
Graphs, tables, page layouts, notebooks, control panels and XOP target windows (e.g., Gizmo 3D plots) are renamed using the **DoWindow** operation (see page V-122) which you can build using the Window Control dialog. See **The Window Control Dialog** on page II-64.

You can use the DataBrowser to rename data folders, waves, and variables. See **Data Browser** on page II-130.

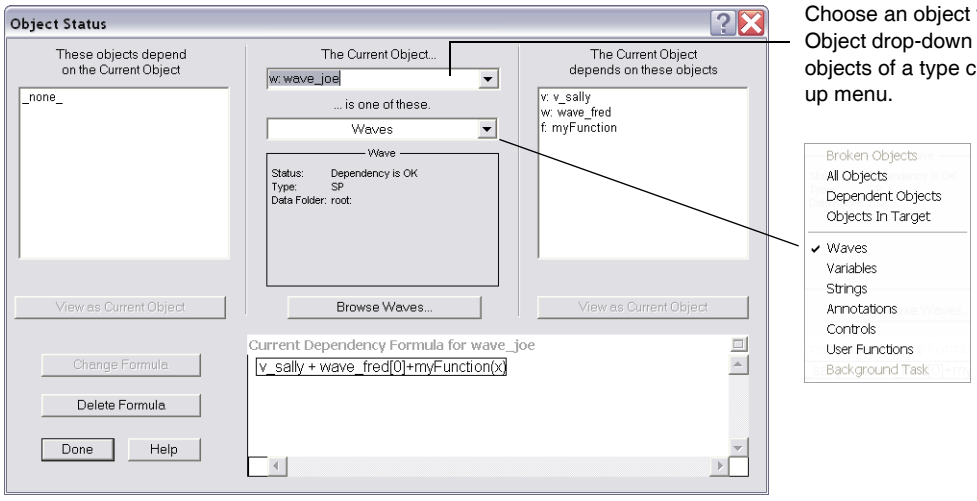
The Object Status Dialog

You can examine the status and interdependencies of named Igor objects with the Object Status dialog. See Chapter IV-9, **Dependencies**, for a discussion of object dependencies. You won't need to use this dialog unless you've got a fairly complex Igor experiment with lots of objects.

Macintosh



Windows



The dialog displays the properties of one object, called the "Current Object". The name of the current object appears at the top center of the dialog. You can choose a new current object from the Current Object pop-up menu.

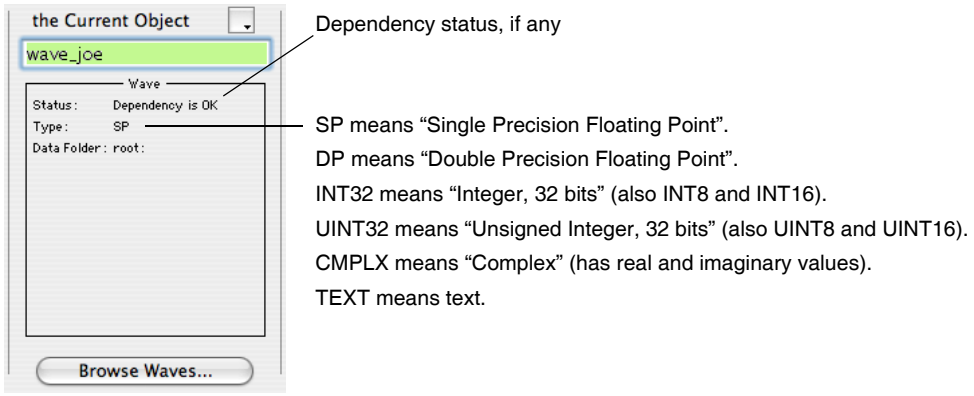
You can also choose a new current object by selecting an object from either of the two **dependency lists** on the left and right of the current object and clicking View as Current Object, or by double-clicking in either list.

Items in the lists and submenus are object names preceded by a key indicating the type of object:

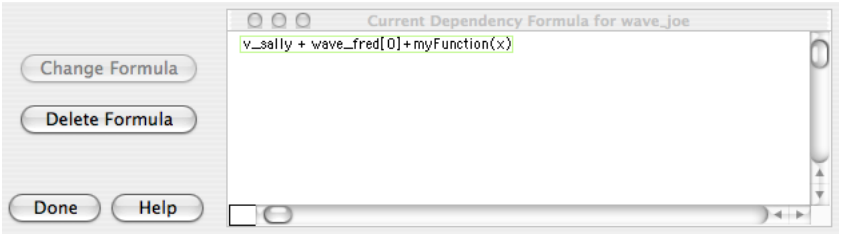
Key	Object Type
a:	annotation
c:	control
f:	function
s:	string

Key	Object Type
t:	task (background task)
v:	variable
w:	wave
=:	dependency formula

The current value, dependency status, and other information about the current object is displayed in the box below the name of the current object. Depending on the type of object, a button may also be present for editing or examining that object.

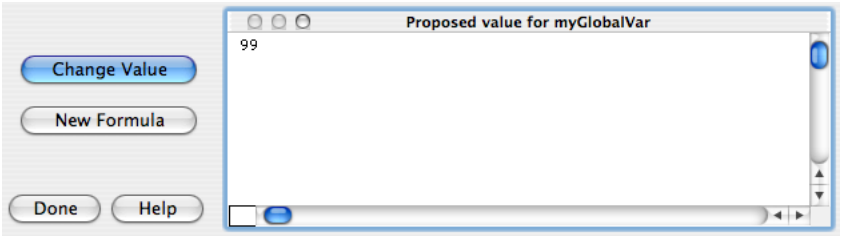


Other current object information is displayed in a windoid near the bottom of the dialog. If the object is a "dependent" object, the dependency formula is shown. This formula can be changed or deleted by clicking the appropriate buttons:

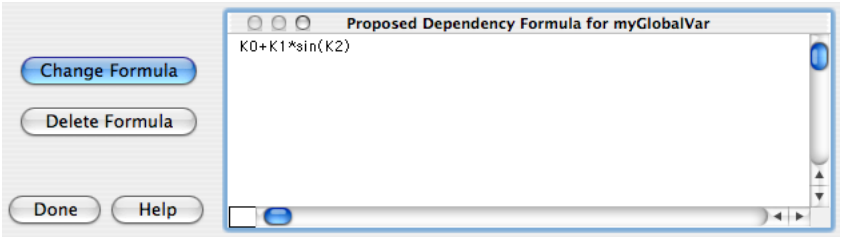


Dependencies and dependency formulas are explained further in Chapter IV-9, **Dependencies**.

You can edit the current value of most nondependent objects in the windoid:



For some nondependent objects, you can also establish a new dependency formula. Click the New Formula button, and an empty formula is created. Enter the desired formula and click Change Formula:



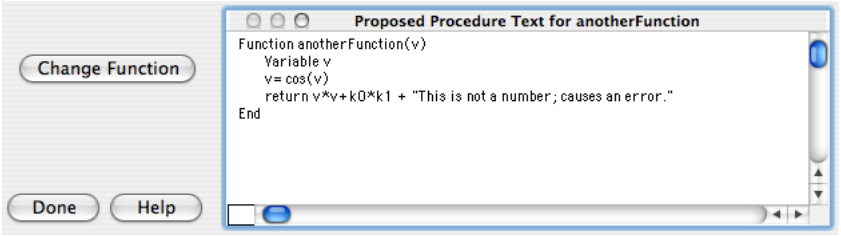
This example established the dependency formula:

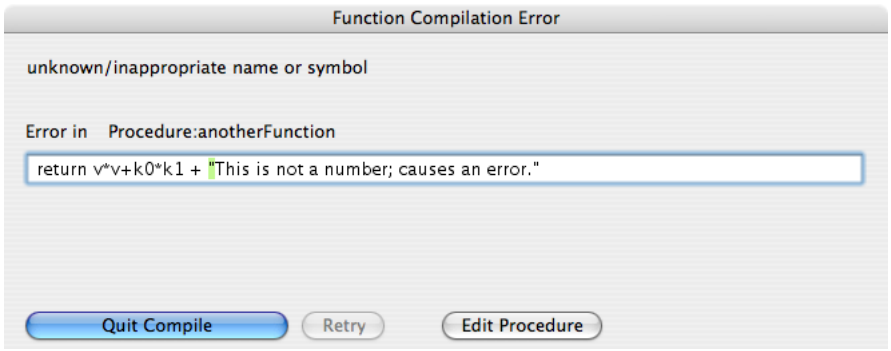
```
myGlobalVar := K0+K1*sin(K2)
```

Whenever `K0`, `K1`, or `K2` changes, `myGlobalVar` will be updated using this formula.

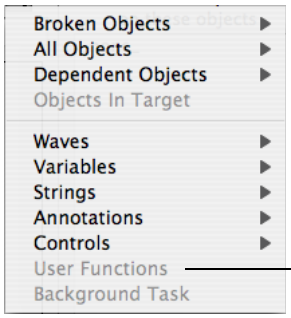
User Functions

You can edit the "current value" of a user-defined function, too. However, if you introduce an error and click Change Function, the function compiler will display an error dialog.





If you then click Quit Compile, the Object Status dialog will come back up, but it will appear as if all the user-defined functions have vanished.



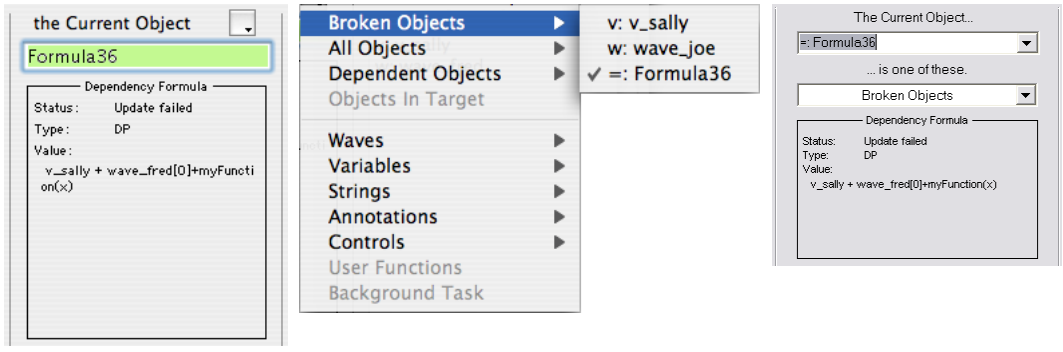
User Functions are not listed when an unresolved compile error exists.

The functions have *not* vanished; Igor just can't display information about functions unless all the procedure windows compile successfully. When you fix the error, the user-defined functions will return.

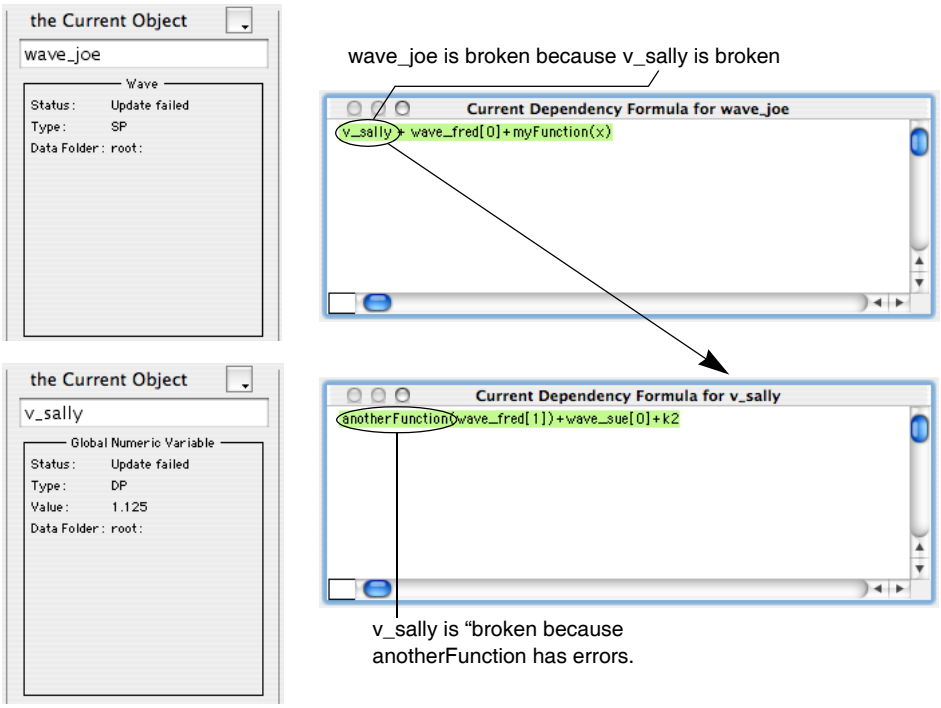
For this reason, it really makes more sense for you to click the Edit Procedure button in the error dialog. This brings up the procedure window in which the function is defined, but it does exit the Object Status dialog first (which would be devoid of function information, anyway). You should fix the error, and then reselect the Object Status dialog. You will find that the function is still the current object, and that all is well.

Broken Objects

If you *don't* fix errors in the user-defined function text, objects that reference the erroneous function will be "**broken**". Those objects are listed in the Broken Objects submenu. The Status field will show why the object is broken.



The usual causes for "Update failed" are either a syntax error in the dependency formula, or some object that the current object depends on (either directly *or indirectly*) has an error, has been renamed, or no longer exists. In this example, it is the unresolved error in the function `anotherFunction` (see **User Functions** on page III-419) which prevents `v_sally` from updating, which in turn causes the update of `wave_joe` through `Formula36` to fail:



Graphics Technology

As of version 6.1, Igor Pro uses more modern graphics code for drawing graphs, tables and page layouts. On Macintosh this involves the radical change of using Apple's Quartz routines rather than the ancient QuickDraw routines. On Windows just slightly more advanced code is used with a small amount of GDI+ instead of GDI.

The new graphics code supports arbitrary rotation for text and imported pictures and, on Macintosh, support for imported PDF pictures which can be placed in graphs, page layouts and formatted notebooks.

The new code does not support the old Macintosh PICT format or the old Windows WMF format. If you need to export PICT or WMF, you can make Igor use the old graphics code by executing this:

```
SetIgorOption UseOldGraphics=1
```

When this command is executed, all graphs and page layout windows are redrawn using the old code.

Windows users may need to revert to the old graphics when exporting EMF to certain programs that do not support the advanced GDI mode. In particular, uses have reported Corel products as not accepting advanced GDI.

You can also turn the old graphics code on if you have a problem with the new code. In this case, please let us know why you needed to do that so we can address the problem.

Pictures

Igor can import pictures from other programs for display in graphs, page layouts and notebooks. It can also export pictures from graphs, page layouts and tables for use in other programs. Exporting is discussed in Chapter III-5, **Exporting Graphics (Macintosh)**, and Chapter III-6, **Exporting Graphics (Windows)**. This section discusses how you can import pictures into Igor, what you can do with them and how Igor stores them.

For information on importing images as data rather than as graphics, see **Loading Other Files** on page II-167.

Importing Pictures

There are three ways to import a picture.

- Pasting from the Clipboard into a graph, layout, or notebook
- Using the Pictures dialog (Misc menu) to import a picture from a file or from the Clipboard
- Using the **LoadPICT** operation (see page V-347) to import a picture from a file or from the Clipboard

Each of these methods, except for pasting into a notebook, creates a *named*, global picture object that you can use in one or more graphs or layouts. Pasting into a notebook creates a picture that is local to the notebook.

This table shows the types of graphics formats from which Igor can import pictures:

Format	How To Import	Notes
PICT	Paste or use Misc→Pictures or LoadPICT	Macintosh only
PDF	Paste or use Misc→Pictures or LoadPICT	Macintosh only
EMF (Enhanced Metafile)	Paste or use Misc→Pictures or LoadPICT	Windows only
BMP (Windows bitmap)	Use Misc→Pictures or LoadPICT	Windows only BMP is sometimes called DIB (Device Independent Bitmap).
PNG (Portable Network Graphics)	Use Misc→Pictures or LoadPICT	Cross-platform bitmap format
JPEG	Use Misc→Pictures or LoadPICT	Cross-platform bitmap format
TIFF (Tagged Image File Format)	Use Misc→Pictures or LoadPICT	Cross-platform bitmap format
EPS (Encapsulated PostScript)	Use Misc→Pictures or LoadPICT	High resolution vector format. Requires PostScript printer. On Windows, a screen preview is displayed on screen.

EPS files usually include a screen preview. The screen preview format is PICT on Macintosh and TIFF on Windows. The screen prevew is not used or needed on Macintosh as the OS converts and displays EPS as PDF on the fly.

The platform-dependent formats (PDF, PICT, EMF, BMP) are drawn as gray boxes when displayed on the nonnative format. The cross-platform formats can be displayed on either platform, except that Macintosh EPS pictures display as boxes on Windows, because the screen preview format (PICT) is platform-dependent. Also, EPS pictures which have no screen preview display as boxes on Windows.

EPS pictures provide the highest quality but on Windows should only be used if you are certain you will be printing on a PostScript printer or exporting to a PostScript-savvy application such as Adobe Illustrator. On Macintosh, they can be used wherever a PDF could be used.

When non-EPS pictures are used in a graphic that is exported as EPS, the picture is rendered as a bitmap which has lower quality than an EPS picture.

See also **Picture Compatibility** on page III-395 for a discussion of Macintosh graphics on Windows and vice-versa.

The Picture Collection Stores Named Pictures

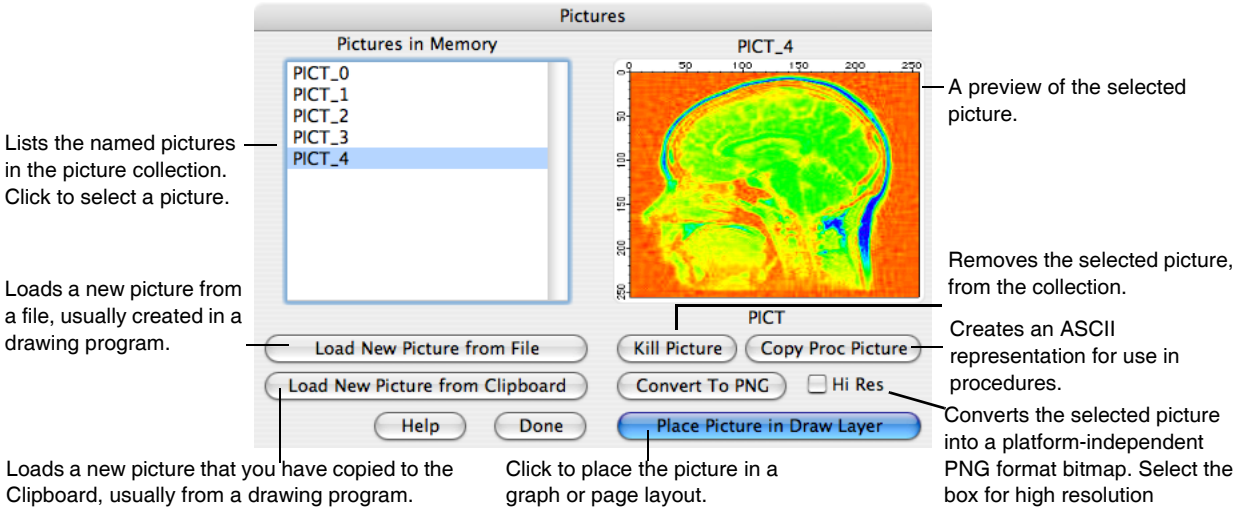
When you create a named picture using one of the techniques listed above, Igor stores it in the current experiment's **picture collection**. When you save the experiment, the picture collection is stored in the experiment file. You can inspect the collection using the Pictures dialog via the Misc menu.

Igor gives names to pictures so they can be referenced from an Igor procedure. For example, if you paste a picture into a layout, Igor assigns it a name of the form "PICT_0" and stores it in the picture collection. If you then close the layout and ask Igor to create a recreation macro, the macro will reference the picture by name.

You can rename a named picture using the Pictures dialog in the Misc menu, the Rename Objects dialog in the Misc menu, the Rename dialog in the Data menu, or the **RenamePICT** operation (see page V-520). You can kill a named picture using the Pictures dialog or the **KillPICTs** operation (see page V-322).

Pictures Dialog

The Pictures dialog permits you to view the picture collection, to add pictures, to remove pictures and to place a picture into a graph or page layout. It also can place a copy of a picture into a formatted notebook. To invoke it, choose Pictures from the Misc menu.



The Kill This Picture button will be dimmed if the selected picture is used in a currently open graph or layout.

Note: Igor determines if a picture is in use by checking to see if it is used in an *open* graph or layout window.

If you kill a graph or layout that contains a picture and create a recreation macro, the recreation macro will *refer* to the picture by name. However, Igor does not check for this. It will consider the picture to be unused and will allow you to kill it. If you later run the recreation macro, an error will occur when the macro attempts to append the picture to the graph or layout. Therefore, don't kill a picture unless you are sure that it is not needed.

NotebookPictures

When you paste a picture into a formatted notebook, you create a notebook picture. These work just like pictures in a word processor document. You can copy and paste them. These pictures will not appear in the Pictures or Rename Objects dialogs.

Igor Extensions

Igor includes a feature that allows a C or C++ programmer to extend its capabilities. An Igor extension is called an "XOP" (short for "external operation"). The term XOP comes from that fact that, originally, adding a command line operation was all that an extension could do. Now extensions can add operations, functions, menus, dialogs and windows.

WaveMetrics XOPs

The "Igor Pro Folder/Igor Extensions" and "Igor Pro Folder/More Extensions" folders contain XOPs that we developed at WaveMetrics. These add capabilities such as file-loading and instrument control to Igor and

also serve as examples of what XOPs can do. These XOPs range from very simple to rather elaborate. Most XOPs come with help files that describe their operation.

The WaveMetrics XOPs are described in the XOP Index help file, accessible through the Windows→Help Windows submenu.

Third Party Extensions

A number of Igor users have written XOPs to customize Igor for their particular fields. Some of these are freeware, some are shareware and some are commercial programs. WaveMetrics publicizes third party XOPs through our Web page. User-developed XOPs are available from http://www.igorexchange.com. Also, we make some third party XOPs available via FTP. See **FTP Sites** on page II-15 for details on FTP.

Activating Extensions

The Igor installer creates a folder called Igor Extensions inside the Igor Pro Folder. The Igor installer puts some XOP files in this folder. XOPs in this folder are loaded when Igor starts up.

The installer puts less-frequently used XOPs in "Igor Pro Folder/More Extensions". These are not available unless you activate them.

If you place in "Igor Pro Folder/Igor Extensions" an alias (Macintosh) or shortcut (Windows) for an XOP file or a folder containing XOP files, Igor loads those files also. However, this is not the recommended way to activate an XOP.

Your operating system may not allow you to make changes to the contents of your Igor Pro Folder, for example, if you do not have permission to write to that folder. For that reason Igor Pro 6.1 or later also loads extensions from another folder - "Igor Pro User Files/Igor Extensions" (see **Igor Pro User Files** on page II-46 for details). You can locate this folder by selecting Help→Show Igor Pro User Files.

If you press the shift key while clicking the Help menu, you can choose Help→Show Igor Pro Folder andUser Files. Igor then opens the Igor Pro Folder and the Igor Pro User Files folder on the desktop, making it easy to drag aliases/shortcuts into "Igor Pro User Files/Igor Extensions". This is the recommended way to activate additional XOPs for a given user.

If you want to activate an extension for all users on a given machine, you can put the alias or shortcut in "Igor Pro Folder/Igor Extensions" folder. Windows 7 does not permit you to manually create a shortcut in the Igor Pro Folder but you can create the shortcut on the desktop and drag it into the Igor Pro Folder.

Changes that you make to either Igor Extensions folder take effect the next time Igor is launched.

See **Creating Igor Extensions** on page IV-181 if you are a programmer interested in writing your own XOPs.

XOPs on Intel Macintosh

To run an XOP on Intel Macintosh it must be ported to the Intel architecture. WaveMetrics has ported most of its XOPs and distributes them as universal executables, which means that they run on both PowerPC and Intel Macintosh. The Igor Pro 6 application is also universal.

Because third-party libraries are not yet available, the following WaveMetrics XOPs have not yet been ported to run with the Intel version of Igor Pro: NIGPIB, NIGPIB2.

In addition, the NetCDF Loader XOP, written by Igor user Katsuhisa Kitano and ported to Mac OS X by WaveMetrics, has not yet been ported to Intel.

If you use a third-party XOP that has not yet been ported to Intel, you can not run it with the Intel version of Igor.

The Igor Pro installer places non-Intel WaveMetrics Macintosh XOPs in the PPC Extensions folder.

If you run the Intel Macintosh version of Igor Pro with a PowerPC-only XOP installed, it will not display an error message and will ignore the PowerPC-only XOP. Similarly, if you run the PowerPC version of Igor with an Intel-only XOP installed, it will not display an error message and will ignore the Intel-only XOP.

Running PowerPC XOPs on Intel Macintosh

If you need to run an XOP that has not been ported to Intel Macintosh, you must run the PowerPC version of Igor Pro. You can do this on an Intel Macintosh by checking the Open Using Rosetta checkbox in the Finder Info window for the Igor Pro application located in the Igor Pro folder.

With the Open Using Rosetta checkbox checked, the PowerPC code in the Igor Pro 6 application package will run any PowerPC XOPs that you invoke.

Memory Management

On Mac OS X, each 32-bit application (such as Igor Pro) is allocated 4 GB of virtual address space. On Windows, 32-bit applications are allocated 2 GB of virtual address space by default but there is a technique (explained below) for increasing this to 3 GB or 4 GB.

Theoretically, applications can allocate memory up the limit of the virtual address space. The operating system allocates physical memory as necessary. If the total amount of allocated virtual memory exceeds the total amount of physical memory, the operating system will temporarily write some data to disk and load other data into physical memory — a process called "swapping". This can make the computer very slow.

When a program is launched, the operating system maps the program's code into its virtual address space, thereby diminishing the space available for data. During the program's initialization, it allocates many blocks of data, thousands in Igor's case, for things like windows, controls and internally-used structures. This further reduces the amount of virtual memory space available for data.

As you work, creating windows, procedures, variables, waves and other objects, Igor allocates more blocks of virtual memory. When you kill a window, wave or other object, Igor deallocates blocks of virtual memory. This leaves free blocks in the virtual memory space.

Blocks of virtual memory space that are free because of deallocation are generally discontiguous. The largest contiguous block is smaller than the total amount of free virtual memory. The free virtual memory space is split into fragments that can not be joined because they are separated by allocated blocks. As you allocate and deallocate memory, the virtual memory space becomes more and more fragmented.

Another cause of fragmentation is the loading of dynamic link libraries.

When you create an object, for example, a wave, Igor needs a contiguous block of virtual memory. If the space needed for the wave is larger than the largest contiguous block of virtual memory, the allocation will fail and Igor will return an out-of-memory error.

Fragmentation sets the limit for the largest wave you can create. This is an issue if you are creating very large waves that require, typically, 250 MB or more. If you are creating a large number of smaller waves, for example, 100 waves of 10 MB each, fragmentation is generally not an issue.

Increasing Virtual Memory Space in Windows VISTA and Windows 7

If you are running Windows VISTA x64 (64-bit) or Windows 7 x64 (64-bit), the operating system allows 32-bit applications like Igor to use 4 GB of virtual address space. You can not increase this.

On Windows 7 x64 with 4 GB of physical memory the largest wave we were able to create was 2000 MB.

If you are running Windows VISTA (32-bit) or Windows 7 (32-bit), by default applications get a 2 GB virtual address space. You can increase this to 3 GB using the BCDEdit program to change your boot settings. To do this, click the Windows Start button, choose Programs→Accessories and then right-click Command Prompt. Choose Run As Administrator. In the Command Prompt window, enter this command:

```
BCDEdit /Set IncreaseUserVa 3072
```

Now reboot. Igor should now have a 3 GB virtual address space.

Increasing Virtual Memory Space in Windows XP

If you are running Windows XP Professional x64 (64-bit), the operating system allows 32-bit applications like Igor to use 4 GB of virtual address space. You can not increase this.

If you are running Windows XP Professional (32-bit), by default applications get a 2 GB virtual address space. You can increase this to 3 GB by changing a setting on your computer, specifically by adding the /3GB flag to your C:\boot.ini file.

Note: Be careful! If you mess up your boot.ini file, your computer will not boot. This procedure should be attempted only by advanced Windows computer users.

The boot.ini is invisible by default. To see it you must use Explorer's Tools→Folder Options to enable viewing hidden files and disable hiding protected system files. Alternatively you can use the System control panel, Advanced Tab, Startup and Recovery section to edit it.

Back up the boot.ini file before making any changes to it.

Open boot.ini in Notepad and add the /3GB flag to the appropriate partition. For example, after adding the flag, the relevant line in boot.ini may look like this:

```
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
/fastdetect /3GB
```

When you next boot, the operating system will allow Igor to use 3 GB of virtual address space instead of 2 GB.

The /3GB flag has no effect on programs that do not have the special flag set.

On Windows XP SP3 with 2 GB of physical memory and a 3 GB virtual address space, the largest wave we were able to create was about 600 MB. We were able to increase this to 880 MB by applying a Microsoft hotfix that reduces memory fragmentation. The hotfix is available from http://support.microsoft.com/kb/894472.

Macintosh System Requirements

Igor Pro 6.1 requires Mac OS X 10.4.0 or later. It runs native on PowerPC and Intel processors.

Igor Pro 6 does not run on Mac OS 9.

Windows System Requirements

Igor Pro requires Windows XP, Windows Vista or Windows 7.

Igor Pro 6.1 does not run on Windows 95, Windows 98, Windows ME, Windows NT or Windows 2000.

Crashes

A crash results from a software bug and prevents a program from continuing to run. Crashes are highly annoying at best and, at worst, can cause you to lose valuable work.

WaveMetrics uses careful programming practices and extensive testing to make Igor as reliable and bug-free as we can. However in Igor as in any complex piece of software it is impossible to exterminate all bugs. Also, crashes can sometimes occur in Igor because of bugs in other software, such as printer drivers, video drivers or system extensions.

We are committed to keeping Igor a solid and reliable program. If you experience a crash, we would like to know about it.

When reporting a crash to WaveMetrics, please include the following information:

- The exact version of Igor Pro (e.g., 6.10) that you are running.
- The exact operating system (e.g., Mac OS X 10.4.7, Windows XP) that you are running.
- A description of what actions preceded the crash and whether it is reproducible.
- A recipe for reproducing the crash, if possible.
- A crash log (described below), if possible.

We have three methods for determining the cause of a crash.

The first method is to reproduce it on our computers. For this we need a recipe from you, if possible.

If we can not reproduce the crash, our second method is to examine the source code looking for possible bugs. For this, we need a detailed description from you of what you were doing when the crash occurred so that we know what part of the source code to examine.

Our last resort is to examine the crash log for a clue as to where the crash occurred. This sometimes provides useful information, sometimes not.

Crash Logs on Mac OS X

When a crash occurs on Mac OS X, most of the time the system is able to generate a crash log. You can usually find it at:

```
/Users/<user>/Library/Logs/DiagnosticReports/Igor Pro_<date>_<machinename>.crash
```

or

```
/Users/<user>/Library/Logs/CrashReporter/Igor Pro.crash.log
```

where <user> is your user name.

For irreproducible crashes, send this log as an attachment to support@wavemetrics.com. Include the other information listed above.

Crashes On Windows

When a crash occurs on Windows XP, but not Windows VISTA or Windows 7, most of the time the system is able to generate a crash log. You can usually find it at:

C:\Documents and Settings\All Users\Application Data\Microsoft\Dr Watson\drwtsn32.log

The Application Data folder is normally hidden. To make it visible, go into the All Users folder and choose Tools→Folder Options. Click the View tab, then the Show Hidden Files and Folders radio button, and the OK button.

You can also search for the drwtsn32.log file. In this case, make sure to include hidden files and folders in the search.

For irreproducible crashes, send this log as an attachment to support@wavemetrics.com. Include the other information listed above.

If you install a software development system such as Microsoft Visual C++, the development system will set itself up as the default debugger and Dr. Watson will not run when a crash occurs. In this case you will have no drwtsn32.log file.

On Windows VISTA and Windows 7, the operating system does not produce a user-accessible crash log. Your best bet is to try to determine a recipe to reproduce the crash or a pattern that leads to the crash and send a report to support@wavemetrics.com.

Chapter III-17

Preferences

Overview

Preferences affect the creation of *new* graphs, panels, tables, layouts, notebooks, and procedure windows, and the *appending* of traces to graphs and columns to tables. In addition, preferences affect the command window, default font, and font size menus in new experiments.

You can turn preferences off or on using the Misc menu. Normally you will run with preferences on.

Preferences are automatically off while a procedure is running so that the effects of the procedure will be the same for all users. See the **Preferences** operation (see page V-497) for further information.

When preferences are off, factory default values are used for settings such as graph size, position, line style, size and color. When preferences are on, Igor applies your preferred values for these settings.

Preferences differ from *settings* (the Miscellaneous Settings dialog) in that settings generally take effect immediately, while preferences are used when something is created. See **Miscellaneous Settings** on page III-411.

Igor Preferences Directory

Preferences are stored in a per-user directory. The location of this directory depends on your operating system and configuration, but here are some typical locations:

Mac OS X	hd:Users:<*user*>:Library:Preferences:WaveMetrics:Igor Pro 6 PowerPC: *or* hd:Users:<*user*>:Library:Preferences:WaveMetrics:Igor Pro 6 Intel:
Windows	C:Documents and Settings:<*user*>:Application Data:WaveMetrics:Igor Pro 6:

where <*user*> is the name of the current user. The preferences directory may be hidden by some operating systems.

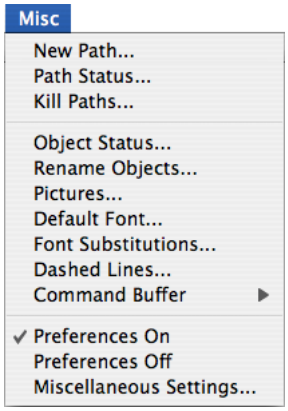
You can find the operating-system-defined location for preferences by executing this command:

```
Print SpecialDirPath("Preferences", 0, 0, 0)
```

Deleting the preferences directory effectively reverts all preferences to factory defaults. You should use the Capture Prefs dialogs, described in **Captured Preferences** operation on page III-431, to revert preferences more selectively.

Other information is stored in this directory, such as the screen position of dialogs, a few dialog settings, colors recently selected in the color palette, window stacking and tiling information, page setups, font substitution settings, and dashed line settings.

How to Use Preferences



Preferences are always on when Igor starts up. You can turn preferences off by choosing Preferences Off in the Misc menu.

You can also turn preferences on and off with the **Preferences** operation (see page V-497).

Preferences are set by Capture Preferences dialogs, the Tile or Stack Windows dialog, and some dialogs such as the Dashed Lines dialog.

There is just one set of preferences for all experiments. This means that preferences set while running one experiment will be in effect when you run the next experiment. This is handy because you only need to specify your preferences once.

In general, *preferences are applied only when something new is created* such as a new graph, a new trace in a graph, a new notebook, a new column in a table, and then only if preferences are on.

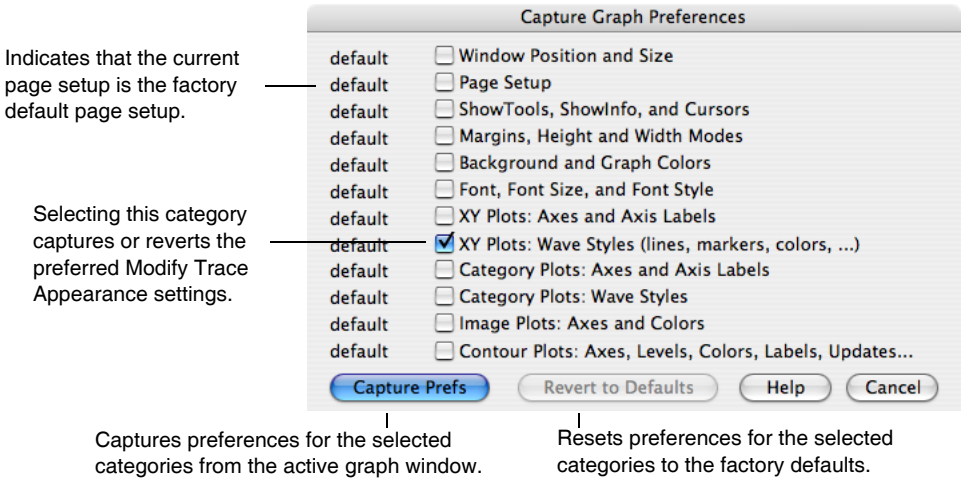
Preferences are normally in effect only for *manual* ("point-and-click") operation, not for user-programmed operations in Igor procedures. See **When Preferences Are Applied** on page III-432.

Captured Preferences

You set most Preference values by capturing the current settings of the active window with the Capture Prefs item in the menu for that window. (The dialog to capture the Command Window preferences is found in the Command/History Settings submenu of the Misc menu.) The dialogs are described in more detail in the chapter that discusses each type of window. For instance, see **Graph Preferences** on page II-298.

As an example, suppose you want your graphs to always draw appended waves with a one-half point blue line, rather than the factory default one point red line. You can set your preference by creating a graph with one wave displayed using a one-half point blue line, and then "capturing" that preferred setting with the Capture Graph Preferences dialog.

Choose Capture Graph Prefs from the Graph menu:



Save the wave style preference by selecting the XY Plots: Wave Styles category and clicking Capture Prefs. From now on, when you append a wave to an existing graph, or create a new graph containing a trace, the trace will be displayed with your preferred one-half point blue line.

Most capture preferences dialogs are like this Capture Graph Preferences dialog; they have various categories with checkboxes. Selecting a category means that you wish to change the preferences for that category. You may either capture the current settings for that category by clicking the Capture Prefs button, or you may revert the preferences for that category by clicking the Revert to Defaults button. If a category has been reverted to its default setting, "default" is indicated to the left of the checkbox. If "default" is not present, this means that the category settings have previously been captured.

Current Captured Preference Values

The Capture Preferences dialogs do not show the current *values* for the settings in the categories. The only way to discover what captured values are is to create a new window of that type and examine the settings with dialogs or readback functions. Any value not set to the factory default value must have been set by the preferences. (To check the factory default values, create a new window with Preferences off.)

For example, to determine what the values captured by the Capture Graph Preferences dialog are:

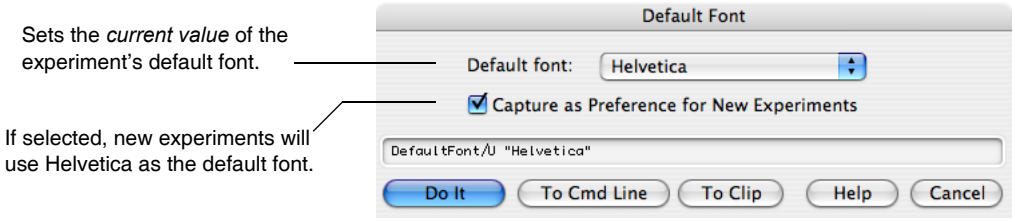
- Turn Preferences on (in the Misc menu).
- Create a new graph.
- Append some waves.

- Create a graph recreation macro using the Window Control dialog, and examine the values assigned by the resulting macro (this works well because graph recreation macros generate commands only to change values from their factory defaults).
- Or, use the Modify Waves Appearance dialog to observe the settings for the displayed waves. These are the captured user-preferred values.

Capturing Other Settings

In addition to the preferences captured by the various Capture Prefs dialogs, Igor remembers a number of other settings. Many of these can be set using the Miscellaneous Settings dialog, described in **Miscellaneous Settings** on page III-411.

Some settings are captured in dialogs whose main purpose is to change the *current* value of some settings. Such a dialog is the Default Font dialog:



It can capture the preferred Default Font with or without changing the current value. The Dashed Lines and Tile or Stack Windows dialogs also work this way.

The positions of dialogs on your desktop are always remembered in the preferences file. On Macintosh only, Recent Colors are remembered if the appropriate checkbox in the Misc Settings dialog is selected.

On both Macintosh and Windows, the font sizes you add to the Text Size menu are automatically remembered, as well as fonts substitution settings and various other dialog settings.

When Preferences Are Applied

In general, *preferences are applied only when something new is created* such as a new graph, a new wave in a graph, a new notebook, a new column in a table, and then only if preferences are on. In some cases, preferences affect what happens when you create a new experiment.

Igor has, in effect, two independent settings for whether preferences are on or off. The Preferences On and Preferences Off items in the Misc menu control the setting for *manual* ("point-and-click") operations, and is initially (and normally) set to "on". Another setting is used when a procedure (macro or function) is executing, and is normally set to "off".

We usually don't want preferences to affect the behavior of procedures. If we allowed preferences to take effect during procedure execution, a change in preferences could change the effect of a procedure, making it unpredictable. For more information, see **Procedures and Preferences** on page IV-178.

Volume IV Programming

Table of Contents

Chapter
IV-1

Working with Commands

Overview

Igor's user interface is unique in providing both a graphical user interface and a parallel path using command line operations. Although you can get by ignoring the command line, if you do so you will miss out on much of Igor's power and flexibility.

Even if you are a casual user you should learn at least the basics of assignment statements. Most users should read this entire chapter — especially those who expect to program Igor.

You can execute commands by typing them into the command line and pressing Return or Enter. You can also incorporate commands into procedures (functions and macros) that you write in the Procedure window. You can execute a procedure by typing its name in the command line or by choosing its item from the Macros menu. You can also use a notebook for entering commands. See **Notebooks as Worksheets** on page III-5.

You can type commands from scratch but often you will let Igor dialogs formulate and execute commands. You can view a record of what you have done in the history area of the command window and you can easily reenter, edit and reexecute commands stored there. See **Command Window Shortcuts** on page II-25 for details.

Multiple Commands

You can place multiple commands on one line if you separate them with semicolons. For example:

```
wave1= x; wave1= wave2/(wave1+1); Display wave1
```

You don't need a semicolon after the last command but it doesn't hurt.

Comments

Comments start with //, which end the executable part of a command line. The comment continues to the end of the line. There is no way to insert a comment in the middle of a command line.

Maximum Length of a Command

The total length of the command line can not exceed 400 characters.

There is no line continuation character in Igor. However, it is nearly always possible to break a single command up into multiple lines using intermediate variables. For example:

```
Variable a = sin(x-x0)/b + cos(y-y0)/c
```

can be rewritten as:

```
Variable t1 = sin(x-x0)/b
Variable t2 = cos(y-y0)/c
Variable a = t1 + t2
```

Parameters

Every place in a command where Igor expects a numeric parameter you can use a numeric expression. Similarly for a string parameter you can use a string expression. In an operation flag (e.g., /N=<number>), you must parenthesize expressions. See **Expressions as Parameters** on page IV-11 for details.

Liberal Object Names

In general, object names in Igor are limited to a restricted set of characters. Only letters, digits and the underscore character are allowed. Such names are called "standard names". This restriction is necessary to identify where the name ends when you use it in a command.

For waves and data folders only, you can also use "liberal" names. Liberal names can include almost any character, including spaces and dots (see **Liberal Object Names** on page III-415 for details). However, to define where a liberal name ends, you must quote them using single quotes.

In the following example, the wave names are liberal because they include spaces and therefore they must be quoted:

```
'wave 1' = 'wave 2'     // Right
wave 1 = wave 2         // Wrong - liberal names must be quoted
```

Note: Providing for liberal names requires extra effort and testing by Igor programmers (see **Programming with Liberal Names** on page IV-147) so you may occasionally experience problems using liberal names with user-defined procedures.

Data Folders

Data folders provide a way to keep separate sets of data from interfering with each other. You can examine and create data folders using the Data Browser (Data menu). There is always a root data folder and this is the only data folder that many users will ever need. Advanced users may want to create additional data folders to organize their data.

You can refer to waves and variables either in the current data folder, in a specific data folder or in a data folder whose location is relative to the current data folder:

```
// wave1 is in the current data folder
wave1 = <expression>

// wave1 is in a specific data folder
root:'Background Curves':wave1 = <expression>

// wave1 is in a data folder inside the current data folder
:'Background Curves':wave1 = <expression>
```

In the first example, we use an object name by itself (wave1) and Igor looks for the object in the current data folder.

In the second example, we use a full data folder path (root:'Background Curves':) plus an object name. Igor looks for the object in the specified data folder.

In the third example, we use a relative data folder path (:'Background Curves':) plus an object name. Igor looks in the current data folder for a subdata folder named Background Curves and looks for the object within that data folder.

Important: The right-hand side of an assignment statement (described under **Assignment Statements** on page IV-4) is evaluated in the context of the data folder containing the destination object. For example:

```
root:'Background Curves':wave1 = wave2 + var1
```

For this to work, wave2 and var1 must be in the Background Curves data folder.

Examples in the rest of this chapter use object names alone and thus reference data in the current data folder. For more on data folders, see Chapter II-8, **Data Folders**.

Types of Commands

There are three fundamentally different types of commands that you can execute from the command line:

- assignment statements
- operation commands
- user-defined procedure commands

Here are examples of each:

```
wave1= sin(2*pi*freq*x)       // assignment statement

Display wave1,wave2 vs xwave  // operation command

MyFunction(1.2,"hello")       // user-defined procedure command
```

As Igor executes commands you have entered, it must determine which of the three basic types of commands you have typed. If a command starts with a wave or variable name then Igor assumes it is an assignment statement. If a command starts with the name of a built-in or external operation then the command is treated as an operation. If a command begins with the name of a user-defined macro, user-defined function or external function then the command is treated accordingly. Each of these types is discussed in greater detail under **Assignment Statements** on page IV-4, **Operation Commands** on page IV-9, and **User-Defined Procedure Commands** on page IV-9.

Note that built-in functions can only appear in the right-hand side of an assignment statement, or as a parameter to an operation or function. Thus, the command:

```
sin(x)
```

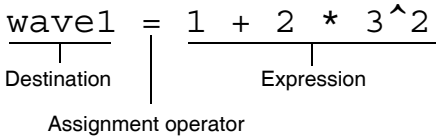
is not allowed and you will see the error, "Expected wave name, variable name, or operation." On the other hand, these commands are allowed:

```
print sin(1.567)              // sin is parameter of print command
wave1 = 5*sin(x)              // sin in right side of assigment
```

If, perhaps due to a misspelling, Igor can not determine what you want to do, it will put up an error dialog and the error will be highlighted in the command line.

Assignment Statements

Assignment statement commands start with a wave or variable name. The command assigns a value to all or part of the named object. An assignment statement consists of three parts: a destination, an assignment operator, and an expression. For example:



This assigns 19 to every point in wave1.

The spaces in the above example are not required. You could write:

```
wave1=1+2*3^2
```

See **Waveform Arithmetic and Assignments** on page II-93 for details on wave assignment statements.

In the following examples, str1 is a string variable, created by the String operation, var1 is a numeric variable, created by the Variable operation, and wave1 is a wave, created by the Make operation.

```
str1 = "Today is " + date()                 // string assignment

str1 += ", and the time is " + time()       // string concatenation

var1 = strlen(str1)                         // variable assignment

var1 = pnt2x(wave1,numpnts(wave1)/2)        // variable assignment

wave1 = 1.2*exp(-0.2*(x-var1)^2)            // wave assignment

wave1[3] = 5                                // wave assignment

wave1[0,;3]= wave2[p/3] *exp(-0.2*x)        // wave assignment
```

These all operate on objects in the current data folder. To operate on an object in another data folder, you need to use a data folder path:

```
root:'run 1':wave1[3] = 5          // wave assignment
```

See Chapter II-8, **Data Folders**, for further details.

If you use liberal wave names (see **Object Names** on page III-415), you must use quotes:

```
'wave 1' = 'wave 2'         // Right
wave 1 = wave 2             // Wrong
```

Assignment Operators

The assignment operator determines the way in which the expression is combined with the destination. Igor supports the following assignment operators:

Operator	Assignment Action
=	Destination contents are set to the value of the expression.
+=	Expression is added to the destination.
-=	Expression is subtracted from the destination.
*=	Destination is multiplied by the expression.
/=	Destination is divided by the expression.
:=	Destination is dynamically updated to the value of the expression whenever the value of any part of the expression changes. The := operator is said to establish a "dependency" of the destination on the expression.

For example:

```
wave1 = 10
```

sets each Y value of the wave1 equal to 10, whereas:

```
wave1 += 10
```

adds 10 to each Y value of wave1. This is equivalent to:

```
wave1 = wave1 + 10
```

The assignment operators =, := and += work with string assignment statements but -=, *= and /= do not. For example:

```
String str1; str1 = "Today is "; str1 += date(); Print str1
```

prints something like "Today is Fri, Mar 31, 2000".

For more information on the := operator, see Chapter IV-9, **Dependencies**.

Operators

Here is a complete list of the operators that Igor supports in the expression part of an assignment statement in order of precedence:

Operator	Effect
^	Exponentiation
- ! ~	Negation, logical complement, bitwise complement
* /	Multiplication, division
+ -	Addition or string concatenation, subtraction
== != > < >= <=	Comparison operators
& | %^	Bitwise AND, bitwise OR, bitwise XOR
&& || ? :	Logical AND, logical OR, conditional operator
$	Substitute following string expression as name

Unary negation changes the sign of its operand. Logical complementation changes nonzero operands to zero and zero operands to 1. Bitwise complementation converts its operand to an unsigned integer by truncation and then changes it to its binary complement.

Exponentiation raises its left-hand operand to a power specified by its right-hand operand. That is, 3^2 is written as 3^2. In an expression a^b, if the result is assigned to a real variable or wave, then a must not be negative if b is not an integer. If the result is used in a complex expression, any combination of negative a, fractional b or complex a or b is allowed.

If the exponent is an integer Igor evaluates the expression using only multiplication. There is no need to write a^2 as a*a to get efficient evaluation — Igor does the equivalent automatically. If, on the other hand, the exponent is not an integer then the evaluation is performed using logarithms, hence the restriction on negative a in a real expression.

Logical OR (||) and logical AND (&&) determine the truth or falseness of pairs of expressions. The AND operation returns true only when both expressions are true; OR will return true if *either* is true. As in C, true is *any* nonzero value, and false is zero. The operations are undefined for NaNs. These operators are not available in complex expressions.

The logical operators are evaluated from left to right, and an operand will not be evaluated if it is not necessary. For the example:

```
if(MyFunc1() && MyFunc2())
```

when MyFunc1() returns false (zero), then MyFunc2() will not be evaluated because the entire expression is already false. This can produce unexpected consequences when the right-hand expression has side effects, such as creating waves or setting global values.

Bitwise AND (&), OR (|), and XOR (%^) convert their operands to an unsigned integer by truncation and then return their binary AND, OR or exclusive OR.

The conditional operator (? :) is a shorthand form of an if-else-endif expression. In the statement:

```
<expression> ? <TRUE> : <FALSE>
```

the first operand, <expression>, is the test condition; if it is nonzero then Igor evaluates the <TRUE> operand; otherwise <FALSE> is evaluated. Only one operand is evaluated according to the test condition. This is the same as if you had written:

```
if( <expression> )
   <TRUE>
else
   <FALSE>
endif
```

The ":" character in the conditional operator must always be separated from the two adjacent operands with a space. If you omit either space, you will get an error ("No such data folder") because the expression can also be interpreted as part of a data folder path. To be safe, always separate the operands from the operator symbols with a space.

The operands must be numeric; for strings, use the **SelectString** function. When using complex expressions with the conditional operator, only the real portion is used when the operator evaluates the expression.

The conditional operator can easily cause confusion, so you should exercise caution when using it. For example, it is unclear from simple inspection what Igor may return for

```
1 ? 2 : 3 ? 4 : 5
```

(4 in this case), whereas

```
1 ? 2 : (3 ? 4 : 5)
```

will return 2. Always use parentheses to remove any ambiguity.

The comparison operators return 1 if the result of the comparison is true or 0 if it is false. For example, the == operator returns 1 if its operands are equal or 0 if they are not equal. The != operator returns the opposite. Because comparison operators return the values 1 or 0 they can be used in interesting ways. The assignment:

```
wave1 = sin(x)*(x<=50) + cos(x)*(x>50)
```

sets wave1 so that it is a sine wave below x=50 and a cosine wave above x=50. See also **Example: Comparison Operators and Wave Synthesis** on page II-98.

Note that the double equal sign, ==, is used to mean equality while the single equal sign, =, is used to indicate assignment.

Because of roundoff error, using == to test two numbers for equality may give incorrect results. It is safer to use <= and >= to see if a number falls in a narrow range. For example, imagine that you want to compare a variable to see if it is equal to one-third. The expression:

```
(v1 == 1/3)
```

is subject to failure because of roundoff. It is safer to use something like

```
((v1 > .33332) && (v1 < .33334))
```

If the numbers are integers then the use of == is safe because integers smaller than 2^{53} (approximately 10^{16}) are represented precisely in double-precision floating point numbers.

The previous discussion on operators has assumed numeric operands. The + operator is the only one that works with both numeric *and* string operands. For example, if str1 is a string variable then the assignment statement

```
str1 = "Today is " + "a nice day"
```

assigns the value "Today is a nice day" to str1. The other string operator, $ is discussed in **String Substitution Using $** on page IV-15.

Unless specified otherwise by parentheses, unary negation or complementation are carried out first followed by exponentiation then by multiplication or division followed by addition or subtraction then by comparison operators. The wave assignment:

```
wave1 = ((1 + 2) * 3) ^ 2
```

assigns the value 81 to every point in wave1, but

```
wave1 = 1 + 2 * 3 ^ 2
```

assigns the value 19.

Note: `-a^b` is an exception to this rule and is evaluated as `-(a^b)`.

The precedence of string substitution, substrings, and wave indexing is somewhat complex. When in doubt, use parenthesis to enforce the precedence you want.

Obsolete Operators

As of Igor Pro 4.0, the old bitwise complement (%~), bitwise AND (%&), and bitwise OR (%|) operators have been replaced by new versions that omit the % character from the operator. These old bitwise operators can still be used interchangeably with the new versions.

Operands

In addition to literal numbers like 3.141 or 27, operators can operate on variables and function values. In the assignment statement:

```
var1 = log(3.7) + var2
```

the operator + operates on the function value returned by log and on the variable var2. Functions and function values are discussed later in this chapter.

Numeric Type

In Igor, each numeric destination object (wave or variable) has its own numeric type. The numeric type consists of the numeric precision (e.g., double precision floating point) and the number type (real or complex). Waves can be single or double precision floating point or various sizes of integer but variables are always double precision floating point.

The numeric precision of the destination does not affect the calculations. With the exception of a few operations that are done in place such as the FFT, all calculations are done in double precision.

Although waves can have integer numeric types, wave expressions are always evaluated in double precision floating point. The floating point values are converted to integers by rounding as the final step before storing the value in the wave. If the value to be stored exceeds the range of values that the given integer type can represent, the results are undefined.

The number type of the destination determines the initial number type (real or complex) of the assignment expression. This is important because Igor can not deal with "surprise" or "runtime" changes in number type. An example would be taking the square root of a negative number requiring that all following arithmetic be done using complex numbers.

Here are some examples:

```
Variable a, b, c, var1
Variable/C cvar1
Make wave1

var1= a*b
cvar1= c*cmplx(a+1,b-1)
wave1= var1 + real(cvar1)
```

The first expression is evaluated using the real number type. The second expression contains a mixture of two types. The multiplication of c with the result of the cmplx function is evaluated as complex while the arguments to the cmplx function are evaluated as real. The third example is evaluated as real except for the argument to the real function which is evaluated as complex.

Constant Type

You can define named numeric and string constants in Igor procedure files and use them in the body of user-defined functions.

Constants are defined in procedure files using following syntax:

```
Constant <name1> = <literal number> [, <name2> = <literal number>]

StrConstant <name1> = <literal string> [, <name2> = <literal string>]
```

You can use the static prefix to limit the scope to the given source file. For debugging, you can use the Override keyword as with functions.

These declarations can be used in the following ways:

```
constant kFoo=1,kBar=2
strconstant ksFoo="hello",ksBar="there"

static constant kFoo=1,kBar=2
static strconstant ksFoo="hello",ksBar="there"

override constant kFoo=1,kBar=2
override strconstant ksFoo="hello",ksBar="there"
```

Programmers may find that using the "k" and "ks" prefixes will make their code easier to read.

Names for numeric and string constants can conflict with all other names. Duplicate constants of a given type are not allowed (except static in different files and when used with Override). The only true conflict is

with variable names and with certain built-in functions that do not take parameters such as pi. Variable names override constants, but constants override functions such as pi.

Dependency Assignment Statements

You can set up global variables and waves so that they automatically recalculate their contents when other global objects change. See Chapter IV-9, **Dependencies**, for details.

Operation Commands

An operation is a built-in or external routine that performs an action but, unlike a function, does not directly return a value. Here are some examples:

```
Make/N=512 wave1
Display wave1
Smooth 5, wave1
```

Operation commands perform the majority of the work in Igor and are automatically generated and executed as you work with Igor using dialogs.

You can use these dialogs to experiment with operations of interest to you. As you click in a dialog, Igor composes a command. This provides a handy way for you to check the syntax of the operation or to generate a command for use in a user-defined procedure. See Chapter V-1, **Igor Reference**, for a complete list of all built-in operations. Another way to learn their syntax is to use the Igor Help Browser's Command Help tab. See **Command Help Tab** on page II-6.

The syntax of operation commands is highly variable but in general consists of the operation name, followed by a list of options (e.g., `/N=512`), followed by a parameter list. The operation name specifies the main action of the operation and determines the syntax of the rest of the command. The list of options specifies variations on the default behavior of the operation. If the default behavior of the operation is satisfactory then no options are required. The parameter list identifies the objects on which the operation is to operate. Some commands take no parameters. For example, in the command:

```
Make/D/N=512 wave1, wave2, wave3
```

the operation name is "`Make`". The list of options is "`/D/N=512`". The parameter list is "`wave1, wave2, wave3`". An option such as "`/D`" or "`/N=512`" is sometimes termed a "flag".

You can use numeric expressions in the parameter list of an operation where Igor expects a numeric parameter, but in an operation option you need to parenthesize the expression. For example:

```
Variable val = 1.0
Make/N=(val) wave0, wave1
Make/N=(numpnts(wave0)) wave2
```

The most common types of parameters are literal numbers or numeric expressions, literal strings or string expressions, names, and waves. In the example above, wave1 is a name parameter when passed to the Make operation. It is a wave parameter when passed to the Display and Smooth operations. A name parameter can refer to a wave that may or may not already exist whereas a wave parameter must refer to an existing wave.

See **Parameter Lists** on page IV-11 for general information that applies to all commands.

User-Defined Procedure Commands

For details on creating your own procedures, refer to chapters starting with IV-2.

User-defined procedure commands start with a macro, user function, or external function name followed by a list of parameters in parentheses. Here are a few examples:

```
MyFunction1(5.6, wave0, "igneous")
MyMacro1(1.2, 1/sqrt(ln(2)), "wave0")
MyMacro1(1.2 ,  ,  )
MyMacro1( )
```

Macro and Function Parameters

As illustrated by the last two examples, you can invoke macros (but not functions) with one or more of the input parameters missing. When you do this, Igor puts up a dialog to allow you to enter the missing parameters. When you run a macro by choosing it from the Macros menu, Igor simply executes the macro with all of the parameters missing as in the last example. After you enter values in the dialog, the macro is executed with those parameters and the now-complete macro command is placed in the history. You can then fetch the command from the history, modify a parameter and then reexecute the command without having to go through the dialog.

You can add similar capabilities to user-defined functions using the **Prompt** (see page V-508) and **DoPrompt** (see page V-121) keywords. You can also use the **PauseForUser** (see page V-476) operation in a function to provide a more sophisticated way to get user input. WaveMetrics encourages programmers to use user-defined functions instead of macros.

There is an additional difference between functions and macros that you should be aware of. Functions can accept numeric, string and wave reference parameters. Macros can accept numeric and string parameters but can not accept literal wave names. For this reason, macros are written to accept waves in the form of strings containing the wave names. It's the difference between wave0 and "wave0" in the first two examples above.

When you are using a package of procedures written by someone else you may need to determine what parameters a particular macro or function requires. If this is not documented you can easily inspect the source code by opening the procedure window and choosing the desired procedure from the pop-up menu at the bottom of the window.

Function Commands

A function is a routine that directly returns a numeric or string value. There are three classes of functions available to Igor users:

- Built-in
- External (XFUNCs)
- User-defined

Built-in numeric functions enjoy one advantage over external or user-defined functions: a few come in real and complex number types and Igor automatically picks the appropriate version depending on the current number type in an expression. External and user-defined functions must have different names when different types are needed. Generally, only real user and external functions need be provided.

For example, in the wave assignment:

```
wave1 = enoise(1)
```

if wave1 is real then the function enoise returns a real value. If wave1 is complex then enoise returns a complex value.

You can use a function as a parameter to another function, to an operation, to a macro or in an arithmetic or string expression so long as the data type returned by the function makes sense in the context in which you use it.

User-defined and external functions can also be used as commands by themselves. Use this to write a user function that has some purpose other than calculating a numeric value, such as displaying a graph or making new waves. Built-in functions cannot be used this way. For instance:

```
MyDisplayFunction(wave0)
```

External and user-defined functions can be used just like built-in functions. In addition, numeric functions can be used in curve fitting. See Chapter IV-3, **User-Defined Functions** and **Fitting to a User-Defined Function** on page III-171.

Most functions consist of a function name followed by a left parenthesis followed by a parameter list and followed by a right parenthesis. In the wave assignment shown at the beginning of this section, the function

name is enoise. The parameter is 1. The parameter is enclosed by parentheses. In this example, the result from the function is assigned to a wave. It can also be assigned to a variable or printed:

```
K0 = enoise(1)
Print enoise(1)
```

User and external functions (but not built-in functions) can be executed on the command line or in other functions or macros without having to assign or print the result. This is useful when the point of the function is not its explicit result but rather its side effects.

Nearly all functions require parentheses even if the parameter list is empty. For example the function date() has no parameters but requires parentheses anyway. There are a few exceptions. For example the function Pi returns π and is used with no parentheses or parameters.

Igor's built-in functions are described in detail in Chapter V-1, **Igor Reference**.

Parameter Lists

Parameter lists are used for operations, functions, and macros and consist of one or more numbers, strings, keywords or names of Igor objects. The parameters in a parameter list must be separated with commas.

Expressions as Parameters

In an operation, function, or macro parameter list which has a numeric parameter you can always use a numeric expression instead of a literal number. A numeric expression is a legal combination of literal numbers, numeric variables, numeric functions, and numeric operators. For example, consider the command

```
SetScale x, 0, 6.283185, "v", wave1
```

which sets the X scaling for wave1. You could also write this as

```
SetScale x, 0, 2*PI, "v", wave1
```

Parentheses Required for /N=(<expression>)

Many operations accept flags of the form "/A=n" where A is some letter and n is some number. You can use a numeric expression for n but you must parenthesize the expression.

For example, both:

```
Make/N=512 wave1
Make/N=(2^9) wave1
```

are legal but this isn't:

```
Make/N=2^9 wave1
```

A variable name is a form of numeric expression. Thus, assuming v1 is the name of a variable:

```
Make/N=(v1)
```

is legal, but

```
Make/N=v1
```

is not. This parenthesization is required only when you use a numeric expression in an operation flag.

String Expressions

A string expression can be used where Igor expects a string parameter. A string expression is a legal combination of literal strings, string variables, string functions and the string operator + which concatenates strings.

Setting Bit Parameters

A number of commands require that you specify a bit value to set certain parameters. In these instances you set a certain bit *number* by using a specific bit *value* in the command. The bit value is 2^n, where *n* is the bit number. So, to set bit 0 use a bit value of 1, to set bit 1 use a bit value of 2, etc.

For the example of the TraceNameList function the last parameter is a bit setting. To select normal traces you must set bit 0:

```
TraceNameList("",";",1)
```

and to select contour traces set bit 1:

```
TraceNameList("",";",2)
```

Most importantly, you can set multiple bits at one time by adding the bit values together. Thus, for TraceNameList you can select both normal (bit 0) and contour (bit 1) traces by using:

```
TraceNameList("",";",3)
```

See also **Using Bitwise Operators** on page IV-33.

Strings

Igor has a rich repertoire of string handling capabilities. See **Strings** on page V-10 for a complete list of Igor string functions. Many of the techniques described in this section will be of interest only to programmers.

Many Igor operations require *string* parameters. For example, to label a graph axis, you can use the Label operation:

```
Label left, "Volts"
```

Other Igor operations, such as Make, require *names* as parameters:

```
Make wave1
```

Using the string substitution technique, described in **String Substitution Using $** on page IV-15, you can generate a name parameter by making a string containing the name and using the $ operator:

```
String stringContainingName = "wave1"
Make $stringContainingName
```

String Expressions

Wherever Igor requires a string parameter, you can use a string expression. A string expression can be:

- A literal string (`"Today is"`)
- The output of a string function (`date()`)
- An element of a text wave (`textWave0[3]`)
- Some combination of string expressions (`"Today is" + date()`)

In addition, you can derive a string expression by indexing into another string expression. For example,

```
Print ("Today is" + date())[0,4]
```

prints “Today”.

A string variable can store the result of a string expression. For example:

```
String str1 = "Today is" + date()
```

A string variable can also be part of a string expression, as in:

```
Print "Hello. " + str1
```

Strings in Text Waves

A text wave contains an array of text strings. Each element of the wave can be treated using all of the available string manipulation techniques. In addition, text waves are commonly used to create category axes in bar charts. See **Text Waves** on page II-103 for further information.

There is a potential ambiguity with string indexing when the string is stored in a text wave. See **String Indexing** on page IV-13.

String Properties

Strings in Igor can be of unlimited length. There are no restrictions on the characters that can be stored in a string except for the null character (ASCII code 0). Storing a null in a string causes problems because the C functions that Igor uses internally to handle strings take null as an end-of-string flag.

Some Igor functions can take an empty string (`""`, no space between the quotation marks) as a parameter.

Escape Characters in Strings

Igor treats the backslash character in a special way when reading literal (quoted) strings in a command line. The backslash is used to define something called an "escape sequence". This just means that the backslash plus the next character or next few characters are treated like a different character — one you could not otherwise include in a quoted string. The escape sequences are:

`\t`	Tab character
`\r`	Return character
`\n`	Linefeed character
`\'`	The ' character
`\"`	The " character
`\\`	The \ character
`\ddd`	An arbitrary ASCII code (ddd is a 3 digit octal number)

For example, if you have a string variable called "fileName", you could print it in the history area using:

```
fileName = "Test"
Printf "The file name is \"%s\"\r", fileName
```

which prints

```
The file name is "Test"
```

In the Printf command line, `\"` embeds a double-quote character in the format string. If you omitted the backslash, the " would end the format string. The `\r` specifies that you want a carriage return in the format string.

String Indexing

Indexing can extract a part of a string. This is done using a string expression followed by one or two numbers in brackets. The numbers are character positions. Zero is the character position of the first character; n-1 is the character position of the last character of an n character expression. For example, assume we create a string variable called s1 and assign a value to it as follows:

```
String s1="hello there"
```

h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10

Then,

`Print s1[0,4]` prints `hello`

```
Print s1[0,0]                   prints   h
Print s1[0]                     prints   h
Print s1[1]+s1[2]+s1[3]         prints   ell
Print (s1+" jack")[6,15]        prints   there jack
```

A string indexed with one number, such as s1[p], is a string with one character in it if p is in range (i.e. 0 ≤ p ≤ n-1). s1[p] is a string with no characters in it if p is not in range. For example:

```
Print s1[0]                     prints   h
Print s1[-1]                    prints   (nothing)
Print s1[10]                    prints   e
Print s1[11]                    prints   (nothing)
```

A string indexed with two numbers, such as s1[p1,p2], contains all of the characters from s1[p1] to s1[p2]. For example:

```
Print s1[0,10]                  prints   hello there
Print s1[-1,11]                 prints   hello there
Print s1[-2,-1]                 prints   (nothing)
Print s1[11,12]                 prints   (nothing)
Print s1[10,0]                  prints   (nothing)
```

Because the syntax for string indexing is identical to the syntax for wave indexing, you have to be careful when using text waves. For example:

```
Make/T textWave0 = {"Red", "Green", "Blue"}

Print textWave0[1]              prints   Green
Print textWave0[1][1]           prints   Green
Print textWave0[1][1][1]        prints   Green
Print textWave0[1][1][1][1]     prints   Green
Print textWave0[1][1][1][1][1]  prints   r
```

The first four examples print row 1 of column 0. Since waves may have up to four dimensions, the first four [1]'s act as dimension indices. The column, layer, and chunk indices were out of range and were clipped to a value of 0. Finally in the last example, we ran out of dimensions and got string indexing. **Warning**: Do not count on this behavior because future versions of Igor may support more than four dimensions.

The way to avoid the ambiguity between wave and string indexing is to use parentheses like so:

```
Print (textWave0[1])[1]         prints   r
```

String Assignment

You can assign values to string variables using string assignment. We have already seen the simplest case of this, assigning a literal string value to a string variable. You can also assign values to a subrange of a string variable, using string indexing. Once again, assume we create a string variable called s1 and assign a value to it as follows:

```
String s1="hello there"
```

Then,

```
s1[0,4]="hi";print s1           prints   hi there
s1[0,4]="greetings";print s1    prints   greetings there
s1[0,0]="j";print s1            prints   jello there
s1[0]="well ";print s1          prints   well hello there
s1[100000]=" jack";print s1     prints   hello there jack
s1[-100]="well ";print s1       prints   well hello there
```

When the s1[p1,p2]= syntax is used, the right-hand side of the string assignment *replaces* the subrange of the string variable identified by the left-hand side, after p1 and p2 are clipped to 0 to n.

When the s1[p]= syntax is used, the right-hand side of the string assignment *is inserted before* the character identified by p after p is limited to 0 to n.

The subrange assignment just described for string variables is not supported when a text wave is the destination. To assign a value to a range of a text wave element, you will need to create a temporary string variable. For example:

```
Make/O/T tw = {"Red", "Green", "Blue"}
String stmp= tw[1]
stmp[1,2]="XX"
tw[1]= stmp;

print tw[0],tw[1],tw[2]        prints    Red GXXen Blue
```

String Substitution Using $

Wherever Igor expects the literal *name* of an operand, such as the name of a wave, you can instead provide a string expression preceded by the $ character. The $ operator evaluates the string expression and returns the *value* as a *name*.

For example, the Make operation expects the name of the wave to be created. Assume we want to create a wave named wave0:

```
   Make wave0              // OK: wave0 is a literal name.

   Make $"wave0"           // OK: $"wave0" evaluates to wave0.

   String str = "wave0"
   Make str                // WRONG: This makes a wave named str.
   Make $str               // OK: $str evaluates to wave0.
```

$ is often used when you write a function which receives the name of a wave to be created as a parameter. Here is a trivial example:

```
Function MakeWave(w)
   String wName      // name of the wave

   Make $wName
End
```

We would invoke this function as follows:

```
MakeWave("wave0")
```

We use $ because we need a wave name but we have a string containing a wave name. If we omitted the $ and wrote:

```
   Make wName
```

Igor would make a wave whose name is wName, not on a wave whose name is wave0.

String substitution is capable of converting a string expression to a *single* name. It can not handle multiple names. For example, the following will *not* work:

```
String list = "wave0;wave1;wave2"
Display $list
```

See **Processing Lists of Waves** on page IV-174 for ways to accomplish this.

$ Precedence Issues In Commands

This section discusses issues that arise when using string substitution in assignment statements in the command line or in a macro. This is somewhat academic because modern Igor programming is done with user-defined functions. In user-defined functions, the ambiguity is removed through the use of "wave references" (described in **Accessing Waves in Functions** on page IV-65).

There is one case in which string substitution does not work as you might expect. Consider this example:

```
String str1 = "wave1"
wave2 = $str1 + 3
```

You might expect that this would cause Igor to set wave2 equal to the sum of wave1 and 3. Instead, it generates an "expected string expression" error. The reason is that Igor tries to concatenate str1 and 3 *before* doing the substitution implied by $. The + operator is also used to concatenate two string expressions, and it has higher precedence than the $ operator. Since str1 is a string but 3 is not, Igor cannot do the concatenation.

You can get around this problem by changing this wave assignment to one of the following:

```
wave2 = 3 + $str1
wave2 = ($str1) + 3
```

Both of these accomplish the desired effect of setting wave2 equal to the sum of wave1 and 3. Similarly,

```
wave2 = $str1 + $str2   // Igor sees "$(str1 + $str2)"
```

generates the same "expected string expression" error. The reason is that Igor is trying to concatenate str1 and $str2. $str2 is a name, not a string. The solution is:

```
wave2 = ($str1) + ($str2)  // sets wave2 to sum of two named waves
```

Another situation arises when using the $ operator and [. The [symbol can be used for either point indexing into a wave, or character indexing into a string. The commands

```
String wvName = "wave0"
$wvName[1,2] = wave1[p]    // sets two values in wave named "wave0"
```

are interpreted to mean that points 1 and 2 of wave0 are set values from wave1.

If you intended "$wvName[1,2] = wave1" to mean that a wave whose name comes from characters 1 and 2 of the wvName string ("av") has all of its values set from wave1, you must use parenthesis:

```
$(wvName[1,2]) = wave1     // sets all values of wave named "av"
```

String Utility Functions

WaveMetrics provides a number of handy utility functions for dealing with strings. To see a list of the built-in string functions, open the Igor Help Browser Command Help tab and then choose String from the pop-up menu of function categories. See also the string utility procedure files provided by WaveMetrics in the WaveMetrics Procedures:Utilities:String Utilities folder.

Special Cases

This section documents some techniques that were devised to handle certain specialized situations that arise with respect to Igor's command language.

Instance Notation

There is a problem that occurs when you have multiple instances of the same wave in a graph or multiple instances of the same object in a layout. For example, assume you want to graph yWave versus xWave0, xWave1, and xWave2. To do this, you need to execute:

```
Display yWave vs xWave0
AppendToGraph yWave vs xWave1
AppendToGraph yWave vs xWave2
```

The result is a graph in which yWave occurs three times. Now, if you try to remove or modify yWave using:

```
RemoveFromGraph yWave
```

or

```
ModifyGraph lsize(yWave)=2
```

Igor will always remove or modify the first instance of yWave.

Instance notation provides a way for you to specify a particular instance of a particular wave. In our example, the command

```
RemoveFromGraph yWave#2
```

will remove instance number 2 of yWave and

```
ModifyGraph lsize(yWave#2)=2
```

will modify instance number 2 of yWave. Instance numbers start from zero so "yWave" is equivalent to "yWave#0". Instance number 2 is the instance of yWave plotted versus xWave2 in our example.

Where necessary to avoid ambiguity, Igor operation dialogs (e.g., Modify Trace Appearance) automatically use instance notation. Operations that accept trace names (e.g., ModifyGraph) or layout object names (e.g., ModifyObject) accept instance notation.

A graph can also display multiple waves with the same name if the waves reside in different datafolders. Instance notation applies to the this case also.

Instance Notation and $

The $ operator can be used with instance notation. The # symbol may be either inside the string operand or may be outside. For example `$"wave0#1"` or `$"wave0"#1`. However, because the # symbol may be inside the string, the string must be parsed by Igor. Consequently, unlike other uses of $, the wave name portion must be surrounded by single quotes if liberal names are used. For example, suppose you have a wave with the liberal name of `'ww#1'` plotted twice. The first instance would be `$"'ww#1'"` and the second `$"'ww#1'#1"` whereas `$"ww#1"` would reference the second instance of the wave ww.

Object Indexing

The ModifyGraph, ModifyTable and ModifyLayout operations, used to modify graphs, tables and page layouts, each support another method of identifying the object to modify. This method, object indexing, is used to generate style macros (see **Graph Style Macros** on page II-300). You may also find it handy in other situations.

Normally, you need to know the name of the object that you want to modify. For example, assume that we have a graph with three traces in it and we want to set the traces' markers from a procedure. We can write:

```
ModifyGraph marker(wave0)=1, marker(wave1)=2, marker(wave2)=3
```

Because it uses the names of particular traces, this command is specific to a particular graph. What do we do if we want to write a command that will set the markers of three traces in *any* graph, regardless of the names of the traces? This is where object indexing comes in.

Using object indexing, we can write:

```
ModifyGraph marker[0]=1, marker[1]=2, marker[2]=3
```

This command sets the markers for the first three traces in a graph, no matter what their names are.

Indexes start from zero. For graphs, the object index refers to traces starting from the first trace placed in the graph. For tables the index refers to columns from left to right. For page layouts, the index refers to objects starting from the first object placed in the layout.

/Z Flag

The ModifyGraph marker command above works fine if you know that there *are* three waves in the graph. It will, however, generate an error if you use it on a graph with fewer than 3 waves. The ModifyGraph operation supports a flag that can be used to handle this:

```
ModifyGraph/Z marker[0]=1, marker[1]=2, marker[2]=3
```

The /Z flag ignores errors if the command tries to modify an object that doesn't exist. The /Z flag works with the SetAxis and Label operations as well as with the ModifyGraph, ModifyTable and ModifyLayout oper-

ations. Like object indexing, the /Z flag is primarily of use in creating style macros, which is done automatically, but it may come in handy for other uses.

Chapter
IV-2

Programming Overview

Overview

You can perform powerful data manipulation and analysis interactively using Igor's dialogs and the command line. However, if you want to automate common tasks or create custom numerical operations then you need to use procedures. You can write them yourself, use procedures supplied by WaveMetrics, or find someone else who has written procedures you can use. Even if you don't write procedures from scratch, it is useful to know enough about Igor programming to be able to understand code written by others.

Programming in Igor entails creating procedures by entering text in a procedure window. After entering a procedure, you can execute it via the command line, by choosing an item from a menu, or using a button in a control panel.

The bulk of the text in a procedure window falls into one of the following categories:

- Pragmas, which send instructions from the programmer to the Igor compiler
- Include statements, which open other procedure files
- Constants, which define symbols used in functions
- Structure definitions, which can be used in functions
- Proc Pictures, which define images used in control panels, graphs, and layouts
- Menu definitions, which add menu items or entire menus to Igor
- Functions — compiled code which is used for nearly all Igor programming
- Macros — interpreted code which, for the most part, is obsolete

Functions are written in Igor's programming language. Like conventional procedural languages such as C or Pascal, Igor's language includes:

- Data storage elements (variables, strings, waves)
- Assignment statements
- Flow control (conditionals and loops)
- Calls to built-in and external operations and functions
- Ability to define and call subroutines

Igor programming is easier than conventional programming because it is much more interactive — you can write a routine and test it right away. It is designed for interactive use within Igor rather than for creating stand-alone programs.

Organizing Procedures

Procedures can be stored in the built-in Procedure window or in separate auxiliary procedure files. Chapter III-13, **Procedure Windows**, explains how to edit the Procedure window and how to create auxiliary procedure files.

At first you will find it convenient to do all of your Igor programming in the built-in Procedure window. In the long run, however, it will be useful to organize your procedures into categories so that you can easily find and access general-purpose procedures and keep them separate from special-case procedures.

This table shows how we categorize procedures and how we store and access the different categories.

Category	What	Where	How
Experiment Procedures	These are specific to a single Igor experiment. They include procedures you write as well as window recreation macros created automatically when you close a graph, table, layout, control panel, or XOP target window (e.g., surface plot).	Usually experiment procedures are stored in the built-in Procedure window. You can optionally create additional procedure windows in a particular experiment but this is usually not needed.	You create an experiment procedure by typing in the built-in Procedure window.
Utility Procedures	These are general-purpose and potentially useful for any Igor experiment. WaveMetrics supplies utility procedures in the WaveMetrics Procedures folder. You can also write your own procedures or get them from colleagues.	WaveMetrics-supplied utility procedure files are stored in the WaveMetrics Procedures folder. Utility procedure files that you or other Igor users create should be stored in your own folder, in the Igor Pro User Files folder (see **Igor Pro User Files** on page II-46 for details) or at another location of your choosing. Place an alias or shortcut for your folder in "Igor Pro User Files/User Procedures".	Use an include statement to use a WaveMetrics or user utility procedure file. Include statements are described in **The Include Statement** on page IV-145.
Global Procedures	These are procedures that you want to be available from all experiments.	Store your global procedure files in "Igor Pro User Files/Igor Procedures" (see **Igor Pro User Files** on page II-46 for details). You can also store them in another folder of your choice and place an alias or shortcut for your folder in "Igor Pro User Files/Igor Procedures".	Igor automatically opens any procedure file in "Igor Pro Folder/Igor Procedures" and "Igor Pro User Files/Igor Procedures" and subfolders or referenced by an alias or shortcut in those folders, and leaves it open in all experiments.

Following this scheme, you will know where to put procedure files that you get from colleagues and where to look for them when you need them.

Utility and global procedures should be general-purpose so that they can be used from any experiment. Thus, they should not rely on specific waves, global variables, global strings, specific windows or any other objects specific to a particular experiment. See **Writing General-Purpose Procedures** on page IV-146 for further guidelines.

After they are debugged and thoroughly tested, you may want to share your procedures with other Igor users. If so, contact WaveMetrics for help in publicizing and distributing them.

WaveMetrics Procedure Files

WaveMetrics has created a large number of utility procedure files that you can use as building blocks. These files are stored in the WaveMetrics Procedures folder. They are described in the WM Procedures Index help file, which you can access through the Windows→Help Windows menu.

You access WaveMetrics procedure files using include statements. Include statements are explained under **The Include Statement** on page IV-145.

Using the Igor Help Browser, you can search the WaveMetrics Procedures folder to find examples of particular programming techniques.

Macros and Functions

There are two kinds of Igor procedures: **macros** and **functions**. They use similar syntax. The main difference between them is that Igor compiles user functions but interprets macros.

Because functions are compiled, they are dramatically faster than macros. Compilation also allows Igor to detect errors in functions when you write the function, whereas errors in macros are detected only when they are executed.

Macros are a legacy of Igor's early days. With rare exceptions, **all new programming should use functions**, not macros. To simplify the presentation of Igor programming, most discussion of macros is segregated into Chapter IV-4, **Macros**.

Scanning and Compiling Procedures

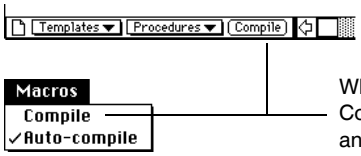
When you modify text in a procedure window, Igor must process it before you can execute any procedures. There are two parts to the processing: scanning and function compilation. In the scanning step, Igor finds out what procedures exist in the window. In the compilation step, Igor's function compiler converts the function text into low-level instructions for later execution.

During scanning, Igor makes the cursor look like scrolling text.

During compilation, Igor makes the cursor look like a meat grinder (text in, bits out).

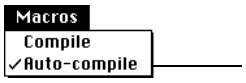
For the sake of brevity, we use the term "compile" to mean "scan and compile" except when we are specifically pointing out the distinction between these two steps.

You can *explicitly* compile the procedures using the Compile button in the Procedure window or the Compile item in the Macros menu.



By default, Igor *automatically* compiles the procedure text at appropriate times. For example, if you type in the Procedure window and then hide it by clicking in the close button, Igor will automatically compile.

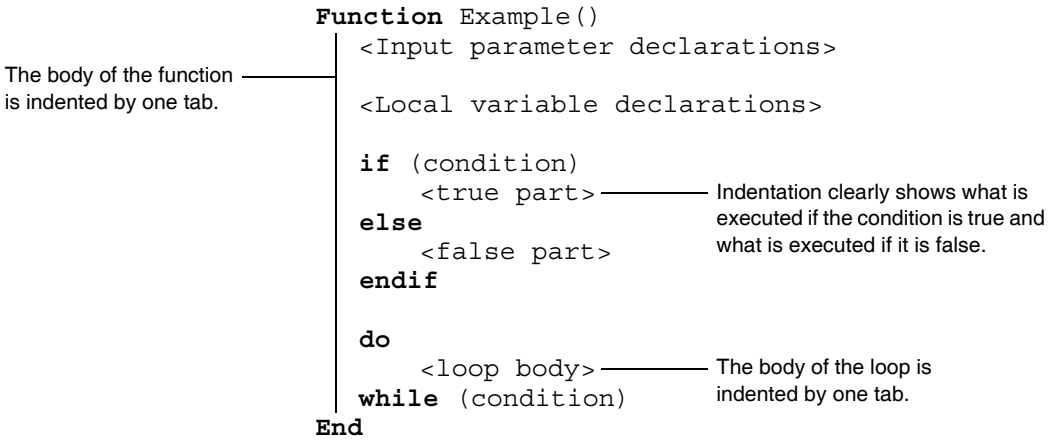
If you have many procedures that take long to compile, you may want to turn auto-compiling off using the Macros menu.



When Auto-compile is deselected, Igor compiles only when you explicitly request it. Igor will still scan the procedures when it needs to know what macros and functions exist.

Indentation Conventions

We use indentation to indicate the structure of a procedure.



The structural keywords, shown in bold here, control the flow of the procedure. The purpose of the indentation is to make the structure of the procedure apparent by showing which lines are within which structural keywords. Matching keywords are at the same level of indentation and all lines within those keywords are indented by one tab.

The Edit menu contains aids for maintaining or adjusting indentation. You can select multiple lines and choose Indent Left or Indent Right. You can have Igor automatically adjust the indentation of a procedure by selecting the whole procedure or a subset and then choosing Adjust Indentation.

Igor does not require that you use indentation but we recommend it for readability.

What's Next

The next chapter covers the core of Igor programming — writing user-defined functions.

Chapter IV-4, **Macros**, explains macros. Because new programming does not use macros, that chapter is mostly of use for understanding old Igor code.

Chapter IV-5, **User-Defined Menus**, explains user-defined menus. It explains how you can add menu items to existing Igor menus and create entire new menus of your own.

Chapter IV-6, **Interacting with the User**, explains other methods of interacting with the user, including the use of dialogs, control panels, and cursors.

Chapter IV-7, **Programming Techniques**, covers an assortment of programming topics. An especially important one is the use of the include statement, which you use to build procedures on top of existing procedures.

Chapter IV-8, **Debugging**, covers debugging using Igor's symbolic debugger.

Chapter IV-9, **Dependencies**, covers dependencies — a way to tie a variable or wave to a formula.

Chapter IV-10, **Advanced Programming**, covers advanced topics, such as communicating with other programs, doing FTP transfers, doing data acquisition, and creating a background task.

Chapter
IV-3

User-Defined Functions

Overview

Most of Igor programming consists of writing user-defined functions.

A function has zero or more numeric, string and wave parameters. You can use local variables to store intermediate results. The function body consists of Igor operations, assignment statements, flow control statements, and calls to other functions.

A function can return a numeric or string result. It can also have a side-effect, such as creating a wave or creating a graph.

Before we dive into the technical details, here is an informal look at some simple examples.

```
Function Hypotenuse(side1, side2)
   Variable side1, side2

   Variable hyp
   hyp = sqrt(side1^2 + side2^2)

   return hyp
End
```

The Hypotenuse function takes two numeric parameters and returns a numeric result. "hyp" is a local variable and sqrt is a built-in function. You could test Hypotenuse by executing the following statement in the command line:

```
Print Hypotenuse(3, 4)
```

Now let's look at a function that deals with text strings.

```
Function/S FirstStr(str1, str2)
   String str1, str2

   String result

   if (CmpStr(str1,str2) < 0)
      result = str1
   else
      result = str2
   endif

   return result
End
```

The FirstStr function takes two string parameters and returns the string that is first in alphabetical order. CmpStr is a built-in function. You could test FirstStr by executing the following statement in the command line:

```
Print FirstStr("ABC", "BCD")
```

Now a function that deals with waves.

```
Function CreateRatioOfWaves(w1, w2, nameOfOutputWave)
   WAVE w1, w2
   String nameOfOutputWave

   Duplicate/O w1, $nameOfOutputWave
   WAVE wOut = $nameOfOutputWave
   wOut = w1 / w2
End
```

The CreateRatioOfWaves function takes two wave parameters and a string parameter. The string is the name to use for a new wave, created by duplicating one of the input waves. The "`WAVE wOut`" statement creates a wave reference for use in the following assignment statement. This function has no direct result (no return statement) but has the side-effect of creating a new wave.

Here are some commands to test CreateRatioOfWaves:

```
Make test1 = {1, 2, 3}, test2 = {2, 3, 4}
CreateRatioOfWaves(test1, test2, "ratio")
Edit test1, test2, ratio
```

Function Syntax

The basic syntax of a function is:

```
Function <Name> (<Parameter list> [<Optional Parameters>]) [:<Subtype>]
   <Parameter declarations>

   <Local variable declarations>

   <Body code>

   <Return statement>
End
```

Here is an example:

```
Function Hypotenuse(side1, side2)
   Variable side1, side2            // Parameter declaration

   Variable hyp                     // Local variable declaration

   hyp = sqrt( side1^2 + side2^2 )  // Body code

   return hyp                       // Return statement
End
```

You could test this function from the command line using one of these commands:

```
Print Hypotenuse(3,4)
Variable/G result = Hypotenuse(3,4); Print result
```

As shown above, the function returns a real, numeric result. The Function keyword can be followed by a flag that specifies a different result type.

Flag	Return Value Type
/C	Complex number
/S	String
/D	Double precision number (obsolete)

The /D flag is obsolete because all calculations are now performed in double precision. However, it is still permitted.

The Function Name

The names of functions must follow the standard Igor naming conventions. Names can consist of up to 31 characters. The first character must be alphabetic while the remaining characters can include alphabetic and numeric characters and the underscore character. Names must not conflict with the names of other Igor objects, functions or operations. Names in Igor are case insensitive.

The Procedure Subtype

You can identify procedures designed for specific purposes by using a subtype. Here is an example:

```
Function ButtonProc(ctrlName) : ButtonControl
   String ctrlName
```

```
   Beep
End
```

Here, "`: ButtonControl`" identifies a function intended to be called when a user-defined button control is clicked. Because of the subtype, this function is added to the menu of procedures that appears in the Button Control dialog. When Igor automatically generates a procedure it generates the appropriate subtype. See **Procedure Subtypes** on page IV-179 for details.

The Parameter List and Parameter Declarations

The parameter list specifies the name for each input parameter. There is no limit on the number of parameters.

All parameters must be declared immediately after the function declaration.

The parameter declaration must declare the type of each parameter using the keywords `Variable`, `String`, `WAVE` or `DFREF`.

If a parameter is a complex number, it must be declared `Variable/C`.

If it is a complex wave, it must be declared `WAVE/C`.

If it is a text wave, it must be declared `WAVE/T`.

Variable and string parameters are usually passed to a subroutine *by value* but can also be passed *by reference*. For an explanation of these terms, see **How Parameters Work** on page IV-44.

Optional Parameters

Following the list of required function input parameters, you can also specify a list of optional input parameters by enclosing the parameter names in brackets. You can supply any number of optional parameter values when calling the function by using the *`ParamName=Value`* syntax. Optional parameters may be of any valid data type. There is no limit on the number of parameters.

All optional parameters must be declared immediately after the function declaration. As with all other variables, optional parameters are initialized to zero. You must use the **ParamIsDefault** function to determine if a particular optional parameter was supplied in the function call.

See **Using Optional Parameters** on page IV-46 for an example.

Local Variable Declarations

The parameter declarations are followed by the local variable declarations if the procedure uses local variables. Local variables exist only during the execution of the procedure. They can be numeric or string and are declared using the `Variable` or `String` keywords. They can optionally be initialized. Here is an example:

```
Function Example(p1)
   Variable p1

   // Here are the local variables
   Variable v1, v2
   Variable v3=0
   Variable/C cv1=cmplx(0,0)
   String s1="test", s2="test2"

   <Body code>
End
```

If you do not supply explicit initialization, Igor automatically initializes local numeric variables with the value zero. Local string variables are initialized with a null value such that, if you try to use the string before you store a value in it, Igor will report an error.

The name of a local variable is allowed to conflict with other names in Igor although they must be unique within the function. Clearly if you create a local variable named "sin" then you will be unable to use Igor's built-in sin function within the function.

You can declare local variables in the body code section of a function as well as in the local variable section.

As of Igor Pro 5, you can define structures and use them as local variables in functions. Structures are defined outside of function definitions and are instantiated as local variables using the STRUCT keyword as described in **Structures in Functions** on page IV-78.

Body Code

This table shows what can appear in body code of a function.

What	Allowed in Functions?	Comment
Assignment statements	Yes	Includes wave, variable and string assignments.
Built-in operations	Yes, with a few exceptions.	See **Operations in Functions** on page IV-89 for exceptions.
Calls to user functions	Yes	
Calls to macros	No	
External functions	Yes	
External operations	Yes, with exceptions.	

As of Igor Pro 5, external operations can be designed to be directly callable from user-defined functions. External operations implemented by old XOPs can not be called directly.

The Execute operation provides a way to work around the limitation on what can be called from user functions. See **The Execute Operation** on page IV-176.

The Return Statement

A return statement often appears at the end of a function, but it can appear anywhere in the function body. You can also have more than one return statement.

The return statement immediately stops executing the function and returns a value to the calling function. If the function is declared as a string function (Function/S), then the return statement must return a string. Otherwise it must return a number.

If there is no return statement or if a function ends without hitting a return statement then the function returns the value NaN (Not a Number) for numeric functions and null for string functions. If the calling function attempts to use the null value, Igor will report an error.

Conditional Statements in Functions

Igor Pro supports two basic forms of conditional statements: if-else-endif and if-elseif-endif statements. Igor also supports multiway branching with switch and strswitch statements.

If-Else-Endif

The form of the if-else-endif structure is

```
if ( <expression> )
   <TRUE part>
else
   <FALSE part>
endif
```

<expression> is a numeric expression that is considered TRUE if it evaluates to any nonzero number and FALSE if it evaluates to zero. The TRUE part and the FALSE part may consist of any number of lines. If the expression evaluates to TRUE, only the TRUE part is executed and the FALSE part is skipped. If the expression evaluates to FALSE, only the FALSE part is executed and the true part is skipped. After the TRUE part or FALSE part code is executed, execution will continue with any code immediately following the if-else-endif statement.

The keyword "else" and the FALSE part may be omitted to give a simple conditional:

```
if ( <expression> )
   <TRUE part>
endif
```

Because Igor is line-oriented, you may not put the `if`, `else` and `endif` keywords all on the same line. They must each be in separate lines with no other code.

If-Elseif-Endif

The if-elseif-endif statement provides a means for creating nested if structures. It has the form:

```
if ( <expression1> )
   <TRUE part 1>
elseif ( <expression2> )
   <TRUE part 2>
[else
   <FALSE part>]
endif
```

These statements follow similar rules as for if-else-endif statements. When any expression evaluates as TRUE (nonzero) the code immediately following the expression will be executed. If all expressions evaluate as FALSE (zero) and there is an else clause, then the statements following the else keyword will be executed. Once any code in a TRUE part or the FALSE part is executed, execution will continue with any code immediately following the if-elseif-endif statement.

Comparisons

The relational comparison operators are used in numeric conditional expressions.

Symbol	Meaning	Symbol	Meaning
==	equal	<=	less than or equal
!=	not-equal	>	greater than
<	less than	>=	greater than or equal

These operators return 1 for TRUE and 0 for FALSE.

The comparison operators work with numeric operands only. To do comparisons with string operands, use the **cmpstr** function.

Comparison operators are usually used in conditional structures but can also be used in arithmetic expressions. For example:

```
wave0 = wave0 * (wave0 < 0)
```

This clips all positive values in wave0 to zero.

See also **Example: Comparison Operators and Wave Synthesis** on page II-98.

Bitwise and Logical Operators

The bitwise and logical operators are also used in conditional expressions.

Symbol	Meaning	Symbol	Meaning
~	Bitwise complement	!	Logical NOT
&	Bitwise AND	&&	Logical AND
|	Bitwise OR	||	Logical OR

The precedence of operators is shown in the table under **Operators** on page IV-5. In the absence of parentheses, an operator with higher precedence (higher in the table) is executed before an operator with lower precedence.

Because the precedence of the arithmetic operators is higher than the precedence of the comparison operators, you can write the following without parentheses:

```
if (a+b != c+d)
   Print "a+b is not equal to c+d"
endif
```

Because the precedence of the comparison operators is higher than the precedence of the logical OR operator, you can write the following without parentheses:

```
if (a==b || c==d)
   Print "Either a equals b or c equals d"
endif
```

For operators with the same precedence, there is no guaranteed order of execution and you must use parentheses to be sure of what will be executed. For example:

```
if ((a&b) != (c&d))
   Print "a ANDED with b is not equal to c ANDED with d"
endif
```

See **Operators** on page IV-5 for more discussion of operators.

Using Bitwise Operators

The bitwise operators are used to test, set, and clear bits. This makes sense only when you are dealing with integer operands.

Bit Action	Operation
Test	AND operator (&)
Set	OR operator (|)
Clear	Bitwise complement operator (~) followed by the bitwise AND operator (&)
Shift left	Multiply by powers of 2
Shift right	Divide by powers of 2

This function illustrates various bit manipulation techniques.

```
Function DemoBitManipulation(vIn)
   Variable vIn

   vIn = trunc(vIn)              // Makes sense with integers only
   Printf "Original value: %d\r", vIn

   Variable vOut
```

```
   if ((vIn & 2^3) != 0)         // Test if bit 3 is set
      Print "Bit 3 is set"
   else
      Print "Bit 3 is cleared"
   endif

   vOut = vIn | (2^3)            // Set bit 3
   Printf "Set bit 3: %d\r", vOut

   vOut = vIn & ~(2^3)           // Clear bit 3
   Printf "Clear bit 3: %d\r", vOut

   vOut = vIn * (2^3)            // Shift three bits left
   Printf "Shift three bits left: %d\r", vOut

   vOut = vIn / (2^3)            // Shift three bits right
   Printf "Shift three bits right: %d\r", vOut
End
```

For a simple demonstration, try this function passing 1 as the parameter.

In this example, 2^3 evaluates to 8 which is a value with only bit 3 set. In a real application, it is a good idea to define a constant that indicates the significance of the bit. These two statements are equivalent:

```
Constant kSystemReadyMask 8      // 8 specified in decimal notation

Constant kSystemReadyMask 0x08   // 8 specified in hexadecimal notation
```

Switch Statements

The switch construct can sometimes be used to simplify complicated flow control. It chooses one of several execution paths depending on a particular value.

Instead of a single form of switch statement, as is the case in C, Igor has two types: *switch* for numeric expressions and *strswitch* for string expressions. The basic syntax of these switch statements is as follows:

```
switch(<numeric expression>)        // numeric switch
   case <literal number or numeric constant>:
      <code>
      [break]
   case <literal number or numeric constant>:
      <code>
      [break]
   . . .
   [default:
      <code>]
endswitch

strswitch(<string expression>)      // string switch
   case <literal string or string constant>:
      <code>
      [break]
   case <literal string or string constant>:
      <code>
      [break]
   . . .
   [default:
      <code>]
endswitch
```

The switch numeric or string expression is evaluated and execution proceeds with the code following the matching case label. When none of the case labels match, execution will continue at the default label, if it is present, or otherwise the switch will exit with no action taken.

All of the case labels must be numeric or string constant expressions and they must all have unique values within the switch statement. The constant expressions can either be literal values or they must be declared using the constant and strconstant keywords for numeric and string switches, respectively. For more about constants, see **Constants** on page IV-40.

Execution proceeds within each case until a break statement is encountered or the endswitch is reached. The break statement explicitly exits the switch construct. Usually, you will put a break statement at the end of each case. If you omit the break statement, execution continues with the next case label. Do this when you want to execute a single action for more than one switch value.

The following examples illustrate how switch constructs can be used in Igor:

```
Constant kThree=3
StrConstant ksHow="how"

Function NumericSwitch(a)
   Variable a

   switch(a)                     // numeric switch
      case 1:
         print "a is 1"
         break
      case 2:
         print "a is 2"
         break
      case kThree:
      case 4:
         print "a is 3 or 4"
         break
      default:
         print "a is none of those"
         break
   endswitch
End

Function StringSwitch(a)
   String a

   strswitch(a)                  // string switch
      case "hello":
         print "a is hello"
         break
      case ksHow:
         print "a is how"
         break
      case "are":
      case "you":
         print "a is are or you"
         break
      default:
         print "a is none of those"
         break
   endswitch
End
```

Loops

Igor implements two basic types of looping structures: do-while and for loops. The do-while loop iterates through the loop code and tests an exit condition at the end of each iteration. The for loop is more complex; the beginning of a for loop includes expressions for initializing and updating variables as well as testing the loop's exit condition at the start of each iteration.

Do-While Loop

The form of the do-while loop structure is:

```
do
   <loop body>
while(<expression>)
```

This loop runs until the expression evaluates to zero or until a break statement is executed.

This example will always execute the body of the loop at least once, like the do-while loop in C.

```
Function Test(lim)
   Variable lim      // We use this parameter as the loop limit.

   Variable sum=0
   Variable i=0      // We use i as the loop variable.
   do
      sum += i       // This is the body; equivalent to sum=sum+i.
      i += 1         // Increment the loop variable.
   while(i < lim)
   return sum
End
```

Nested Do-While Loops

A nested loop is a loop within a loop. Here is an example:

```
Function NestedLoopTest(numOuterLoops, numInnerLoops)
   Variable numOuterLoops, numInnerLoops

   Variable i, j

   i = 0
   do
      j = 0
      do
         <inner loop body>
         j += 1
      while (j < numInnerLoops)
      i += 1
   while (i < numOuterLoops)
End
```

While Loop

This fragment will execute the body of the loop zero or more times, like the while loop in C.

```
do
   if (i > lim)
      break       // This breaks out of the loop.
   endif
   <loop body>
   i += 1
while(1)          // This would loop forever without the break.
...               // Execution continues here after the break.
```

In this example, the loop increment is 1 but it can be any value.

For Loop

The basic syntax of a for loop is:

```
for(<initialize>;<continue test>;<update>)
   <loop body>
endfor
```

Here is a simple example:

```
Function Example1()
   Variable i

   for(i=0;i<5;i+=1)
      print i
   endfor
End
```

The beginning of a for loop consists of three semicolon-separated expressions. The first is usually an assignment statement that initializes one or more variables. The second is a conditional expression used to determine if the loop should be terminated — if true, nonzero, the loop is executed; if false, zero, the loop terminates. The third expression usually updates one or more loop variables.

When a for loop executes, the initialization expression is evaluated only once at the beginning. Then, for each iteration of the loop, the continuation test is evaluated at the start of every iteration, terminating the loop if needed. The third expression is evaluated at the end of the iteration and usually increments the loop variable.

All three expressions in a for statement are optional and can be omitted independent of the others; only the two semicolons are required. The expressions can consist of multiple assignments, which must be separated by commas.

In addition to the test expression, for loops may also be terminated by break or return statements within the body of the loop. A continue statement executed within the loop skips the remaining body code and execution continues with the loop's update expression.

Here is a more complex example:

```
Function Example2()
   Variable i,j

   for(i=0,j=10; ;i+=1,j*=2)
      if( i==2 )
         continue
      endif
      Print i,j
      if( i==5 )
         break
      endif
   endfor
End
```

Break Statement

A break statement terminates execution of do-while loops, for loops, and switch statements. The break statement continues execution with the statement after the *enclosing* loop's while, endfor, or endswitch statement. A nested do-while loop example demonstrates this:

```
   …
   Variable i=0, j

   do                // Starts outer loop.
      if (i > numOuterLoops)
         break       // Break #1, exits from outer loop.
      endif
```

```
      j = 0
      do              // Start inner loop.
         if (j > numInnerLoops)
            break     // Break #2, exits from inner loop only.
         endif
         j += 1
      while (1)       // Ends inner loop.
      …               // Execution continues here from break #2.
      i += 1
   while (1)          // Ends outer loop.
   …                  // Execution continues here from break #1.
```

Continue Statement

The continue statement can be used in do-while and for loops to short-circuit the loop and return execution back to the top of the loop. When Igor encounters a continue statement during execution, none of the code following the continue statement is executed during that iteration.

Flow Control for Aborts

Igor Pro includes a specialized flow control construct and keywords that you can use to test for and respond to abort conditions. The AbortOnRTE and AbortOnValue keywords can be used to trigger aborts, and the try-catch-endtry construct can be used to control program execution when aborts occur. These are advanced techniques. If you are just starting with Igor programming, you may want to skip this section and come back to it later.

AbortOnRTE Keyword

The AbortOnRTE keyword can be used to raise an abort whenever a runtime error occurs. Use AbortOnRTE immediately after (on the same line preceded by a semicolon) a command or on a line following a sequence of commands for which you wish to catch a runtime error.

See **AbortOnRTE** on page V-16 for further details. For a usage example see **try-catch-endtry Example** on page IV-39.

AbortOnValue Keyword

The AbortOnValue keyword can be used to abort function execution when a specified abort condition is satisfied. When AbortOnValue triggers an abort, it can also return a numeric abort code that you can use to characterize the cause.

AbortOnValue is a low overhead, short-hand replacement for an

```
if(abortTest)
   Abort
endif
```

code block that you would normally use in a procedure to test for an abort condition.

See **AbortOnValue** on page V-16 for further details. For a usage example see **try-catch-endtry Example** on page IV-39.

try-catch-endtry Flow Control

The try-catch-entry flow control construct has the following syntax:

```
try
   <possible abort code>
catch
   <code to handle abort>
endtry
```

The try-catch-entry flow control construct can be used to catch and respond to abort conditions in user functions. Code within the try-catch area tests for abort conditions and when the first abort condition is satisfied, execution will immediately jump to code within the catch-endtry area where execution proceeds with code to handle the abort condition. Normal flow (no aborts) will skip past all code within the catch-endtry area.

When an abort occurs within the try-catch area, the construct returns a numeric code in the V_AbortCode variable, which provides information about the cause of the abort.

See **try-catch-endtry** on page V-713 for further details.

try-catch-endtry Example

The following example demonstrates how abort flow control may be used. Copy the code to your Procedure window and execute the `foo` function with 0 to 6 as input parameters.

```
Function foo(a)
   Variable a

   print "A"
   try
      print "B1"
      AbortOnValue  a==1 || a==2,33
      print "B2"
      bar(a)
      print "B3"
      try
         print "C1"
         if( a==4 || a==5 )
            Make $""; AbortOnRTE
         endif
         Print "C2"
      catch
         Print "D1"
         // will be runtime error so pass along to outer catch
         AbortOnValue a==5, V_AbortCode
         Print "D2"
      endtry
      Print "B4"
      if( a==6 )
         do
         while(1)
      endif
      Print "B5"
   catch
      print "Abort code= ", V_AbortCode
      if( V_AbortCode == -4 )
         Print "Runtime error= ", GetRTError(1)
      endif
      if( a==1 )
         abort "Aborting again"
      endif
      Print "E"
   endtry
   print "F"
End

Function bar(b)
   Variable b

   Print "Bar A"
   AbortOnValue b==3,99
```

```
   Print "Bar B"
End
```

Constants

You can define named numeric and string constants in Igor procedure files and use them in the body of user-defined functions.

Constants are defined in procedure files using following syntax:

```
Constant <name1> = <literal number> [, <name2> = <literal number>]
StrConstant <name1> = <literal string> [, <name2> = <literal string>]
```

For example:

```
Constant kIgorStartYear=1989,kIgorEndYear=2020
StrConstant ksPlatformMac="Macintosh",ksPlatformWin="Windows"

Function Test1()
   Variable v1 = kIgorStartYear
   String s1 = ksPlatformMac
   Print v1, s1
End
```

Constants declared like this are public and can be used in any function in any procedure file. A typical use would be to define constants in a utility procedure file that could be used from other procedure files as parameters to the utility routines. Be sure to use precise names to avoid conflicts with public constants declared in other procedure files.

If you are defining constants for use in a single procedure file, for example to improve readability or make the procedures more maintainable, you should use the `static` keyword (see **Static** on page V-594 for details) to limit the scope to the given procedure file.

```
static Constant kStart=1989,kEnd=2020
static StrConstant ksMac="Macintosh",ksWin="Windows"
```

We suggest that you use the “k” prefix for numeric constants and the “ks” prefix for string constants. This makes it immediately clear that a particular keyword is a constant.

Names for numeric and string constants are allowed to conflict with all other names. Duplicate constants of a given type are not allowed (except static in different files and when used with the override keyword). The only true conflict is with variable names and with certain built-in functions that do not take parameters such as pi. Variable names (including local variable names, waves, NVARs, and SVARs) override constants, but constants override functions such as pi.

Pragmas

A pragma is a statement in a procedure file that sets a compiler mode or passes other information from the programmer to Igor. The form of a pragma statement is:

```
#pragma keyword [= parameter]
```

The pragma statement must be flush against the left margin of the procedure window, with no indentation.

Currently, Igor supports the following pragmas:

```
#pragma rtGlobals = value
#pragma version = versionNumber
#pragma IgorVersion = versionNumber
#pragma ModuleName = name
#pragma IndependentModule = name
```

The effect of a pragma statement lasts until the end of the procedure file that contains it.

The rtGlobals Pragma

The rtGlobals pragma controls aspects of the Igor compiler and runtime error checking in user-defined functions.

Prior to Igor Pro 3, to access a global (wave or variable) from a user-defined function, the global had to already exist. Igor Pro 3 introduced "runtime lookup of globals" under which the Igor compiler did not require globals to exist at compile time but rather connected references, declared with WAVE, NVAR and SVAR statements, to globals at runtime. Igor Pro 6.20 introduced stricter compilation of wave references and runtime checking of wave index bounds.

You enable and disable these behaviors using an rtGlobals pragma. For example:

```
#pragma rtGlobals = 3   // Strict wave reference mode, runtime bounds checking
```

A given rtGlobals pragma governs just the procedure file in which it appears. The pragma must be flush left in the procedure file and is typically put at the top of the file.

The rtGlobals pragma is defined as follows:

#pragma rtGlobals=0	Specifies the old, pre-Igor Pro 3 behavior. This is obsolete and should not be used.
#pragma rtGlobals=1	Turns on runtime lookup of globals. In Igor Pro 5 or later, this is the default setting if there is no rtGlobals pragma in a given procedure file.
#pragma rtGlobals=2	Forces old experiments out of compatibility mode. This is superceded by rtGlobals=3. It is described under **Legacy Code Issues** on page IV-90.
#pragma rtGlobals=3	Turns on runtime lookup of globals, strict wave references and runtime checking of wave index bounds. Requires Igor Pro 6.2 or later.

If your procedures will run only with Igor Pro 6.20 or later, rtGlobals=3 is recommended. Otherwise rtGlobals=1 is recommended.

If your procedures must run with old Igor versions but you want to use rtGlobals=3 when possible, you can do this:

```
#if IgorVersion() >= 6.20
#pragma rtGlobals = 3   // Strict wave reference mode, runtime bounds checking
#endif
```

Since rtGlobals=1 is the default, this will use rtGlobals=1 for Igor Pro 6.02A through 6.12 and will use rtGlobals=3 for Igor Pro 6.20 or later. It will not work for versions prior to 6.02A because they don't support conditional compilation (#if).

Under strict wave references (rtGlobals=3), you must create a wave reference for any use of a wave. Without strict wave references (rtGlobals=1), you do not need to create a wave reference unless the wave is used in an assignment statement. For example:

```
Function Test()
   jack = 0       // Error under rtGlobals=1 and under rtGlobals=3
   Display jack   // OK under rtGlobals=1, error under rtGlobals=3

   Wave jack      // jack is now a wave reference rather than a bare name
   Display jack   // OK under rtGlobals=1 and under rtGlobals=3
End
```

Even with rtGlobals=3, this compiles without error:

```
Function Test()
   // Make creates an automatic wave reference when used with a simple name
   Make jack
```

```
   Display jack   // OK under rtGlobals=1 and rtGlobals=3
End
```

See **Automatic Creation of WAVE References** on page IV-56 for details.

Under runtime wave index checking (rtGlobals=3), Igor reports an error if a wave index is out-of-bounds:

```
Function Test()
   Make/O/N=5 jack = 0  // Creates automatic wave reference

   jack[4] = 123        // OK
   jack[5] = 234        // Runtime error under rtGlobals=3.
                        // Clipped under rtGlobals=1.

   Variable index = 5
   jack[index] = 234    // Runtime error under rtGlobals=3.
                        // Clipped under rtGlobals=1.

   // Create and use a dimension label for point 4 of jack
   SetDimLabel 0,4,four,jack
   jack[%four] = 234    // OK

   // Use a non-existent dimension label.
   jack[%three] = 345   // Runtime error under rtGlobals=3.
                        // Clipped under rtGlobals=1.
   // Under rtGlobals=1, this statement writes to point 0 of jack.
End
```

NOTE: All Igor documentation assumes that rtGlobals=1 is in effect unless otherwise stated.

See also: **Runtime Lookup of Globals** on page IV-50
Automatic Creation of WAVE References on page IV-56
Automatic Creation of NVAR and SVAR References on page IV-55
Legacy Code Issues on page IV-90

The version Pragma

The version pragma sets the version of the procedure file. It is optional and is of interest mostly if you are the developer of a package used by a widespread group of users.

For details on the version pragma, see **Procedure File Version Information** on page IV-145.

The IgorVersion Pragma

The IgorVersion pragma is also optional and of interest to developers of packages. It gives you a way to prevent procedures from running, or at least generating compilation errors that cannot be fixed, under versions of Igor Pro older than the specified version number.

For example, the statement:

```
#pragma IgorVersion = 4.0
```

requires Igor Pro 4.0 or later for the procedure file. This will be helpful if you share your code with others because you can ensure that your procedures will only run with versions of Igor Pro that fully support all of the new or updated features you may have used. This Igor version check was added with Igor Pro 4.0, and it will not work with previous versions.

The ModuleName Pragma

The ModuleName pragma gives you the ability to use static functions and Proc Pictures in a global context, such as in the action procedure of a control or on the Command Line. Using this pragma entails a two step process: define a name for the procedure file, and then use a special syntax to access objects in the named procedure file.

To define a module name for a procedure file use the format:

```
#pragma ModuleName= name
```

This statement associates the specified module name with the procedure file in which the statement appears.

You can then use objects from the named procedure file by preceding the object name with the name of the module and the # character. For example:

```
#pragma ModuleName= myGreatProcedure

Static Function foo(a)
   Variable a

   return a+100
End
```

Then on the command line you can execute:

```
Print myGreatProcedure#foo(3)
```

You should make sure the name is unlikely to clash with other names you or others might use. WaveMetrics will use names with a `WM_` prefix, so you should avoid such names. The ModuleName pragma will not work with versions previous to Igor Pro 5.0.

For further discussion see **Regular Modules** on page IV-212.

The IndependentModule Pragma

The IndependentModule pragma is a way for you to designate groups of one or more procedure files that are compiled and linked separately. Once compiled and linked, the code remains in place and is usable even though other procedures may fail to compile. This allows functioning control panels and menus to continue to work regardless of user programming errors.

A file is designated as an independent module using

```
#pragma IndependentModule=imName
```

This is similar to `#pragma ModuleName=`*modName* (see **The ModuleName Pragma** on page IV-42) and, just as in the case of calling static functions in a procedure with #pragma ModuleName, calling nonstatic function in an IndependentModule from outside the module requires the use of *imName#functionName*() syntax.

For further discussion see **Independent Modules** on page IV-214.

Unknown Pragmas

Starting with Igor Pro version 3.04, Igor ignores pragmas that it does not know about. This allows newer versions of Igor to use new pragmas while older versions ignore them. The downside of this change is that, if you misspell a pragma keyword, Igor will not warn you about it.

Proc Pictures

Proc Pictures are binary PNG or JPEG images encoded as printable ASCII procedures in procedure files. They are intended for programmers who need images as part of the user interface for a procedure package. They can be used with the **DrawPICT** operation (see page V-128) and with the **Picture** keyword (see page V-480) in certain controls.

The syntax for defining and using a Proc Picture is illustrated in the following example:

```
// PNG: width= 56, height= 44
Picture myPictName
   ASCII85Begin
   M,6r;%14!\!!!!.8Ou6I!!!!Y!!!!M#Qau+!5G;q_uKc;&TgHDFAm*iFE_/6AH5;7DfQssEc39jTBQ
   =U!7FG,5u`*!m?g0PK.mR"U!k63rtBW)]$T)Q*!=Sa1TCDV*V+l:Lh^NW!fu1>;(.<VU1bs4L8&@Q_
```

```
   <4e(%"^F50:Jg6);j!CQdUA[dh6]%[OkHSC,ht+Q7ZO#.6U,IgfSZ!R1g':oO_iLF.GQ@RF[/*G98D
   bjE.g?NCte(pX-($m^\_FhhfL`D9uO6Qi5c[r4849Fc7+*)*O[tY(6<rkm^)/KLIc]VdDEbF-n5&Am
   2^hbTu:U#8ies_W<LGkp_LEU1bs4L8&?fqRJ[h#sVSSz8OZBBY!QNJ
   ASCII85End
End

Function test()
   NewPanel
   DrawPict 0,0,1,1,ProcGlobal#myPictName
End
```

The ASCII text in the `myPictName` procedure between the `ASCII85Begin` and `ASCII85End` is similar to output from the Unix `btoa` command or, on Macintosh, StuffIt's binary to ASCII operation. (You must remove any extra header and trailer information if you use either of these utilities to create a Proc Picture.)

You can create Proc Pictures in Igor Pro from normal, global pictures using the Picture dialog (see **Pictures** on page III-421). Select a picture in the dialog and click the Copy Proc Picture button to place the text on the Clipboard and then paste it in your procedure file. If the existing picture is not a JPEG or PNG, it is converted to PNG.

Proc Pictures can be either global or local in scope. Global pictures can be used in all experiment procedure files; local pictures can be used only within the procedure file where they are defined. Proc Pictures are global by default and the picture name must be unique for all procedure files in an experiment. Proc Pictures can be made local in scope by declaring them Static (see **Static** on page V-594).

When accessing a Proc Picture from DrawPict, the picture name must be preceded by either the `ProcGlobal` keyword or the procedure module name, and the two names joined together with `#`. This naming convention is necessary to avoid potential conflicts with any existing experiment global pictures defined via the Pictures dialog.

For a global Proc Picture, you must use the `ProcGlobal` keyword as the prefix:

`ProcGlobal#`*`gProcPictName`*

For a static Proc Picture, you must use the module name defined in the procedure file by the `#pragma ModuleName = `*`modName`* statement (see **ModuleName** on page V-424) as the prefix:

`modName``#`*`ProcPictName`*

How Parameters Work

There are two ways of passing parameters from a routine to a subroutine: **pass-by-value** and **pass-by-reference**. "Pass-by-value" means that the routine passes the *value* of an expression to the subroutine. "Pass-by-reference" means that the routine passes *access to a variable* to the subroutine. The important difference is that, in pass-by-reference, the subroutine can change the original variable in the calling routine.

The C and Pascal languages allow programmers to use either of these techniques, at their option while FORTRAN uses pass-by-reference only. Like C, Igor also allows either method for numeric and string variables.

Example of Pass-By-Value

```
Function Routine()
   Variable v = 4321
   String s = "Hello"

   Subroutine(v, s)
End

Function Subroutine(v, s)
   Variable v
   String s

   Print v, s
```

```
   // These lines have NO EFFECT on the calling routine.
   v = 1234
   s = "Goodbye"
End
```

Note that v and s are *local variables* in Routine. In Subroutine, they are *parameters* which act very much like local variables. The names "v" and "s" are local to the respective functions. The v in Subroutine is not the same variable as the v in Routine although it initially has the same value.

The last two lines of Subroutine set the value of the local variables v and s. They have no effect on the value of the variables v and s in the calling Routine. What is passed to Subroutine is the numeric value 4321 and the string value "Hello".

Pass-By-Reference

You can specify that a parameter to a function is to be passed by reference rather than by value. In this way, the function called can change the value of the parameter and update it in the calling function. This is much like using pointers to arguments in C. This technique is needed and appropriate only when you need to return more than one value from a function.

The variable or string being passed must be a local variable and can not be a global variable. To designate a variable or string parameter for pass-by-reference, simply prepend an ampersand symbol (&) before the name in the parameter declaration:

```
Function Subroutine(num1,num2,str1)
   Variable &num1, num2
   String &str1

   num1= 12+num2
   str1= "The number is"
End
```

and then call the function with the name of a local variable in the reference slot:

```
Function Routine()
   Variable num= 1000
   String str= "hello"
   Subroutine(num,2,str)
   print str, num
End
```

When executed, Routine prints "The number is 14" rather than "hello 1000", which would be the case if pass-by-reference were not used.

A pass-by-reference parameter can be passed to another function that expects a reference:

```
Function SubSubroutine(b)
   Variable &b
   b= 123
End

Function Subroutine(a)
   Variable &a
   SubSubroutine(a)
End

Function Routine()
   Variable num
   Subroutine(num)
   print num
End
```

Note: Functions with pass-by-reference parameters can only be called from other functions — not from the command line.

How Waves Are Passed

Here is an example of a function "passing a wave" to a subroutine.

```
Function Routine()
   Make/O wave0 = x
   Subroutine(wave0)
End

Function Subroutine(w)
   WAVE w

   w = 1234    // This line DOES AFFECT the wave referred to by w.
End
```

We are really not passing a wave to Subroutine, but rather we are passing a reference to a wave. The parameter `w` is the wave reference.

Waves are global objects that exist independent of functions. The subroutine can use the wave reference to modify the contents of the wave. Using the terminology of "pass-by-value" and "pass-by-reference", the wave reference is passed by value, but this has the effect of "passing the wave" by reference.

Using Optional Parameters

Following is an example of a function with two optional input parameters:

```
Function opParamTest(a,b, [c,d])
   Variable a,b,c,d           // c and d are optional parameters

   // Determine if the optional parameters were supplied
   if( ParamIsDefault(c) && ParamIsDefault(d) )
      Print "Missing optional parameters c and d."
   endif

   Print a,b,c,d
End
```

Executing on the Command Line with none of the optional inputs:

```
•opParamTest(8,6)
  Missing optional parameters c and d.
  8  6  0  0
```

Executing with an optional parameter as an input:

```
•opParamTest(8,6, c=66)
  8  6  66  0
```

Note that the optional parameter is explicitly defined in the function call using the *ParamName=Value* syntax. All optional parameters must come after any required function parameters.

Local Versus Global Variables

Numeric and string variables can be **local** or **global**. "Local" means that the variable can be accessed only from within the procedure in which it was created. "Global" means that the variable can be accessed from the command line or from within any procedure. The following table shows the characteristics of local and global variables:

Local Variables	Global Variables
Are part of a procedure.	Are part of an Igor experiment.
Created using Variable or String within a procedure.	Created using Variable or String or Variable/G or String/G from the command line or within a procedure.

Local Variables	Global Variables
Used to store temporary results while the procedure is executing.	Used to store values that are saved from one procedure invocation to the next or to store data that is accessed by many procedures.
Cease to exist when the procedure ends.	Exist until you use KillVariables, KillStrings, or KillDataFolder.
Can be accessed only from within the procedure in which they were created.	Can be accessed from the command line or from within any procedure.

Local variables are private to the function in which they are declared and vanish when the function exits. Global variables are public and persistent — they exist until you explicitly delete them.

If you write a function whose main purpose is to display a dialog to solicit input from the user, you may want to store the user's choices in global variables and use them to initialize the dialog the next time the user invokes it. This is an example of saving values from one invocation of a function to the next.

If you write a set of functions that loads, displays and analyzes experimentally acquired data, you may want to use global variables to store values that describe the conditions under which the data was acquired and to store the results from the analyses. These are examples of data accessed by many procedures.

Local Variables Used by Igor Operations

When invoked from a function, a number of Igor's operations return results via local variables. For example, the WaveStats operation creates a number of local variables with names such as V_avg, V_sigma, etc. The following function illustrates this point.

```
Function PrintWaveAverage(w)
   WAVE w

   WaveStats/Q w
   Print V_avg
End
```

When the Igor compiler compiles the WaveStats operation, it creates various local variables, V_avg among them. When the WaveStats operation runs, it stores results in these local variables.

In addition to creating local variables, a few operations, such as CurveFit and FuncFit, check for the existence of specific local variables to provide optional behavior. For example:

```
Function ExpFitWithMaxIterations(w, maxIterations)
   WAVE w
   Variable maxIterations

   Variable V_FitMaxIters = maxIterations

   CurveFit exp w
End
```

The CurveFit operation looks for a local variable named V_FitMaxIters, which sets the maximum number of iterations before the operation gives up.

The documentation for each operation lists the special variables that it creates or looks for.

Converting a String into a Reference Using $

The $ operator converts a string expression into an object reference. The referenced object is usually a wave but can also be a global numeric or global string variable, a window, a symbolic path or a function. This is a common and important technique.

We often use a *string* to pass the *name* of a wave to a procedure or to algorithmically generate the name of a wave. Then we use the $ operator to convert the string into a wave reference so that we can operate on the wave.

The following trivial example shows why we need to use the $ operator:

```
Function DisplayXY(xWaveNameStr, yWaveNameStr)
   String xWaveNameStr, yWaveNameStr

   Display $yWaveNameStr vs $xWaveNameStr
End
```

Here we use $ to convert the string parameters into wave references. The function will display the wave whose name is stored in the yWaveNameStr string versus the wave whose name is stored in the xWaveNameStr string.

If we omitted the $ operators, we would have

```
Display yWaveNameStr vs xWaveNameStr
```

This would result in an error because the Display operation would look for a wave named yWaveName and would not find it.

As shown in the following example, $ can create references to global numeric and string variables as well as to waves.

```
Function Test(vStr, sStr, wStr)
   String vStr, sStr, wStr

   NVAR v = $vStr       // v is local name for global numeric var
   v += 1
   SVAR s = $sStr       // s is local name for global string var
   s  += "Hello"
   WAVE w = $wStr       // w is local name for global wave
   w += 1
End

Variable/G gVar = 0; String/G gStr = ""; Make/O/N=5 gWave = p
Test("gVar", "gStr", "gWave")
```

The NVAR, SVAR and WAVE references are necessary in functions so that the compiler can identify the kind of object. This is explained under **Accessing Global Variables and Waves** on page IV-50.

Using $ to Refer to a Window

A number of Igor operations modify or create windows, and optionally take the name of a window. You need to use a string variable if the window name is not determined until run time but must convert the string into a name using $.

For instance, this function creates a graph using a name specified by the calling function:

```
Function DisplayXY(xWaveNameStr, yWaveNameStr, graphNameStr)
   String xWaveNameStr, yWaveNameStr
   String graphNameStr     // Contains name to use for the new graph

   Display /N=$graphNameStr $yWaveNameStr vs $xWaveNameStr
   Cursor /W=$graphNameStr A, $yWaveNameStr, 0
End
```

The $ operator in /N=$graphNameStr converts the contents of the string graphNameStr into a graph name as required by the Display operation /N flag. If you forget $, the command would be:

```
   Display /N=graphNameStr $yWaveNameStr vs $xWaveNameStr
```

This would create a graph literally named graphNameStr.

After creating the graph, this example uses $graphNameStr again to specify the target of the Cursor operation.

Using $ In a Data Folder Path

$ can also be used to convert a string to a name in a data folder path. This is used when one of many data folders must be selected algorithmically.

Assume you have a string variable named dfName that tells you in which data folder a wave should be created. You can write:

```
Make/O root:$(dfName):wave0
```

The parentheses are necessary because the $ operator has low precedence, so Igor would interpret this:

```
Make/O root:$dfName:wave0            // ERROR
```

to mean:

```
Make/O root:$(dfName:wave0)          // ERROR
```

To avoid this, you must use parentheses like this:

```
Make/O root:$(dfName):wave0          // OK
```

Compile Time Versus Runtime

Because Igor user-defined functions are compiled, you need to be aware of the difference between "compile time" and "runtime".

Compile time is when Igor analyzes the text of all functions and produces low-level instructions that can be executed quickly later. This happens when you modify a procedure window and then:

- Choose Compile from the Macros menu.
- Click the Compile button at the bottom of a procedure window.
- Activate a nonprocedure window.

Runtime is when Igor actually executes a function's low-level instructions. This happens when:

- You invoke the function from the command line.
- The function is invoked from another procedure.
- Igor updates a dependency which calls the function.
- You use a button or other control that calls the function.

Conditions that exist at compile time are different from those at runtime. For example, a function can reference a global variable. The global does not need to exist at compile time, but it does need to exist at runtime. This issue is discussed in detail in the following sections.

Here is another example of the distinction between compile time and runtime:

```
Function Example(w)
   WAVE w

   w= sin(x)
   FFT w
   w= r2polar(w)
End
```

The declaration "`WAVE w`" specifies that w is expected to be a real wave. This is correct until the FFT executes and thus the first wave assignment produces the correct result. After the FFT is executed at runtime, however, the wave becomes complex. The Igor compiler does not know this and so it will compile the second wave assignment on the assumption that w is real. A compile-time error will be generated com-

plaining that r2polar is not available for this number type — i.e., real. To provide Igor with the information that the wave is complex after the FFT you need to rewrite the function like this:

```
Function Example(w)
   WAVE w

   w= sin(x)
   FFT w
   WAVE/C wc= w
   wc= r2polar(wc)
End
```

A statement like "`WAVE/C wc= w`" has the compile-time behavior of creating a symbol, wc, and specifying that it refers to a complex wave. It has the runtime behavior of making wc refer to a specific wave. The runtime behavior can not occur at compile time because the wave may not exist at compile time.

Accessing Global Variables and Waves

Global numeric variables, global string variables and waves can be referenced from any function. A function can refer to a global that does not exist at compile-time. For the Igor compiler to know what type of global you are trying to reference, you need to declare references to globals.

Consider the following function:

```
Function BadExample()
   gStr1 = "The answer is:"
   gNum1 = 1.234
   wave0 = 0
End
```

The compiler can not compile this because it doesn't know what gStr1, gNum1 and wave0 are. We need to specify that they are a global string variable, a global numeric variable and a wave, respectively:

```
Function GoodExample1()
   SVAR gStr1 = root:gStr1
   NVAR gNum1 = root:gNum1
   WAVE wave0 = root:wave0

   gStr1 = "The answer is:"
   gNum1 = 1.234
   wave0 = 0
End
```

The SVAR statement specifies two important things for the compiler: first, that gStr1 is a global string variable; second, that gStr1 refers to a global string variable named gStr1 in the root data folder. Similarly, the NVAR statement identifies gNum1 and the WAVE statement identifies wave0. With this knowledge, the compiler can compile the function.

The technique illustrated here is called "runtime lookup of globals" because the compiler compiles code that associates the symbols gStr1, gNum1 and wave0 with specific global variables at runtime.

Runtime Lookup of Globals

The syntax for runtime lookup of globals is:

```
NVAR <local name1>[= <path to var1>][, <loc name2>[= <path to var2>]]…
SVAR <local name1>[= <path to str1>][, <loc name2>[= <path to str2>]]…
WAVE <local name1>[= <path to wave1>][, <loc name2>[= <path to wave2>]]…
```

NVAR and SVAR create a reference to a global numeric or string variable and WAVE creates a reference to a wave. At compile time, these statements identify the referenced objects. At runtime, the connection is made between the local name and the actual object. Consequently, the object must exist when these statements are executed.

<local name> is the name by which the global variable, string or wave is to be known within the user function. It does not need to be the same as the name of the global variable. The example function could be rewritten as follows:

```
Function GoodExample2()
   SVAR str1 = root:gStr1     // str1 is the local name.
   NVAR num1 = root:gNum1     // num1 is the local name.
   WAVE w = root:wave0        // w is the local name.

   str1 = "The answer is:"
   num1 = 1.234
   w = 0
End
```

If you use a local name that is the same as the global name, and if you want to refer to a global in the current data folder, you can omit the <path to …> part of the declaration:

```
Function GoodExample3()
   SVAR gStr1           // Refers to gStr1 in current data folder.
   NVAR gNum1           // Refers to gNum1 in current data folder.
   WAVE wave0           // Refers to wave0 in current data folder.

   gStr1 = "The answer is:"
   gNum1 = 1.234
   wave0 = 0
End
```

GoodExample3 accesses globals in the current data folder while GoodExample2 accesses globals in a specific data folder.

If you use <path to …>, it may be a simple name (gStr1) or it may include a full or partial path to the name.

The following are valid examples, referencing a global numeric variable named gAvg:

```
   NVAR gAvg= gAvg
   NVAR avg= gAvg
   NVAR gAvg
   NVAR avg= root:Packages:MyPackage:gAvg
   NVAR avg= :SubDataFolder:gAvg
   NVAR avg= $"gAvg"
   NVAR avg= $("g"+ "Avg")
   NVAR avg= ::$"gAvg"
```

As illustrated above, the local name can be the same as the name of the global object and the lookup expression can be either a literal name or can be computed at runtime using $<string expression>.

In some cases, it may be convenient to create global variables in a temporary data folder as a way of passing results back to the calling routine. For example:

```
Function MyWaveStats(inputWave)
   WAVE inputWave

   NewDataFolder/O/S tmpMyWaveStatsDF

   WaveStats/Q inputWave

   Variable/G gNumPoints = V_npnts
   Variable/G gAvg = V_avg
   Variable/G gSdev = V_sdev
End

Function Test()
   Make/O testwave= gnoise(1)

   MyWaveStats(testwave)      // Create temp output data folder.
```

```
   NVAR gNumPoints,gAvg,gSdev // Create references to globals.
   Printf "Points: %g; Avg: %g; SDev: %g\r", gNumPoints,gAvg,gSdev
   KillDataFolder :           // Kill temp output data folder.
End
```

Note that the NVAR statement must appear after the MyWaveStats call because NVAR associates the local names with existing global variables at runtime.

Put WAVE Declaration After Wave Is Created

A wave declaration serves two purposes. At compile time, it tells Igor the local name and type of the wave. At runtime, it connects the local name to the wave. In order for the runtime purpose to work, you must put wave declaration after the wave is created.

```
Function BadExample()
   String path = "root:Packages:MyPackage:wave0"
   Wave w = $path          // WRONG: Wave does not yet exist.
   Make $path
   w = p                   // w is not connected to any wave.
End

Function GoodExample()
   String path = "root:Packages:MyPackage:wave0"
   Make $path
   Wave w = $path          // RIGHT
   w = p
End
```

Both of these functions will successfully compile. BadExample will fail at runtime because w is not associated with a wave, because the wave does not exist when the "Wave w = $path" statement executes.

This rule also applies to NVAR and SVAR declarations.

Runtime Lookup Failure

At runtime, it is possible that a NVAR, SVAR or WAVE statement may fail. For example, `NVAR v1= var1` will fail if var1 does not exist in the current data folder when the statement is executed. You can use the NVAR_Exists, SVAR_Exists, and WaveExists functions to test if a given global reference is valid.

One cause for failure is putting a WAVE statement in the wrong place. For example:

```
Function BadExample()
   WAVE w = resultWave
   <Call a function that creates a wave named resultWave>
   Display w
End
```

This function will compile successfully but will fail at runtime. The reason is that the "`WAVE w = resultWave`" statement has the runtime behavior of associating the local name w with a particular wave. But that wave does not exist until the following statement is executed. The function should be rewritten as:

```
Function GoodExample()
   <Call a function that creates a wave named resultWave>
   WAVE w = resultWave
   Display w
End
```

Runtime Lookup Failure and the Debugger

You can break whenever a runtime lookup fails using the symbolic debugger (described in Chapter IV-8, **Debugging**). It is a good idea to do this, because it lets you know about runtime lookup failures at the moment they occur.

Sometimes you may create a WAVE, NVAR or SVAR reference knowing that the referenced global may not exist at runtime. Here is a trivial example:

```
Function Test()
   WAVE w = testWave
   if (WaveExists(testWave))
      Printf "testWave had %d points.\r", numpnts(testWave)
   endif
End
```

If you enable the debugger's WAVE checking and if you execute the function when testWave does not exist, the debugger will break and flag that the WAVE reference failed. But you wrote the function to handle this situation, so the debugger break is not helpful in this case.

The solution is to rewrite the function using WAVE/Z instead of just WAVE. The /Z flag specifies that you know that the runtime lookup may fail and that you don't want to break if it does. You can use NVAR/Z and SVAR/Z in a similar fashion.

Accessing Complex Global Variables and Waves

You must specify if a global numeric variable or a wave is complex using the /C flag:

```
NVAR/C gc1= gc1
WAVE/C gcw1= gcw1
```

Accessing Text Waves

Text waves must be accessed using the /T flag:

```
WAVE/T tw= MyTextWave
```

Accessing Global Variables and Waves Using Liberal Names

There are two ways to reference an Igor object: using a literal name or path or using a string variable. For example:

```
Wave w = root:MyDataFolder:MyWave   // Using literal path
String path = "root:MyDataFolder:MyWave"
Wave w = $path                      // Using string variable
```

Things get more complicated when you use a liberal name rather than a standard name. A standard name starts with a letter and includes letters, digits and the underscore character. A liberal name includes other characters such as spaces or punctuation.

In general, you must quote liberal names using a single quote so that Igor can determine where the name starts and where it ends. For example:

```
Wave w = root:'My Data Folder':'My Wave'  // Using literal path
String path = "root:'My Data Folder':'My Wave'"
Wave w = $path                            // Using string variable
```

However, there is an exception to the quoting requirement. The rule is:

> You must quote a literal liberal name and you must quote a liberal path stored in a string variable but you must not quote a simple literal liberal name stored in a string variable.

The following functions illustrate this rule:

```
// Literal liberal name must be quoted
Function DemoLiteralLiberalNames()
   NewDataFolder/O root:'My Data Folder'

   Make/O root:'My Data Folder':'My Wave' // Literal name must be quoted
```

```
   SetDataFolder root:'My Data Folder'    // Literal name must be quoted

   Wave w = 'My Wave'                     // Literal name must be quoted
   w = 123

   SetDataFolder root:
End

// String liberal PATH must be quoted
Function DemoStringLiberalPaths()
   String path = "root:'My Data Folder'"
   NewDataFolder/O $path

   path = "root:'My Data Folder':'My Wave'"  // String path must be quoted
   Make/O $path

   Wave w = $path
   w = 123

   SetDataFolder root:
End

// String liberal NAME must NOT be quoted
Function DemoStringLiberalNames()
   SetDataFolder root:

   String dfName = "My Data Folder"       // String name must NOT be quoted
   NewDataFolder/O $dfName

   String wName = "My Wave"               // String name must NOT be quoted
   Make/O root:$(dfName):$wName

   Wave w = root:$(dfName):$wName         // String name must NOT be quoted
   w = 123

   SetDataFolder root:
End
```

The last example illustrates another subtlety. This command would generate an error at compile time:

```
   Make/O root:$dfName:$wName       // ERROR
```

because Igor would intepret it as:

```
   Make/O root:$(dfName:$wName)     // ERROR
```

To avoid this, you must use parentheses like this:

```
   Make/O root:$(dfName):$wName     // OK
```

Runtime Lookup Example

In this example, a function named Routine calls another function named Subroutine and needs to access a number of result values created by Subroutine. To make it easy to clean up the temporary result globals, Subroutine creates them in a new data folder. Routine uses the results created by Subroutine and then deletes the temporary data folder.

```
Function Subroutine(w)
   WAVE w

   NewDataFolder/O/S SubroutineResults    // Results go here
```

```
   WaveStats/Q w           // WaveStats creates local variables
   Variable/G gAvg= V_avg  // Return the V_avg result in global gAvg
   Variable/G gMin= V_min
   String/G gWName= NameOfWave(w)

   SetDataFolder ::        // Back to original data folder
End

Function Routine()
   Make aWave= {1,2,3,4}
   Subroutine(aWave)
   NVAR theAvg= :SubroutineResults:gAvg   // theAvg is local name
   NVAR theMin= :SubroutineResults:gMin
   SVAR theName= :SubroutineResults:gWName
   Print theAvg,theMin,theName
   KillDataFolder SubroutineResults    // We are done with results
End
```

Note that the NVAR statement must appear *after* the procedure (Subroutine in this case) that creates the global variable. This is because NVAR has both a compile-time and a runtime behavior. At compile time, it creates a name that Igor can compile (theAvg in this case). At runtime, it actually looks up and creates a link to the global (variable gAvg stored in data folder SubroutineResults in this case).

Often a Function will need to access quite a large number of global variables stored in a data folder. In such cases, you can write more compact code using the ability of NVAR, SVAR or WAVE to access multiple objects in the current data folder as illustrated here:

```
Function Routine2()
   Make/O aWave= {1,2,3,4}
   Subroutine(aWave)

   String dfSav=GetDataFolder(1)
   SetDataFolder :SubroutineResults
   NVAR gAvg,gMin             // Access two variables via one NVAR
   SVAR gWName
   SetDataFolder dfSav

   Print gAvg,gMin,gWName
   KillDataFolder SubroutineResults    // We are done with results
End
```

Automatic Creation of NVAR and SVAR References

The Igor compiler sometimes automatically creates NVAR and SVAR references. For example:

```
Function Example1()
   Variable/G gVar1
   gVar1= 1

   String/G gStr1
   gStr1= "hello"
End
```

In this example we did not use NVAR or SVAR references and yet we were still able to compile assignment statements referring to global variables which will not exist until runtime. This is a feature of `Variable/G` and `String/G` that automatically create local references for simple object names.

Simple object names are names which are known at compile time for objects which will be created in the current data folder at runtime. `Variable/G` and `String/G` do not create references if you use $<name>, a partial data folder path or a full data folder path to specify the object.

Wave References

A wave reference is a lightweight object used in user-defined functions to specify the wave of interest in assignment statements and other commands. You can think of a wave reference as an identification number that Igor uses to identify a particular wave.

Wave reference variables hold wave references. They can be created as local variables, passed as parameters and returned as function results.

Here is a simple example:

```
Function Test(wIn)
   Wave wIn          // Reference to the input wave received as parameter

   String newName = NameOfWave(wIn) + "_out" // Compute output wave name

   Duplicate/O wIn, $newName                 // Create output wave

   Wave wOut = $newName       // Create wave reference for output wave
   wOut += 1                  // Use wave reference in assignment statement
End
```

This function might be called from the command line or from another function like this:

```
Make/O/N=5 wave0 = p
Test(wave0)                   // Pass wave reference to Test function
```

A Wave statement has both a compile-time and a runtime effect.

At compile time, it tells Igor what type of object the declared name references. In the example above, it tells Igor that wOut references a wave as opposed to a numeric variable, a string variable, a window or some other type of object. The Igor compiler allows wOut to be used in a waveform assignment statement (wOut += 1) because it knows that wOut references a wave.

The compiler also needs to know if the wave is real, complex or text. Use Wave/C to create a complex wave reference and Wave/T to create a text wave reference. Wave by itself creates a real wave reference.

At runtime the Wave statement stores a reference to a specific wave in the wave reference variable (wOut in this example). The referenced wave must already exist when the wave statement executes. Otherwise Igor will store a NULL reference in the wave reference variable and you will get an error when you attempt to use it. We put the "`Wave wOut = $newName`" statement *after* the Duplicate operation to insure that the wave exists when the Wave statement is executed. Putting the Wave statement before the command that creates the wave is a common error.

Automatic Creation of WAVE References

The Igor compiler sometimes automatically creates WAVE references. For example:

```
Function Example1()
   Make wave1
   wave1= x^2
End
```

In this example we did not use a WAVE reference and yet we were still able to compile an assignment statement referring to a wave which will not exist until runtime. This is a feature of the **Make** operation (see page V-366) which automatically creates local references for simple object names. The **Duplicate** operation (see page V-133) and many other operations that create output waves also automatically create local wave references for simple object names.

Simple object names are names which are known at compile time for objects which will be created in the current data folder at runtime. Make and Duplicate do not create references if you use $<name>, a partial

data folder path or a full data folder path to specify the object. However, as of Igor Pro 6.1, you can append /WAVE=<name> after a $ or path specification to explicitly create a WAVE reference.

In the case of Make and Duplicate with simple object names, the type of the automatically created wave reference (real, complex or text) is determined by flags. Make/C and Duplicate/C create complex wave references. Make/T and Duplicate/T create text wave references. Make and Duplicate without type flags create real wave references. See **WAVE Reference Types** on page IV-58 and **WAVE Reference Type Flags** on page IV-58 for a complete list of type flags and further details.

Most built-in operations that create output waves (often called "destination" waves) also automatically create wave references. For example, if you write:

```
DWT srcWave, destWave
```

it is as if you wrote:

```
DWT srcWave, destWave
WAVE destWave
```

After the discrete wavelet transform executes, you can reference `destWave` without an explicit wave reference.

Standalone WAVE Reference Statements

In cases where Igor does not automatically create a wave reference, because the output wave is not specified using a simple object name, you need to explicitly create a wave reference if you want to access the wave in an assignment statement.

You can create an explicit standalone wave reference using a statement following the command that created the output wave. In this example, the name of the output wave is specified as a parameter and therefore we can not use a simple object name when calling Make:

```
Function Example2(nameForOutputWave)
   String nameForOutputWave   // String contains the name of the wave to make

   Make $nameForOutputWave          // Make a wave
   Wave w = $nameForOutputWave      // Make a wave reference
   w= x^2
End
```

If you make a text wave or a complex wave, you need to tell the Igor compiler about that by using Wave/T or Wave/C. The compiler needs to know the type of the wave in order to properly compile the assignment statement.

Inline WAVE Reference Statements

In Igor Pro 6.1 or later, you can create a wave reference variable using /WAVE=<name> in the command that creates the output wave. For example:

```
Function Example3(nameForOutputWave)
   String nameForOutputWave

   Make $nameForOutputWave/Wave=w   // Make a wave and a wave reference
   w= x^2
End
```

Here /Wave=w is an inline wave reference statement. It does the same thing as the standalone wave reference in the preceding section.

Here are some more examples of inline wave declarations:

```
Function Example4()
   String name = "wave1"
   Duplicate/O wave0, $name/WAVE=wave1
```

```
   Differentiate wave0 /D=$name/WAVE=wave1
End
```

When using an inline wave reference statement, you do not need to, and in fact can not, specify the type of the wave using Wave/T or Wave/C. Just use Wave by itself regardless of the type of the output wave. The Igor compiler automatically creates the right kind of wave reference. For example:

```
Function Example5()
   Make real1, $"real2"/WAVE=r2     // real1, real2 and r2 are real
   Make/C cplx1, $"cplx2"/WAVE=c2   // cplx1, cplx2 and c2 are complex
   Make/T text1, $"text2"/WAVE=t2   // text1, text2 and t2 are text
End
```

Inline wave reference statements are accepted by those operations which automatically create a wave reference for a simple object name.

Inline wave references are not allowed after a simple object name.

Inline wave references are allowed on the command line but do nothing.

WAVE Reference Types

When WAVE references are created at compile time, they are created with a specific numeric type or are defined as text. The compiler then uses this type when creating expressions based on the WAVE reference or when trying to match two instances of the same name. For example:

```
Make rWave

Make/C cWave

Make/T tWave
```

creates single precision real, single precision complex, and text WAVE reference variables. These types then define what kind of right-hand side expression to compile:

```
rWave= expression       // will be real

cWave= expression       // will be complex

tWave= expression       // will be text
```

At the present time, these are the only three types of expressions that the compiler creates and numeric expressions are always evaluated in double precision. However, in the future, the compiler might be extended to create integer or single precision code. As a result of these considerations, the compiler is sometimes quite picky about the exact congruence between two declarations of WAVE reference variables of the same name (which create just a single instance). For example:

```
WAVE aWave
if( !WaveExists(aWave) )
   Make/D aWave
endif
```

will give a compile error complaining about inconsistent type for a WAVE reference. This is because the default type for the WAVE reference is single precision real. In this case, you will need to use the /D flag like so:

```
WAVE/D aWave
```

WAVE Reference Type Flags

The **WAVE** reference (see page V-722) along with certain operations such as Duplicate can accept the following flags identifying the type of WAVE reference:

/B	8-bit signed integer destination waves, unsigned with /U.
/C	complex destination waves.

/D	double precision destination waves.
/I	32-bit signed integer destination waves, unsigned with /U.
/S	single precision destination waves.
/T	text destination waves.
/U	unsigned destination waves.
/W	16-bit signed integer destination waves, unsigned with /U.
/DF	Wave holds data folder references. For advanced programmers only running Igor Pro 6.1 or later.
/WAVE	Wave holds wave references. For advanced programmers only running Igor Pro 6.1 or later.

These are the same flags used by the **Make** operation (see page V-366) except in this case they do not affect the actual wave and are used only specify what kind of wave is expected at runtime. This information is used if, later in the function, you create a wave assignment statement using a duplicated wave as the destination:

```
Function DupIt(wv)
   WAVE/C wv                  // complex wave

   Duplicate/O/C wv,dupWv     // dupWv is complex
   dupWv[0]=cmplx(5.0,1.0)    // no error, because dupWv known complex
   . . .
End
```

If Duplicate did not have the /C flag, you would get a "function not available for this number type" message when compiling the assignment of `dupWv` to the result of the `cmplx` function.

Problems with Automatic Creation of WAVE References

Operations that change a wave's type or which can create output waves of more than one type, such as **FFT**, **IFFT** and **WignerTransform** present special issues.

In some cases, the wave reference automatically created by an operation might be of the wrong type. For example, the FFT operation automatically creates a complex wave reference for the destination wave, so if you write:

```
FFT/DEST=destWave srcWave
```

it is as if you wrote:

```
FFT/DEST=destWave srcWave
WAVE/C destWave
```

However, if a real wave reference for the same name already exists, FFT/DEST will not create a new wave reference. For example:

```
Wave destWave                 // Real wave reference
. . .
FFT /DEST=destWave srcWave    // FFT does not create wave reference
                              // because it already exists.
```

In this case, you would need to create a complex wave reference using a different name, like this:

```
Wave/C cDestWave = destWave
```

The output of the FFT operation can sometimes be real, not complex. For example:

```
FFT/DEST=destWave/OUT=2 srcWave
```

The /OUT=2 flag creates a real destination wave. Thus the complex wave reference automatically created by the FFT operation is wrong and can not be used to subsequently access the destination wave. In this case, you must explicitly create a real wave reference, like this:

```
FFT/DEST=destWave/OUT=2 srcWave
WAVE realDestWave = destWave
```

Note that you can not write:

```
FFT/DEST=destWave/OUT=2 srcWave
WAVE destWave
```

because the FFT operation has already created a complex wave reference named `destWave`, so the compiler will generate an error. You must use a different name for the real wave reference.

The **IFFT** has a similar problem but in reverse. IFFT automatically creates a real wave reference for the destination wave. In some cases, the actual destination wave will be complex and you will need to create an explicit wave reference in order to access it.

WAVE Reference Is Not Needed to Pass a Wave Parameter

You don't need to use a WAVE reference when passing a literal wave name as a parameter to an operation or function. For example:

```
Function Test()
   Display jack
   Display root:jack
   Variable tmp = mean(jack,0,100)
End
```

In these examples, Igor knows that the parameters of the Display operation and the first parameter of the mean function are waves and therefore does not need the additional information provided by the WAVE reference.

Wave Reference Function Results

In Igor Pro 6.1 or later, advanced programmers can create functions that return wave references using Function/WAVE:

```
Function/WAVE Test(wIn) // /WAVE flag says function returns wave reference
   Wave wIn    // Reference to the input wave received as parameter

   String newName = NameOfWave(wIn) + "_out" // Compute output wave name

   Duplicate/O wIn, $newName                 // Create output wave

   Wave wOut = $newName       // Create wave reference for output wave
   wOut += 1                  // Use wave reference in assignment statement

   return wOut                // Return wave reference
End
```

This function might be called from another function like this:

```
Make/O/N=5 wave0 = p
Wave wOut = Test(wave0)
Display wave0, wOut
```

Wave Reference Waves

In Igor Pro 6.1 or later, you can create waves that contain wave references using the Make /WAVE flag. You can use a wave reference wave as a list of waves for further processing and in multithreaded wave assignment using the **MultiThread** keyword.

Wave reference waves are recommended for advanced programmers only.

Note: Wave reference waves are saved only in packed experiment files. They are not saved in unpacked experiments and are not saved by the SaveData operation or the Data Browser's Save Copy button. In general, they are intended for temporary computation purposes only.

Here is an example:

```
Make/O wave0, wave1, wave2              // Make some waves
Make/O/WAVE wr                          // Make a wave reference wave
wr[0]=wave0; wr[1]=wave1; wr[2]=wave2   // Assign values
```

The wave reference wave wr could now be used, for example, to pass a list of waves to a function that performs display or analysis operations.

Make/WAVE without any assignment creates a wave containing null wave references. Similarly, inserting points or redimensioning to a larger size initializes the new points to null. Deleting points or redimensioning to a smaller size deletes any free waves if the deleted points contained the only reference to them.

To determine if a given wave is a type that stores wave references, use the **WaveType** function with the optional selector = 1.

In the next example, a subroutine supplies a list of references to waves to be graphed by the main routine. A wave reference wave is used to store the list of wave references.

```
Function MainRoutine()
   Make/O/WAVE/N=5 wr      // Will contain references to other waves
   wr= Subroutine(p)       // Fill w with references

   WAVE w= wr[0]           // Get reference to first wave
   Display w               // and display in a graph

   Variable i
   for(i=1;i<5;i+=1)
      WAVE w= wr[i]        // Get reference to next wave
      AppendToGraph w      // and append to graph
   endfor
End

Function/WAVE Subroutine(i)
   Variable i

   String name = "wave"+num2str(i)

   // Create a wave with a computed name and also a wave reference to it
   Make/O $name/WAVE=w = sin(x/(8+i))

   return w       // Return the wave reference to the calling routine
End
```

For an example using a wave reference wave for multiprocessing, see **Wave Reference MultiThread Example** on page IV-286.

Data Folder References

The data folder reference is a lightweight object that refers to a data folder, analogous to the wave reference which refers to a wave. You can think of a data folder reference as an identification number that Igor uses to identify a particular data folder.

Data folder references require Igor Pro 6.1 or later.

Data folder reference variables (DFREFs) hold data folder references. They can be created as local variables, passed as parameters and returned as function results.

The most common use for a data folder reference is to save and restore the current data folder during the execution of a function:

```
Function Test()
   DFREF saveDFR = GetDataFolderDFR()  // Get reference to current data folder

   NewDataFolder/O/S MyNewDataFolder   // Create a new data folder
   . . .                               // Do some work in it

   SetDataFolder saveDFR               // Restore current data folder
End
```

Data folder references can be used in commands where Igor accepts a data folder path. For example, this function shows three equivalent methods of accessing waves in a specific data folder:

```
Function Test()
   // Method 1: Using paths
   Display root:Packages:'My Package':yWave vs root:Packages:'My Package':xWave

   // Method 2: Using the current data folder
   String dfSave = GetDataFolder(1) // Save the current data folder
   SetDataFolder root:Packages:'My Package'
   Display yWave vs xWave
   SetDataFolder dfSave             // Restore current data folder

   // Method 3: Using data folder references
   DFREF dfr = root:Packages:'My Package'
   Display dfr:yWave vs dfr:xWave
End
```

Using data folder references instead of data folder paths can streamline programs that make heavy use of data folders.

Using Data Folder References

In an advanced application, the programmer often defines a set of named data objects (waves, numeric variables and string variables) that the application acts on. These objects exist in a data folder. If there is just one instance of the set, it is possible to hard-code data folder paths to the objects. Often, however, there will be a multiplicity of such sets, for example, one set per graph or one set per channel in a data acquisition application. In such applications, procedures must be written to act on the set of data objects in a data folder specified at runtime.

One way to specify a data folder at runtime is to create a path to the data folder in a string variable. While this works, you wind up with code that does a lot of concatenation of data folder paths and data object names. Using data folder references, such code can be streamlined.

You create a data folder reference variable with a DFREF statement. For example, assume your application defines a set of data with a wave named wave0, a numeric variable named num0 and a string named str0 and that we have one data folder containing such a set for each graph. You can access your objects like this:

```
Function DoSomething(graphName)
   String graphName
   DFREF dfr = root:Packages:MyApplication:$graphName
   WAVE w0 = dfr:wave0
   NVAR n0 = dfr:num0
   SVAR s0 = dfr:str0
   . . .
End
```

Igor accepts a data folder reference in any command in which a data folder path would be accepted. For example:

```
Function Test()
   Display root:MyDataFolder:wave0  // OK

   DFREF dfr = root:MyDataFolder
   Display dfr:wave0                // OK

   String path = "root:MyDataFolder:wave0"
   Display $path                    // OK. $ converts string to path.

   path = "root:MyDataFolder"
   DFREF dfr = $path                // OK. $ converts string to path.
   Display dfr:wave0                // OK

   String currentDFPath
   currentDFPath = GetDataFolder(1) // OK
   DFREF dfr = GetDataFolder(1)     // ERROR: GetDataFolder returns a string
                                    // not a path.
End
```

The /SDFR Flag

In Igor Pro 6.20 or later you can also use the /SDFR (source data folder reference) flag in a WAVE, NVAR or SVAR statement. The utility of /SDFR is illustrated by this example which shows three different ways to reference multiple waves in the same data folder:

```
Function Test()
   // Assume a data folder exists at root:Run1

   // Use explicit paths
   Wave wave0=root:Run1:wave0, wave1=root:Run1:wave1, wave2=root:Run1:wave2

   // Use a data folder reference
   DFREF dfr = root:Run1
   Wave wave0=dfr:wave0, wave1=dfr:wave1, wave2=dfr:wave2

   // Use the /SDFR flag
   DFREF dfr = root:Run1
   Wave/SDFR=dfr wave0, wave1, wave2
End
```

The DFREF Type

In Functions, you can define data folder reference variables using the DFREF declaration:

```
DFREF localname [= <DataFolderRef or path>] [<more defs]
```

You can then use the data folder reference in those places where you can use a data folder path. For example:

```
DFREF dfr= root:df1
Display dfr:wave1             // Equivalent to Display root:df1:wave1
```

The syntax is limited to a single name after the data folder reference, so this is not legal:

```
Display dfr:subfolder:wave1   // Illegal
```

You can use DFREF to define input parameters for user-defined functions. For example:

```
Function Test(df)
   DFREF df
   Display df:wave1
End
```

You can also use DFREF to define fields in structures. However, you can not directly use a DFREF structure field in those places where Igor is expecting a path and object name. So, instead of:

```
Display s.dfr:wave1              // Illegal
```

you would need to write:

```
DFREF dftmp= s.dfr
Display dftmp:wave1              // OK
```

You can use a DFREF structure field where just a path is expected. For example:

```
SetDataFolder s.dfr              // OK
```

Built-in DFREF Functions

Some built-in functions take string data folder paths as parameters or return them as results. Those functions can not take or return data folder references. Here are equivalent DFREF versions that take or return data folder references:

```
GetDataFolderDFR()

GetIndexedObjNameDFR(dfr, type, index)

GetWavesDataFolderDFR(wave)

CountObjectsDFR(dfr, type)
```

These additional data folder reference functions are available:

```
DataFolderRefStatus(dfr)

NewFreeDataFolder()

DataFolderRefsEqual(dfr1, dfr2)
```

Just as operations that take a data folder path accept a data folder reference, these DFREF functions can also accept a data folder path:

```
Function Test()
   DFREF dfr = root:MyDataFolder
   Print CountObjectsDFR(dfr,1)                  // OK
   Print CountObjectsDFR(root:MyDataFolder,1)    // OK
End
```

Checking Data Folder Reference Validity

The **DataFolderRefStatus** function returns zero if the data folder reference is invalid. You should use it to test any DFREF variables that might not be valid, for example, when you assign a value to a data folder reference and you are not sure that the referenced data folder exists:

```
Function Test()
   DFREF dfr = root:MyDataFolder    // MyDataFolder may or may not exist
   if (DataFolderRefStatus(dfr) != 0)
      . . .
   endif
End
```

For historical reasons, an invalid DFREF variable will often act like root.

Data Folder Reference Function Results

A user-defined function can return a data folder reference. This might be used for a subroutine that returns a set of new objects to the calling routine. The set can be returned in a new data folder and the subroutine can return a reference it.

For example:

```
Function/DF Subroutine(newDFName)
   String newDFName
   NewDataFolder/O $newDFName
   DFREF dfr = $newDFName
   Make/O dfr:wave0, dfr:wave1
   return dfr
End

Function/DF MainRoutine()
   DFREF dfr = Subroutine("MyDataFolder")
   Display dfr:wave0, dfr:wave1
End
```

Data Folder Reference Waves

In Igor Pro 6.1 or later, you can create waves that contain data folder references using the **Make** /DF flag. You can use a data folder reference wave as a list of data folders for further processing and in multithreaded wave assignment using the MultiThread keyword.

Data folder reference waves are recommended for advanced programmers only.

Note: Data folder reference waves are saved only in packed experiment files. They are not saved in unpacked experiments and are not saved by the SaveData operation or the Data Browser's Save Copy button. In general, they are intended for temporary computation purposes only.

Make/DF without any assignment creates a wave containing null data folder references. Similarly, inserting points or redimensioning to a larger size initializes the new points to null. Deleting points or redimensioning to a smaller size deletes any free data folders if the wave contained the only reference to them.

To determine if a given wave is a type that stores data folder references, use the **WaveType** function with the optional selector = 1.

Although programmers may find multiple uses of data folder reference waves, the main impetus for their development was to make it easier to improve calculation performance by multithreading on multi-core machines.

For an example using a data folder reference wave for multiprocessing, see **Data Folder Reference Multi-Thread Example** on page IV-285.

Accessing Waves in Functions

To access a wave, we need to create, one way or another, a wave reference. The section **Accessing Global Variables and Waves** on page IV-50 explained how to access a wave using a WAVE reference. This section introduces several additional techniques. Other useful details may be found in **Returning Created Waves from User Functions** on page III-142.

We can create the wave reference by:

- Declaring a wave parameter
- Using $<string expression>
- Using a literal wave name
- Using a wave reference function

Each of these techniques is illustrated in the following sections.

Each example shows a function and commands that call the function. The function itself illustrates how to deal with the wave within the function. The commands show how to pass enough information to the function so that it can access the wave. Other examples can be found in **Writing Functions that Process Waves** on page III-143.

Wave Reference Passed as Parameter

This is the simplest method. The function might be called from the command line or from another function.

```
Function Test(w)
   WAVE w                  // Wave reference passed as a parameter

   w += 1                  // Use in assignment statement
   Print mean(w,-INF,INF)  // Pass as function parameter
   WaveStats w             // Pass as operation parameter
   AnotherFunction(w)      // Pass as user function parameter
End

Make/O/N=5 wave0 = p
Test(wave0)
Test($"wave0")
String wName = "wave0"; Test($wName)
```

In the first call to Test, the wave reference is a literal wave name. In the second call, we create the wave reference using $<literal string>. In the third call, we create the wave reference using $<string variable>. $<literal string> and $<string variable> are specific cases of the general case $<string expression>.

If the function expected to receive a reference to a text wave, we would declare the parameter using:

```
   WAVE/T w
```

If the function expected to be receive a reference to a complex wave, we would declare the parameter using:

```
   WAVE/C w
```

If you need to return a large number of values to the calling routine, it is sometimes convenient to use a parameter wave as an output mechanism. The following example illustrates this technique:

```
Function MyWaveStats(inputWave, outputWave)
   WAVE inputWave
   WAVE outputWave

   WaveStats/Q inputWave

   outputWave[0] = V_npnts
   outputWave[1] = V_avg
   outputWave[2] = V_sdev
End

Function Test()
   Make/O testwave= gnoise(1)

   Make/O/N=20 tempResultWave
   MyWaveStats(testwave, tempResultWave)
   Variable npnts = tempResultWave[0]
   Variable avg = tempResultWave[1]
   Variable sdev = tempResultWave[2]
   KillWaves tempResultWave

   Printf "Points: %g; Avg: %g; SDev: %g\r", npnts, avg, sdev
End
```

Wave Accessed Via String Passed as Parameter

This technique is of most use when the wave might not exist when the function is called. It is appropriate for functions that create waves.

```
Function Test(wName)
   String wName            // String containing a name for wave

   Make/O/N=5 $wName
   WAVE w = $wName         // Create a wave reference
   Print NameOfWave(w)
```

```
End

Test("wave0")
```

This example will create wave0 if it does not yet exist or overwrite it if it does exist. If we knew that the wave had to already exist, we could and should use the wave parameter technique shown in the preceding section. In this case, since the wave may not yet exist, we can not use a wave parameter.

Notice that we create a wave reference immediately after making the wave. Once we do this, we can use the wave reference in all of the ways shown in the preceding section. We can not create the wave reference before making the wave because a wave reference must refer to an existing wave.

The following command demonstrates that $wName and the wave reference w can refer to a wave that is not in the current data folder.

```
NewDataFolder root:Folder1; Test("root:Folder1:wave0")
```

Wave Accessed Via String Calculated in Function

This technique is used when creating multiple waves in a function or when algorithmically selecting a wave or a set of waves to be processed.

```
Function Test(baseName, startIndex, endIndex)
   String baseName
   Variable startIndex, endIndex

   Variable index = startIndex
   do
      WAVE w = $(baseName + num2istr(index))
      WaveStats/Q w
      Printf "Wave: %s; average: %g\r", NameOfWave(w), V_avg
      index += 1
   while (index <= endIndex)
End

Make/O/N=5 wave0=gnoise(1), wave1=gnoise(1), wave2=gnoise(1)
Test("wave", 0, 2)
```

We need to use this method because we want the function to operate on any number of waves. If the function were to operate on a small, fixed number of waves, we could use the wave parameter method.

As in the preceding section, we create a wave reference using $<string expression>.

Wave Accessed Via Literal Wave Name

In data acquisition or analysis projects, you often need to write procedures that deal with runs of identically-structured data. Each run is stored in its own data folder and contains waves with the same names. In this kind of situation, you can write a set of functions that use literal wave names specific for your data structure.

```
Function CreateRatio()
   WAVE dataA, dataB
   Duplicate dataA, ratioAB
   WAVE ratioAB
   ratioAB = dataA / dataB
End

Make/O/N=5 dataA = 1 + p, dataB = 2 + p
CreateRatio()
```

The CreateRatio function assumes the structure and naming of the data. The function is hard-wired to this naming scheme and assumes that the current data folder contains the appropriate data.

Wave Accessed Via Wave Reference Function

A wave reference function is a built-in Igor function that returns a reference to a wave. Wave reference functions are typically used on the right-hand side of a WAVE statement. For example:

```
WAVE w = WaveRefIndexed("",i,4)
```

A common use for a wave reference function is to get access to waves displayed in a graph, using the TraceNameToWaveRef function. Here is an example.

```
Function PrintAverageOfDisplayedWaves()
   String list, traceName

   list = TraceNameList("",";",1)   // List of traces in top graph
   Variable index = 0
   do
      traceName = StringFromList(index, list)   // Next trace name
      if (strlen(traceName) == 0)
         break                                  // No more traces
      endif
      WAVE w = TraceNameToWaveRef("", traceName)// Get wave ref
      WaveStats/Q w
      Printf "Wave: %s; average: %g\r", NameOfWave(w), V_avg
      index += 1
   while (1)               // loop till break above
End

Make/O/N=5 wave0=gnoise(1), wave1=gnoise(1), wave2=gnoise(1)
Display wave0, wave1, wave2
PrintAverageOfDisplayedWaves()
```

There are other wave reference functions (see **Wave Reference Functions** on page IV-173), but WaveRefIndexed and TraceNameToWaveRef are the most used.

Destination Wave Parameters

Many operations create waves. Examples are Make, Duplicate and Differentiate. Such operations take "destination wave" parameters. A destination wave parameter can be:

A simple name

```
Differentiate fred /D=jack
```

A path

```
Differentiate fred /D=root:FolderA:jack
```

$ followed by a string expression

```
String str = "root:FolderA:jack"
Differentiate fred /D=$str
```

A simple wave reference

```
Wave/Z jack
Differentiate fred /D=jack
```

Other wave references

```
Wave/Z w = jack
Differentiate fred /D=w

Wave/Z w = root:FolderA:jack
Differentiate fred /D=w

Wave/Z jack = root:FolderA:jack
Differentiate fred /D=jack

STRUCT MyStruct s// Contains wave ref field w
Differentiate fred /D=s.w
```

The wave reference works only in a user-defined function. The other techniques work in functions, in macros and from the command line.

The first and fourth cases are actually the same because, if you use a simple name, the Igor compiler automatically creates a wave reference as if you had used a simple wave reference as explained under **Automatic Creation of WAVE References** on page IV-56.

Using the first four techniques, the destination wave may or may not already exist. It will be created if it does not exist and overwritten if it does exist.

In the last technique, the name of the wave reference is different from the name of the destination wave or the wave reference points to a data folder other than the current data folder or both. This technique works properly only if the referenced wave already exists. Otherwise the destination wave is a wave with the name of the wave reference itself (w in the first two examples, jack in the last) in the current data folder. This situation is further explained below under **Destination Wave Reference Issues** on page IV-70.

Wave Reference as Destination Wave

Here are the rules for wave references when used as destination waves:

1. If a simple name (not a wave reference) is passed as the destination wave parameter, the destination wave is a wave of that name in the current data folder whether it exists or not.
2. If a path or $ followed by a string containing a path is passed as the destination wave parameter, the destination wave is specified by the path whether the specified wave exists or not.
3. If a wave reference is passed as the destination wave parameter, the destination wave is the referenced wave if it exists. See **Destination Wave Reference Issues** on page IV-70 for what happens if it does not exist.

Exceptions To Destination Wave Rules

The Make operation is an exception in regard to wave references. The following statements make a wave named w in the current data folder whether root:FolderA:jack exists or not:

```
Wave/Z w = root:FolderA:jack
Make/O w
```

Prior to Igor Pro 6.20, many operations behaved like Make in this regard. In Igor Pro 6.20 and later, most operations behave like Differentiate.

Updating of Destination Wave References

When a simple name is provided in a user-defined function, Igor automatically creates a wave reference variable of that name at compile time if one does not already exist. This is an "implicit" wave reference.

When a wave reference variable exists, whether implicit or explicit, during the execution of the command, the operation stores a reference to the destination wave in that wave reference variable. You can use the wave reference to access the destination wave in subsequent commands.

Similarly, when the destination is specified using a wave reference field in a structure, the operation updates the field to refer to the destination wave.

Inline Wave References With Destination Waves

When the destination wave is specified using a path or a string containing a path, you can use an inline wave reference to create a reference to the destination wave. For example:

```
String str = "root:FolderA:jack"
Differentiate $str /D=$dest/WAVE=wDest
Display wDest
```

Here the Igor compiler creates a wave reference named wDest. The Differentiate operation stores a reference to the destination wave (root:FolderA:jack in this case) in wDest which can then be used to access the destination wave in subsequent commands.

Inline wave references do not determine which wave is the destination wave but merely provide a wave reference pointing to the destination wave when the command finishes.

Destination Wave Reference Issues

You will get unexpected behavior when a wave reference variable refers to a wave with a different name or in a different data folder and the referenced wave does not exist. For example, if the referenced wave does not exist:

```
Wave/Z w = jack
Differentiate fred /D=w    // Creates a wave named w in current data folder

Wave/Z w = root:FolderA:jack
Differentiate fred /D=w    // Creates a wave named w in current data folder

Wave/Z jack = root:FolderA:jack
Differentiate fred /D=jack // Creates a wave named jack in current data folder

STRUCT MyStruct s          // Contains wave ref field w
Differentiate fred /D=s.w  // Creates a wave named w in current data folder
```

In a situation like this, you should add a test using WaveExists to verify that the destination wave is valid and throw an error if not or otherwise handle the situation. For example:

```
Wave/Z w = root:FolderA:jack
if (!WaveExists(w))
   Abort "Destination wave does not exist"
endif
Differentiate fred /D=w
```

As noted above, when you use a simple name as a destination wave, the Igor compiler automatically creates a wave reference. If the automatically-created wave reference conflicts with a pre-existing wave reference, the compiler will generate an error. For example, this function generates an "inconsistent type for wave reference error":

```
Function InconsistentTypeError()
   Wave/C w                   // Explicit complex wave reference
   Differentiate fred /D=w    // Implicit real wave reference
End
```

Another consideration involves loops. Suppose in a loop you have code like this:

```
   SetDataFolder <something depending on loop index>
   Duplicate/O srcWave, jack
```

You may think you are creating a wave named jack in each data folder but, because the contents of the automatically-created wave refrence variable jack is non-null after the first iteration, you will simply be overwriting the same wave over and over. To fix this, use

```
   Duplicate/O srcWave,$"jack"/WAVE=jack
```

This will create a wave named jack in the current data folder and store a reference to it in a wave reference variable also named jack.

Changes in Destination Wave Behavior

Igor's handling of destination wave references was improved for Igor Pro 6.20. Previously some operations treated wave references as simple names, did not set the wave reference to refer to the destination wave on output, and exhibited other non-standard behavior.

Trace Name Parameters

The ModifyGraph operation takes trace name parameters. A trace name is not the same as a wave. An example may clarify this subtle point:

```
Function Test()
   Wave w = root:FolderA:wave0
   Display w
   ModifyGraph rgb(w) = (65535,0,0)          // WRONG
End
```

This is wrong because ModifyGraph is looking for the name of a trace in a graph and w is not the name of a trace in a graph. The name of the trace in this case is wave0. The function should be written like this:

```
Function Test()
   Wave w = root:FolderA:wave0
   Display w
   ModifyGraph rgb(wave0) = (65535,0,0)      // CORRECT
End
```

In the next example, the wave is passed to the function as a parameter so the name of the trace is not so obvious:

```
Function Test(w)
   Wave w
   Display w
   ModifyGraph rgb(w) = (65535,0,0)          // WRONG
End
```

This is wrong for the same reason as the first example: w is not the name of the trace. The function should be written like this:

```
Function Test(w)
   Wave w
   Display w

   String name = NameOfWave(w)
   ModifyGraph rgb($name) = (65535,0,0)      // CORRECT
End
```

User-defined Trace Names

As of Igor Pro 6.20, you can provide user-defined names for traces using /TN=<name> with **Display** and **AppendToGraph**. For example:

```
Make/O jack=sin(x/8)
NewDataFolder/O foo; Make/O :foo:jack=sin(x/9)
NewDataFolder/O bar; Make/O :bar:jack=sin(x/10)
Display jack/TN='jack in root', :foo:jack/TN='jack in foo'
AppendToGraph :bar:jack/TN='jack in bar'
ModifyGraph mode('jack in bar')=7,hbFill('jack in bar')=6
ModifyGraph rgb('jack in bar')=(0,0,65535)
```

Warning: Using this feature will make your experiment hard to load into versions prior to 6.20. Also, any code that assumes the trace name contains the wave name will fail.

Free Waves

Free waves are waves that are not part of any data folder hierarchy. Their principal use is in multithreaded wave assignment using the MultiThread keyword in a function. They can also be used for temporary storage within functions.

Free waves are recommended for advanced programmers only.

Free waves require Igor Pro 6.1 or later.

Note: Free waves are saved only in packed experiment files. They are not saved in unpacked experiments and are not saved by the SaveData operation or the Data Browser's Save Copy button. In general, they are intended for temporary computation purposes only.

You create free waves using the **NewFreeWave** function and the **Make**/FREE and **Duplicate**/FREE operations.

Here is an example:

```
Function ReverseWave(w)
   Wave w

   Variable lastPoint = numpnts(w) - 1
   if (lastPoint > 0)
      // This creates a free wave named wTemp.
      // It also creates an automatic wave reference named wTemp
      // which refers to the free wave.
      Duplicate /FREE w, wTemp

      w = wTemp[lastPoint-p]
   endif
End
```

In this example, wTemp is a free wave. As such, it is not contained in any data folder and therefore can not conflict with any other wave.

As explained below under **Free Wave Lifetime** on page IV-73, a free wave is automatically deleted when there are no more references to it. In this example that happens when the function ends.

You can access a free wave only using the wave reference returned by NewFreeWave, Make/FREE or Duplicate/FREE.

Free waves can not be used in situations where global persistence is required such as in graphs, tables and controls. In other words, you should use free waves for computation purposes only.

For a discussion of multithreaded assignment statements, see **Automatic Parallel Processing with MultiThread** on page IV-283. For an example using free waves, see **Wave Reference MultiThread Example** on page IV-286.

Free Wave Created When Free Data Folder Is Deleted

In addition to the explicit creation of free waves, a wave that is created in a free data folder becomes free if the host data folder is deleted but a reference to the wave still exists. For example:

```
Function/WAVE Test()
   DFREF dfSav= GetDataFolderDFR()

   SetDataFolder NewFreeDataFolder()
   Make jack={1,2,3}// jack is not a free wave at this point.
   // This also creates an automatic wave reference named jack.

   SetDataFolder dfSav
   // The free data folder is now gone but jack persists
   // because of the wave reference to it and is now a free wave.

   return jack
End
```

Free Wave Created For User Function Input Parameter

If a user function takes a wave reference as in input parameter, you can create and pass a short free wave using a list of values as illustrated here:

```
Function Test(w)
   WAVE/D w
   Print w
End
```

You can invoke this function like this:

```
Test({1,2,3})
```

Igor automatically creates a free wave and passes it to Test. The free wave is automatically deleted when Test returns.

The data type of the free wave is determined by the type of the function's wave parameter. In this case the free wave is created as double-precision floating point because the wave parameter is defined using the /D flag. If /D were omitted, the free wave would be single-precision floating point. Wave/C/D would give a double-precision complex free wave. Wave/T would give a text free wave.

This list of values syntax is allowed only for user-defined functions because only they have code to test for and delete free waves upon exit.

Free Wave Lifetime

A free wave is automatically deleted when the last reference to it disappears.

Wave references can be stored in:

1. Wave reference variables in user-defined functions
2. Wave reference fields in structures
3. Elements of a wave reference wave (created by Make/WAVE)

The first case is the most common.

A wave reference disappears when:

1. The wave reference variable containing it is explicitly cleared using WaveClear.
2. The wave reference variable containing it is reassigned to refer to another wave.
3. The wave reference variable containing it goes out-of-scope and ceases to exist when the function in which it was created returns.
4. The wave reference wave element containing it is deleted or the wave reference wave is killed.

When there are no more references to a free wave, Igor automatically deletes it. This example illustrates the first three of these scenarios:

```
Function TestFreeWaveDeletion1()
   Wave w = NewFreeWave(2,3)  // Create a free wave with 3 points
   WaveClear w                // w no longer refers to the free wave
   // There are no more references to the free wave so it is deleted

   Wave w = NewFreeWave(2,3)  // Create a free wave with 3 points
   Wave w = root:wave0        // w no longer refers to the free wave
   // There are no more references to the free wave so it is deleted

   Wave w = NewFreeWave(2,3)  // Create a free wave with 3 points
End   // Wave reference w ceases to exist so the free wave is deleted
```

In the preceding example we used NewFreeWave which creates a free wave named 'f' that is not part of any data folder. Next we will use Make/FREE instead of NewFreeWave. Using Make allows us to give the free

wave a name of our choice but it is still free and not part of any data folder. When reading this example, keep in mind that "Make jack" creates an automatic wave reference variable named jack:

```
Function TestFreeWaveDeletion2()
   Make /D /N=3 /FREE jack    // Create a free DP wave with 3 points
   // Make created an automatic wave reference named jack

   Make /D /N=5 /FREE jack    // Create a free DP wave with 5 points
   // Make created an automatic wave reference named jack
   // which refers to the 5-point jack.
   // There are now no references to 3-point jack so it is automatically deleted.

End   // Wave reference jack ceases to exist so free wave jack is deleted
```

In the next example, a subroutine returns a reference to a free wave to the calling routine:

```
Function/WAVE Subroutine1()
   Make /D /N=3 /FREE jack=p  // Create a free DP wave with 3 points
   return jack                // Return reference to calling routine
End

Function MainRoutine1()
   WAVE w= Subroutine1()   // Wave reference w references the free wave jack
   Print w
End   // Wave reference w ceases to exist so free wave jack is deleted
```

In the final example, the wave jack starts as an object in a free data folder (see **Free Data Folders** on page IV-75). It is not free because it is part of a data folder hierarchy even though the data folder is free. When the free data folder is deleted, jack becomes a free wave.

When reading this example, keep in mind that the free data folder is automatically deleted when there are no more references to it. Originally, it survives because it is the current data folder and therefore is referenced by Igor internally. When it is no longer the current data folder, there are no more references to it and it is automatically deleted:

```
Function/WAVE Subroutine2()
   DFREF dfSav= GetDataFolderDFR()

   // Create free data folder and set as current data folder
   SetDataFolder NewFreeDataFolder()

   // Create wave jack and an automatic wave reference to it
   Make jack={1,2,3}    // jack is not free - it is an object in a data folder

   SetDataFolder dfSav  // Change the current data folder
   // There are now no references to the free data folder so it is deleted
   // but wave jack remains because there is a reference to it.
   // jack is now a free wave.

   return jack          // Return reference to free wave to calling routine
End

Function MainRoutine2()
   WAVE w= Subroutine2()   // Wave reference w references free wave jack
   Print w
End   // Wave reference w ceases to exist so free wave jack is deleted
```

Not shown in this section is the case of a free wave that persists because a reference to it is stored in a wave reference wave. That situation is illustrated by the **Automatic Parallel Processing with MultiThread** on page IV-283 example.

Converting a Free Wave to a Global Wave

You can use **MoveWave** to move a free wave into a global data folder, in which case it ceases to be free. If the wave was created by NewFreeWave its name will be 'f'. You can use MoveWave to provide it with a more descriptive name.

Here is an example illustrating Make/FREE:

```
Function Test()
   Make/FREE/N=(50,50) w
   SetScale x,-5,8,w
   SetScale y,-7,12,w
   w= exp(-(x^2+y^2))
   NewImage w
   if( GetRTError(1) != 0 )
      Print "Can't use a free wave here"
   endif
   MoveWave w,root:wasFree
   NewImage w
End
```

Note that MoveWave requires that the new wave name, wasFree in this case, be unique within the destination data folder.

To determine if a given wave is free or global, use the **WaveType** function with the optional selector = 2.

Free Data Folders

Free data folders are data folders that are not part of any data folder hierarchy. Their principal use is in multithreaded wave assignment using the MultiThread keyword in a function. They can also be used for temporary storage within functions.

Free data folders are recommended for advanced programmers only.

Free data folders require Igor Pro 6.1 or later.

Note: Free data folders are saved only in packed experiment files. They are not saved in unpacked experiments and are not saved by the SaveData operation or the Data Browser's Save Copy button. In general, they are intended for temporary computation purposes only.

You create a free data folder using the NewFreeDataFolder function. You access it using the data folder reference returned by that function.

After using SetDataFolder with a free data folder, be sure to restore it to the previous value, like this:

```
Function Test()
   DFREF dfrSave = GetDataFolderDFR()

   SetDataFolder NewFreeDataFolder()   // Create new free data folder.

   . . .

   SetDataFolder dfrSave
End
```

This is good programming practice in general but is especially important when using free data folders.

Free data folders can not be used in situations where global persistence is required such as in graphs, tables and controls. In other words, you should use objects in free data folders for short-term computation purposes only.

For a discussion of multithreaded assignment statements, see **Automatic Parallel Processing with MultiThread** on page IV-283. For an example using free data folders, see **Data Folder Reference MultiThread Example** on page IV-285.

Free Data Folder Lifetime

A free data folder is automatically deleted when the last reference to it disappears.

Data folder references can be stored in:

1. Data folder reference variables in user-defined functions
2. Data folder reference fields in structures
3. Elements of a data folder reference wave (created with Make/DF)
4. Igor's internal current data folder reference variable

A data folder reference disappears when:

1. The data folder reference variable containing it is explicitly cleared using KillDataFolder.
2. The data folder reference variable containing it is reassigned to refer to another data folder.
3. The data folder reference variable containing it goes out-of-scope and ceases to exist when the function in which it was created returns.
4. The data folder reference wave element containing it is deleted or the data folder reference wave is killed.
5. The current data folder is changed which causes Igor's internal current data folder reference variable to refer to another data folder.

When there are no more references to a free data folder, Igor automatically deletes it.

In this example, a free data folder reference variable is cleared by KillDataFolder:

```
Function Test1()
   DFREF dfr= NewFreeDataFolder()      // Create new free data folder.
   // The free data folder exists because dfr references it.
   . . .
   KillDataFolder dfr   // dfr no longer refers to the free data folder
End
```

KillDataFolder kills the free data folder only if the given DFREF variable contains the last reference to it.

In the next example, the free data folder is automatically deleted when the DFREF that references it is changed to reference another data folder:

```
Function Test2()
   DFREF dfr= NewFreeDataFolder()      // Create new free data folder.
   // The free data folder exists because dfr references it.
   . . .
   DFREF dfr= root:
   // The free data folder is deleted since there are no references to it.
End
```

In the next example, a free data folder is created and a reference is stored in a local data folder reference variable. When the function ends, the DFREF ceases to exist and the free data folder is automatically deleted:

```
Function Test3()
   DFREF dfr= NewFreeDataFolder()      // Create new free data folder.
   // The free data folder exists because dfr references it.
   . . .
End   // The free data folder is deleted because dfr no longer exists.
```

The fourth case, where a data folder reference is stored in a data folder reference wave, is discussed under **Data Folder Reference Waves** on page IV-65.

In the next example, the free data folder is referenced by Igor's internal current data folder reference variable because it is the current data folder. When the current data folder is changed, there are no more references to the free data folder and it is automatically deleted:

```
Function Test4()
   SetDataFolder NewFreeDataFolder()   // Create new free data folder.
   // The free data folder persists because it is the current data folder
   // and therefore is referenced by Igor's internal
   // current data folder reference variable.
   . . .

   // Change Igor's internal current data folder reference
   SetDataFolder root:
   // The free data folder is since there are no references to it.
End
```

Free Data Folder Objects Lifetime

Next we consider what happens to objects in a free data folder when the free data folder is deleted. In this event, numeric and string variables in the free data folder are unconditionally automatically deleted. A wave is automatically deleted if there are no wave references to it. If there is a wave reference to it, the wave survives and becomes a free wave. Free waves are waves that exists outside of any data folder as explained under **Free Waves** on page IV-71.

For example:

```
Function Test()
   SetDataFolder NewFreeDataFolder()   // Create new free data folder.
   // The free data folder exists because it is the current data folder.

   Make jack      // Make a wave and an automatic wave reference

   . . .

   SetDataFolder root:
   // The free data folder is deleted since there are no references to it.
   // Because there is a reference to the wave jack, it persists
   // and becomes a free wave.

   . . .

End   // The wave reference to jack ceases to exist so jack is deleted
```

When this function ends, the reference to the wave jack ceases to exist, there are no references to jack, and it is automatically deleted.

Next we look at a slight variation. In the following example, Make does not create an automatic wave reference because of the use of $, and we do not create an explicit wave reference:

```
Function Test()
   SetDataFolder NewFreeDataFolder()   // Create new free data folder.
   // The free data folder exists because it is the current data folder.

   Make $"jack"         // Make a wave but no wave reference
   // jack persists because the current data folder references it.

   . . .

   SetDataFolder root:
   // The free data folder is since there are no references to it.
   // jack is also deleted because there are no more references to it.
```

```
   . . .

End
```

Converting a Free Data Folder to a Global Data Folder

You can use **MoveDataFolder** to move a free data folder into the global hierarchy. The data folder and all of its contents then become global. The name of a free data folder created by NewFreeDataFolder is 'free-root'. You should rename it after moving to a global context. For example:

```
Function Test()
   DFREF saveDF = GetDataFolderDFR()
   DFREF dfr = NewFreeDataFolder()     // Create free data folder
   SetDataFolder dfr                   // Set as current data folder
   Make jack=sin(x/8)                  // Create some data in it
   SetDataFolder saveDF                // Restore original current data folder
   MoveDataFolder dfr, root:           // Free DF becomes root:freeroot
   RenameDataFolder root:freeroot,TestDF  // Rename with a proper name
   Display root:TestDF:jack
End
```

Note that MoveDataFolder requires that the data folder name, freeroot in this case, be unique within the destination data folder.

Structures in Functions

You can define structures in procedure files and use them in functions. Structures can be used only in user-defined functions as local variables and their behavior is defined almost entirely at compile time. Runtime or interactive definition and use of structures is not currently supported (for this purpose, use Data Folders (see Chapter II-8, **Data Folders**), the **StringByKey** function (see page V-675), or the **NumberByKey** function (see page V-458)).

Use of structures is an advanced technique. If you are just starting with Igor programming, you may want to skip this section and come back to it later.

Defining Structures

Structures are defined in a procedure file with the following syntax:

```
Structure structureName
   memType memName [arraySize] [, memName [arraySize]]
   …
EndStructure
```

Structure member types (*memType*) can be any of the following Igor objects: Variable, String, WAVE, NVAR, SVAR, DFREF, FUNCREF, or STRUCT. The DFREF object type requires Igor Pro 6.1 or later.

Igor structures also support additional member types, as given in the next table, for compatibility with C programming structures and disk files.

Igor Member Type	C Equivalent	Byte Size
char	signed char	1
uchar	unsigned char	1
int16	short int	2
uint16	unsigned short int	2
int32	long int	4
uint32	unsigned long int	4

Igor Member Type	C Equivalent	Byte Size
float	float	4
double	double	8

The Variable and double types are identical although Variable can be also specified as complex (using the /C flag).

The optional *arraySize* must be specified using an expression involving literal numbers or locally-defined constants enclosed in brackets. The value must be between 1 and 400 for all but STRUCT where the upper limit is 100. The upper limit is a consequence of how memory is allocated on the stack frame.

Structures are two-byte aligned. This means that if an odd number of bytes has been allocated and then a nonchar field is defined, an extra byte of padding will be inserted in front of the new field. This is mainly of concern only when reading and writing structures from and to disk files.

The Structure keyword can be preceded with the **Static** keyword (see page V-594) to make the definition apply only to the current procedure window. Without the Static designation, structures defined in one procedure window may be used in any other.

Using Structures

To use ("instantiate") a structure in a function, you must allocate a STRUCT variable using:

```
STRUCT sName name
```

where *sName* is the name (or "tag" if you're a C programmer) of an existing structure and *name* is the local variable name (sometimes called the "instance name"). (See **STRUCT** on page V-678 for details.)

To access a member of a structure, specify the STRUCT variable name followed by a "." and the member name:

```
STRUCT Point pt
pt.v= 100
```

When a member is defined as an array:

```
Structure mystruct
   Variable var1
   Variable var2[10]
   …
EndStructure
```

you must specify `[index]` to use a given element in the array:

```
STRUCT mystruct ms
ms.var2[n]= 22
```

Structure and field names must be literal and can not use `$str` notation. The *index* value can be calculated at runtime (it doesn't have to be a literal number). If the field is itself a STRUCT, continue to append "." and field names as needed. See the **Example** on page IV-80.

You can define an array of structures as a field in a structure:

```
Structure mystruct
   STRUCT Point pt[100]    // Allowed as a sub-structure
EndStructure
```

However, you can not define an array of structures as a local variable in a function:

```
STRUCT Point pt[100]       // Not allowed as a function local variable
```

WAVE, NVAR, SVAR, and FUNCREF members can be initialized using the same syntax as used for the corresponding nonstructure variables. See **Runtime Lookup of Globals** on page IV-50 and **Function References** on page IV-84.

Structures may be passed to functions **only** by reference (as with Fortran), which allows them to be both input and output parameters (see **Pass-By-Reference** on page IV-45). The syntax is:

```
STRUCT sName  &varName
```

In a user function you define the input parameter:

```
Function myFunc(s)
   STRUCT mystruct &s
   …
End
```

Char and uchar arrays can be treated as zero-terminated strings by leaving off the brackets. Because the Igor compiler knows the size, the entire array can be used with no zero termination. Like normal string variables, concatenation using += is allowed but substring assignment using `[p1,p2]= subStr` is not supported.

Structures, including substructures, may be copied using simple assignment from one structure to the other. The source and destination structures must defined using the same structure name (tag).

The **Print** operation (see page V-498) can print individual elements of a structure or can print a summary of the entire STRUCT variable.

Example

Here is a (somewhat contrived) example using structures. Try executing `foo(2)`:

```
Constant kCaSize= 5

Structure substruct
   Variable v1
   Variable v2
EndStructure

Structure mystruct
   Variable var1
   Variable var2[10]
   String s1
   WAVE fred
   NVAR globVar1
   SVAR globStr1
   FUNCREF myDefaultFunc afunc
   STRUCT substruct ss1[3]
   char ca[kCaSize+1]
EndStructure

Function foo(n)
   Variable n

   Make/O/N=20 fred
   Variable/G globVar1= 111
   String/G aGlobStr="a global string var"

   STRUCT mystruct ms
   ms.var1= 11
   ms.var2[n]= 22
   ms.s1= "string s1"
   WAVE ms.fred     // could have =name if want other than waves named fred
   NVAR ms.globVar1
   SVAR ms.globStr1= aGlobStr
   FUNCREF myDefaultFunc ms.afunc= anotherfunc
   ms.ss1[n].v1= ms.var1/2
   ms.ss1[0]= ms.ss1[n]
   ms.ca= "0123456789"
   bar(ms,n)
   print ms.var1,ms.var2[n],ms.s1,ms.globVar1,ms.globStr1,ms.ss1[n].v1
   print ms.ss 1[n].v2,ms.ca,ms.afunc()
   print "a whole wave",ms.fred
   print "the whole ms struct:",ms

   STRUCT substruct ss
   ss= ms.ss1[n]
```

```
   print "copy of substruct",ss
End

Function bar(s,n)
   STRUCT mystruct &s
   Variable n

   s.ss1[n].v2= 99
   s.fred= sin(x)
   Display s.fred
End

Function myDefaultFunc()
   return 1
End

Function anotherfunc()
   return 2
End
```

Note the use of WAVE, NVAR, SVAR and FUNCREF in the function foo. These keywords are required both in the structure definition and again in the function, when the structure members are initialized.

Built-In Structures

Igor includes a few special purpose, predefined structures for use with certain operations. Some of those structures use these predefined general purpose structures:

```
Structure Rect
   Int16 top,left,bottom,right
EndStructure

Structure Point
   Int16 v,h
EndStructure

Structure RGBColor
   UInt16 red, green, blue
EndStructure
```

A number of operations use built-in structures that the Igor programmer can use. See the command reference information for details about these structures and their members.

Operation	Structure Name
Button	**WMButtonAction**
CheckBox	**WMCheckboxAction**
CustomControl	**WMCustomControlAction**
ListBox	**WMListboxAction**
ModifyFreeAxis	**WMAxisHookStruct**
PopupMenu	**WMPopupAction**
SetVariable	**WMSetVariableAction**
SetWindow	**WMWinHookStruct**
Slider	**WMSliderAction**
TabControl	**WMTabControlAction**

Applications of Structures

Structures are useful for reading and writing disk files. The **FBinRead** operation (see page V-153) and the **FBinWrite** operation (see page V-154) understand structure variables and will read or write the entire structure from or to a disk file. The individual fields of the structure will be byte-swapped if you use the /B flag.

Structures can be used in complex programming projects to reduce the dependency on global objects and to simplify passing data to and getting data from functions. For example, a base function might allocate a local structure variable and then pass that variable on to a large set of lower level routines. Because structure variables are passed by reference, data written into the structure by lower level routines is available to the higher level. Without structures, you would have to pass a large number of individual parameters or use global variables and data folders.

Using Structures with Windows and Controls

Predefined structures are available for "new-style" (new as of Igor Pro 5) action procedures for controls and hook functions for windows. See the documentation for each of the individual control listed in Chapter III-14, **Controls and Control Panels** and in Chapter V-1, **Igor Reference**, and the **SetWindow** operation (see page V-569).

Advanced programmers should also be aware of userdata that can be associated with windows using the **SetWindow** operation (see page V-569). Userdata is binary data that persists with individual windows; it is suitable for storing structures. Storing structures in a window's userdata is very handy in eliminating the need for global variables and reduces the bookkeeping needed to synchronize those globals with the window's life cycle. Userdata is also available for use with controls. See the **ControlInfo** operation (see page V-63), **GetWindow** operation (see page V-225), **GetUserData** operation (see page V-224), and **SetWindow** operation (see page V-569) operations.

Example

Here is an example illustrating Igor- and user-defined structures along with userdata in a control. Put the following in the procedure window of a new experiment and run the Panel0 macro. Then click on the buttons. Note that the buttons will remember their state even if the experiment is saved and reloaded. To fully understand this example, examine the definition of WMButtonAction in the **Button** operation (see page V-38).

```
#pragma rtGlobals=1        // Use modern global access method.

Structure mystruct
   Int32 nclicks
   double lastTime
EndStructure

Function ButtonProc(bStruct) : ButtonControl
   STRUCT WMButtonAction &bStruct

   if( bStruct.eventCode != 1 )
      return 0         // we only handle mouse down
   endif

   STRUCT mystruct s1
   if( strlen(bStruct.userdata) == 0 )
      print "first click"
   else
      StructGet/S s1,bStruct.userdata
      String ctime= Secs2Date(s1.lastTime, 1 )+" "+Secs2Time(s1.lastTime,1)
   // Warning: Next command is wrapped to fit on the page.
      printf "button %s clicked %d time(s), last click =
%s\r",bStruct.ctrlName,s1.nclicks,ctime
   endif
   s1.nclicks += 1
   s1.lastTime= datetime
   StructPut/S s1,bStruct.userdata
   return 0
End

Window Panel0() : Panel
   PauseUpdate; Silent 1         // building window…
   NewPanel /W=(150,50,493,133)
```

```
   SetDrawLayer UserBack
   Button b0,pos={12,8},size={50,20},proc=ButtonProc,title="Click"
   Button b1,pos={65,8},size={50,20},proc=ButtonProc,title="Click"
   Button b2,pos={119,8},size={50,20},proc=ButtonProc,title="Click"
   Button b3,pos={172,8},size={50,20},proc=ButtonProc,title="Click"
   Button b4,pos={226,8},size={50,20},proc=ButtonProc,title="Click"
EndMacro
```

Limitations of Structures

Although structures can reduce the need for global variables, they do not eliminate them altogether. A structure variable, like all local variables in functions, disappears when its host function returns. In order to maintain state information, you will need to store and retrieve structure information using global variables. You can do this using a global variable for each field or, with certain restrictions, you can store entire structure variables in a single global using the **StructPut** operation (see page V-679) and the **StructGet** operation (see page V-679).

As of Igor Pro 5.03, a structure can be passed to an external operation or function. See the Igor XOP Toolkit manual for details.

Static Functions

You can create functions that are local to the procedure file in which they appear by inserting the keyword *Static* in front of *Function* (see **Static** on page V-594 for usage details). The main reason for using this technique is to minimize the possibility of your function names conflicting with other names. You can use common intuitive names instead of devising a special and sometimes ugly naming strategy.

Functions normally have global scope and are available in any part of an Igor experiment, but the static keyword limits the scope of the function to its procedure file and hides it from all other procedure files. Static functions can only be used in the file in which they are defined. They can not be called from the command line and they cannot be accessed from macros.

Because static functions cannot be executed from the command line, you will have write a public test function to test them.

You can break this rule and access a static function using a module name (see **Regular Modules** on page IV-212). This technique can test the function from the command line.

Non-static functions (functions without the static keyword) are sometimes called “public” functions.

ThreadSafe Functions

ThreadSafe user functions provide support for computers with multiple processors and can be used for preemptive multitasking background tasks. A ThreadSafe function is one that can operate correctly during simultaneous execution by multiple threads.

Note: Writing a multitasking program is for expert programmers only. Intermediate programmers can write thread-safe curve-fitting functions and multithreaded assignment statements (see **Automatic Parallel Processing with MultiThread** on page IV-283). Beginning programmers should gain experience with regular programming before using multitasking.

You can create thread safe functions by inserting the keyword `ThreadSafe` in front of `Function`. For example:

```
ThreadSafe Function myadd(a,b)
   Variable a,b

   return a+b
End
```

Only a subset of functions and operations can be used in a ThreadSafe function. Generally, all numeric or utility functions can be used but those that access windows can not. To determine if a routine is ThreadSafe, use the Command Help tab of the Help Browser.

Although file operations are listed as ThreadSafe, they have certain limitations when running in a ThreadSafe function. If a file load hits a condition that normally would need user assistance, the load is aborted. No printing to history is done and there is no support for symbolic paths (use **PathInfo** and pass the path as a string input parameter).

ThreadSafe functions can call other ThreadSafe functions but may not call non-ThreadSafe functions. Non-ThreadSafe functions can call ThreadSafe functions.

When ThreadSafe functions execute in the main thread, they have normal access to data folders, waves, and variables. But when running in a preemptive thread, ThreadSafe functions use their own private data folders, waves, and variables. When a thread is started, waves can be passed to the function as input parameters. Such waves are then flagged as being in use by the thread, which prevents any changes to the size of the wave. When all threads under a given main thread are finished, the waves return to normal. You can pass data folders between the main thread and preemptive threads but such data folders are never shared.

See **ThreadSafe Functions and Multitasking** on page IV-288 for a discussion of programming with preemptive multitasking threads.

Function Overrides

In some rare cases, you may need to temporarily change an existing function. When that function is part of a package provided by someone else (or by WaveMetrics) it may be undesirable or difficult to edit the original function. By using the keyword "Override" in front of "Function" you can define a new function that will be used in place of another function of the same name that is defined in a *different and later* procedure file.

Although it is difficult to determine the order in which procedure files are compiled, the main procedure window is always first. Therefore, always define override functions in the main procedure file.

Although you can override static functions, you may run into a few difficulties. If there are multiple files with the same static function name, your override will affect all of them, and if the different functions have different parameters then you will get a link error.

Here is an example of the Override keyword. In this example, start with a new experiment and create a new procedure window. Insert the following in the new window (not the main procedure window).

```
Function foo()
   print "this is foo"
End

Function Test()
   foo()
End
```

Now, on the command line, execute Test(). You will see "this is foo" in the history.

Open the main procedure window and insert the following:

```
Override Function foo()
   print "this is an override version of foo"
End
```

Now execute Test() again. You will see the "this is an override version of foo" in the history.

Function References

Function references provide a way to pass a function to a function. This is a technique for advanced programmers. If you are just starting with Igor programming, you may want to skip this section and come back to it later.

To specify that an input parameter to a function is a function reference, use the following syntax:

```
Function Example(f)
   FUNCREF myprotofunc f
   . . .
End
```

This specifies that the input parameter f is a function reference and that a function named myprotofunc specifies the kind of function that is legal to pass. The calling function passes the name of a function as the f parameter. The called function can use f just as it would use the prototype function.

If a valid function is not passed then the prototype function will be called instead. The prototype function can either be a default function or it can contain error handling code that makes it obvious that a proper function was not passed.

A similar syntax can be used to create function reference variables in the body of a function:

```
FUNCREF protoFunc f = funcName
FUNCREF protoFunc f = $"str"
FUNCREF protoFunc f = funcRef
```

As shown, the right hand side can take either a literal function name, a $ expression that evaluates to a function name at runtime, or it can take another FUNCREF variable.

FUNCREF variables can refer to external functions as well as user functions. However, the prototype function must be a user function and it must not be static.

Although you can store a reference to a static function in a FUNCREF variable, you can not then use that variable with Igor operations that take a function as an input. FuncFit is an example of such an operation.

Following are some example functions and FUNCREFs that illustrate several concepts:

```
Function myprotofunc(a)
   Variable a

   print "in myprotofunc with a= ",a
End

Function foo1(var1)
   Variable var1

   print "in foo1 with a= ",var1
End

Function foo2(a)
   Variable a

   print "in foo2 with a= ",a
End

Function foo3(a)
   Variable a

   print "in foo3 with a= ",a
End

Function badfoo(a,b)
   Variable a,b

   print "in badfoo with a= ",a
End

Function bar(a,b,fin)
   Variable a,b
   FUNCREF myprotofunc fin
```

```
   if( a==1 )
      FUNCREF myprotofunc f= foo1
   elseif( a==2 )
      FUNCREF myprotofunc f= $"foo2"
   elseif( a==3 )
      FUNCREF myprotofunc f= fin
   endif
   f(b)
End
```

For the above functions, the following table shows the results for various invocations of the bar function executed on the command line:

Command	Result
bar(1,33,foo3)	in foo1 with a= 33
bar(2,44,foo3)	in foo2 with a= 44
bar(3,55,foo3)	in foo3 with a= 55
bar(4,55,foo3)	in myprotofunc with a= 55

Executing `bar(3,55,badfoo)` will generate a syntax error dialog that will highlight "badfoo" in the command. This error results because the format of the badfoo function does not match the format of the prototype function, myprotofunc.

Conditional Compilation

Compiler directives can be used to conditionally include or exclude blocks of code. This is especially useful when an XOP may or may not be available. It is also convenient for testing and debugging code. For example, to enable a block of procedure code depending on the presence or absence of an XOP use

```
#if Exists("nameOfAnXopRoutine")
   <proceedure code using XOP routines>
#endif
```

The conditional compiler directives are modeled after the C/C++ language. Unlike other #keyword directives, these may be indented. For defining symbols, the directives are:

```
#define symbol

#undef symbol
```

For conditional compilation, the directives are:

```
#ifdef symbol

#ifndef symbol

#if expression

#elif expression

#else

#endif
```

Expressions are ordinary Igor expressions, but cannot involve any user-defined objects. They evaluate to TRUE if the absolute value is > 0.5.

Conditionals must be either completely outside or completely inside function definitions; they cannot straddle a function definition. Conditionals cannot be used within macros but the **defined** function can.

Nesting depth is limited to 16 levels. Trailing text other than a comment is illegal.

Note that #define is used purely for defining symbols (there is nothing like C's preprocessor) and the only use of a symbol is with #if, #ifdef, #ifndef and the defined function.

The defined function was added in Igor Pro 6.20 allowing the use of:

```
#if defined(symbol)
```

Unlike C, you cannot use `#if defined(`*`symbol`*`)`.

Symbols exist only in the file where they are defined; the only exception is for symbols defined in the main procedure window, which are available to all other procedures except independent modules. In addition, you can define global symbols that are available in all procedure windows (including independent modules) using:

```
SetIgorOption poundDefine=symb
```

This adds one symbol to a global list. You can query the global list using:

```
SetIgorOption poundDefine=symb?
```

This sets V_flag to TRUE if symb exists. To remove a symbol from the global list use:

```
SetIgorOption poundUndefine=symb
```

For non-independent module procedure windows, a symbol is defined if it exists in the global list *or* in the main procedure window's list *or* in the given procedure window.

For independent module procedure windows, a symbol is defined if it exists in the global list *or* in the given procedure window; it does not use the main procedure window list.

A symbol defined in a global list is not undefined by a #undef in a procedure window.

Predefined Global Symbols

As of Igor Pro 6.20, the following global symbols are automatically predefined as appropriate, and available in all procedure windows:

Symbol	Automatically Predefined If
MACINTOSH	The Igor application is a Macintosh application.
WINDOWS	The Igor application is a Windows application.
IGOR64	The Igor application is a 64-bit application.

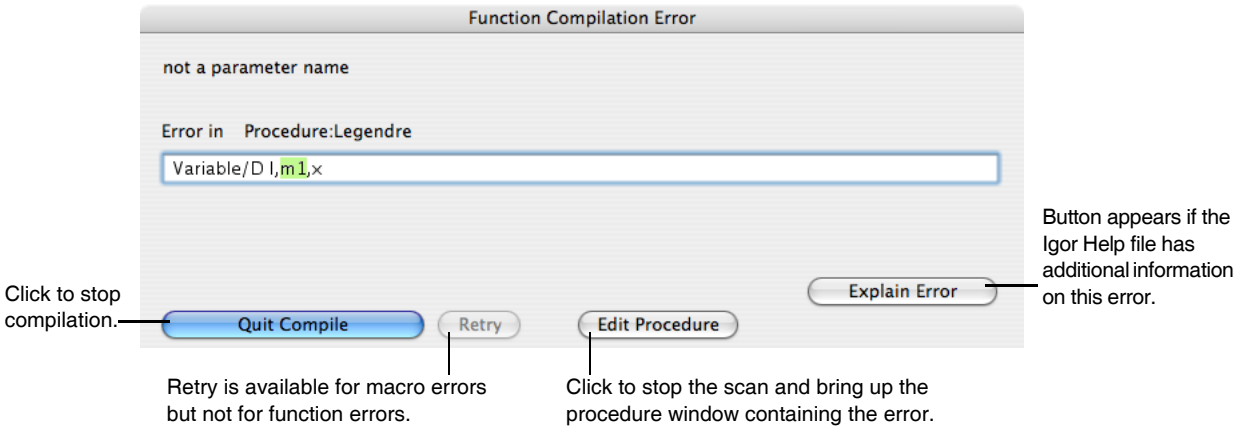
Conditional Compilation Examples

```
#define MYSYMBOL

#ifdef MYSYMBOL

Function foo()
   print "This is foo when MYSYMBOL is defined"
End

#else

Function foo()
   print "This is foo when MYSYMBOL is NOT defined"
End

#endif   // MYSYMBOL

// This works in Igor Pro 6.20 or later
#if defined(MACINTOSH)
   <conditionally compiled code here>
#endif
```

```
// This works in Igor Pro 6.00 or later
#if CmpStr("Macintosh",IgorInfo(2)) == 0
   <conditionally compiled code here>
#endif
```

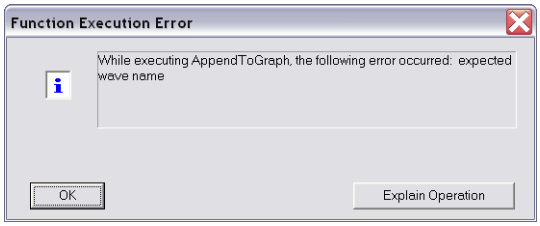
Function Errors

During function compilation, Igor checks and reports syntactic errors and errors in parameter declarations. The normal course of action is to edit the offending function and try to compile again.



Runtime errors in functions are not reported on the spot. Instead, Igor saves information about the error and function execution continues. Igor presents an error dialog only after the last function ceases execution and Igor returns to the idle state. If multiple runtime errors occur, only the first is reported.

When a runtime error occurs, after function execution ends, Igor will present a dialog:



In this example, we tried to pass to the AppendToGraph function a reference to a wave that did not exist. To find the source of the error, you should use Igor's debugger and set it to break on error (see **Debugging on Error** on page IV-185 for details).

Sophisticated programmers may want to detect and deal with runtime errors on their own. The **GetRTError** function (see page V-221) can be used to check if an error occurred, and optionally to clear the error so that Igor doesn't report it.

Coercion in Functions

The term "coercion" means the conversion of a value from one numeric precision or numeric type to another. Consider this example:

```
Function foo(awave)
   WAVE/C awave

   Variable/C var1
```

```
   var1= awave[2]*cmplx(2,3)
   return real(var1)
End
```

The parameter declaration specifies that awave is complex. You can pass any kind of wave you like but it will be coerced into complex before use. For example, if you pass a real valued integer wave the value at point index 2 will be converted to double precision and zero will be used for the imaginary part.

Operations in Functions

You can call most operations from user-defined functions. To provide this capability, WaveMetrics had to create special code for each operation. Some operations weren't worth the trouble or could cause problems. If an operation can't be invoked from a function, an error message is displayed when the function is compiled. The operations that can't be called from a function are:

AppendToLayout	Layout	Modify	OpenProc
PrintGraphs	Quit	See Also	Stack
StackWindows	Tile	TileWindows	

If you need to invoke one of these operations from a user-defined function, use the Execute operation. See **The Execute Operation** on page IV-176.

Note that the ModifyGraph, ModifyTable and ModifyLayout operations *can* be called from a function.

Also, you can use the **NewLayout** operation (see page V-437) instead of Layout, the **AppendLayoutObject** operation (see page V-23) instead of AppendToLayout, and the **RemoveLayoutObjects** operation (see page V-518) instead of RemoveFromLayout.

External operations implemented by old XOPs also can not be called directly from user-defined functions. Again, the solution is to use the Execute operation.

Updates During Function Execution

An update is an action that Igor performs which consists of:

- Reexecuting formulas for dependent objects whose antecedents have changed (see Chapter IV-9, **Dependencies**);
- Redrawing graphs and tables which display waves that have changed;
- Redrawing page layouts containing graphs, tables, or annotations that have changed;
- Redrawing windows that have been uncovered.

When no procedure is executing, Igor continually checks whether an update is needed and does an update if necessary.

When a function is executing, Igor does no automatic updates at all. However, you can force an update by calling the **DoUpdate** operation (see page V-121). Call DoUpdate if you don't want to wait for the next automatic update which will occur when function execution finishes.

Aborting Functions

There are two ways to prematurely stop procedure execution: a user abort or a programmed abort. Both stop execution of all procedures, no matter how deeply nested.

On Macintosh, you can abort procedure execution by pressing Command-period. On Windows, press Ctrl+Break or click the Abort button in the status bar. You may need to hold the keys down for a while because Igor looks at the keyboard periodically and if you don't press the keys long enough, Igor will not see them.

A user abort does not directly return. Instead it sets a flag that stops loops from looping and then returns using the normal calling chain. For this reason some code will still be executed after an abort but execution should stop quickly. This behavior releases any temporary memory allocations made during execution.

A **programmed abort** occurs during procedure execution according to conditions set by the programmer.

The simplest programmed abort occurs when the **Abort** operation (see page V-16) is executed. Here is an example:

```
if( numCells > 10 )
   Abort "Too many cells! Quitting."
endif
// code here doesn't execute if numCells > 10
```

Other programmed aborts can be triggered using the AbortOnRTE and AbortOnValue flow control keywords. The try-catch-endtry flow control construct can be used for catching and testing for aborts. See **Flow Control for Aborts** on page IV-38 for more details.

Legacy Code Issues

This section discusses changes that have occurred in Igor programming over the years. If you are writing new code, you don't need to be concerned with these issues. If you are working with existing code, you may run into some of them.

If you are just starting to learn Igor programming, you have enough to think about already, so it is a good idea to skip this section. Once you are comfortable with the modern techniques described above, come back and learn about these antiquated techniques.

Old-Style Comments and Compatibility Mode

As of Igor Pro 4.0, the old comment character, a vertical bar (|), has been replaced and should no longer be used when writing new procedures. In Igor Pro 4.0 and later, the | character is used as the bitwise OR operator (see **Operators** on page IV-5).

In order to run old procedures which use | as the comment symbol, Igor supports a compatibility mode. This mode is a property of an experiment. When the current experiment is in compatibility mode, Igor interprets | as a comment symbol. Normally, when the current experiment is not in compatibility mode, Igor interprets | as bitwise OR.

You put the current experiment in compatibility mode by executing the following on the command line:

```
Silent 100
```

You take the current experiment out of compatibility mode by executing:

```
Silent 101
```

When you execute these commands, Igor automatically recompiles all procedure files in the current experiment using the new mode.

All experiments created pre-Igor 4 are automatically in compatibility mode until you update them and their procedures.

We strongly recommend that you update old experiments and procedures so that you don't need to use compatibility mode. Until you do so, your old experiments will not work with new procedure files that use | as bitwise OR. Here are the steps to update an experiment and its procedures:

1. In each old procedure file, replace each | symbol that is used to introduce a comment with //.
   Use Edit→Find and Edit→Find Same to find each occurrence of |. Do not do a mass replace because you may inadvertently change a | symbol used in the old bitwise OR operator (%|, which is still supported) or in a string.
2. On the command line, take the experiment out of compatibility mode by executing:

```
Silent 101
```

Igor will recompile procedures.

3. If there are any remaining obsolete uses of |, Igor will display a compile error dialog. Fix the error and recompile the procedures until you get no more errors.

If you create procedure files that are used by other people (either in your group or for public use) and you want to use the new logic operations, such as | (bitwise OR) or || (logical OR), which require compatibility mode to be off (`Silent 101`), then you can specify

```
#pragma rtGlobals=2
```

in place of the normal `rtGlobals=1`.

If your procedure file is included in an experiment running in compatibility mode (`Silent 100`) then an alert dialog will be presented that will allow the user to turn compatibility mode off. However, keep in mind that when the procedures are recompiled in the new mode, the user's other procedures will generate errors if they use the obsolete comment symbol.

Text After Flow Control

Prior to Igor Pro 4, Igor ignored any extraneous text after a flow control statement. Such text was an error, but Igor did not detect it.

Igor now checks for extra text after flow control statements. When found, a dialog is presented asking the user if such text should be considered an error or not. The answer lasts for the life of the Igor session.

Because previous versions of Igor ignored this extra text, it may be common for existing procedure files to have this problem. The text may in many cases simply be a typographic error such as an extra closing parenthesis:

```
if( a==1 ))
```

In other cases, the programmer may have thought they were creating an elseif construct:

```
else if( a==2 )
```

even though the "if(a==2)" part was simply ignored. In some cases this may represent a bug in the programmer's code but most of the time it is asymptomatic.

Global Variables Used by Igor Operations

The section **Local Variables Used by Igor Operations** on page IV-47 explains that certain Igor operations create and set certain special local variables. That is true if the procedure file uses runtime lookup of globals (rtGlobals=1 is in effect). In very old procedure files that use the obsolete direct reference method of accessing variables, the operations create and set global variables.

Also explained in **Local Variables Used by Igor Operations** on page IV-47 is the fact that some operations, such as CurveFit, look for certain special local variables which modify the behavior of the operations. For historic reasons, operations that look for special variables will look for global variables in the current data folder if the local variable is not found. This is true whether rtGlobals=1 is in effect or not. This behavior is unfortunate and may be removed from Igor some day. New programming should use local variables for this purpose.

Direct Reference to Globals

The section **Accessing Global Variables and Waves** on page IV-50 explains how to access globals from a procedure file that uses runtime lookup of globals (rtGlobals=1 is in effect). This section explains accessing globals in very old procedure files that use the obsolete direct reference method of accessing variables.

Prior to Igor Pro 3, Igor required that globals referenced from functions had to exist when the function was compiled. This was called the "direct reference" method of accessing globals. In Igor Pro 3, a new method, called the "runtime lookup" method, was added.

In a particular procedure file, the presence of the statement:

```
#pragma rtGlobals = 1
```

specifies that the file uses the runtime lookup method. If this statement is absent, or if the number used is 0 instead of the usual 1, the file uses the old direct reference method.

All new programming should use the runtime lookup method and consequently all new procedure files should contain the rtGlobals pragma as shown above.

Future versions of Igor may remove the old method. Therefore Igor programmers should modify old procedure files to use it.

This section explains the old method. It is of interest only if you need to work with old files.

To compile a direct reference to a global, the global must exist at compile time. There are three ways to meet this requirement:

1. Create the global from the command line before compiling the procedures.
2. Declare the global in a function before the first use of the global.
3. Let Igor automatically create the global.

Method 1 works for waves, numeric variables and string variables. However, methods 2 and 3 work for numeric and string variables only, not for waves.

To use method 1, execute a Make, Variable, or String command from the command line before compiling procedures that refer to the global. This technique is not good for use with a separate utility procedure file because you would need to remember to create the globals each time you opened the file.

To use method 2, create a function at the top of the procedure file that declares all of the global variables used in the procedure file. For example:

```
#pragma rtGlobals=0        // Use old direct reference method

Function InitFilterProcGlobals()
   Variable/G gCutoffFrequency, gFilterType
   String/G gFilterErrorMessage

   gCutoffFrequency = 1.0
   gFilterType = 0
   gFilterErrorMessage = ""
End
```

This method works because, with rtGlobals=0, Igor creates the declared global variables and strings at compile time. When Igor compiles the Variable/G and String/G declarations, it creates the globals right then and there. Thus, the globals are guaranteed to exist when referenced later in this function or in any other function compiled after this function.

Although the globals are created at compile time, they will have default values (0 for numeric globals, "" for string globals) until the function actually runs. Compile-time declaration creates the globals but does not set their initial values.

Method number 3 occurs automatically when you compile, with rtGlobals=0, an operation (e.g., WaveStats) that returns results via global variables. Therefore the following will compile without error:

```
#pragma rtGlobals=0        // Use old direct reference method

Function WaveStdDev(w)
   WAVE w

   WaveStats/Q w
   return V_sdev
End
```

V_sdev is one of many global variable created by the WaveStats operation at compile time (if rtGlobals=0 is in effect) but set to a value at runtime. Thus, when you compile WaveStats in a function, Igor creates these global variables.

By default (if a given procedure file contains no rtGlobals statement), rtGlobals=0 is in effect. The "#pragma rtGlobals=1" statement affects only the procedure file in which it appears. If you enter it in your main procedure window, it will not affect any other included or explicitly opened procedure files. Use this to modernize one procedure file at a time.

The rtGlobals pragma can appear anywhere in the procedure file, even inside a function, and it can appear more than once. However, it should be sufficient in nearly all cases to have just one rtGlobals pragma near the beginning of the file.

Here are the steps for converting a procedure file to use the runtime lookup method for accessing globals:

1. Insert the #pragma rtGlobals=1 statement, with no indentation, at or near the top of the procedure in the file.
2. Click the Compile button to compile the procedures.
3. If you use direct reference to access a global, Igor will display an error dialog indicating the line on which the error occurred. Add an NVAR, SVAR or WAVE reference.
4. If you encountered an error in step 3, return to step 2.

Chapter
IV-4

Macros

Overview

When we first created Igor, some time in the last millennium, it supported automation through macros. The idea of the macro was to allow users to collect commands into conveniently callable routines. Igor interpreted and executed each command in a macro as if it were entered in the command line.

WaveMetrics soon realized the need for a faster, more robust technology that would support full-blown programming. This led to the addition of user-defined functions. Because functions are compiled, they execute much more quickly. Also, compilation allows Igor to catch syntactic errors sooner. Functions have a richer set of flow control capabilities and support many other programming refinements. Over time, the role of macros has diminished in favor of functions.

Macros are still supported and there are still a few uses in which they are preferred. When you close a graph, table, page layout, or control panel, Igor offers to automatically create a window recreation macro which you can later run to resurrect the window. You can also ask Igor to automatically create a window style macro using the Window Control dialog. The vast majority of programming, however, should be done using functions.

The syntax and behavior of macros are similar to the syntax and behavior of functions, but the differences can be a source of confusion for someone first learning Igor programming. If you are just starting, you can safely defer reading the rest of this chapter until you need to know more about macros, if that time ever comes.

Comparing Macros and Functions

Like functions, macros are created by entering text in procedure windows. Each macro has a name, a parameter list, parameter declarations, and a body. Unlike functions, a macro has no return value.

Macros and functions use similar syntax. Here is an example of each. To follow along, open the Procedure window (Windows menu) and type in the macro and function definitions.

```
Macro MacSayHello(name)
   String name

   Print "Hello "+name
End

Function FuncSayHello(name)
   String name

   Print "Hello "+name
End
```

Now click in the Command window to bring it to the front. Use this to type commands into the command line and to compile the Procedure window.

If you execute the following on the command line

```
MacSayHello("John"); FuncSayHello("Sue")
```

you will see the following output printed in the history area:

```
Hello John
Hello Sue
```

This example may lead you to believe that macros and functions are nearly identical. In fact, there are a lot of differences. The most important differences are:

- Macros automatically appear in the Macros menu. Functions must be explicitly added to a menu, if desired, using a menu definition.
- Most errors in functions are detected when procedures are compiled. Most errors in macros are detected when the macro is executed.
- Functions run a lot faster than macros.
- Functions support wave parameters, for loops and switches while macros do not.

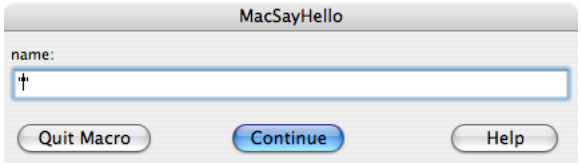
- Functions have a richer syntax.

If you look in the Macros menu, you will see MacSayHello but not FuncSayHello.

If you execute "`FuncSayHello()`" on the command line you will see an error dialog. This is because you must supply a parameter. You can execute, for example:

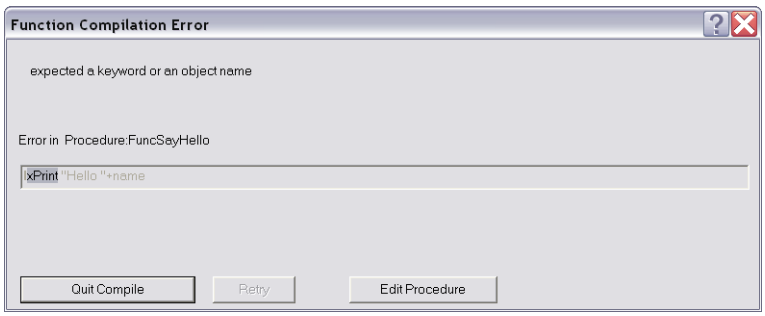
```
FuncSayHello("Sam")
```

On the other hand, if you run MacSayHello from the Macros menu or if you execute "`MacSayHello()`" on the command line, you will see a dialog that you use to enter the name before continuing:



This is called the "missing parameter dialog". It is described under **The Missing Parameter Dialog** on page IV-101. Functions can display a similar dialog, called a simple input dialog, with a bit of additional programming.

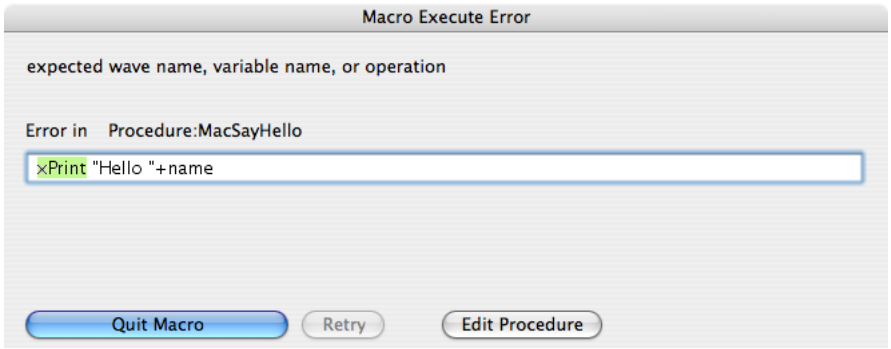
Now try this: In both procedures, change "Print" to "xPrint". Then click in the command window. You will see a dialog like this:



Click the Edit Procedure button and change "xPrint" back to "Print" in the function but not in the macro. Then click in the command window.

Notice that no error was reported once you fixed the error in the function. This is because only functions are compiled and thus only functions have their syntax completely checked at compile time. Macros are interpreted and most errors are found only when the line in the procedure window in which they occur is executed. To see this, run the macro by executing "`MacSayHello("Sam")`" on the command line.

You will then see this dialog:



Notice the box around the line containing the error. This box means you can edit its contents. If you change "xPrint" to "Print" in this dialog you will see that the Retry button becomes enabled. If you click this button,

you can continue execution of the macro. When the macro finishes, take a look at the Procedure window. You will notice that the correction you made in the dialog was put in the Procedure window and your "broken" macro is now fixed.

Macro Syntax

Here is the basic syntax for macros.

```
<Defining keyword> <Name> ( <Input parameter list> ) [:<Subtype>]
   <Input parameter declarations>

   <Local variable declarations>

   <Body code>
End
```

The Defining Keyword

<Defining keyword> is one of the following:

Defining Keyword	Creates Macro In
Window	Windows menu.
Macro	Macros menu.
Proc	—

The Window keyword is used by Igor when it automatically creates a window recreation macro. Except in rare cases, you will not write window recreation macros but instead will let Igor create them automatically.

The Procedure Name

The names of macros must follow the standard Igor naming conventions. Names can consist of up to 31 characters. The first character must be alphabetic while the remaining characters can include alphabetic and numeric characters and the underscore character. Names must not conflict with the names of other Igor objects, functions or operations. Names in Igor are case insensitive.

The Procedure Subtype

You can identify procedures designed for specific purposes by using a subtype. Here is an example:

```
Proc ButtonProc(ctrlName) : ButtonControl
   String ctrlName

   Beep
End
```

Here, "`: ButtonControl`" identifies a macro intended to be called when a user-defined button control is clicked. Because of the subtype, this macro is added to the menu of procedures that appears in the Button Control dialog. When Igor automatically generates a procedure it generates the appropriate subtype. See **Procedure Subtypes** on page IV-179 for details.

The Parameter List and Parameter Declarations

The parameter list specifies the name for each input parameter. Macros have a limit of 10 parameters.

The parameter declaration must declare the type of each parameter using the keywords `Variable` or `String`. If a parameter is a complex number, it must be declared `Variable/C`.

Note: There should be no blank lines or other commands until after all the input parameters are defined. There should be one blank line after the parameter declarations, before the rest of the procedure. Igor will report errors if these conditions are not met.

Variable and string parameters in macros are always passed to a subroutine by value.

When macros are invoked with some or all of their input parameters missing, Igor displays a missing parameter dialog to allow the user to enter those parameters. In the past this has been a reason to use macros. However, as of Igor Pro 4, functions can present a similar dialog to fetch input from the user, as explained under **The Simple Input Dialog** on page IV-122.

Local Variable Declarations

The input parameter declarations are followed by the local variable declarations if the macro uses local variables. Local variables exist only during the execution of the macro. They can be numeric or string and are declared using the `Variable` or `String` keywords. They can optionally be initialized. Here is an example:

```
Macro Example(p1)
   Variable p1

   // Here are the local variables
   Variable v1, v2
   Variable v3=0
   Variable/C cv1=cmplx(0,0)
   String s1="test", s2="test2"

   <Body code>
End
```

If you do not supply explicit initialization, Igor automatically initializes local numeric variables with the value zero and local string variables with the value "".

The name of a local variable is allowed to conflict with other names in Igor although they must be unique within the macro. Clearly if you create a local variable named “sin” then you will be unable to use Igor’s built-in sin function within the macro.

You can declare a local variable in any part of a macro with one exception. If you place a variable declaration inside a loop in a macro then the declaration will be executed multiple times and Igor will generate an error since local variable names must be unique.

Body Code

The local variable declarations are followed by the body code. This table shows what can appear in body code of a macro.

What	Allowed in Macros?	Comments
Assignment statements	Yes	Includes wave, variable and string assignments.
Built-in operations	Yes	
External operations	Yes	
External functions	Yes	
Calls to user functions	Yes	
Calls to macros	Yes	
if-else-endif	Yes	
if-elseif-endif	No	
switch-case-endswitch	No	
strswitch-case-endswitch	No	
try-catch-endtry	No	
structures	No	
do-while	Yes	
for-endfor	No	

What	Allowed in Macros?	Comments
Comments	Yes	Comments start with //.
break	Yes	Used in loop statements.
continue	No	
default	No	
return	Yes, but with no return value.	

Conditional Statements in Macros

The conditional if-else-endif statement is allowed in macros. It works the same as in functions. See **If-Else-Endif** on page IV-31.

Loops in Macros

The do-while loop is supported in macros. It works the same as in functions. See **Do-While Loop** on page IV-36.

Return Statement in Macros

The return keyword immediately stops executing the current macro. If it was called by another macro, control returns to the calling macro.

A macro has no return value so return is used just to prematurely quit the macro. Most macros will have no return statement.

Invoking Macros

There are several ways to invoke a macro:

- From the command line
- From the Macros, Windows or user-defined menus
- From another macro
- From a button or other user control

The menu in which a macro appears, if any, is determined by the macro's type and subtype.

This table shows how a macro's type determines the menu that Igor puts it in.

Macro Type	Defining Keyword	Menu
Macro	`Macro`	Macros menu
Window Macro	`Window`	Windows menu
Proc	`Proc`	—

If a macro has a subtype, it may appear in a different menu. This is described under **Procedure Subtypes** on page IV-179. You can put macros in other menus as described in Chapter IV-5, **User-Defined Menus**.

You can not directly invoke a macro from a user function. You can invoke it indirectly, using the **Execute** operation (see page V-145).

Using $ in Macros

As shown in the following example, the $ operator can create references to global numeric and string variables as well as to waves.

```
Macro MacroTest(vStr, sStr, wStr)
   String vStr, sStr, wStr

   $vStr += 1
   $sStr += "Hello"
   $wStr += 1
End

Variable/G gVar = 0; String/G gStr = ""; Make/O/N=5 gWave = p
MacroTest("gVar", "gStr", "gWave")
```

See **String Substitution Using $** on page IV-15 for additional examples using $.

Waves as Parameters in Macros

The only way to pass a wave to a macro is to pass the name of the wave in a string parameter. You then use the $ operator to convert the string into a wave reference. For example:

```
Macro PrintWaveStdDev(w)
   String w

   WaveStats/Q $w
   Print V_sdev
End

Make/O/N=100 test=gnoise(1)
Print NamedWaveStdDev("test")
```

The Missing Parameter Dialog

When a macro that is declared to take a set of input parameters is executed with some or all of the parameters missing, it displays a dialog in which the user can enter the missing values. For example:

```
Macro MacCalcDiag(x,y)
   Variable x=10
   Prompt x, "Enter X component: "     // Set prompt for y param
   Variable y=20
   Prompt y, "Enter Y component: "     // Set prompt for x param

   Print "diagonal=",sqrt(x^2+y^2)
End
```

If invoked from the command line or from another macro with parameters missing, like this:

```
   MacCalcDiag()
```

Igor displays the Missing Parameter dialog in which the parameter values can be specified.

The Prompt statements are optional. If they are omitted, the variable name is used as the prompt text.

There must be a blank line after the set of input parameter and prompt declarations and there must not be any blank lines within the set.

The missing parameter dialog supports the creation of pop-up menus as described under **Pop-Up Menus in Simple Dialogs** on page IV-123. One difference is that in a missing parameter dialog, the menu item list can be continued using as many lines as you need. For example:

```
Prompt color, "Select Color", popup "red;green;blue;"
"yellow;purple"
```

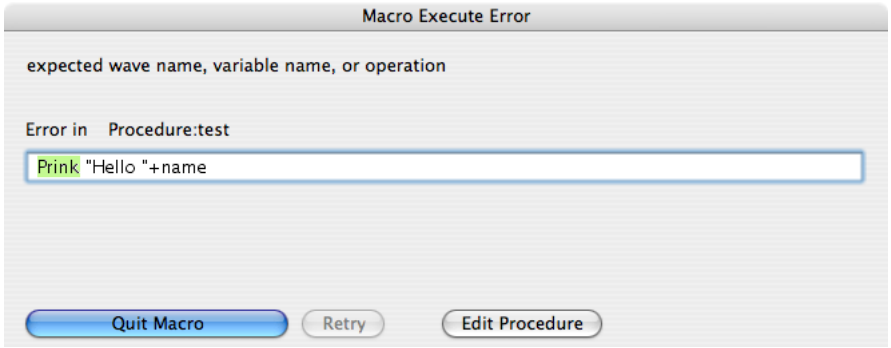
Macro Errors

Igor can find errors in macros at two times:

- When it scans the macros
- When it executes the macros

After you modify procedure text, scanning occurs when you activate a nonprocedure window, click the Compile button or choose Compile from the Macros menu. At this point, Igor is just looking for the names of procedures. The only errors that it detects are name conflicts and ill-formed names. If it finds such an error, it displays a dialog that you use to fix it.

Igor detects other errors when the macro executes. Execution errors may be recoverable or nonrecoverable. If a recoverable error occurs, Igor puts up a dialog in which you can edit the erroneous line.



You can fix the error and retry or quit macro execution.

If the error is nonrecoverable, you get a similar dialog except that you can't fix the error and retry. This happens with errors in the parameter declarations and errors related to if-else-endif and do-while structures.

The Silent Option

Normally Igor displays each line of a macro in the command line as it executes the line. This gives you some idea of what is going on. However it also slows macro execution down considerably. You can prevent Igor from showing macro lines as they are executed by using the `Silent 1` command.

You can use `Silent 1` from the command line. It is more common to use it from within a macro. The effect of the `Silent` command ends at the end of the macro in which it occurs. Many macros contain the following line:

```
Silent 1; PauseUpdate
```

The Slow Option

You can observe the lines in a macro as they execute in the command line. However, for debugging purposes, they often whiz by too quickly. The `Slow` operation slows the lines down. It takes a parameter which controls how much the lines are slowed down. Typically, you would execute something like "`Slow 10`" from the command line and then "`Slow 0`" when you are finished debugging.

You can also use the `Slow` operation from within a macro. You must explicitly invoke "`Slow 0`" to revert to normal behavior. It does not automatically revert at the end of the macro from which it was invoked.

We never use this feature. Instead, we generally use print statements for debugging or we use the Igor symbolic debugger, described in Chapter IV-8, **Debugging**.

Accessing Variables Used by Igor Operations

A number of Igor's operations return results via variables. For example, the WaveStats operation creates a number of variables with names such as V_avg, V_sigma, etc.

When you invoke these operations from the command line, they create global variables in the current data folder.

When you invoke them from a user-defined function, they create local variables.

When you invoke them from a macro, they create local variables unless a global variable already exists. If both a global variable and a local variable exist, Igor uses the local variable.

In addition to creating variables, a few operations, such as CurveFit and FuncFit, check for the existence of specific variables to provide optional behavior. The operations look first for a local variable with a specific name. If the local variable is not found, they then look for a global variable.

Updates During Macro Execution

An update is an action that Igor performs. It consists of:

- Reexecuting assignments for dependent objects whose antecedents have changed (see Chapter IV-9, **Dependencies**);
- Redrawing graphs and tables which display waves that have changed;
- Redrawing page layouts containing graphs, tables, or annotations that have changed;
- Redrawing windows that have been uncovered.

When no procedure is executing, Igor continually checks whether an update is needed and does an update if necessary.

During macro execution, Igor checks if an update is needed after executing each line. You can suspend checking using the PauseUpdate operation. This is useful when you want an update to occur when a macro finishes but not during the course of the macro's execution.

PauseUpdate has effect only inside a macro. Here is how it is used.

```
Window Graph0() : Graph
   PauseUpdate; Silent 1
   Display /W=(5,42,400,250) w0,w1,w2
   ModifyGraph gFont="Helvetica"
   ModifyGraph rgb(w0)=(0,0,0),rgb(w1)=(0,65535,0),rgb(w2)=(0,0,0)
   <more modifies here...>
End
```

Without the PauseUpdate, Igor would do an update after each modify operation. This would take a long time.

At the end of the macro, Igor automatically reverts the state of update-checking to what it was when this macro was invoked. You can use the ResumeUpdate operation if you want to resume updates before the macro ends or you can call DoUpdate to force an update to occur at a particular point in the program flow. Such explicit updating is rarely needed.

Aborting Macros

There are two ways to prematurely stop macro execution: a user abort or a programmed abort. Both stop execution of all macros, no matter how deeply nested.

On Macintosh, you can abort macro execution by pressing Command-period. On Windows, press Ctrl+Break or click the Abort button in the status bar. You may need to hold the keys down for a while because Igor looks at the keyboard periodically and if you don't press the keys long enough, Igor will not see them.

On either platform, you can abort macro execution by choosing Abort Procedure Execution from the Macros menu.

A user abort does not directly return. What it does is set a flag that stops loops from looping and then returns using the normal calling chain. For this reason some code will still be executed after an abort but execution should stop quickly. This behavior releases any temporary memory allocations made during execution.

A **programmed abort** occurs when the Abort operation is executed. Here is an example:

```
if( numCells > 10 )
   Abort "Too many cells! Quitting."
endif
// code here doesn't execute if numCells > 10
```

Converting Macros to Functions

If you have old Igor procedures written as macros, as you have occasion to revisit them, you can consider converting them to functions. In most cases, this is a good idea. An exception is if the macros are so complex that there would be a substantial risk of introducing bugs. In this case, it is better to leave things as they are.

If you decide to do the conversion, here is a checklist that you can use in the process.

1. Back up the old version of the procedures.
2. Change the defining keyword from Macro or Proc to Function.
3. If the macro contained Prompt statements, then it was used to generate a missing parameter dialog. Change it to generate a simple input dialog as follows:
   a. Remove the parameters from the parameter list. The old parameter declarations now become local variable declarations.
   b. Make sure that the local variable for each prompt statement is initialized to some value.
   c. Add a DoPrompt statement after all of the Prompt statements.
   d. Add a test on V_Flag after the DoPrompt statement to see if the user canceled.
4. Look for statements that access global variables or strings and create NVAR and SVAR references for them.
5. Look for any waves used in assignment statements and create WAVE references for them.
6. Compile procedures. If you get an error, fix it and repeat step 6.

Chapter
IV-5

User-Defined Menus

Overview

You can add your own menu items to many Igor menus by writing a menu definition in a procedure window. A simple menu definition looks like this:

```
Menu "Macros"
   "Load Data File/1"
   "Do Analysis/2"
   "Print Report"
End
```

This adds three items to the Macros menu. If you choose Load Data File or press Command-1 (*Macintosh*) or Ctrl+1 (*Windows*), Igor will execute the procedure LoadDataFile which, presumably, you have written in a procedure window. The command executed when you select a particular item is derived from the text of the item. This is an *implicit* specification of the item's execution text.

You can also *explicitly* specify the execution text:

```
Menu "Macros"
   "Load Data File/1", Beep; LoadWave/G
   "Do Analysis/2"
   "Print Report"
End
```

Now if you choose Load Data File, Igor will execute "`Beep; LoadWave/G`".

When you choose a user menu item, Igor checks to see if there is execution text for that item. If there is, Igor executes it. If not, Igor makes a procedure name from the menu item string. It does this by removing any characters that are not legal characters in a procedure name. Then it executes the procedure. For example, choosing an item that says

```
"Set Sampling Rate..."
```

executes the SetSamplingRate procedure.

If a procedure window is the top window and if Option (*Macintosh*) or Alt (*Windows*) is pressed when you choose a menu item, Igor tries to find the procedure in the window, rather than executing it.

A menu definition can add submenus as well as regular menu items.

```
Menu "Macros"
   Submenu "Load Data File"
      "Text File"
      "Binary File"
   End

   Submenu "Do Analysis"
      "Method A"
      "Method B"
   End

   "Print Report"
End
```

This adds three items to the Macros menu, two submenus and one regular item. You can nest submenus to any depth.

There are some limits to the number of menu items that you can add. Igor can handle no more than 100 user-defined main menus and not more than 300 user-defined submenus.

The user-defined menus added by procedure files opened in the current experiment (such as the Example Experiments menu item in the File menu added by the WMMenus.ipf procedure file in "Igor Pro Folder/Igor Procedures") count against those limits.

In the Windows version of Igor, there is also a limit of 31 items in a menu, and each special user-defined menu counts as several user-defined menus or submenus. For example, a "*COLORPOP*" submenu counts as 6 submenus against the allowed 200. See **Specialized Menu Item Definitions** on page IV-112.

Menu Definition Syntax

The syntax for a menu definition is:

```
Menu <Menu title string> [,<menu options>]
   [<Menu help strings>]
   <Menu item string> [,<menu item flags>] [,<execution text>]
   [<Item help strings>]
   …
   Submenu <Submenu title string>
      [<Submenu help strings>]
      <Submenu item string> [,<execution text>]
      [<Item help strings>]
      …
   End
End
```

<Menu title string> is the title of the menu to which you want to add items. Often this will be Macros but you can also add items to Analysis, Misc and many other built-in Igor menus, including some submenus and the graph marquee and layout marquee menus. If <Menu title string> is not the title of a built-in menu then Igor creates a new main menu on the menu bar.

<Menu options> are optional comma-separated keywords that change the behavior of the menu. The allowed keywords are dynamic, hideable, and contextualmenu. For usage, see **Dynamic Menu Items** (see page IV-109), **HideIgorMenus** (see page V-242), and **PopupContextualMenu** (see page V-487) respectively.

<Menu help strings> specifies the help for the menu title. This is optional. See **Help for User Menus** on page IV-108 for details.

<Menu item string> is the text to appear for a single menu item, a semicolon-separated string list to define **Multiple Menu Items** (see page IV-111), or **Specialized Menu Item Definitions** (see page IV-112) such as a color, line style, or font menu.

<Menu item flags> are optional flags that modify the behavior of the menu item. The only flag currently supported is /Q, which prevents Igor from storing the executed command in the history area. This is useful for menu commands that are executed over and over through a keyboard shortcut. This feature was introduced in Igor Pro 5. Using it will cause errors in earlier versions of Igor. Menus defined with the contextualmenu keyword implicitly set all the menu item flags in the entire menu to /Q; it doesn't matter whether /Q is explicitly set or not, the executed command is not stored in the history area.

<Execution text> is an Igor command to execute for the menu item. If omitted, Igor makes a procedure name from the menu item string and executes that procedure. Use "" to prevent command execution (useful only with PopupContextualMenu/N).

<Item help strings> specifies the help for the menu item. This is optional. It can appear after a main menu item or after a submenu item. See **Help for User Menus** on page IV-108 for details.

The Submenu keyword introduces a submenu with <Submenu title string> as its title. The submenu continues until the next End keyword.

<Submenu item string> acts just like <Menu item string>.

Built-in Menus That Can Be Extended

Here are the titles of built-in Igor menus to which you can add items.

Add Controls	AllTracesPopup	Analysis	Append to Graph	Control
Data	Edit	File	Graph	GraphMarquee
Help	Layout	LayoutMarquee	Load Waves	Macros
Misc	Statistics	New	Notebook	Open File
Panel	Procedure	Save Waves	Table	TracePopup

These menu titles must appear in double quotes when used in a menu definition.

Use these menu titles to identify the menu to which you want to append items even if you are working with a version of Igor translated into a language other than English.

The GraphMarquee, LayoutMarquee, TracePopup, and AllTracesPopup menus are contextual menus. See **Marquee Menus** on page IV-119 and **Trace Menus** on page IV-119.

All other Igor menus, including menus added by XOPs, can not accept user-defined items.

The **HideIgorMenus** operation (see page V-242) and the **ShowIgorMenus** operation (see page V-571) hide or show most of the built-in main menus (not the Marquee and Popup menus). User-defined menus that add items to built-in menus are normally not hidden or shown by these operations. When a built-in menu is hidden, the user-defined menu items create a user-defined menu with only user-defined items. For example, this user-defined menu:

```
Menu "Table"
   "Append Columns to Table...", DoIgorMenu "Table", "Append Columns to Table"
End
```

will create a Table menu with only one item in it after the HideIgorMenus "Table" command is executed.

To have your user-defined menu items hidden along with the built-in menu items, add the hideable keyword after the Menu definition:

```
Menu "Table", hideable
   "Append Columns to Table...", DoIgorMenu "Table", "Append Columns to Table"
End
```

Adding a New Main Menu

You can add an entirely new menu to the main menu bar by using a menu title that is not used by Igor. For example:

```
Menu "Test"
   "Load Data File"
   "Do Analysis"
   "Print Report"
End
```

Help for User Menus

You can specify help for the menus and menu items that you define. On Mac OS X, the help text appears if you enable Igor Tips through the Help menu. On Windows, Igor uses the help text to provide status line help.

```
Menu "Test"
   help = {"This is the help for the Test menu."}

   "Load Data File"
   help = {"This is the help for the Load Data File item."}

   "Do Analysis"
   help = {"This is the help for the Do Analysis item."}
```

```
   "Print Report"
   help = {"This is the help for the Print Report item."}
End
```

The first help line specifies the help text for the menu title. The subsequent lines specify the help text to appear for the respective menu items.

On Macintosh, help does not appear for menu and submenu titles. It does appear over menu and submenu items.

Because of technical difficulties, help for user-defined items in the Edit menu will not appear. Also, on Macintosh, help for user-defined items in the Help menu will not appear.

The full form of the menu help specification is:

```
help = {"String 1", "String 2", "String 3", "String 4"}
```

String 1 supplies the text to appear when the menu or menu item is enabled. String 2 supplies the text to appear when the menu or menu item is disabled. String 3 supplies the text to appear when the menu or menu item is checked. String 4 supplies the text to appear when the menu or menu item is marked with a marker other than check.

If you omit one or more of these strings, Igor uses the text for string 1 for the corresponding state. Thus, the following are equivalent:

```
help = {"Test 1", "Test 2", "Test 1", "Test 1"}
help = {"Test 1", "Test 2"}
```

In most cases, it will be sufficient to provide just the text for the enabled state since you will most likely not go to the trouble of enabling, disabling or checking your menu items.

Dynamic Menu Items

In the examples shown so far all of the user-defined menu items are static. Once defined, they never change. This is sufficient for the vast majority of cases and is by far the easiest way to define menu items.

Igor also provides support for dynamic user-defined menu items. A dynamic menu item changes depending on circumstances. The item might be enabled under some circumstances and disabled under others. It might be checked or deselected. Its text may toggle between two states (e.g. “Show Tools” and “Hide Tools”).

Because dynamic menus are much more difficult to program than static menus and also slow down Igor’s response to a menu-click, we recommend that you keep your use of dynamic menus to a minimum. The effort you expend to make your menu items dynamic may not be worth the time you spend to do it.

For a menu item to be dynamic, you must define it using a string expression instead of the literal strings used so far. Here is an example.

```
Function DoAnalysis()
   Print "Analysis Done"
End

Function ToggleTurboMode()
   Variable prevMode = NumVarOrDefault("root:gTurboMode", 0)
   Variable/G root:gTurboMode = !prevMode
End

Function/S MacrosMenuItem(itemNumber)
   Variable itemNumber

   Variable turbo = NumVarOrDefault("root:gTurboMode", 0)

   if (itemNumber == 1)
      if (strlen(WaveList("*", ";", ""))==0) // any waves exist?
```

```
         return "(Do Analysis"   // disabled state
      else
         return "Do Analysis"    // enabled state
      endif
   endif

   if (itemNumber == 2)
      if (turbo)
         return "!"+num2char(18)+"Turbo"  // Turbo with a check
      else
         return "Turbo"
      endif
   endif
End

Menu "Macros", dynamic
   MacrosMenuItem(1)
   help= {"Do analysis", "Not available because there are no waves."}

   MacrosMenuItem(2), /Q, ToggleTurboMode()
   help= {"When checked, turbo mode is on."}
End
```

In this example, the text for the menu item is computed by the MacrosMenuItem function. It computes text for item 1 and for item 2 of the menu. Item 1 can be enabled or disabled. Item 2 can be checked or unchecked.

The dynamic keyword specifies that the menu definition contains a string expression that needs to be reevaluated each time the menu item is drawn. This rebuilds the user-defined menu each time the user clicks in the menu bar. Under the current implementation, *all* user menus are rebuilt each time the user clicks in the menu bar if *any* user-defined menu is declared dynamic. If you use a large number of user-defined items, the time to rebuild the menu items may be noticeable.

There is another technique for making menu items change. You define a menu item using a string expression rather than a literal string but you do not declare the menu dynamic. Instead, you call the BuildMenu operation whenever you need the menu item to be rebuilt. Here is an example:

```
Function ToggleItem1()
   String item1Str = StrVarOrDefault("root:MacrosItem1Str","On")
   if (CmpStr(item1Str,"On") == 0)     // Item is now "On"?
      String/G root:MacrosItem1Str = "Off"
   else
      String/G root:MacrosItem1Str = "On"
   endif
   BuildMenu "Macros"
End

Menu "Macros"
   StrVarOrDefault("root:MacrosItem1Str","On"), /Q, ToggleItem1()
End
```

Here, the menu item is controlled by the global string variable MacrosItem1Str. When the user chooses the menu item, the ToggleItem1 function runs. This function changes the MacrosItem1Str string and then calls BuildMenu, which rebuilds the user-defined menu the next time the user clicks in the menu bar. Under the current implementation, Igor will rebuild *all* user-defined menus if BuildMenu is called for *any* user-defined menu.

Optional Menu Items

A dynamic user-defined menu item *disappears* from the menu if the menu item string expression evaluates to ""; the remainder of the menu definition line is then ignored. This makes possible a variable number of items in a user-defined menu list. This example adds a menu listing the names of up to 8 waves in the current data folder. If the current data folder contains less than 8 waves, then only those that exist are shown in the menu:

```
Menu "Waves", dynamic
   WaveName("",0,4), DoSomething($WaveName("",0,4))
   WaveName("",1,4), DoSomething($WaveName("",1,4))
   WaveName("",2,4), DoSomething($WaveName("",2,4))
   WaveName("",3,4), DoSomething($WaveName("",3,4))
   WaveName("",4,4), DoSomething($WaveName("",4,4))
   WaveName("",5,4), DoSomething($WaveName("",5,4))
   WaveName("",6,4), DoSomething($WaveName("",6,4))
   WaveName("",7,4), DoSomething($WaveName("",7,4))
End

Function DoSomething(w)
   Wave/Z w

   if( WaveExists(w) )
      Print "DoSomething: wave's name is "+NameOfWave(w)
   endif
End
```

This works because WaveName returns "" if the indexed wave doesn't exist.

Note that each potential item must have a menu definition line that either appears or disappears.

Multiple Menu Items

A menu item string that contains a semicolon-separated "string list" (see **StringFromList** on page V-675) generates a menu item for each item in the list. For example:

```
Menu "Multi-Menu"
   "first item;second item;", DoItem()
End
```

Multi-Menu will be a two-item menu. When either item is selected the DoItem procedure is called.

This begs the question: How does the DoItem procedure know which item was selected? The answer is that DoItem must call the **GetLastUserMenuInfo** operation (see page V-216) and examine the appropriate returned variables, usually V_value (the selected item's number, 1 for the first item, 2 for the second, etc.) or S_Value (the selected item's text).

The string list can be dynamic, too. The above "Waves" example can be rewritten to handle an arbitrary number of waves (maximum 31 on Windows) using this definition:

```
Menu "Waves", dynamic
   WaveList("*",";",""), DoItem()
End

Function DoItem()
   GetLastUserMenuInfo           // sets S_value, V_value, etc.
   WAVE/Z w= $S_value
   if( WaveExists(w) )
      Print "The wave's name is "+NameOfWave(w)
   endif
End
```

Consolidating Menu Items Into a Submenu

It is common to have many utility procedure files open at the same time. Each procedure file could add menu items which would clutter Igor's menus. When you create a utility procedure file that adds multiple menu items, it is usually a good idea to consolidate all of the menu items into one submenu. Here is an example.

Let's say we have a procedure file full of utilities for doing frequency-domain data analysis and that it contains the following:

```
Function ParzenDialog()
   …
End

Function WelchDialog()
   …
End

Function KaiserDialog()
   …
End
```

We can consolidate all of the menu items into a single submenu in the Analysis menu:

```
Menu "Analysis"
   Submenu "Windows"
      "Parzen…", ParzenDialog()
      "Welch…", WelchDialog()
      "Kaiser…", KaiserDialog()
   End
End
```

Specialized Menu Item Definitions

A menu item string that contains certain special values adds a specialized menu such as a color menu.

Only one specialized menu item string is allowed in each menu or submenu, it must be the first item, and it must be the only item.

Menu Item String	Result
"*CHARACTER*	"Character menu, no character is initially selected, font is Geneva, font size is 12. See **Menu Limits** on page IV-113 for *CHARACTER* menu limitations.
"*CHARACTER*(Symbol)	"Character menu of Symbol font.
"*CHARACTER*(Symbol,36)	"Character menu of Symbol font in 36 point size.
"*CHARACTER*(,36)	"Character menu of Geneva font in 36 point size.
"*CHARACTER*(Symbol,36,p)	"Character menu of Symbol font in 36 point size, initial character is p (π).
"*COLORTABLEPOP*	"Color table menu, initial table is Grays.
"*COLORTABLEPOP*(YellowHot)	"Color table menu, initial table is YellowHot. See **CTabList** on page V-81 for a list of color tables.
"*COLORTABLEPOP*(YellowHot,1)	"Color table menu with the colors drawn reversed.
"*COLORPOP*	"Color menu, initial color is black.
"*COLORPOP*(0,65535,0)	"Color menu, initial color is green.
"*FONT*	"Font menu, no font is initially selected, does not include "default" as a font choice. See **Menu Limits** on page IV-113 for *FONT* menu limitations.
"*FONT*(Arial)	"Font menu, Arial is initially selected.
"*FONT*(Arial,default)	"Font menu with Arial initially selected and including "default" as a font choice.
"*LINESTYLEPOP*	"Line style menu, no line style is initially selected.

Menu Item String	Result
"*LINESTYLEPOP*(3)	"Line style menu, initial line style is style=3 (coarse dashed line).
"*MARKERPOP*	"Marker menu, no marker is initially selected.
"*MARKERPOP*(8)	"Marker menu, initial marker is 8 (empty circle).
"*PATTERNPOP*	"Pattern menu, no pattern is initially selected.
"*PATTERNPOP*(1)	"Pattern menu, initial pattern is 1 (SW-NE light diagonal).

To retrieve the selected color, line style, etc., the execution text must be a procedure that calls the **GetLastUserMenuInfo** operation (see page V-216). Here's an example of a color submenu implementation:

```
Menu "Main", dynamic
   "First Item", /Q, HandleFirstItem()
   Submenu "Color"
      CurrentColor(), /Q, SetSelectedColor() // must be first item
                                             // can't add items here
   End
   "Third Item", /Q, HandleThirdItem()
End

Function/S CurrentColor()
   Variable/G root:red, root:green, root:blue
   NVAR red= root:red
   NVAR green= root:green
   NVAR blue= root:blue
   String menuText
   sprintf menuText, "*COLORPOP*(%d,%d,%d)", red, green, blue
   return menuText
End

Function SetSelectedColor()
   GetLastUserMenuInfo  // sets V_red, V_green, V_blue, S_value, V_value
   Variable/G root:red= V_red
   Variable/G root:green= V_green
   Variable/G root:blue= V_blue

   Make/O/N=(2,2,3) root:colorSpot
   Wave colorSpot= root:colorSpot
   colorSpot[][][0]= V_red
   colorSpot[][][1]= V_green
   colorSpot[][][2]= V_blue

   CheckDisplayed/A colorSpot
   if( V_Flag == 0 )
      NewImage colorSpot
   endif
End
```

Menu Limits

Although the maximum font size allowed is 99, the *CHARACTER* menus are limited to about 800 pixels wide by 600 pixels high and most font sizes above 48 or so don't increase the font size of the displayed characters.

The *FONT* menus are quite different on Macintosh and Windows.

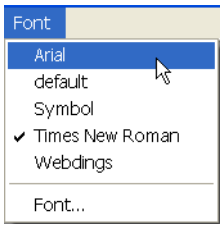
The Macintosh menu is just a long menu of all the fonts, optionally with "default" at the bottom.

On Windows, menus are limited to 31 items. If you specify just "*FONT*", the resulting menu item has just a "Font…" item. You can list up to 28 font names including the initially-selected font (the first one listed) and optionally a divider represented by "-". "default" can be one of the listed font names. A "Font…" item

always appears in the menu to select any installed font. Fonts that you list but which are not installed are ignored and won't appear in the menu. Those that do appear are sorted alphabetically. For example:

```
Menu " Font"        // (The "Font" menu is restricted, but " Font" isn't)
    "*FONT*(Times New Roman,default,Arial,Webdings,Symbol)", DoFont()
End

Function DoFont()
    GetLastUserMenuInfo
    Print S_value
End
```

produces a Font menu with "Orange LET" initially selected and all the listed fonts as (sorted) font choices:



On Windows, most specialized menu items count as more than one menu against the limits of 100 user-defined main menus or 200 user-defined submenus:

Menu Kind	Effective Number of Menus (Windows only)
"*FONT*"	1
"*LINESTYLEPOP"*	1
"*PATTERNPOP*"	3
"*MARKERPOP*"	2
"*CHARACTER*"	8
"*COLORPOP*"	6
"*COLORTABLEPOP*"	2

Special Characters in Menu Item Strings

You can control some aspects of a menu item by using special characters. These special characters are based on the behavior of the Macintosh menu manager and are only partially supported on Windows (see **Special Menu Characters on Windows** on page IV-116). They affect user-defined menus in the main menu bar. On Macintosh, but not on Windows, they also affect user-defined pop-up menus in control panels, graphs and simple input dialogs.

By default, special character interpretation is enabled in user-defined menu bar menus and is disabled in user-defined control panel, graph and simple input dialog pop-up menus. This is almost always what you would want. In some cases, you might want to override the default behavior. This is discussed under **Enabling and Disabling Special Character Interpretation** on page IV-116.

This table shows the special characters and their effect if special character interpretation is enabled. See **Special Menu Characters on Windows** on page IV-116 for Windows-specific considerations.

Character	Behavior
/	Creates a keyboard shortcut for the menu item. The character after the slash defines the item's keyboard shortcut. For example, `"Low Pass/1"` makes the item "Low Pass" with a keyboard shortcut for Command-1 (*Macintosh*) or Ctrl-1 (*Windows*). You can also use function keys. To avoid conflicts with Igor, use the numeric keys and the function keys only. See **Keyboard Shortcuts** on page IV-118 and **Function Keys** on page IV-118 for further details. Keyboard shortcuts are not supported in the graph marquee and layout marquee menus.
-	Creates a divider between menu items. If a hyphen (minus sign) is the first character in the item then the item will be a disabled divider. This can be a problem when trying to put negative numbers in a menu. Use a leading space character to prevent this. The string "(-" also disables the corresponding divider.
(	Disables the menu item. If a left parenthesis appears anywhere in the item then the item will be disabled.
!	Adds a mark to the menu item. If an exclamation point appears in the item, the character after the exclamation point will appear to the left of the menu item. For example, "Low Pass!*" makes an item "Low Pass" with an asterisk to the left. To mark an item with a check, use `"Low Pass!" + num2char(18)` This is necessary because the character code (18) for a check mark is a nonprinting control character that is not displayed correctly in most fonts.
<	Controls the typographic style of the item. This is rarely used since it tends to make the menu too garish.
^	Draws an icon as the menu item. This is not supported in Igor. Do not use it.
;	Separates one menu item from the next. Example: `"Item 1;Item 2"`

Whereas it is standard practice to use a semicolon to separate items in a pop-up menu in a control panel, graph or simple input dialog, you should avoid using the semicolon in user-defined main-menu-bar menus. It is clearer if you use one item per line. It is also necessary in some cases (see **Menu Definition Syntax** on page IV-107).

If a left angle bracket appears in the item, then the style of type for the item is controlled by the character following the angle bracket as follows:

Character Sequence	Makes Item
<B	bold
<I	italic
<U	underlined
<O	outlined
<S	shadowed

For example, "Low Pass<U" makes the item "Low Pass" with an underline.

If special character interpretation is disabled, these characters will appear in the menu item instead of having their special effect. The semicolon character is treated as a separator of menu items even if special character interpretation is disabled.

Special Menu Characters on Windows

On Windows, these characters are treated as special in menu bar menus but not in pop-up menus in graphs, control panels, and simple input dialogs. The following table shows which special characters are supported.

Character	Meaning	Status
/	Defines accelerator	Supported
-	Divider	Supported
(	Disables item	Supported
!	Adds mark to item	Partially supported — see discussion below
<	Controls typography	Not supported — ignored
^	Specifies icon	Not supported — ignored
;	Separates items	Supported

Windows does not directly support marking a menu item with anything other than a checkmark. Therefore, on Windows, when Igor sees the "!" special character, it checks the menu item regardless of which character follows the "!". The following user-defined menu item definition will produce a checked item on both platforms:

```
"Test!" + num2char(18)
```

"`num2char(18)`" returns the character code for a check mark on the Macintosh.

In general, Windows does not allow using Ctrl+<punctuation> as an accelerator. Therefore, in the following example, the accelerator will not do anything:

```
Menu "Macros"
   "Test/["       // "/[" will not work on Windows.
End
```

On Windows, you can designate a character in a menu item as a mnemonic keystroke by preceding the character with an ampersand:

```
Menu "Macros"
   "&Test", Print "This is a test"
End
```

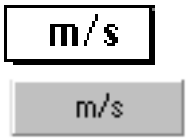
This designates "T" as the mnemonic keystroke for Test. To invoke this menu item, press Alt and release it, press the "M" key to highlight the Macros menu, and press the "T" key to invoke the Test item. If you hold the Alt key pressed while pressing the "M" and "T" keys and if the active window is a procedure window, Igor will not execute the item's command but rather will bring up the procedure window and display the menu definition. This is a programmer's shortcut.

Note: The mnemonic keystroke is not supported on Macintosh. If you use an ampersand in a menu item, it will appear in the menu item on the Macintosh. For this reason, if you care about cross-platform compatibility, you should not use ampersands in your menu items.

Enabling and Disabling Special Character Interpretation

The interpretation of special characters in menu items can sometimes get in the way. For example, you may want a menu item to say "m/s" or "A<B". With special character interpretation enabled, the first of these would become "m" with "s" as the keyboard shortcut and the second would become "A" in a bold typeface.

Igor provides WaveMetrics-defined escape sequences that allow you to override the default state of special character interpretation. These escape sequences are case sensitive and must appear at the very start of the menu item:

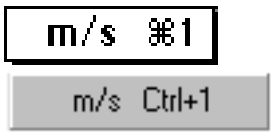
Escape Code	Effect	Example	
"\\M0"	Turns special character interpretation off.	"\\M0m/s"	
"\\M1"	Turns special character interpretation on.	"\\M1m/s"	

The most common use for this will be in a user-defined menu bar menu in which the default state is on and you want to display a special character in the menu item text itself. That is what you can do with the "\\M0" escape sequence.

Another possible use on Macintosh is to create a disabled menu item in a control panel, graph or simple input dialog pop-up menu. The default state of special character interpretation in pop-up menus is off. To disable the item, you either need to turn it on, using "\\M1" or to use the technique described in the next paragraph.

What if we want to include a special character in the menu item itself *and* have a keyboard shortcut for that item? The first desire requires that we turn special character interpretation off and the second requires that we turn it on. The WaveMetrics-defined escape sequence can be extended to handle this. For example:

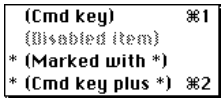
```
"\\M0:/1:m/s"
```



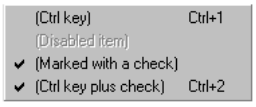
The initial "\\M0" turns normal special character interpretation off. The first colon specifies that one or more special characters are coming. The /1 makes Command-1 (*Macintosh*) or Ctrl+1 (*Windows*) the keyboard shortcut for this item. The second colon marks the end of the menu commands and starts the regular menu text which is displayed in the menu without special character interpretation. The final result is as shown above.

Any of the special characters can appear between the first colon and the second. For example:

```
Menu "Macros"
   "\\M0:/1:(Cmd key)"
   "\\M0:(:(Disabled item)"
   "\\M0:!*:(Marked with *)"
   "\\M0:/2!*:(Cmd key plus *)"
End
```




```
Menu "Macros"
   "\\M0:/1:(Ctrl key)"
   "\\M0:(:(Disabled item)"
   "\\M0:!"+num2char(18)+":(Marked with a check)"
   "\\M0:/2!"+num2char(18)+":(Ctrl key plus check)"
End
```



The \\M escape code affects just the menu item currently being defined. In the following example, special character interpretation is enabled for the first item but not for the second:

```
"\\M1(First item;(Second item"
```

To enable special character interpretation for both items, we need to write:

```
"\\M1(First item;\\M1(Second item"
```

Keyboard Shortcuts

A keyboard shortcut is a set of one or more keys which invoke a menu item. In a menu item string, a keyboard shortcut is introduced by the / special character. For example:

```
Menu "Macros"
   "Test/1"       // The keyboard shortcut is Cmd-1 (Macintosh)
End               // or Ctrl-1 (Windows).
```

All of the alphabetic keyboard shortcuts (/A…/Z) are used by Igor. Numeric keyboard shortcuts (/0…/9) are available for use in user menu definitions as are Function Keys, described below.

You can define a numeric keyboard shortcut that includes one or more modifier keys. The modifier keys are Shift and Option (*Macintosh*) or Alt (*Windows*). For example:

```
Menu "Macros"
   "Test/1"       // Cmd-1, Ctrl-1
   "Test/S1"      // Shift-Cmd-1, Ctrl-Shift-1.
   "Test/O1"      // Option-Cmd-1, Ctrl-Alt-1
   "Test/OS1"     // Option-Shift-Cmd-1, Ctrl-Shift-Alt-1
End
```

On Macintosh, “L” can be used to indicated that the Control key must be pressed. This is not supported on Windows.

Function Keys

Most keyboards have function keys labeled F1 through F12. In Igor, you can treat a function key as a keyboard shortcut that invokes a menu item.

Note: Mac OS X reserves nearly all function keys for itself. In order to use function keys for an application, you must check a checkbox in the Keyboard control panel. Even then the OS will intercept some function keys.

Note: On Windows, Igor uses F1 for help-related operations. F1 will not work as a keyboard shortcut on Windows. Also, the Windows OS reserves Ctrl-F4 and Ctrl-F6 for closing and reordering windows.

Here is a simple function key example:

```
Menu "Macros"
   "Test/F5"      // The keyboard shortcut is F5.
End
```

As with other keyboard shortcuts, you can specify that one or more modifier keys must be pressed along with the function key. The modifier keys are Shift and Option (*Macintosh*) or Alt (*Windows*).

```
Menu "Macros"
   // Function keys with and without modifiers
   "Test/F5"      // F5
   "Test/SF5"     // Shift-F5
   "Test/OF5"     // Option-F5, Alt-F5
   "Test/SOF5"    // Shift-Option-F5, Shift-Alt-F5
End
```

By including the “C” modifier character, you specify that the Command (*Macintosh*) or Ctrl (*Windows*) key must also be pressed:

```
Menu "Macros"
   // Cmd-Function or Ctrl-Function keys with and without modifiers
   "Test/CF5"     // Cmd-F5, Ctrl-F5
   "Test/SCF5"    // Shift-Cmd-F5, Ctrl-Shift-F5
   "Test/OCF5"    // Option-Cmd-F5, Ctrl-Alt-F5
   "Test/OSCF5"   // Option-Shift-Cmd-F5, Ctrl-Shift-Alt-F5
End
```

On Macintosh only, you can also use Control as a modifier by adding an “L” between the slash and the “F5”.

Although some keyboards have function keys labeled F13 and higher, they do not behave consistently and are not supported.

Marquee Menus

Igor has two menus called "marquee menus". In graphs and page layouts you create a marquee when you drag diagonally. Igor displays a dashed-line box indicating the area of the graph or layout that you have selected. If you click inside the marquee, you get a marquee menu.

You can add your own menu items to a marquee menu by creating a GraphMarquee or LayoutMarquee menu definition. For example:

```
Menu "GraphMarquee"
   "Print Marquee Coordinates", GetMarquee bottom; Print V_left, V_right
End
```

The use of keyboard shortcuts is not supported in marquee menus.

See **Marquee Menu as Input Device** on page IV-140 for details.

Trace Menus

Igor has two "trace" menus named "TracePopup" and "AllTracesPopup". When you control-click or right-click in a graph on a trace you get the TracesPopupMenu. If you hold down Shift while clicking, you get the AllTracesPopup (standard menu items in that menu operated on all the traces in the graph). You can append menu items to these menus with Menu "TracePopup" and Menu "AllTracesPopup" definitions.

For example:

```
Menu "TracePopup"
   "IdentifyTrace", /Q, IdentifyTrace()
End

Function IdentifyTrace()
   GetLastUserMenuInfo
   Print S_graphName, S_traceName
End
```

Popup Contextual Menus

You can create a custom pop-up contextual menu to respond to a control-click or right-click. For an example, see **Creating a Contextual Menu** on page IV-139.

Chapter
IV-6

Interacting with the User

Overview

The following sections describe the various programming techniques available for getting input from and for interacting with a user during the execution of your procedures. These techniques include:

- The simple input dialog
- Control panels
- Cursors
- Marquee menus

The simple input dialog provides a bare bones but functional user interfaces with just a little programming. In situations where more elegance is required, control panels provide a better solution.

Modal and Modeless User Interface Techniques

Before the rise of the graphical user interface, computer programs worked something like this: The user would start the program which would then ask the user for input. The program would then do some processing, after which it would ask the user for more input. In this model, the program is in charge and the user must respond with specific input at specific points of program execution. This is called a "modal" user interface because the program has one mode in which it will only accept specific input and another mode in which it will only do processing.

The Macintosh changed all this with the idea of event-driven programming. In this model, the computer waits for an event such as a mouse click or a key press and then acts on that event. The user is in charge and the program responds. This is called a "modeless" user interface because the program will accept any user action at any time.

You can use both techniques in Igor. Your program can put up a modal dialog asking for input and then do its processing or you can use control panels to build a sophisticated modeless event-driven system.

Event-driven programming is quite a bit more work than dialog-driven programming. You have to be able to handle user actions in any order rather than progressing through a predefined sequence of steps. In real life, a combination of these two methods is often used.

The Simple Input Dialog

The simple input dialog is a way by which a function can get input from the user in a modal fashion. It is very simple to program and is also somewhat simple in appearance.

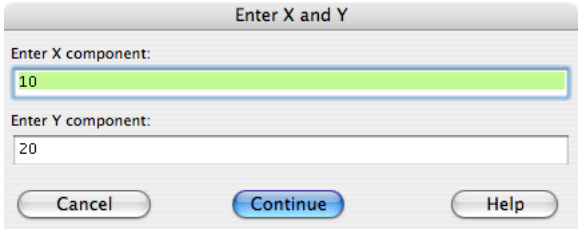
A simple input dialog is presented to the user when a DoPrompt statement is executed in a function. Parameters to DoPrompt specify the title for the dialog and a list of local variables. For each variable, you must include a Prompt statement that provides the text label for the variable.

Generally, the simple input dialog is used in conjunction with routines that run when the user chooses an item from a menu. This is illustrated in the following example which you can type into the procedure window of a new experiment:

```
Menu "Macros"
   "Calculate Diagonal...", CalcDiagDialog()
End

Function CalcDiagDialog()
   Variable x=10,y=20
   Prompt x, "Enter X component: "  // Set prompt for x param
   Prompt y, "Enter Y component: "  // Set prompt for y param
   DoPrompt "Enter X and Y", x, y
   if (V_Flag)
      return -1                     // User canceled
   endif
```

```
   Print "Diagonal=",sqrt(x^2+y^2)
End
```

If you run the CalcDiagDialog function, you will see the following dialog:



If the user presses Continue without changing the default values, "Diagonal= 22.3607" will be printed in the history. If the user presses Cancel, nothing will be printed because DoPrompt sets the V_Flag variable to 1 in this case.

The simple input dialog allows for up to 10 numeric or string variables. When more than 5 items are used, the dialog uses two columns and you may have to limit the length of your Prompt text.

The simple input dialog is unique in that you can enter not only literal numbers or strings but also numeric expressions or string expressions. Any literal strings that you enter must be quoted.

If the user presses the Help button, Igor will search for a help topic with a name derived from the dialog title. If such a help topic is not found, then generic help about the simple input dialog will be presented. In both cases, the input dialog will remain until the user presses either Continue or Cancel.

Pop-Up Menus in Simple Dialogs

The simple input dialog supports pop-up menus as well as text items. The pop-up menus can contain an arbitrary list of items such as a list of wave names. To use a pop-up menu in place of the normal text entry item in the dialog, you use the following syntax in the prompt declaration:

```
Prompt <variable name>, <title string>, popup <menu item list>
```

The popup keyword indicates that you want a pop-up menu instead of the normal text entry item. The menu list specifies the items in the pop-up menu separated by semicolons. For example:

```
Prompt color, "Select Color", popup "red;green;blue;"
```

If the menu item list is too long to fit on one line, you can compose the list in a string variable like so:

```
String stmp= "red;green;blue;"
stmp += "yellow;purple"
Prompt color, "Select Color", popup stmp
```

The pop-up menu items support the same special characters as the user-defined menu definition (see **Special Characters in Menu Item Strings** on page IV-114) except that items in pop-up menus are limited to 50 characters, keyboard shortcuts are not supported, and special characters are disabled by default.

You can use pop-up menus with both numeric and string parameters. When used with numeric parameters the number of the item chosen is placed in the variable. Numbering starts from one. When used with string parameters the text of the chosen item is placed in the string variable.

There are a number of functions, such as the **WaveList** function (see page V-725) and the **TraceNameList** function (see page V-710), that are useful in creating pop-up menus.

To obtain a menu item list of all waves in the current data folder, use:

```
WaveList("*", ";", "")
```

To obtain a menu item list of all waves in the current data folder whose names end in "_X", use:

```
WaveList("*_X", ";", "")
```

To obtain a menu item list of all traces in the top graph, use:

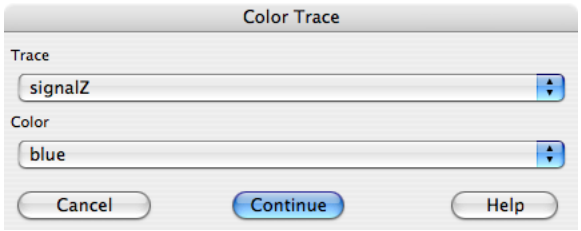
```
TraceNameList("", ";", 1)
```

For a list of all contours in the top graph, use ContourNameList. For a list of all images, use ImageNameList. For a list of waves in a table, use WaveList.

This next example creates two pop-up menus in the simple input dialog.

```
Menu "Macros"
   "Color Trace...", ColorTraceDialog()
End

Function ColorTraceDialog()
   String traceName
   Variable color=3
   Prompt traceName,"Trace",popup,TraceNameList("",";",1)
   Prompt color,"Color",popup,"red;green;blue"
   DoPrompt "Color Trace",traceName,color
   if( V_Flag )
      return 0          // user canceled
   endif

   if (color == 1)
      ModifyGraph rgb($traceName)=(65000, 0, 0)
   elseif(color == 2)
      ModifyGraph rgb($traceName)=(0, 65000, 0)
   elseif(color == 3)
      ModifyGraph rgb($traceName)=(0, 0, 65000)
   endif
End
```

If you choose Color Trace from the Macros menu, Igor brings up the simple input dialog with two pop-up menus. The first menu contains a list of all traces in the target window which is assumed to be a graph. The second menu contains the items red, green and blue with blue (item number 3) initially chosen.



After you choose the desired trace and color from the pop-up menus and click the Continue button, the function continues execution. The string parameter traceName will contain the name of the trace chosen from the first pop-up menu. The numeric parameter color will have a value of 1, 2 or 3, corresponding to red, green and blue.

In the preceding example, we needed a trace name to pass to the ModifyGraph operation. In another common situation, we need a wave reference to operate on. For example:

```
Menu "Macros"
   "Smooth Wave In Graph...",SmoothWaveInGraphDialog()
End

Function SmoothWaveInGraphDialog()
   String traceName
   Prompt traceName,"Wave",popup,TraceNameList("",";",1)
   DoPrompt "Smooth Wave In Graph",traceName
```

```
   WAVE w = TraceNameToWaveRef("", traceName)
   Smooth 5, w
End
```

The traceName parameter alone is not sufficient to specify which wave we want to smooth because it does not identify in which data folder the wave resides. The TraceNameToWaveRef function returns a wave reference which solves this problem. See **Wave Reference Functions** on page IV-173 for details.

Saving Parameters for Reuse

It is possible to write a procedure that presents a simple input dialog with default values for the parameters saved from the last time it was invoked. To accomplish this, we use global variables to store the values between calls to the procedure. Here is an example that saves one numeric and one string variable.

```
Function TestDialog()
   String savDF = GetDataFolder(1)
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S :TestDialog

   Variable num = NumVarOrDefault("gNum", 42)
   Prompt num, "Enter a number"
   String str = StrVarOrDefault("gStr", "Hello")
   Prompt str, "Enter a string"
   DoPrompt "test",num,str

   Variable/G gNum = num      // Save for next time
   String/G gStr = str

   // Put function body here
   Print num,str

   SetDataFolder savDF
End
```

This example illustrates the NumVarOrDefault and StrVarOrDefault functions. These functions return the value of a global variable or a default value if the global variable does not exist. 42 is the default value for gNum. NumVarOrDefault will return 42 if gNum does not exist. If gNum does exist, it will return the value of gNum. Similarly, “Hello” is the default value for gStr. StrVarOrDefault will return “Hello” if gStr does not exist. If gStr does exist, it will return the value of gStr.

Multiple Simple Input Dialogs

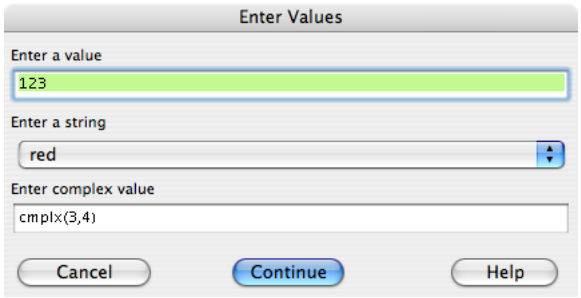
Prompt statements can be located anywhere within the body of a function and they do not need to be grouped together, although it will aid code readability if associated Prompt and DoPrompt code is kept together. Functions may contain multiple DoPrompt statements, and Prompt statements can be reused or redefined.

The following example illustrates multiple simple input dialogs and prompt reuse:

```
Function Example()
   Variable a= 123
   Variable/C ca= cmplx(3,4)
   String s

   Prompt a,"Enter a value"
   Prompt ca,"Enter complex value"
   Prompt s,"Enter a string", popup "red;green;blue"
   DoPrompt "Enter Values",a,s,ca
   if(V_Flag)
      Abort "The user pressed Cancel"
   endif

   Print "a= ",a,"s= ",s,"ca=",ca
```

```
   Prompt a,"Enter a again please"
   Prompt s,"Type a string"
   DoPrompt "Enter Values Again", a,s

   if(V_Flag)
      Abort "The user pressed Cancel"
   endif

   Print "Now a=",a," and s=",s
End
```

When this function is executed, it will produce two simple input dialogs, one after the other after the user clicks Continue; following is the first dialog:



Displaying an Open File Dialog

You can display an Open File dialog to allow the user to choose a file to be used with a subsequent command. For example, the user can choose a file which you will then use in a **LoadWave** command. The Open File dialog is displayed using an **Open**/D/R command. Here is an example:

```
Function/S DoOpenFileDialog()
   Variable refNum
   String message = "Select a file"
   String outputPath
   String fileFilters = "Data Files (*.txt,*.dat,*.csv):.txt,.dat,.csv;"
   fileFilters += "All Files:.*;"

   Open /D /R /F=fileFilters /M=message refNum
   outputPath = S_fileName

   return outputPath    // Will be empty if user canceled
End
```

Here the Open operation does not actually open a file but instead displays an Open File dialog. If the user chooses a file and clicks the Open button, the Open operation returns the full path to the file in the S_fileName output string variable. If the user cancels, Open sets S_fileName to "".

The /M flag is used to set the prompt message.

The /F flag is used to control the file filter which determines what kinds of files the user can select. This is explained further under **Open File Dialog File Filters**.

Displaying a Multi-Selection Open File Dialog

In Igor Pro 6.1 or later, you can display an Open File dialog to allow the user to choose multiple files to be used with subsequent commands. The multi-selection Open File dialog is displayed using an **Open**/D/R/MULT=1 command. The list of files selected is returned via S_fileName in the form of a carriage-return-delimited list of full paths.

Here is an example:

```
Function/S DoOpenMultiFileDialog()
   Variable refNum
   String message = "Select one or more files"
   String outputPaths
   String fileFilters = "Data Files (*.txt,*.dat,*.csv):.txt,.dat,.csv;"
   fileFilters += "All Files:.*;"

   Open /D /R /MULT=1 /F=fileFilters /M=message refNum
   outputPaths = S_fileName

   if (strlen(outputPaths) == 0)
      Print "Cancelled"
   else
      Variable numFilesSelected = ItemsInList(outputPaths, "\r")
      Variable i
      for(i=0; i<numFilesSelected; i+=1)
         String path = StringFromList(i, outputPaths, "\r")
         Printf "%d: %s\r", i, path
      endfor
   endif

   return outputPaths   // Will be empty if user canceled
End
```

Here the Open operation does not actually open a file but instead displays an Open File dialog. Because /MULT=1 was used, if the user chooses one or more files and clicks the Open button, the Open operation returns the list of full paths to files in the S_fileName output string variable. If the user cancels, Open sets S_fileName to "".

The list of full paths is delimited with a carriage return character, represented by "\r" in the example above. We use carriage return as the delimiter because the customary delimiter, semicolon, is a legal character in a Macintosh file name.

The /M flag is used to set the prompt message.

The /F flag is used to control the file filter which determines what kinds of files the user can select. This is explained further under **Open File Dialog File Filters**.

Open File Dialog File Filters

The **Open** operation displays the open file dialog if you use the /D/R flags or if the file to be opened is not fully specified using the pathName and fileNameStr parameters. The Open File dialog includes a file filter menu that allows the user to choose the type of file to be opened. By default this menus contain "Plain Text Files" and "All Files". You can use the /T and /F flags to override the default filter behavior.

The /T flag uses obsolescent Macintosh file types or file name extensions consisting of a dot plus three characters. The /F flag, added in Igor Pro 6.10, supports file name extensions only (not Macintosh file types) and extensions can be from one to 31 characters. Procedures written for Igor Pro 6.10 or later should use the /F flag in most cases but can use /T or both /T and /F. Procedures that must run with Igor Pro 6.0x and earlier must use the /T flag.

Using the /T=typeStr flag, you specify acceptable Macintosh-style file types represented by four-character codes (e.g., "TEXT") or acceptable three-character file name extensions (e.g., ".txt"). The pattern "????" means "any type of file" and is represented by "All Files" in the filter menu.

typeStr may contain multiple file types or extensions (e.g., "TEXTEPSF????" or ".txt.eps????"). Each file type or extension must be exactly four characters in length. Consequently the /T flag can accommodate only three-character file name extensions. Each file type or extension creates one entry in the Open File dialog filter menu.

If you use the /T flag, the Open operation automatically adds a filter for All Files ("????") if you do not add one explicitly.

Igor maps Macintosh file types to extensions. For example, if you specify /T="TEXT", you can open files with the extension ".txt" as well as any file whose Macintosh file type property is 'TEXT'. Igor does similar mappings for other extensions. See **File Types and Extensions** on page III-404 for details.

Using the /F=fileFilterStr flag, you specify a filter menu string plus acceptable file name extensions for each filter. fileFilterStr specifies one or more filters in a semicolon-separated list. For example, this specifies three filters:

```
String fileFilters = "Data Files (*.txt,*.dat,*.csv):.txt,.dat,.csv;"
fileFilters += "HTML Files (*.htm,*.html):.htm,.html;"
fileFilters += "All Files:.*;"
Open /F=fileFilters . . .
```

Each file filter consists of a filter menu string (e.g., "Data Files") followed by a colon, followed by one or more file name extensions (e.g., ".txt,.dat,.csv") followed by a semicolon. The syntax is rigid - no extra characters are allowed. In this example the filter menu would contain "Data Files" and would accept any file with a ".txt", ".dat", or ".csv" extension. ".*" creates a filter that accepts any file.

If you use the /F flag, it is up to you to add a filter for All Files as shown above. It is recommended that you do this.

On the Macintosh, selecting All Files allows you to navigate into packages (folders that appear in the Finder to be files).

Displaying a Save File Dialog

You can display a Save File dialog to allow the user to choose a file to be created or overwritten by a subsequent command. For example, the user can choose a file which you will then create or overwrite via a Save command. The Save File dialog is displayed using an **Open**/D command. Here is an example:

```
Function/S DoSaveFileDialog()
   Variable refNum
   String message = "Save a file"
   String outputPath
   String fileFilters = "Data Files (*.txt):.txt;"
   fileFilters += "All Files:.*;"

   Open /D /F=fileFilters /M=message refNum
   outputPath = S_fileName

   return outputPath    // Will be empty if user canceled
End
```

Here the Open operation does not actually open a file but instead displays a Save File dialog. If the user chooses a file and clicks the Save button, the Open operation returns the full path to the file in the S_fileName output string variable. If the user cancels, Open sets S_fileName to "".

The /M flag is used to set the prompt message.

The /F flag is used to control the file filter which determines what kinds of files the user can create. This is explained further under **Save File Dialog File Filters**.

Save File Dialog File Filters

The Save File dialog includes a file filter menu that allows the user to choose the type of file to be saved. By default this menus contain "Plain Text File" and, on Windows only, "All Files". You can use the /T and /F flags to override the default filter behavior.

The /T and /F flags work as explained under Open File Dialog File Filters. Using the /F flag for a Save File dialog, you would typically specify just one filter plus All Files, like this:

String fileFilters = "Data File (*.dat):.dat;"

fileFilters += "All Files:.*;"

Open /F=fileFilters . . .

On Windows, the file filter chosen in the Save File dialog determines the extension for the file being saved. For example, if the "Plain Text Files" filter is selected, the ".txt" extension is added if you don't explicitly enter it in the File Name edit box. However if you select the "All Files" filter then no extension is automatically added and the final file name is whatever you enter in the File Name edit box. You should include the "All Files" filter if you want the user to be able to specify a file name with any extension. If you want to force the file name extension to an extension of your choice rather than the user's, omit the "All Files" filter.

On Macintosh, the final file name is whatever you enter in the Save As edit box. The only significance of the selected file filter is that it controls the extension used if you click the Add/Fix Extension button. "All Files" has no significance in the Macintosh Save File dialog and therefore is ignored if it is the last file type specified using /T.

On the Macintosh, in the rare event that you want to save a file inside a package (a folder that appears as a file in the Finder), press Command-Option as the dialog is presented. This will allow you to navigate inside packages.

Using Open in a Utility Routine

To be as general and useful as possible, a utility routine that acts on a file should have a pathName parameter and a fileName parameter, like this:

```
Function ShowFileInfo(pathName, fileName)
   String pathName   // Name of symbolic path or "" for dialog.
   String fileName   // File name or "" for dialog.

   <Show file info here>
End
```

This provides flexibility to the calling function. The caller can supply a valid symbolic path name and a simple leaf name in fileName, a valid symbolic path name and a partial path in fileName, or a full path in fileName in which case pathName is irrelevant.

If pathName and fileName fully specify the file of interest, you want to just open the file and perform the requested action. However, if pathName and fileName do not fully specify the file of interest, you want to display an Open File dialog so the user can choose the file. This is accomplished by using the **Open** operation's /D=2 flag (added in Igor Pro 6.1).

With /D=2, if pathName and fileName fully specify the file, the Open operation merely sets the S_fileName output string variable to the full path to the file. If pathName and fileName do not fully specify the file, Open displays an Open File dialog and then sets the S_fileName output string variable to the full path to the file. If the user cancels the Open File dialog, Open sets S_fileName to "". In all cases, Open/D=2 just sets S_fileName and does not actually open the file.

If pathName and fileName specify an alias (*Macintosh*) or shortcut (*Windows*), Open/D=2 returns the file referenced by the alias or shortcut.

Here is how you would use Open /D=2.

```
Function ShowFileInfo(pathName, fileName)
   String pathName      // Name of symbolic path or "" for dialog.
   String fileName      // File name or "" for dialog.
```

```
   Variable refNum

   Open /D=2 /R /P=$pathName refNum as fileName    // Sets S_fileName

   if (strlen(S_fileName) == 0)
      Print "ShowFileInfo was canceled"
   else
      String fullPath = S_fileName
      Print fullPath
      Open /R refNum as fullPath
      FStatus refNum    // Sets S_info
      Print S_info
      Close refNum
   endif
End
```

In this case, we wanted to open the file for reading. To create a file and open it for writing, omit /R from both calls to Open.

Pause For User

The **PauseForUser** operation (see page V-476) allows an advanced programmer to create a more sophisticated semimodal user interface. When you invoke it from a procedure, Igor suspends procedure execution and the user can interact with graph, table or control panel windows using the mouse or keyboard. Execution continues when the user kills the main window specified in the PauseForUser command.

Pausing execution can serve two purposes. First, the programmer can pause function execution so that the user can, for example, adjust cursors in a graph window before continuing with a curve fit. In this application, the programmer creates a control panel with a continue button that the user presses after adjusting the cursors in the target graph. Pressing the continue button kills the host control panel (see example below).

In the second application, the programmer may wish to obtain input from the user in a more sophisticated manner than can be done using DoPrompt commands. This method uses a control panel as the main window with no optional target window. It is similar to the control panel technique shown above, except that it is modal.

Following are some examples of how you can use the **PauseForUser** operation (see page V-476) in your own user functions.

PauseForUser Simple Cursor Example

This example shows how to allow the user to adjust cursors on a graph while a procedure is executing. Most of the work is done by the UserCursorAdjust function. UserCursorAdjust is called by the Demo function which first creates a graph and shows the cursor info panel.

This example illustrates two modes of PauseForUser. When called with autoAbortSecs=0, UserCursorAdjust calls PauseForUser without the /C flag in which case PauseForUser retains control until the user clicks the Continue button.

When called with autoAbortSecs>0, UserCursorAdjust calls PauseForUser/C. This causes PauseForUser to handle any pending events and then return to the calling procedure. The procedure checks the V_flag variable, set by PauseForUser, to determine when the user has finished interacting with the graph. PauseForUser/C, which requires Igor Pro 6.1 or later, is for situations where you want to do something while the user interacts with the graph.

To try this yourself, copy and paste all three routines routines below into the procedure window of a new experiment and then run the Demo function with a value of 0 and again with a value such as 30.

```
Function UserCursorAdjust(graphName,autoAbortSecs)
   String graphName
   Variable autoAbortSecs

   DoWindow/F $graphName                  // Bring graph to front
```

```
   if (V_Flag == 0)                        // Verify that graph exists
      Abort "UserCursorAdjust: No such graph."
      return -1
   endif

   NewPanel /K=2 /W=(187,368,437,531) as "Pause for Cursor"
   DoWindow/C tmp_PauseforCursor          // Set to an unlikely name
   AutoPositionWindow/E/M=1/R=$graphName  // Put panel near the graph

   DrawText 21,20,"Adjust the cursors and then"
   DrawText 21,40,"Click Continue."
   Button button0,pos={80,58},size={92,20},title="Continue"
   Button button0,proc=UserCursorAdjust_ContButtonProc
   Variable didAbort= 0
   if( autoAbortSecs == 0 )
      PauseForUser tmp_PauseforCursor,$graphName
   else
      SetDrawEnv textyjust= 1
      DrawText 162,103,"sec"
      SetVariable sv0,pos={48,97},size={107,15},title="Aborting in "
      SetVariable sv0,limits={-inf,inf,0},value= _NUM:10
      Variable td= 10,newTd
      Variable t0= ticks
      Do
         newTd= autoAbortSecs - round((ticks-t0)/60)
         if( td != newTd )
            td= newTd
            SetVariable sv0,value= _NUM:newTd,win=tmp_PauseforCursor
            if( td <= 10 )
               SetVariable sv0,valueColor= (65535,0,0),win=tmp_PauseforCursor
            endif
         endif
         if( td <= 0 )
            DoWindow/K tmp_PauseforCursor
            didAbort= 1
            break
         endif

         PauseForUser/C tmp_PauseforCursor,$graphName
      while(V_flag)
   endif
   return didAbort
End

Function UserCursorAdjust_ContButtonProc(ctrlName) : ButtonControl
   String ctrlName

   DoWindow/K tmp_PauseforCursor             // Kill panel
End

Function Demo(autoAbortSecs)
   Variable autoAbortSecs

   Make/O jack;SetScale x,-5,5,jack
   jack= exp(-x^2)+gnoise(0.1)
   DoWindow Graph0
   if( V_Flag==0 )
      Display jack
      ShowInfo
   endif
```

```
   if (UserCursorAdjust("Graph0",autoAbortSecs) != 0)
      return -1
   endif

   if (strlen(CsrWave(A))>0 && strlen(CsrWave(B))>0)// Cursors are on trace?
      CurveFit gauss,jack[pcsr(A),pcsr(B)] /D
   endif
End
```

PauseForUser Advanced Cursor Example

Now for something a bit more complex. Here we modify the preceding example to include a Cancel button. For this, we need to return information about which button was pressed. Although we could do this by creating a single global variable in the root data folder, we use a slightly more complex technique using a temporary data folder. This technique is especially useful for more complex panels with multiple output variables because it limits any worry about name conflicts to the data folder itself. It also allows much easier clean up because we can kill the entire data folder and everything in it with just one operation.

```
Function UserCursorAdjust(graphName)
   String graphName

   DoWindow/F $graphName   // Bring graph to front
   if (V_Flag == 0)        // Verify that graph exists
      Abort "UserCursorAdjust: No such graph."
      return -1
   endif

   NewDataFolder/O root:tmp_PauseforCursorDF
   Variable/G root:tmp_PauseforCursorDF:canceled= 0

   NewPanel/K=2 /W=(139,341,382,450) as "Pause for Cursor"
   DoWindow/C tmp_PauseforCursor          // Set to an unlikely name
   AutoPositionWindow/E/M=1/R=$graphName  // Put panel near the graph

   DrawText 21,20,"Adjust the cursors and then"
   DrawText 21,40,"Click Continue."
   Button button0,pos={80,58},size={92,20},title="Continue"
   Button button0,proc=UserCursorAdjust_ContButtonProc
   Button button1,pos={80,80},size={92,20}
   Button button1,proc=UserCursorAdjust_CancelBProc,title="Cancel"

   PauseForUser tmp_PauseforCursor,$graphName

   NVAR gCaneled= root:tmp_PauseforCursorDF:canceled
   Variable canceled= gCaneled   // Copy from global to local
                                 // before global is killed
   KillDataFolder root:tmp_PauseforCursorDF

   return canceled
End

Function UserCursorAdjust_ContButtonProc(ctrlName) : ButtonControl
   String ctrlName

   DoWindow/K tmp_PauseforCursor    // Kill self
End

Function UserCursorAdjust_CancelBProc(ctrlName) : ButtonControl
   String ctrlName

   Variable/G root:tmp_PauseforCursorDF:canceled= 1
   DoWindow/K tmp_PauseforCursor    // Kill self
End
```

And now a demo that uses the new version:

```
Function Demo()
   Make/O jack;SetScale x,-5,5,jack
   jack= exp(-x^2)+gnoise(0.1)
   DoWindow Graph0
   if (V_Flag==0)
      Display jack
      ShowInfo
   endif
   Variable rval= UserCursorAdjust("Graph0")
   if (rval == -1)            // Graph name error?
      return -1;
   endif
   if (rval == 1)             // User canceled?
      DoAlert 0,"Canceled"
      return -1;
   endif
   CurveFit gauss,jack[pcsr(A),pcsr(B)] /D
End
```

PauseForUser Control Panel Example

The following is a very simple example of using a control panel as modal dialog. The panel was designed by first manually creating a data folder with a few variables and then creating a panel. When the panel was designed properly, it was closed to create a recreation macro. Lines from the macro were then used in the body of the function.

```
Function UserGetInputPanel_ContButton(ctrlName) : ButtonControl
   String ctrlName

   DoWindow/K tmp_GetInputPanel     // kill self
End

// Call with these variables already created and initialized:
//    root:tmp_PauseForUserDemo:numvar
//    root:tmp_PauseForUserDemo:strvar
Function DoMyInputPanel()
   NewPanel /W=(150,50,358,239)
   DoWindow/C tmp_GetInputPanel        // set to an unlikely name
   DrawText 33,23,"Enter some data"
   SetVariable setvar0,pos={27,49},size={126,17},limits={-Inf,Inf,1}
   SetVariable setvar0,value= root:tmp_PauseForUserDemo:numvar
   SetVariable setvar1,pos={24,77},size={131,17},limits={-Inf,Inf,1}
   SetVariable setvar1,value= root:tmp_PauseForUserDemo:strvar
   Button button0,pos={52,120},size={92,20}
   Button button0,proc=UserGetInputPanel_ContButton,title="Continue"

   PauseForUser tmp_GetInputPanel
End

Function Demo1()
   NewDataFolder/O root:tmp_PauseForUserDemo
   Variable/G root:tmp_PauseForUserDemo:numvar= 12
   String/G root:tmp_PauseForUserDemo:strvar= "hello"

   DoMyInputPanel()

   NVAR numvar= root:tmp_PauseForUserDemo:numvar
   SVAR strvar= root:tmp_PauseForUserDemo:strvar

   printf "You entered %g and %s\r",numvar,strvar

   KillDataFolder root:tmp_PauseForUserDemo
End
```

For comparison, here is the equivalent using the simple input dialog technique:

```
Function Demo2()
   Variable numvar= 12
   String strvar= "hello"
   Prompt numvar,"numvar:"
   Prompt strvar,"strvar:"
   DoPrompt "Enter some data",numvar,strvar
   printf "You entered %g and %s\r",numvar,strvar
End
```

Progress Windows

Sometimes when performing a long calculation, you may want to put up an indicator that the calculation is in progress, perhaps showing how far along it is, and perhaps providing an abort button. As of Igor Pro 6.1, you can use a control panel window for this task using the **DoUpdate** /E and /W flags and the mode=4 setting for **ValDisplay**.

DoUpdate /W=win /E=1 marks the specified window as a progress window that can accept mouse events while user code is executing. The /E flag need be used only once to mark the panel but it does not hurt to use it in every call. This special state of the control panel is automatically cleared when procedure execution finishes and Igor's outer loop again runs.

For a window marked as a progress window, DoUpdate sets V_Flag to 2 if a mouse up happened in a button since the last call. When this occurs, the full path to the subwindow containing the button is stored in S_path and the name of the control is stored in S_name.

Here is a simple example that puts up a progress window with a progress bar and a quit button. Try each of the four input flag combinations.

```
// Try simpletest(0,0) and simpletest(1,0), simpletest(0,1) and simpletest(1,1)
Function simpletest(indefinite, useIgorDraw)
   Variable indefinite
   Variable useIgorDraw// True to use Igor's own draw method rather than native

   NewPanel /N=ProgressPanel /W=(285,111,739,193)
   ValDisplay valdisp0,pos={18,32},size={342,18}
   ValDisplay valdisp0,limits={0,100,0},barmisc={0,0}
   ValDisplay valdisp0,value= _NUM:0
   if( indefinite )
      ValDisplay valdisp0,mode= 4// candy stripe
   else
      ValDisplay valdisp0,mode= 3// bar with no fractional part
   endif
   if( useIgorDraw )
      ValDisplay valdisp0,highColor=(0,65535,0)
   endif
   Button bStop,pos={375,32},size={50,20},title="Stop"
   DoUpdate /W=ProgressPanel /E=1// mark this as our progress window

   Variable i,imax= indefinite ? 10000 : 100
   for(i=0;i<imax;i+=1)
      Variable t0= ticks
      do
      while( ticks < (t0+3) )
      if( indefinite )
         ValDisplay valdisp0,value= _NUM:1,win=ProgressPanel
      else
         ValDisplay valdisp0,value= _NUM:i+1,win=ProgressPanel
      endif
      DoUpdate /W=ProgressPanel
      if( V_Flag == 2 )// we only have one button and that means stop
         break
```

```
      endif
   endfor
   KillWindow ProgressPanel
End
```

When performing complex calculations, it is often difficult to insert DoUpdate calls in the code. In this case, you can use a window hook that responds to event #23, spinUpdate. This is called at the same time that the beachball cursor spins. The hook can then update the window's control state and then call DoUpdate/W on the window. If the window hook returns non-zero, then an abort is performed. If you desire a more controlled quit, you might set a global variable that your calculation code can test. The following example provides an indefinite indicator and an abort button. Note that if the abort button is pressed, the window hook kills the progress window since otherwise the abort would cause the window to remain.

```
// Example: spinnertest(100)
Function spinnertest(nloops)
   Variable nloops

   Variable useIgorDraw=0  // set true for Igor draw method rather than native

   NewPanel/FLT /N=myProgress/W=(285,111,739,193)
   ValDisplay valdisp0,pos={18,32},size={342,18}
   ValDisplay valdisp0,limits={0,100,0},barmisc={0,0}
   ValDisplay valdisp0,value= _NUM:0
   ValDisplay valdisp0,mode=4    // candy stripe
   if( useIgorDraw )
      ValDisplay valdisp0,highColor=(0,65535,0)
   endif
   Button bStop,pos={375,32},size={50,20},title="Abort"
   SetActiveSubwindow _endfloat_
   DoUpdate/W=myProgress/E=1     // mark this as our progress window

   SetWindow myProgress,hook(spinner)= MySpinner

   Variable t0= ticks,i
   for(i=0;i<nloops;i+=1)
      PerformLongCalc(1e6)
   endfor
   Variable timeperloop= (ticks-t0)/(60*nloops)

   KillWindow myProgress

   print "time per loop=",timeperloop
End

Function MySpinner(s)
   STRUCT WMWinHookStruct &s

   if( s.eventCode == 23 )
      ValDisplay valdisp0,value= _NUM:1,win=$s.winName
      DoUpdate/W=$s.winName
      if( V_Flag == 2 )    // we only have one button and that means abort
         KillWindow $s.winName
         return 1
      endif
   endif
   return 0
End

Function PerformLongCalc(nmax)
   Variable nmax
```

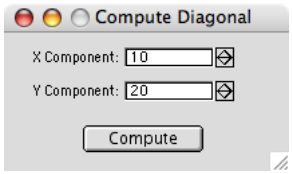
```
   Variable i,s
   for(i=0;i<nmax;i+=1)
      s+= sin(i/nmax)
   endfor
End
```

Control Panels and Event-Driven Programming

The CalcDiagDialog function shown above creates a modal dialog. "Modal" means that the function retains complete control until the user clicks Cancel or Continue. The user can not activate another window or choose a menu item until the dialog is dismissed.

This section shows how to implement the same functionality using a control panel as a modeless dialog. "Modeless" means that the user can activate another window or choose a menu item at any time. The modeless window accepts input whenever the user wants to enter it but does not block the user from accessing other windows.

The control panel looks like this:



The code implementing this control panel is given below. Before we look at the code, here is some explanation of the thinking behind it.

The X Component and Y Component controls are SetVariable controls. Each SetVariable control must be attached to a global variable. To keep the global variables from cluttering the user's space, we buried them in a data folder named root:Packages:DiagonalControlPanel.

We want the globals to be automatically created so the user does not need to worry about them, so we wrote a routine (DisplayDiagonalControlPanel) that makes sure that the variables and their containing data folders exist. The routine then creates the control panel or, if it already exists, just brings it to the front. We added a menu item to the Macros menu so the user can easily invoke DisplayDiagonalControlPanel.

We built the control panel manually using techniques explained in Chapter III-14, **Controls and Control Panels**. Then we closed it so Igor would create a display recreation macro which we named DiagonalControlPanel. We then manually tweaked the macro to attach the SetVariable controls to the desired globals and to set the panel's behavior when the user clicks the close button.

Here are the procedures.

```
// Add a menu item to display the control panel.
Menu "Macros"
   "Display Diagonal Control Panel", DisplayDiagonalControlPanel()
End

// This is the display recreation macro, created by Igor
// and then manually tweaked. The parts that were tweaked
// are shown in bold. NOTE: Some lines are wrapped to fit on the page.
Window DiagonalControlPanel() : Panel
   PauseUpdate; Silent 1   // building window...

   NewPanel/W=(162,95,375,198)/K=1 as "Compute Diagonal"

   SetVariable XSetVar,pos={22,11},size={150,15},title="X Component:"
   SetVariable XSetVar,limits={-Inf,Inf,1},value=
                  root:Packages:DiagonalControlPanel:gXComponent
```

```
   SetVariable YSetVar,pos={22,36},size={150,15},title="Y Component:"
   SetVariable YSetVar,limits={-Inf,Inf,1},value=
                  root:Packages:DiagonalControlPanel:gYComponent

   Button ComputeButton,pos={59,69},size={90,20},
                        proc=ComputeDiagonalProc,title="Compute"
EndMacro

// This is the action procedure for the Compute button.
// We created it using the Button dialog.
Function ComputeDiagonalProc(ctrlName) : ButtonControl
   String ctrlName

   String dfSave = GetDataFolder(1)
   SetDataFolder root:Packages:DiagonalControlPanel

   NVAR gXComponent, gYComponent    // Access current data folder.
   Variable diagonal
   diagonal = sqrt(gXComponent^2 + gYComponent^2)
   Printf "Diagonal=%g\r", diagonal

   SetDataFolder dfSave
End

// This is the top level routine which makes sure that the globals
// and their enclosing data folders exist and then makes sure that
// the control panel is displayed.
Function DisplayDiagonalControlPanel()
   // If the panel is already created, just bring it to the front.
   DoWindow/F DiagonalControlPanel
   if (V_Flag != 0)
      return 0
   endif

   String dfSave = GetDataFolder(1)

   // Create a data folder in Packages to store globals.
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S root:Packages:DiagonalControlPanel

   // Create global variables used by the control panel.
   Variable xComponent = NumVarOrDefault(":gXComponent", 10)
   Variable/G gXComponent = xComponent
   Variable yComponent = NumVarOrDefault(":gYComponent", 20)
   Variable/G gYComponent = yComponent

   // Create the control panel.
   Execute "DiagonalControlPanel()"

   SetDataFolder dfSave
End
```

Although this example is very simple, it illustrates the process of creating a control panel that functions as a modeless dialog. There are many more examples of this in the Examples folder. See Chapter III-14, **Controls and Control Panels**, for more information on building control panels.

Detecting a User Abort

If you have written a user-function that takes a long time to execute, you may want to provide a way for the user to abort it. Your first instinct might be to create a control panel with a Stop button. This won't work

because Igor does not look at controls while your user function is running. Instead, you must detect that the user is pressing Escape, as this example illustrates:

```
Function PressEscapeToAbort(phase, title, message)
   Variable phase    // 0: Display control panel with message.
                     // 1: Test if Escape key is pressed.
                     // 2: Close control panel.
   String title      // Title for control panel.
   String message    // Tells user what you are doing.

   if (phase == 0)   // Create panel
      DoWindow/F PressEscapePanel
      if (V_flag == 0)
         NewPanel/K=1 /W=(100,100,350,200)
         DoWindow/C PressEscapePanel
         DoWindow/T PressEscapePanel, title
      endif
      TitleBox Message,pos={7,8},size={69,20},title=message
      String abortStr = "Press escape to abort"
      TitleBox Press,pos={6,59},size={106,20},title=abortStr
      DoUpdate
   endif

   if (phase == 1)   // Test for Escape key
      Variable doAbort = 0
      if (GetKeyState(0) & 32)      // Is Escape key pressed now?
         doAbort = 1
      else
         if (strlen(message) != 0)  // Want to change message?
            TitleBox Message,title=message
            DoUpdate
         endif
      endif
      return doAbort
   endif

   if (phase == 2)   // Kill panel
      DoWindow/K PressEscapePanel
   endif

   return 0
End

Function Demo()
   // Create panel
   PressEscapeToAbort(0, "Demonstration", "This is a demo")

   Variable startTicks = ticks
   Variable endTicks = startTicks + 10*60
   Variable lastMessageUpdate = startTicks

   do
      String message
      message = ""
      if (ticks>=lastMessageUpdate+60) // Time to update message?
         Variable remaining = (endTicks - ticks) / 60
         sprintf message, "Time remaining: %.1f seconds", remaining
         lastMessageUpdate = ticks
      endif

      if (PressEscapeToAbort(1, "", message))
         Print "Test aborted by Escape key."
         break
```

```
      endif
   while(ticks < endTicks)

   PressEscapeToAbort(2, "", "")       // Kill panel.
End
```

Creating a Contextual Menu

You can use the **PopupContextualMenu** operation to create a pop-up menu in response to a control-click (*Macintosh*) or right-click. You would do this from a window hook function or from the action procedure for a control in a control panel.

In this example, we create a control panel with a list. When the user right-clicks on the list, Igor sends a mouse-down event to the listbox procedure, TickerListProc in this case. The listbox procedure uses the eventMod field of the WMListboxAction structure to determine if the click is a right-click. If so, it calls HandleTickerListRightClick which calls PopupContextualMenu to display the contextual menu.

```
Menu "Macros"

   "Show Demo Panel", ShowDemoPanel()
End

static Function HandleTickerListRightClick()
   String popupItems = ""
   popupItems += "Refresh;"

   PopupContextualMenu popupItems
   strswitch (S_selection)
      case "Refresh":
         DoAlert 0, "Here is where you would refresh the ticker list."
         break
   endswitch
End

Function TickerListProc(lba) : ListBoxControl
   STRUCT WMListboxAction &lba

   switch (lba.eventCode)
      case 1:                     // Mouse down
         if (lba.eventMod & 0x10)// Right-click?
            HandleTickerListRightClick()
         endif
         break
   endswitch

   return 0
End

Function ShowDemoPanel()
   DoWindow/F DemoPanel
   if (V_flag != 0)
      return 0 // Panel already exists.
   endif

   // Create panel data.
   Make/O/T ticketListWave = {{"AAPL","IBM","MSFT"}, {"90.25","86.40","17.17"}}

   // Create panel.
   NewPanel /N=DemoPanel /W=(321,121,621,321) /K=1
   ListBox TickerList,pos={48,16},size={200,100},fSize=12
   ListBox TickerList,listWave=root:ticketListWave
```

```
   ListBox TickerList,mode= 1,selRow= 0, proc=TickerListProc
End
```

Cursors as Input Device

You can use the cursors on a trace in a graph to identify the data to be processed.

The examples shown above using PauseForUser are modal - the user adjusts the cursors in the middle of procedure execution and can do nothing else. This technique is nonmodal — the user is expected to adjust the cursors before invoking the procedure.

This function does a straight-line curve fit through the data between cursor A (the round cursor) and cursor B (the square cursor). This example is written to handle both waveform and XY data.

```
Function FitLineBetweenCursors()
   Variable isXY

   // Make sure both cursors are on the same wave.
   WAVE wA = CsrWaveRef(A)
   WAVE wB = CsrWaveRef(B)
   String dfA = GetWavesDataFolder(wA, 2)
   String dfB = GetWavesDataFolder(wB, 2)
   if (CmpStr(dfA, dfB) != 0)
      Abort "Both cursors must be on the same wave."
      return -1
   endif

   // Find the wave that the cursors are on.
   WAVE yWave = CsrWaveRef(A)

   // Decide if this is an XY pair.
   WAVE xWave = CsrXWaveRef(A)
   isXY = WaveExists(xWave)

   if (isXY)
      CurveFit line yWave(xcsr(A),xcsr(B)) /X=xWave /D
   else
      CurveFit line yWave(xcsr(A),xcsr(B)) /D
   endif
End
```

This technique is demonstrated in the **Fit Line Between Cursors** example experiment in the "Examples:Curve Fitting" folder.

Advanced programmers can set things up so that a hook function is called whenever the user adjusts the position of a cursor. For details, see **Cursors — Moving Cursor Calls Function** on page IV-294.

Marquee Menu as Input Device

A marquee is the dashed-line rectangle that you get when you click and drag diagonally in a graph or page layout. It is used for expanding and shrinking the range of axes and for specifying an area of a layout. You can use the marquee as an input device for your procedures. This is a relatively advanced technique.

This menu definition adds a user-defined item to the graph marquee menu:

```
Menu "GraphMarquee"
   "Print Marquee Coordinates", PrintMarqueeCoords()
End
```

To add an item to the layout marquee menu, use LayoutMarquee instead of GraphMarquee.

When the user chooses Print Marquee Coordinates, the following function runs. It prints the coordinates of the marquee in the history area. It assumes that the graph has left and bottom axes.

```
Function PrintMarqueeCoords()
   String format
   GetMarquee/K left, bottom
   format = "flag: %g; left: %g; top: %g; right: %g; bottom: %g\r"
   printf format, V_flag, V_left, V_top, V_right, V_bottom
End
```

The ability to add items to the marquee menus through GraphMarquee and LayoutMarquee menu definitions was added in Igor Pro 5. Before that, the GraphMarquee and LayoutMarquee procedure subtype keywords were used. This technique is no longer recommended but is still supported.

The use of the marquee menu as in input device is demonstrated in the **Marquee Demo** and **Delete Points from Wave** example experiments.

Polygon as Input Device

This technique is similar to the marquee technique except that you can identify a nonrectangular area. It is implemented using **FindPointsInPoly** operation (see page V-174).

Chapter
IV-7

Programming Techniques

Overview

This chapter discusses programming techniques and issues that go beyond the basics of the Igor programming language and which all Igor programmers should understand. Techniques for advanced programmers are discussed in Chapter IV-10, **Advanced Programming**.

The Include Statement

The include statement is a "compiler directive" that you can put in your procedure file to automatically open another procedure file. A typical include statement looks like this:

```
#include <Split Axis>
```

This statement automatically opens the file "Split Axis.ipf" in the WaveMetrics Procedures folder. Once this statement has been compiled, you can call routines from that file. This is the recommended way to access procedure files that contain utility routines.

The # character at the beginning specifies that a compiler directive follows. Compiler directives must start at the far left edge of the procedure window with no leading tabs or spaces. Include statements can appear anywhere in the file but it is conventional to put them near the top.

It is possible to include a file which in turn includes other files that will automatically open.

Included files are opened when procedures are compiled. If you remove an include statement from a procedure file, the file will automatically close when you next compile.

There are four forms of the include statement specifying where files are located:

1. Igor searches for the named file in "Igor Pro Folder/WaveMetrics Procedures" and in any subfolders:

   `#include <`*`fileName`*`>`

   Example: `#include <Split Axis>`
2. Igor searches for the named file in "Igor Pro Folder/User Procedures" and "Igor Pro User Files/User Procedures" and in any subfolders:

   `#include "`*`fileName`*`"`

   Example: `#include "Test Utility Procs"`
3. Igor looks for the file only in the exact location specified:

   `#include "`*`full file path`*`"`

   Example: `#include "Hard Disk:Desktop Folder:Test Utility Procs"`
4. Igor looks for the file relative to the Igor Pro folder or the Igor Pro User Files folder or the folder containing the procedure file that contains the #include statement:

   `#include ":`*`partial file path`*`"`

   Example: `#include ":Spectroscopy Procedures:Voigt Procs"`

   Igor looks first relative to the Igor Pro Folder. If that fails, it looks relative to the **Igor Pro User Files** folder (requires Igor Pro 6.20 or later). If that fails it looks relative to the procedure file containing the #include statement (requires Igor Pro 6.00 or later) or, if the #include statement is in the built-in procedure window, relative to the experiment file (requires Igor Pro 6.02 or later).

   The name of the file being included must end with the standard ".ipf" extension but the extension is not used in the include statement.

Procedure File Version Information

If you create a procedure file to be used by other Igor users, it's a good idea to add version information to the file. You do this by putting a #pragma version statement in the procedure file. For example:

```
#pragma version = 1.10
```

This statement must appear with no indentation and must appear in the first 50 lines of the file. If Igor does not find the #pragma version statement and if Igor is running on Macintosh, it then looks for a `'vers',1`

resource. If Igor finds neither the #pragma statement nor the resource, it treats the file as version 1.00. 'vers' resources are obsolete but are still recognized by Igor.

Igor looks for the version information when the user invokes the File Information dialog from the Procedure menu. If the file has version information, Igor displays the version next to the file name in the dialog.

Igor also looks for version information when it opens an included file. An include statement can require a certain version of the included file using the following syntax:

```
#include <Bivariate Histogram> version>=1.03
```

If the required version of the procedure file is not available, Igor displays a warning to inform the user that the procedure file needs to be updated.

Turning the Included File's Menus Off

Normally an included procedure file's menus and menu items are displayed. However, if you are including a file merely to call one of its procedures, you may not want that file's menu items to appear. To suppress the file's menus and menu items, use:

```
#include <Decimation> menus=0
```

To use both the menus and version options, you must separate them with a comma:

```
#include <Decimation> menus=0, version>=1.1
```

Optionally Including Files

Compilation usually ceases if the included file isn't found. On occasion it is advantageous to allow compilation to proceed if an included file isn't present or is the wrong version. Optionally including a procedure file is appropriate only if the file truly isn't needed to make procedures compile or operate.

Use the "optional" keyword to optionally include a procedure file:

```
#include <Decimation> version>=1.1, optional
```

The "optional" keyword requires Igor 6.13 or later.

Writing General-Purpose Procedures

Procedures can be placed on a scale ranging from general to specific. Usually, the high-level procedures of a program are specific to the task at hand and call on more general, lower-level procedures. The most general procedures are often called "utility" procedures.

You can achieve a high degree of productivity by building a library of utility procedures that you reuse in different programs. In Igor, you can think of the routines in an experiment's built-in Procedure window as a program. It can call utility routines which should be stored in a separate procedure file so that they are available to multiple experiments.

The files stored in the WaveMetrics Procedures folder contain general-purpose procedures on which your high-level procedures can build. Use the include statement (see **The Include Statement** on page IV-145) to access these and other utility files.

When you write utility routines, you should keep them as general as possible so that they can be reused as much as possible. Here are some guidelines.

- Avoid the use of global variables as inputs or outputs. Using globals hard-wires the routine to specific names which makes it difficult to use and limits its generality.
- If you use globals to store state information between invocations of a routine, package the globals in data folders.

  WaveMetrics packages usually create data folders inside "root:Packages". We use the prefix "WM" for data folder names to avoid conflict with other packages. Follow this lead and should pick your own data folder names so as to minimize the chance of conflict with WaveMetrics packages or with others.

- Choose a clear and specific name for your utility routine.

  By choosing a name that says precisely what your utility routine does, you minimize the likelihood of collision with the name of another procedure. You also increase the readability of your program.
- Make functions which are used only internally by your procedures static.

  By making internal functions static (i.e., private), you minimize the likelihood of collision with the name of another procedure.

Programming with Liberal Names

Standard names in Igor may contain letters, numbers and the underscore character only. Starting with Igor Pro 3.0, it became possible to use wave names and data folder names that contain almost any character (see **Liberal Object Names** on page III-415). However, if you do use liberal names, some existing Igor procedures and extensions may break. Programmers will need to make changes to their procedures to ensure they will work with liberal names.

Whenever a liberal name is used in a command or expression, the name must be enclosed in single quotes. For example:

```
Make 'Wave 1', wave2, 'miles/hour'
'Wave 1' = wave2 + 'miles/hour'
```

Without the single quotes, Igor has no way to know where a particular name ends. This is a problem whenever Igor parses a command or statement. Igor parses commands at the following times:

- When it compiles a user-defined function.
- When it compiles the right-hand side of an assignment statement, including a formula (:= dependency expression).
- When it interprets a macro.
- When it interprets a command that you enter in the command line or via an Igor Text file.
- When it interprets a command you submit for execution via the Execute operation.
- When it interprets a command that an XOP submits for execution via the XOPCommand or XOPSilentCommand callback routines.

When you use an Igor dialog to generate a command, Igor automatically uses quotes where necessary.

Programmers need to be concerned about liberal names whenever they create a command and then execute it (via an Igor Text file, the Execute operation or the XOPCommand and XOPSilentCommand callback routines) or when creating a formula. In short, when you create something that Igor has to parse, names must be quoted if they are liberal. Names that are not liberal can be quoted or unquoted.

If you have a procedure that builds up a command in a string variable and then executes it via the Execute operation, you must use the PossiblyQuoteName function to provide the quotes if needed.

Here is a trivial example showing the old way of doing this along with the new, liberal-name aware way.

```
Function AddOneToWave(w)
   WAVE w

   String cmd
   String name = NameOfWave(w)

   // Here is the old way
   sprintf cmd, "%s += 1", name
   Execute cmd

   // Here is the new, liberal-name aware way
   sprintf cmd, "%s += 1", PossiblyQuoteName(name)
   Execute cmd
End
```

Imagine that you pass a wave named *wave 1* to this function. Using the old method, the Execute operation would see the following text:

```
wave 1 += 1
```

Igor will generate an error because it will try to find an operation, macro, function, wave or variable named *wave*.

Using the new method, the Execute operation would see the following text:

```
'wave 1' += 1
```

This works correctly because Igor sees 'wave 1' as a single name.

Depending on exactly what they do with their parameters, Igor procedures and extensions written before Igor Pro 3.0 or which have not been tested with liberal names may cause errors like this to occur. The safe route is to test all of the procedures and extensions that you rely on before routinely using liberal names.

When you pass a string expression containing a simple name to the $ operator, the name must not be quoted because Igor does no parsing:

```
String w = "wave 1"; Display $w                         // Right
String w = "'wave 1'"; Display $w                       // Wrong
```

However, when you pass a *path* in a string to $, any liberal names in the path must be quoted:

```
String path = "root:'My Data':'wave 1'"; Display $w   // Right
String path = "root:My Data:wave 1"; Display $w       // Wrong
```

For further explanation of liberal name quoting rules, see **Accessing Global Variables and Waves Using Liberal Names** on page IV-53.

Some built-in string functions return a list of waves. WaveList is the most common example. If liberal wave names are used, you will have to be extra careful when using the list of names. For example, you can't just append the list to the name of an operation that takes a list of names and then execute the resulting string. Rather, you will have to extract each name in turn, possibly quote it and then append it to the operation.

For example, the following will work if all wave names are standard but not if any wave names are liberal:

```
Execute "Display " + WaveList("*", ",", "")
```

Now you will need a string function that extracts out each name, possibly quotes it, and creates a new string using the new names separated by commas. The PossiblyQuoteList function in the WaveMetrics procedure file Strings as Lists does this. The preceding command becomes:

```
Execute "Display " + PossiblyQuoteList(WaveList("*", ",", ""), ",")
```

To include the Strings as Lists file in your procedures, see **The Include Statement** on page IV-145.

For details on using liberal names in user-defined functions, see **Accessing Global Variables and Waves Using Liberal Names** on page IV-53.

Programming with Data Folders

For general information on data folders, including syntax for using data folders in command line operations, see Chapter II-8, **Data Folders**.

Data folders provide a powerful way for intermediate and advanced programmers to organize their data and to reduce clutter. However, using data folders introduces some complexity in Igor programming. The name of a variable or wave is no longer sufficient to uniquely identify it because the name alone does not indicate in which data folder it resides.

There are two main uses for data folders:

- To store globals used by procedures to keep track of their state.
- To store runs of data with identical structures.

Storing Procedure Globals

If you create packages of procedures, for example data acquisition, data analysis, or a specialized plot type, you typically need to store global variables, strings and waves to maintain the state of your package. You should keep all such items in a data folder that you create with a unique name to reduce clutter and avoid conflicts with other packages.

If your package is inherently global in nature, data acquisition for example, create your data folder within a data folder named Packages that you create (if necessary) in the root data folder. For example:

```
Function MyDAQInit()
   String savDF= GetDataFolder(1)   // Save current DF for restore.

   NewDataFolder/O/S root:Packages     // Make sure this exists.

   if( DataFolderExists("WMDataAcq") ) // Already created WMDataAcq?
      SetDataFolder WMDataAcq          // Our stuff is in here.
      < do reinitialization if necessary >
   else
      NewDataFolder/S WMDataAcq        // Our stuff goes in here.
      Variable/G avar                  // Create at runtime.
      < do other initialization if necessary >
   endif

   SetDataFolder savDF                 // Restore current DF.
End
```

On the other hand, if your package needs to create a group of variables and waves as a result of an operation on a single input wave, it makes sense to create your data folder in the one that holds the input wave (see the **GetWavesDataFolder** function on page V-224). Similarly, if you create a package that takes an entire data folder as input (via a string parameter containing the path to the data folder), does operations on what it finds inside and then needs to create a folder of output, it should create the output data folder in the one, or possibly at the same level as the one containing the input.

If your package creates and maintains windows, it should create a master data folder in root:Packages and then create individual folders within the master that correspond to each instance of a window. For example, a Smith Chart package would create a master data folder as root:Packages:SmithCharts: and then create individual folders inside the master with names that correspond to the individual graphs.

A package designed to operate on traces within graphs would go one step further and create a data folder for each trace that it has worked on, inside the data folder for the graph. This technique is illustrated by the Smooth Control Panel procedure file. For a demonstration, see Examples:Feature Demos:Smoothing Control Panel.pxp.

Storing Runs of Data

If you acquire data using a well-established experimental protocol, your data will have a well-defined structure. Each time you run your protocol, you produce a new data set with the same structure. Storing each data set in its own data folder avoids name conflicts between corresponding items of different data sets. It also simplifies the writing of procedures to analyze and compare data set.

To use a trivial example, your data might have a structure like this:

```
<Run of data>
   String: conditions
   Variable: temperature
   Wave: appliedVoltage
   Wave: luminosity
```

Having defined this structure, you could then write procedures to:

1. Load your data into a data folder
2. Create a graph showing the data from one run
3. Create a graph comparing the data from many runs

Writing these procedures is greatly simplified by the fact that the names of the items in a run are fixed. For example, step 2 could be written as:

```
Function GraphOneRun()  // Graphs data run in the current data folder.
   SVAR conditions
   NVAR temperature
   WAVE appliedVoltage, luminosity

   Display luminosity vs appliedVoltage

   String text
   sprintf text, "Temperature: %g.\rExperimental conditions: %s.",
         temperature, conditions
   Textbox text
End
```

To create a graph, you would first set the current data folder to point to a run of data and then you would invoke the GraphOneRun function.

The Data Folder Tutorial experiment, in the Learning Aids:Tutorials folder, shows in detail how to accomplish the three steps listed above.

Setting and Restoring the Current Data Folder

There are many reasons why you might want to save and restore the current data folder. In this example, we have a function that does a curve fit to a wave passed in as a parameter. The CurveFit operation creates two waves, W_coef and W_sigma, in the current data folder. If you use the /D option, it also creates a destination wave in the current data folder. In this function, we make sure that the W_coef, W_sigma and destination waves are all created in the same data folder as the source wave.

```
Function DoLineFit(w)
   WAVE w

   String dfSav = GetDataFolder(1)
   SetDataFolder GetWavesDataFolder(w,1)
   CurveFit line w /D
   SetDataFolder dfSav
End
```

Many other operations create output waves in the current data folder. Depending on what your goal is, you may want to use the technique shown here to control where the output waves are created.

A function should always save and restore the current data folder unless it is designed explicitly to change the current data folder.

Determining a Function's Target Data Folder

There are three common methods for determining the data folder that a function works on or in:

1. Passing a wave in the target data folder as a parameter
2. Having the function work on the current data folder
3. Passing a string parameter that points to the target data folder

For functions that operate on a specific wave, method 1 is appropriate.

For functions that operation on a large number of variables within a single data folder, methods 2 or 3 are appropriate. In method 2, the calling routine sets the data folder of interest as the current data folder. In method 3, the called function does this, and restores the original current data folder before it returns.

Clearing a Data Folder

There are times when you might want to clear a data folder before running a procedure, to remove things left over from a preceding run. If the data folder contains no child data folders, you can achieve this with:

```
KillWaves/A/Z; KillVariables/A/Z
```

If the data folder does contain child data folders, you could use the KillDataFolder operation. This operation kills a data folder and its contents, including any child data folders. You could kill the main data folder and then recreate it. A problem with this is that, if the data folder or its children contain a wave that is in use, you will generate an error which will cause your function to abort.

Here is a handy function that kills the contents of a data folder and the contents of its children without killing any data folders and without attempting to kill any waves that may be in use.

```
Function ZapDataInFolderTree(path)
   String path

   String savDF= GetDataFolder(1)
   SetDataFolder path

   KillWaves/A/Z
   KillVariables/A/Z
   KillStrings/A/Z

   Variable i
   Variable numDataFolders = CountObjects(":", 4)
   for(i=0; i<numDataFolders; i+=1)
      String nextPath = GetIndexedObjName(":", 4, i)
      ZapDataInFolderTree(nextPath)
   endfor

   SetDataFolder savDF
End
```

Using Strings

This section explains some common ways in which Igor procedures use strings. The most common techniques use built-in functions such as StringFromList and FindListItem. In addition to the built-in functions, there are a number of handy Igor procedure files in the WaveMetrics Procedures:Utilities:String Utilities folder.

Using Strings as Lists

Procedures often need to deal with lists of items. Such lists are usually represented as semicolon-separated text strings. The StringFromList function is used to extract each item, often in a loop. For example:

```
Function Test()
   Make jack,sam,fred,sue
   String list = WaveList("*",";","")
   Print list

   Variable numItems=ItemsInList(list), i
   for(i=0; i<numItems; i+=1)
      Print StringFromList(i,list)     // Print ith item
   endfor
End
```

Using Keyword-Value Packed Strings

A collection of disparate values is often stored in a list of keyword-value pairs. For example, the TraceInfo and AxisInfo functions return such a list. The information in these strings follows this form

```
KEYWORD:value;KEYWORD:value; ...
```

The **keyword** is always upper case and followed by a colon. Then comes the **value** and a semicolon. When parsing such a string, you should avoid reliance on the specific ordering of keywords — the order is likely to change in future releases of Igor.

Two functions will extract information from keyword-value strings. NumberByKey is used when the data is numeric, and StringByKey is used when the information is in the form of text.

Using Strings with Extractable Commands

The **AxisInfo** and **TraceInfo** readback functions provide much of their information in a form that resembles the commands that you use to make the settings in the first place. For example, to set an axis to log mode, you would execute something like this:

```
ModifyGraph log(left)=1
```

If we now look at the characters of the string returned by AxisInfo we see:

```
AXTYPE:left;CWAVE:jack; ... RECREATION:grid(x)=0;log(x)=1 ...
```

To use this information, we need to extract the value following "log(x)=" and then use it in a ModifyGraph command with the extracted value with the desired graph as the target. Here is a user function that sets the log mode of the bottom axis to the same value as the left axis:

```
Function CopyLogMode()
   String info,text
   Variable pos
   info=AxisInfo("","left")
   pos= StrSearch(info,"RECREATION:",0) // skip to start of "RECREATION:"
   pos += strlen("RECREATION:")         // skip "RECREATION:", too
                                        // get text after "log(x)="
   text= StringByKey("log(x)",info[pos,inf],"=",";")

   String cmd = "ModifyGraph log(bottom)=" + text
   Execute cmd
End
```

Regular Expressions

A regular expression is a pattern that is matched against a subject string from left to right. Regular expressions are used to identify lines of text containing a particular pattern and to extracts substrings matching a particular pattern.

A regular expression can contain regular characters that match the same character in the subject and special characters, called "metacharacters", that match any character, a list of specific characters, or otherwise identify patterns.

The regular expression syntax is based on PCRE — the Perl-Compatible Regular Expression Library.

Igor syntax is similar to regular expressions supported by various UNIX and POSIX `egrep(1)` commands. Igor's implementation does not support the Unicode (UTF-8) section of PCRE.

See **Regular Expressions References** on page IV-170 for details on PCRE.

Regular Expression Operations and Functions

Here are the Igor operations and functions that work with regular expressions:

Grep

The **Grep** operation identifies lines of text that match a pattern.

The subject is each line of a file or each row of a text wave or each line of the text in the clipboard.

Output is stored in a file or in a text wave or in the clipboard.

Grep Example

```
Function DemoGrep()
   Make/T source={"Monday","Tuesday","Wednesday","Thursday","Friday"}
   Make/T/N=0 dest
   Grep/E="sday" source as dest           // Find rows containing "sday"
   Print dest
End
```

The output from Print is:

```
   dest[0]= {"Tuesday","Wednesday","Thursday"}
```

GrepList

The **GrepList** function identifies items that match a pattern in a string containing a delimited list.

The subject is each item in the input list.

The output is a delimited list returned as the function result.

GrepList Example

```
Function DemoGrepList()
   String source = "Monday;Tuesday;Wednesday;Thursday;Friday"
   String dest = GrepList(source, "sday") // Find items containing "sday"
   Print dest
End
```

The output from Print is:

```
   Tuesday;Wednesday;Thursday;
```

GrepString

The **GrepString** function determines if a particular pattern exists in the input string.

The subject is the input string.

The output is 1 if the input string contains the pattern or 0 if not.

GrepString Example

```
Function DemoGrepString()
   String subject = "123.45"
   String regExp = "[0-9]+"            // Match one or more digits
   Print GrepString(subject,regExp)    // True if subject contains digit(s)
End
```

The output from Print is: 1

SplitString

The **SplitString** operation identifies subpatterns in the input string.

The subject is the input string.

Output is stored in one or more string output variables.

SplitString Example

```
Function DemoSplitString()
   String subject = "Thursday, May 7, 2009"
   String regExp = "([[:alpha:]]+), ([[:alpha:]]+) ([[:digit:]]+), ([[:digit:]]+)"
   String dayOfWeek, month, dayOfMonth, year
   SplitString /E=(regExp) subject, dayOfWeek, month, dayOfMonth, year
   Print dayOfWeek, month, dayOfMonth, year
End
```

The output from Print is:

```
Thursday  May  7  2009
```

Basic Regular Expressions

Here is a **Grep** command that uses "Fred" as the regular expression. "Fred" contains no metacharacters so this command identifies lines of text containing the literal string "Fred". It examines each line from the input file, afile.txt. All lines containing the pattern "Fred" are written to the output file, "FredFile.txt":

```
Grep/P=myPath/E="Fred" "afile.txt" as "FredFile.txt"
```

Character matching is case-sensitive by default. Prepending the Perl 5 modifier (?i) gives a case-insensitive pattern that matches upper and lower-case versions of “Fred”:

```
// Copy lines that contain "Fred", "fred", "FRED", "fREd", etc
Grep/P=myPath/E="(?i)fred" "afile.txt" as "AnyFredFile.txt"
```

To copy lines that do not match the regular expression, use the Grep /E flag with the optional *reverse* parameter set to 1 to reverse the sense of the match:

```
// Copy lines that do NOT contain "Fred", "fred", "fREd", etc
Grep/P=myPath/E={"(?i)fred",1} "afile.txt" as "NotFredFile.txt"
```

Note: Igor doesn't use the Perl opening and closing regular expression delimiter character which is the forward slash. In Perl you would use `"/Fred/"` and `"/(?i)fred/"`.

Regular Expression Metacharacters

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of “metacharacters”, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of metacharacters: those that are recognized anywhere in the pattern except within brackets, and those that are recognized in brackets. Outside brackets, the metacharacters are as follows:

\	General escape character with several uses
^	Match start of string
$	Match end of string
.	Match any character except newline (by default) (To match newline, see **Matching Newlines** on page IV-161)
[	Start character class definition (for matching one of a set of characters)
|	Start of alternative branch (for matching one or the other of two patterns)
(	Start subpattern
)	End subpattern
?	0 or 1 quantifier (for matching 0 or 1 occurrence of a pattern) Also extends the meaning of (Also quantifier minimizer
*	0 or more quantifier (for matching 0 or more occurrence of a pattern)
+	1 or more quantifier (for matching 1 or more occurrence of a pattern) Also possessive quantifier
{	Start min/max quantifier (for matching a number or range of occurrences)

Character Classes in Regular Expressions

A character class is a set of characters and is used to specify that one character of the set should be matched. Character classes are introduced by a left-bracket and terminated by a right-bracket. For example:

```
[abc]       Matches a or b or c
[A-Z]       Matches any character from A to Z
[A-Za-z]    Matches any character from A to Z or a to z.
```

POSIX character classes specify the characters to be matched symbolically. For example:

```
[[:alpha:]] Matches any alphabetic character (A to Z or a to z).
[[:digit:]] Matches 0 to 9.
```

In a character class the only metacharacters are:

\	General escape character
^	Negate the class, but only if ^ is the first character
-	Indicates character range
[	POSIX character class (only if followed by POSIX syntax)
]	Terminates the character class

Backslash in Regular Expressions

The backslash character has several uses. First, if it is followed by a nonalphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, the * character normally means "match zero or more of the preceding subpattern". If you want to match a * character, you write * in the pattern. This escaping action applies whether or not the following character would otherwise be interpreted as a metacharacter, so it is always safe to precede a nonalphanumeric with backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write \\.

Note: Because Igor also has special uses for backslash (see **Escape Characters in Strings** on page IV-13), you must double the number of backslashes you would normally use for a Perl or grep pattern. Each pair of backslashes sends one backslash to, say, the **Grep** command.

For example, to copy lines that contain a backslash followed by a *z* character, the Perl pattern would be "\\z", but the equivalent Igor Grep expression would be /E="\\\\z".

Igor's input string parser converts "\\" to "\" so, when you write /E="\\\\z", the regular expression engine sees /E="\\z".

This difference is important enough that the PCRE and Igor Patterns (using Grep /E syntax) are both shown below when they differ.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between \Q and \E. This is different from Perl in that $ and @ are handled as literals in \Q...\E sequences in PCRE, whereas in Perl, $ and @ cause variable interpolation. Note the following examples:

Igor Pattern	PCRE Pattern	PCRE Matches	Perl Matches
\\Qabc$xyz\\E	\Qabc$xyz\E	abc$xyz	abc followed by the contents of $xyz
\\Qabc\\$xyz\\E	\Qabc\$xyz\E	abc\$xyz	abc\$xyz
\\Qabc\\E\\$\\Qxyz\\E	\Qabc\E\$\Qxyz\E	abc$xyz	abc$xyz

The \Q...\E sequence is recognized both inside and outside character classes.

Backslash and Nonprinting Characters

A second use of backslash provides a way of encoding nonprinting characters in patterns in a visible manner. There is no restriction on where nonprinting characters can occur, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

Igor Pattern	PCRE Pattern	Character Matched
\\a	\a	Alarm, that is, the BEL character (hex 07)
\\cx	\cx	“Control-x”, where x is any character
\\e	\e	Escape (hex 1B)
\\f	\f	Formfeed (hex 0C)
\\n	\n	Newline (hex 0A)
\\r	\r	Carriage return (hex 0D)
\\t	\t	Tab (hex 09)
\\ddd	\ddd	Character with octal code ddd, or backreference
\\xhh	\xhh	Character with hex code hh

Backslash and Nonprinting Characters Arcania

The material in this section is arcane and rarely needed. We recommend that you skip it.

The precise effect of \cx is if x is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus \cz becomes hex 1A, but \c{ becomes hex 3B, while \c; becomes hex 7B.

After \x, from zero to two hexadecimal digits are read (letters can be in upper or lower case). If characters other than hexadecimal digits appear between \x{ and }, or if there is no terminating }, this form of escape is not recognized. Instead, the initial \x will be interpreted as a basic hexadecimal escape, with no following digits, giving a character whose value is zero.

After \0 up to two further octal digits are read. In both cases, if there are fewer than two digits, just those that are present are used. Thus the sequence \0\x\07 specifies two binary zeros followed by a BEL character (code value 7). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a back reference. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE rereads up to three octal digits following the backslash, and generates a single byte from the least significant 8 bits of the value. Any subsequent digits stand for themselves. For example:

Igor Pattern	PCRE Pattern	Character(s) Matched
\\040	\040	Another way of writing a space
\\40	\40	A space, provided there are fewer than 40 previous capturing subpatterns
\\7	\7	Always a back reference
\\11	\11	Might be a back reference, or another way of writing a tab
\\011	\011	Always a tab
\\0113	\0113	A tab followed by the character “3”
\\113	\113	Might be a back reference, otherwise the character with octal code 113
\\377	\377	Might be a back reference, otherwise the byte consisting entirely of 1 bits
\\81	\81	Either a back reference or a binary zero followed by the two characters “8” and “1”

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single byte value can be used both inside and outside character classes. In addition, inside a character class, the sequence \b is interpreted as the backspace character (hex 08), and the sequence \X is interpreted as the character X. Outside a character class, these sequences have different meanings (see **Backslash and Nonprinting Characters** on page IV-156).

Backslash and Generic Character Types

The third use of backslash is for specifying generic character types. The following are always recognized:

Igor Pattern	PCRE Pattern	Character(s) Matched
\\d	\d	Any decimal digit
\\D	\D	Any character that is not a decimal digit
\\s	\s	Any whitespace character
\\S	\S	Any character that is not a whitespace character
\\w	\w	Any “word” character
\\S	\W	Any “nonword” character

Each pair of escape sequences, such as \d and \D, partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

For compatibility with Perl, \s does not match the vertical tab character (VT-ASCII code 11). This makes it different from the POSIX "space" class. The \s whitespace characters are horizontal tab (HT-9), linefeed (LF-10), formfeed (FF-12), carriage-return (CR-13), and space (32).

A “word” character is an underscore or any character that is a letter or digit.

Backslash and Simple Assertions

The fourth use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described in **Assertions** on page IV-166. The backslashed assertions are:

Igor Pattern	PCRE Pattern	Character(s) Matched
\\b	\b	At a word boundary
\\B	\B	Not at a word boundary
\\A	\A	At start of subject
\\Z	\Z	At end of subject or before newline at end
\\z	\z	At end of subject
\\G	\G	At first matching position in subject

These assertions may not appear in character classes (but note that \b has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match \w or \W (i.e. one matches \w and the other matches \W), or the start or end of the string if the first or last character matches \w, respectively.

While PCRE defines additional simple assertions (\A, \Z, \z and \G), they are not any more useful to Igor’s regular expression commands than the ^ and $ characters.

Circumflex and Dollar

Outside a character class, in the default matching mode, the circumflex character ^ is an assertion that is true only if the current matching point is at the start of the subject string. Inside a character class, circumflex has an entirely different meaning (see **Character Classes and Brackets** on page IV-159).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an “anchored” pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character $ is an assertion that is true only if the current matching point is at the end of the subject string, or immediately before a newline character that is the last character in the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

Dot, Period, or Full Stop

Outside a character class, a dot in the pattern matches any one character in the subject, including a nonprinting character, but not (by default) newline. Dot has no special meaning in a character class.

The match option setting (?s) changes the default behavior of dot so that it matches any character including newline. The setting can appear anywhere before the dot in the pattern. See **Matching Newlines** on page IV-161 for details.

Character Classes and Brackets

An opening bracket introduces a character class which is terminated by a closing bracket. A closing bracket on its own is not special. If a closing bracket is required as a member of the class, it must be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class `[aeiou]` matches any English lower case vowel, whereas `[^aeiou]` matches any character that is not an English lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters that are in the class by enumerating those that are not.

When caseless matching is set, using the `(?i)` match option setting, any letters in a class represent both their upper case and lower case versions, so for example, the caseless pattern `(?i)[aeiou]` matches *A* as well as *a*, and the caseless pattern `(?i)[^aeiou]` does not match *A*.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, `[d-m]` matches any letter between *d* and *m*, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

To include a right-bracket in a range you must use `\]`. As usual, this would be represented in a literal Igor string as `\\]`.

Though it is rarely needed, you can specify a range using octal numbers, for example `[\000-\037]`.

The character types `\d`, `\D`, `\p`, `\P`, `\s`, `\S`, `\w`, and `\W` may also appear in a character class, and add the characters that they match to the class. For example, `[\dABCDEF]` matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class `[^\W_]` matches any letter or digit, but not underscore. The corresponding **Grep** command would begin with

```
Grep/E="[^\\W_]"…
```

The only metacharacters that are recognized in character classes are backslash, hyphen (only where it can be interpreted as specifying a range), circumflex (only at the start), opening bracket (only when it can be interpreted as introducing a POSIX class name — see **POSIX Character Classes** on page IV-159), and the terminating closing bracket. However, escaping other nonalphanumeric characters does no harm.

POSIX Character Classes

Perl supports the POSIX notation for character classes. This uses names enclosed by `[:` and `:]` within the enclosing brackets. PCRE also supports this notation. For example,

```
[01[:alpha:]%]
```

matches “0”, “1”, any alphabetic character, or “%”.

The supported class names, all of which must appear between `[:` and `:]` inside a character class specification, are

`alnum`	Letters and digits
`alpha`	Letters
`ascii`	Character codes 0 - 127

blank	Space or tab only
cntrl	Control characters
digit	Decimal digits (same as \d)
graph	Printing characters, excluding space
lower	Lower case letters
print	Printing characters, including space
punct	Printing characters, excluding letters and digits
space	White space (not quite the same as \s)
upper	Upper case letters
word	"Word" characters (same as \w)
xdigit	Hexadecimal digits

The "space" characters are horizontal tab (HT-9), linefeed (LF-10), vertical tab (VT-11), formfeed (FF-12), carriage-return (CR-13), and space (32). Notice that this list includes the VT character (code 11). This makes "space" different from \s, which does not include VT (for Perl compatibility).

The class name word is a Perl extension, and blank is a GNU extension from Perl 5.8. Another Perl extension is negation that is indicated by a ^ character after the colon. For example,

```
[12[:^digit:]]
```

matches "1", "2", or any nondigit. PCRE (and Perl) also recognize the POSIX syntax [.ch.] and [=ch=] where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

Alternation

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternative patterns may be specified, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined in **Subpatterns** on page IV-162), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

Match Option Settings

Character matching options can be changed from within the pattern by a sequence of Perl option letters enclosed between (? and). The option letters are:

Option	PCRE Name	Characters Matched
i	PCRE_CASELESS	Upper and lower case.
m	PCRE_MULTILINE	^ matches start of string and just after a new line (\n). $ matches end of string and just before a new line. Without (?m), ^ and $ match only the start and end of the entire string.
s	PCRE_DOTALL	. matches all characters including newline. Without (?s), . does not match newlines. See **Matching Newlines** on page IV-161.
U	PCRE_UNGREEDY	Reverses the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by ?.

For example, (?i) sets caseless matching.

You can unset these options by preceding the letter with a hyphen: (?-i) turns off caseless matching.

You can combine a setting and unsetting such as `(?i-U)`, which sets PCRE_CASELESS while unsetting PCRE_UNGREEDY.

When an option change occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. If the change is placed right at the start of a pattern, PCRE extracts it into the global options.

An option change within a subpattern affects only that part of the current pattern that follows it, so

```
(a(?i)b)c
```

matches abc and aBc and no other strings.

Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

```
(a(?i)b|c)
```

matches “ab”, “aB”, “c”, and “C”, even though when matching “C” the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time.

The other PCRE 5-specific options, such as (?e), (?A), (?D), (?S), (?x) and (?X), are implemented but are not very useful with Igor's regular expression commands.

Matching Newlines

Igor generally uses carriage-return (ASCII code 13) return to indicate a new line in text. This is represented in a literal string by "\r". For example:

```
Print "Hello\rGoodbye"
```

prints two lines of text to the history area.

However, in regular expressions, the newline character is linefeed (ASCII code 10). This is represented in a literal string by "\n". Carriage-return has no special status in regular expressions.

The match option setting `(?s)` changes the default behavior of dot so that it matches any character including newline. The setting can appear anywhere before the dot in the pattern. For example:

```
Function DemoNewline(includeNewline)
   Variable includeNewline

   String subject = "Hello\nGoodbye"   // \n represents newline

   String regExp
   if (includeNewline)
      regExp = "(?s)(.+)"     // One or more of any character
   else
      regExp = "(.+)"         // One or more of any character except newline
   endif

   String result
   SplitString /E=(regExp) subject, result
   Print result
End
```

The output from DemoNewline(0) is:

```
Hello
```

The output from DemoNewline(1) is:

```
Hello\nGoodbye
```

Here is a more realistic example:

```
Function DemoDotAll()
   String theString = ExampleHTMLString()

   // This regular expression attempts to extract items from
   // HTML code for an ordered list. It fails because the HTML
   // code contains newlines and, by default, . does not match newlines.
   String regExpFails="<li>(.*?)</li>.*<li>(.*?)</li>"
   String str1, str2
   SplitString/E=regExpFails theString, str1, str2
   Print V_flag, " subpatterns were matched using regExpFails"
   Printf "\"%s\", \"%s\"\r", str1, str2

   // This regular expression works because the "(?s)" match
   // option setting causes . to match newlines.
   String regExpWorks="(?s)<li>(.*?)</li>.*<li>(.*?)</li>"
   SplitString/E=regExpWorks theString, str1, str2
   Print V_flag, " subpatterns were matched using regExpWorks"
   Printf "\"%s\", \"%s\"\r", str1, str2
End
Function/S ExampleHTMLString()   // Returns HTML code containing newlines
   String theString = ""
   theString += "<ol>\n"
   theString += "\t<li>item 1</li>\n"
   theString += "\t<li>item 2</li>\n"
   theString += "<\ol>\n"
   return theString
End

•DemoDotAll()
  0   subpatterns were matched using regExpFails
  "", ""
  2 subpatterns were matched using regExpWorks
  "item 1", "item 2"
```

Subpatterns

Subpatterns are used to group alternatives, to match a previously-matched pattern again, and to extract match text using **SplitString**.

Subpatterns are delimited by parentheses, which can be nested. Turning part of a pattern into a subpattern localizes a set of alternatives. For example, this pattern (which includes two vertical bars signifying alternation):

```
cat(aract|erpillar|)
```

matches one of the words "cat", "cataract", or "caterpillar". Without the parentheses, it would match "cataract", "erpillar" or the empty string.

Named Subpatterns

Specifying subpatterns by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with this difficulty, PCRE supports the naming of subpatterns, something that Perl does not provide. The Python syntax (`?P<`*name*`>`...) is used. For example:

```
My (?P<catdog>cat|dog) is cooler than your (?P=catdog)
```

Here catdog is the name of the first and only subpattern. `?P<catdog>` names the subpattern and `(?P=catdog)` matches the previous match for that subpattern.

Names consist of alphanumeric characters and underscores, and must be unique within a pattern.

Named capturing parentheses are still allocated numbers as well as names. This has the same effect as the previous example:

```
My (?P<catdog>cat|dog) is cooler than your \\1
```

Repetition

Repetition is specified by quantifiers:

?	0 or 1 quantifier
	Example: `[abc]?` - Matches 0 or 1 occurrences of a or b or c
*	0 or more quantifier
	Example: `[abc]*` - Matches 0 or more occurrences of a or b or c
+	1 or more quantifier
	Example: `[abc]+` - Matches 1 or more occurrences of a or b or c
{n}	n times quantifier
	Example: `[abc]{3}` - Matches 3 occurrences of a or b or c
{n,m}	n to m times quantifier
	Example: `[abc]{3,5}` - Matches 3 to 5 occurrences of a or b or c

Quantifiers can follow any of the following items:

- A literal data character
- The . metacharacter
- The \C escape sequence
- An escape such as \d that matches a single character
- A character class
- A back reference (see **Back References** on page IV-166)
- A parenthesized subpattern (unless it is an assertion)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in braces, separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

```
z{2,4}
```

matches “zz”, “zzz”, or “zzzz”.

If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
[aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, while

```
\d{8}
```

matches exactly 8 digits. A left brace that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, `{,6}` is not a quantifier, but a literal string of four characters.

The quantifier `{0}` is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience (and historical compatibility) the three most common quantifiers have single-character abbreviations:

`*`	Equivalent to `{0,}`
`+`	Equivalent to `{1,}`
`?`	Equivalent to `{0,1}`

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

Quantifier Greediness

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between `/*` and `*/` and within the comment, individual `*` and `/` characters may appear. An attempt to match C comments by applying the pattern

```
/\*.*\*/        or    Grep/E="/\\*.*\\*/"
```

to the string

```
/* first comment */  not comment  /* second comment */
```

fails because it matches the entire string owing to the greediness of the `.*` item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*.*?\*/       or    Grep/E="/\\*.*?\\*/"
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches.

Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d          or    Grep/E="\\d??\\d"
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the PCRE_UNGREEDY option `(?U)` is set, the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behavior.

Quantifiers With Subpatterns

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more memory is required for the compiled pattern, in proportion to the size of the minimum or maximum.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
(tweedle[dume]{3}\s*)+  or    Grep/E="(tweedle[dume]{3}\\s*)+"
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
/(a|(b))+/
```

matches "aba" the value of the second captured substring is "b".

Atomic Grouping and Possessive Quantifiers

With both maximizing and minimizing repetition, failure of what follows normally reevaluates the repeated item to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern `\d+foo` when applied to the subject line

```
123456bar
```

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the `\d+` item, and then with 4, and so on, before ultimately failing. "Atomic grouping" provides the means for specifying that once a subpattern has matched, it is not to be reevaluated in this way.

If we use atomic grouping for the previous example, the matcher would give up immediately on failing to match "foo" the first time. The notation is a kind of special parenthesis, starting with `(?>` as in this example:

```
(?>\d+)foo      or    Grep/E="(?>\\d+)foo"
```

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both `\d+` and `\d+?` are prepared to adjust the number of digits they match in order to make the rest of the pattern match, `(?>\d+)` can only match an entire sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as in the example above, a simpler notation, called a "possessive quantifier" can be used. This consists of an additional `+` character following a quantifier. Using this notation, the previous example can be rewritten as

```
\d++foo         or    Grep/E="\\d++foo"
```

Possessive quantifiers are always greedy; the setting of the PCRE_UNGREEDY option is ignored. They are a convenient notation for the simpler forms of atomic group. However, there is no difference in the meaning or processing of a possessive quantifier and the equivalent atomic group.

The possessive quantifier syntax is an extension to the Perl syntax. It originates in Sun's Java package.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of an atomic group is the only way to avoid some failing matches taking a very long time indeed. The pattern

```
(\D+|<\d+>)*[!?]   or    Grep/E="(\\D+|<\\d+>)*[!?]"
```

matches an unlimited number of substrings that either consist of nondigits, or digits enclosed in `<>`, followed by either `!` or `?`. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the internal `\D+` repeat and the external `*` repeat in a large number of ways, and all have to be tried. (The example uses `[!?]` rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed so that it uses an atomic group, like this:

```
((?>\D+)|<\d+>)*[!?]    or    Grep/E="((?>\\D+)|<\\d+>)*[!?]"
```

sequences of nondigits cannot be broken, and failure happens quickly.

Back References

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. See **Backslash and Nonprinting Characters** on page IV-156 for further details of the handling of digits following a backslash.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see **Subpatterns as Subroutines** on page IV-170 for a way of doing that). So the pattern

```
(sens|respons)e and \1ibility or /E="(sens|respons)e and \\1ibility"
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example,

```
((?i)rah)\s+\1     or     Grep/E="((?i)rah)\\s+\\1"
```

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

Back references to named subpatterns use the Python syntax `(?P<name>)`. We could rewrite the above example as follows:

```
(?P<p1>(?i)rah)\s+(?P=p1)   or     Grep/E="(?P<p1>(?i)rah)\\s+(?P=p1)"
```

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

```
(a|(bc))\2
```

always fails if it starts to match "a" rather than "bc". Because there may be many capturing parentheses in a pattern, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. An empty comment (see **Regular Expression Comments** on page IV-169) can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, `(a\1)` never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

```
(a|b\1)+        or     Grep/E="(a|b\\1)+"
```

matches any number of *a*'s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

Assertions

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as `\b`, `\B`, `^` and `$` are described in **Backslash and Simple Assertions** on page IV-158.

More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it. An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed.

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Lookahead Assertions

Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example,

```
\w+(?=;)        or    Grep/E="\\w+(?=;)"
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

```
(?!foo)bar
```

does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion (?!foo) is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve the other effect.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with (?!) because an empty string always matches, so an assertion that requires there not to be an empty string must always fail.

Lookbehind Assertions

Lookbehind assertions start with (?<= for positive assertions and (?<! for negative assertions. For example,

```
(?<!foo)bar
```

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several alternatives, they do not all have to have the same fixed length. Thus

```
(?<=bullock|donkey)
```

is permitted, but

```
(?<!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl (at least for 5.8), which requires all branches to match the same length of string. An assertion such as

```
(?<=ab(c|de))
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

```
(?<=abc|abde)
```

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed width and then try to match. If there are insufficient characters before the current position, the match is deemed to fail.

Atomic groups can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

```
abcd$
```

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each *a* in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
^.*abcd$
```

the initial .* matches the entire string at first, but when this fails (because there is no following *a*), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for *a* covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
^(?>.*)(?<=abcd)
```

or, equivalently, using the possessive quantifier syntax,

```
^.*+(?<=abcd)
```

there can be no backtracking for the .* item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

Using Multiple Assertions

Several assertions (of any sort) may occur in succession. For example,

```
(?<=\d{3})(?<!999)foo    or     Grep/E="(?<=\\d{3})(?<!999)foo"
```

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does not match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abcfoo". A pattern to do that is

```
(?<=\d{3}...)(?<!999)foo or     Grep/E="(?<=\\d{3}...)(?<!999)foo"
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example,

```
(?<=(?<!foo)bar)baz
```

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

```
(?<=\d{3}(?!999)...)foo or Grep/E=" (?<=\\d{3}(?!999)...)foo"
```

is another pattern that matches "foo" preceded by three digits and any three characters that are not "999".

Conditional Subpatterns

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

```
(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are three kinds of condition. If the text between the parentheses consists of a sequence of digits, the condition is satisfied if the capturing subpattern of that number has previously matched. The number must be greater than zero. Consider the following pattern, which contains nonsignificant white space to make it more readable and to divide it into three parts for ease of discussion:

```
( \( )?    [^()]+    (?(1) \) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of nonparentheses, optionally enclosed in parentheses.

If the condition is the string (R), it is satisfied if a recursive call to the pattern or subpattern has been made. At "top level", the condition is false. This is a PCRE extension. Recursive patterns are described in the next section.

If the condition is not a sequence of digits or (R), it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing nonsignificant white space, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
\d{2}-[a-z]{3}-\d{2}  |  \d{2}-\d{2}-\d{2})
```

The condition is a positive lookahead assertion that matches an optional sequence of nonletters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

Regular Expression Comments

The sequence (?# marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

Recursive Patterns

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth. Perl provides a facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at runtime, and the code can refer to the expression itself. A Perl pattern to solve the parentheses problem can be created like this:

```
$re = qr{\( (?: (?>[^()]+) | (?p{$re}) )* \)}x;
```

The (?p{...}) item interpolates Perl code at runtime, and in this case refers recursively to the pattern in which it appears. Obviously, PCRE cannot support the interpolation of Perl code. Instead, it supports some special syntax for recursion of the entire pattern, and also for individual subpattern recursion.

The special item that consists of (? followed by a number greater than zero and a closing parenthesis is a recursive call of the subpattern of the given number, provided that it occurs inside that subpattern. (If not, it is a "subroutine" call, which is described in **Subpatterns as Subroutines** on page IV-170.) The special item (?R) is a recursive call of the entire regular expression.

For example, this PCRE pattern solves the nested parentheses problem (additional nonfunction whitespace has been added to separate the expression into parts):

```
\( ( (?>[^()]+) | (?R) )* \)
```

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of nonparentheses, or a recursive match of the pattern itself (that is a correctly parenthesized substring). Finally there is a closing parenthesis.

If this were part of a larger pattern, you would not want to recurse the entire pattern, so instead you could use this:

```
( \( ( (?>[^()]+) | (?1) )* \))
```

We have put the pattern into parentheses, and caused the recursion to refer to them instead of the whole pattern. In a larger pattern, keeping track of parenthesis numbers can be tricky. It may be more convenient to use named parentheses instead. For this, PCRE uses (?P>name), which is an extension to the Python syntax that PCRE uses for named parentheses (Perl does not provide named parentheses). We could rewrite the above example as follows:

```
(?P<pn> \( ( (?>[^()]+) | (?P>pn) )* \))
```

This particular example pattern contains nested unlimited repeats, and so the use of atomic grouping for matching strings of nonparentheses is important when applying the pattern to strings that do not match. For example, when this pattern is applied to

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

it yields "no match" quickly. However, if atomic grouping is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

At the end of a match, the values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If you want to obtain intermediate values, a callout function can be used (see **Subpatterns as Subroutines** on page IV-170). If the pattern above is matched against

```
(ab(cd)ef)
```

the value for the capturing parentheses is "ef", which is the last value taken on at the top level. If additional parentheses are added, giving

```
\(( ( (?>[^()]+) | (?R) )* ) \)
     ↑                      ↑
```

the string they capture is "ab(cd)ef", the contents of the top level parentheses. If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion, which it does by using pcre_malloc, freeing it via pcre_free afterward. If no memory can be obtained, the match fails with the PCRE_ERROR_NOMEMORY error.

Do not confuse the `(?R)` item with the condition `(R)`, which tests for recursion. Consider this pattern, which matches text in angle brackets, allowing for arbitrary nesting. Only digits are allowed in nested brackets (that is, when recursing), whereas any characters are permitted at the outer level.

```
< (?: (?(R) \d++  | [^<>]*+) | (?R)) * >
```

In this pattern, `(?(R)` is the start of a conditional subpattern, with two different alternatives for the recursive and nonrecursive cases. The `(?R)` item is the actual recursive call.

Subpatterns as Subroutines

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. An earlier example pointed out that the pattern

```
(sens|respons)e and \1ibility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern

```
(sens|respons)e and (?1)ibility
```

is used, it does match "sense and responsibility" as well as the other two strings. Such references must, however, follow the subpattern to which they refer.

Regular Expressions References

The regular expression syntax supported by **Grep**, **GrepString**, **GrepList**, and **SplitString** is based on the *PCRE — Perl-Compatible Regular Expression Library* by Philip Hazel, University of Cambridge, Cambridge, England. The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5.

Visit <http://pcre.org/> for more information about the PCRE library, and <http://www.perldoc.com/> for more about Perl regular expressions. This description, Regular Expressions In Igor, is taken from the PCRE documentation.

A good introductory book on regular expressions is: Forta, Ben, *Regular Expressions in 10 Minutes*, Sams Publishing, 2004.

A good comprehensive book on regular expressions is: Friedl, Jeffrey E. F., *Mastering Regular Expressions*, 2nd ed., 492 pp., O'Reilly Media, 2002.

Working with Files

Here are the built-in operations that you can use to read from or write to files:

Operation	What It Does
Open	Opens an existing file for reading or writing. Can also create a new file. Can also append to an existing file. Returns a file reference number that you must pass to the other file operations. Use Open/D to present a dialog that allows the user to choose a file without actually opening the file.
fprintf	Writes formatted text to an open file.
wfprintf	Writes wave data as formatted text to an open file.
FSetPos	Sets the position at which the next file read or write will be done.
FStatus	Given a file reference number, returns miscellaneous information about the file.
FBinRead	Reads binary data from a file into an Igor string, variable or wave. This is used mostly to read nonstandard file formats.
FBinWrite	Writes binary data from an Igor string, variable or wave. This is used mostly to write nonstandard file formats.
FReadLine	Reads a line of text from a text file into an Igor string variable. This can be used to parse an arbitrary format text file.
Close	Closes a file opened by the Open operation.

Before working with a file, you must use the Open operation to obtain a file reference number that you use with all the remaining commands. The Open operation creates new files, appends data to an existing file, or reads data from an existing file. There is no facility to deal with the resource fork of a Macintosh file.

Sometimes you may write a procedure that uses the Open operation and the Close operation but, because of an error, the Close operation never gets to execute. You then correct the procedure and want to rerun it. The file will still be open because the Close operation never got a chance to run. In this case, execute:

```
Close/A
```

from the command line to close all open files.

Finding Files

The TextFile and IndexedFile functions help you to determine what files exist in a particular folder. IndexedFile is just a more general version of TextFile.

You can also use the Open operation with the /D flag to present an open dialog.

Other File– and Folder–Related Operations and Functions

Igor Pro also supports a number of operations and functions for file or folder manipulation:

CopyFile	CopyFolder
DeleteFile	DeleteFolder
GetFileFolderInfo	SetFileFolderInfo
MoveFile	MoveFolder
CreateAliasShortcut	
NewPath	PathInfo
ParseFilePath	

Warning: Use caution when writing code that deletes or moves files or folders. These actions are not undoable.

Because the DeleteFolder, CopyFolder and MoveFolder operations have the potential to do a lot of damage if used incorrectly, they require user permission before overwriting or deleting a folder. The user controls the permission process using the Miscellaneous Settings dialog (Misc menu).

Writing to a Text File

You can generate output text files from Igor procedures or from the command line in formats acceptable to other programs. To do this, you need to use the **Open** and **Close** operations and the **fprintf** operation (page V-183) and the **wfprintf** operation (page V-735).

The following commands illustrate how you could use these operations to create an output text file. Each operation is described in detail in following sections:

```
Variable f1                          //make refNum variable
Open f1 as "A New File"              //create and open file
fprintf f1, "wave1\twave2\twave3\r" //write column headers
wfprintf f1, "" wave1, wave2, wave3 //write wave data
Close f1                             //close file
```

Open and Close Operations

You use the **Open** operation (page V-460) to open a file. For our purposes, the syntax of the Open operation is:

```
Open [/R/A/P=pathName/M=messageStr] variableName [as "filename"]
```

variableName is the name of a numeric variable. The Open operation puts a file reference number in that variable. You will need the reference number to access the file after you've opened it.

A file specifications consists of a path (directions for finding a folder) and a file name. In the Open operation, you can specify the path in three ways:

Using a full path as the filename parameter:

```
Open refNum as "hd:Data:Run123.dat"
```

Or using a symbolic path name and a file name:

```
Open/P=DataPath refNum as "Run123.dat"
```

Or using a symbolic path name and a partial path including the file name:

```
Open/P=HDPath refNum as ":Data:Run123.dat"
```

A symbolic path is a short name that refers to a folder on disk. See **Symbolic Paths** on page II-34.

If you do not provide enough information to find the folder and file of interest, Igor puts up a dialog which lets you select the file to open. If you supply sufficient information, Igor will just open the file without putting up the dialog.

To open a file for reading, use the /R flag. To add new data to a file, use the /A flag. If you omit both of these flags, you will overwrite any data that is already in the file.

If you open a file for writing (you don't use /R) then, if there exists a file with the specified name, Igor opens the file and overwrites the existing data in the file. If there is no file with the specified name, Igor creates a new file.

Warning: If you're not careful you can inadvertently lose data using the Open operation by opening for writing without using the /A flag. To avoid this, use the /R (open for read) or /A (append) flags.

Wave Reference Functions

It is common to write a user-defined function that operates on all of the waves in a data folder, on the waves displayed in a graph or table, or on a wave identified by a cursor in a graph. For these purposes, you need to use wave reference functions.

Wave reference functions are built-in Igor functions that return a reference that can be used in a user-defined function. Here is an example that works on the top graph. Cursors are assumed to be placed on a region of a trace in the graph.

```
Function WaveAverageBetweenCursors()
   WAVE/Z w = CsrWaveRef(A)
   if (!WaveExists(w))        // Cursor is not on any wave.
      return NaN
   endif

   Variable xA = xcsr(A)
   Variable xB = xcsr(B)
   Variable avg = mean(w, xA, xB)

   return avg
End
```

CsrWaveRef returns a wave reference that identifies the wave a cursor is on.

An older function, CsrWave, returns the *name* of the wave the cursor is on. It would be tempting to use this to determine the wave the cursor is on, but it would be incorrect. The name of a wave by itself does not uniquely identify a wave because it does not specify the data folder in which the wave resides. For this reason, we usually need to use the wave reference function CsrWaveRef instead of CsrWave.

This example uses a wave reference function to operate on the waves displayed in a graph:

```
Function SmoothWavesInGraph()
   String list = TraceNameList("", ";", 1)
   String traceName
   Variable index = 0
   do
      traceName = StringFromList(index, list)
      if (strlen(traceName) == 0)
         break          // No more traces.
      endif
      WAVE w = TraceNameToWaveRef("", traceName)
      Smooth 5, w
      index += 1
   while(1)
End
```

Here are the wave reference functions. See Chapter V-1, **Igor Reference**, for details on each of them.

Function	Comment
CsrWaveRef	Returns a reference to the Y wave to which a cursor is attached.
CsrXWaveRef	Returns a reference to the X wave when a cursor is attached to an XY pair.
WaveRefIndexed	Returns a reference to a wave in a graph or table or to a wave in the current data folder.
XWaveRefFromTrace	Returns a reference to an X wave in a graph. Used with the output of TraceNameList.
ContourNameToWaveRef	Returns a reference to a wave displayed as a contour plot. Used with the output of ContourNameList.
ImageNameToWaveRef	Returns a reference to a wave displayed as an image. Used with the output of ImageNameList.

Function	Comment
TraceNameToWaveRef	Returns a reference to a wave displayed as a waveform or as the Y wave of an XY pair in a graph. Used with the output of TraceNameList.
TagWaveRef	Returns a reference to a wave to which a tag is attached in a graph. Used in creating a smart tag.
WaveRefsEqual	Returns the truth two wave references are the same.

It is possible to create a user-defined function that returns a wave reference via a structure parameter. For simple applications, it may be easier to create a string function that returns a full path to the wave. Here is an example:

```
Function/S CursorAWave()
   WAVE/Z w= CsrWaveRef(A)
   if (WaveExists(w)==0)
      return ""
   endif
   return GetWavesDataFolder(w,2)
End

Function DemoCursorAWave()
   WAVE/Z w= $CursorAWave()
   if (WaveExists(w)==0)
      Print "oops: no wave"
   else
      Print "Cursor A is on the wave", WaveName(w)
   endif
End
```

DemoCursorAWave uses the $ operator to convert the full path to the wave returned by CursorAWave into a wave reference stored in w.

Processing Lists of Waves

Igor users often want to use a string list of waves in places where Igor is looking for just the name of a single wave. For example, they would like to do this:

```
Display "wave0;wave1;wave2"
```

or, more generally:

```
Function DisplayListOfWaves(list)
   String list                // e.g., "wave0;wave1;wave2"

   Display $list
End
```

Unfortunately, Igor can't handle this. However, there are techniques for achieving the same result.

Graphing a List of Waves

This example illustrates the basic technique for processing a list of waves.

```
Function DisplayWaveList(list)
   String list             // A semicolon-separated list.

   String theWave
   Variable index=0

   do
      // Get the next wave name
      theWave = StringFromList(index, list)
      if (strlen(theWave) == 0)
```

```
         break                        // Ran out of waves
      endif
      if (index == 0)                 // Is this the first wave?
         Display $theWave
      else
         AppendToGraph $theWave
      endif
      index += 1
   while (1)          // Loop until break above
End
```

To make a graph of all of the waves in the current data folder, you would execute

```
DisplayWaveList(WaveList("*", ";", ""))
```

Operating on the Traces in a Graph

In a previous section, we showed an example that operates on the waves displayed in a graph. It used a wave reference function, TraceNameToWaveRef. If you want to write a function that operates on *traces* in a graph, you would *not* use wave reference functions. That's because Igor operations that operate on *traces* expect *trace names*, not wave references. For example:

```
Function GrayOutTracesInGraph()
   String list = TraceNameList("", ";", 1)
   String traceName
   Variable index = 0
   do
      traceName = StringFromList(index, list)
      if (strlen(traceName) == 0)
         break                        // No more traces.
      endif

      // WRONG: ModifyGraph expects a trace name and w is not a trace name
      WAVE w = TraceNameToWaveRef("", traceName)
      ModifyGraph rgb(w)=(50000,50000,50000)

      // RIGHT
      ModifyGraph rgb($traceName)=(50000,50000,50000)

      index += 1
   while(1)
End
```

Using a Fixed-Length List

In the previous examples, the number of waves in the list was unimportant and all of the waves in the list served the same purpose. In this example, the list has a fixed number of waves and each wave has a different purpose.

```
Function DoLineFit(list)
   String list          // List of waves names: source, dest, weight
   String source, dest, weight

   // Pick out the expected wave names
   source = StringFromList(0, list)
   dest = StringFromList(1, list)
   weight = StringFromList(2, list)

   CurveFit line $source /D=$dest /W=$weight
End
```

You would invoke this function as follows:

```
DoLineFit("wave0;wave1;wave2")
```

Operating on Qualified Waves

This example illustrates how to operate on waves that match a certain criterion. It is broken into two functions - one that creates the list of qualified waves and a second that operates on them. This organization gives us a general purpose routine (ListOfMatrices) that we would not have if we wrote the whole thing as one function.

```
Function/S ListOfMatrices()
   String list = ""
   Variable index=0
   do
      WAVE/Z w=WaveRefIndexed("",index,4)    // Get next wave.
      if (WaveExists(w) == 0)
         break                               // No more waves.
      endif
      if (DimSize(w,1)>0 && DimSize(w,2)==0)
         // Found matrix. Add to list with separator.
         list += NameOfWave(w) + ";"
      endif
      index += 1
   while(1)             // Loop till break above.
   return list
End

Function ChooseAndDisplayMatrix()
   String theList = ListOfMatrices()

   String theMatrix
   Prompt theMatrix, "Matrix to display:", popup theList
   DoPrompt "Display Matrix", theMatrix
   if (V_Flag != 0)
      return -1
   endif

   WAVE m = $theMatrix
   NewImage m
End
```

The ExecuteCmdOnList Function

The ExecuteCmdOnList function is implemented by a WaveMetrics procedure file and executes any command for each wave in the list. For example, the following commands do a WaveStats operation on each wave.

```
Make wave0=gnoise(1), wave1=gnoise(10), wave2=gnoise(100)
ExecuteCmdOnList("WaveStats %s", "wave0;wave1;wave2;")
```

The ExecuteCmdOnList function is supplied in the Execute Cmd On List procedure file. See **The Include Statement** on page IV-145 for instructions on including a procedure file.

This technique is on the kludgy side. If you can achieve your goal in a more straightforward fashion without heroic efforts, you should use regular programming techniques.

The Execute Operation

Execute is a built-in operation that executes a string expression as if you had typed it into the command line. The main purpose of Execute is to get around the restrictions on calling macros and external operations from user functions.

We try to avoid using Execute because it makes code obscure and difficult to debug. If you can write a procedure without Execute, do it. However, there are some cases where using Execute can save pages of code or achieve something that would otherwise be impossible.

It is a good idea to compose the command to be executed in a local string variable and then pass that string to the Execute operation. Use this to print the string to the history for debugging. For example:

```
String cmd
sprintf cmd, "GBLoadWave/P=%s/S=%d \"%s\"", pathName, skipCount, fileName
Print cmd          // For debugging
Execute cmd
```

It is not necessary to use Execute to call the GBLoadWave external operation because it can be called directly from a user function. Prior to Igor Pro 5, this was not possible.

When you use Execute, you must be especially careful in the handling of wave names. See **Programming with Liberal Names** on page IV-147 for details.

When Execute runs, it is as if you typed a command in the command line. Local variables in macros and functions are not accessible. The example in **Calling an External Operation From a User-Defined Function** on page IV-177 shows how to use the sprintf operation to solve this problem.

Using a Macro From a User-Defined Function

A macro can not be called directly from a user function. To do so, we must use Execute. This is a trivial example for which we would normally not resort to Execute but which clearly illustrates the technique.

```
Function Example()
   Make wave0=enoise(1)
   Variable/G V_avg              // Create a global
   Execute "MyMacro(\"wave0\")"  // Invokes MyMacro("wave0")
   return V_avg
End

Macro MyMacro(wv)
   String wv
   WaveStats $wv     // Sets global V_avg and 9 other local vars
End
```

Execute does not supply good error messages. If the macro generates an error, you may get a cryptic message. Therefore, debug the macro *before* you call it with the Execute operation.

Calling an External Operation From a User-Defined Function

Prior to Igor Pro 5, external operations could not be called directly from user-defined functions and had to be called via Execute. Now it is possible to write an external operation so that it can be called directly. However, old XOPs that have not been updated still need to be called through Execute. This example shows how to do it.

If you attempt to directly use an external operation which does not support it, you will see an error dialog telling you to use Execute for that operation.

The external operation in this case is VDTWrite which sends text to the serial port. It is implemented by the VDT XOP.

```
Function SetupVoltmeter(range)
   Variable range    // .1, .2, .5, 1, 2, 5 or 10 volts

   String voltmeterCmd
   sprintf voltmeterCmd, "DVM volts=%g", range
   String vdtCmd
   sprintf vdtCmd "VDTWrite \"%s\"\r\n", voltmeterCmd
   Execute vdtCmd
End
```

In this case, we are sending the command to a voltmeter that expects something like:

```
DVM volts=.2<CR><LF>
```

to set the voltmeter to the 0.2 volt range.

The parameter that we send to the Execute operation is:

```
VDTWrite "DVM volts=.2\r\n"
```

The backslashes used in the second sprintf call insert two quotation marks, a carriage return, and a linefeed in the command about to be executed.

A newer VDT2 XOP exists which includes external operations that *can* be directly called from user-functions. Thus, new programming should use the VDT2 XOP and will not need to use Execute.

Other Uses of Execute

Execute can also accept as an argument a string variable containing commands that are algorithmically constructed. Here is a simple example:

```
Function Fit(w, fitType)
   WAVE w               // Source wave
   String fitType       // One of the built-in Igor fit types

   String name = NameOfWave(w)
   name = PossiblyQuoteName(name)   // Liberal names need quotes

   String cmd
   sprintf cmd, "CurveFit %s %s", fitType, name
   Execute cmd
End
```

Use this function to do any of the built-in fits on the specified wave. Without using the Execute operation, we would have to write it as follows:

```
Function Fit(w, fitType)
   WAVE w                  // Source wave
   String fitType          // One of the built-in Igor fit types

   strswitch(fitType)
      case "line":
         CurveFit line w
         break

      case "exp":
         CurveFit exp w
         break

      <and so on for each fit type>
   endswitch
End
```

Note the use of sprintf to prepare the string containing the command to be executed. The following would not work because when Execute runs, local variables and parameters are not accessible:

```
Execute "CurveFit fitType name"
```

Deferred Execution Using the Operation Queue

It is sometimes necessary to execute a command after the current function has finished. This is done using the Execute/P operation. For details, see **Operation Queue** on page IV-250.

Procedures and Preferences

You can set many preferences. Most of the preferences control the style of new objects, including new graphs, traces and axes added to graphs, new tables, columns added to tables and so on.

Preferences are usually used only for manual “point-and-click” operation. We usually don’t want preferences to affect the behavior of procedures. The reason for this is that we want a procedure to do the same

thing no matter who runs it. Also, we want it to do the same thing tomorrow as it does today. If we allowed preferences to take effect during procedure execution, a change in preferences could change the effect of a procedure, making it unpredictable.

By default, preferences do not take effect during procedure execution. If you want to override the default behavior, you can use the Preferences operation. From within a procedure, you can execute "`Preferences 1`" to turn preferences on. This affects the procedure and any subroutines that it calls. It stays in effect until you execute "`Preferences 0`".

When a *macro* ends, the state of preferences reverts to what it was when that macro started. If you change the preference setting within a *function*, the preferences state does *not* revert when that function ends. You must turn preferences on, save the old preferences state, execute Igor operations, and then restore the preferences state. For example:

```
Function AppendWithCapturedAxis()
   Variable oldPrefState
   Preferences 1;                    // Turn preferences on and
   oldPrefState = V_Flag             // save the old state.

   Make wave0=x
   Display/L=myCapturedAxis wave0    // myCapturedAxis is pref axis.

   Preferences oldPrefState          // Restore old prefs state.
End
```

Experiment Initialization Procedures

When Igor loads an experiment, it checks to see if there are any commands in the procedure window before the first macro, function or menu declaration. If there are such commands Igor executes them. This provides a way for you to initialize things. These initialization commands can invoke procedures that are declared later in the procedure window.

Also see **BeforeFileOpenHook** on page IV-259 and **IgorStartOrNewHook** on page IV-263 for other initialization methods.

Procedure Subtypes

A procedure subtype identifies the purpose for which a particular procedure is intended and the appropriate menu from which it can be chosen. For example, the Graph subtype puts a procedure in the Graph Macros submenu of the Windows menu.

```
Window Graph0() : Graph
   PauseUpdate; Silent 1      // building window...
   Display/W=(5,42,400,250) wave0,wave1,wave2
   <more commands>
End
```

When Igor automatically creates a procedure, for example when you close and save a graph, it uses the appropriate subtype. When you create a curve fitting function using the Curve Fitting dialog, the dialog automatically uses the FitFunc subtype. You usually don't need to use subtypes for procedures that you write.

This table shows the available subtypes and how they are used.

Subtype	Effect	Available for
Graph	Displayed in Graph Macros submenu.	Macros
GraphStyle	Displayed in Graph Macros submenu and in Style pop-up menu in New Graph dialog.	Macros
GraphMarquee	Displayed in graph marquee. This keyword is no longer recommended. See **Marquee Menu as Input Device** on page IV-140 for details.	Macros and functions
Table	Displayed in Table Macros submenu.	Macros
TableStyle	Displayed in Table Macros submenu and in Style pop-up menu in New Table dialog.	Macros
Layout	Displayed in Layout Macros submenu.	Macros
LayoutStyle	Displayed in Layout Macros submenu and in Style pop-up menu in New Layout dialog.	Macros
LayoutMarquee	Displayed in layout marquee. This keyword is no longer recommended. See **Marquee Menu as Input Device** on page IV-140 for details.	Macros and functions
Panel	Displayed in Panel Macros submenu.	Macros
FitFunc	Displayed in Function pop-up menu in Curve Fitting dialog.	Functions
ButtonControl	Displayed in Procedure pop-up menu in Button Control dialog.	Macros and functions
CheckBoxControl	Displayed in Procedure pop-up menu in Checkbox Control dialog.	Macros and functions
PopupMenuControl	Displayed in Procedure pop-up menu in PopupMenu Control dialog.	Macros and functions
SetVariableControl	Displayed in Procedure pop-up menu in SetVariable Control dialog.	Macros and functions

Memory Considerations

The maximum amount of memory available to Igor Pro (or any application) is 4 GB on Macintosh and 2, 3, or 4 GB on Windows regardless of the amount of installed RAM. Virtual memory on disk is always used, being constrained only by available disk space and access speed. See **Memory Management** on page III-425 for more information.

These memory limits are adequate for most applications although you may be surprised by out of memory errors with much less apparent memory usage. Oftentimes such errors are caused by fragmentation of RAM and the inability of either operating system to combine adjacent blocks of deallocated memory to create a contiguous memory block.

Most often such memory fragmentation occurs when you create and destroy many large waves during function execution. The best way to avoid these memory problems is to create these large waves once at the start of your function and then reuse them rather than killing and remaking them. The reason is that if you use Redimension to shrink a large wave, then a following Redimension that increases the wave size should succeed as it will be expanding back into the same memory block. You can also use Make/O can in place of Redimension.

Wave Reference Counting

The method Igor Pro uses to deallocate memory when waves are killed was improved in Igor Pro 6. The new method uses reference counting to determine when a wave is no longer referenced anywhere and memory can be safely deallocated, which should reduce the likelihood of out of memory errors, especially on Macintosh.

Because memory is not deallocated until all references to a given wave are gone, memory may not be freed when you think. Consider the function:

```
Function myfunc()
   Make/N=10E6 bigwave
   // do stuff
   FunctionThatKillsBigwave()
   // do more stuff
End
```

The memory allocated for bigwave is not deallocated until the function returns because an automatic WAVE reference variable of the same name still references the wave. To free memory for the second part, use the **WAVEClear** operation (page V-722):

```
Function myfunc()
   Make/N=10E6 bigwave
   // do stuff
   FunctionThatKillsBigwave()
   WAVEClear bigwave
   // do more stuff
End
```

The WAVEClear command takes a list of WAVE reference variables and stores NULL into them. It is the same as:

```
WAVE/Z wavref= $""
```

If you see the message "BUG: DecrementWaveRefCount" printed in the history area, please notify support@wavemetrics.com. If possible, please provide the steps necessary to reproduce the message.

If you suspect the new method is causing problems (a crash could occur if a reference was missed and a killed wave was accessed,) you can revert to the old method by executing

```
SetIgorOption doWAVERefCount= 0
```

You should do this in a blank experiment because it affects function compiling. You can revert to the new method by restarting Igor Pro or by executing

```
SetIgorOption doWAVERefCount= 2.
```

Creating Igor Extensions

Igor includes a feature that allows a C or C++ programmer to extend its capabilities. Using Apple's Xcode or Microsoft Visual C++, a programmer can add command line operations and functions to Igor. These are called external operations and external functions. Experienced programmers can also add menu items, dialogs and windows.

A file containing external operations or external functions is called an "XOP file" or "XOP" (pronounced "ex-op") for short.

Here are some things for which you might want to write an XOP:

- To do number-crunching on waves.
- To import data from a file format that Igor does not support.
- To acquire data directly into Igor waves.
- To communicate with a remote computer, instrument or database.

The main reasons for writing something as an XOP instead of writing it as an Igor procedure are:

- It needs to be blazing fast.
- You already have a lot of C code that you want to reuse.
- You need to call drivers for data acquisition.
- It requires programming techniques that Igor doesn't support.

- You want to add your own dialogs or windows.

Writing an XOP that does number-crunching is considerably easier than writing a stand-alone program. You don't need to know anything about the Macintosh Toolbox or the Windows API. Writing an XOP that adds dialogs or windows to Igor requires more knowledge but is still easier than writing a stand-alone program.

If you are a C or C++ programmer and would like to extend Igor with your own XOP, you will need to purchase the Igor External Operations Toolkit from WaveMetrics. This toolkit contains documentation on writing XOPs as well as the source code for many of the WaveMetrics sample XOPs. It supplies a large library of routines that enable communication between Igor and the external code. You will also need the a recent version of Xcode, or Microsoft Visual C++.

Chapter
IV-8

Debugging

Debugging Procedures

There are two techniques for debugging procedures in Igor:

- Using print statements
- Using the symbolic debugger

For most situations, the symbolic debugger is the most effective tool. In some cases, a strategically placed print statement is sufficient.

Debugging With Print Statements

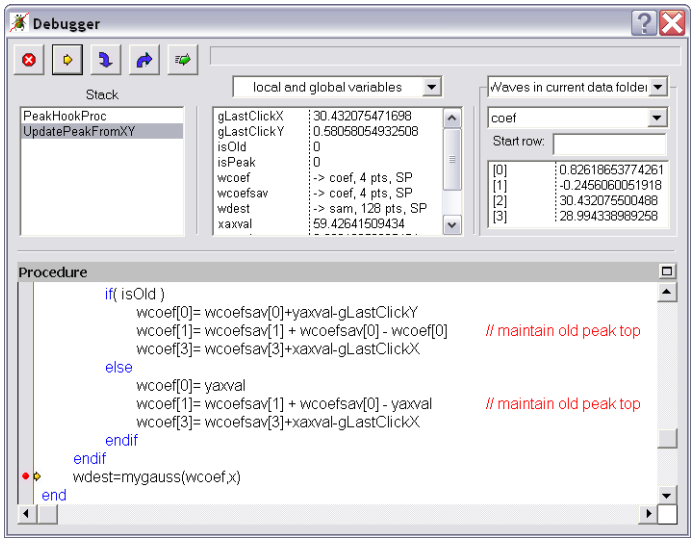
This technique involves putting print statements at a certain point in a procedure to display debugging messages in Igor's history area. In this example, we use Printf to display the value of parameters to a function and then Print to display the function result.

```
Function Test(w, num, str)
   Wave w
   Variable num
   String str

   Printf "Wave=%s, num=%g, str=%s\r", NameOfWave(w), num, str

   <body of function>

   Print result
   return result
End
```

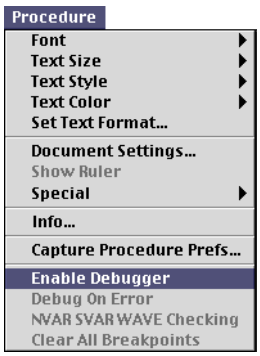
See **Creating Formatted Text** on page IV-230 for details on the Printf operation.

The Debugger

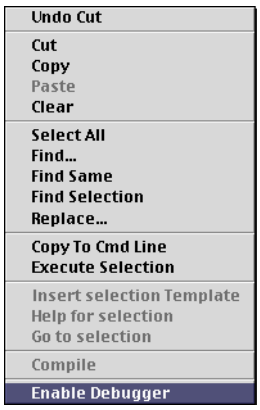
When a procedure doesn't produce the results you want, you can use Igor's built-in debugger to observe the execution of user-defined macros and functions while single-stepping through the lines of code.



The debugger is normally disabled. Enable it using either the Procedure menu or by Control-clicking (*Macintosh*) or right-clicking (*Windows*) in any procedure window to get the pop-up menu:



Procedure Menu



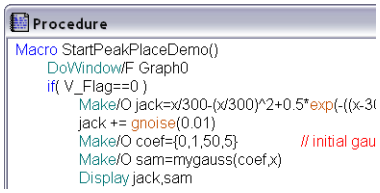
Contextual menu

The debugger window will automatically appear when one of the following events occurs:

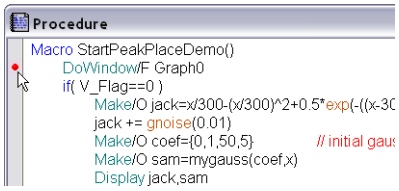
1. A breakpoint that you previously set is encountered.
2. An error occurs, and you have enabled debugging on that kind of error.
3. An error dialog is presented, and you click the Debug button.

Setting Breakpoints

When you want to observe a particular routine in action, set a breakpoint on the line where you want the debugger to appear. To do this, open the procedure window which contains the routine, and click in the left "breakpoint margin". The breakpoint margin appears only if the debugger has been enabled:



Debugger Disabled



Debugger Enabled

When a line of code marked with the red dot (denoting a set breakpoint) is about to execute, the debugger window will appear.

Click the red dot again to clear the breakpoint. Control-click (*Macintosh*) or right-click (*Windows*) and use the pop-up menu to clear all breakpoints or disable a breakpoint on the currently selected line of the procedure window.

Debugging on Error

You can automatically open the debugger window when an error occurs. There are two categories of errors to choose from:

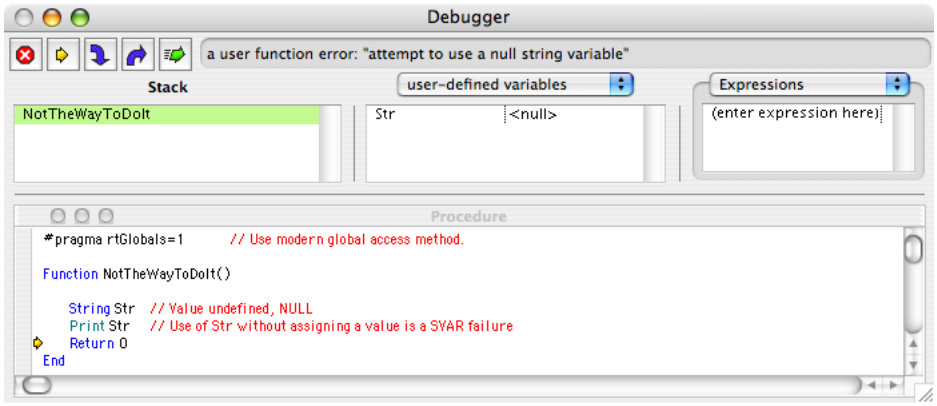
Debug On Error	Any runtime error except failed NVAR, SVAR, or WAVE references.
NVAR SVAR WAVE Checking	Failed NVAR, SVAR, or WAVE references.

You can use the /Z flag to hide failures from SVAR, NVAR and WAVE checking. This is appropriate where the reference is subsequently checked with WaveExists, NVAR_Exists, or SVAR_Exists:

```
WAVE/Z wv=<pathToPossiblyMissingWave>

if( WaveExists(wv) )
   <do something with wv>
endif
```

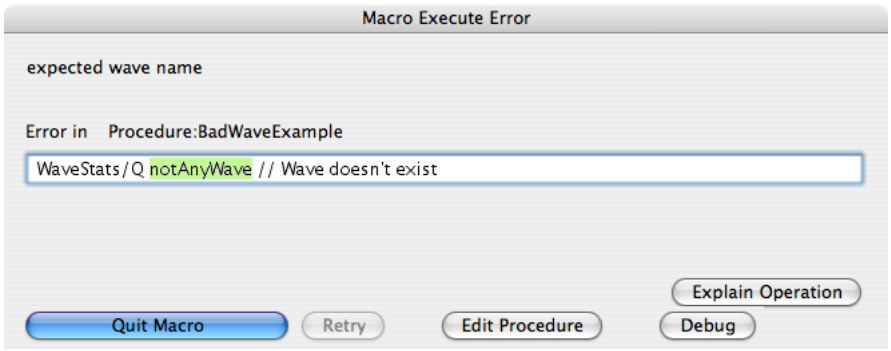
See **Runtime Lookup of Globals** on page IV-50 for details.

Use the Procedure or pop-up menus to choose either or both error categories. If the selected error occurs, the debugger will open and an error message will appear in the debugger window's status area. The error message was generated by the command *above* the yellow arrow:



Macro Execute Error: The Debug Button

When the debugger is enabled and an error occurs in a Macro, an error dialog is presented that will (usually) have a Debug button in it. Click the button to open the debugger window.



Errors in macros (or procs) are reported immediately after they occur.

When an error is reported by a function, a different dialog appears long after the error was actually committed. The Debug On Error option catches errors in functions immediately after they are committed.

Stepping Through Your Code

Begin by enabling the debugger and setting a breakpoint on the line of code you are interested in, or begin when the debugger automatically opens because of an error. Use the buttons at the top of the debugger window to step through your code:

The Stop button ceases execution of the running function or macro before it completes. This is equivalent to clicking Igor's Abort button (*Windows*) or pressing Command-period (*Macintosh*) while the procedure is running.

Keyboard shortcuts: (none)

Note: Pressing Command-period on a Macintosh while the debugger window is showing is equivalent to clicking the *Go* button, not the Stop button.

The Step Over button executes the next line. If the line contains a call to one or more subroutines, execution continues until the subroutines return or until an error or breakpoint is encountered. Upon return, execution halts until you click a different button.

Keyboard shortcuts: Enter, keypad Enter, or Return

The Step Into button executes the next line. If that line contains a call to one or more subroutines, execution halts when the first subroutine is entered. The Stack list of currently executing routines shows the most recently entered routine as the last item in the list.

Keyboard shortcuts: +, =, or keyPad +

The Step Out button executes until the current subroutine is exited, or an error or breakpoint is encountered.

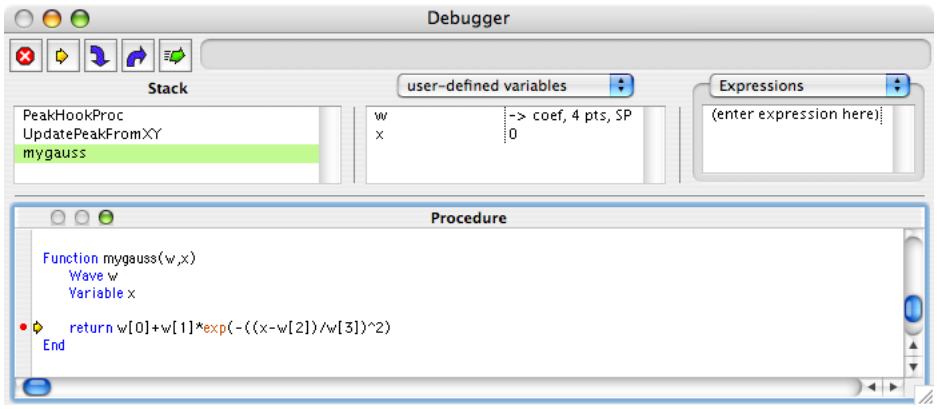
Keyboard shortcuts: -, _ (underscore) or keypad -

The Go button resumes program execution. The debugger window remains open until execution completes or an error or breakpoint is encountered.

Keyboard shortcuts: Esc

The Stack and Variables Lists

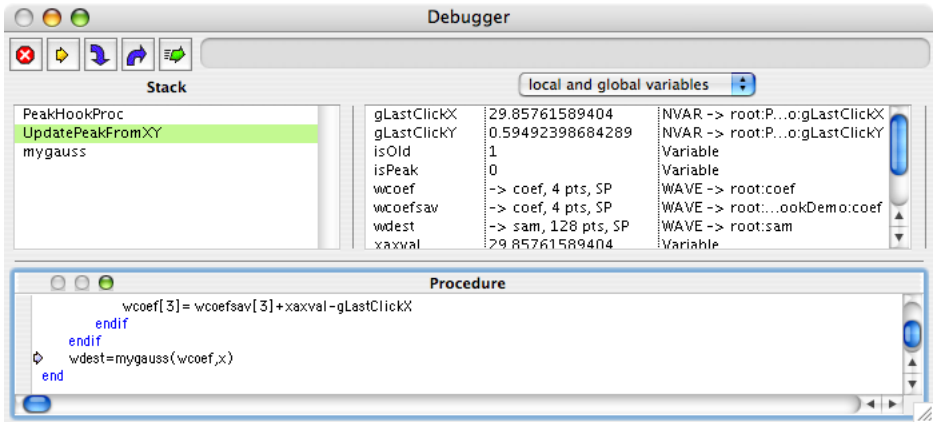
The Stack List shows the routine that is currently executing and the chain of routines that called it. The top item in the list is the routine that began execution and the bottom item is the routine which is currently executing.



In this example, the routine that started execution is PeakHookProc, which most recently called UpdatePeakFromXY, which then called the currently executing mygauss user function.

The Variables List (to the right of the Stack List) shows that the function parameters w and x have the values coef (a wave) and 0 (a number). The pop-up menu controls which variables are displayed in the list; the example shows only user-defined local variables.

You can examine the variables associated with any routine in the Stack List by simply selecting the routine:



Here we've selected UpdatePeakFromXY, the routine that called mygauss (see the light blue arrow). Notice that the Variables List is showing the variables that are local to UpdatePeakFromXY.

For illustration purposes, the Variables List has been resized by dragging the dividing line, and the pop-up menu has been set to show local and global variables and type information.

The Variables List Columns

The Variables List shows either two or three columns, depending on whether the "show variable types" item in the Variable pop-up menu is checked.

The first column is the name of the local variable. Note that the name of an NVAR, SVAR, or WAVE reference is a local name referring to a global object.

The second column is the value of the local variable. Double-click the second column to edit strings or variables.

In the case of a wave, the size and precision of the wave are shown here. The "->" characters mean "refers to". In our example wcoef is a local name that refers to a (global) wave named coef, which is one-dimensional, has 4 points, and is single precision.

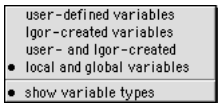
To determine the value of a particular wave element, use the **Wave, Structures, and Expressions Pane** on page IV-191.

The third (optional) column shows what the type of the variable is, whether Variable, String, NVAR, SVAR, or WAVE. For global references, the full path (including the data folder) to the global is shown.

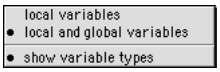
Note: The currentDF item is separated by a dashed line because currentDF is not really a global variable; it is a convenient name to identify the current data folder. See Chapter II-8, **Data Folders**, for more information about data folders.

Variables Pop-Up Menu

The Variables pop-up menu controls which information is displayed in the Variables List. The menu items differ when the routine chosen from the Stack List is a function or a macro/proc:



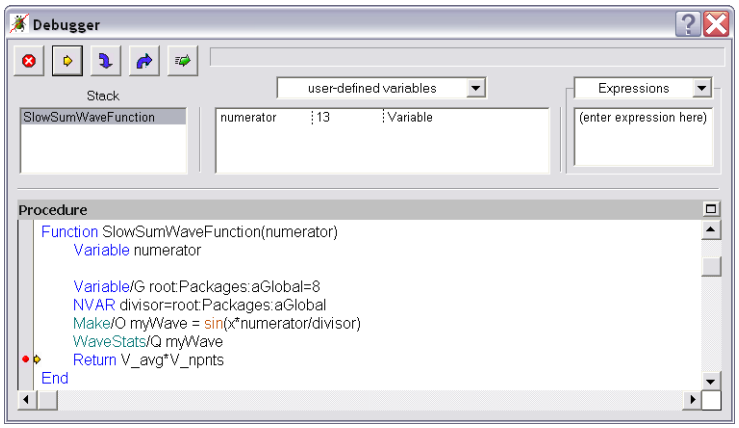
Pop-up Menu for Functions



Pop-up Menu for Macros, Procs, and Windows

Function Variables

The SlowSumWaveFunction example below illustrates how different kinds of variables in functions are classified:

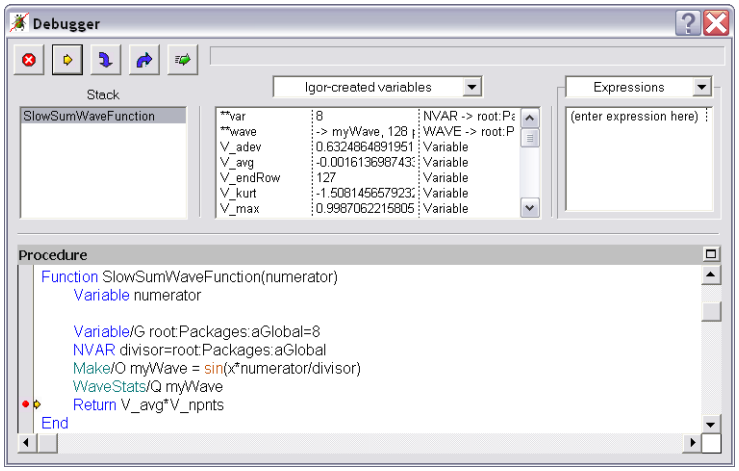


User-defined variables in functions include all items passed as parameters (numerator in this example) and any local strings and variables.

Local variables exist while a procedure is running, and cease to exist when the procedure returns; they never exist in a data folder like globals do.

NVAR, SVAR, WAVE, Variable/G and String/G references point to global variables, and therefore, aren't listed as user-defined (local) variables.

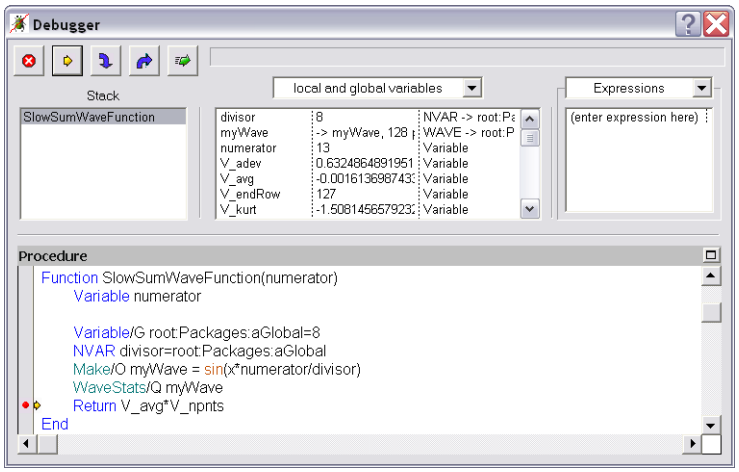
Use "Igor-created variables" to show local variables that Igor creates for functions when they call an operation or function that returns results in specially-named variables. The **WaveStats** operation (see page V-729), for example, defines V_adev, V_avg, and other variables to contain the statistics results:



Note: Some global references are created automatically for commands such as the Make operation; they have names that start with "**". These are shown only when Igor-created variables is selected.

The "user- and Igor-created" menu item shows both kinds of local variables.

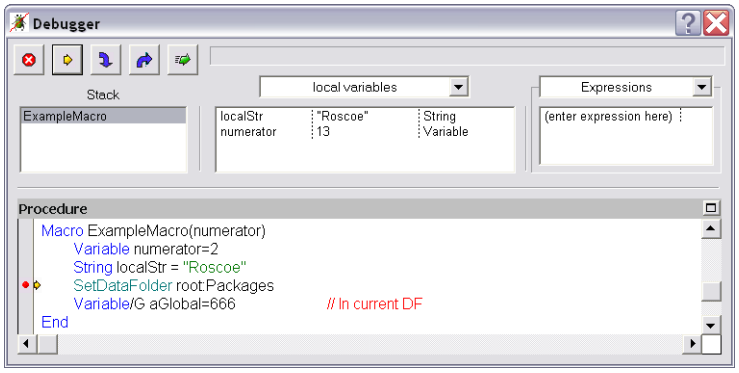
The "local and global variables" item shows user-created local variables, most Igor-created local variables, and references to global variables and waves through NVAR, SVAR, and WAVE references:



Choosing "local and global variables" also displays the current data folder at the end of the list as the mythical currentDF variable (see **The Variables List Columns** on page IV-188).

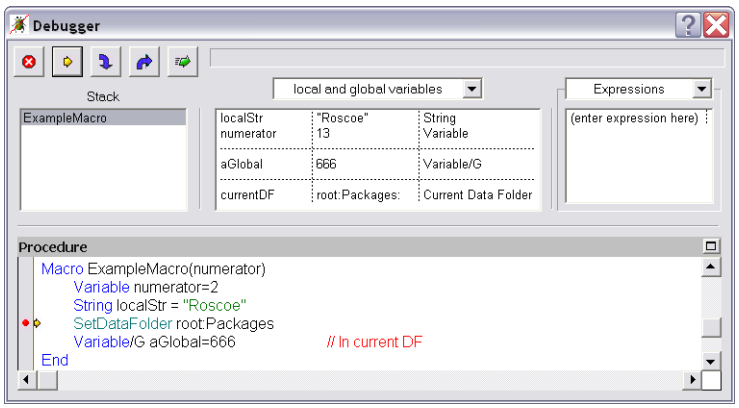
Macro Variables

The ExampleMacro below illustrates how variables in Macros, Procs or Window procedures are classified as locals or globals:



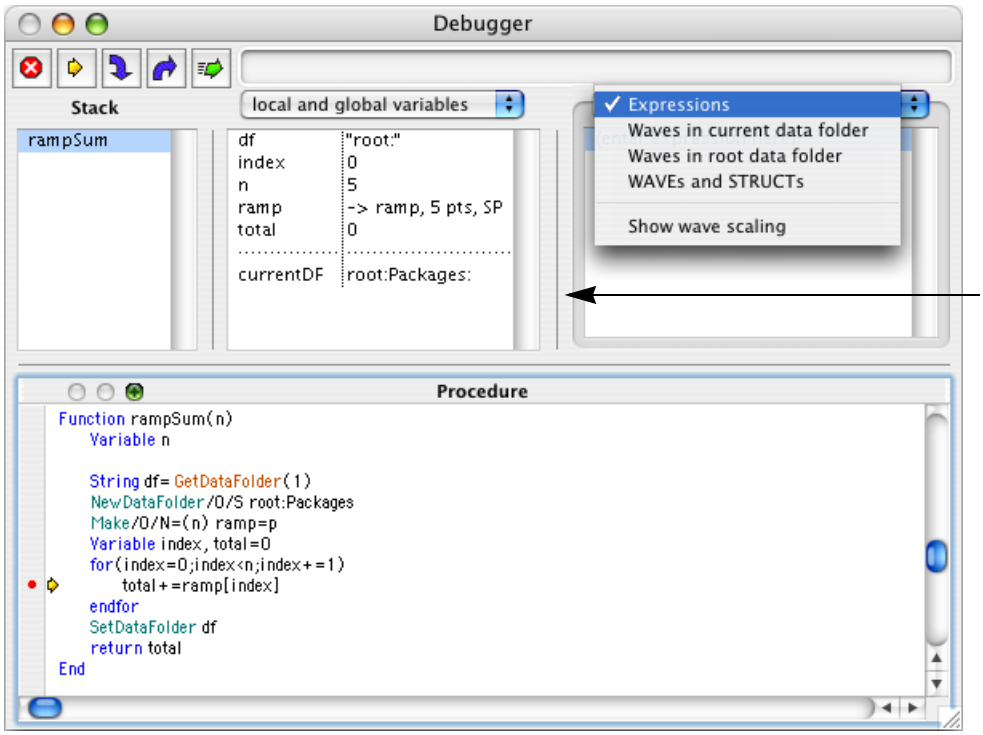
Local variables in macros include all items passed as parameters (numerator in this example) and local variables and local strings (localStr), and Igor-created local variables created by operations such as WaveStats.

Global variables in macros include all items in the current data folder, whether they are used in the macro or not. If the data folder changes because of a SetDataFolder operation, the list of global variables also changes. Note that there are no NVAR, SVAR, WAVE, or STRUCT references in a macro.



Wave, Structures, and Expressions Pane

The waves, structures, and expressions pane is on the right side of the variables list:



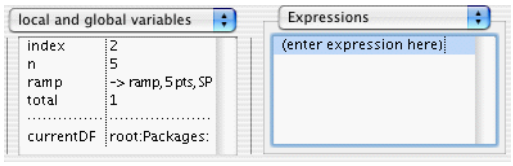
This pane is hidden when there isn't enough room or when the divider between the pane and the variables list is dragged all the way to the right. Drag the divider to the left to show the pane. You may need to widen the window to make room.

The pop-up menu controls what is shown in the pane:

Pop-up Menu Selection	Pane Contents
Expressions	Numeric or string expressions which are evaluated in the context of the selected function or macro.
Waves in current data folder	A list of waves in the current data folder and the contents of one selected wave.
Waves in root data folder	The same, but for the root: data folder.
WAVEs and STRUCTs	A list of WAVE and STRUCT references in the selected function. Disabled when a macro or proc is selected in the Stack list.
Show wave scaling	When unchecked, wave indexes are shown using rows, columns, layers, and chunks. The values are enclosed in square brackets. When checked, wave indexes are shown using the scaled values (see **Waveform Model of Data** on page II-77) and the value are enclosed in parentheses

Expressions

Replace the "(enter expression here)" prompt:



by clicking it, typing a numeric or string expression, and pressing Return.

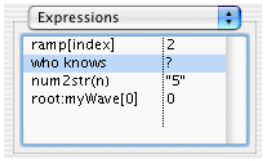
Adding an expression adds a blank row at the end of the list that can be double-clicked to enter another expression. You can edit any of the expressions by double-clicking and typing.

The expression can be removed by selecting it and pressing Delete or Backspace.

The result of the expression will be recomputed when stepping through procedures. The expressions are evaluated in the context of the currently selected procedure.

Global expressions are evaluated in the context of the current data folder, though you can specify the data folder explicitly as in the example below.

If an expression is invalid the result is shown as "?":

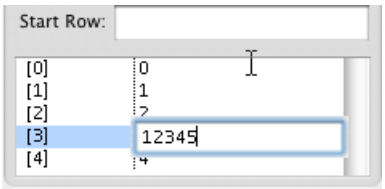


The expressions are discarded when a new Igor experiment is opened or when Igor quits.

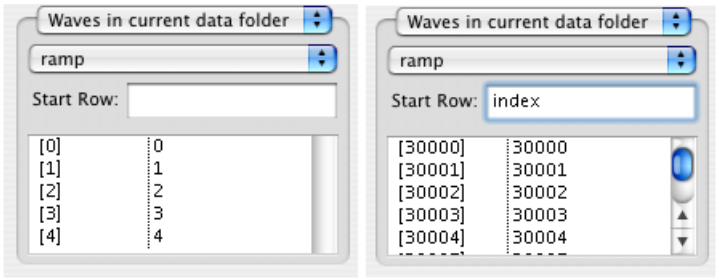
Waves in Current or Root Data Folder

The debugger shows the waves in the specified data folder. You can select one of the waves to inspect a one-dimensional portion.

You can edit the value of a wave element by double-clicking the value column and editing the value there. Press return to enter the value.

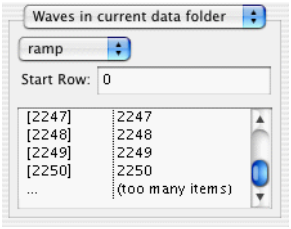


For a one-dimensional wave, you can specify the first row of the wave to display in the "Start Row:" control, or leave it blank to display starting with row 0. To look beyond the start of very long waves, you can enter a number or numeric expression into "Start Row:"



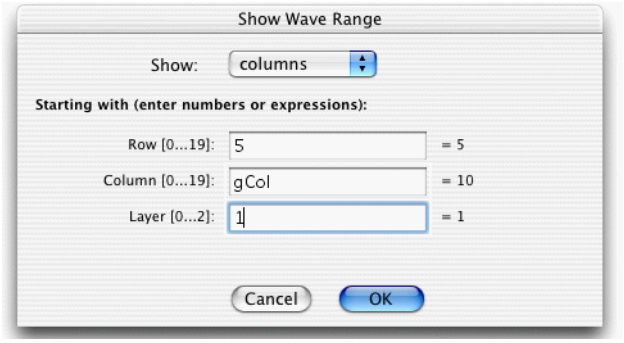
The wave value list will display only a limited number of rows, 1024 by default, in order to keep the debugger from being sluggish. When the list cannot display all of the rows, the last row in the list will say "(too many items)". You can change the maximum number of rows displayed here with the SetIgorOption command:

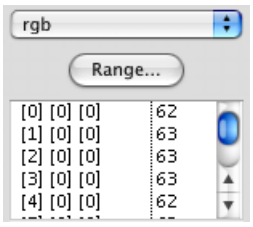
```
SetIgorOption DebuggerWaveRows=numRows
```



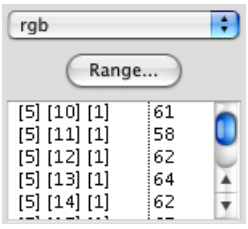
Use the Start Row to bring the row you wish to inspect within view.

Multidimensional waves can also be viewed, but only a one-dimensional subset. To view a particular range of values use the "Range..." button (which appears only when a multidimensional wave is chosen in the pop-up menu) and the resulting Show Wave Range dialog:



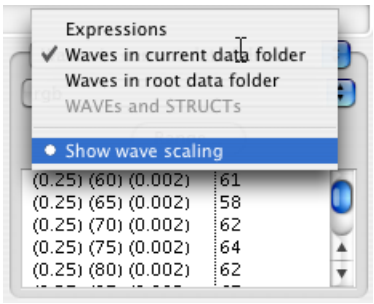


Default: rows 0 … n of column 0, layer 0



rows 5 … of column 10, layer 1

Choose "Show wave scaling" to display wave indexes using the scaled values (see **Waveform Model of Data** on page II-77). Scaled values are displayed in parentheses:



WAVEs and STRUCTs

WAVE and STRUCT references are allowed only in functions, so this pop-up menu choice is disabled when the routine selected in the Stack list is a macro or proc.

A WAVE reference points to a global wave, which can be in any data folder.

A STRUCT reference points to a structure which exists on the runtime stack and does not exist as a global object in any data folder.

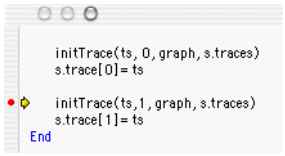
Each is composed of individual elements, and these can be inspected in the list in this pane.

Here's the code of an example contrived to demonstrate inspecting a structure's elements using the WAVEs and STRUCTs pane:

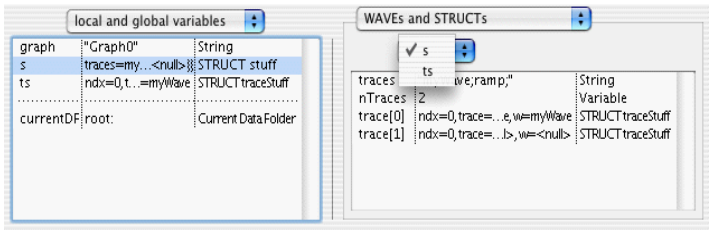
```
Structure stuff
   String traces
   Variable nTraces
   STRUCT traceStuff trace[2]
EndStructure

Structure traceStuff
   Variable ndx
   String trace
   WAVE w
EndStructure

Function top(graph)
   String graph

   STRUCT stuff s

   s.traces=TraceNameList(graph, ";", 1 )
   s.nTraces= ItemsInList(s.traces)

   STRUCT traceStuff ts

   initTrace(ts, 0, graph, s.traces)
   s.trace[0]= ts

   initTrace(ts,1, graph, s.traces)       // breakpoint set here
   s.trace[1]= ts
End

Function initTrace(ts, index, graph, traces)
   STRUCT traceStuff &ts
   Variable index
   String graph, traces
```

```
    ts.ndx= index
    ts.trace= StringFromList(index,traces)
    WAVE ts.w=TraceNameToWaveRef(graph, ts.trace)
End
```

Running top("Graph0") with the breakpoint at the second initTrace call



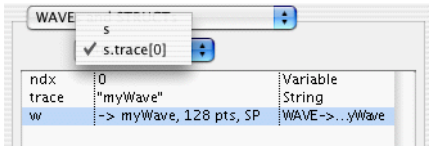
results in the Variables list shown below. You can see in the Variables list that "s" is the name of structure of type "stuff":



Double-clicking the structure s row in the Variables list (or selecting s from the pop-up menu of WAVEs and STRUCTs) displays the content of the structure.

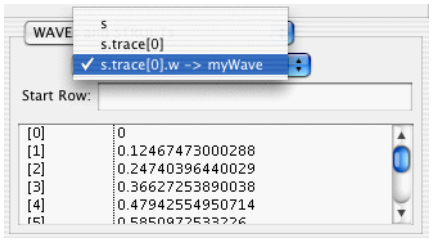
You can see that the listing of the contents of a structure on a single line is of limited use!

To see the contents of STRUCT traceStuff trace[0], double-click the trace[0] row in the WAVEs and STRUCTs list:



You can edit the value of ndx and trace by double-clicking the second column.

To see the contents of s.trace[0].w, double-click w's row:



You can edit the values of myWave by double-clicking the second column.

Use the pop-up menu to view previous levels of the s structure. When you choose the top-level ("s"), the menu will again be filled with all the WAVEs and STRUCTs in the currently selected function.

The Procedure Pane

The procedure pane contains a copy of the procedure window of the routine selected in the Stack List. You can set and clear breakpoints in this pane just as you do in a procedure window, using the breakpoint margin and the Control-click (*Macintosh*) or right-click (*Windows*) menu.

A very useful feature of the debugger is the automatic text expression evaluator that shows the value of a variable or expression under the cursor. On the Macintosh, the value is displayed in the top-right corner of the debugger. On Windows, a tooltip is displayed near the cursor.

This is often faster than scrolling through the Variables List or entering an expression in the Expressions List to determine the value of a variable, wave, or structure member reference.

The value of a variable can be displayed whether or not the variable name is selected. To evaluate an expression such as "wave[ii]+3", the expression must be selected and the cursor must be over the selection.

The debugger won't evaluate expressions that include calls to user-defined functions; this prevents unintended side effects (a function could overwrite a wave's contents, for example). You can remove this limitation by creating the global variable root:V_debugDangerously and setting it to 1.

After You Find a Bug

Editing in the debugger window is disabled because the code is currently executing. Tracking down the routine after you've exited the debugger is easy if you follow these steps:

1) Scroll the debugger text pane back to the name of the routine you want to modify, and select it.
2) Control-click (*Macintosh*) or right-click (*Windows*) the name, and choose "Go to <routineName>" from the pop-up menu.
3) Exit the debugger by clicking the "Go" button or by pressing Escape.

Now the selected routine will be visible in the top procedure window, where you can edit it.

Debugger Shortcuts

Action	Shortcut
To enable debugger	Choose Enable Debugger from the Procedure menu or choose Enable Debugger from the procedure window's pop-up menu after Control-clicking (*Macintosh*) or right-clicking (*Windows*).
To automatically enter the debugger when an error occurs	Choose Debug on Error from the Procedure menu or choose Enable Debugger from a procedure window's pop-up menu after Control-clicking (*Macintosh*) or right-clicking (*Windows*).
To set or clear a breakpoint	Click in the left margin of the procedure window or click anywhere on the procedure window line where you want to set or clear the breakpoint and choose Set Breakpoint or Clear Breakpoint from a procedure window's pop-up menu after Control-clicking (*Macintosh*) or right-clicking (*Windows*).
To enable or disable a breakpoint	Shift-click a breakpoint in the left margin of the procedure window.
	Click anywhere on the procedure window line where you want to enable or disable the breakpoint and choose Enable Breakpoint or Disable Breakpoint procedure window's pop-up menu after Control-clicking (*Macintosh*) or right-clicking (*Windows*).
To execute the next command	On *Macintosh* press Enter, keypad Enter, or Return. For *Windows*, if no button has the focus, press Enter or Return. Otherwise, click the yellow arrow button.
To step into a subroutine	Press the +, =, or keypad + keys, or click the blue descending arrow button.
To step out of a subroutine to the calling routine	Press the -, _ (underscore) or keypad - keys, or click the blue ascending arrow button.
To resume executing normally	Press Escape (Esc), or click the green arrow button.

Action	Shortcut
To cancel execution	Click the red stop sign button.
To edit the value of a macro or function variable	Double-click the second column of the variables list, edit the value, and press Return or Enter.
To set the value of a function's string to null	Double-click the second column of the variables list, type "<null>" (without the quotes), and press Return or Enter.
To view the current value of a macro or function variable	Move the cursor to the procedure text of the variable name and wait. On *Macintosh*, the value appears to the right of the debugger buttons. On *Windows*, the value appears in a tooltip window.
To view the current value of an expression	Select the expression text with the cursor, position the cursor over the selection, and wait. (Expressions involving user-defined functions will not be evaluated unless V_debugDangerously is set to 1.)
To view global values in the current data folder	Choose "local and global variables" from the debugger pop-up menu.
To view type information about variables	Choose "show variable types" from the debugger pop-up menu.
To resize the columns in the variables list	Drag a divider in the list to the left or right.
To show or hide the Waves, Structs, and Expressions pane	Drag the divider on the right side of the Variables list left or right.

Chapter
IV-9

Dependencies

Dependency Formulas

Igor Pro supports "dependent objects". A dependent object can be a wave, a global numeric variable or a global string variable that has been linked to an expression. The expression to which an object is linked is called the object's "dependency formula" or "formula" for short.

The value of a dependent object is updated whenever any other global object involved in the formula is modified (even if its value stays the same). We say that the dependent object *depends* on these other global objects *through* the formula.

You might expect that an assignment such as:

```
wave1 = sin(K0*x/16)
```

meant that wave1 would be updated whenever K0 changed. It doesn't; values are computed for wave1 only once, and the relationship between wave1 and K0 is forgotten.

However, if the = in the above assignment is replaced with :=

```
wave1 := sin(K0*x/16)
```

then Igor *does* create just such a dependency. Now whenever K0 changes, the contents of wave1 will be updated. In this example, wave1 is a dependent object. It depends on K0, and sin(K0*x/16) is wave1's dependency formula.

You can also establish a dependency using the SetFormula operation, like this:

```
SetFormula wave1, "sin(K0*x/16)"
```

Wave1 depends on K0 because K0 is a changeable variable. Wave1 *also* depends on the function x (x is not a variable) that returns the X scaling of the destination wave (wave1). When the X scaling of wave1 changes, the values that the x function returns change, so this dependency assignment is reevaluated. The remaining terms (sin and 16) are not changeable, so wave1 does not depend on them.

Like other assignment statements, the data folder context for the right-hand side is that of the destination object. Therefore, in the following example, wave2 and var1 must be in the data folder named foo, var2 must be in root and var3 must be in root:bar.

```
root:foo:wave1 := wave2*var1 + ::var2 + root:bar:var3
```

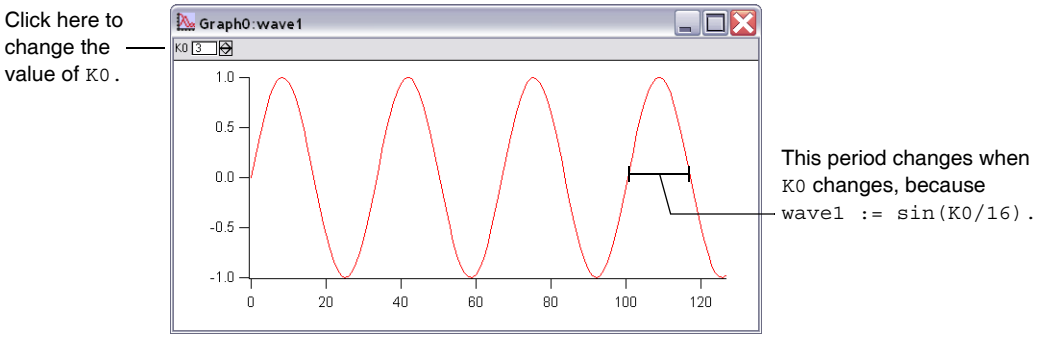
Data Folders are described inChapter II-8, **Data Folders**.

A dependency assignment is often used in conjunction with SetVariable controls (see page III-362) and Value Display controls (see page III-363).

Here's a simple example. Execute these commands on the command line:

```
K0=1
Make/O wave1:=sin(K0*x/16)
Display /W=(4,53,399,261) wave1
ControlBar 23
SetVariable setvar0,size={60,15},value=K0
```

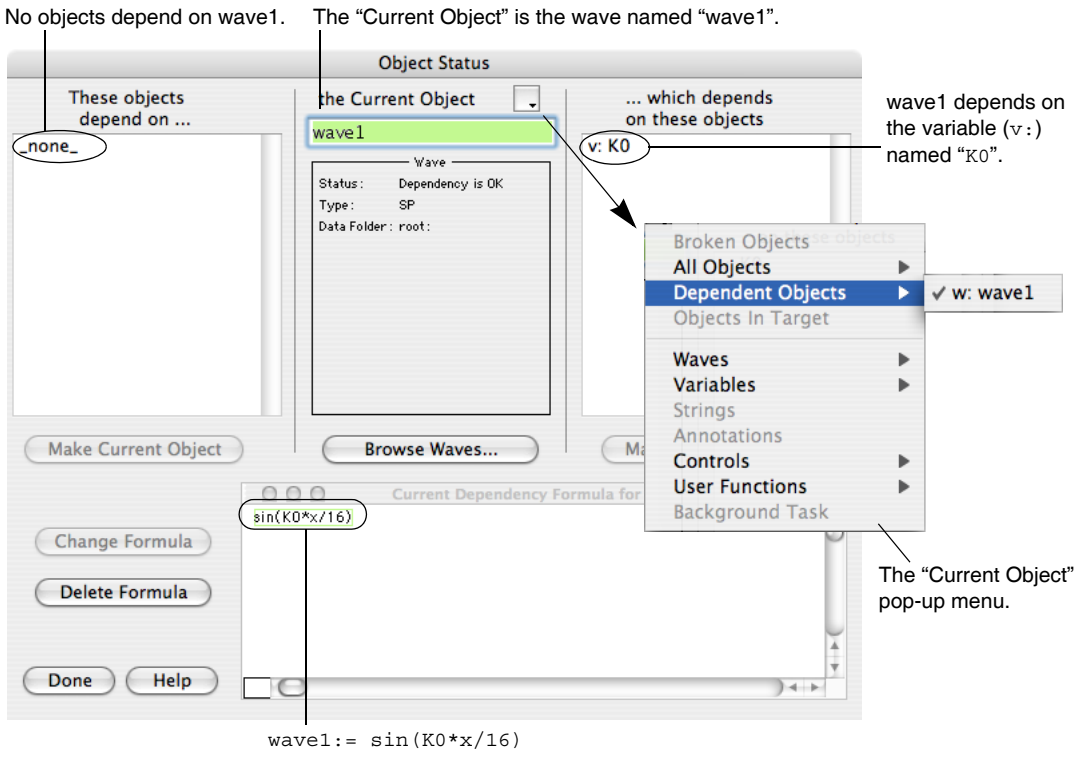
This results in the following graph:



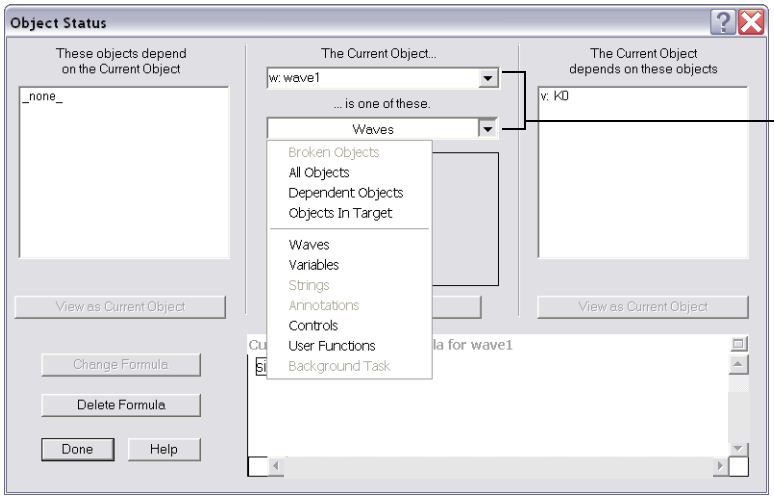
Click the SetVariable control's up and down arrows to adjust K0 and observe that wave1 is automatically updated.

Dependencies and the Object Status Dialog

You can use the Object Status dialog in the Misc menu to check dependencies. On the Macintosh, all dependent objects are listed in the Current Object pop-up menu under Dependent Objects:



The Windows version of the Object Status dialog is essentially the same but for a slightly different arrangement of the pop-up menus:



The Status field in the box below the current object name indicates any dependency status:

- "No dependency" means that the current object does not depend on anything.
- "Dependency is OK" means that the dependency formula successfully updated the current object.
- "Update failed" means that the dependency formula used to compute the current object's value failed, probably because there is a syntax error in the formula or one of the objects referenced in the formula does not exist or has been renamed. If an update fails, then the objects that depend on that update are broken and they appear in the Broken Objects submenu. See **Broken Dependent Objects** on page IV-206.

You can create a new dependency formula with the New Formula button, delete one with the Delete Formula button, change an existing one by typing in the Dependency Formula window and clicking the Change Formula button, and undo that change by clicking the Restore Formula button.

This is discussed further in **The Object Status Dialog** on page III-417.

You can also read the text of a dependency formula with the string function GetFormula, and set it with the operation SetFormula.

Numeric and String Variable Dependencies

Dependencies can also be created for global (but not local) user-defined numeric and string variables. Here is a user-defined function that creates a dependency (the global variable `recalculateThis` will depend on another global variable `dependsOnThis`):

```
Function TestRecalculation()
   Variable/G recalculateThis
   Variable/G dependsOnThis= 1

   // Create dependency on global variable
   SetFormula recalculateThis, "dependsOnThis"

   Print recalculateThis      // Prints original value

   dependsOnThis = 2          // Changes something recalculateThis

   DoUpdate                   // Make Igor recalculate formulae

   Print recalculateThis      // Prints updated value
End
```

Running this function prints the following to the history area:

```
•TestRecalculation()
   1
   2
```

The call to DoUpdate is needed because Igor recalculates dependency formulas only when no user-defined functions are running or when DoUpdate is called.

This function uses SetFormula to create the dependency because the := operator is not allowed in user-defined functions.

Note: You can not create a dependency *for* system numeric variables K0...K19 or veclen. You can create a dependency for something else *on* those variables, as in the first example of this chapter. In general, it is actually best if you do *not* use system variables in dependencies, since they are involved in Curve Fitting. You should define your own global variables for use in dependencies.

Wave Dependencies

The assignment statement:

dependentWaveName := *formula*

creates a dependency and links the dependency formula to the dependent wave. Whenever any change is made to any object in the formula, the contents of the dependent wave are updated.

The command

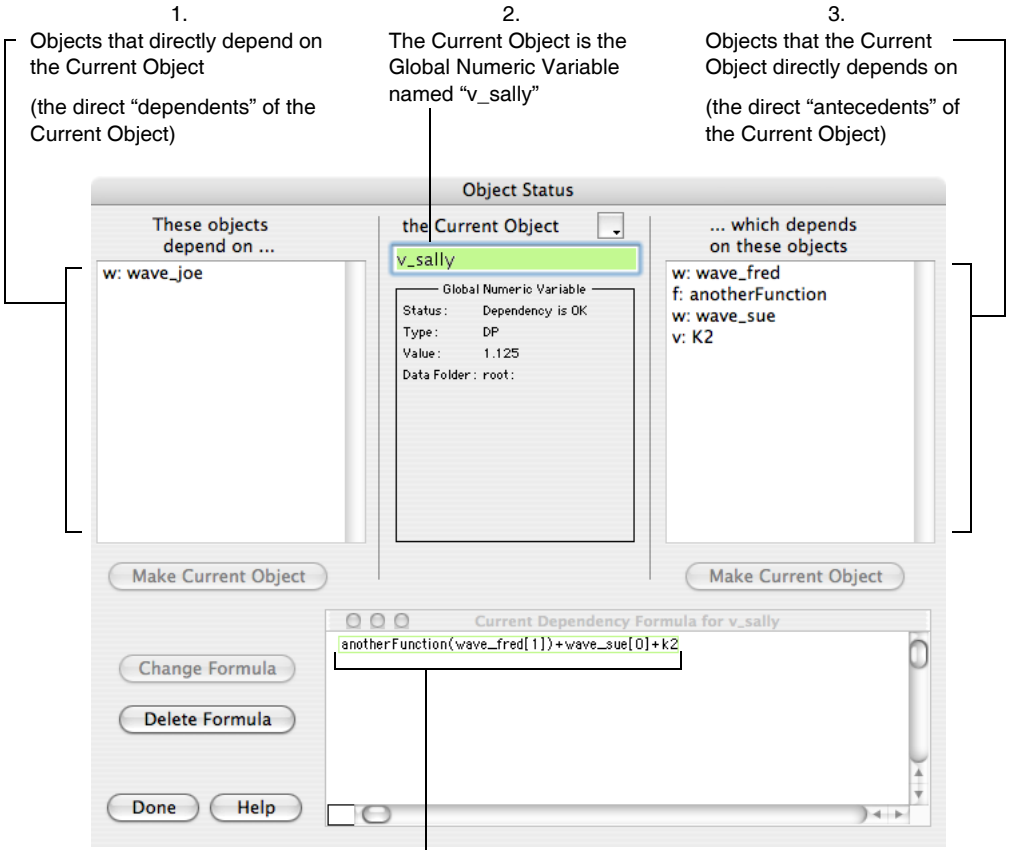
SetFormula *dependentWaveName*, "*formula*"

establishes the same dependency.

Cascading Dependencies

"Cascading dependencies" refers to the situation that arises when an object depends on a second object, which in turn depends on a third object, etc. When an object changes, all objects that directly depend on that object are updated, *and* objects that depend directly on those updated objects are updated until no more updates are needed.

The Object Status dialog shows three levels of dependency anchored on the Current Object:

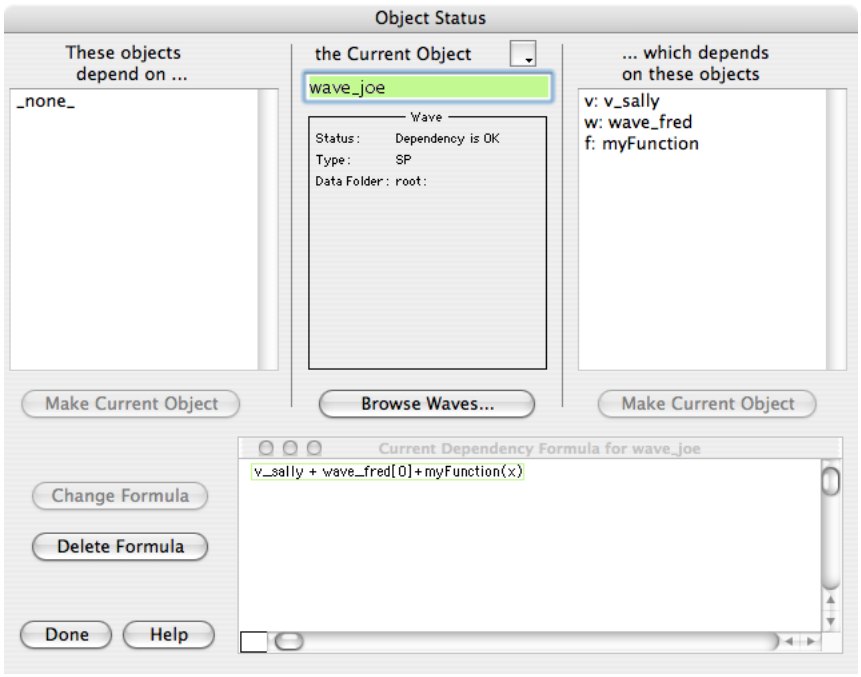


In this example, the current object is the global numeric variable v_sally. The only object directly dependent on v_sally is the wave wave_joe. V_sally directly depends on waves wave_fred and wave_sue, global numeric variable K2, and user-defined function anotherFunction. These dependencies exist because of the dependency assignment:

```
v_sally := anotherFunction(wave_fred[1]) + wave_sue[0] + K2
```

which can be changed or deleted with the dialog.

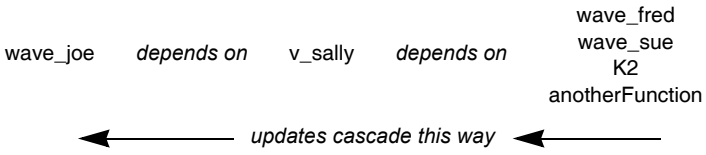
Wave_joe depends on v_sally for reasons that will become apparent only when wave_joe is made the current object:



The dependency was created by the dependency assignment:

```
wave_joe := v_sally + wave_fred[0]+myFunction(x)
```

Combining the dependency of wave_joe on v_sally with what v_sally depends on, you can see that wave_joe also *indirectly* depends on wave_sue, wave_fred, K2, and anotherFunction:



If you change K2, the dependencies will cascade so that v_sally and then wave_joe are updated. We call objects that wave_joe depends on directly or indirectly its "antecedents" (literally, "those that go before").

Deleting a Dependency

A dependency is deleted when the dependent object is assigned a value using the = operator:

```
recalculateThis := dependsOnThis    // creates dependency
recalculateThis = 0                 // deletes the dependency
```

This method of deleting a dependency does not work in user-defined functions. You must use the SetFormula operation.

For example:

```
Execute "recalculateThis = 0"
```

will delete the dependency even in a user-defined function.

You can also delete this dependency using the SetFormula operation.

```
SetFormula recalculateThis, ""
```

Wave dependencies are also deleted by operations that overwrite the values of their wave parameters. Some of these operations are:

```
FFT       Convolve        Correlate    Smooth    GraphWaveEdit
Hanning   Differentiate   Integrate    UnWrap
```

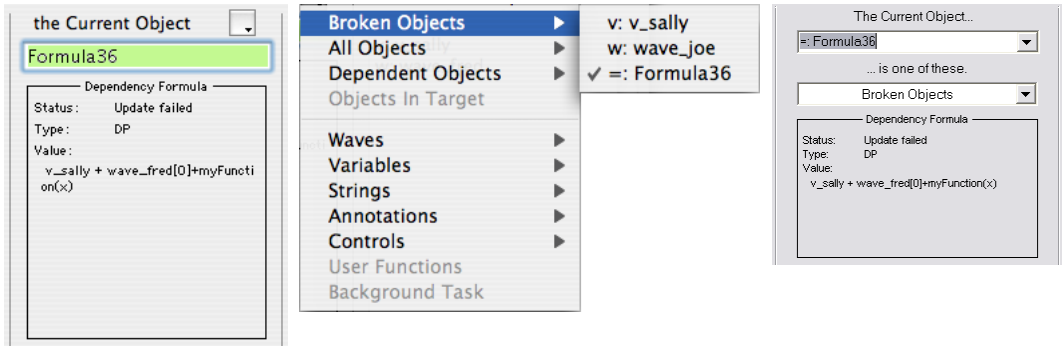
Dependencies can also be deleted via the Object Status dialog in the Misc menu.

Broken Dependent Objects

Igor compiles the text of a dependency formula to low-level code and stores both the original text and the low-level code with the dependent object. At various times, Igor may need to recompile the dependency formula text. At that time it is possible to get a compilation error if one of the objects in the formula has been renamed or deleted, or if the formula contains a syntax error.

When this happens, the dependent object will no longer update but will retain its last value. We call such an object "broken". If you suspect this kind of problem has happened, invoke the Object Status dialog using the Misc menu.

Any such broken objects will show up in the Broken Objects submenu of the Current Object pop-up menu:



When Dependencies are Updated

Dependency updates take place at the same time that graphs are updated. This happens after each line in a macro is executed, or when DoUpdate is called from a macro or user function, or continuously if a macro or function is not running.

Dependency formulas used as input to the SetBackground and ValDisplay operations, and in some other contexts, can alternately be specified as a literal string of characters using the following syntax:

```
#"text_of_the_dependency_expression"
```

Note that what follows the # char must be a literal string — not a string expression.

This will set the dependency formula *without* compiling it or checking it for validity. It is mainly for use internally but if you find yourself in a situation where you need to set the dependency formula of an object to something that is not currently valid but will be in the future then feel free to use this alternate method.

Programming with Dependencies

You cannot use := to create dependencies in user-defined functions. Instead you must use the **SetFormula** operation (see page V-559).

```
Function TestFunc()
   Variable/G varNum=-666
   Make wave1
   SetFormula wave1, "varNum"    // Equivalent to wave1 := varNum
End
```

Using Operations in Dependency Formulas

The dependency formula must be a single expression — and you can not use operations, such as FFT's, or other command type operations. However, you can invoke user-defined functions which *in turn* invoke operations:

```
Function MakeDependencyUsingOperation()
   Make/O/N=128 data = p         // A ramp from 0 to 127
   Variable/G power

   SetFormula power, "RMS(data)" // Dependency on function and wave
   Print power

   data = p * 2                  // Changes something power depends
   DoUpdate                      // Make Igor recalc formulae
   Print power
EndMacro

Function RMS(w)
   Wave w

   WaveStats/Q w           // An operation! One output is V_rms
   return V_rms
End
```

When `MakeDependencyUsingOperation` is executed, it prints the following in the history area:

```
•MakeDependencyUsingOperation()
   73.4677
   146.935
```

Dependency Caveats

The extensive use of dependencies can create a confusing tangle that can be difficult to manage. Although you can use the Object Status dialog to explore the dependency hierarchy, you can still become very confused very quickly, especially when the dependencies are highly cascaded. You should use dependencies only where they are needed. Use conventional assignments for the majority of your calculations.

There is no built-in limit to the depth of dependency cascading except that speed considerations will limit the depth to about 20. Similarly there will be a practical limit to the total number of objects with dependencies. The actual limit can not be specified in advance.

Dependency formulas are generally not recalculated when a user-defined function is running unless you explicitly call the DoUpdate operation. However, they can run at other hard to predict times (especially on Windows) and you should not make any assumptions as to the timing or the current data folder when they run.

The text of the dependency formula that is saved for a dependent object is the original literal text. The dependency formula needs to be recompiled from time to time, for example when procedures are compiled. Therefore, any objects used in the formula must persist until the formula is deleted.

We recommend that you never use $ expressions in a dependency formula.

Chapter

IV-10

Advanced Programming

This chapter contains usage notes on a number of advanced programming topics.

Regular Modules

Regular modules, or "modules" for short, provide a way to avoid name conflicts between procedure files. Regular modules are distinct from "independent modules" which are discussed in the next section.

By default, a procedure file is in the built-in ProcGlobal module. A procedure file that does not contain a #pragma ModuleName statement (or a #pragma IndependentModule statement - discussed below) is in ProcGlobal. Neither #pragma ModuleName nor #pragma IndependentModule are allowed in the built-in procedure window which is always in ProcGlobal.

When you execute a function from the command line or use the **Execute** operation, you are operating in the ProcGlobal context.

Functions in ProcGlobal are either public, or, if they are declared using the static keyword, private. For example:

```
// In a procedure file with no #pragma ModuleName or #pragma IndependentModule

static Function Test()           // Private to its procedure file
   Print "Test in ProcGlobal"
End

Function TestInProcGlobal()      // Public
   Print "TestInProcGlobal in ProcGlobal"
End
```

Because it is declared static, the Test function is private to the procedure file containing it. Each procedure file can have its own static Test function without causing a name conflict. The TestInProcGlobal function is public so there can be only one public function with this name.

In this example the static Test function is accessible only from the procedure file in which it is defined. Sometimes you have a need to avoid name conflicts but still want to be able to call functions from other procedure files, from control action procedures or from the command line. This is where a regular module is useful.

You specify that a procedure file is in a different module (other than ProcGlobal) using the ModuleName pragma. For example:

```
#pragma ModuleName = ModuleA     // The following procedures are in ModuleA

static Function Test()           // Semi-private
   Print "Test in ModuleA"
End

Function TestModuleA()           // Public
   Print "Test in ModuleA"
End
```

Because it is declared static, this Test function will not cause name conflicts with other modules, including the ProcGlobal module. In this sense, it is private. But because it is in a named regular module (ModuleA), it can be called from other procedure files using a qualified name:

```
ModuleA#Test()                   // Call Test from ModuleA
```

This qualified name syntax overrides the static nature of Test and tells Igor that you want to execute the Test function defined in ModuleA. The only way to access a static function from another procedure file is to put it in a regular module and use a qualified name.

If you are writing a non-trivial set of procedures, it is a good idea to use a module and to declare your functions static, especially if other people will be using your code. This prevents name conflicts with other procedures that you or other programmers write. Make sure to choose a distinctive module name.

Regular Modules in Action Procedures and Hook Functions

Control action procedures and hook functions are called by Igor at certain times. They are executed in the ProcGlobal context. This means that a static function can not be used as an action procedure or a hook function without using a qualified name. For example:

```
// In a procedure file with no #pragma ModuleName or #pragma IndependentModule

static Function ButtonProc(ba) : ButtonControl
   STRUCT WMButtonAction &ba

   switch (ba.eventCode)
      case 2: // mouse up
         Print "Running ProcGlobal#ButtonProc"
         break
   endswitch

   return 0
End

Function CreatePanel()
   NewPanel /W=(375,148,677,228)
   // This will not work because ButtonProc is private to the procedure file
   Button button0,pos={106,23},size={98,20},title="Click Me"
   Button button0,proc=ButtonProc
End
```

When you click the Click Me button, Igor tries to run the ButtonProc action procedure. However, because it is static, it is not accessible from outside the procedure file so Igor displays an error.

There are two possible solutions for this problem:

1. Make ButtonProc global by removing the static keyword

2. Use a regular module

If you make ButtonProc global, you run the risk of a name conflict with some other programmer's ButtonProc function. You can prevent this by changing ButtonProc to a very distinctive name, like AcmeDataAcqButtonProc, but this becomes tedious.

Here is the solution using a module:

```
#pragma ModuleName = RegularModuleA

static Function ButtonProc(ba) : ButtonControl
   STRUCT WMButtonAction &ba

   switch (ba.eventCode)
      case 2: // mouse up
         Print "Running RegularModuleA#ButtonProc"
         break
   endswitch

   return 0
End

static Function CreatePanel()
   NewPanel /W=(375,148,677,228)
```

```
   Button button0,pos={106,23},size={98,20},title="Click Me"
   Button button0,proc=RegularModuleA#ButtonProc
End
```

RegularModuleA is the name we have chosen for the regular module for demonstration purposes. You should choose a more descriptive module name.

The use of a qualified name, RegularModuleA#ButtonProc, allows Igor to find and execute the static ButtonProc function in the RegularModuleA module even though ButtonProc is running in the ProcGlobal context.

To protect the CreatePanel function from name conflicts we also made it static. To create the panel, execute:

```
RegularModuleA#CreatePanel()
```

Regular Modules and User-Defined Menus

Menu item execution text also runs in the ProcGlobal context. If you want to call a routine in a regular module you must use a qualified name.

Continuing the example from the preceding section, here is how you would write a menu definition:

```
#pragma ModuleName = RegularModuleA

Menu "Macros"
   "Create Panel", RegularModuleA#CreatePanel()
End
```

See also **Independent Modules** below, **Controls and Control Panels** on page III-357, **User-Defined Hook Functions** on page IV-251 and **User-Defined Menus** on page IV-105.

Independent Modules

An independent module is a set of procedure files that are compiled separately from all other procedures. Because it is compiled separately, an independent module can run when other procedures are in an uncompiled state because the user is editing them or because an error occurred in the last compile. This allows the independent module's control panels and menus to continue to work regardless of user programming errors.

Creating an independent module adds complications and requires a solid understanding of Igor programming. You should use an independent module if it is important that your procedures be runnable at all times. For example, if you have created a data acquisition package that must run regardless of what the user is doing, that would be a good candidate for an independent module.

A file is designated as being part of an independent module using the IndependentModule pragma:

```
#pragma IndependentModule = imName
```

Make sure to use a distinctive name for your independent module.

The IndependentModule pragma is not allowed in the built-in procedure window which is always in the ProcGlobal module.

An independent module creates an independent namespace. Function names in an independent module do not conflict with the same names used in other modules. To call an independent module function from another module, including the default ProcGlobal module, the function must be public (non-static) and you must use a qualified name as illustrated in the next section.

Independent Modules - A Simple Example

Here is a simple example using an independent module. This code must be in its own procedure file and not in the built-in procedure file:

```
#pragma IndependentModule = IndependentModuleA

static Function Test()              // static means private to file
   Print "Test in IndependentModuleA"
End

// This must be non-static to call from command line (ProcGlobal context)
Function CallTestInIndependentModuleA()
   Test()
End
```

From the command line (the ProcGlobal context):

```
CallTestInIndependentModuleA()                      // Error

IndependentModuleA#CallTestInIndependentModuleA()   // OK

IndependentModuleA#Test()                           // Error
```

The first command does not work because the functions in the independent module are accessible only using a qualified name. The second command does work because it uses a qualified name and because the function is public (non-static). The third command does not work because the function is private (static) and therefore is accessible only from the file in which it is defined. A static function in an independent module is not accessible from outside the procedure file in which it is defined unless it is in an enclosed regular module as described under **Regular Modules Within Independent Modules** on page IV-218.

SetIgorOption IndependentModuleDev=1

By default, the debugger is disabled for independent modules. It can be enabled using:

```
SetIgorOption IndependentModuleDev=1
```

Also by default, independent module procedure windows that are read-only or write-protected are not listed in the Windows→Procedure Windows submenu unless you use `SetIgorOption IndependentModuleDev=1`.

Procedures loaded from "Igor Pro Folder/Igor Procedures" and "Igor Pro User Files/Igor Procedures" or those loaded via the #include mechanism are write-protected and therefore are not visible until you execute `SetIgorOption IndependentModuleDev=1`.

When `SetIgorOption IndependentModuleDev=1` is in effect, the Windows→Procedure Windows submenu shows all procedure windows, and those that belong to an independent module are listed with the independent module name in brackets:

This syntax is used in the **WinList**, **FunctionList**, **DisplayProcedure**, and **ProcedureText** functions and operations.

To get the user experience, as opposed to the programmer experience, return to normal operation by executing:

```
SetIgorOption IndependentModuleDev=0
```

Independent Module Development Tips

Development of an independent module may be easier if it is first done as for normal code. Add the module declaration

```
#pragma IndependentModule = moduleName
```

only after the code has been fully debugged and is working properly.

A procedure file that is designed to be #included should ideally work inside or outside of an independent module. Read the sections on independent modules below to learn what the issues are.

When programming an independent module, you will usually want to execute:

```
SetIgorOption IndependentModuleDev=1
```

Independent Modules and #include

If you #include a procedure file from an independent module, Igor copies the #included file into memory and makes it part of the independent module by inserting a #pragma IndependentModule statement at the start of the copy. If the same file is included several times, there will be several copies, each with a different independent module name.

Warning: Do not edit the procedure windows created by #including into an independent module because they are temporary and your changes will not be saved. You would not want to save them anyway because Igor has modified them.

Warning: Do not #include files that already contain a #pragma IndependentModule statement unless the independent module name is the same.

Limitations of Independent Modules

Independent modules are not for every-day programming and are somewhat more difficult to create than normal modules because of the following limitations:

1. Macros and Procs are not supported.
2. Button and control dialogs do not list functions in an independent module.
3. Functions in an independent module can not call functions in other modules except through the Execute operation.
4. The IndependentModule pragma requires Igor Pro 6 or later.

Independent Modules in Action Procedures and Hook Functions

Normally you must use a qualified name to invoke a function defined in an independent module from the ProcGlobal context. Control action procedures and hook functions execute in the ProcGlobal context. But, as a convenience and to make #include files more useful, Igor eliminates this requirement when you create controls and specify hook functions from a user-defined function.

When you execute an operation that creates a control or specifies a hook function while running in an independent module, Igor examines the specified control action function name or hook function name. If the named function is defined in the same independent module, Igor automatically inserts the independent module name. This means you can write something like:

```
#pragma IndependentModule = IndependentModuleA
Button b0, proc=ButtonProc
SetWindow hook(Hook1)=HookFunc
```

You don't have to write:

```
#pragma IndependentModule = IndependentModuleA
Button b0, proc=IndependentModuleA#ButtonProc
SetWindow hook(Hook1)=IndependentModuleA#HookFunc
```

Such independent module name insertion is only done when an operation called from a function defined in an independent module. It is not done if the operation is executed from the command line or via **Execute**.

The control action function or hook function must be public (non-static) (except as describe under **Regular Modules Within Independent Modules** on page IV-218).

Here is a working example:

```
#pragma IndependentModule = IndependentModuleA

Function ButtonProc(ba) : ButtonControl   // Must not be static
   STRUCT WMButtonAction &ba

   switch (ba.eventCode)
      case 2: // mouse up
         Print "Running IndependentModuleA#ButtonProc"
         break
   endswitch

   return 0
End

Function CreatePanel()
   NewPanel /W=(375,148,677,228)
   Button button0,pos={106,23},size={98,20},title="Click Me"
   Button button0,proc=ButtonProc
End
```

Independent Modules and User-Defined Menus

Independent modules can contain user-defined menus. When you choose a user-defined menu item, Igor determines if the menu item was defined in an independent module. If so, and if the menu item's execution text starts with a call to a function defined in the independent module, then Igor prepends the independent module name before executing the text. This means that the second and third menu items in the following example both call IndependentModuleA#DoAnalysis:

```
#pragma IndependentModule = IndependentModuleA

Menu "Macros"
   "Load Data File/1", Beep; LoadWave/G
   "Do Analysis/2", DoAnalysis() // Igor automatically prepends IndependentModuleA#
   "Do Analysis/3", IndependentModuleA#DoAnalysis()
End

Function DoAnalysis()
   Print "DoAnalysis in IndependentModuleA"
End
```

This behavior on Igor's part makes it possible to #include a procedure file that creates menu items into an independent module and have the menu items work. However, in many cases you will not want a #included file's menu items to appear. You can suppress them using `menus=0` option in the #include statement. See **Turning the Included File's Menus Off** on page IV-146.

Note: If a procedure file with menu definitions is included into multiple independent modules, the menus are repeatedly defined (see **Independent Modules and #include** on page IV-216). Judicious use of the menus=0 option in the #include statements helps prevent this. See **Turning the Included File's Menus Off** on page IV-146.

When the execution text doesn't start with a user-defined function name, as for the first menu item in this example, Igor executes the text without altering it.

Independent Modules and Popup Menus

In an independent module, implementing a popup menu whose items are determined by a function call at click time requires special care. For example, outside of an independent module, this works:

```
Function/S myPopMenuList()
   return "item 1;item2;"
End
...
PopupMenu pop0 value=#"myPopMenuList()"      // Note the quotation marks
```

But inside an independent module you need this:

```
#pragma IndependentModule=myIM
Function/S myPopMenuList()
   return "item 1;item2;"
End
...
String cmd= GetIndependentModuleName()+"#myPopMenuList()"
PopupMenu pop0 value=#cmd                    // No enclosing quotation marks
```

GetIndependentModuleName returns the name of the independent module to which the currently-running function belongs or "ProcGlobal" if the currently-running function is not part of an independent module.

You could change the command string to:

```
PopupMenu pop0 value=#"myIM##myPopMenuList()"
```

but using GetIndependentModuleName allows you to disable the IndependentModule pragma by commenting it out and have the code still work which can be useful during development. With the pragma commented out you are running in ProcGlobal context and GetIndependentModuleName returns "ProcGlobal".

When the user clicks the popup menu, Igor generates the menu items by evaluating the text specified by the PopupMenu value keyword as an Igor expression. The expression (`"myIM#myPopMenuList()"` in this case) is evaluated in the ProcGlobal context. In order for Igor to find the function in the independent module, it must be public (non-static) (except as describe under **Regular Modules Within Independent Modules** on page IV-218) and you must use a qualified name.

Note that `#cmd` is not the same as `#"cmd"`. The `#cmd` form was introduced with Igor Pro 6. The string variable cmd is evaluated when PopupMenu runs which occurs in the context of the independent module. The contents of cmd (`"myIM#myPopMenuList()"` in this case) are stored in the popup menu's internal data structure. When the popup menu is clicked, Igor evaluates the stored text as an Igor expression. This causes the function `myIM#myPopMenuList` to run.

With the older `#"cmd"` syntax, the stored text is evaluated only when the popup menu is clicked, not when the PopupMenu operation runs, and this evaluation occurs in the ProcGlobal context. It is too late to capture the independent module in which the text should be evaluated.

Regular Modules Within Independent Modules

It is usually not necessary but you can create a regular module within an independent module. For example:

```
#pragma IndependentModule = IndependentModuleA
#pragma ModuleName = RegularModuleA
Function Test()
   Print "Test in RegularModuleA within IndependentModuleA"
End
```

Here RegularModuleA is a regular module within IndependentModuleA.

To call the function Test from outside of the independent module you must qualify the call like this:

```
IndependentModuleA#RegularModuleA#Test()
```

This illustrates that the independent module establishes its own namespace (IndependentModuleA) which can host one level of sub-namespace (RegularModuleA). By contrast, a regular module creates a namespace within the global namespace (called ProcGlobal) and can not host additional sub-namespaces.

This nesting of modules is useful to prevent name conflicts in a large independent module project comprising multiple procedure files. Otherwise it is not necessary.

Because all procedure files in a given independent module are compiled separately from all other files, function names never conflict with those outside the group and there is little or no need to use the static designation on functions in an independent module. However, if need be, you can call static functions in a regular module inside an independent module from outside the independent module using a triple-qualified name:

IndependentModuleName#*RegularModuleName*#*FunctionName*()

Calling Routines From Other Modules

Code in an independent module can not directly call routines in other modules and usually should not need to. If you must call a routine from another module, you can do it using the **Execute** operation. You must use a qualified name. For example:

```
Execute "ProcGlobal#foo()"
```

To call a function in a regular module, you must prepend ProcGlobal and the regular module name to the function name:

```
Execute "ProcGlobal#MyRegularModule#foo()"
```

Calling a nonstatic function in a different independent modules requires prepending just the other independent module name:

```
Execute "OtherIndependentModule#bar()"
```

Calling static functions in other independent modules requires prepending the independent module name and a regular module name:

```
Execute "OtherIndependentModule#RegularModuleName#staticbar()"
```

Using Execute Within an Independent Module

If you need to call a function in the current independent module using Execute, you can compose the name using the **GetIndependentModuleName** function. For example, outside of an independent module the commands would be:

```
String cmd = "WS_UpdateWaveSelectorWidget(\"Panel0\", \"selectorWidgetName\")"
Execute cmd
```

But inside an independent module the commands are:

```
#pragma IndependentModule=myIM
String cmd="WS_UpdateWaveSelectorWidget(\"Panel0\", \"selectorWidgetName\")"
cmd = GetIndependentModuleName() + "#" + cmd    // Make qualified name
Execute cmd
```

You could change the command string to:

```
cmd = "myIM#" + cmd
```

but using GetIndependentModuleName allows you to disable the IndependentModule pragma by commenting it out and have the code still work which can be useful during development. With the pragma commented out you are running in ProcGlobal context and GetIndependentModuleName returns "ProcGlobal".

Independent Modules and Dependencies

GetIndependentModuleName is also useful for defining dependencies using functions in the current independent module. Dependencies are evaluated in the global procedure context (ProcGlobal). In order for dependencies to evaluate correctly, the dependency must use GetIndependentModuleName to create a formula to pass to the **SetFormula** operation. For example, outside of an independent module, this works:

```
String formula = "foo(root:wave0)"
SetFormula root:aVariable $formula
```

But inside an independent module you need this:

```
#pragma IndependentModule=myIM
String formula = GetIndependentModuleName() + "#foo(root:wave0)"
SetFormula root:aVariable $formula
```

Independent Modules and Pictures

To allow DrawPICT to use a picture in the picture gallery, you must prepend GalleryGlobal# to the picture name:

```
DrawPICT 0,0,1,1,GalleryGlobal#PICT_0
```

Without GalleryGlobal, only Proc Pictures can be used defined in an independent module. GalleryGlobal is available in Igor Pro 6.1 or later.

Making Regular Procedures Independent-Module-Compatible

You may want to make an existing set of procedures into an independent module. Alternatively, you may want to make an existing procedure independent-module-compatible so that it can be #included into an independent module. This section outlines the necessary steps.

1. If you are creating an independent module, add the IndependentModule pragma:

   ```
   #pragma IndependentModule=<NameOfIndependentModule>
   ```

2. Change any Macro or Proc procedures to functions.
3. Make Execute commands suitable for running in the ProcGlobal context or in an independent module using GetIndependentModuleName. See **Using Execute Within an Independent Module** on page IV-219.
4. Make PopupMenu controls that call a string function to populate the menu work in the ProcGlobal context or in an independent module using GetIndependentModuleName. See **Independent Modules and Popup Menus** on page IV-218.
5. Make any dependencies work in the ProcGlobal context or in an independent module using GetIndependentModuleName. See **Independent Modules and Dependencies** on page IV-220.

See also **Regular Modules** on page IV-212, **Controls and Control Panels** on page III-357, **User-Defined Hook Functions** on page IV-251, **User-Defined Menus** on page IV-105, and **GetIndependentModuleName** on page V-214.

Sound

Two operations are provided for playing of sound through the computer speakers:

- PlaySound
- PlaySnd (*Macintosh*)

The **PlaySound** operation takes the sound data from a wave.

The obsolete **PlaySnd** operation gets its data from a Macintosh 'snd ' resource stored in a file.

A number of sound input operations are provided: **SoundInStatus** (page V-583) , **SoundInSet** (page V-583) , **SoundInRecord** (page V-582) , **SoundInStartChart** (page V-583) and **SoundInStopChart** (page V-584) . Several example experiments that use these routines can be found in your Igor Pro Folder in the Examples folder.

The SndLoadSaveWave XOP loads and saves various sound file formats into and from waves. It also adds a Load Sound File menu item to the Load Waves submenu and a Save Sound File menu item to the Save Waves submenu as well as several command line operations. See the SndLoadSaveWave help file in the More Extensions:File Loaders folder for details.

Movies

If you have Apple's QuickTime installed on your computer, you can create or play movies. You can even add a soundtrack to your movies.

The process of creating a movie is very simple. You use the following operations to create a new movie file, add video frames and audio, close the file and then play the movie. Refer to Chapter V-1, **Igor Reference**, for details on the operations.

- NewMovie
- AddMovieFrame
- AddMovieAudio
- CloseMovie
- PlayMovie
- PlayMovieAction

The NewMovie operation creates a movie file and also defines the movie frame rate and optional audio track specifications. Before calling NewMovie, you need to prepare the first frame of your movie as the target graph. If you will be using audio you also need to prepare a sound wave. The sound wave can be of any time duration but usually will either be the entire length of the movie or will be the length of one video frame. If you will be placing your movie on a Web page or transferring it across platforms, be sure to use the /L flag with NewMovie to create a "flattened" movie.

After creating the file and the first video frame and optional audio, you use AddMovieFrame to add as many video frames as you wish. You may also add more audio using the AddMovieAudio operation. Finally you use the CloseMovie and PlayMovie operations.

When you write a procedure to generate a movie, you need to call the DoUpdate operation after all modifications to the graph and before calling AddMovieFrame. This allows Igor to process any changes you have made to the graph.

You may use the PlayMovie operation to play any QuickTime movie — not just those you create with Igor.

You can extract individual frames from a movie and can control movie playback using PlayMovieAction.

For examples of programming with movies, see the Examples:Movies & Audio folder.

Timing

There are two methods you can use when you want to measure elapsed time:

- The ticks counter using the ticks function
- The microsecond timer using StartMSTimer() and StopMSTimer()

Ticks Counter

You can easily measure elapsed time with a precision of 1/60th of a second using the ticks function. It returns the tick count which starts at zero when you first start your computer and is incremented at a rate of approximately 60 Hz rate from then on.

Here is an example of typical use:

```
…
Variable t0
…
t0= ticks
<operations you wish to time>
printf "Elapsed time was %g seconds\r",(ticks-t0)/60
…
```

Microsecond Timer

You can measure elapsed time to microsecond accuracy for durations up to 35 minutes using the microsecond timer. See the **startMSTimer** function (page V-594) for details and an example.

Packages

A package is a set of files that adds a significant set of functionality to Igor. Packages consist of procedure files and may also include XOPs, help files and other supporting files.

A package usually adds one or more items to Igor's menus that allow the user to interactively load the package, access its functionality, and unload the package.

A package typically provides some level of user-interface, such as a menu item and a control panel, for accessing the added functionality. It may store settings in experiments or in global preferences.

A package is typically loaded into memory and unloaded at the user's request.

Igor comes pre-configured with numerous WaveMetrics packages accessed through the Data→Packages, Analysis→Packages, Misc→Packages, Windows→New→Packages and Graph→Packages submenus as well as others. Take a peek at these submenus to see what packages are supplied with Igor.

Menu items for WaveMetrics packages are added to Igor's menus by the WMMenus.ipf procedure file which is shipped in the Igor Procedures folder. (WMMenus.ipf is hidden unless you enable independent module development. See **Independent Modules** on page IV-214.)

A large package of 3D-plotting support routines is accessible through the Gizmo menu that appears in the main menu bar when a Gizmo 3D window is active.

The following section explains how to add a package to Igor.

Creating a Package

This section shows how to create a package through a simple example. The package is called "Sample Package". It adds a Load Sample Package item to the Macros menu. When the user chooses Load Sample Package, the package's procedure file is loaded. This adds two additional items to the Macros menu: Hello From Sample Package and Unload Sample Package.

The package consists of two procedure files stored in a folder in the Igor Pro User Files folder. If you are not familiar with Igor Pro User Files, take a short detour and read **Special Folders** on page II-44 and **Igor Pro User Files** on page II-46.

The sample package is installed as follows:

```
Igor Pro 6 User Files
   Sample Package
```

```
      Sample Package Loader.ipf
      Sample Package.ipf
   Igor Procedures
      Alias or shortcut pointing to the "Sample Package Loader.ipf" file
   User Procedures
      Alias or shortcut pointing to the "Sample Package" folder
```

Putting the alias/shortcut for the "Sample Package Loader.ipf" in Igor Procedures causes Igor to load that file at launch time. The file adds the "Load Sample Package" item to the Macros menu. (See **Global Procedure Files** on page III-345 for details.)

Putting the alias/shortcut for the "Sample Package" folder in User Procedures causes Igor to search that folder when a #include is invoked. (See **Shared Procedure Files** on page III-345 for details.)

A real package might include other procedure files and a help file in the "Sample Package" folder.

To try this out yourself, follow these steps:

1. Create the "Sample Package" folder in your Igor Pro User Files folder.

   You can locate your Igor Pro User Files folder using the Help menu.

2. Create a new procedure file named "Sample Package Loader.ipf" in the "Sample Package" folder and enter the following contents in the file:

   ```
   Menu "Macros"
      "Load Sample Package", /Q, LoadSamplePackage()
   End

   Function LoadSamplePackage()
      Execute/P/Q/Z "INSERTINCLUDE \"Sample Package\""
      Execute/P/Q/Z "COMPILEPROCEDURES "// Note the space before final quote
   End
   ```

   Save the procedure file.

3. Create a new procedure file named "Sample Package.ipf" in the "Sample Package" folder and enter the following contents in the file:

   ```
   Menu "Macros"
      "Hello From Sample Package", HelloFromSamplePackage()
      "Unload Sample Package", UnloadSamplePackage()
   End

   Function HelloFromSamplePackage()
      DoAlert /T="Sample Package Wants to Say" 0, "Hello!"
   End

   Function UnloadSamplePackage()
      Execute /P /Q /Z "DELETEINCLUDE \"Sample Package\""
      Execute /P /Q /Z "COMPILEPROCEDURES "// Note the space before final quote
   End
   ```

   Save the procedure file.

4. In the desktop, make an alias or shortcut for "Sample Package Loader.ipf" file and put it in the Igor Procedures folder in the Igor Pro User Files folder.

   This causes Igor to load the "Sample Package Loader.ipf" file at launch time. This is how the Load Sample Package menu item gets into the Macros menu.

5. In the desktop, make an alias or shortcut for the "Sample Package" folder and put it in the User Procedures folder in the Igor Pro User Files folder.

   This causes Igor to search the "Sample Package" folder when a #include is invoked. This allows Igor to find the "Sample Package.ipf" file when it is #included.

6. Quit and restart Igor so that Igor will load the "Sample Package Loader.ipf" file.

   If you prefer you can just manually make sure that "Sample Package Loader.ipf" is open and "Sample Package.ipf" is closed. This simulates the state of affairs after restarting Igor.

7. Choose Windows→Procedure Windows and verify that Igor has loaded the "Sample Package Loader.ipf" file.

8. Click the Macros menu and verify that the "Load Sample Package" item is present.

9. Choose Macros→Load Sample Package.

   The LoadSamplePackage function runs, adds a #include statement to the built-in procedure window, and forces procedures to be recompiled. This cause Igor to load the "Sample Package.ipf" procedure file which contains the bulk of the package's procedures and adds items to the Macros menu.

10. Click the Macros menu and notice that the "Hello From Sample Package" and "Unload Sample Package" items have been added.

11. Choose Macros→Hello From Sample Package.

    The package displays an alert. A real package would do something more exciting.

12. Choose Macros→Unload Sample Package.

    The UnloadSamplePackage function runs, removes the #include statement from the built-in procedure window, and forces procedures to be recompiled. This cause Igor to unload the "Sample Package.ipf" procedure.

13. Click the Macros menu and notice that the "Hello From Sample Package" and "Unload Sample Package" items have been removed.

Most real packages do not create Unload menu items. Instead they provide an Unload Package button in a control panel or automatically unload when a control panel is closed. Or they might not support unloading.

A real package typically does not include "Package" in its name or in its menu items.

Lightweight Packages

A lightweight package is one that consists of at most a few procedure files and does not create clutter in the current experiment unless it is actually used.

If your package is lightweight you might prefer to dispense with loading and unload it and just keep it loaded all the time. To do this you would organize your files like this:

```
Igor Pro 6 User Files
   Your Package
      Your Package Part 1.ipf
      Your Package Part 2.ipf
   Igor Procedures
      Alias or shortcut pointing to the "Your Package" folder
```

Here both of your package procedure files are global, meaning that Igor loads them at launch time and never unloads them. You do not need procedures for loading and unloading your package.

If you have an ultra-light package, consisting of just a single procedure file, you can dispense with the "Your Package" folder and put the procedure file directly in the Igor Procedures folder.

Managing Package Data

When you create a package of procedures, you need some place to store private data used by the package to keep track of its state. It's important to keep this data separate from the user's data to avoid clutter and to protect your data from inadvertent changes.

Private data should be stored in a data folder named after the package inside a generic data folder named Packages. For example, if your package is named My Package you would store your private data in `root:Packages:My Package`.

There are two general types of private data that you might need to store: overall package data and per-instance data. For example, for a data acquisition package, you may need to store data describing the state of the acquisition as a whole and other data on a per-channel basis.

Creating Globals in a Specific Data Folder

Here is an example of creating the overall private data for a package:

```
Function CreatePackageData()
   // Create the package data folder if it does not already exist
   NewDataFolder/O root:Packages
   NewDataFolder/O root:Packages:'My Package'

   // Create a data folder reference variable
   DFREF dfr = root:Packages:'My Package'

   Make dfr:wave1
   Variable/G dfr:gVar1
   String/G dfr:gStr1

   WAVE wave1 = dfr:wave1
   NVAR var1 = dfr:gVar1
   SVAR str1 = dfr:gStr1

   wave1= x^2
   var1= 1
   str1= "hello"
End
```

Creating Globals In A Per-Instance Data Folder

In the previous example, the absolute path to the data folder was known at the time the functions were written. This is the case when there is only one such data folder.

In other cases, there may be many data folders with different names containing the same waves and variables. Each data folder represents an instance of a particular object.

For example, if you write a data acquisition package that deals with multiple data acquisition boards, you will probably want to store a set of data for each board with each set in its own data folder. Here is an example of a function that creates globals for a particular instance of the data acquisition board:

```
Function CreateBoardGlobals(boardName)
   String boardName     // e.g., "DAQ 1", DAQ 2", . . .

   NewDataFolder/O root:Packages
   NewDataFolder/O root:Packages:'My Package'
   NewDataFolder/O root:Packages:'My Package':$boardName

   // Create a data folder reference variable
   DFREF dfr = root:Packages:'My Package':$boardName

   Variable/G dfr:gBoardGain = 1
   String/G dfr:gBoardStatusStr = "OK"
End
```

Here the $ operator is used to convert the boardName string into a data folder name used for the per-instance data folder.

Saving Package Preferences

If you are writing a sophisticated package of Igor procedures you may want to save preferences for your package. For example, if your package creates a control panel that can be opened in any experiment, you may want it to remember its position on screen between invocations. Or you may want to remember various settings in the panel from one invocation to the next.

Such "state" information can be stored either separately in each experiment or it can be stored just once for all experiments in preferences. These two approaches both have their place, depending on circumstances. But if your package creates a control panel that is intended to be present at all times and used in any experiment then the preferences approach is usually the best fit.

If you choose the preferences approach, you will store your package preference file in a directory created for your package. Your package directory will be in the Packages directory, inside Igor's own preferences directory.

The location of Igor's Packages directory depends on the operating system and the particular user's configuration. You can find where it is on a particular system by executing:

```
Print SpecialDirPath("Packages", 0, 0, 0)
```

Important:You must choose a very distinctive name for your package because that is the only thing that prevents some other package from overwriting yours. All package names starting with "WM" are reserved for WaveMetrics.

A package name is limited to 31 characters and must be a legal name for a directory on disk.

There are two ways to store package preference data:

In a special-format binary file stored in your package directory.

As Igor waves and variables in an Igor experiment file stored in your package directory.

The special-format binary file approach is relatively simple to implement but is not suitable for storing very large amounts of data. In most cases it is not necessary to store very large amounts of data so this is the way to go.

The use of the Igor experiment file supports storing a large amount of preference data but creates a problem of synchronizing your preference data stored in memory and your preference data stored on disk. It also leads to a proliferation of preference data stored in various experiments. You should avoid using this technique if possible.

Saving Package Preferences in a Special-Format Binary File

This approach supports preference data consisting of a collection of numeric and string data. You define a structure encapsulating your package preference data. You use the **LoadPackagePreferences** operation (page V-346) to load your data from disk and the **SavePackagePreferences** operation (page V-542) to save it to disk.

The LoadPackagePreferences and SavePackagePreferences were added in Igor Pro 5.04B07 so if you use this technique, your package will require that version or later.

SavePackagePreferences stores data from your package's preferences data structure in memory. LoadPackagePreferences returns that data to you via the same structure.

SavePackagePreferences also creates a directory for your package preferences and stores your data in a file in that directory. Your package directory is located in the Packages directory in Igor's preferences directory. The job of storing the preferences data in the file is handled transparently which, by default, automatically flushes your data to the file when the current experiment is saved or closed and when Igor quits.

You would call LoadPackagePreferences every time you need to access your package preference data and SavePackagePreferences every time you want to change your package preference data. You pass to these operations an instance of a structure that you define.

Here are example functions from the Package Preferences Demo experiment that use the LoadPackagePreferences and SavePackagePreferences operations to implement preferences for a particular package:

```
// NOTE: The package name you choose must be distinctive!
static StrConstant kPackageName = "Acme Data Acquisition"
static StrConstant kPrefsFileName = "PanelPreferences.bin"
static Constant kPrefsVersion = 100
static Constant kPrefsRecordID = 0

Structure AcmeDataAcqPrefs
   uint32version    // Preferences structure version number. 100 means 1.00.
   double panelCoords[4]      // left, top, right, bottom
   uchar phaseLock
   uchar triggerMode
   double ampGain
   uint32 reserved[100]       // Reserved for future use
EndStructure

// DefaultPackagePrefsStruct(prefs)
// Sets prefs structure to default values.
static Function DefaultPackagePrefsStruct(prefs)
   STRUCT AcmeDataAcqPrefs &prefs

   prefs.version = kPrefsVersion

   prefs.panelCoords[0] = 5          // Left
   prefs.panelCoords[1] = 40         // Top
   prefs.panelCoords[2] = 5+190      // Right
   prefs.panelCoords[3] = 40+125     // Bottom
   prefs.phaseLock = 1
   prefs.triggerMode = 1
   prefs.ampGain = 1.0

   Variable i
   for(i=0; i<100; i+=1)
      prefs.reserved[i] = 0
   endfor
End

// SyncPackagePrefsStruct(prefs)
// Syncs package prefs structures to match state of panel.
// Call this only if the panel exists.
static Function SyncPackagePrefsStruct(prefs)
   STRUCT AcmeDataAcqPrefs &prefs

   // Panel does exists. Set prefs to match panel settings.
   prefs.version = kPrefsVersion

   GetWindow AcmeDataAcqPanel wsize
   // NewPanel uses device coordinates. We therefore need to scale from
   // points (returned by GetWindow) to device units for windows created
   // by NewPanel.
   Variable scale = ScreenResolution / 72
   prefs.panelCoords[0] = V_left * scale
   prefs.panelCoords[1] = V_top * scale
   prefs.panelCoords[2] = V_right * scale
   prefs.panelCoords[3] = V_bottom * scale

   ControlInfo /W=AcmeDataAcqPanel PhaseLock
   prefs.phaseLock = V_Value         // 0=unchecked; 1=checked

   ControlInfo /W=AcmeDataAcqPanel TriggerMode
   prefs.triggerMode = V_Value       // Menu item number starting from on

   ControlInfo /W=AcmeDataAcqPanel AmpGain
   prefs.ampGain = str2num(S_value)  // 1, 2, 5 or 10
End

// InitPackagePrefsStruct(prefs)
// Sets prefs structures to match state of panel or
// to default values if panel does not exist.
static Function InitPackagePrefsStruct(prefs)
   STRUCT AcmeDataAcqPrefs &prefs

   DoWindow AcmeDataAcqPanel
```

```
   if (V_flag == 0)
      // Panel does not exist. Set prefs struct to default.
      DefaultPackagePrefsStruct(prefs)
   else
      // Panel does exists. Sync prefs struct to match panel state.
      SyncPackagePrefsStruct(prefs)
   endif
End

static Function LoadPackagePrefs(prefs)
   STRUCT AcmeDataAcqPrefs &prefs

   // This loads preferences from disk if they exist on disk.
   LoadPackagePreferences kPackageName, kPrefsFileName, kPrefsRecordID, prefs

   // If error or prefs not found or not valid, initialize them.
   if (V_flag!=0 || V_bytesRead==0 || prefs.version!=kPrefsVersion)
      InitPackagePrefsStruct(prefs) // Set from panel if it exists or to default values.
      SavePackagePrefs(prefs)       // Create initial prefs record.
   endif
End

static Function SavePackagePrefs(prefs)
   STRUCT AcmeDataAcqPrefs &prefs

   SavePackagePreferences kPackageName, kPrefsFileName, kPrefsRecordID, prefs
End
```

The package preferences structure, AcmeDataAcqPrefs in this case, must not use fields of type Variable, String, WAVE, NVAR, SVAR or FUNCREF because these fields refer to data that may not exist when LoadPackagePreferences is called. It can use fields of type char, uchar, int16, uint16, int32, uint32, float and double as well as fixed-size arrays of these types and substructures with fields of these types.

Use the reserved field to add fields to the structure in a backward-compatible fashion. For example, a subsequent version of the structure might look like this:

```
Structure AcmeDataAcqPrefs
   uint32   // Preferences structure version number. 100 means 1.00.
   double panelCoords[4]      // left, top, right, bottom
   uchar phaseLock
   uchar triggerMode
   double ampGain
   uint32 triggerDelay
   uint32 reserved[99]        // Reserved for future use
EndStructure
```

Here the triggerDelay field was added and size of the reserved field was reduced to keep the overall size of the structure the same. The AcmeDataAcqLoadPackagePrefs function would also need to be changed to set the default value of the triggerDelay field.

If you need to change the structure such that its size changes or its fields are changed in an incompatible manner then you must change your structure version, which will overwrite old preferences with new preferences.

A functioning example using this technique can be found in:

"Igor Pro Folder:Examples:Programming:Package Preferences Demo.pxp"

In the example above we store just one structure in the preference file. However LoadPackagePreferences and SavePackagePreferences allow storing any number of structures of the same or different types in the preference file. You can store either multiple instances of the same structure or multiple different structures. You must assign a unique nonnegative integer as a record ID for each structure stored and pass this record ID to LoadPackagePreferences and SavePackagePreferences. You could use this feature, for example, to store a different structure for each type of control panel that your package presents. Since all data is cached in memory you should not attempt to store hundreds or thousands of structures.

In almost all cases a particular package will need just one preference file. For the rare cases where this is inconvenient, LoadPackagePreferences and SavePackagePreferences allow each package to create any

number of preference files, each with a distinct file name. All of the preference files for a particular package are stored in the same directory, the package's preferences directory. Each file can store a different set of structure. However, the code that implements this feature is not tuned to handle large numbers of files so you should not use this feature indiscriminately.

Saving Package Preferences in an Experiment File

This approach supports package preference data consisting of waves, numeric variables and string variables. It is more difficult to implement than the special-format binary file approach and is not recommended except for expert programmers and then only if the previously described approach is not suitable.

You use the SaveData operation to store your waves and variables in a packed experiment file in your package directory on disk. You can later use the LoadData operation to load the waves and variables into a new experiment.

You must create your package directory as illustrated by the SavePackagePrefs function below.

The following example functions save and load package preferences. These functions assume that the package preferences consist of all waves and variables at the top level of the package's data folder. You may need to customize these functions for your situation.

```
// SavePackagePrefs(packageName)
// Saves the top-level waves, numeric variables and string variables
// from the data folder for the named package into a file in the Igor
// preferences hierarchy on disk.
Function SavePackagePrefs(packageName)
    String packageName              // NOTE: Use a distinctive package name.

    // Get path to Packages preferences directory on disk.
    String fullPath = SpecialDirPath("Packages", 0, 0, 0)
    fullPath += packageName

    // Create a directory in the Packages directory for this package
    NewPath/O/C/Q tempPackagePrefsPath, fullPath

    fullPath += ":Preferences.pxp"

    String saveDF = GetDataFolder(1)
    SetDataFolder root:Packages:$packageName
    SaveData/O/Q fullPath           // Save the preference file
    SetDataFolder saveDF

    // Kill symbolic path but leave directory on disk.
    KillPath/Z tempPackagePrefsPath
End

// LoadPackagePrefs(packageName)
// Loads the data from the previously-saved package preference file,
// if it exist, into the package's data folder.
// Returns 0 if the preference file existed, -1 if it did not exist.
// In either case, this function creates the package's data folder if it
// does not already exist.
// LoadPackagePrefs does not affect any other data already in the
// package's data folder.
Function LoadPackagePrefs(packageName)
    String packageName              // NOTE: Use a distinctive package name.

    Variable result = -1
    String saveDF = GetDataFolder(1)

    NewDataFolder/O/S root:Packages     // Ensure root:Packages exists
    NewDataFolder/O/S $packageName      // Ensure package data folder exists

    // Find the disk directory in the Packages directory for this package
    String fullPath = SpecialDirPath("Packages", 0, 0, 0)
    fullPath += packageName
    GetFileFolderInfo/Q/Z fullPath
    if (V_Flag == 0)                    // Disk directory exists?
        fullPath += ":Preferences.pxp"
        GetFileFolderInfo/Q/Z fullPath
        if (V_Flag == 0)                // Preference file exist?
            LoadData/O/R/Q fullPath     // Load the preference file.
            result = 0
```

```
      endif
   endif

   SetDataFolder saveDF
   return result
End
```

The hard part of using the experiment file for saving package preferences is not in saving or loading the package preference data but in choosing when to save and load it so that the latest preferences are always used. There is no ideal solution to this problem but here is one strategy:

1. When package preference data is needed (e.g., you are about to create your control panel and need to know the preferred coordinates), check if it exists in memory. If not load it from disk.
2. When the user does a New Experiment or quits Igor, if package preference data exists in memory, save it to disk. This requires that you create an IgorNewExperimentHook function and an IgorQuitHook function.
3. When the user opens an experiment file, if it contains package preference data, delete it and reload from disk. This requires that you create an AfterFileOpenHook function. This is necessary because the package preference data in the just opened experiment is likely to be older than the data in the package preference file.

Creating Formatted Text

The printf, sprintf, and fprintf operations print formatted text to Igor's history area, to a string variable or to a file respectively. The wfprintf operation prints formatted text based on data in waves to a file.

All of these operations are based on the C printf function which prints the contents of a variable number of string and numeric variables based on the contents of a format string. The format string can contain literal text and conversion specifications. Conversion specifications define how a variable is to be printed.

Here is a simple example:

```
printf "The minimum is %g and the maximum is %g\r", V_min, V_max
```

In this example, the format string is `"The minimum is %g and the maximum is %g\r"` which contains some literal text along with two conversion specifications — both of which are "`%g`" — and an escape code ("`\r`") indicating "carriage-return". If we assume that the Igor variable V_min = .123 and V_max = .567, this would print the following to Igor's history area:

```
The minimum is .123 and the maximum is .567
```

We could print this output to an Igor string variable or to a file instead of to the history using the **sprintf** (see page V-590) or **fprintf** (see page V-183) operations.

Printf Operation

The syntax of the printf operation is:

```
printf format [, parameter [, parameter ]. . .]
```

where *format* is the format string containing literal text or format specifications. The number and type of parameters depends on the number and type of format specifications in the format string. The parameters, if any, can be literal numbers, numeric variables, numeric expressions, literal strings, string variables or string expressions.

The conversion specifications are very flexible and make printf a powerful tool. They can also be quite involved. The simplest specifications are:

Specification	What It Does
%g	Converts a number to text using integer, floating point or exponential notation depending on the number's magnitude.
%e	Converts a number to text using exponential notation.
%f	Converts a number to text using floating point notation.

Specification	What It Does
%d	Converts a number to text using integer notation.
%s	Converts a string to text.

Here are some examples:

```
printf "%g, %g, %g\r", PI, 6.022e23, 1.602e-19
```

prints:

```
3.14159, 6.022e+23, 1.602e-19
```

```
printf "%e, %e, %e\r", PI, 6.022e23, 1.602e-19
```

prints:

```
3.141593e+00, 6.022000e+23, 1.602000e-19
```

```
printf "%f, %f, %f\r", PI, 6.022e23, 1.602e-19
```

prints:

```
3.141593, 602200000000000027200000.000000, 0.000000
```

```
printf "%d, %d, %d\r", PI, 6.022e23, 1.602e-19
```

prints:

```
3, 2147483647, 0
```

```
printf "%s, %s\r", "Hello, world", "The time is " + Time()
```

prints:

```
Hello, world, The time is 11:43:40 AM
```

Note that the output for 6.022e23 when printed using the %d conversion specification is wrong. This is because 6.022e23 is too big a number to represent as an 32 bit integer.

If you want better control of the output format, you need to know more about conversion specifications. It gets quite involved. See the **printf** operation on page V-499.

sprintf Operation

The sprintf operation is very similar to printf except that it prints to a string variable instead of to Igor's history. The syntax of the sprintf operation is:

```
sprintf stringVariable, format [, parameter [, parameter ]. . .]
```

where *stringVariable* is the name of the string variable to print to and the remaining parameters are as for printf. sprintf is useful for generating text to use as prompts in macros, in axis labels and in annotations.

fprintf Operation

The fprintf operation is very similar to printf except that it prints to a file instead of to Igor's history. The syntax of the fprintf operation is:

```
fprintf variable, format [, parameter [, parameter ]. . .]
```

where *variable* is the name of a numeric variable containing the file reference number for the file to print to and the remaining parameters are as for printf. You get the file reference number using the Open operation, described under **Open and Close Operations** on page IV-172.

For debugging purposes, if you specify 1 for the file reference number, Igor prints to the history area instead of to a file, as if you used printf instead of fprintf.

wfprintf Operation

The wfprintf operation is very similar to printf except that it prints the contents of one to 100 waves to a file. The syntax of the wfprintf operation is:

```
wfprintf variable, format [/R=(start,end)] wavelist
```

variable is the name of a numeric variable containing the file reference number for the file to print to.

Example Using fprintf and wfprintf

Here is an example of a command sequence that creates some waves and put values into them and then writes them to an output file with column headers.

```
Make/N=25 wave1, wave2, wave3
wave1 = 100+x; wave2 = 200+x; wave3 = 300+x
Variable f1
Open f1
fprintf f1, "wave1, wave2, wave3\r"
wfprintf f1, "%g, %g, %g\r" wave1, wave2, wave3
Close f1
```

This generates a comma delimited file. To generate a tab delimited file, use:

```
fprintf f1, "wave1\twave2\twave3\r"
wfprintf f1, "%g\t%g\t%g\r" wave1, wave2, wave3
```

Since tab-delimited is the default format for wfprintf, this last command is equivalent to:

```
wfprintf f1, "" wave1, wave2, wave3
```

Client/Server Overview

Igor Pro can act as a server — accepting commands and data from a client program and returning results, as a client — sending commands and data to a server program, or as both at the same time.

For the Macintosh, see **Apple Events** on page IV-232 for server information and **AppleScript** on page IV-234 for client capabilities. Previous versions of Igor supported an Apple technology called Program-to-Program Communication (PPC). Apple's operating systems no longer support PPC so it is no longer supported in Igor.

For Windows, see **ActiveX Automation** on page IV-236. Igor can play the role of an Automation server but not an Automation client. However it is possible to generate script files that allow Igor to indirectly play the role of client.

On Windows, Igor also supports Dynamic Data Exchange (DDE) and can be a DDE server or DDE client. Because DDE is obsolescent and being phased out by Microsoft, new programming should use ActiveX automation. Igor's support for DDE is described in the Obsolete Topics help file.

Apple Events

This topic is of interest to *Macintosh* programmers. Windows users should see the section for **ActiveX Automation** on page IV-236.

This topic contains information for Igor users who wish to control Igor from other programs (e.g., AppleScript). It also contains information useful to people who are writing their own programs and wish to use Igor Pro as a compute/graphing engine.

There is also a mechanism that allows Igor to act like a controller and initiate Apple event communication with other programs. See **AppleScript** on page IV-234.

It is assumed that the reader of this material is an experienced Igor user.

Apple Event Capabilities

Igor Pro supports the following Apple events:

Event	Suite	Action
Open Application	Required	Basically a nop; don't use.
Open Document	Required	Loads an experiment.
Print Document	Required	NA; don't use.
Quit Application	Required	Quits.
Close	Core	Acts on experiment, window or PPC port.
Save	Core	Acts on experiment only.
Open	Core	NA; don't use.
Do Script	Misc	Executes commands; can return ASCII results.
Eval Expression	Misc	Same as Do Script; obsolete but included for compatibility.

Apple Events — Basic Scenario

You use the Open Document event to cause Igor to load an experiment with whatever goodies you find useful (macros, useful waves and variables or whatever). You then use the Do Script or Eval Expression events to send commands to Igor for execution and to retrieve results. To get data into Igor you write files and then send commands to Igor to load the data. To get data waves from Igor you do the reverse. To get PICT data you send commands to Igor that cause it to write a PICT file that you can read. You may then close the experiment and start over with a new one. You will not likely use the Save event.

Apple Events — Obtaining Results from Igor

To return information from Igor, you will need to embed special commands in the script you send to Igor for execution. When Igor encounters these commands, it appends results to a packet that is returned to your application after script execution ends. The special commands are variations on the standard Igor commands FBinWrite and fprintf. Both of these commands take a file reference number as a parameter. If the magic value zero is used rather than a real file reference number, then the data that would normally be written to a file is appended to the result packet.

As far as Igor is concerned, there is no difference between the Do Script and Eval Expression events. However, old applications may expect results from Eval Expression and not from Do Script.

To use waves and PICTs with Apple events, you will need to write or read the data via standard Igor files. For example, you might include

```
SavePICT/P=myPath as "a PICT file"
```

in a script that you send to Igor for execution. You could then read the file in your application.

Apple Event Details

This information is intended for programmers familiar with Apple events terminology.

Some of the following events can act on experiments or windows.

To specify an experiment, use object class cDocument (`'docu'`) and specify either formAbsolutePosition with index=1 or formName with name=*name of experiment*.

To specify a window, use object class cWindow (`'cwin'`) and either formAbsolutePosition or formName with name=*title of window*.

Event	Class	Code	Action
Open Application	`'aevt'`	`'oapp'`	Basically a nop; don't use.
Open Document	`'aevt'`	`'odoc'`	Loads an experiment. Direct object is assumed to be coercible to a File System Spec record.
Print Document	`'aevt'`	`'pdoc'`	NA; don't use.
Quit Application	`'aevt'`	`'quit'`	Quits the program. If the experiment was modified, then Igor attempts to interact with the user to get save/no save directions. If interaction is not allowed, then an error is returned and nothing is done. To prevent errors, send the close event with appropriate save options prior to sending quit.
Close	`'core'`	`'clos'`	Acts on an experiment or window. For a window, the save/no save/ask optional parameter (keyAESaveOptions) is allowed and refers to making/replacing a recreation macro. For a document (experiment), keyAESaveOptions is allowed and an additional optional parameter keyAEDestination my be used to specify where to save (must be coercible to a FSS). If this is not given and the experiment is untitled and if an attempt to interact with the user fails then the experiment is not saved and an error (such as errAENoUserInteraction) is returned. Note that if the optional destination is given then the save options are ignored (why give a destination and then say no save?).
Save	`'core'`	`'save'`	Acts on experiment only. Takes same optional destination parameters as Close. A save with a destination is the same as a Save as.
Do Script	`'misc'`	`'dosc'`	Same as Eval Expression.
Eval Expression	`'aevt'`	`'eval'`	Executes commands. Acts just as if commands had been typed into the command line except the individual command lines are preceded by a "¶" symbol rather than the usual "•" symbol. Also, errors are returned in the error reply parameter of the event rather than putting up a dialog. **Note**: You can suppress history logging by executing the command, "Silent 2", and you can turn it back on by executing "Silent 3". Direct parameter must be text and not a file. Text can be of any length. You can return a string containing results by using the fprintf command with a file reference number of zero.

AppleScript

This topic is of interest to *Macintosh* programmers. Windows users should see the section for **ActiveX Automation** on page IV-236.

Igor supports the creation and execution of simple AppleScripts in order to send commands to other programs.

To execute an AppleScript program, you first compose it in a string and then pass it to the ExecuteScriptText operation, which in turn passes the text to Apple's scripting module for compilation and execution. The result (which might be an error message) is placed in a string variable named S_value. Igor does not save the compiled script so every time you call ExecuteScriptText your script will have to be recompiled. See the **ExecuteScriptText** operation on page V-145 for additional details.

The documentation for the **ExecuteScriptText** operation (page V-145) includes an example that shows how to execute a Unix command.

Because there is no easy way to edit a script or to see where errors occur, you should first test your script using Apple's Script Editor application.

You can use "Silent 2" to prevent commands your script sends to Igor from being placed in the history area.

You can send commands to Igor without using the tell keyword.

You should check your quoting carefully. Your text must be quoted both for Igor and for Apple's scripting system. For example,

```
ExecuteScriptText "Do Script \"Print \\\"hello\\\"\""
```

You should compose scripts in string variables one line at a time to improve readability.

If an error occurs that you can't figure out, print the string, copy from the history and paste into a Script Editor for debugging.

If the script returns a text return value, it will be quoted within the S_value string.

Don't forget to include the carriage return escape code, \r, at the end of each line of a multiline script.

The first time you call this routine, it may take an extra long time while the Mac OS loads the scripting modules.

Executing Unix Commands on Mac OS X

On Mac OS X, you can use AppleScript to send a command to the Unix shell. Here is a function that illustrates this:

```
Function/S ExecuteUnixShellCommand(uCommand, printCommandInHistory,
printResultInHistory)
    String uCommand                    // Unix command to execute
    Variable printCommandInHistory
    Variable printResultInHistory

    if (printCommandInHistory)
        printf "Unix command: %s\r", uCommand
    endif

    String cmd
    sprintf cmd, "do shell script \"%s\"", uCommand
    ExecuteScriptText cmd

    if (printResultInHistory)
        Print S_value
    endif

    return S_value
End
```

You can test the function with this command:

```
ExecuteUnixShellCommand("ls", 1, 1)
```

Life is a bit more complicated if the command that you want to execute contains spaces or other nonstandard Unix command characters. For example, imagine that you want to execute this:

```
ls /System/Library/Image Capture
```

These commands will not work because of the space in the command:

```
String unixCmd = "ls /System/Library/Image Capture"
ExecuteUnixShellCommand(unixCmd, 1, 1)
```

You need to quote the entire Unix command. In order to do this such that the quotes will make it through Igor's parser and AppleScript's parser, you must do this:

```
String unixCmd = "ls \\\"/System/Library/Image Capture\\\""
ExecuteUnixShellCommand(unixCmd, 1, 1)
```

Igor's parser converts \\ to \ and \" to ", so AppleScript sees this:

```
"ls \"/System/Library/Image Capture\""
```

AppleScript's parser converts \" to " so Unix sees this:

```
ls "/System/Library/Image Capture"
```

On Macintosh, if your system is set to use Japanese as the preferred language, this technique of embedding backslashes will not work. This is because, in the Japanese script, the code for backslash is used for the yen symbol. AppleScript receives yen symbols where backslash is intended and returns an error. A possible workaround is to put your commands in a Unicode file, created in TextEdit for example, and then use ExecuteUnixShellCommand to run execute the commands in Unicode file.

ActiveX Automation

ActiveX Automation, often called just Automation, is Microsoft's technology for allowing one program to control another. The program that does the controlling is called the Automation client. The program that is controlled is called the Automation Server. The client initiates things by making calls to the server which carries out the requested actions and returns results.

Automation client programs are most often written in Visual Basic or C++. They can also be written in other programming languages and in various scripting languages such as VBScript, JavaScript, Perl, Python and so on.

As of Igor Pro 5, Igor can play the role of Automation Server. If you want to write an client program to drive Igor, see "Automation Server Overview" in the "Automation Server" help file in "\Igor Pro Folder\Miscellaneous\Windows Automation".

Igor Pro does not directly support playing the role of Automation client. However, it is possible to write an Igor program which generates a script file which can act like an Automation client. See the CallMicrosoftWord experiment in the Igor Pro Folder:Examples:Programming folder.

An intermediate-level C++ programmer can also write an Igor XOP which plays the role of Automation client.

Igor Command Line

This information is for Windows programmers only. You can call Igor Pro from a Windows batch file or even from Igor itself using ExecuteScriptText using this operation-like syntax:

Igor.exe

Igor.exe [**/I /N /Automation**][***pathToFileOrCommands***][***pathToFile***]…
Igor.exe [**/I /Q /X /Automation**] "***commands***"
Igor.exe /SN=num /KEY="*key*" /NAME="*name*" [/ORG="*org*" /QUIT]

Parameters

The usual parameter to Igor.exe is a file which Igor opens. It is recommended that both the path to Igor.exe and the path to the file parameter be enclosed in quotes:

```
"C:\Program Files\WaveMetrics\Igor Pro Folder\Igor.exe" "C:\Igor Files\exp.pxp"
```

Multiple files can be opened by appending the path to the file(s) with an intervening space:

```
"C:\Program Files\WaveMetrics\Igor Pro Folder\Igor.exe" "C:\Dir\exp.pxp"
"C:\Dir\exp.dat"
```

With the /X flag, only one parameter is allowed and is interpreted as Igor commands:

```
"C:\Program Files\WaveMetrics\Igor Pro Folder\Igor.exe" /X "Make/O data=x;Display data"
```

The /SN, /KEY, and/NAME flags must all be used to successfully register Igor Pro. The optional /ORG parameter defaults to "".

Flags

Note: The / symbol can be replaced with a - symbol (ActiveX Automation uses a -Automation parameter when calling Igor Pro).

/Automation — Used automatically (along with /I) by the operating system when launching Igor Pro as an Automation Server. The command parameter that the OS sends is defined in the Registry, put there by the Igor installer or by the user, by merging an IgorProCOM.reg file. This flag isn't intended for use in batch files or ExecuteScriptText. For more details on ActiveX Automation Server, see "Automation Server Overview" in the "Automation Server" help file.

It can, however be used from the command line or a batch file to communicate with other programs in combination with /X. The /Automation flag keeps Igor windows hidden, which may be useful when calling Igor Pro from a web server CGI program.

/I — Launches a new "instance" of Igor that will open the file or execute the commands. Pressing Ctrl while launching Igor is the same as using the /I flag.

Without /I, files are opened and commands are executed by any Igor.exe that is currently running. If a parameter is an experiment file, the currently open experiment is closed before opening the new one (see the /Y and /N flags).

/KEY="*key*" — Specifies the license activation key. Use a value of the form:

```
/KEY="ABCD-EFGH-IJKL-MNOP-QR"
```

Do not omit the quotes, or it will fail.

/N — Forces the current experiment to be closed without saving if any of the file parameters are an experiment file.

To save a currently open experiment, use:

```
Igor.exe /X "SaveExperiment"
```

/NAME="*name*" — Defines the name of the licensed user(s). Cannot be "".

/ORG="*org*" — Specifies the optional name of the licensed organization. Default is "". Because Windows interprets the & symbol to mean "underline the next character when displayed in a dialog window," use && to display one & character in the About Igor dialog.

/Q — Doesn't show

```
Command line: /X "Make/O data=s;Display data"
```

in Igor's history window when using /X or /SN, etc.

/QUIT — Quits Igor Pro after entering license information when used with /SN, /KEY, and /NAME. Otherwise /QUIT is ignored.

To quit Igor Pro, use:

```
Igor.exe /X "Quit/N"
```

/SN=*num* — Specifies the license serial number.

/X — Executes the commands in the first (and only allowed) parameter. Use semicolons to separate commands.

Details

As of Igor 6.2, if a copy of Igor.exe is already running and if Igor.exe is launched again without /X, /SN or any path to a file, a new instance of Igor.exe is started.

Previous to Igor 6.2, launching Igor under those conditions would only activate the already-running instance of Igor.exe.

This means that double-clicking the Igor.exe icon will start another instance of Igor.exe, but double-clicking an experiment file will still open that file in the frontmost instance of Igor (or start up Igor if it isn't running), as it always has.

Example

This function launches another instance of Igor to open an experiment file:

```
Function LaunchAnotherIgor(expPath)
   String expPath       // Full Windows path to experiment file
                        // e.g.,  "C:\\Igor Files\\Experiment.pxp"

   String quote = "\""  // String containing a double-quote

   // Get path to Igor Pro folder in Macintosh file format.
   PathInfo Igor        // Stores output in S_path.

   // Get path to Igor in Windows format.
   String igorPath= ParseFilePath(5, S_path, "\\", 0, 0) + "Igor.exe"

   String scriptText = quote + igorPath + quote + " /I " + quote + expPath + quote
   ExecuteScriptText scriptText
End
```

These batch file commands register Igor Pro with the given (fictional) serial number and license activation key:

```
Igor.exe /SN=1234567/KEY="ABCD-EFGH-IJKL-MNOP-QR"/NAME="Me" /ORG="You && Me, Inc." /QUIT
```

Igor as a WWW CGI-Bin Server

You can use Igor Pro as part of a Web server on both Macintosh and Windows. For example, you might create a form that allows users to enter parameters and then see an Igor created graph based on the user's inputs.

On the Macintosh your Web server executes a compiled AppleScript (that you write based on our example) that communicates with Igor using the DoScript Apple event.

On Windows, we supply an IgorCGI.exe executable program than communicates with Igor using DDE.

On Windows, we supply an IgorCGI.exe executable program that communicates with Igor Pro using DDE. Another option is to communicate directly using ActiveX Automation Server (see "Automation Server Overview" in the "Automation Server" help file) or on the Igor command line.

On both platforms, we provide example procedures that you keep in the Igor Procedures folder to help you decode the values users enter in the form with their Web browser.

For more information and sample code, see Technical Note PTN004 — "Igor CGI on Macintosh" and Technical Note PTN005 — "Igor CGI on Windows".

Network Communications

The following sections contain material related to the network communication and Internet-related capabilities of Igor Pro:

URLs

URLs, or Uniform Resource Locators, are compact strings that represent a resource available via the Internet. The description of the URL standard is described in RFC1738 (http://www.rfc-editor.org/rfc/rfc1738.txt) and updated in RFC3986 (http://www.rfc-editor.org/rfc/rfc3986.txt).

Each URL is composed of several different parts, most of which are optional:

```
<scheme>://<username>:<password>@<host>:<port>/<path>?<query>
```

Some examples of valid URLs are:

```
http://www.example.com
http://www.example.com/afolder?key1=45&key2=66
http://myusername:Passw0rD@www.example.com:8010/index.html
ftp://ftp.wavemetrics.com
file://C:\Data:Trial1:control.ibw
file://hd:Test:TestFile1.txt
```

For most operations and functions that take a *urlStr* parameter, only the scheme and host parts of the URL are required. See the **Supported Network Schemes** section for information on which schemes are supported by which operations and functions, and which port is used by default if it is not provided as part of the URL.

Usernames and Passwords

You can provide a username and password as part of the URL. However authentication credentials may not be supported by all schemes (such as file://). Some operations allow you to provide a username and password by using a flag, such as the /U and /W flags with **FTPDownload**.

If a URL contains a username and password in the URL and the authentication flags are also used, the values specified in the flags override values provided in the URL.

If you do not provide a username and password as part of the URL, and you do not use the authentication flags, then no authentication is attempted. An exception to this rule is that the FTP operations will login to the FTP server using "anonymous" as the username and a generic email address as the password.

If either the username or password contains special or reserved characters, those characters must be percent-encoded.

Supported Network Schemes

Different operations and functions support different schemes:

Operation	Supported Schemes	Default Port
FetchURL	http	80
	ftp	21
	file	Not applicable
FTP operations*	ftp	21

* Includes **FTPUpload**, **FTPDownload**, **FTPDelete**, and **FTPCreateDirectory**.

Percent Encoding

Percent encoding is a way to encode characters in URLs that would otherwise have a special meaning or could be misinterpreted by servers. For example, a space character in a URL is encoded as "%20" using a percent character followed by the hex code for a space in the ASCII character set.

Most URLs contain only the letters A-Z and a-z, the digits 0-9, and a few other characters such as the underscore (_), hyphen (-), period (.), and tilde (~).

A URL may also contain "reserved characters" that may have special meaning depending on the way that they are used. Every URL contains the reserved characters ":" and "/" and may also contain one or more of the following reserved characters: !*'();@&=+$,?#[].

All operations and functions provided by Igor Pro that accept a URL string parameter expect that the URL has already been percent-encoded as necessary.

In most cases you don't need to worry about percent encoding because most URLs don't use reserved characters except for their special meaning. If you need to use a reserved character in a way that differs from the character's special meaning, you must percent-encode the character. You can use the **URLEncode** function for this purpose.

It is important that you not pass your entire URL to URLEncode to be encoded because that URL will not be understood by a server. URLEncode percent-encodes all reserved characters in the string you pass to it, because it cannot distinguish between reserved characters used for their special meaning and reserved characters used outside of their special meaning. Instead, you must pass each piece of the URL through URLEncode so that the final URL uses the correct syntax.

As an example, we'll use URLEncode to properly encode a URL that contains the following parts:

Part Name	Example
Scheme	http
Username	A. MacGyver
Password	yj@!2M
Host	www.example.com
Path	/tape/duct
Query	discount=10%&color=red

Without any percent-encoding, the URL is:

```
http://A. MacGyver:yj@!2M@www.example.com/tape/duct?discount=10%&color=red
```

If this URL were passed to **FetchURL**, the result would be an error because the URL contains several reserved characters that are not intended to be used in their standard way. For example, the "@" character indicates the separation between the username:password information and the start of the host name, but in this case the password itself also contains the "@" character. In addition, the "%" character is typically used to indicate that the next two characters represent a percent-encoded character, but in this example it is also part of the query. Finally, the username contains a space character. The space character is not technically a reserved character, but should be percent-encoded to ensure that it is handled correctly.

The following table shows the values of the parts of the URL that need to be percent-encoded by passing them through the URLEncode function:

Part Name	Encoded Value
Username	A%2E%20MacGyver
Password	yj%40%212M
Host	www.example.com
Path	/tape/duct
Query	discount=10%25&color=red

The properly percent-encoded URL is:

```
http://A%2E%20MacGyver:yj%40%212M@www.example.com/tape/duct?discount=10%25&color=red
```

For keyword-value pairs that make up the query part, each keyword and value must be percent-encoded separately because the "=" character that separates the key from the value and the "&" character that separates the pairs in the list must not be percent-encoded.

For more information on percent-encoding and reserved characters, see http://en.wikipedia.org/wiki/Percent-encoding.

Safe Handling of Passwords

Some operations and functions support the use of a username and password when making a network connection. If you use sensitive passwords you must take certain precautions to prevent them from being accidentally revealed.

1. Always use the /V=0 flag when using a username or password with the /U (username) and /W (password) flags. Otherwise, the debugging information that is, by default, printed to the command history window will contain those values and anyone who sees the experiment could see them.
2. Do not hard code username or password values into procedures, since anyone with access to the procedure file could read them.
3. Do not store username or password values in global variables. Since global variables are saved with an experiment, if someone else had access to your experiment they could see this information.

Here is an example of how a username and sensitive password can be used in a secure manner:

```
Function SafeLogin()
   String username = ""
   String password = ""
   Prompt username, "Username"
   Prompt password, "Password"
   DoPrompt "Enter username and password", username, password
   if (V_flag == 1)
      // User hit cancel button, so do nothing.
      return 0
   endif

   // Percent-encode in case username and password contain reserved characters.
   String encodedUser = URLEncode(username)
   String encodedPass = URLEncode(password)

   String theURL
   sprintf theURL, "http://%s:%s@www.example.com", encodedUser, encodedPass
   String response = FetchURL(theURL)
   // NOTE: For FTP operations, make sure to use /V=0 so that the username
   // and password are not printed to the history.

   return 0
End
```

Note that the user is prompted to provide the username and password when the function is called and that only local string variables are used to store the username and password. The values in those string variables are not stored once the function is done executing.

Note also that the password is not hidden during entry in the dialog. Igor currently does not provide a way to do this.

Network Timeouts and Aborts

Some network calls may return an error code to Igor if they timeout. Depending on the specific operation or function, there can be a number of causes for a timeout.

If a network connection cannot be made after a period of time it will timeout. The amount of time allowed for a connection to be established is dependent on several factors.

You can always abort a network operation or function by pressing cmd-period (Macintosh) or Ctrl-Break (Windows). You must hold the keys down until Igor aborts the operation.

Network Connections From Multiple Threads

All network-related operations and functions are thread-safe, which means that they can be called from multiple preemptive threads at the same time. This capability can be useful when:

- You want to retrieve information from several different URLs as quickly as possible.
- You want to do a long download or other operation in the background to avoid tieing Igor up.

The following example illustrates the first of these cases. It uses **FetchURL** to retrieve a list of the most frequently downloaded books from the Project Gutenberg web site. It then uses FetchURL to download the entire text of the top four books and prints the number of characters in each.

```
ThreadSafe Function GetThePage(url)
   String url

   String response = FetchURL(url)
   return strlen(response)
End

Function GutenbergTopCharacterCount()
   String topBooksURL = "http://www.gutenberg.org/browse/scores/top"
   String baseURL = "http://www.gutenberg.org/files/"

   // Get the contents of the page.
   String response = FetchURL(topBooksURL)
   Variable error = GetRTError(1)
   if (error || numtype(strlen(response)) != 0)
      Print "Error getting the list of most popular books."
      return 0
   endif

   String topBooksHTML = response

   // Remove all line endings.
   topBooksHTML = ReplaceString("\n", topBooksHTML, "")
   topBooksHTML = ReplaceString("\r", topBooksHTML, "")

   // Parse the page to get the section of the page
   // with the list of the most popular books from yesterday.
   // This could break if the format of the web page changes.
   String regExp = "(?i)<h2 id=\"books-last1\">.*?<ol>(.*?)</ol>"
   String topYesterdayHTML = ""
   SplitString/E=regExp topBooksHTML, topYesterdayHTML
   if (V_flag != 1)
      Print "Error parsing the top 100 books section."
      return 0
   endif

   // Replace the line endings.
   topYesterdayHTML = ReplaceString("</li><li>", topYesterdayHTML, "\r")
```

```
   // Create a wave to store text info about the top four books.
   Variable numBooksToUse = 4
   Make/O/T/N=(numBooksToUse, 2) topBooksInfo

   Make/O/N=(numBooksToUse) characterCounts

   Variable n
   String bookNumStr
   Variable bookNum
   String titleAuthor
   String thisLine
   Variable pos
   String bookURL = ""
   regExp = "(?i)a href=\".*?(\d+)\">(.+?)</a>"
   for (n=0; n<numBooksToUse; n+=1)
      // For each book we're going to look at, get the
      // partial URL and the title/author text.
      thisLine = StringFromList(n, topYesterdayHTML, "\r")
      SplitString/E=regExp thisLine, bookNumStr, titleAuthor
      if (V_flag != 2)
         Print "Error parsing the URL and title/author information."
         return 0
      endif

      // Remove the (###) stuff at the end of titleAuthor if it's there.
      pos = strsearch(titleAuthor, "(", 0)
      if (pos > 0)
         titleAuthor = titleAuthor[0, pos - 1]
      endif

      bookNum = str2num(bookNumStr)

      // Store the information about the book in the text wave.
      sprintf bookURL, "%s%d/%d.txt", baseURL, bookNum, bookNum
      topBooksInfo[n][0] = bookURL
      topBooksInfo[n][1] = titleAuthor
   endfor

   // Download each book (using multiple threads if possible)
   // and count the number of characters in each.
   MultiThread characterCounts = GetThePage(topBooksInfo[p][0])

   // Print the results.
   Print "The top four books by download from yesterday are:"
   for (n=0; n<numBooksToUse; n+=1)
      Printf "%s (%d characters)\r", topBooksInfo[n][1], characterCounts[n]
   endfor
End
```

Here is an example of what the output was when this help file was written:

The top four books by download from yesterday are:

```
  Ulysses by James Joyce (1573044 characters)
  Alice's Adventures in Wonderland by Lewis Carroll (167529 characters)
  Piper in the Woods by Philip K. Dick (62214 characters)
  Pride and Prejudice by Jane Austen (704160 characters)
```

File Transfer Protocol (FTP)

The **FTPDownload**, **FTPUpload**, **FTPDelete**, and **FTPCreateDirectory** operations support simple transfers of files and directories over the Internet.

Since Igor's SaveNotebook operation can generate HTML files from notebooks, it is possible to write an Igor procedure that downloads data, analyzes it, graphs it, and uploads an HTML file to a directory used by a Web server. You can then use the BrowseURL operation to verify that everything worked as expected. For a demo of some of these features, see "Igor Pro Folder:Examples:Feature Demos:Web Page Demo.pxp".

FTP Limitations

All FTP operations run "synchronously". This means that, if the operation executes in the main thread, Igor can not do anything else. However, it is possible to perform these operations using an Igor preemptive thread so that they execute in the background and you can continue to use Igor for other purposes. For more information, see **Network Connections From Multiple Threads** on page IV-242.

Igor does not currently provide any way for the user to browse the remote server from within Igor itself.

Igor does not provide any secure way to store passwords. Consequently, you should not use Igor for FTP in situations where tight security is required. See **Safe Handling of Passwords** on page IV-241 for an example of how to securely prompt the user for a password.

Igor does not provide any support for using proxy servers. Proxy servers are security devices that stand between the user and the Internet and permit some traffic while prohibiting other traffic. If your site uses a proxy server, FTP operations may fail. Your network administrator may be able to provide a solution.

Igor does not include operations for listing a server directory or changing its current directory.

Downloading a File

The following commands transfer a file from an FTP server to a local hard disk:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String localPath = "hd:Test:TestFile1.txt"          // Macintosh
FTPDownload/U="ftpTestAccount"/W="dropbox" url, localPath
```

These commands transfer the file TestFile1.txt from the WaveMetrics FTP server into the Test directory on the local hard disk "hd". The Test directory must already exist on the local hard disk. The TestFile1.txt file may or may not exist on the local hard disk. If it does not exist, the FTPDownload command will create it. If it does exist, FTPDownload will ask if you want to overwrite it. To overwrite it without being asked, use the /O flag.

Warning: If you elect to overwrite it, all previous contents of the local TestFile1.txt file will be obliterated.

As of this writing, "ftpTestAccount" is the user name and "dropbox" is the password for the "test" directory on the WaveMetrics FTP server.

Now consider these commands:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String localPath = "hd:Test:FileA.txt"
FTPDownload/U="ftpTestAccount"/W="dropbox" url, localPath
```

This does the same thing as the previous example except that the resulting file on the local hard disk will be named FileA.txt instead of TestFile1.txt.

FTPDownload presents a dialog asking you to specify the local file name and location in the following cases:

1. You use the /I (interactive) flag.
2. The parent directory specified by the local path does not exist. In the examples above, the parent directory is Test.
3. The specified local file exists and you have not used the /O (overwrite) flag.

Downloading a Directory

The following commands transfer a directory from an FTP server to a local hard disk:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestDir1"
String localPath = "hd:Test:TestDir1"
FTPDownload/D/U="ftpTestAccount"/W="dropbox" url, localPath
```

Note the use of the /D flag to specify that you are transferring a directory.

This command transfers the TestDir1 directory from the WaveMetrics FTP server into the Test directory on the local hard disk "hd". The Test directory must already exist on the local hard disk. The TestDir1 directory may or may not exist. If it does not exist, the FTPDownload command will create it. If it does exist, FTPDownload will ask if you want to overwrite it. To overwrite it without being asked, use the /O flag.

Warning: If you elect to overwrite it, all previous contents of the local TestDir1 will be obliterated.

Now consider these commands:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestDir1"
String localPath = "hd:Test:TestDir2"
FTPDownload/D/U="ftpTestAccount"/W="dropbox" url, localPath
```

This does the same thing as the previous example except that the resulting directory on the local hard disk will be named TestDir2 instead of TestDir1.

Warning: If you elect to overwrite it, all previous contents of the local TestDir2 will be obliterated.

The local path that you specify must not end with a colon or backslash. For example, if you execute:

```
FTPDownload/D "ftp://ftp.wavemetrics.com/pub/test/TestDir1", "hd:Test:TestDir1:"
```

FTPDownload will present a dialog asking you to specify the local directory because the local path ends with a colon and FTPDownload is looking for the name of the directory to be created on the local hard disk.

FTPDownload presents a dialog asking you to specify the local directory in the following cases:

1. You use the /I (interactive) flag.
2. The parent directory specified by the local path does not exist. In the examples above, the parent directory is Test.
3. The specified directory (TestDir1 in the example above) exists and you have not used the /O (overwrite) flag.
4. FTPDownload gets an error when it tries to create the specified directory. This could happen, for example, if the directory name that you specify is not a legal directory name. For example, the name might be too long or use characters that are forbidden in the local file system.

Uploading a File

The following commands upload a file to an FTP server:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String localPath = "hd:Test:TestFile1.txt"
FTPUpload/U="ftpTestAccount"/W="dropbox" url, localPath
```

These commands transfer the file TestFile1.txt from the local hard disk into the test directory on the WaveMetrics FTP server. The test directory is created if it does not already exist.

Note: The /O flag has no effect on the FTPUpload operation when uploading a file. FTPUpload *always* overwrites an existing server file, whether /O is used or not.

Warning: If you overwrite a server file, all previous contents of the file are obliterated.

To overwrite an existing file on the server, you must have permission to delete files on that server. The server administrator determines what permission a particular user has.

Now consider these commands:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile2.txt"
String localPath = "hd:Test:TestFile1.txt"
FTPUpload/U="ftpTestAccount"/W="dropbox" url, localPath
```

This does the same thing as the previous example except that the resulting file on the FTP server will be named TestFile2.txt instead of TestFile1.txt.

FTPUpload presents a dialog asking you to specify the local file in the following cases:

1. You use the /I (interactive) flag.
2. The local parent directory (e.g., Test) or the local file (e.g., TestFile1.txt) does not exist.

Uploading a Directory

The following commands upload a directory to an FTP server:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestDir1"
String localPath = "hd:Test:TestDir1"
FTPUpload/D/U="ftpTestAccount"/W="dropbox" url, localPath
```

These commands transfer the TestDir1 directory from the local hard disk into the test directory on the WaveMetrics FTP server. The test and TestDir1 directories are created on the server if they do not already exist. If the TestDir1 directory does exist, FTPUpload overwrites it.

Note: FTPUpload *always* overwrites an existing server directory, whether /O is used or not.

Warning: If you overwrite a server directory using /O or /O=1, all previous contents of the directory will be obliterated.

If /O=2 is used, FTPUpload performs a merge of the directory contents. This means that files and directories in the source overwrite files and directories on the server that have the same name, but files and directories on the server whose names do not conflict with those in the source directory are not modified.

To overwrite an existing directory on the server, you must have permission to delete directories on that server. The server administrator determines what permission a particular user has.

Now consider these commands:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestDir2"
String localPath = "hd:Test:TestDir1"
FTPUpload/D/U="ftpTestAccount"/W="dropbox" url, localPath
```

This does the same thing as the previous example except that the resulting directory on the FTP server will be named TestDir2 instead of TestDir1.

The local path that you specify must not end with a colon or backslash. For example, if you execute:

```
FTPUpload/D "ftp://ftp.wavemetrics.com/pub/test/TestDir1", "hd:Test:TestDir1:"
```

FTPUpload will present a dialog asking you to specify the local directory because the local path ends with a colon and FTPUpload is looking for the name of the directory to be uploaded.

FTPUpload presents a dialog asking you to specify the local directory in the following cases:

1. You use the /I (interactive) flag.
2. The specified directory (TestDir1 in the example above) or any of its parents do not exist.

If you don't have permission to remove and to create directories on the server, FTPUpload will fail and return an error.

Creating a Directory

The following commands create a new directory on an FTP server:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/newDirectory1"
FTPCreateDirectory/U="ftpTestAccount"/W="dropbox" url
```

If the /pub/test/newDirectory1 directory already exists on the server, this command does nothing. This is not treated as an error, though the V_Flag output variable is set to -1 to indicate that the directory already existed.

If you don't have permission to create directories on the server, FTPCreateDirectory fails and returns an error.

Deleting a Directory

The following commands delete a directory on an FTP server:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/oldDirectory1"
FTPDelete/D/U="ftpTestAccount"/W="dropbox" url
```

If you don't have permission to delete directories on the server, or if the specified directory does not exist on the server, FTPDelete fails and returns an error.

FTP Transfer Types

The FTP protocol supports two types of transfers: image and ASCII. Image transfer is appropriate for binary files. ASCII transfer is appropriate for text files.

In an image transfer, also called a binary transfer, the data on the receiving end will be a replica of the data on the sending end. In an ASCII transfer, the receiving FTP agent changes line terminators to match the local convention. On Macintosh and Unix, the conventional line terminator is linefeed (LF, ASCII code 0x0A). On Windows, it is carriage-return plus linefeed (CR+LF, ASCII code 0x0D + ASCII code 0x0A).

If you transfer a text file using an image transfer, the file may not use the local conventional line terminator, but the data remains intact. Igor Pro can display text files that use any of the three conventional line terminators, but some other programs, especially older programs, may display the text incorrectly.

On the other hand, if you transfer a binary file, such as an Igor experiment file, using an ASCII transfer, the file will almost certainly be corrupted. The receiving FTP agent will convert any byte that happens to have the value 0x0D to 0x0A or vice versa. If the local convention calls for CRLF, then a single byte 0x0D will be changed to two bytes, 0x0D0A. In either case, the file will become unusable.

FTP Troubleshooting

FTP involves a lot of hardware and software on both ends and a network in between. This provides ample opportunity for errors.

Here are some tips if you experience errors using the FTP operations.

1. Use an FTP client or web browser to connect to the FTP site. This confirms that your network is operating, the FTP server is operating, and that you are using the correct URL.
2. Use an FTP client or web browser to verify that the user name and password that you are using is correct or that the server allows anonymous FTP access.

   Many web browser accept URLs of the form:

   ```
   ftp://username:password@ftp.example.com
   ```

   However the password is not transferred securely.
3. Use an FTP client or web browser to verify that the directory structure of the FTP server is what you think it is.
4. Using an FTP client or web browser, do the operation that you are attempting to do with Igor. This verifies that you have sufficient permissions on the server.
5. Use /V=7 to tell the Igor operation to display status information in the history area.

6. Try the simplest transfer you can. For example, try to download a single file that you know exists on the server.
7. If you have access to the FTP server, examine the FTP server log for clues.

Hypertext Transfer Protocol (HTTP)

The **FetchURL** function supports simple URL requests over the Internet from web or FTP servers and to local files. For example, you can use FetchURL to get the source code of a web page in text form, and then process the text to extract specific information from the response.

HTTP Limitations

At this time, **FetchURL** and **BrowseURL** routines work with the HTTP protocol.

Currently not supported are features such as using network proxy servers, using the HTTP POST method to submit forms and upload files to a web server, and making secure network connections using the Secure Socket Layer (SSL) protocol.

Downloading a Web Page Via HTTP

This example uses FetchURL to download the contents of the WaveMetrics home page into a string, and then counts the number of times that the string "Igor" occurs in the text of the page.

```
Function DownloadWebPageExample()
   String webPageText = FetchURL("http://www.wavemetrics.com")
   if (numtype(strlen(webPageText)) == 2)
      Print "There was an error while downloading the web page."
   endif
   Variable count, pos
   do
      pos = strsearch(webPageText, "Igor", pos, 2)
      if (pos == -1)
         break    // No more occurrences of "Igor"
      else
         pos += 1
         count += 1
      endif
   while (1)
   Printf "The text \"Igor\" was found %d times on the web page.\r", count
End
```

Downloading a File Via HTTP

This example uses FetchURL to download a file from a web server. Because FetchURL does not support storing the downloaded data into a file directly, we store the data in memory and then use Igor to write that data to a file on disk.

Though the example uses a URL that begins with http://, FetchURL also supports ftp:// and file:///. You could use the code below with a different URL to download a file from an FTP server or even to access a local on-disk file.

```
Function DownloadWebFileExample()
   String url = "http://www.wavemetrics.net/IgorManual.zip"

   // Based on the URL, determine what the destination
   // file name should be. This will be the default in the
   // Save As... dialog.
   String urlStrParam = RemoveEnding(url, "/")
   Variable parts = ItemsInList(urlStrParam, "/")
   String destFileNameStr = StringFromList(parts - 1, urlStrParam, "/")
   if (strlen(destFileNameStr) < 1)
```

```
      Print "Error: Could not determine the name of the destination file."
      return 0
   endif

   Variable refNum
   Open/D/M="Save File As..."/T="????" refNum as destFileNameStr
   String fullFilePath = S_fileName

   if (strlen(fullFilePath) > 0) // No error and user didn't cancel in dialog.
      // Open the selected file so that it can later be written to.
      Open/Z/T="????" refNum as fullFilePath
      if (V_flag != 0)
         Print "There was an error opening the local destination file."
      else
         String response = FetchURL(url)
         Variable error = GetRTError(1)
         if (error == 0 && numtype(strlen(response)) == 0)
            FBinWrite refNum, response
            Close refNum
            Print "The file was successfully downloaded as " + fullFilePath
         else
            Close refNum
            DeleteFile/Z fullFilePath  // Clean up the empty file.
            Print "There was an error downloading the file."
         endif
      endif
   endif
End
```

Making a Query Via HTTP

Another use for HTTP requests is to get the server's response to a query. While it's not possible to upload files to a web server or simulate the submission of complicated web forms (these would require the HTTP POST method, which is not supported), many simple web forms use the HTTP GET method, which FetchURL supports. For example, you can simulate the submission of the basic Google search form using the following code.

```
Function WebQueryExample()
   String keywords
   String baseURL = "http://www.google.com/search"

   // Prompt the user to enter search keywords.
   Prompt keywords, "Search for"
   DoPrompt "", keywords
   if (V_flag == 1)           // User clicked cancel button.
      return 0
   endif

   // Pass the search terms through URLEncode to
   // properly percent-encode them.
   keywords = URLEncode(keywords)

   // Build the full URL.
   String url = ""
   sprintf url, "%s?q=%s", baseURL, keywords

   // Fetch the results.
   String response
   response = FetchURL(url)
   Variable error = GetRTError(1)
   if (error != 0 || numtype(strlen(response)) != 0)
```

```
      Print "Error fetching search results."
      return -1
   endif

   // Try to extract the URL of the first result.
   String regExp = "<h3 class=\"r\">.+?href=\"(.+?)\".*"
   String firstURL
   SplitString/E=regExp response, firstURL
   if (V_flag == 1)
      BrowseURL firstURL
   else
      Print "Could not extract the first result from the"
      Print "results page. Your search terms might not"
      Print "have given any results, or the format of"
      Print "the results may have changed so that the"
      Print "first result cannot be extracted."
   endif
End
```

HTTP Troubleshooting

Here are some tips if you experience errors using **FetchURL**:

1. Use a web browser to connect to the site. This confirms that your network is operating, the server is operating, and that you are using the correct URL.
2. FetchURL generates an error if it cannot connect to the destination server, which could happen if your computer is not connected to the network or if the target URL contains an invalid host name or port number.

   However if the URL contains an invalid path or if the destination URL requires you to provide a username and password, FetchURL will likely not generate an error. The reason is these errors typically result in a web page being returned, though not the one you expected. If you need to check that a call to FetchURL returned a valid web page and not an error web page, you must do that in your own code. One possibility would be to try searching the page for key phrases, such as "File Not Found" or "Page Not Found".

Operation Queue

Igor supports an operation queue that allows for the execution of what were previously illegal operations. Items in the operation queue execute only when nothing else is happening. Macros and functions must not be running and the command line must be empty.

You may append to the operation queue using

`Execute/P <command string>`

The /P posts the command to operation queue. You can also specify the /Q (quiet) or /Z (ignore error) flags. See **Execute/P** operation (page V-145) for details about /Q and /Z.

The command string may be either special commands that are unique to the operation queue or may be ordinary Igor commands. The special commands are:

INSERTINCLUDE *procedureSpec*

DELETEINCLUDE *procedureSpec*

COMPILEPROCEDURES

NEWEXPERIMENT

LOADFILE *filePath*

MERGEEXPERIMENT *filePath*

INSERTINCLUDE and DELETEINCLUDE insert or delete #include lines in the main procedure window. *procedureSpec* is whatever you would use in a #include statement except for "#include" itself.

COMPILEPROCEDURES does just what it says, compiles procedures. You must call it after operations such as INSERTINCLUDE that modify, add, or remove procedure files.

NEWEXPERIMENT closes the current experiment without saving.

LOADFILE opens the file specified by *filePath*. *filePath* is either a full path or a path relative to the Igor Folder. The file may be any of a number of file types Igor can open. If the file is an experiment, be sure to execute NEWEXPERIMENT first to avoid putting up a dialog. If you want to save the changes in an experiment before loading another, you can use the standard SaveExperiment operation.

MERGEEXPERIMENT merges the experiment file specified by *filePath* into the current experiment. Before using this, make sure you understand the caveats regarding merging experiments. See **Merging Experiments** on page II-32 for details.

Note: The special operation queue keywords must be all caps and must have one space after the keyword.

Here is an example:

```
Function DemoQueue()
   Execute/P "INSERTINCLUDE <Multi-peak fitting 1.3>"
   Execute/P "INSERTINCLUDE <Peak Functions>"
   Execute/P "COMPILEPROCEDURES "
   Execute/P "CreateFitSetupPanel()"
   Execute/P "Sleep 00:00:04"
   Execute/P "NEWEXPERIMENT "
   Execute/P "LOADFILE :Examples:Feature Demos:Live mode.pxp"
   Execute/P "DoWindow/F Graph0"
   Execute/P "StartButton(\"StartButton\")"
End
```

In the above example, the example experiment Live mode.pxp was chosen because it happened to have a .pxp extension which allows the example to work on both Macintosh and Windows — the file name and path must be exact.

One important use of the operation queue is providing easy access to useful procedure packages. The "Igor Pro Folder/Igor Procedures" folder contains a procedure file that has Menu definitions similar to the following (**Caution**! The three commands under the example submenu are very long and are wrapped to fit on the page):

```
Menu "Analysis"
    Submenu "Packages"
        "Multipeak Fitting",Execute/P "INSERTINCLUDE  <Multi-peak fitting 1.3>";Execute/P
"INSERTINCLUDE  <Peak Functions>";Execute/P "COMPILEPROCEDURES ";Execute/P "CreateFitSetupPanel()"
        "Wave Arithmetic",Execute/P "INSERTINCLUDE  <Wave Arithmetic Panel>";Execute/P
"COMPILEPROCEDURES ";Execute/P "InitWaveArith()"
        "Probability Graph",Execute/P "INSERTINCLUDE  <Probability Graph>";Execute/P
"COMPILEPROCEDURES ";Execute/P " Probability_Axis()"
    End
End
```

To try this out, start Igor and choose one of the items from the Analysis→Packages menu.

User-Defined Hook Functions

Igor Pro will call specific user-defined functions, called "hook" functions, if they exist, when Igor Pro performs certain actions. Hook functions allow savvy programmers to customize Igor's behavior. In some cases the hook function may inform Igor that the action has been completely handled, and that Igor shouldn't perform the action. For example, you could write a hook function to load data from a certain kind of text file that Igor can not handle directly.

This section discusses general hook functions that do not apply to a particular window. For information on window-specific events, see Window Hook Functions.

There are two ways to get Igor to call your general hook function. The first is by using a predefined function name. For example, if you create a function named AfterFileOpenHook, Igor will automatically call it after opening a file. The second way is to explicitly tell Igor that you want it to call your hook using the **SetIgorHook** operation.

If you use a predefined hook function name, you should make the function static (private to the file containing it) so that other procedure files can use the same predefined name. This is discussed under **Static Hook Functions** (see page IV-263).

Here are the predefined hook functions.

Action	Hook Function Called
Procedures have been successfully compiled	**AfterCompiledHook**
A file or experiment has just been opened	**AfterFileOpenHook**
A target window has been created	**AfterWindowCreatedHook**
The Debugger window is about to open	**BeforeDebuggerOpensHook**
An experiment is about to be saved	**BeforeExperimentSaveHook**
A file or XOP is about to be opened	**BeforeFileOpenHook**
Igor about to open a new experiment	**IgorBeforeNewHook**
Igor about to quit	**IgorBeforeQuitHook**
Igor building and enabling menus or about to handle a menu selection	**IgorMenuHook**
Igor quitting	**IgorQuitHook**
Igor starting or new experiment	**IgorStartOrNewHook**

To create hook functions, you must write functions with the specified names and store them in any procedure file. If you store the procedure file in "Igor Pro User Files/Igor Procedures" (see **Igor Pro User Files** on page II-46 for details), Igor will automatically open the file and compile the functions when it starts up and will execute the IgorStartOrNewHook function if it exists.

You can use BeforeFileOpenHook to load your own custom data files via drag-and-drop. Your BeforeFileOpenHook function must detect your type of file by examining the file's type and creator codes or by examining its contents. If the file is of the right type, you then call your XOP or use the FBinRead or FReadLine operation to load data from the file. You then avoid opening the file by returning a value of 1. If the file is not your file, you return 0, which loads the file. See the examples in the **BeforeFileOpenHook** operation (page IV-259) description.

Windows system files with .bin, .com, .dll, .exe, and .sys extensions aren't passed to the hook functions.

This example checks the type of file being opened. If it is an Excel file, it loads the file using the XLLoadWave XOP. It is intended to allow you to load an Excel file by dragging its icon onto the Igor Pro application icon.

```
static Function BeforeFileOpenHook(refNum,fileName,path,type,creator,kind)
   Variable refNum,kind
   String fileName,path,type,creator

   String xop="XLLoadWave"    // name of XOP command to load data
   Variable isExcel,handledOpen=0

   // This tests the file name extension.
   String extension = ParseFilePath(4, fileName, ":", 0, 0)
   isExcel = CmpStr(extension, "xls")==0

   // This tests the Macintosh file type.
   // "XLS ", XLS3, XLS4, XLS5, XLW4, XLW5 are Excel file types
   isExcel += CmpStr(type[0,2],"XLS")==0
   isExcel += CmpStr(type[0,2],"XLW")==0
```

```
   if (isExcel)
      if (exists(xop)==4)  // Load only if XOP installed
         Close refNum
         String cmd
         cmd = xop + "/A/T/P="+path+" \""+fileName+"\""
         Execute cmd
         handledOpen=1     // we handled the open event
      endif
   endif

   return handledOpen   // 1 tells Igor not to open the file itself
End
```

A return value of 1 indicates that you handled the open file event and that Igor should not open it. A return value of 0 specifies that you did not handle the event and that Igor should deal with it.

For more examples, see **BeforeFileOpenHook** on page IV-259.

Another possible use of a hook function is to predefine global variables, strings, waves, symbolic paths, or data folders when a new experiment is started up:

```
static Function IgorStartOrNewHook(igorApplicationNameStr)
   String igorApplicationNameStr

   NewPath/O pathToMyData "HD:My Data:"
   Variable/G root:V_no_MIME_TSV_Load = 1 // See AfterFileOpenHook
   Make/O/T root:TestNames={"RDC #1","KLZ #2","ARB #3","MOR #4"}
End
```

The following sections describe the individual hook functions in detail.

AfterCompiledHook

AfterCompiledHook(*refNum*, *fileNameStr*, *pathNameStr*, *fileTypeStr*, *fileCreatorStr*, *fileKind*)

AfterCompiledHook is a user-defined function that Igor calls after the procedure windows have all been compiled successfully.

You can use AfterCompiledHook to initialize global variables or data folders, among other things.

The function result from AfterCompiledHook must be 0. All other values are reserved for future use.

See Also

SetIgorHook, **User-Defined Hook Functions** on page IV-251.

AfterFileOpenHook

AfterFileOpenHook(*refNum*, *fileNameStr*, *pathNameStr*, *fileTypeStr*, *fileCreatorStr*, *fileKind*)

AfterFileOpenHook is a user-defined function that Igor calls *after* it has opened a file or experiment that the user has double-clicked or dragged onto the Igor icon, or a file opened as a result of an "open" Apple event from another program (such as a Web browser).

Note: AfterFileOpenHook is not called when the Open File or Load Waves menus are selected.

The parameters contain information about the file, which has already been opened for read-only access.

AfterFileOpenHook's return value is ignored unless *fileKind* is 9. If the returned value is zero, the default action is performed.

Parameters

Variable *refNum* is the file reference number. You use this number with the **FReadLine**, **FStatus**, **FSetPos**, **FBinWrite**, **FBinRead**, **fprintf**, and **wfprintf** operations to read from or write to the file. Normally, the file is opened for read-only operations, but for experiment files and XOP files, *refNum* will be -1, meaning the file has not been opened for you. You can close the file and reopen it for write access, if you wish.

Igor always closes the file when the user-defined function returns, and *refNum* becomes invalid (don't store the value of *refNum* in a global for use by other routines, since the file it refers to has been closed).

String *fileNameStr* contains the name of the file (including any extension).

String *pathNameStr* contains the name of the symbolic path. *pathNameStr* is not the value of the path. Use the PathInfo operation to determine the path's value.

String *fileTypeStr* contains the file type. This was conceived with the Macintosh in mind. Under Windows, this is usually the file extension, such as ".txt". However, if the file is an Igor-registered file, then *fileTypeStr* is set to one of the Macintosh file type codes (for cross-platform compatibility). Some Igor-registered files types are listed in the following table:

Type of File	*fileTypeStr* Contents		Extension
Igor Experiment, packed	IGsU		.pxp
Igor Experiment, packed, stationery/template	IGsS		.pxt
Igor Experiment, unpacked	IGSU		.uxp
Igor Experiment, unpacked, stationery/template	IGSS		.uxt
Igor XOP	IXOP		.xop
Igor Procedure	TEXT		.ipf
Igor Notebook (formatted)	WMT0	(zero, not oh)	.ifn
Igor Notebook (unformatted)	TEXT		.txt
Igor Notebook stationery/template (formatted)	WMTS		.ift
Igor Text (data and commands)	TEXT		.itx *or* .awav
Igor Binary	IGBW		.ibw *or* .bwav
Igor Published Edition	edtp		
Igor Help	WMT0	(zero, not oh)	.ihf

In addition, some other nonregistered extensions are converted to Mac-like file types:

Type of File	*fileTypeStr*		Extension
Encapsulated PostScript	EPS	(trailing space)	.eps *or* .epsf
Rich Text	RTF	(trailing space)	.rtf
Internet Shortcut	LINK		.url*
Text (also unformatted Igor Notebook)	TEXT		.txt
Batch file	TEXT		.bat
ASCII or data	TEXT		.csv, .dat, *or* .tsv

* The .url extension isn't shown in Windows Explorer windows.

Regardless of the value of *fileTypeStr*, the *fileNameStr* parameter has the file name and the extension.

String *fileCreatorStr* contains the creator code. Also conceived with the Macintosh in mind, this is usually "IGR0" for Igor-registered files, the full file path to the application registered to open the file, or the full file path to the file itself if unregistered by any application.

Some Windows examples are:

File	*fileCreatorStr*
My Experiment.pxp	IGR0 (zero, not oh)
My Igor Data.ibw	IGR0 (zero, not oh)
Better.bmp	C:\Program Files\MS\MSPAINT.EXE
AFile.bat	C:\Mine\AFile.bat

String *fileCreatorStr* contains the creator code. Some Macintosh examples are:

Application	*fileCreatorStr* Contents
Igor or Igor Pro	IGR0 (zero, not oh)
TeachText or SimpleText	ttxt

Variable *fileKind* is a number that identifies what kind of file Igor thinks it is. If the user's AfterFileOpenHook routine returns 0, Igor performs the Default Action listed in the table:

Kind of File	*fileKind*	Default Action, if Any
Unknown	0	
Igor Experiment, packed (stationery, too)	1	
Igor Experiment, unpacked (stationery, too)	2	
Igor XOP	3	
Igor Binary File	4	
Igor Text (data and commands)	5	
Text, no numbers detected in first two lines	6	
General Numeric text (no tabs)	7	
Numeric text Tab-Separated-Values	8	
Numeric text Tab-Separated-Values, MIME	9	Display loaded data in a new table and a new graph.
Text, with tabs	10	
Igor Notebook (unformatted or formatted)	11	
Igor Procedure	12	
Igor Help	13	

Details

AfterFileOpenHook's return value is ignored, except when *fileKind* is 9 (Numeric text, Tab-Separated-Values, MIME). If you return a value of 0, Igor executes the default action, which displays the loaded data in a table and a graph. If you return a value of 1, Igor does nothing.

Note: Another way to disable the MIME-TSV default action is define a global variable named V_no_MIME_TSV_Load (in the root data folder) and set its value to 1. In this case any file of *fileKind* = 9 is reassigned a *fileKind* of 8.

The default action for *fileKind* = 9 makes Igor a MIME-TSV document Helper Application for Web browsers such as Netscape or Internet Explorer.

The exact criteria for Igor to consider a file to be of kind 9 are:

- *fileTypeStr* must be "TEXT" or "WMT0" (that's a zero, not an oh).
- Either the first line of the file must begin with a # character, or the name of the file must end with ".tsv" in either lower or upper case.
- The first line must contain one or more column titles. If the line starts with a # character, the first column title must not start with "include", "pragma" or the ! character. Spaces are allowed in the titles, but if two or more title columns are present, they must be separated by one tab character.
- The second line must contain one or more numbers. If two or more numbers, they must be separated by one tab character, and the first line's words must also be separated by tabs.

When the MIME-TSV file contains one column of data, it is graphed as a series of Y values.

Short columns (less than 50 values) are graphed with lines and markers, longer columns with lines only. Preferences are turned on when the graph is made.

Two columns are assumed to be X followed by Y, and are graphed as Y versus X. More columns do not affect the graph, though they are shown in the table.

Example

```
// This hook function prints the first line of opened TEXT files
// into the history area
Function AfterFileOpenHook(refNum,file,pathName,type,creator,kind)
   Variable refNum,kind
   String file,pathName,type,creator
   // Check that the file is open (read only), and of correct type
   if( (refNum >= 0) && (CmpStr(type,"TEXT")==0) )  // also "text", etc.
      String line1
      FReadLine refNum, line1 // Read the line (and carriage return)
      Print line1       // Print line in the history window.
   endif
   return 0             // don't prevent MIME-TSV from displaying
End
```

See Also

BeforeFileOpenHook and **SetIgorHook**.

BeforeDebuggerOpensHook

BeforeDebuggerOpensHook(*errorInRoutineStr, stoppedByBreakpoint*)

BeforeDebuggerOpensHook is a user-defined function that Igor calls when the debugger window is about to be opened, whether by hitting a breakpoint or when Debug on Error is enabled.

BeforeDebuggerOpensHook can be used to prevent the debugger window opening for certain error codes or in selected user-defined functions when Debug on Error is enabled. This is a feature for advanced programmers only. Most programmers will not need it.

This hook does not work well for macros or procs, because their runtime errors don't automatically open the debugger, but instead present an error dialog from which the user manually enters the debugger by clicking the Debug button.

Parameters

String *errorInRoutineStr* contains the name of the routine (function or macro) the debugger will be stopping in as a fully-qualified name, comprised of at least "`ModuleName#RoutineName`", suitable for use with FunctionInfo.

If the routine is in a regular module procedure window (see **Regular Modules** on page IV-212), *errorInRoutineStr* will be a triple name such as "`MyIM#MyModule#MyFunction`".

Variable *stoppedByBreakpoint* is 0 if the debugger is about to be shown because of Debug on Error, or non-zero if the debugger encountered a user-set breakpoint (see **Setting Breakpoints** on page IV-185).

If a breakpoint exists at the line where an error caused the debugger to appear, *stoppedByBreakpoint* will be non-zero, even though the cause was Debug on Error.

Details

If BeforeDebuggerOpensHook returns 0 or NaN (or doesn't return a value), the debugger window is opened normally.

If it returns 1, the debugger window is not shown and program execution continues.

All other return values are reserved for future use.

Example

The following hypothetical example:

1. Prevents breakpoints from bringing up the debugger, unless DEBUGGING is defined.
2. Prints the name of the routine with the error, and the error message.
3. Beeps before the debugger appears.

```
Function ProvokeDebuggerInFunction()
   DebuggerOptions enable=1, debugOnError=1     // Enable debug on error

   ProvokeDebugger()
End

Function ProvokeDebugger()
   Variable var=0 // Put a breakpoint here.
                  // Without a #define DEBUGGING, the breakpoint is skipped.
   Make/O $""     // Cause an error
   Print "Back from bad Make command in function"
End

static Function BeforeDebuggerOpensHook(pathToErrorFunction,isUserBreakpoint)
   String pathToErrorFunction
   Variable isUserBreakpoint

   #ifndef DEBUGGING
      if( isUserBreakpoint )
         return 1 // Ignore user breakpoints we forgot to clear.
                  // Don't use this during development!
      endif
   #endif

   Print "stackCrawl = ", GetRTStackInfo(0)
   Print "FunctionInfo = ", FunctionInfo(pathToErrorFunction)

   // Don't clear errors unless you're preventing the debugger from appearing
   Variable clearErrors= 0
   Variable rtErr= GetRTError(clearErrors)   // Get the error #

   Variable substitutionOption= exists(pathToErrorFunction)== 3 ? 3 : 2
   String errorMessage= GetErrMessage(rtErr,substitutionOption)

   Beep     // Audible cue that the debugger is showing up!

   Print "Error \""+errorMessage+"\" in "+pathToErrorFunction+"

   return 0 // Return 0 to show the debugger; an unexpected error occurred.
End

•ProvokeDebuggerInFunction()  // Execute this in the command line
  stackCrawl =
      ProvokeDebuggerInFunction;ProvokeDebugger;BeforeDebuggerOpensHook;
  FunctionInfo =
      NAME:ProvokeDebugger;PROCWIN:Procedure;MODULE:;INDEPENDENTMODULE:;...
```

```
Error "Expected name" in ProcGlobal#ProvokeDebugger
Back from bad Make command in function
```

See Also
SetWindow, **SetIgorHook**, and **User-Defined Hook Functions** on page IV-251

Static Functions on page IV-83, **Regular Modules** on page IV-212, **Independent Modules** on page IV-214

FunctionInfo, **GetRTStackInfo**, **GetRTError**, **GetRTErrMessage**

Conditional Compilation on page IV-86

AfterWindowCreatedHook

AfterWindowCreatedHook(*windowNameStr, winType*)
AfterWindowCreatedHook is a user-defined function that Igor calls when a target window is first created.

AfterWindowCreatedHook can be used to set a window hook on target windows created by the user or by other procedures.

Parameters
String *windowNameStr* contains the name of the created window.

Variable *winType* is the type of the window, the same value as returned by **WinType**.

Details
"Target windows" are graphs, tables, layout, panels, and notebook windows.

AfterWindowCreatedHook is not called when an Igor experiment is being opened.

Igor ignores the value returned by AfterWindowCreatedHook.

See Also
SetWindow, **SetIgorHook**, and **User-Defined Hook Functions** on page IV-251.

BeforeExperimentSaveHook

BeforeExperimentSaveHook(*refNum, fileNameStr, pathNameStr, fileTypeStr, fileCreatorStr, fileKind*)
BeforeExperimentSaveHook is a user-defined function that Igor calls when an experiment is about to be saved by Igor.

Igor ignores the value returned by BeforeExperimentSaveHook.

Parameters
For a full explanation of the parameters, see **AfterFileOpenHook** on page IV-253.

Variable *refNum* is -1. The experiment file is not open.

String *fileNameStr* contains the name of the experiment file (including any extension).

String *pathNameStr* contains the name of the symbolic path.

String *fileTypeStr* contains the file type code.

String *fileCreatorStr* contains the file creator code.

Variable *fileKind* is a number that identifies what kind of file Igor will be saving:

Kind of File	*fileKind*
Igor Experiment, packed*	1
Igor Experiment, unpacked*	2

* Including stationery experiment files.

Details
You can determine the full directory and file path of the experiment by calling the **PathInfo** operation with $*pathNameStr*.

Example

This (somewhat frivolous) example prints the full file path of the about-to-be-saved experiment to the history area, and deletes all unused symbolic paths.

```
#pragma rtGlobals=1        // treat S_path as local string variable

Function BeforeExperimentSaveHook(rN,fileName,path,type,creator,kind)
    Variable rN,kind
    String fileName,path,type,creator

    PathInfo $path             // puts path value into (local) S_path
    Printf "Saved “%s” experiment\r",S_path+fileName

    KillPath/A/Z               // Delete all unneeded symbolic paths
End
```

See Also

The **SetIgorHook** operation.

BeforeFileOpenHook

BeforeFileOpenHook(*refNum, fileNameStr, pathNameStr, fileTypeStr, fileCreatorStr, fileKind*)

BeforeFileOpenHook is a user-defined function that Igor calls when a file *is about to be opened* by Igor because the user dragged it onto the Igor icon, double-clicked it, or because Igor received an “open” Apple event from another program (such a Web browser).

Note: BeforeFileOpenHook is not called when the Open File or Load Waves menu is selected.

The parameters contain information about the file, which has already been opened for read-only access.

The value returned by BeforeFileOpenHook informs Igor whether the user-defined function handled the open and therefore Igor should not perform its default action. In some cases, this return value is ignored, and Igor performs the default action anyway.

Parameters

(For a full explanation of the parameters, see **AfterFileOpenHook** on page IV-253.)

Variable *refNum* is the file reference number.

String *fileNameStr* contains the name of the file (including any extension).

String *pathNameStr* contains the name of the symbolic path.

String *fileTypeStr* contains the file type code.

String *fileCreatorStr* contains the file creator code.

Variable *fileKind* is a number that identifies what kind of file Igor thinks it is:

Kind of File	*fileKind*	Default Action, if Any
Unknown	0	
Igor Experiment, packed *	1	(Hook not called)
Igor Experiment, unpacked*	2	(Hook not called)
Igor XOP	3	
Igor Binary file	4	Data loaded
Igor Text (data and commands)	5	Data loaded, commands executed
Text, no numbers detected in first two lines	6	Opened as unformatted notebook
General Numeric text (no tabs)	7	Data loaded as general text
Numeric text Tab-Separated-Values	8	Data loaded as delimited text
Numeric text Tab-Separated-Values, MIME	9	Display loaded data in a new table and a new graph.

Kind of File	*fileKind*	Default Action, if Any
Text, with tabs	10	Opened as unformatted notebook
Igor Notebook (unformatted or formatted)	11	Opened as notebook
Igor Procedure	12	*Always* opened as procedure file
Igor Help	13	*Always* opened as help file

* Including stationery experiment files.

Details

BeforeFileOpenHook must return 1 if Igor is not to take action on the file (it won't be opened), or 0 if Igor is permitted to take action on the file. Igor ignores the return value for *fileKind* values of 3, 12, and 13. The hook function is not called for Igor experiments (*fileKind* values of 1 and 2).

Igor always closes the file when the user-defined function returns, and *refNum* becomes invalid (don't store the value of *refNum* in a global for use by other routines, since the file it refers to has been closed).

Example

This example checks the first line of the file about to be opened to determine whether it has a special, presumably user-specific, format. If it does, then LoadMyFile (another user-defined function) is called to load it. LoadMyFile presumably loads this custom data file, and returns 1 if it succeeded. If it returns 0 then Igor will open it using the Default Action from the above table.

Another example can be found in the discussion in **User-Defined Hook Functions** on page IV-251.

```
Function BeforeFileOpenHook(refNum,fileName,path,type,creator,kind)
   Variable refNum,kind
   String fileName,path,type,creator

   Variable handledOpen=0
   if( CmpStr(type,"TEXT")==0 )              // text files only
      String line1
      FReadLine refNum, line1 // First line (and carriage return)
      if( CmpStr(line1[0,4],"XYZZY") == 0 )  // My special file
         FSetPos refNum, 0             // rewind to start of file
         handledOpen= LoadMyFile(refNum)  // returns 1 if loaded OK
      endif
   endif
   return handledOpen   // 1 tells Igor not to open the file
End
```

See Also

AfterFileOpenHook and **SetIgorHook**.

IgorBeforeNewHook

IgorBeforeNewHook(*igorApplicationNameStr*)

IgorBeforeNewHook is a user-defined function that Igor calls before a new experiment is opened in response to the New Experiment, Revert Experiment, or Open Experiment menu items in the File menu.

You can use IgorBeforeNewHook to clean up the current experiment, or to avoid losing unsaved data even if the user chooses to not save the current experiment.

Igor ignores the value returned by IgorBeforeNewHook.

Parameters

igorApplicationNameStr contains the name of the currently running Igor Pro application.

See Also

IgorStartOrNewHook and **SetIgorHook**.

IgorBeforeQuitHook

IgorBeforeQuitHook(*unsavedExp*,*unsavedNotebooks*,*unsavedProcedures*)

IgorBeforeQuitHook is a user-defined function that Igor calls just before Igor is about to quit (before any save-related dialogs have been presented).

Parameters

Variable *unsavedExp* is 0 if the experiment is saved, 1 if unsaved.

Variable *unsavedNotebooks* is the count of unsaved notebooks.

Variable *unsavedProcedures* is the count of unsaved procedures.

Note: The save state of packed procedure and notebook files is part of *unsavedExp*, not *unsavedNotebooks* or *unsavedProcedures*. This applies to adopted procedure and notebook files and new procedure and notebook windows that have never been saved.

Details

IgorBeforeQuitHook will normally return 0. In these cases it will present the "Do you want to save" dialogs, and if the user approves, it will call IgorQuitHook.

If IgorBeforeQuitHook returns 1, then the current experiment, notebooks, or procedures will not be saved; no dialogs will be presented to the user, and it will not call IgorQuitHook.

See Also

IgorQuitHook and **SetIgorHook**.

IgorMenuHook

IgorMenuHook(*isSelection, menuStr, itemStr, itemNo, topWindowNameStr, wType*)

IgorMenuHook is a user-defined function that Igor calls just before and just after menu selection (whether by mouse or keyboard).

Parameters

Variable *isSelection* is 0 before a menu item has been selected, 1 when a menu item has been selected.

String *menuStr* is the name of the selected menu (in English and as used by SetIgorMenuMode). *menuStr* is "" when *isSelection* is 0.

String *itemStr* is the name of the selected menu item or "" when *isSelection* is 0.

Variable *itemNo* is the item number of the selected menu item; 1 is the first item in the selected menu. *itemNo* is 0 when *isSelection* is 0.

String *topWindowNameStr* is the name of the top window (the window to which window-specific menu commands like Copy and Paste apply).

Not all windows have names, so *topWindowNameStr* is set specially in those cases:

Window	***topWindowNameStr***
graph, panel, notebook, layout, table	Window name. See the Window Control Dialog and **DoWindow** operation (page V-122) about window names.
command/history window	"kwCmdHist" (as used for **GetWindow**).
procedure window	The window title as shown in the window's title bar. The standard procedure window is "Procedure".
Igor Extensions ("XOP") window	The window title as shown in the window's title bar.

Variable *wType* identifies the kind of window that *topWindowNameStr* names. It returns the same values as the **WinType** function (see page V-744). For procedure windows *wType* is 8, for the command/history window it is -1, and for XOP windows or unknown windows *wType* is 0.

Details

IgorMenuHook is called with *isSelection* set to 0 after all the menus have been enabled and before a mouse click or Command-key (*Macintosh*) or Ctrl+key (*Windows*) is handled.

The return value should normally be 0. If the return value is nonzero (1 is usual) then the top window's hook function (see **SetWindow** operation on page V-569) is not called for the enablemenu event.

IgorMenuHook is called with *isSelection* set to 1 after the menu has been selected and before Igor has acted on the selection.

If the hook function returns 0, Igor proceeds to call the top window's hook function for the menu event. (If the window hook function is present and returns nonzero Igor ignores the menu selection, otherwise Igor handles the menu selection normally.)

If the hook function returns nonzero (1 is again usual), Igor does not call the remaining hook functions and Igor ignores the menu selection.

Menu Event Details

Menu building, enabling, and selection by Igor, user menus, window hooks and IgorMenuHook are sequenced this way:

Menu building and enabling are performed in this order:

1. Igor updates built-in menus according to the frontmost and the target window.
2. Dynamic user-defined menus are updated.
3. IgorMenuHook(0, …) is called. If IgorMenuHook returns nonzero (which is not recommended), step 4 is skipped.
4. The top window's window hook functions (see **SetWindow** operation on page V-569) are called with the enablemenu event. The return value is ignored.
5. Igor extensions update their menus and their items in built-in menus.

Menu selections are handled in this order:

1. Igor extensions may handle the menu selection. If the selection was handled, the following steps are skipped.
2. If the selected menu item is not one of Igor's built-in menus, the steps 3 and 4 are skipped.
3. The top window's window hook(s) is (are) called with the menu event. If a hook returns nonzero the remaining window hooks and the following steps are skipped.
4. IgorMenuHook(1, …) is called. If IgorMenuHook returns nonzero, the following steps are skipped.
5. If a user-defined menu was chosen, Igor executes the associated command, and the remaining step is skipped.
6. Igor handles the selection of a built-in menu item.

Example

This user hook function invokes the Export Graphics menu item when Command-C (*Macintosh*) or Ctrl+C (*Windows*) is selected for all graphs, preventing Igor from performing the usual Copy.

```
Function IgorMenuHook(isSel, menuStr, itemStr, itemNo, topWindowName, wt)
    Variable isSel
    String menuStr, itemStr
    Variable itemNo
    String topWindowName
    Variable wt

    Variable handled= 0
    if( Cmpstr(menuStr,"Edit") == 0 && CmpStr(itemStr,"Copy") == 0 )
        if( wt == 1 )                                   // graph
            // DoIgorMenu would cause recursion, so we defer execution
            Execute/P/Q/Z "DoIgorMenu \"Edit\", \"Export Graphics\""
            handled= 1
        endif
    endif

    return handled
End
```

See Also

SetWindow, **Execute**, and **SetIgorHook**.

IgorQuitHook

IgorQuitHook(*igorApplicationNameStr*)

IgorQuitHook is a user-defined function that Igor calls when Igor is about to quit.

The value returned by IgorQuitHook is ignored.

Parameters

String *igorApplicationNameStr* contains the name of the currently running Igor Pro application (including the .exe extension under Windows).

Details

You can determine the full directory and file path of the Igor application by calling the **PathInfo** operation with the Igor path name. See the example in **IgorStartOrNewHook** on page IV-263.

See Also

IgorBeforeQuitHook and **SetIgorHook**.

IgorStartOrNewHook

IgorStartOrNewHook(*igorApplicationNameStr*)

IgorStartOrNewHook is a user-defined function that Igor calls when starting up and when creating a new experiment. It is also called if Igor is launched as a result of double-clicking a saved Igor experiment.

Igor ignores the value returned by IgorStartOrNewHook.

Parameters

String *igorApplicationNameStr* contains the name of the currently running Igor Pro application (including the .exe extension under Windows).

Details

You can determine the full directory and file path of the Igor application by calling the **PathInfo** operation with the Igor path name.

Example

This example prints the full path of Igor application and sets the annotation halo size to zero whenever Igor starts up or creates a new experiment:

```
#pragma rtGlobals=1        // treat S_path as local string variable

Function IgorStartOrNewHook(igorApplicationNameStr)
   String igorApplicationNameStr

   Variable/G root:V_TBBufZone= 0  // See General Annotation Properties
                                   // on page III-48
   PathInfo Igor              // puts path value into (local) S_path
   printf "“%s” (re)starting\r", S_path + igorApplicationNameStr
End
```

See Also

IgorBeforeNewHook and **SetIgorHook**.

Static Hook Functions

To allow for multiple procedure files to define the same predefined hook function, you should declare your hook function static. For example:

```
static Function IgorStartOrNewHook(igorApplicationNameStr)
   String igorApplicationNameStr
```

The use of the static keyword makes the function private to the procedure file containing it and allows other procedure files to have their own static function with the same name.

Igor calls static hook functions after the **SetIgorHook** (see page V-560) functions are called. The static hook functions themselves are called in the order in which their procedure file was opened. You should not rely on any execution order among the static hook functions. However, any hook function which returns a

nonzero result prevents remaining hook functions from being called and prevents Igor from performing it's usual processing of the hook event. In most cases hook functions should exercise caution in returning any value other than 0. For hook functions only, returning a NaN or failing to return a value (which returns a NaN) is considered the same as returning 0.

The **IgorStartOrNewHook** (see page IV-263) hook function is especially useful to initialize a related set of procedures packaged together in an auxiliary procedure file.

Window Hook Functions

A window hook function is a user-defined function that receives notifications of events that occur in a specific window. Your window hook function can detect and respond to events of interest. You can then allow Igor to also process the event or inform Igor that you have handled it.

This section discusses window hook functions that apply to a specific window. For information on general events hooks, see **User-Defined Hook Functions** on page IV-251.

To handle window events, you first write a window hook function and then use the **SetWindow** operation to install the hook on a particular window. This example shows how you would detect arrow key events in a particular window. To try it, paste the code below into the procedure window and then execute DemoWindowHook():

```
Function MyWindowHook(s)
   STRUCT WMWinHookStruct &s

   Variable hookResult = 0 // 0 if we do not handle event, 1 if we handle it.

   switch(s.eventCode)
      case 11:              // Keyboard event
         switch (s.keycode)
            case 28:
               Print "Left arrow key pressed."
               hookResult = 1
               break
            case 29:
               Print "Right arrow key pressed."
               hookResult = 1
               break
            case 30:
               Print "Up arrow key pressed."
               hookResult = 1
               break
            case 31:
               Print "Down arrow key pressed."
               hookResult = 1
               break
         endswitch
         break
   endswitch

   return hookResult // If non-zero, we handled event and Igor will ignore it.
End

Function DemoWindowHook()
   DoWindow/F DemoGraph    // Does graph exist?
   if (V_flag == 0)
      Display /N=DemoGraph // Create graph
      SetWindow DemoGraph, hook(MyHook)=MyWindowHook  // Install window hook
   endif
End
```

The window hook function receives a WMWinHookStruct structure as a parameter. WMWinHookStruct is a built-in structure that contains all of the information you might need to respond to an event. One of its fields, the eventCode field, specifies what kind of event occurred.

If your hook function returns 1, this tells Igor that you handled the event and Igor does not handle it. If your hook function returns 0, this tells Igor that you did not handle the event, so Igor does handle it.

This example uses a named window hook. In this case the name is MyHook. More than one procedure file can install a hook on a given window. The purpose of the name is to allow a package to install and remove its own hook function without disturbing the hook functions of other packages. Choose a hook name that is unlikely to conflict with other hook names.

Earlier versions of Igor supported only one unnamed hook function. This meant that only one package could hook any particular window. Unnamed hook functions are still supported for backward compatibility but new code should always use named hook functions.

Window Hooks and Subwindows

Igor calls window hook functions for top-level windows only, not for subwindows. If you want to hook a subwindow, you must set the hook on the top-level window. In the hook function, test to see if the subwindow is active. For example, this code, at the start of a window hook function, insures that the hook runs only if a subwindow named G0 in a panel named Panel0 is active.

```
GetWindow $s.winName activeSW
String activeSubwindow = S_value
if (CmpStr(activeSubwindow, "Panel0#G0") != 0)
   return 0
endif
```

Named Window Hook Functions

A named window hook function takes one parameter - a `WMWinHookStruct` structure. This built-in structure provides your function with information about the status of various window events.

The named window hook function has this format:

```
Function MyWindowHook(s)
   STRUCT WMWinHookStruct &s

   Variable hookResult = 0

   switch(s.eventCode)
      case 0:              // Activate
         // Handle activate
         break

      case 1:              // Deactivate
         // Handle deactivate
         break

      // And so on . . .
   endswitch

   return hookResult       // 0 if nothing done, else 1
End
```

If you handle a particular event and you want Igor to ignore it, return 1 from the hook function.

Named Window Hook Events

Here are the events passed to a named window hook function:

eventCode	eventName	Notes
0	"Activate"	
1	"Deactivate"	
2	"Kill"	
3	"Mousedown"	
4	"Mousemoved"	
5	"Mouseup"	
6	"Resize"	
7	"Cursormoved"	See **Cursors — Moving Cursor Calls Function** on page IV-294.
8	"Modified"	A modification to the window has been made. This is sent to graph and notebook windows only. It is an error to try to kill a notebook window from the window hook during the modified event.
9	"Enablemenu"	
10	"Menu"	
11	"Keyboard"	
12	"moved"	
13	"renamed"	
14	"subwindowKill"	One of the window's subwindows is about to be killed.
15	"hide"	The window or one of its subwindows is about to be hidden.
16	"show"	The window or one of its subwindows is about to be unhidden.
17	"killVote"	Window is about to be killed. Return 2 to prevent the window from being killed, otherwise return 0. **Note**: Don't delete data structures during this event, use killVote only to decide whether the window kill should actually happen. Delete data structures in the kill event. See **Window Hook Deactivate, Kill, Show and Hide Events** on page IV-269.
18	"showTools"	
19	"hideTools"	
20	"showInfo"	
21	"hideInfo"	
22	"mouseWheel"	
23	"spinUpdate"	This event is sent only to windows marked via DoUpdate/E=1 as progress windows. It is sent when Igor spins the beachball cursor. See **Progress Windows** on page IV-134 for details.

WMWinHookStruct

The WMWinHookStruct structure has members as described in the following tables:

Base WMWinHookStruct Structure Members

Member	Description
char winName[MAX_PATH_LENGTH+1]	hcSpec of the affected (sub)window.
STRUCT Rect winRect	Local coordinates of the affected (sub)window.
STRUCT Point mouseLoc	Mouse location.
double ticks	Tick count when event happened.
Int32 eventCode	See see **eventCode** table on page IV-266.
char eventName[31+1]	Name-equivalent of eventCode, see **eventCode** table on page IV-266. Added in Igor 5.03.
Int32 eventMod	Bitfield of modifiers. See description for MODIFIERS:*flags*.

Members of WMWinHookStruct Structure Used with menu Code

Member	Description
char menuName[255+1]	Name of menu (in English) as used by **SetIgorMenuMode**.
char menuItem[255+1]	Text of the menu item as used by SetIgorMenuMode

Members of WMWinHookStruct Structure Used with keyboard Code

Member	Description
Int32 keycode	ASCII value of key struck. Function keys are not available but navigation keys are translated to specific values and will be the same on Macintosh and Windows.

Members of WMWinHookStruct Structure Used with cursormoved Code

Member	Description
char traceName[MAX_OBJ_NAME+1]	The name of the trace or image to which the moved cursor is attached or which supplies the X (and Y) values. Can be "" if the cursor is free.
char cursorName[2]	Cursor name A through J.
double pointNumber	Point number of the trace or the X (row) point number of the image where the cursor is attached. If the cursor is "free", pointNumber is actually the fractional relative *xValue* as used in the **Cursor**/F/P command.
double yPointNumber	Valid only when the cursor is attached to a two-dimensional item such as an image, contour, or waterfall plot, or when the cursor is free.

Members of `WMWinHookStruct` Structure Used with `cursormoved` Code

Member	Description
	If attached to an image, contour, or waterfall plot, `yPointNumber` is the Y (column) point number of the image where the cursor is attached. If the cursor is "free", `yPointNumber` is actually the fractional relative *yValue* as used in the **`Cursor`**`/F/P` command.
`Int32 isFree`	Has value of 1 if the cursor is not attached to anything, or value of 0 if it is attached to a trace, image, contour, or waterfall.

Members of `WMWinHookStruct` Structure Used with `mouseWheel` Code

Member	Description
`double wheelDy`	Vertical lines to scroll. Typically +1 or -1.
`double wheelDx`	Horizontal lines to scroll. Typically +1 or -1. On Windows, horizontal mouse wheel requires Vista.

Members of `WMWinHookStruct` Used with `renamed` Code

Member	Description
`char oldWinName[MAX_OBJ_NAME+1]`	Old name of the window or subwindow. Not the absolute path *hcSpec*, just the name.

User-Modifiable Members of `WMWinHookStruct` Structure

Member	Description
`Int32 doSetCursor`	Set TRUE to change cursor to that specified by `cursorCode`.
`Int32 cursorCode`	Standard cursor as set by hookcursor=*number*.

Set Cursor Hook Example

This example uses a named hook to set the cursor as the mouse is moved horizontally across the bottom half of the window:

```
static constant kUHC_mousemoved=4

Function MyWinHook(s)
   STRUCT WMWinHookStruct &s
   Variable rval= 0
   switch(s.eventCode)
      case kUHC_mousemoved:
         if( s.mouseLoc.v > ((s.winRect.top+s.winRect.bottom)/2) )
            s.doSetCursor= 1
            s.cursorCode=round(27*(s.mouseLoc.h-s.winRect.left) /
                                 (s.winRect.right-s.winRect.left))
            rval= 1           // we have taken over this event
         endif
         break
   EndSwitch
```

```
   return rval
End

Function DemoSetCursorHook()
   Make/O jack=x
   Display jack
   SetWindow kwTopWin,hook(testhook)= MyWinHook
End
```

Panel Done Button Example

This example uses a window hook and button action procedure to implement a panel dialog with a Done button such that the panel can't be closed by clicking the panel's close widget, but can be closed by the Done button's action procedure:

```
Proc ShowDialog()
   PauseUpdate; Silent 1         // building window...
   NewPanel/N=Dialog/W=(225,105,525,305) as "Dialog"
   Button done,pos={119,150},size={50,20},title="Done"
   Button done,proc=DialogDoneButtonProc
   TitleBox warning,pos={131,83},size={20,20},title=""
   TitleBox warning,anchor=MC,fColor=(65535,16385,16385)
   SetWindow Dialog hook(dlog)=DialogHook, hookevents=2
EndMacro

Function DialogHook(s)
   STRUCT WMWinHookStruct &s
   Variable statusCode= 0
      strswitch( s.eventName )
      case "killVote":
         TitleBox warning win=$s.winName, title="Press the Done button!"
         Beep
         statusCode=2                  // prevent panel from being killed.
         break
      case "mousemoved":               // to reset the warning
         TitleBox warning win=$s.winName, title=""
         break
   endswitch
   return statusCode
End

Function DialogDoneButtonProc(ba) : ButtonControl
   STRUCT WMButtonAction &ba
   switch( ba.eventCode )
      case 2:              // mouse up
         // turn off the named window hook
         SetWindow $ba.win hook(dlog)=$""
         // kill the window AFTER this routine returns
         Execute/P/Q/Z "DoWindow/K "+ba.win
         break
   endswitch
   return 0
End
```

Window Hook Deactivate, Kill, Show and Hide Events

The actions caused by these events (eventCode 2, 14, 15, 16 and 17) potentially affect multiple subwindows.

If you kill a subwindow, the root window's hook function(s) receives a subwindowKill event for that subwindow and any child subwindows.

If you kill a root window, the root window's hook function(s) receives a subwindowKill event for each child subwindow, and then the root window's hook function(s) receive a kill event.

Likewise, hiding and showing windows can result in subwindows being hidden or shown. In each case, the window hook function receives a hide or show event for each affected window or subwindow.

The winName member of WMWinHookStruct will be set to the full subwindow path of the subwindow that is affected.

Exterior subwindows are a special case because they are subwindows, but you can attach a hook function to an exterior subwindow. The hook function attached to the root window does not receive events affecting exterior subwindows. To handle subwindowKill, hide, or show events when an exterior subwindow is killed, hidden, or shown as a result of killing, hiding, or showing its parent window, you must have a hook function attached to the exterior subwindow.

As a further subtlety, the hook function(s) attached to an exterior subwindow will receive a subwindowKill event if the exterior subwindow is killed as a result of killing the parent window. But it will receive a regular kill event if it is killed directly. Normal subwindows always receive only subwindowKill events.

The kill-related events are sent in this order when a window or subwindow is killed:

1. A killVote event is sent to the root window's hook function(s). If any hook function returns 2, no further events are generated and the window is not killed.
2. If the window is not a subwindow and wasn't created with /K=*k* the standard window close dialog appears. If the close is cancelled, the window is not killed, the window will receive an activate event when the dialog is dismissed, and no further events are generated. Otherwise, proceed to step 3.
3. If the window being killed has subwindows, starting from the bottom-most subwindow and working back toward the window being killed:

3a. If the subwindow is a graph or panel, action procedures for controls contained in the subwindow are called with event -1, "control being killed".

3b. The root window's hook function(s) receive a subwindowKill event for the subwindow. If any hook function returns 1, no further subwindow hook events or control being killed events are sent, but the window killing process continues.

Steps 3a and 3b are repeated for each subwindow until the window or subwindow being killed is reached.

4. If the killed window is a root window, a kill event is sent to the root window's hook function(s). If any hook function returns 2, no further events are generated and the window is not killed. This method of preventing a window from closing is to be avoided: use the killVote event or the window-equivalent of `NewPanel/K=2`.

There are several ways to prevent a window being killed. You might want to do this in order to enforce use of a Done or Do It button, or to prevent killing a control panel while some hardware action is taking place.

The best method is to use /K=2 when creating the window (see **Display** or **NewPanel**). Then the only way to kill the window is via the DoWindow/K command, or KillWindow command. In general, you would provide a button that kills the window after checking for any conditions that would prevent it.

The KillVote event is more flexible but harder to use. It gives your code a chance to decide whether or not killing is allowed. This means the user can close and kill the window with the window close box when it is allowed.

Returning 2 for the window kill event is not recommended. If you have old code that uses this method, we strongly recommend changing it to return 2 for the killVote event. New code should never return 2 for the kill event.

Events 14-17 were added to Igor Pro version 6.02.

Unnamed Window Hook Functions

Unnamed window hook functions are supported for backward compatibility only. New code should use named window hook functions. See **Named Window Hook Functions** on page IV-265.

Each window can have one unnamed hook function. You designate a function as the unnamed window hook function using the **SetWindow** operation with the hook keyword.

The unnamed hook function is called when various window events take place. The reason for the hook function call is stored as an event code in the hook function's infoStr parameter.

The hook function is not called during experiment creation or load time so as to prevent the hook function from failing because the experiment is not fully recreated.

The hook function has the following syntax:

```
Function procName(infoStr)
   String infoStr
   String event= StringByKey("EVENT",infoStr)
   …
   return statusCode    // 0 if nothing done, else 1
End
```

`infoStr` is a string containing a semicolon-separated list of ***key:value*** pairs:

Key	Value
EVENT	*eventKey* See list of *eventKey* values below.
HCSPEC	Absolute path of the window or subwindow. See **Subwindow Command Concepts** on page III-95.
`MODIFIERS`	Bit flags as follows: Bit 0: Set if mouse button is down. Bit 1: Set if Shift is down. Bit 2: Set if Option (*Macintosh*) or Alt (*Windows*) is down. Bit 3: Set if Command (*Macintosh*) or Ctrl (*Windows*) is down. Bit 4: Contextual menu click: right-click or Control-click (*Macintosh*), or right-click (*Windows*). See **Setting Bit Parameters** on page IV-12 for details about bit settings.
`OLDWINDOW`	Previous name of the window or subwindow (for renamed event). Not the old absolute path *hcSpec*, just the name. WINDOW and HCSPEC contain the new name and new *hcSpec*.
`WINDOW`	Name of the window.

The value accompanying the EVENT keyword is one of the following:

eventKey	Meaning
`activate`	Window has just been activated.
`copy`	Copy menu item has been selected.
`cursormoved`	A graph cursor was moved. This event is sent only if bit 2 of the **SetWindow** operation hookevents flag is set.
`deactivate`	Window has just been deactivated.

eventKey	Meaning
enablemenu	Menus are being built and enabled.
hide	Window or subwindows about to be hidden.
hideInfo	The window info box or window has just been hidden by **HideInfo**.
hideTools	The window tool palette or window has just been hidden by **HideTools**.
kill	Window is being killed. Returning a value of 2 as the hook function result prevents Igor from killing the window.
killVote	Window is about to be killed. Return 2 to prevent that, otherwise return 0. See **Window Hook Deactivate, Kill, Show and Hide Events** on page IV-269.
menu	A built-in menu item has been selected.
modified	A modification to the window has been made. This is sent to graph and notebook windows only. It is an error to try to kill a notebook window from the window hook during the modified event.
mousedown	Mouse button was clicked. This event is sent only if bit 0 of the **SetWindow** operation hookevents flag is set.
mousemoved	The mouse moved. This event is sent only if bit 1 of the **SetWindow** operation hookevents flag is set.
mouseup	Mouse button was released. This event is sent only if bit 0 of the **SetWindow** operation hookevents flag is set.
moved	Window has just been moved.
renamed	Window has just been renamed. The previous name is available under the OLDWINDOW key.
resize	Window has just been resized.
show	Window or subwindow is about to be unhidden.
showInfo	The window info box or window has just been shown by **ShowInfo**.
showTools	The window tool palette or window has just been shown by **ShowTools**.
subwindowKill	One of the window's subwindows is about to be killed.

The modified event is issued only when a graph updates (See **DoUpdate**, **PauseUpdate**, and **ResumeUpdate**). Most changes to the graph are reported by the `modified` event, but not all: changing an annotation will not trigger the event, nor will adding, removing, or modifying a control or showing or hiding the drawing tools while using the /A flag. The modified event is not sent while a trace is being dragged or when the values of a trace's wave change (unless one the trace's axes is autoscaled). However, changing an axis range or indeed changing almost anything about axes or showing or hiding the info pane will send the modified event (only one event per graph update). When in doubt, use a print statement to determine when the event is sent.

If mouse events are enabled then the following key:value pairs will also be present in `infoStr`:

Key	Value
MOUSEX	X coordinate in pixels of the mouse.
MOUSEY	Y coordinate in pixels of the mouse.
TICKS	Time event happened.

Note that a mouseup event may or may not correspond to a previous mousedown. If the user clicks in the window, drags out and releases the button then the mouseup event will be missing. If the user clicks in another window, drags into this one and then releases then a mouseup will be sent that had no previous mousedown.

In the case of mousedown or mousemoved messages, a nonzero return value will skip normal processing of the message. This is most useful with mousedown.

The `cursormoved` event is not reported if Option (*Macintosh*) or Alt (*Windows*) is held down.

If the `cursormoved` event is enabled then the following key:value pairs will also be present in `infoStr`:

Key	Value
`CURSOR`	Name of the cursor that moved (A through J).
`TNAME`	Name of the trace the cursor is attached to (invalid if `ISFREE=1`).
`ISFREE`	1 if the cursor is "free" (not attached to a trace), 0 if it is attached to a trace or image.
`POINT`	Point number of the trace if not a free cursor. If the cursor is attached to an image, value is the row number of the image. If a free cursor, value is the fraction of the plot width, 0 being the left edge of the plot area, and 1 being the right edge.
`YPOINT`	Column number if the cursor is attached to an image, NaN if attached to a trace. If a free cursor, value is the fraction of the plot height, 0 being the top edge, and 1 being the bottom edge.

When the a menu event is reported then the following key:value pairs will also be present in `infoStr`:

Key	Value
`MENUNAME`	Name of menu (in English) as used by **SetIgorMenuMode**.
`MENUITEM`	Text of menu item as used by **SetIgorMenuMode**.

The `enablemenu` event does not pass `MENUNAME` or `MENUITEM`.

The `menu` and `enablemenu` messages are not sent when drawing tools are in use in a graph or layout or when waves are being edited in a graph.

Returning a value of 0 for the `enablemenu` message is recommended, though the return value is (currently) ignored.

You can use the **SetIgorMenuMode** operation to alter the enable state of Igor's built-in menus in a way you find appropriate for the window. If you do this, usually you will also handle the menu message and perform your idea of an appropriate action.

Note: Dynamic user-defined menus (see **Dynamic Menu Items** on page IV-109) are built and enabled by using string functions in the menu definitions.

Returning a value of 0 for any menu message allows Igor to perform the normal action. Returning any other value (1 is commonly used) tells Igor to skip performing the normal action.

See the user function description with **IgorMenuHook** on page IV-261 for details on the sequence of menu building, enabling, and handling.

Custom Marker Hook Functions

A custom marker hook function takes one parameter - a WMMarkerHookStruct structure. This structure will provide your function with information you need to draw a marker. See **Structures in Functions** on page IV-78 for background information on structures.

The function prototype used with a custom marker hook has the format:

```
Function MyMarkerHook(s)
   STRUCT WMMarkerHookStruct &s
   <code to draw marker>
   ...
   return statusCode       // 0 if nothing done, else 1
End
```

Your function can use the DrawXXX operations to create the marker. The function is called each time the marker is drawn and should not do anything other than drawing the marker. The function should return 1 if it handled the marker or 0 if not. The marker range can be any positive integers less than 1000 and can overlap built-in marker numbers.

WMMarkerHookStruct

The WMMarkerHookStruct structure has the following members:

WMMarkerHookStruct Structure Members

Member	Description
`Int32 usage`	0= normal draw, 1= legend draw (others reserved).
`Int32 marker`	Marker number minus start (i.e., starts from zero).
`float x,y`	Location of desired center of marker
`float size`	Half width/height of marker
`Int32 opaque`	1 if marker should be opaque
`float penThick`	Stroke width
`STRUCT RGBColor mrkRGB`	Fill color
`STRUCT RGBColor eraseRGB`	Background color
`STRUCT RGBColor penRG`	Stroke color

When your marker function is called, the pen thickness and colors of the drawing environment of the target window are already set consistent with the penThick, mrkRGB, eraseRGB and penRGB members.

Marker Hook Example

Here is an example that draws audiology symbols:

```
Function AudiologyMarkerProc(s)
   STRUCT WMMarkerHookStruct &s

   if( s.marker > 3 )
      return 0
   endif

   Variable size= s.size - s.penThick/2

   if( s.opaque )
      SetDrawEnv linethick=0,fillpat=-1
      DrawRect s.x-size,s.y-size,s.x+size,s.y+size
      SetDrawEnv linethick=s.penThick
   endif
   SetDrawEnv fillpat= 0      // polys are not filled
```

```
    if( s.marker == 0 )        // 90 deg U open to the right
        DrawPoly s.x+size,s.y-size,1,1,{size,-size,-size,-size,-size,size,size,size}
    elseif( s.marker == 1 )    // 90 deg U open to the left
        DrawPoly s.x-size,s.y-size,1,1,{-size,-size,size,-size,size,size,-size,size}
    elseif( s.marker == 2 )    // Cap Gamma
        DrawPoly s.x+size,s.y-size,1,1,{size,-size,-size,-size,-size,size}
    elseif( s.marker == 3 )    // Cap Gamma reversed
        DrawPoly s.x-size,s.y-size,1,1,{-size,-size,size,-size,size,size}
    endif
    return 1
End

Window Graph1() : Graph
    PauseUpdate; Silent 1      // building window...
    Make/O/N=10 testw=sin(x)
    Display /W=(35,44,430,252) testw,testw,testw,testw
    ModifyGraph offset(testw#1)={0,-0.2},offset(testw#2)={0,-0.4},
                offset(testw#3)={0,-0.6}
    ModifyGraph mode=3,marker(testw)=100,marker(testw#1)=101,marker(testw#2)=102,
                marker(testw#3)=103
    SetWindow kwTopWin,markerHook={AudiologyMarkerProc,100,103}
EndMacro
```

Data Acquisition

Igor Pro provides a number of facilities to allow working with live data:

- Live mode traces in graphs
- FIFOs and Charts
- Background task
- External operations and external functions
- Controls and control panels
- User-defined functions

Live mode traces in graphs are useful when you acquiring complete waveforms in a single short operation and you want to update a graph many times per second to create an oscilloscope type display. See **Live Graphs and Oscilloscope Displays** on page II-297 for details.

First-In-First-Out buffers (FIFOs) and Charts are used when you have a continuous stream of data that you want to capture and, perhaps, monitor. See **FIFOs and Charts** on page IV-276 details.

You can set up a background task that will periodically perform data acquisition while allowing you to continue to work with Igor in the foreground. The background operations are *not* done using interrupts and therefore are easily disrupted by foreground operations (that would be you). Background tasks are useful only for relatively infrequent tasks that can be quickly accomplished and do not cause a cascade of graph updates or other things that take a long time. See **Background Tasks** on page IV-279 for details.

You can create an instrument-like front panel for your data acquisition setup using user-defined controls in a panel window. Refer to Chapter III-14, **Controls and Control Panels**, for details. There are many example experiments that can be found in the Examples folder.

Igor Pro comes with an XOP called VDT2 for communicating with instruments via serial port (RS232) and another XOP called NIGPIB2 for communicating via General Purpose Interface Bus (GPIB). See the Igor Pro Folder:More Extensions:Data Acquisition folder.

Sound I/O can be done using the built-in **SoundInRecord** and **PlaySound** operations.

WaveMetrics produces the NIDAQ Tools software package for doing data acquisition using National Instruments cards. NIDAQ Tools is built on top of Igor using all of the techniques mentioned in this section. Information about NIDAQ Tools is available via the WaveMetrics Web site <http://www.wavemetrics.com/Products/NIDAQTools/nidaqtools.htm>.

Third parties have created data acquisition packages that use other hardware. Information about these is also available at <http://www.wavemetrics.com/Products/thirdparty.htm>.

If an XOP package is not available for your hardware you can write your own. For this, you will need to purchase the XOP Toolkit product from WaveMetrics. See **Creating Igor Extensions** on page IV-181 for details.

FIFOs and Charts

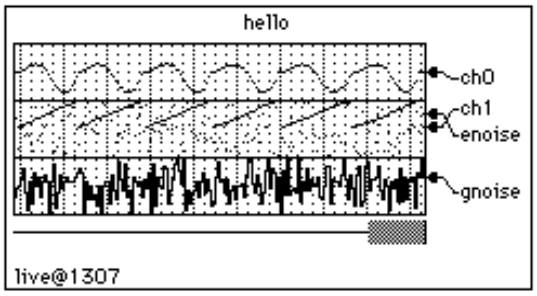
The following information will be of interest principally to people engaged in data acquisition activities. It is possible that there are other uses for these capabilities and you may want to read the following summary to see if you can think of any alternate applications. Most people who use FIFOs and Chart controls will do so via canned packages provided by expert Igor programmers (such as yourself) and will not need to know the details that follow.

Summary

FIFOs are invisible data objects that can act as a First-In-First-Out buffer between a data source and a disk file. Data is placed in a FIFO either via the AddFIFOData operation or via an XOP package designed to interface to a particular piece of hardware. Chart controls provide a graphical view of a portion of the data in a FIFO. When data acquisition is complete a FIFO can operate as a bidirectional buffer to a disk file. This allows the user to review the contents of a file by scrolling the chart "paper" back and forth. FIFOs can be used without a chart but charts have no use without a FIFO to monitor.

FIFOs can have an arbitrary number of channels each with its own number type, scaling info and units. All channels of a given FIFO share a common "timebase". A given chart control can monitor an arbitrary selection of channels from a single FIFO. Each trace can have its own display gain, color and line style and can either have its own area on the "paper" or can share an area with one or more other traces. There can be multiple chart controls active an one time in one or more panel or graph windows.

Here is a typical chart control:



Programming with FIFOs

You can create a FIFO by using the NewFIFO operation. When you are done using a FIFO you use the KillFIFO operation. A freshly created FIFO is not useful until either channels are created with the NewFIFOChan operation or until the FIFO is attached to a disk file for review using a variant of the CtrlFIFO operation. You can obtain information about a FIFO using the FIFOStatus operation and you can extract data from a FIFO using the FIFO2Wave operation. Once a FIFO is set up and ready to accept data, you can insert data using the AddFIFOData operation. Alternately, you can insert data using an XOP package. Once data is stored in a file you can review the data using a FIFO or extract data using user-defined functions. See the example experiment, "FIFO File Parse", for sample utility routines.

As with background tasks, FIFOs are considered transient objects — they are not saved and restored as part of an experiment.

A FIFO does not need to be attached to a file to be useful. Note, however, that the oldest data will be lost when a FIFO overflows.

A FIFO set up to acquire data does not become valid until the start command is issued. Chart controls will report invalid FIFOs on their status line. FIFO2Wave will give an error if it is invoked on an invalid FIFO. A stopped FIFO remains valid until the first command is issued that could potentially change the FIFO's setup.

Data in a running FIFO is written to disk when Igor notices that the FIFO is half full or when the AddFIFOData command is issued and the FIFO is full. The amount of time it takes to write data to disk can be quite considerable and at the same time unpredictable. If the computer disk cache size is large then writes to disk will be less frequent but when they do occur they will take a long time. This will matter to you most if you are attempting to take data rapidly using software (perhaps using an Igor background task). If you are taking data via interrupt transfer to an intermediate buffer of adequate size or if your hardware has an adequate internal buffer then the disk write latency may not be a concern. If dead time due to disk writes is a concern then you may want to decrease the size of the disk cache and you may want to run with a relatively small FIFO. Note that if you change the size of the disk cache you may have to reboot for the change to take effect.

When the stop command is given to a running FIFO then it goes into review mode and remains valid. If the FIFO is attached to a file then the entire contents of the file can be reviewed or be transferred to a wave using the FIFO2Wave command.

The act of attaching a FIFO to an existing file for review using the rfile keyword of the CtrlFIFO command reads in the file contents and sets itself up for review. You should not use the NewFIFOChan command or any of the other CtrlFIFO keywords except size. Here is all that is required to review a preexisting file:

```
Variable refnum
Open/R/P=mypath refnum as "my file"
NewFIFO dave
CtrlFIFO dave,rfile=refnum
```

If any chart controls have been set up to monitor FIFO dave then they will automatically configure themselves to display all the channels of dave using default parameters.

The connection between FIFOs and chart controls relies on Igor's dependency manager. The dependency manager does not automatically run during function execution — you have to explicitly call it by executing the DoUpdate command.

The dependency manager sends messages to a chart control when:

- a FIFO is created
- a FIFO is killed
- a FIFO becomes valid (start command)
- data is added to a FIFO

In particular, if inside a user function, you kill a FIFO and then create it again you should call DoUpdate after the kill so that the chart control notices the kill and can get ready for the creation.

FIFO File Format

This information is for users who may wish to create FIFO files with their own programs or for those who need to analyze data stored in a FIFO file. You will need to have a reading familiarity with the C programming language to understand the following. Note, the following information may be out of date. For the most up to date information, refer to the most recent version of the auxiliary file named NamedFIFO.h located in the "Miscellaneous:More Documentation:" folder.

Consider the following data structures….

```
#define CUR_FIFOFILE_VERSION 0

typedef struct FIFOFIleHeader{
   long typeP1,typeP2;        // 'IGOR','fifo'
   long version;              // CUR_FIFOFILE_VERSION
   long datasize; // bytes of data following ChartChunkInfo field if known
   long hsize;    // size of following ChartChunkInfo field; data follows
}FIFOFIleHeader;
```

```
#define MAX_NOTESIZE 255
#define FIFO_CHAN_VERSION_NUM 0x01

typedef struct ChartChanInfo{
   long ntype;          // number type -- NT_FP32 or NT_I16 or ...
   double offset,gain;        // result= (measval-offset)*gain
   double fsPlus,fsMinus;     // value of + & - full scale
   char name[MAX_OBJ_NAME+1]; // name of this channel
   char units[4];             // SU abbrev of units
   long chanRefcon;           // for use by data acquisition sw
}ChartChanInfo;

typedef struct ChartChunkInfo{
   long type;              // 'chrt'
   short version;          // version number of this data structure
   short pad1;             // maintain 32 bit allignment
   unsigned long startDate;   // datetime of start command
   char note[MAX_NOTESIZE+1];    // room for a short note from user
   double deltaT;    // data acquisition speed (if known,in seconds)
   long xopRefcon;               // for use by data acquisition sw
   long nchan;                   // number of channels
   ChartChanInfo info[];         // info for each channel
}ChartChunkInfo;
```

The FIFO file consists of the FIFOFileHeader followed by the ChartChunkInfo and finally by chunks of data until the end of the file. It is expected that the format of this file will undergo evolutionary changes. You should be prepared to keep up with such changes. In particular you should always check for the proper version numbers when trying to interpret such a file.

Charts

An Igor Chart works in conjunction with a FIFO to display data as it is acquired or to review data that has previously been acquired.

The information provided here pertains to using rather than programming a chart. If you are interested in programming a chart application, you should examine the examples provided by WaveMetrics in the Examples folder.

Chart Basics

An Igor Chart is neither an analytical tool nor a presentation quality graphic. It is meant only for real time monitoring of incoming data or to review data from a FIFO file. When you want an analytical or presentation quality graph you must transfer the data to a wave and then use a conventional Igor graph.

An Igor Chart emulates a mechanical chart recorder that writes on paper with moving pens as the paper scrolls by under the pens. It differs from a real chart recorder in that the paper of the latter moves at a constant velocity whereas the "paper" of an Igor chart moves only when data becomes available in the FIFO it is monitoring. If data is placed in the FIFO at a constant rate then the "paper" will scroll by at a constant rate. However, since there can be no guarantee that the data is coming in at a constant rate, we refer to the horizontal axis not in terms of time but rather in terms of data sample number.

A given chart can monitor an arbitrary selection of channels from a single FIFO. Each chart trace can have its own display gain, color and line style and can either have its own area on the "paper" or can share an area with one or more other traces. There can be multiple charts active an one time in one or more control panel or graph windows.

You create a chart using the **Chart** operation (page V-43). You can obtain information about a given chart using the **ControlInfo** operation (page V-63).

Additional Notes

Charts sometimes try to auto-configure themselves to match their FIFO. Generally this action is exactly what you want and is unobtrusive. Here are the rules that charts use:

When the FIFO becomes invalid or if it ceases to exist then the chart marks itself as being in auto-configure mode. If the FIFO then becomes valid the chart will read the FIFO information and configure itself to monitor all channels. It tries to set the ppStrip parameter to a value appropriate for the deltaT value of the FIFO. It does so by assuming a desirable update rate of around 10 strips per second. Thus, for example, if deltaT was 1 millisecond then ppStrip would be set to 100. The moral is: deltaT had better be valid or weird values of ppStrip may be created.

Any chart channel configuration commands executed after the FIFO becomes invalid but before the FIFO becomes valid again will prevent auto-configuration from taking place.

Background Tasks

Background tasks allow procedures to run periodically "in the background" while you continue to interact normally with Igor. This is useful for data acquisition, simulations and other processes that run indefinitely, over long periods of time, or need to run at regular intervals. Using a background task allows you to continue to interact with Igor while your data acquisition or simulation runs.

Originally Igor supported just one unnamed background task controlled using the **CtrlBackground** operation (page V-81). New code should use the **CtrlNamedBackground** operation (page V-82) to create named background tasks instead, as shown in the following sections. You can run any number of named backgrounds tasks.

In addition to the documentation provided here, the Background Task Demo experiment provides sample code that is designed to be redeployed for other projects. We recommend reading this documentation first and then opening the demo by choosing File→Example Experiments→Programming→Background Task Demo.

Background Task Example #1

You create and control background tasks using the **CtrlNamedBackground** operation. The main parameters of CtrlNamedBackground are the background task name, the name of a procedure to be called periodically, and the period. Here is a simple example:

```
Function TestTask(s)    // This is the function that will be called periodically
   STRUCT WMBackgroundStruct &s

   Printf "Task %s called, ticks=%d\r", s.name, s.curRunTicks
   return 0             // Continue background task
End

Function StartTestTask()
   Variable numTicks = 2 * 60    // Run every two seconds (120 ticks)
   CtrlNamedBackground Test, period=numTicks, proc=TestTask
   CtrlNamedBackground Test, start
End

Function StopTestTask()
   CtrlNamedBackground Test, stop
End
```

You start this background task by calling StartTestTask from the command line or from another procedure. StartTestTask creates a background task named Test, sets the period which is specified in units of ticks (1 tick = 1/60th of a second), and specifies the user-defined function to be called periodically (TestTask in this example).

You stop the Test background task by calling StopTestTask.

As shown above, the background procedure takes a **WMBackgroundStruct** parameter. In most cases you won't need to access it.

Background Task Exit Code

The background procedure (TestTask in the example above) returns an exit code to Igor. The code is one of the following values:

0: The background procedure executed normally.

1: The background procedure wants to stop the background task.

2: The background procedure encountered an error and wants to stop the background task.

Normally the background procedure should return 0 and the background task will continue to run. If you return a non-zero value, Igor stops the background task. You can tell Igor to terminate the background task by returning the value 1 from the background function.

If you forget to add a return statement to your background procedure, this acts like a non-zero return value and stops the background task.

Background Task Period

The CtrlNamedBackground operation's period keyword takes an integer parameter expressed in ticks. A tick is approximately 1/60th of a second. Thus the timing of Igor background tasks has a nominal resolution of 1/60th of a second.

You can override the specified period in the background task procedure by writing to the nextRunTicks field of the **WMBackgroundStruct** structure. This is needed only if you want your procedure to run at irregular intervals.

The actual time between calls to the background procedure is not guaranteed. Igor runs the background task from its outer loop, when Igor is doing nothing else. If you do something in Igor that takes a long time, for example performing a lengthy curve fit, running a user-defined function that takes a long time, or saving a large experiment, Igor's outer loop does not run so the background task will not run. If you do something that causes a compilation of Igor procedures to fail, the background task is not called. On Macintosh, the background task is not called while a menu is displayed or while the mouse button is pressed.

If you need your background task to continue running even if you edit other procedures in Igor, you need to make your project an independent module. See **Independent Modules** on page IV-214 for details.

If you need precise timing that can not be interrupted, things get much more complicated. You need to do your data acquisition in an Igor thread running in an independent module or in a thread created by an XOP that you write. See **ThreadSafe Functions and Multitasking** on page IV-288 for details.

The shortest supported period is one tick. The minimum actual period for the background task depends on your hardware and what your background task is doing. If you set the period too low for your background task, interacting with Igor becomes sluggish.

It is very easy to bog your computer down using background tasks. If the background task takes a long time to execute or if it triggers something that takes a long time (like a wave dependency formula or updating a complex graph) then it may appear that the system is hung. It is not, but it may take longer to respond to user actions than you are willing to wait.

Background Task Limitations

The principal limitation of Igor background tasks is that they are stopped while other operations are taking place. Thus, although you can type commands into the command line without disrupting the background task, when you press Return the task is stopped until execution of the command line is finished.

Background tasks do not run if procedures are in an uncompiled state. If you need your background task to continue running even if you edit other procedures in Igor, you need to make your project an independent module. See **Independent Modules** on page IV-214 for details.

On Macintosh, the background task does not run when the mouse button is pressed or when a menu is displayed.

Background Tasks and Errors

If a background task procedure contains a bug, it will typically generate an error each time the procedure runs. Normally an error generates an error dialog. If this happened over and over again, it would prevent you from fixing the bug.

Igor handles such repeated errors as follows: The first time an error occurs during the execution of the background task procedure, Igor displays an error dialog. On subsequent errors, Igor prints an error message in the history. After printing 10 such error messages, Igor stops printing messages. When you click a control, execute a command from the command line or execute a command through a menu item, the process starts over.

If the Igor debugger is enabled and Debug on Error is turned on, Igor will break into the debugger each time an error occurs in the background task procedure. You may have to turn Debug on Error off to give you time to stop the background task. You can do this from within the debugger by right-clicking.

Background Tasks and Dialogs

By default, a background task created by CtrlNamedBackground continues to run while a dialog is displayed. You can change this behavior using the CtrlNamedBackground dialogsOK keyword.

If you allow background tasks to run while an Igor dialog is present, you should ensure that your background task does not kill anything that a dialog might depend on. It should not kill waves or variables. It should never directly modify a window (except for a status panel) and especially should never remove something from a window (such as a trace from a graph). Otherwise your background task may kill something that the dialog depends on which will cause a crash.

Background Task Tips

Background tasks should be designed to execute quickly. They do not run in separate threads threads and they hang Igor's event processing as long as they run. For maximum responsiveness, your task procedure should take no more than a fraction of a second to run even when the period is long. If you have to perform a lengthy computation, let the user know what is going on, perhaps via a message in a status control panel.

Background tasks should never attempt to put up dialogs or directly wait for user input. If you need to get the attention of the user, you should design your system to include a status control panel with an area for messages or some other change in appearance. If you need to wait for the user, you should do so by monitoring global variables set by nonbackground code such as a button procedure in a panel.

Your task procedure should always leave the current data folder unchanged on exit.

Background Task Example #2

Here is an example that uses many of the concepts discussed above. The task prints a message in the history area at one second intervals five times, performs a "lengthy calculation", and then waits for the user to give the go-ahead for another run.

The task does its own timing and consequently is set to run at the maximum rate (60 times per second). The task procedure, MyBGTask, tests to see if one second has elapsed since the last time it printed a message. In a real application, you might test to see if some external event has occurred.

To try the example, copy the code below to the Procedure window and execute:

```
BGDemo()
```

```
Function BGDemo()
   DoWindow/F BGDemoPanel        // bring panel to front if it exists
   if( V_Flag != 0 )
      return 0                   // panel already exists
   endif

   String dfSav= GetDataFolder(1)// so we can leave current DF as we found it
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S root:Packages:MyDemo  // our variables go here

   // still here if no panel, create globals if needed
   if( NumVarOrDefault("inited",0) == 0 )
      Variable/G inited= 1

      Variable/G lastRunTicks= 0 // value of ticks function last time we ran
      Variable/G runNumber= 0             // incremented each time we run
      // message displayed in panel using SetVariable...
      String/G message="Task paused. Click Start to resume."

      Variable/G running=0                // when set, we do our thing
   endif

   SetDataFolder dfSav
   NewPanel /W=(150,50,449,163)
   DoWindow/C BGDemoPanel                 // set panel name
   Button StartButton,pos={21,12},size={50,20},proc=BGStartStopProc,title="Start"
   SetVariable msg,pos={21,43},size={300,17},title=" ",frame=0
   SetVariable msg,limits={-Inf,Inf,1},value= root:Packages:MyDemo:message
End

Function MyBGTask(s)
   STRUCT WMBackgroundStruct &s

   NVAR running= root:Packages:MyDemo:running

   if( running == 0 )
      return 0                     // not running -- wait for user
   endif

   NVAR lastRunTicks= root:Packages:MyDemo:lastRunTicks

   if( (lastRunTicks+60) >= ticks )
      return 0                     // not time yet, wait
   endif

   NVAR runNumber= root:Packages:MyDemo:runNumber

   runNumber += 1

   printf "Hello from the background, #%d\r",runNumber

   if( runNumber >= 5 )
      runNumber= 0
      running= 0                   // turn ourself off after five runs

      // run again when user says to
      Button StopButton,win=BGDemoPanel,rename=StartButton,title="Start"

      // Simulate a long calculation after a run
      String/G root:Packages:MyDemo:message="Performing long calculation. Please wait."
      ControlUpdate /W=BGDemoPanel msg
      DoUpdate /W=BGDemoPanel      // Required on Macintosh for control to be redrawn

      Variable t0= ticks
      do
         if (GetKeyState(0) & 32)
            Print "Lengthy process aborted by Escape key"
            break
         endif
      while( ticks < (t0+60*3) )    // delay for 3 seconds

      String/G root:Packages:MyDemo:message="Task paused. Click Start to resume."
   endif

   lastRunTicks= ticks

   return 0
End
```

```
Function BGStartStopProc(ctrlName) : ButtonControl
    String ctrlName

    NVAR running= root:Packages:MyDemo:running
    if( CmpStr(ctrlName,"StartButton") == 0 )
        running= 1
        Button $ctrlName,rename=StopButton,title="Stop"
        String/G root:Packages:MyDemo:message=""
        CtrlNamedBackground MyBGTask, proc=MyBGTask, period=1, start
    endif
    if( CmpStr(ctrlName,"StopButton") == 0 )
        running= 0
        Button $ctrlName,rename=StartButton,title="Start"
        CtrlNamedBackground MyBGTask, stop
        String/G root:Packages:MyDemo:message="Task paused. Press Start to resume."
    endif
End
```

Background Task Example #3

For another example including code that you can easily redeploy for your own project, see the Background Task Demo experiment (choose File→Example Experiments→Programming→Background Task Demo).

Old Background Task Techniques

Originally Igor supported just one unnamed background task. This is still supported for backward compatibility but new code should use CtrlNamedBackground to create and control named background tasks instead.

The unnamed background task is designated using **SetBackground**, controlled using **CtrlBackground** and killed using **KillBackground**. The **BackgroundInfo** operation returns information about the unnamed background task.

The SetBackground, CtrlBackground, KillBackground and BackgroundInfo operations work only with the unnamed background task. For named background tasks, the CtrlNamedBackground operation provides all necessary functionality.

By default, a background task created by CtrlBackground does not run while a dialog is displayed. You can change this behavior using the CtrlBackground dialogsOK keyword.

Automatic Parallel Processing with MultiThread

In Igor Pro 6.1 or later, intermediate level Igor programmers can make use of multiple processors to speed up wave assignment statements in user-defined functions. To do this, simply insert the keyword MultiThread in front of a normal wave assignment. For example, in a function:

```
Make wave1
Variable a=4
MultiThread wave1= sin(x/a)
```

The expression, on the righthand side of the assignment statement, is compiled as ThreadSafe even if the host function is not.

Note: Because of the overhead of spawning threads, you should use MultiThread only when the destination has a large number of points or the expression takes a significant amount of time to evaluate. Otherwise, you may see a performance penalty rather than an improvement.

The assignment is automatically parceled into as many threads as there are processors, each evaluating the righthand expression for a different output point.

The MultiThread keyword causes Igor to evaluate the expression for multiple output points simultaneously. Do not make any assumptions as to the order of processing and certainly do not try to use a point from the destination wave other than the current point in the expression. For example, do not do something like this:

```
wave1= wave1[p+1] - wave1[p-1]   // Result will be indeterminate
```

Expressions like that can give indeterminate results even in the absence of threading.

Here is a simple example to try on your own machine:

```
Function TestMultiThread(n)
   Variable n                    // Number of wave points

   Make/O/N=(n) testWave

   // To prime processor data cache so comparison will be valid
   testWave= 0

   Variable t1,t2
   Variable timerRefNum

   // First, non-threaded
   timerRefNum = StartMSTimer
   testWave= sin(x/8)
   t1= StopMSTimer(timerRefNum)

   // Now, automatically threaded
   timerRefNum = StartMSTimer
   MultiThread testWave= sin(x/8)
   t2= StopMSTimer(timerRefNum)

   Variable processors = ThreadProcessorCount
   Print "On a machine with",processors,"cores,MultiThread is", t1/t2,"faster"
End
```

Here is the output on a Mac Pro:

```
•TestMultiThread(100)
  On a machine with  8  cores, MultiThread is  0.059746  faster

•TestMultiThread(10000)
  On a machine with  8  cores, MultiThread is  3.4779  faster

•TestMultiThread(1000000)
  On a machine with  8  cores, MultiThread is  6.72999  faster

•TestMultiThread(10000000)
  On a machine with  8  cores, MultiThread is  8.11069  faster
```

The first result shows that the MultiThread keyword slowed the assignment down. This is because the assignment involved a small number of points and MultiThread has some overhead.

The remaining results illustrate that MultiThread can provide increased speed for assignments involving large waves.

In the last result, the speed improvement factor was greater than the number of processors. This is explained by the fact that, once running, a ThreadSafe expression has slightly less overhead than a normal expression.

If the right hand side involves calling user-defined functions, those functions must be ThreadSafe (see **ThreadSafe Functions** on page IV-83) and must also follow these rules:

1. Do not do anything to waves that are passed as parameters that might disturb memory. For example, do not change the number of points in the wave or change its data type or kill it or write to a text wave.
2. Do not write to a variable that is passed by reference.

3. Note that any waves or global variables created by the function will disappear when then wave assignment is finished.

Failure to heed rule #1 will likely result in a crash.

Although it is legal to use the MultiThread mechanism in a ThreadSafe function that is already running in a preemptive thread via **ThreadStart**, it is not recommended and will likely result in a substantial loss of speed.

For an example using MultiThread, open the Mandelbrot demo experiment file by choosing "File→Example Experiments→Programming→MultiThreadMandelbrot".

Data Folder Reference MultiThread Example

Advanced programmers can use waves containing data folder references and wave references along with MultiThread to perform multithreaded calculations more involved than evaluating an arithmetic expression. Here we use **Free Data Folders** (see page IV-75) to facilitate multithreading.

In this example, we extract each of the planes of a 3D wave, perform a filtering operation on the planes, and then finally assemble the planes into an output 3D wave. The main function, Test, executes a multithreaded assignment statement where the expression includes a call to a subroutine named Worker.

Because MultiThread is used, multiple instances of Worker execute simultaneously on different cores. Each instance runs in its own thread, working on a different plane. Each instance returns one filtered plane in a wave named M_ImagePlane in a thread-specific free data folder. The use of free data folders allows each instance of Worker to work on its own M_ImagePlane wave without creating a name conflict.

When the multithreaded assignment is finished, the main function assembles an output 3D wave by concatenating the filtered planes.

```
// Extracts a plane from the 3D input wave, filters it, and returns the
// filtered output as M_ImagePlane in a new free data folder
ThreadSafe Function/DF Worker(w3DIn, plane)
   WAVE w3DIn
   Variable plane

   DFREF dfSav= GetDataFolderDFR()

   // Create a free data folder to hold the extracted and filtered plane
   DFREF dfFree= NewFreeDataFolder()
   SetDataFolder dfFree

   // Extract the plane from the input wave into M_ImagePlane.
   // M_ImagePlane is created in the current data folder
   // which is a free data folder.
   ImageTransform/P=(plane) getPlane, w3DIn

   // Filter the plane
   WAVE wOut= M_ImagePlane
   MatrixFilter/N=21 gauss,wOut

   SetDataFolder dfSav

   // Return a reference to the free data folder containing M_ImagePlane
   return dfFree
End

Function Test()
   Variable numPlanes = 50

   // Create a 3D wave and fill it with data
   Make/O/N=(200,200,numPlanes) src3D= (p==(2*r))*(q==(2*r))
```

```
   // Create a wave to hold data folder references returned by Worker.
   // /DF specifies the data type of the wave as "data folder reference".
   Make/DF/N=(numPlanes) dfw

   Variable timerRefNum = StartMSTimer

   MultiThread dfw= Worker(src3D,p)

   Variable elapsedTime = StopMSTimer(timerRefNum) / 1E6

   Print "Assignment statement took ", elapsedTime, " seconds"

   // At this point, dfw holds data folder references to 50 free
   // data folders created by Worker. Each free data folder holds the
   // extracted and filtered data for one plane of the source 3D wave.

   // Create an output wave named out3D by cloning the first filtered plane
   DFREF df= dfw[0]
   Duplicate/O df:M_ImagePlane, out3D

   // Concatenate the remaining filtered planes onto out3D
   Variable i
   for(i=1; i<numPlanes; i+=1)
      df= dfw[i]     // Get a reference to the next free data folder
      Concatenate {df:M_ImagePlane}, out3D
   endfor

   // dfw holds references to the free data folders. By killing dfw,
   // we kill the last reference to the free data folders which causes
   // them to be automatically deleted. Because there are no remaining
   // references to the various M_ImagePlane waves, they too are
   // automatically deleted.
   KillWaves dfw
End
```

On an eight-core Mac Pro, without the MultiThread keyword above, Test printed:

```
  Assignment statement took   4.16909   seconds
```

and with MultiThread:

```
  Assignment statement took   0.614999   seconds
```

for a speed up of about 6.8 times.

Wave Reference MultiThread Example

In the preceding example, free data folders were used to hold data processed by threads. Since each free data folder held just a single wave, the example can be simplified by using free waves instead of free data folders. So here we perform the same threaded filtering of planes using free waves.

Because MultiThread is used, multiple instances of Worker execute simultaneously on different cores. Each instance runs in its own thread, working on a different plane. Each instance returns one filtered plane in a free wave named M_ImagePlane. The use of free waves allows each instance of Worker to work on its own M_ImagePlane wave without creating a name conflict.

This version of the example relies on the fact that a wave in a free data folder becomes a free wave when the free data folder is automatically deleted. See **Free Wave Lifetime** on page IV-73 for details.

```
ThreadSafe Function/WAVE Worker(w3DIn, plane)
   WAVE w3DIn
```

```
   Variable plane

   DFREF dfSav= GetDataFolderDFR()

   // Create a free data folder and set it as the current data folder
   SetDataFolder NewFreeDataFolder()

   // Extract the plane from the input wave into M_ImagePlane.
   // M_ImagePlane is created in the current data folder
   // which is a free data folder.
   ImageTransform/P=(plane) getPlane, w3DIn

   // Filter the plane
   WAVE wOut= M_ImagePlane
   MatrixFilter/N=21 gauss,wOut

   // Restore the current data folder
   SetDataFolder dfSav

   // Since the only reference to the free data folder created above
   // was the current data folder, there are now no references it.
   // Therefore, Igor has automatically deleted it.
   // Since there IS a reference to the M_ImagePlane wave in the free
   // data folder, M_ImagePlane is not deleted but becomes a free wave.

   return wOut       // Return a reference to the free M_ImagePlane wave
End

Function Test()
   Variable numPlanes = 50

   // Create a 3D wave and fill it with data
   Make/O/N=(200,200,numPlanes) srcData= (p==(2*r))*(q==(2*r))

   // Create a wave to hold data folder references returned by Worker.
   // /WAVE specifies the data type of the wave as "wave reference".
   Make/WAVE/N=(numPlanes) ww

   Variable timerRefNum = StartMSTimer

   MultiThread ww= Worker(srcData,p)

   Variable elapsedTime = StopMSTimer(timerRefNum) / 1E6

   Print "Assignment statement took ", elapsedTime, " seconds"

   // At this point, ww holds wave references to 50 M_ImagePlane free waves
   // created by Worker. Each M_ImagePlane holds the extracted and filtered
   // data for one plane of the source 3D wave.

   // Create an output wave named out3D by cloning the first filtered plane
   WAVE w= ww[0]
   Duplicate/O w, out3D

   // Concatenate the remaining filtered planes onto out3D
   Variable i
   for(i=1;i<numPlanes;i+=1)
      WAVE w= ww[i]
      Concatenate {w}, out3D
   endfor
```

```
   // ww holds references to the free waves. By killing ww, we kill
   // the last reference to the free waves which causes them to be
   // automatically deleted.
   KillWaves ww
End
```

ThreadSafe Functions and Multitasking

Experienced programmers can use **ThreadSafe Functions** (see page IV-83) to improve execution speed on computers with multiple processors and to create preemptive multitasking background tasks.

In Igor Pro 6.10 or later, a much simpler method that can be used by intermediate level programmers is available - see **Automatic Parallel Processing with MultiThread** on page IV-283. But to write a complex multitasking application, you need to use the techniques described in this section.

Preemptive multitasking uses the following functions and operations:

ThreadGroupCreate	**ThreadGroupWait**
ThreadGroupGetDF	**ThreadProcessorCount**
ThreadGroupPutDF	**ThreadReturnValue**
ThreadGroupRelease	**ThreadStart**

To run a ThreadSafe function preemptively, you first create a thread group using **ThreadGroupCreate** and then call **ThreadStart** to start your worker function. Usually you will use the same function for each thread of a group although they can be different.

The worker function must be defined as ThreadSafe and must return a real or complex numeric result. The return value can be obtained after the function finishes by calling **ThreadReturnValue**.

The worker function can take variable and wave parameters. It can not take pass-by-reference parameters or data folder reference parameters.

Any waves you pass to the worker are accessible to both the main thread and to your preemptive thread. Such waves are marked as being in use by a thread and Igor will refuse to perform any manipulations that could change the size of the wave.

You can determine if any threads of a group are still running by calling **ThreadGroupWait**. Use zero for milliseconds to wait to just test or provide a large value to cause the main thread to sleep until the threads are finished. If you know the maximum time the threads should take, you can use that value so you can print an error message or take other action if the threads don't return in time.

Once you are finished with a given thread group, call **ThreadGroupRelease**.

The Igor Debugger can not be used with preemptive threads. You will need to use print statements for debugging.

The hard part of using multithreading is devising a scheme for partitioning your data processing algorithms into threads.

Thread Data Environment

When a thread is started, Igor creates a root data folder for that thread. This root data folder and any data objects that the thread creates in it are private to the thread. This constitutes a separate data hierarchy for each thread.

Data is transferred, when you request it, from the main thread to a preemptive thread and vice-versa using input and output queues. The "currency" of these queues is the data folder, which provides considerable flexibility for passing data to threads and for retrieving results. Each thread group has an input queue to

which the main thread may post data and an output queue from which the main thread may retrieve results.

The terms “input“ and “output“ are relative to the preemptive thread. The main thread posts a data folder to the input queue to send input to the preemptive thread. The preemptive thread retrieves the data folder from the input queue. After processing, the preemptive thread may post a data folder to the output queue. The main thread reads output from the preemptive thread by retrieving the data folder from the output queue.

Use **ThreadGroupPutDF** to post data folders and **ThreadGroupGetDF** to retrieve them. These are called from both the main thread and from preemptive threads.

ThreadGroupPutDF clips the specified data folder (and everything it contains) out of the source thread and puts it in the queue. From the standpoint of the source thread, it is as if KillDataFolder had been called. While a data folder resides in a queue, it is not accessible by any thread. See the documentation for **ThreadGroupPutDF** for some warnings about its use.

ThreadGroupGetDF stitches the data folder from the queue into the current data folder of the calling thread and gives it a unique name starting with its original name. It is then owned by the calling thread.

Parallel Processing

In this example, we attempt to improve the speed of filling columns of a 2D wave with a sin function. The traditional method is compared with parallel processing. Notice how much more complicated the multi-threaded version, MTFillWave, is compared to the single threaded STFillWave.

```
ThreadSafe Function MyWorkerFunc(w,col)
    WAVE w
    Variable col

    w[][col]= sin(x/(col+1))

    return stopMSTimer(-2)            // Time when we finished
End

Function MTFillWave(dest)
    WAVE dest

    Variable ncol= DimSize(dest,1)
    Variable i,col,nthreads= ThreadProcessorCount
    variable mt= ThreadGroupCreate(nthreads)

    for(col= 0;col<ncol;)
        for(i=0;i<nthreads;i+=1)
            ThreadStart mt,i,MyWorkerFunc(dest,col)
            col+=1
            if( col>=ncol )
                break
            endif
        endfor

        do
            variable tgs= ThreadGroupWait(mt,100)
        while( tgs != 0 )
    endfor
    variable dummy= ThreadGroupRelease(mt)
End

Function STFillWave(dest)
    WAVE dest

    Variable ncol= DimSize(dest,1)
    Variable col

    for(col= 0;col<ncol;col+=1)
        MyWorkerFunc(dest,col)
    endfor
End

Function ThreadTest(rows)
    Variable rows
```

```
    Variable cols=10

    make/o/n=(rows,cols) jack

    Variable i

    for(i=0;i<10;i+=1)      // get any pending pause events out of the way
    endfor

    Variable ttime= stopMSTimer(-2)

    Variable t0= stopMSTimer(-2)
    MTFillWave(jack)
    Variable t1= stopMSTimer(-2)
    STFillWave(jack)
    Variable t2= stopMSTimer(-2)

    ttime= (stopMSTimer(-2) - ttime)*1e-6

    // Times are in microseconds
    printf "ST: %d, MT: %d; ",t2-t1,t1-t0
    printf "speed up factor: %.3g; tot time= %.3gs\r",(t2-t1)/(t1-t0),ttime
End
```

The empty loop above is necessary because of periodic pauses in execution to check for user aborts. If a pause was pending, we want to get it out of the way beforehand to avoid it affecting the first timing test.

After starting Igor Pro, there is initially some extra overhead associated with creating new threads. Consequently, in the test results to follow, the first test is run twice.

Results for Mac Mini 1.66 GHz Core Duo, OS X 10.4.6:

```
•ThreadTest(100)
  ST: 223, MT: 1192; speed up factor: 0.187; tot time= 0.00146s
•ThreadTest(100)
  ST: 211, MT: 884; speed up factor: 0.239; tot time= 0.0011s
•ThreadTest(1000)
  ST: 1991, MT: 1821; speed up factor: 1.09; tot time= 0.00381s
•ThreadTest(10000)
  ST: 19857, MT: 11921; speed up factor: 1.67; tot time= 0.0318s
•ThreadTest(100000)
  ST: 199174, MT: 113701; speed up factor: 1.75; tot time= 0.313s
•ThreadTest(1000000)
  ST: 2009948, MT: 1146113; speed up factor: 1.75; tot time= 3.16s
```

As you can see, when there is sufficient work to be done, the speed up factor approaches the theoretical maximum of 2 for dual processors.

Now on the same computer but booting into Windows XP Pro:

```
•ThreadTest(100)
  ST: 245, MT: 523; speed up factor: 0.468; tot time= 0.000776s
•ThreadTest(100)
  ST: 399, MT: 247; speed up factor: 1.61; tot time= 0.000655s
•ThreadTest(1000)
  ST: 3526, MT: 1148; speed up factor: 3.07; tot time= 0.00468s
•ThreadTest(10000)
  ST: 34830, MT: 10467; speed up factor: 3.33; tot time= 0.0453s
•ThreadTest(100000)
  ST: 350253, MT: 99298; speed up factor: 3.53; tot time= 0.45s
•ThreadTest(1000000)
  ST: 2837645, MT: 1057275; speed up factor: 2.68; tot time= 3.89s
```

So, what is happening here? The speed-up factors for Windows XP are greater than for Mac OS X, but mostly because the ST version is much slower. We do not known why the ST version runs more slowly — the Benchmark 2.01 example experiment shows similar values for OS X vs. XP on this same computer.

Input/Output Queues

In this example, we create data folders containing a data wave and a string variable, which specifies the task to be performed, and then post them to a thread group. We use a different data folder for output although you can also simply reuse the input data folder.

```
ThreadSafe Function MyWorkerFunc()
```

```
    do
        do
            String tdf= ThreadGroupGetDF(0,1000)
            if( strlen(tdf) == 0 )
                if( GetRTError(2) ) // New in 6.20 to allow this distinction:
                    Print "worker closing down due to group release"
                else
                    Print "worker thread still waiting for input queue"
                endif
            else
                break
            endif
        while(1)

        SetDataFolder tdf
        SVAR todo
        WAVE jack

        // To reuse the input data folder, simply remove the next two
        // statments as well as the KillDataFolder below.
        SetDataFolder ::

        NewDataFolder/S outDF

        Duplicate jack,outw // WARNING: outw must be cleared. See WAVEClear below
        String/G did= todo
        if( CmpStr(todo,"sin") )
            outw= sin(outw)
        else
            outw= cos(outw)
        endif


        // Needed if you change the code to reuse input data folder. Allows put.
        WAVEClear jack

        // Clear outw so Duplicate above does not try to use it and to allow put.
        WAVEClear outw

        ThreadGroupPutDF 0,:

        KillDataFolder tdf      // We are done with the input data folder
    while(1)

    return 0
End

Function TestThreadQueue()
    Variable i,ntries= 5,nthreads= 2

    variable/G tgID= ThreadGroupCreate(nthreads)

    for(i=0;i<nthreads;i+=1)
        ThreadStart tgID,i,MyWorkerFunc()
    endfor

    for(i=0;i<ntries;i+=1)
        NewDataFolder/S forThread
        String/G todo
        if( mod(i,3) == 0 )
            todo= "sin"
        else
            todo= "cos"
        endif
        Make/N= 5 jack= x + gnoise(0.1)

        WAVEClear jack

        ThreadGroupPutDF tgID,:
    endfor

    for(i=0;i<ntries;i+=1)
        do
            DFREF dfr= ThreadGroupGetDFR(tgID,1000) // Get results in free data folder
```

```
         if ( DatafolderRefStatus(dfr) == 0 )
            Print "Main still waiting for worker thread results."
         else
            break
         endif
      while(1)

      SVAR/SDFR=dfr did
      WAVE/SDFR=dfr outw

      Print "task= ",did,"results= ",outw

      // The next two statements are not really needed as the same action
      // will happen the next time through the loop or, for the last iteration,
      // when this function returns.
      WAVEClear outw
      KillDataFolder dfr
   endfor

   // This terminates the MyWorkerFunc by setting an abort flag
   Variable tstatus= ThreadGroupRelease(tgID)
   if( tstatus == -2 )
      Print "Thread would not quit normally, had to force kill it. Restart Igor."
   endif
End
```

Typical output:

```
•TestThreadQueue()
   task=   sin  results=
outw[0]= {0.994567,0.660904,-0.516692,-0.996884,-0.63106}
   task=   cos  results=
outw[0]= {0.0786631,0.709576,0.873524,0.0586175,-0.718122}
   task=   cos  results=
outw[0]= {-0.23686,0.848603,0.871922,0.0992451,-0.856209}
   task=   sin  results=
outw[0]= {0.999734,0.531563,-0.172071,-0.931296,-0.750942}
   task=   cos  results=
outw[0]= {-0.166893,0.767707,0.925874,0.114511,-0.662994}
   worker closing down due to group release
   worker closing down due to group release
```

Preemptive Background Task

In this example, we create a single worker thread that runs while the user can do other things. A normal cooperative named background task will retrieve results. Although the named task will sometimes be blocked (as described in **Background Tasks** on page IV-279) the preemptive worker thread will always be running or waiting for data.

We put the code for the background tasks in an independent module (see **The IndependentModule Pragma** on page IV-43) so that the user can recompile procedures, which is done automatically when a recreation macro is created.

One use for a preemptive background task is when you have lengthy computations but want to continue to do other things, such as creating graphics for publication. Although you can do anything you want while the task runs in the experiment, if you load a different experiment, the thread will be killed.

Our "lengthy computation" is simply creating a wave of sin values but which is prolonged by code delays of a few seconds before posting the results. The named background task checks the output queue every 10 ticks (when it is not blocked) and updates a graph with data retrieved from the queue.

Copy the following code to a new procedure window (not the main procedure window):

```
#pragma IndependentModule= PreemptiveExample

ThreadSafe Function MyWorkerFunc()
   do

      String tdf= ThreadGroupGetDF(0,inf)
      if( strlen(tdf) == 0 )
         return -1              // Thread is being killed
      endif
```

```
        SetDataFolder tdf
        WAVE Kenith                // Array of frequencies to calculate
        SetDataFolder ::

        Variable i, n= numpnts(Kenith)

        for(i=0;i<n;i+=1)

            NewDataFolder/S resultsDF
            Make jack= sin(Kenith[i]*x)

            Variable t0= ticks
            do
               // waste cpu for a few seconds
            while(ticks < (t0+120))

            WAVEClear jack      // Must have no references to allow put.

            ThreadGroupPutDF 0,:
        endfor

        KillDataFolder tdf      // We are done with the input data folder

    while(1)

    return 0
End

Function myResultsFunc(s)
    STRUCT WMBackgroundStruct &s

    String dfSav= GetDataFolder(1)

    SetDataFolder root:testdf
    NVAR tgID
    String tdf= ThreadGroupGetDF(tgID,0)
    if( strlen(tdf) != 0 )
        SetDataFolder tdf
        WAVE jack           // this is the output from the thread
        AppendToGraph/W=ThreadResultsGraph jack
    endif

    SetDataFolder dfSav
    return 0
End
```

And copy this function to the main procedure window:

```
Function TestThreadQueue()
    String dfSav= GetDataFolder(1)

    NewDataFolder/O/S root:testdf // thread group ID and result datafolders go here

    variable/G tgID= ThreadGroupCreate(1)

    ThreadStart tgID,0,PreemptiveExample#MyWorkerFunc()

    // MyWorkerFunc is now running and waiting for input data
    // now, let's give it something to do
    NewDataFolder/S tasks
    Make/N=5 Kenith= 1/(10+p/2+enoise(0.2))// array of frequencies to calculate
    WAVEClear Kenith

    ThreadGroupPutDF tgID,:// thread is now crunching away

    Display as "output"
    DoWindow/C ThreadResultsGraph     // results will be appended here...

    // ...by this named task
    CtrlNamedBackground myResultsTask,period=10,proc=PreemptiveExample#myResultsFunc,start

    SetDataFolder dfSav     // be a good citizen and restore current df
End

Function PostMoreFreqs()
    NVAR tgID= root:testdf:tgID

    NewDataFolder/S moretasks
    Make/N=50 Kenith= 1/(15+p/2+enoise(0.2))// array of frequencies to calculate
    WAVEClear Kenith
```

```
   ThreadGroupPutDF tgID,:       // thread continues crunching
End
```

Once the code is compiled, on the command line execute

```
TestThreadQueue()
```

After the action stops, send more tasks to the background thread by executing

```
PostMoreFreqs()
```

While this is running, experiment with creating graphs, using dialogs, creating functions, etc. Note that both tasks run indefinitely. If you want to start over, you will need to stop the threads and kill the graph.

Cursors — Moving Cursor Calls Function

Note: The following section "The Old Easy Way" is now outdated as of Igor Pro 5 (though it still works). Use the **SetWindow** operation (page V-569) hook and the cursormoved event instead. The window hook isolates handling the cursor to the graph window in which the cursor is moving, and is no more difficult to program than the old global CursorMovedHook which is called for cursors moving in any graph window.

The Old Easy Way

You can write a "hook function" — which must be named "CursorMovedHook" — and Igor will automatically call it with one string argument containing information about the graph, trace or image, and cursor in the following format:

```
GRAPH:graphName;CURSOR:<A - J>;TNAME:traceName; MODIFIERS:modifierNum;
ISFREE:freeNum;POINT:xPointNumber; [YPOINT:yPointNumber;]
```

The CursorMovedHook function is called whenever any cursor is moved in any graph, unless Option (*Macintosh*) or Alt (*Windows*) is held down.

The *traceName* value is the name of the graph trace or image to which it is attached or which supplies the x (and y) values.

The *modifierNum* value represents the state of some of the keyboard keys summed together:

1 If Command (*Macintosh*) or Ctrl (*Windows*) is pressed.

2 If Control (*Macintosh only*) is pressed.

4 If Shift is pressed.

8 If Caps Lock is pressed.

The Option key (*Macintosh*) or Alt key (*Windows*) is not represented because it prevents the hook from being called.

The YPOINT keyword and value are present only when the cursor is attached to a two-dimensional item such as an image, contour, or waterfall plot or when the cursor is free.

If cursor is free, POINT and YPOINT values are fractional relative positions (see description in **Cursor** operation on page V-84). If TNAME is empty, fields POINT, ISFREE and YPOINT are not present.

This example hook function simply prints the information in the history window:

```
Function CursorMovedHook(info)
   String info
   Print info
End
```

Whenever any cursor on any graph is moved, this CursorMovedHook function will print something like the following in the history area:

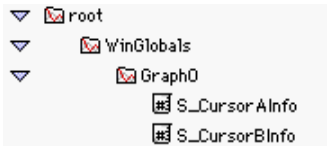
```
GRAPH:Graph0;CURSOR:A;TNAME:jack;MODIFIERS:0;ISFREE:0;POINT:6;
```

See **Example Cursor Global User Function** on page IV-296 for a more sophisticated use of this info string.

The Hard Way

It is also possible to use Igor's dependency mechanism to cause functions to execute automatically when the user moves cursors on a given graph. The advantage of this method is that the function(s) are specific to the given graph.

To do this, create a data folder named WinGlobals and within that create a data folder of the same name as the desired graph. Within that data folder create a string variable named S_CursorAInfo or S_CursorBInfo.



Cursor Globals

If these variables exist (and if Option (*Macintosh*) or Alt (*Windows*) is not held down), then when the user moves a cursor, Igor will set them to an informational string in the following format (all one line):

```
GRAPH:graphName;CURSOR:<A - J>;TNAME:traceName;MODIFIERS:modifierNum;
ISFREE:freeNum;POINT:xPointNumber;[YPOINT:yPointNumber;]
```

Moving cursor A, for example, will update S_CursorAInfo. If the cursor is being removed, traceName will be zero length.

Creating the Cursor Globals

Here is code that creates the necessary global variables for the top graph:

```
Function CursorGlobalsForGraph()
   String graphName= WinName(0,1)
   if( strlen(graphName) )
      String df= GetDataFolder(1);
      NewDataFolder/O root:WinGlobals
      NewDataFolder/O/S root:WinGlobals:$graphName
      String/G S_CursorAInfo, S_CursorBInfo
      SetDataFolder df
   endif
End
```

And code that deletes the globals:

```
Function RemoveCursorGlobals()
   String graphName= WinName(0,1)
   if( strlen(graphName) )
      KillDataFolder root:WinGlobals:$graphName
   endif
End
```

(You can arrange for RemoveCursorGlobals() to be called automatically when Graph0 is closed by setting the "window hook function" of Graph0. See the **SetWindow** operation.)

Establishing a Dependency Between Cursor Globals and a User Function

For a user-defined function to be automatically called when, for example, you move the A cursor for Graph0, you must establish a dependency involving the global WinGlobals:Graph0:S_CursorAInfo and the user-defined function.

This example establishes a dependency between additional variables and each of the cursor info globals involving the example user function shown in the next section (CursorMoved). This dependency updates WinGlobals:Graph0:dependentA by CursorMoved whenever S_CursorAInfo is changed due to cursor A being moved (and likewise for dependentB and S_CursorBInfo):

```
Function CursorDependencyForGraph()
   String graphName= WinName(0,1)
   if( strlen(graphName) )
      String df= GetDataFolder(1);
```

```
      NewDataFolder/O root:WinGlobals
      NewDataFolder/O/S root:WinGlobals:$graphName
      String/G S_CursorAInfo, S_CursorBInfo
      Variable/G dependentA
      SetFormula dependentA, "CursorMoved(S_CursorAInfo, 0)"
      Variable/G dependentB
      SetFormula dependentB,"CursorMoved(S_CursorBInfo, 1)"
      SetDataFolder df
   endif
End
```

Example Cursor Global User Function

The example user function takes two arguments. Yours must have at least one: the cursor info string that Igor automatically updates when the cursor is moved.

Inside the user function, you can parse the cursor info parameter with the **StringByKey** (page V-675) and **NumberByKey** (page V-458) functions. You can obtain additional information about the cursor using the usual routines such as hcsr and vcsr.

In almost all cases, the top graph will be the one referenced by the info string. However, it is possible to write a user function that moves the cursors and then brings a different graph to the top before your dependency fires. Therefore it is possible (though unlikely) that calling a routine like hcsr could generate an error if you don't first check that the top graph is the one referenced in the info string. (You can use `DoWindow/F graphName` to ensure that the graph is top-most.)

This simple example prints to the history area the value of a five-point average around the new cursor position whenever the cursor moves:

```
Function CursorMoved(info, isB)
   String info
   Variable isB            // 0 if A cursor, nonzero if B cursor

   Variable result= NaN    // error result
   // Check that the top graph is the one in the info string.
   String topGraph= WinName(0,1)
   String graphName= StringByKey("GRAPH", info)
   if( CmpStr(graphName, topGraph) == 0 )
      // If the cursor is being turned off
      // the trace name will be zero length.
      String tName= StringByKey("TNAME", info)
      if( strlen(tName) )  // cursor still on
         String cn
         Variable xVal
         if( isB )
            xVal= hcsr(B)
            cn= "Cursor B"
         else
            xVal= hcsr(A)
            cn= "Cursor A"
         endif
         // compute the local 5-point mean
         WAVE w= TraceNameToWaveRef(graphName, tName)
         Variable pointNum= NumberByKey("POINT",info)
         Variable x1= pnt2x(w,pointNum-2)
         Variable x2= pnt2x(w,pointNum+2)
         result= mean(w,x1,x2)
         Print cn+" on "+tName+" moved to x= ",xVal,"mean= ",result
      endif
   endif
   return result
End
```

The Result

Whenever either cursor on Graph0 is moved, the CursorMoved function will print something like the following in the history area:

```
Cursor A on wave0#1 moved to x=   33  mean=   0.0239056
```

Volume V Reference

Table of Contents

Igor Reference

This volume contains detailed information about Igor Pro's built-in operations, functions, and keywords. They are listed alphabetically after the category sections that follow.

External operations (XOPs) and external functions (XFUNCs) are not covered here. For information about them, use the Command Help tab of the Igor Help Browser.

Built-In Operations by Category

Note: some operations may appear in more than one category.

Graphs

AppendText	AppendToGraph	AppendToLayout	CheckDisplayed
ColorScale	ColorTab2Wave	ControlBar	Cursor
DefaultFont	DefineGuide	DelayUpdate	Display
DoUpdate	DoWindow	ErrorBars	GetAxis
GetMarquee	GetSelection	GetWindow	GraphNormal
GraphWaveDraw	GraphWaveEdit	HideInfo	HideTools
KillFreeAxis	KillWindow	Label	Legend
ModifyFreeAxis	ModifyGraph	ModifyWaterfall	MoveSubwindow
MoveWindow	NewFreeAxis	NewWaterfall	PauseUpdate
PrintGraphs	RemoveFromGraph	RenameWindow	ReorderImages
ReorderTraces	ReplaceText	ReplaceWave	ResumeUpdate
SaveGraphCopy	SetActiveSubwindow	SetAxis	SetMarquee
SetWindow	ShowInfo	ShowTools	StackWindows
Tag	TextBox	TileWindows	

Contour and Image Plots

AppendImage	AppendMatrixContour	AppendToLayout	AppendXYZContour
CheckDisplayed	ColorScale	ColorTab2Wave	DefineGuide
DoUpdate	DoWindow	HideTools	ImageLoad
ImageSave	KillWindow	ModifyContour	ModifyImage
MoveSubwindow	NewImage	PauseUpdate	RemoveContour
RemoveImage	ReplaceWave	SetActiveSubwindow	SetWindow
ShowTools	Tag	TileWindows	

Tables

AppendToLayout	AppendToTable	CheckDisplayed	DelayUpdate
DoUpdate	DoWindow	Edit	GetSelection
GetWindow	KillWindow	ModifyTable	MoveWindow
PauseUpdate	PrintTable	RemoveFromTable	RenameWindow
ResumeUpdate	SaveTableCopy	SetWindow	StackWindows
TileWindows			

Layouts

AppendLayoutObject	AppendText	AppendToLayout	DefaultFont
DelayUpdate	DoUpdate	DoWindow	GetMarquee
GetSelection	GetWindow	HideTools	KillWindow

Layout
Legend
ModifyLayout
MoveWindow
NewLayout
PauseUpdate
PrintLayout
See Also
RemoveLayoutObjects
RenameWindow
ReplaceText
ResumeUpdate
SetMarquee
SetWindow
ShowTools
Stack
StackWindows
TextBox
Tile
TileWindows

Subwindows

DefineGuide
GetMarquee
KillWindow
MoveSubwindow
RenameWindow
SetActiveSubwindow
SetMarquee

Other Windows

CloseProc
DisplayProcedure
DoWindow
GetSelection
GetWindow
HideProcedures
HideTools
KillWindow
ModifyPanel
MoveWindow
NewNotebook
NewPanel
Notebook
NotebookAction
OpenNotebook
PrintNotebook
RenameWindow
SaveNotebook
SetWindow
ShowTools
StackWindows
TileWindows

All Windows

Append
AutoPositionWindow
DoWindow
GetSelection
GetUserData
GetWindow
KillWindow
Modify
MoveWindow
Remove
RenameWindow
SetWindow
StackWindows
TileWindows

Wave Operations

AddMovieAudio
Append
AppendToGraph
AppendToTable
CheckDisplayed
ColorTab2Wave
Concatenate
CopyScales
DeletePoints
Display
Duplicate
Edit
Extract
FIFO2Wave
FindSequence
FindValue
GraphWaveDraw
GraphWaveEdit
InsertPoints
KillWaves
LoadData
LoadWave
Make
MoveWave
Note
PlaySound
Redimension
Remove
RemoveFromGraph
RemoveFromTable
Rename
ReplaceWave
Reverse
Rotate
Save
SetDimLabel
SetScale
SetWaveLock
WAVEClear
WaveStats
wfprintf

Analysis

APMath
boundingBall
Convolve
Correlate
ConvexHull
Cross
CurveFit
CWT
Differentiate
DSPDetrend
DSPPeriodogram
DWT
EdgeStats
FastOp
FFT
FilterFIR
FilterIIR
FindLevel
FindLevels
FindPeak
FindPointsInPoly
FindRoots
FindValue
FuncFit
FuncFitMD
FastGaussTransform
Hanning
HilbertTransform
Histogram
IFFT
IndexSort
Integrate

Integrate1D	IntegrateODE	Interp3DPath	Interpolate3D
Loess	LombPeriodogram	MakeIndex	NeuralNetworkRun
NeuralNetworkTrain	Optimize	PCA	PrimeFactors
Project	PulseStats	RatioFromNumber	Resample
Smooth	SmoothCustom	Sort	SphericalInterpolate
SphericalTriangulate	Triangulate3D	Unwrap	WaveMeanStdv
WaveStats	WaveTransform	WignerTransform	WindowFunction

Matrix Operations

Concatenate	Extract	FFT	IFFT
ImageFilter	Loess	MatrixConvolve	MatrixCorr
MatrixEigenV	MatrixFilter	MatrixGaussJ	MatrixInverse
MatrixLinearSolve	MatrixLinearSolveTD	MatrixLLS	MatrixLUBkSub
MatrixLUD	MatrixMultiply	MatrixOp	MatrixSchur
MatrixSolve	MatrixSVBkSub	MatrixSVD	MatrixTranspose
Reverse	WaveTransform		

Analysis of Functions

FindRoots	Integrate1D	IntegrateODE	Optimize

Signal Processing

Convolve	Correlate	CWT	DSPDetrend
DSPPeriodogram	DWT	EdgeStats	FFT
FilterFIR	FilterIIR	FindLevel	FindLevels
FindPeak	Hanning	HilbertTransform	IFFT
ImageWindow	LinearFeedbackShiftRegister	LombPeriodogram	PulseStats
Resample	Rotate	SmoothCustom	Unwrap
WignerTransform	WindowFunction		

Image Analysis

ColorScale	ColorTab2Wave	DWT	ImageAnalyzeParticles
ImageBlend	ImageBoundaryToMask	ImageEdgeDetection	ImageFileInfo
ImageFilter	ImageFocus	ImageGenerateROIMask	ImageHistModification
ImageHistogram	ImageInfo	ImageInterpolate	ImageLineProfile
ImageLoad	ImageMorphology	ImageNameList	ImageNameToWaveRef
ImageRegistration	ImageRemoveBackground	ImageRestore	ImageRotate
ImageSave	ImageSeedFill	ImageSnake	ImageStats
ImageThreshold	ImageTransform	ImageUnwrapPhase	ImageWindow
Loess	MatrixFilter		

Statistics

EdgeStats	FPClustering	Histogram	ImageHistModification
ImageHistogram	ImageStats	KMeans	PCA
PulseStats	SetRandomSeed	StatsAngularDistanceTest	StatsANOVA1Test
StatsANOVA2NRTest	StatsANOVA2RMTest	StatsANOVA2Test	StatsChiTest
StatsCircularCorrelationTest	StatsCircularMeans	StatsCircularMoments	StatsCircularTwoSampleTest

StatsCochranTest StatsContingencyTable StatsDIPTest StatsDunnettTest
StatsFriedmanTest StatsFTest StatsHodgesAjneTest StatsJBTest
StatsKendallTauTest StatsKSTest StatsKWTest StatsLinearCorrelationTest
StatsLinearRegression StatsMultiCorrelationTest StatsNPMCTest StatsNPNominalSRTest
StatsQuantiles StatsRankCorrelationTest StatsResample StatsSample
StatsScheffeTest StatsSignTest StatsSRTest StatsTTest
StatsTukeyTest StatsVariancesTest StatsWatsonUSquaredTest StatsWatsonWilliamsTest
StatsWheelerWatsonTest StatsWilcoxonRankTest StatsWRCorrelationTest WaveMeanStdv
WaveStats

Geometry

boundingBall ConvexHull FindPointsInPoly Interp3DPath
Interpolate3D Project SphericalInterpolate SphericalTriangulate
Triangulate3D

Drawing

DrawAction DrawArc DrawBezier DrawLine
DrawOval DrawPICT DrawPoly DrawRect
DrawRRect DrawText GraphNormal GraphWaveDraw
GraphWaveEdit HideTools SetDashPattern SetDrawEnv
SetDrawLayer ShowTools ToolsGrid

Programming & Utilities

Abort BackgroundInfo Beep BuildMenu
ChooseColor CloseProc CtrlBackground CtrlNamedBackground
DefaultGUIFont DefaultGUIControls Debugger DebuggerOptions
DelayUpdate DisplayHelpTopic DisplayProcedure DoAlert
DoIgorMenu DoUpdate DoXOPIdle Execute
Execute/P ExecuteScriptText ExperimentModified GetLastUserMenuInfo
Grep HideIgorMenus HideProcedures IgorVersion
KillBackground KillStrings KillVariables LoadPackagePreferences
MarkPerfTestTime MeasureStyledText MoveString MoveVariable
MoveWave ParseOperationTemplate PauseForUser PauseUpdate
Preferences PrintSettings PutScrapText Quit
Rename ResumeUpdate SavePackagePreferences SetBackground
SetFormula SetIgorHook SetIgorMenuMode SetIgorOption
SetProcessSleep SetRandomSeed SetWaveLock ShowIgorMenus
Silent Sleep Slow SplitString
sprintf sscanf String StructGet
StructPut ThreadGroupPutDF ThreadStart ToCommandLine
Variable WAVEClear

Files & Paths

BrowseURL Close CopyFile CopyFolder
CreateAliasShortcut DeleteFile DeleteFolder FBinRead
FBinWrite fprintf FReadLine FSetPos

FStatus FTPCreateDirectory FTPDelete FTPDownload
FTPUpload GetFileFolderInfo Grep ImageFileInfo
ImageLoad ImageSave KillPath KillPICTs
KillWaves LoadData LoadPICT LoadWave
MoveFile MoveFolder NewNotebook NewPath
Open OpenNotebook OpenProc PathInfo
ReadVariables RemovePath RenamePath RenamePICT
Save SaveData SaveExperiment SaveGraphCopy
SaveNotebook SavePICT SaveTableCopy SetFileFolderInfo
wfprintf

Data Folders

cd Dir DuplicateDataFolder KillDataFolder
MoveDataFolder MoveVariable MoveWave NewDataFolder
pwd RenameDataFolder ReplaceWave root
SetDataFolder

Movies & Sound

AddMovieAudio AddMovieFrame Beep CloseMovie
ImageFileInfo NewMovie PlayMovie PlayMovieAction
PlaySnd PlaySound SoundInRecord SoundInSet
SoundInStartChart SoundInStatus SoundInStopChart

Controls & Cursors

Button Chart CheckBox ControlBar
ControlInfo ControlUpdate Cursor CustomControl
DefaultGUIFont DefaultGUIControls GetUserData GroupBox
HideInfo HideTools KillControl ListBox
ModifyControl ModifyControlList NewPanel popup
PopupContextualMenu PopupMenu PopupMenuControl SetVariable
ShowInfo ShowTools Slider TabControl
TitleBox ValDisplay

FIFOs

AddFIFOData AddFIFOVectData Chart ControlInfo
CtrlFIFO FIFO2Wave FIFOStatus KillFIFO
NewFIFO NewFIFOChan SoundInStartChart

Printing

Print printf PrintGraphs PrintLayout
PrintNotebook PrintSettings PrintTable sprintf
wfprintf

Built-In Functions by Category

Note: some functions may appear in more than one category.

Numbers

e	Inf	NaN	numtype
Pi	VariableList		

Trig

acos	asin	atan	atan2
cos	cot	csc	sawtooth
sec	sin	sinc	sqrt
tan			

Exponential

acosh	alog	asinh	atanh
cosh	coth	cpowi	exp
ln	log	sinh	tanh

Complex

cabs	cequal	cmplx	conj
cpowi	imag	magsqr	p2rect
r2polar	real		

Rounding

abs	cabs	ceil	floor
limit	max	min	mod
round	sign	trunc	

Conversion

char2num	cmplx	date2secs	imag
LowerStr	magsqr	num2char	num2istr
num2str	p2rect	pnt2x	r2polar
real	Secs2Date	Secs2Time	str2num
UpperStr	x2pnt		

Time and Date

CreationDate	date	dateToJulian	date2secs
DateTime	JulianToDate	modDate	Secs2Date
Secs2Time	startMSTimer	stopMSTimer	ticks
time			

Matrix Analysis

MatrixDet	MatrixDot	MatrixRank	MatrixTrace

Wave Analysis

area	areaXY	BinarySearch	BinarySearchInterp
ContourZ	FakeData	faverage	faverageXY
interp	Interp2D	Interp3D	mean
p	poly	poly2D	PolygonArea
q	r	s	sum
t	Variance	x	y
z			

About Waves

BinarySearch	BinarySearchInterp	ContourInfo	ContourNameToWaveRef
ContourZ	CreationDate	CsrInfo	CsrWave
CsrWaveRef	CsrXWave	CsrXWaveRef	deltax
DimDelta	DimOffset	DimSize	EqualWaves
exists	FindDimLabel	GetDimLabel	GetWavesDataFolder
GetWavesDataFolderDFR	hcsr	ImageInfo	ImageNameToWaveRef
leftx	modDate	NameOfWave	NewFreeWave
note	numpnts	p	pcsr
pnt2x	q	qcsr	r
rightx	s	t	TagVal
TagWaveRef	TraceInfo	TraceNameToWaveRef	WaveCRC
WaveDims	WaveExists	WaveInfo	WaveList
WaveName	WaveRefIndexed	WaveRefsEqual	WaveType
WaveUnits	x	x2pnt	xcsr
XWaveName	XWaveRefFromTrace	y	z
zcsr			

Special

airyA	airyAD	airyB	airyBD
Besseli	Besselj	Besselk	Bessely
bessI	bessJ	bessK	bessY
beta	betai	binomial	binomialln
binomialNoise	chebyshev	chebyshevU	dawson
digamma	ei	enoise	erf
erfc	erfcw	expInt	expnoise
factorial	fresnelCos	fresnelCS	fresnelSin
gamma	gammaInc	gammaNoise	gammln
gammp	gammq	Gauss	Gauss1D
Gauss2D	gcd	gnoise	hermite
hermiteGauss	hyperG0F1	hyperG1F1	hyperG2F1
hyperGNoise	hyperGPFQ	inverseErf	inverseErfc
laguerre	laguerreA	laguerreGauss	legendreA
logNormalNoise	lorentzianNoise	MandelbrotPoint	MarcumQ

poissonNoise
poly
poly2D
sphericalBessJ
sphericalBessJD
sphericalBessY
sphericalBessYD
sphericalHarmonics
sqrt
ZernikeR

Statistics

binomialln	binomialNoise	enoise	erf
erfc	expnoise	faverage	faverageXY
gamma	gammaInc	gammaNoise	gammln
gammp	gammq	gnoise	inverseErf
inverseErfc	lorentzianNoise	logNormalNoise	mean
max	min	norm	poissonNoise
StatsCorrelation	StatsBetaCDF	StatsBetaPDF	StatsBinomialCDF
StatsBinomialPDF	StatsCauchyCDF	StatsCauchyPDF	StatsChiCDF
StatsChiPDF	StatsCMSSDCDF	StatsCorrelation	StatsDExpCDF
StatsDExpPDF	StatsErlangCDF	StatsErlangPDF	StatsErrorPDF
StatsEValueCDF	StatsEValuePDF	StatsExpCDF	StatsExpPDF
StatsFCDF	StatsFPDF	StatsFriedmanCDF	StatsGammaCDF
StatsGammaPDF	StatsGeometricCDF	StatsGeometricPDF	StatsHyperGCDF
StatsHyperGPDF	StatsInvBetaCDF	StatsInvBinomialCDF	StatsInvCauchyCDF
StatsInvChiCDF	StatsInvCMSSDCDF	StatsInvDExpCDF	StatsInvEValueCDF
StatsInvExpCDF	StatsInvFCDF	StatsInvFriedmanCDF	StatsInvGammaCDF
StatsInvGeometricCDF	StatsInvKuiperCDF	StatsInvLogisticCDF	StatsInvLogNormalCDF
StatsInvMaxwellCDF	StatsInvMooreCDF	StatsInvNBinomialCDF	StatsInvNCChiCDF
StatsInvNCFCDF	StatsInvNormalCDF	StatsInvParetoCDF	StatsInvPoissonCDF
StatsInvPowerCDF	StatsInvQCDF	StatsInvQpCDF	StatsInvRayleighCDF
StatsInvRectangularCDF	StatsInvSpearmanCDF	StatsInvStudentCDF	StatsInvTopDownCDF
StatsInvTriangularCDF	StatsInvUSquaredCDF	StatsInvVonMisesCDF	StatsInvWeibullCDF
StatsKuiperCDF	StatsLogisticCDF	StatsLogisticPDF	StatsLogNormalCDF
StatsLogNormalPDF	StatsMaxwellCDF	StatsMaxwellPDF	StatsMedian
StatsMooreCDF	StatsNBinomialCDF	StatsNBinomialPDF	StatsNCChiCDF
StatsNCChiPDF	StatsNCFCDF	StatsNCFPDF	StatsNCTCDF
StatsNCTPDF	StatsNormalCDF	StatsNormalPDF	StatsParetoCDF
StatsParetoPDF	StatsPermute	StatsPoissonCDF	StatsPoissonPDF
StatsPowerCDF	StatsPowerNoise	StatsPowerPDF	StatsQCDF
StatsQpCDF	StatsRayleighCDF	StatsRayleighPDF	StatsRectangularCDF
StatsRectangularPDF	StatsRunsCDF	StatsSpearmanRhoCDF	StatsStudentCDF
StatsStudentPDF	StatsTopDownCDF	StatsTriangularCDF	StatsTriangularPDF
StatsTrimmedMean	StatsUSquaredCDF	StatsVonMisesCDF	StatsVonMisesPDF
StatsWaldCDF	StatsWaldPDF	StatsWeibullCDF	StatsWeibullPDF
StudentA	StudentT	sum	Variance
WaveMax	WaveMin	wnoise	

Windows

AnnotationInfo	AnnotationList	AxisInfo	AxisList
AxisValFromPixel	ChildWindowList	ContourInfo	CsrInfo
CsrWave	CsrXWave	GuideInfo	GuideNameList
hcsr	ImageInfo	LayoutInfo	pcsr
PixelFromAxisVal	ProcedureText	qcsr	SpecialCharacterInfo

SpecialCharacterList	TagVal	TraceInfo	vcsr
WinList	WinName	WinRecreation	WinType
xcsr	XWaveName	zcsr	

Strings

AddListItem	char2num	cmpstr	FontSizeHeight
FontSizeStringWidth	GrepList	GrepString	IndexedDir
IndexedFile	LowerStr	num2char	num2istr
num2str	PadString	PossiblyQuoteName	RemoveEnding
RemoveFromList	RemoveListItem	ReplaceStringByKey	SelectString
str2num	StringByKey	stringCRC	StringFromList
StringList	stringmatch	strlen	strsearch
TextFile	UnPadString	UpperStr	URLDecode
URLEncode	WhichListItem		

Names

CheckName	CleanupName	ContourNameList	ContourNameToWaveRef
ControlNameList	CTabList	FontList	FunctionList
GetDefaultFont	GetIndependentModuleName	GetIndexedObjName	GetWavesDataFolder
GetWavesDataFolderDFR	ImageNameList	ImageNameToWaveRef	IndependentModuleList
IndexedDir	IndexedFile	MacroList	NameOfWave
StringList	TraceFromPixel	TraceNameList	TraceNameToWaveRef
UniqueName	VariableList	WaveList	WaveName
WinList	WinName	XWaveName	

Lists

AnnotationList	AxisList	ChildWindowList	ContourNameList
ControlNameList	CountObjects	CountObjectsDFR	DataFolderDir
FindListItem	FontList	FunctionInfo	FunctionList
GetIndexedObjName	GetWindow	GuideNameList	ImageNameList
IndependentModuleList	ItemsInList	ListMatch	MacroList
NumberByKey	OperationList	PathList	PICTList
RemoveByKey	RemoveFromList	RemoveListItem	ReplaceNumberByKey
ReplaceStringByKey	SortList	StringByKey	StringFromList
StringList	TableInfo	TraceNameList	VariableList
WaveList	WaveRefIndexed	WhichListItem	WinList

Programming

CaptureHistory	CaptureHistoryStart	ControlNameList	DDEExecute
DDEInitiate	DDEPokeString	DDEPokeWave	DDERequestString
DDERequestWave	DDEStatus	DDETerminate	exists
FakeData	FuncRefInfo	FunctionInfo	GetDefaultFont
GetDefaultFontSize	GetDefaultFontStyle	GetErrMessage	GetFormula
GetKeyState	GetRTError	GetRTErrMessage	GetRTStackInfo
GetScrapText	GuideInfo	GuideNameList	Hash
i	IgorInfo	ilim	j

jlim	NameOfWave	numtype	NumVarOrDefault
NVAR_Exists	ParamIsDefault	PICTInfo	PixelFromAxisVal
ProcedureText	ScreenResolution	SelectNumber	SelectString
SpecialDirPath	startMSTimer	stopMSTimer	stringCRC
StrVarOrDefault	SVAR_Exists	TableInfo	TagVal
ThreadGroupCreate	ThreadGroupGetDF	ThreadGroupGetDFR	ThreadGroupRelease
ThreadGroupWait	ThreadProcessorCount	ThreadReturnValue	WaveCRC
WinType			

Data Folders

CountObjects	DataFolderDir	DataFolderExists	DataFolderRefsEqual
DataFolderRefStatus	GetDataFolder	GetDataFolderDFR	GetIndexedObjName
GetWavesDataFolder	GetWavesDataFolderDFR	NewFreeDataFolder	

I/O (files, paths, and PICTs)

FetchURL	FunctionPath	IndexedDir	IndexedFile
ParseFilePath	PathList	PICTInfo	PICTList
SpecialDirPath	TextFile	URLDecode	URLEncode

Built-In Keywords

Procedure Declarations

End	EndMacro	EndStructure	Function
Macro	Picture	Proc	Structure
Window			

Procedure Subtypes

ButtonControl	CheckBoxControl	CursorStyle	FitFunc
Graph	GraphMarquee	GraphStyle	GridStyle
Layout	LayoutMarquee	LayoutStyle	Panel
PopupMenuControl	SetVariableControl	Table	TableStyle

Object References

DFREF	FUNCREF	NVAR	STRUCT
SVAR	WAVE		

Flow Control

AbortOnRTE	AbortOnValue	break	catch
continue	default	do-while	endtry
for-endfor	if-elseif-endif	if-endif	return
strswitch-case-endswitch	switch-case-endswitch	try-catch-endtry	

Other Programming Keywords

#define	#if-#elif-#endif	#if-#endif	#ifdef-#endif
#ifndef-#endif	#include	#pragma	#undef
Constant	DoPrompt	GalleryGlobal	IgorVersion
IndependentModule	Menu	ModuleName	MultiThread
Override	popup	ProcGlobal	Prompt
root	rtGlobals	Static	Strconstant
String	Submenu	ThreadSafe	Variable
version			

Built-in Structures

Point	Rect	RGBColor	WMAxisHookStruct
WMBackgroundStruct	WMButtonAction	WMCheckboxAction	WMCustomControlAction
WMFitInfoStruct	WMGizmoHookStruct	WMListboxAction	WMMarkerHookStruct
WMPopupAction	WMSetVariableAction	WMSliderAction	WMTabControlAction
WMWinHookStruct			

Hook Functions

See Chapter IV-10, **Advanced Programming**, **User-Defined Hook Functions** on page IV-251.

AfterCompiledHook	AfterFileOpenHook	AfterWindowCreatedHook	BeforeDebuggerOpensHook
BeforeExperimentSaveHook	BeforeFileOpenHook	IgorBeforeNewHook	IgorBeforeQuitHook
IgorMenuHook	IgorQuitHook	IgorStartOrNewHook	

Alphabetic Listing of Functions, Operations and Keywords

This section alphabetically lists all built-in functions, operations and keywords. Much of this information is also accessible online in the Command Help tab of the Igor Help Browser.

External operations (XOPs) and external functions (XFUNCs) are not covered here. For information about them, use the Command Help tab of the Igor Help Browser and the XOP help file in the same folder as the XOP file.

Reference Syntax Guide

In the descriptions of functions and operations that follow, italics indicate parameters for which you can supply numeric or string expressions. Nonitalic parameters are to be entered literally as they appear. Commas, slashes, braces and parentheses in these descriptions are always literals. Bold brackets may also be literals; they are used to specify point ranges or indices (the description will make this clear).

Nonbold brackets surround optional flags or parameters.

Ellipses (…) indicate that the preceding element may be repeated a number of times. The exact number of repetitions varies, and should be found in the description.

Italicized parameters represent values you supply. Italic words ending with "*Name*" are names (wave names, for example), and those ending with "*Str*" (and the words "*string*" or "*str*") are strings. Some Igor functions can take an empty string ("", no space between the quotation marks) as a parameter. Italic words ending with "*Spec*" (meaning "specification") are usually further defined in the description. If none of these endings are employed, the italic word is a numeric expression, such as a literal number, the name of a variable or function, or some valid combination.

Strings and names are different, but you can use a string where a name is expected using "string substitution": precede a string expression with the $ operator. See **String Substitution Using $** on page IV-15.

A syntax description may span several lines, but the actual command you create must occupy a single line. Igor has no line continuation character like FORTRAN programs or Unix command shells.

Many operations have optional "flags". Flags that accept a value (such as the Make operation's /N=*n* flag) sometimes require additional parentheses. For example:

```
Make/N=1 aNewWave
```

is acceptable because here *n* is the literal "1". To use a numeric expression for *n*, parentheses are needed:

```
Make/N=(numberOfPoints) aNewWave              // error if no parentheses!
```

And, yes, the use of just one variable constitutes an "expression"!

Some operations have multiple forms. The syntax for each is shown on a separate line.

For more about using functions, operations and keywords, see Chapter IV-1, **Working with Commands**, Chapter IV-2, **Programming Overview**, and Chapter IV-10, **Advanced Programming**.

#define

#define *symbol*
The #define statement is a conditional compilation directive that defines a *symbol* for use only with #ifdef or #ifndef expressions. #undef removes the definition.

Details
The defined *symbol* exists only in the file where it is defined; the only exception is in the main procedure window where the scope covers all other procedures except independent modules. See **Conditional Compilation** on page IV-86 for information on defining a global *symbol*.

#define cannot be combined inline with other conditional compilation directives.

See Also
The #**undef**, #**ifdef-#endif**, and #**ifndef-#endif** statements.

Conditional Compilation on page IV-86.

#if-#elif-#endif

```
#if expression1
   <TRUE part 1>
#elif expression2
   <TRUE part 2>
[…]
[#else
   <FALSE part>]
#endif
```

In a #if-#elif-#endif conditional compilation statement, when an expression evaluates as TRUE (absolute value > 0.5), then only code corresponding to the TRUE part of that expression is compiled, and then the conditional statement is exited. If all expressions evaluate as FALSE (zero) then *FALSE part* is compiled when present.

Details
Conditional compiler directives must be either entirely outside or inside function definitions; they cannot straddle a function fragment. Conditionals cannot be used within Macros.

See Also
Conditional Compilation on page IV-86 for more usage details.

#if-#endif

```
#if expression
   <TRUE part>
[#else
   <FALSE part>]
#endif
```

A #if-#endif conditional compilation statement evaluates *expression*. If *expression* is TRUE (absolute value > 0.5) then the code in *TRUE part* is compiled, or if FALSE (zero) then the optional *FALSE part* is compiled.

Details
Conditional compiler directives must be either entirely outside or inside function definitions; they cannot straddle a function fragment. Conditionals cannot be used within Macros.

See Also
Conditional Compilation on page IV-86 for more usage details.

#ifdef-#endif

```
#ifdef symbol
   <TRUE part>
[#else
   <FALSE part>]
#endif
```

A #ifdef-#endif conditional compilation statement evaluates *symbol*. When *symbol* is defined the code in *TRUE part* is compiled, or if undefined then the optional *FALSE part* is compiled.

Details

Conditional compiler directives must be either entirely outside or inside function definitions; they cannot straddle a function fragment. Conditionals cannot be used within Macros.

symbol must be defined before the conditional with #define.

See Also

The #**define** statement and **Conditional Compilation** on page IV-86 for more usage details.

#ifndef-#endif

```
#ifndef symbol
   <TRUE part>
[#else
   <FALSE part>]
#endif
```

An #ifndef-#endif conditional compilation statement evaluates *symbol*. When *symbol* is undefined the code in *TRUE part* is compiled, or if defined then the optional *FALSE part* is compiled.

Details

Conditional compiler directives must be either entirely outside or inside function definitions; they cannot straddle a function fragment. Conditionals cannot be used within Macros.

symbol must be defined before the conditional with #define.

See Also

The #**define** statement and **Conditional Compilation** on page IV-86 for more usage details.

#include

```
#include "file spec" or <file spec>
```

A #include statement in a procedure file automatically opens another procedure file. You should use #include in any procedure file that you write if it requires that another procedure file be open. A #include statement must always appear flush against the left margin in a procedure window.

Parameters

file spec is the procedure file name, which can incorporate a full or partial path. The form used depends of the procedure file location: <*file spec*> is in "Igor Pro Folder/WaveMetrics Procedures" and "*file spec*" is in "Igor Pro Folder/User Procedures" or "Igor Pro User Files/User Procedures".

See Also

The Include Statement on page IV-145 for usage details.

Igor Pro User Files on page II-46.

#pragma

```
#pragma [rtGlobals = 0, 1, or 2][IgorVersion = versNum]
        [version = versNum][ModuleName = name]
        [IndependentModule = name]
```

#pragma introduces a compiler directive, which is a message to the Igor procedure compiler. A #pragma statement must always appear flush against the left margin in a procedure window.

See Also

rtGlobals, **IndependentModule**, **ModuleName**, and **version** for details about these particular #pragma directives. Also see **Silent** and **Pragmas** on page IV-40.

#undef

```
#undef symbol
```

A #undef statement removes a nonglobal *symbol* created previously by #define. See **Conditional Compilation** on page IV-86 for information on undefining a global *symbol.*

See Also

The #**define** statement and **Conditional Compilation** on page IV-86 for more usage details.

Abort

Abort [*errorMessageStr*]

The Abort operation aborts procedure execution.

Parameters

The optional *errorMessageStr* is a string expression, which, if present, specifies the message to be displayed in the error alert.

Details

Abort provides a way for a procedure to abort execution when it runs into an error condition.

See Also

Aborting Functions on page IV-89 and **Aborting Macros** on page IV-103. The **DoAlert** operation.

AbortOnRTE

AbortOnRTE

The AbortOnRTE flow control keyword raises an abort with a runtime error. AbortOnRTE should be used after an operation that might give rise to a runtime error. It has very low overhead and should not significantly slow program execution.

Details

In terms of programming style, you should consider using AbortOnRTE (preceded by a semicolon) on the same line as the command that may give rise to an abort condition.

When using AbortOnRTE after a related sequence of commands, then it should be placed on its own line.

Example

Abort if the wave does not exist:

```
WAVE someWave; AbortOnRTE
```

See Also

Flow Control for Aborts on page IV-38 and **AbortOnRTE Keyword** on page IV-38 for further details.

The **try-catch-endtry** flow control statement.

AbortOnValue

AbortOnValue *abortCondition, abortCode*

The AbortOnValue flow control keyword will abort function execution when the *abortCondition* is nonzero and it will then return the numeric *abortCode*. No dialog will be displayed when such an abort occurs.

Parameters

abortCondition can be any valid numeric expression using comparison or logical operators.

abortCode is a nonzero numeric value returned to any abort or error handling code by AbortOnValue whenever it causes an abort.

See Also

Flow Control for Aborts on page IV-38 and **AbortOnValue Keyword** on page IV-38 for further details.

The **AbortOnRTE** keyword and the **try-catch-endtry** flow control statement.

abs

abs(*num*)

The abs function returns the absolute value of the real number *num*. To calculate the absolute value of a complex number, use the cabs function.

See Also

The **cabs** function.

acos

acos(*num*)

The acos function returns the inverse cosine of *num* in radians in the range [0,π].

In complex expressions, *num* is complex and acos returns a complex value.

acosh

acosh(*num*)

The acosh function returns the inverse hyperbolic cosine of *num*. In complex expressions, *num* is complex and acosh returns a complex value.

AddFIFOData

AddFIFOData *FIFOName, FIFO_channelExpr* [, *FIFO_channelExpr*]…

The AddFIFOData operation evaluates *FIFO_channelExpr* expressions as double precision floating point and places the resulting values into the named FIFO.

Details

There must be one *FIFO_channelExpr* for each channel in the FIFO.

See Also

FIFOs are used for data acquisition. See **FIFOs and Charts** on page IV-276.

Other operations used with FIFOs: **NewFIFO**, **NewFIFOChan**, **CtrlFIFO**, and **FIFOStatus**.

AddFIFOVectData

AddFIFOVectData *FIFOName, FIFO_channelKeyExpr* [, *FIFO_channelKeyExpr*]…

The AddFIFOVectData operation is similar to AddFIFOData except the expressions use a keyword to allow either a single numeric value for a normal channel or a wave containing the data for a special image vector channel.

Details

There must be one *FIFO_channelKeyExpr* for each channel in the FIFO.

A *FIFO_channelKeyExpr* may be one of:

```
num = numericExpression
vect = wave
```

For best results, the wave should have the same number of points as used to define the FIFO channel and the same number type. See the **NewFIFOChan** operation.

See Also

FIFOs and Charts on page IV-276.

AddListItem

AddListItem(*itemStr, listStr* [, *listSepStr* [, *itemNum*]])

The AddListItem function returns *listStr* after adding *itemStr* to it. *listStr* should contain items separated by the *listSepStr* character, such as "abc;def;".

Use AddListItem to add an item to a string containing a list of items separated by a single character, such as those returned by functions like **TraceNameList** or **AnnotationList**, or to a line from a delimited text file.

listSepStr and *itemNum* are optional; their defaults are ";" and 0, respectively.

Details

By default *itemStr* is added to the start of the list. Use the optional list index *itemNum* to add *itemStr* at a different location. The returned list will have *itemStr* at the index *itemNum* or at ItemsInList(*returnedListStr*)-1 when *itemNum* equals or exceeds ItemsInList(*listStr*).

itemNum can be any value from -infinity (`-Inf`) to infinity (`Inf`). Values from -infinity to 0 prepend *itemStr* to the list, and values from ItemsInList(*listStr*) to infinity append *itemStr* to the list.

itemStr may be `""`, in which case an empty item (consisting of only a separator) is added.

If *listSepStr* is `""`, then *listStr* is returned unchanged (unless *listStr* contains only list separators, in which case an empty string (`""`) is returned).

listStr is treated as if it ends with a *listSepStr* even if it doesn't.

Only the first character of *listSepStr* is used.

Examples

```
Print AddListItem("hello","kitty;cat;")        // prints "hello;kitty;cat;"
Print AddListItem("z", "b,c,", ",", 1)         // prints "b,z,c,"
Print AddListItem("z", "b,c,", ",", 999)       // prints "b,c,z,"
Print AddListItem("z", "b,c,", ",", Inf)       // prints "b,c,z,"
Print AddListItem("", "b-c-", "-")             // prints "-b-c-"
```

See Also

The **FindListItem**, **FunctionList**, **ItemsInList**, **RemoveByKey**, **RemoveFromList**, **RemoveListItem**, **StringFromList**, **StringList**, **TraceNameList**, **VariableList**, and **WaveList** functions.

AddMovieAudio

AddMovieAudio *soundWave*

The AddMovieAudio operation adds audio samples to the audio track of the currently open movie.

Parameters

soundWave contains audio samples with an amplitude from -128 to +127 and with the same time scale as the prototype *soundWave* used to open the movie.

Details

You can create movies with 16-bit and stereo sound by providing a sound wave in the appropriate format. To specify 16-bit sound, the wave type must be signed 16-bit integer (/W flag in **Make** or **Redimension**). To specify stereo, use a wave with two columns (or any other number of channels as desired).

See Also

The **NewMovie** operation.

AddMovieFrame

AddMovieFrame [/PICT=*pictName*]

The AddMovieFrame operation adds the top graph or the specified picture to the currently open movie.

When you write a procedure to generate a movie, you need to call the **DoUpdate** operation after all modifications to the graph and before calling AddMovieFrame. This allows Igor to process any changes you have made to the graph.

If the /PICT flag is provided, then the specified picture from the picture collection (see **Pictures** on page III-421) is used in place of the top graph. This requires Igor Pro 6.12 or later.

See Also

The **NewMovie** operation.

airyA

airyA(*x* [, *accuracy*])

The airyA function returns the value of the Airy $Ai(x)$ function:

$Ai(x) = \frac{1}{\pi}\sqrt{\frac{x}{3}}K_{1/3}\left(\frac{2}{3}x^{3/2}\right)$ where K is the modified Bessel function.

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **airyAD** and **airyB** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

airyAD

airyAD(*x* [, *accuracy*])

The airyAD function returns the value of the derivative of the Airy function.

Details
See the **bessI** function for details on accuracy and speed of execution.

See Also
The **airyA** function.

airyB

airyB(*x* [, *accuracy*])
The airyB function returns the value of the Airy $Bi(x)$ function:

$$Bi(x) = \sqrt{\frac{x}{3}}\left[I_{-1/3}\left(\frac{2}{3}x^{3/2}\right) + I_{1/3}\left(\frac{2}{3}x^{3/2}\right)\right]$$ where I is the modified Bessel function.

Details
See the **bessI** function for details on accuracy and speed of execution.

See Also
The **airyBD** and **airyA** functions.

References
Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

airyBD

airyBD(*x* [, *accuracy*])
The airyBD function returns the value of the derivative $Bi'(x)$ of the Airy function.

Details
See the **bessI** function for details on accuracy and speed of execution.

See Also
The **airyB** function.

alog

alog(*num*)
The alog function returns 10^{num}.

AnnotationInfo

AnnotationInfo(*winNameStr*, *annotationNameStr* [, *options*])
The AnnotationInfo function returns a string containing a semicolon-separated list of information about the named annotation in the named graph or page layout window or subwindow.

The main purpose of AnnotationInfo is to use a tag or textbox as an input mechanism to a procedure. This is illustrated in the "Tags as Markers Demo" sample experiment, which includes handy utility functions (supplied by AnnotationInfo Procs.ipf).

Parameters
winNameStr can be "" to refer to the top graph or layout window or subwindow.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

options is an optional parameter that controls the text formatting in the annotation output. The default value is 0.

Omit *options* or use 0 for *options* to escape the returned annotation text, which is appropriate for printing the output to the history or for using the text in an Execute operation.

Use 1 for *options* to not escape the returned annotation text because you intend to extract the text for use in a subsequent command such as Textbox or Tag.

Details
The string contains thirteen pieces of information. The first twelve pieces are prefaced by a keyword and colon and terminated with a semicolon. The last piece is the annotation text, which is prefaced with a

keyword and a colon but is not terminated with a semicolon.

Keyword	Information Following Keyword
ABSX	X location, in points, of the anchor point of the annotation. For graphs, this is relative to the top-left corner of the graph window. For layouts, it is relative to the top-left corner of the page.
ABSY	Y location, in points, of the anchor point of the annotation. For graphs, this is relative to the top-left corner of the graph window. For layouts, it is relative to the top-left corner of the page.
ATTACHX	For tags, it is the X value of the wave at the point where the tag is attached, as specified with the Tag operation. For textboxes, color scales, and legends, this will be zero and has no meaning.
AXISX	X location of the anchor point of the annotation. For tags or color scales in graphs, it is in terms of the X axis against which the tagged wave is plotted. For textboxes and legends in graphs, it is in terms of the first X axis. For layouts, this has no meaning and is always zero.
AXISY	Y location of the anchor point of the annotation. For layouts, this has no meaning and is always zero. For tags or color scales in graphs, it is in terms of the Y axis against which the tagged wave is plotted. For textboxes and legends in graphs, it is in terms of the first Y axis.
AXISZ	Z value of the image or contour level trace to which the tag is attached or NaN if the trace is not a contour level trace or the annotation is not a tag.
COLORSCALE	Parameters used in a ColorScale operation to create the annotation.
FLAGS	Flags used in a Tag, Textbox, ColorScale, or Legend operation to create the annotation.
RECT	The outermost corners of the annotation (values are in points): RECT:*left*, *top*, *right*, *bottom*
TEXT	Text that defines the contents of the annotation or the main axis label of a color scale.
TYPE	Annotation type: “Tag”, “Textbox”, “ColorScale”, or “Legend”.
XWAVE	For tags, it is the name of the X wave in the XY pair to which the tag is attached. If the tag is attached to a single wave rather than an XY pair, this will be empty. For textboxes, color scales, and legends, this will be empty and has no meaning.
XWAVEDF	For tags, the full path to the data folder containing the X wave associated with the trace to which the tag is attached. For textboxes, color scales, and legends, this will be empty and has no meaning.
YWAVE	For tags, it is the name of the trace or image to which the tag is attached. See **ModifyGraph (traces)** and **Instance Notation** on page IV-16 for discussions of trace names and instance notation. For color scales, it is the name of the wave displayed in associated the contour plot, image plot, f(z) trace, or the name of the color scale’s cindex wave. For textboxes and legends, this will be empty and has no meaning.
YWAVEDF	Full path to the data folder containing the Y wave or blank if the annotation is not a tag or color scale.

AnnotationList

`AnnotationList(`*`winNameStr`*`)`

The AnnotationList function returns a semicolon-separated list of annotation names from the named graph or page layout window or subwindow.

Parameters

winNameStr can be `""` to refer to the top graph or layout window.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

APMath

APMath [*flags*] *destStr = Expression*

The APMath operation provides arbitrary precision calculation of basic mathematical expressions. It converts the final result into the assigned string *destStr*, which can then be printed or used to represent a value (at the given precision) in another APMath operation.

Parameters

destStr Specifies a destination string for the assignment expression. If *destStr* is not an existing variable, it is created by the operation. When executing in a function, *destStr* will be a local variable if it does not already exist.

Expression Algebraic expression containing constants, local, global, and reference variables or strings, as well as wave elements together with the following operators:

Operator	Precedence
^	**Highest**
* /	
+ -	**Lowest**

Operators

Supported operators are:

+	Scalar addition.
-	Scalar subtraction.
*	Scalar multiplication.
/	Scalar division.
^	Exponentiation.

Functions

The following functions are supported:

sqrt(*x*)	Square root of *x*.
cbrt(*x*)	Cube root of *x*.
pi	Value of π (without parentheses).
sin(*x*)	Sine of *x*.
cos(*x*)	Cosine of *x*.
tan(*x*)	Tangent of *x*.
asin(*x*)	Inverse sine of *x*.
acos(*x*)	Inverse cosine of *x*.
atan(*x*)	Inverse tangent of *x*.
atan2(*y*,*x*)	Inverse tangent of *y*/*x*.
log(*x*)	Logarithm of *x*.
log10(*x*)	Logarithm based 10 of *x*.
exp(*x*)	Exponential function e^*x*.
pow(*x*,*n*)	*x* to the power *n* (*n* not necessarily integer).
sinh(*x*)	Hyperbolic sine of *x*.
cosh(*x*)	Hyperbolic cosine of *x*.
tanh(*x*)	Hyperbolic tangent of *x*.
asinh(*x*)	Inverse hyperbolic sine of *x*.
acosh(*x*)	Inverse hyperbolic cosine of *x*.
atanh(*x*)	Inverse hyperbolic tangent of *x*.
ceil(*x*)	Smallest integer larger than *x*.

`comp(x,y)`	Returns 0 for $x == y$, 1 if $x > y$ and -1 if $y > x$.
`factorial(n)`	Factorial of integer n.
`floor(x)`	Greatest integer smaller than x.
`gcd(x,y)`	Greatest common divisor of x and y.
`lcd(x,y)`	Lowest common denominator of x and y (given by $x*y$/gcd(x,y).
`sgn(x)`	Sign of x or zero if $x == 0$.

Flags

/EX=*exDigits*	Specifies the number of extra digits added to the precision digits (/N) for intermediate steps in the calculation.
/N=*numDigits*	Specifies the precision of the final result. To add digits to the intermediate computation steps, use /EX.
/V	Verbose mode; prints the result in the history in addition to performing the assignment.
/Z	No error reporting.

Details

By default, all arbitrary precision math calculations are performed with *numDigits*=50 and *exDigits*=6, which yields a final result using at least 56 decimal places. Because none of the built-in variable types can express numbers with such high accuracy, the arbitrary precision numbers must be stored as strings. The operation automatically converts between strings and constants. It evaluates all of the numerical functions listed above using the specified accuracy. If you need functions that are not supported by this operation, you may have to precompute them and store the results in a local variable.

The operation stores the result in *destStr*, which may or may not exist prior to execution. When you execute the operation from the command line, *destStr* becomes a global string in the current data folder if it does not already exist. If it exists, then the result of the operation overwrites its value (as with any normal string assignment). In a user function, *destStr* can be a local string, an SVAR, or a string passed by reference. If *destStr* is not one of these then the operation creates a local string by that name.

Arbitrary precision math calculations are much slower (by a factor of about 300) than equivalent floating point calculations. Execution time is a function of the number of digits, so you should use the /N flag to limit the evaluation to the minimum number of required digits.

Examples

Evaluate pi to 50 digits:

```
APMath/V aa=pi
```

Evaluate ratios of large factorials:

```
APMath/v aa=factorial(500)/factorial(499)
```

Evaluate ratios of large exponentials:

```
APMath/v aa=exp(-1000)/exp(-1001)
```

Division of mixed size values:

```
APMath/v aa=1-sgn(1-(1-0.00000000000000000001234)/(1-0.000000000000000000012345)))
```

you'll get a different result trying to evaluate this using double precision.

Difference between built-in pi and the arbitrary precision pi:

```
Variable/G biPi=pi
APMath/v aa=biPi-pi
```

Precision control:

```
Function test()
   APMath aa=pi                  // Assign 50 digit pi to the string aa.
   APMath/v bb=aa                // Create local string bb equal to aa.
   APMath/v bb=aa-pi             // Subtract arb. prec. pi from aa.
                                 // note the default exDigits=6.
   APMath/v/n=50/ex=0 bb=aa-pi   // setting exDigits=0.
End
```

Numerical recreation:

```
APMath/v/n=16 aa=111111111^2
```

Append

Append

The Append operation is interpreted as **AppendToGraph**, **AppendToTable**, or **AppendToLayout**, depending on the target window. This does not work when executing a user-defined function. Therefore we now recommend that you use **AppendToGraph**, **AppendToTable**, or **AppendLayoutObject** rather than Append.

AppendImage

AppendImage [**/G=*g*/W=*winName***][***axisFlags***] ***matrix*** [**vs** {***xwaveName*, *ywaveName***}]

The AppendImage operation appends the matrix as an image to the target or named graph. By default the image is plotted versus the left and bottom axes.

Parameters

matrix is either an NxM matrix for false color or indexed color images or can be a 3D NxMx3 wave containing a layer of data for red, a layer for green and a layer for blue. If matrix contains multiple planes other than three or if it contains 3 planes and multiple chunks, the **ModifyImage** plane keyword can be used to specify the desired subset to display.

If you provide *xwaveName* and *ywaveName*, *xwaveName* provides X coordinate values, and *ywaveName* provides Y coordinate values. This makes an image with uneven pixel sizes. In both cases, you can use * to specify calculated values. See **Details** if you use *xwaveName* or *ywaveName*.

Flags

axisFlags	Flags /L, /R, /B, and /T are the same as used by **AppendToGraph**.
/G=*g*	*g*=1: Suppresses the autodetection of three plane images as direct (RGB) color.
	g= 0: Same as no /G flag (default).
/W=*winName*	Appends to the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.
	When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

When appending an image to a graph, you can supply optional X and Y waves to define the coordinates of the rectangle edges. *These waves need to contain one more data point than the X (row) or Y (column) dimension of the matrix to define the end of the last rectangle*.

For false color, the values in the matrix are linearly mapped into a color table. See the **ModifyImage ctab** keyword. For indexed color, the values in the matrix are interpreted as Z values to be looked up in a user-supplied 3 column matrix of colors. See the **ModifyImage cindex** keyword. Direct color NxMx3 waves contain the actual red, green, and blue values for each pixel. If the number type is unsigned bytes, then the range of intensity ranges from 0 to 255. For all other number types, the intensity ranges from 0 to 65535.

By default, nondirect color matrices are initially displayed as false color using the Grays color table and autoscale mode.

If the matrix is complex, the image is displayed in terms of the magnitude of the Z value, that is, $\text{sqrt}(\text{real}^2 + \text{imag}^2)$.

See Also

The **NewImage**, **ModifyImage**, and **RemoveImage** operations. For general information on image plots see Chapter II-15, **Image Plots**.

AppendLayoutObject

AppendLayoutObject [***flags***] ***objectType objectName***

The AppendLayoutObject operation appends a single object to the top layout or to the layout specified via the /W flag.

Unlike the AppendToLayout operation, AppendLayoutObject can be used in user-defined functions. Therefore, AppendLayoutObject should be used in new programming instead of AppendToLayout.

Parameters

objectType identifies the type of object to be appended. It is one of the following keywords: `graph`, `table`, `picture`.

objectName is the name of the graph, table or picture to be appended.

Use a space between *objectType* and *objectName*. A comma is not allowed.

Flags

/D=*fidelity*	Draws layout objects in low fidelity (*fidelity*=0) or high fidelity (*fidelity*=1; default). This affects drawing on the screen only, not exporting or printing. Low fidelity is somewhat faster but less accurate and should be used only for graphs that take a very long time to draw.
/F=*frame*	Specifies the type of frame enclosing the object. *frame* =0: None. *frame* =1: Single frame (default). *frame* =2: Double frame. *frame* =3: Triple frame. *frame* =4: Shadow frame.
/T=*trans*	Sets the transparency of the object background to opaque (*trans* =0; default) or transparent (*trans* =1). For transparency to be effective, the object itself must also be transparent. Annotations have their own transparent/opaque settings. Graphs are transparent only if their backgrounds are white. PICTs may have been created transparent or opaque. Opaque PICTs cannot be made transparent.
/R=(*l*, *t*, *r*, *b*)	Sets the size and position of the object. If omitted, the object is placed with a default size and position. *l*, *t*, *r*, and *b* are the left, top, right, and bottom coordinates of the object, respectively. Coordinates are expressed in units of points, relative to the top/left corner of the paper.
/W=*winName*	Appends the object to the named page layout window. If /W is omitted or if *winName* is `$""`, the top page layout is used.

See Also

NewLayout, **ModifyLayout**, **RemoveLayoutObjects**, **TextBox**, and **Legend**.

AppendMatrixContour

AppendMatrixContour [***axisFlags***] [**/F=*formatStr* /W=*winName***] ***zWave***
[**vs** {***xWave, yWave***}]

The AppendMatrixContour operation appends to the target or named graph a contour plot of a matrix of z values with autoscaled contour levels, using the Rainbow color table.

Note: There is no DisplayContour operation. Use `Display; AppendMatrixContour`.

Parameters

zWave must be a matrix (2D wave).

To contour a set of XYZ triplets, use **AppendXYZContour**.

If you provide the *xWave* and *yWave* specification, *xWave* provides X values for the rows, and *yWave* provides Y values for the columns. This results in an "uneven grid" of Z values.

If you omit the *xWave* and *yWave* specification, Igor uses the *zWave*'s X and Y scaled indices as the X and Y values. Igor also uses the *zWave*'s scaled indices if you use * (asterisk symbol) in place of *xWave* or *yWave*.

In a procedure, to modify the appearance of contour levels before the contour is calculated and displayed with the default values, append "`;DelayUpdate`" and immediately follow the AppendMatrixContour command with the appropriate **ModifyContour** commands. All but the last ModifyContour command should also have `;DelayUpdate` appended.

On the command line, the Display command and following AppendMatrixContour and ModifyContour commands can be typed all on one line with semicolons between:

```
Display; AppendMatrixContour MyMatrix; ModifyContour …
```

Flags

axisFlags	Flags /L, /R, /B, /T are the same as used by **AppendToGraph**.
/F=*formatStr*	Determines the names assigned to the contour level traces. See **Details**.
/W=*winName*	Appends to the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

AppendMatrixContour creates and displays contour level traces. You can modify these all together using the Modify Contour Appearance dialog or individually using the Modify Trace Appearance dialog. In most cases, you will not need to modify the individual traces.

By default, Contour level traces are automatically named with names that show the *zWave* and the contour level, for example, "zWave=1.5". You will see these trace names in the Modify Trace Appearance dialog and in Legends. In most cases, the default trace names will be just fine.

If you want to control the names of the contour level traces (which you might want to do for names in a Legend), use the /F=*formatStr* flag. This flag uses a format string as described for the **printf** operation. The default format string is `"%.17s=%g"`, resulting in trace names such as "zWave=1.5". *formatStr* must contain at least `%f` or `%g` (used to insert the contour level) or `%d` (used to insert the zero-based index of the contour level). Include `%s`, to insert the *zWave* name.

Here are some examples of format strings.

formatStr	Examples of Resulting Name	Format
`"%g"`	"100", "1e6", "-2.05e-2"	(<level>)
`"z=%g"`	"z=100", "z=1e6", "z=-2.05e-2"	(z=<level>)
`"%s %f"`	"zWave 100.000000"	(<wave>, space, <level>)
`"[%d]=%g"`	"[0]=100", "[1]=1e6"	([<index>]=<level>)

Examples

```
Make/O/N=(25,25) w2D                    // Make a matrix
SetScale x -1, 1, w2D                   // Set row scaling
SetScale y -1, 1, w2D                   // Set column scaling
w2D = sin(x) * cos(y)                   // Store values in the matrix
Display; AppendMatrixContour w2D; DelayUpdate
ModifyContour w2D autoLevels={*,*,9}    // roughly 9 automatic levels
```

See also

AppendToGraph for details about other axis flags. The **AppendXYZContour**, **ModifyContour**, and **RemoveContour** operations. For general information on contour plots, see Chapter II-14, **Contour Plots**.

AppendText

AppendText [**/W=*winName*/N/NOCR** [**=*n***]] ***textStr***

The AppendText operation appends a carriage return and *textStr* to the most recently created or changed annotation, or to the named annotation in the target or graph or layout window. Annotations include tags, textboxes, color scales, and legends.

Parameters

textStr can contain escape codes to control font, font size and other stylistic variations.

Flags

/N=*name*	Appends *textStr* to the named tag or textbox.
/NOCR[=*n*]	Omits the initial appending of a carriage return (allows a long line to be created with multiple AppendText commands). /NOCR=0 is the same as no /NOCR, and /NOCR=1 is the same as just /NOCR.
/W=*winName*	Appends to an annotation in the named graph, layout window, or subwindow. Without /W, AppendText appends to an annotation in the topmost graph or layout

window or subwindow. This must be the first flag specified when AppendText is used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

A textbox, tag, or legend can contain at most 100 lines. A color scale can have at most one line, and this line is the color scale's main axis label.

See Also

The **Tag**, **TextBox**, **ColorScale**, **ReplaceText**, and **Legend** operations.

AppendToGraph

AppendToGraph [***flags***] ***waveName*** [**,** ***waveName***]…[**vs** ***xwaveName***]

The AppendToGraph operation appends the named waves to the target or named graph. By default the waves are plotted versus the left and bottom axes.

Parameters

The *waveName*s parameters are the names of existing waves.

vs *xwaveName* plots the data values of *waveName*s against the data values of *xwaveName*.

Subsets of data, including individual rows or columns from a matrix, may be specified using **Subrange Display Syntax** on page II-288.

You can provide a custom name for a trace by appending /TN=traceName to the waveName specification. This may be useful when displaying waves with the same name but from different data folders. See **User-defined Trace Names** on page IV-71 for more information. This feature was added in Igor Pro 6.20.

Flags

/B [=*axisName*]	Plots X coordinates versus the standard or named bottom axis.
/C=(*r*,*g*,*b*)	*r*, *g*, and *b* specify the amount of red, green, and blue in the color of the appended waves as an integer from 0 to 65535.
/L [=*axisName*]	Plots Y coordinates versus the standard or named left axis.
/NCAT	Causes trace to be plotted normally on what otherwise is a category plot. X values are just category numbers but can be fractional. Category numbers start from zero. This can be used to overlay the original data points for a box plot.
/Q	Uses a special, quick update mode when appending to a pair of existing axes. A side effect of this mode is that waves that are appended are marked as not modified. This will prevent other graphs containing these waves, if any, from being updated properly.
/R [=*axisName*]	Plots Y coordinates versus the standard or named right axis.
/T [=*axisName*]	Plots X coordinates versus the standard or named top axis.
/VERT	Plots data vertically. Similar to SwapXY (**ModifyGraph (axes)**) but on a trace-by-trace basis.
/W=*winName*	Appends to the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

See Also

The **Display** operation.

AppendToLayout

AppendToLayout [***flags***] ***objectSpec*** [**,** ***objectSpec***]…

The AppendToLayout operation appends the specified objects to the top layout.

The AppendToLayout operation can not be used in user-defined functions. Use the AppendLayoutObject operation instead.

Parameters

The optional *objectSpec* parameters identify a graph, table, textbox or PICT to be added to the layout. An object specification can also specify the location and size of the object, whether the object should have a frame or not, whether it should be transparent or opaque, and whether it should be displayed in high fidelity or not. See the **Layout** operation for details.

Flags

/G=*g*	Specifies grout, the spacing between tiled objects. Units are points unless /I, /M, or /R are specified.
/I	*objectSpec* coordinates are in inches.
/M	*objectSpec* coordinates are in centimeters.
/R	*objectSpec* coordinates are in percent of printing part of the page.
/S	Stacks objects.
/T	Tiles objects.

See Also

The **Layout** and **AppendLayoutObject** operations for use with user-defined functions.

AppendToTable

AppendToTable [**/W=*winName***] ***columnSpec*** [**, *columnSpec***]…

The AppendToTable operation appends the specified columns to the top table. *columnSpec*s are the same as for the **Edit** operation; usually they are just the names of waves.

Flags

/W=*winName*	Appends columns to the named table window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

See Also

Edit for details about *columnSpec*s, and **RemoveFromTable**.

AppendXYZContour

AppendXYZContour [**/W=*winName* /F=*formatStr***][***axisFlags***] ***zWave*** [**vs** {***xWave, yWave***}]

The AppendXYZContour operation appends to the target or named graph a contour of a 2D wave consisting of XYZ triples with autoscaled contour levels and using the Rainbow color table.

To contour a matrix of Z values, use **AppendMatrixContour**.

Note: There is no DisplayContour operation. Use `Display; AppendXYZContour`.

Parameters

If you provide the *xWave* and *yWave* specification, *xWave* provides X values for the rows, and *yWave* provides Y values for the columns, *zWave* provides Z values and all three waves must be 1D. All must have at least four rows and must have the same number of rows.

If you omit the *xWave* and *yWave* specification, *zWave* must be a 2D wave with 4 or more rows and 3 or more columns. The first column is X, the second is Y, and the third is Z. Any additional columns are ignored.

If any of X, Y, or Z in a row is blank, (NaN), that row is ignored.

To modify the appearance of contour levels before the contour is calculated and displayed with the default values, append "`;DelayUpdate`" and immediately follow the AppendXYZContour command with the appropriate **ModifyContour** commands. All but the last **ModifyContour** command should also have `;DelayUpdate` appended

On the command line, the **Display** command and subsequent AppendXYZContour commands and any **ModifyContour** commands can be typed all on one line with semicolons between:

```
Display; AppendXYZContour MyMatrix; ModifyContour …
```

Flags

axisFlags	Flags /L, /R, /B, and /T are the same as used by **AppendToGraph**.
/F=*formatStr*	Determines names assigned to the contour level traces. This is the same as for **AppendMatrixContour**.
/W=*winName*	Appends to the named graph window or subwindow. When omitted, action affects the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

AppendXYZContour creates and displays contour level traces. You can modify these as a group using the Modify Contour Appearance dialog or individually using the Modify Trace Appearance dialog. In most cases, you will have no need to modify the traces individually.

See **AppendMatrixContour** for a discussion of how the contour level traces are named.

Examples

```
Make/O/N=(100) xW, yW, zW               // Make X, Y, and Z waves
xW = sawtooth(2*PI*p/10)                // Generate X values
yW = trunc(p/10)/10                     // Generate Y values
zW = sin(2*PI*xW)*cos(2*PI*yW)          // Generate Z values
Display; AppendXYZContour zW vs {xW, yW}; DelayUpdate
ModifyContour zW autoLevels={*,*,9}     // roughly 9 automatic levels
```

area

area(*waveName* [, *x1*, *x2*])

The area function returns the signed area between the named wave and the line y=0 from x=*x1* to x=*x2* using trapezoidal integration, accounting for the wave's X scaling. If your data are in the form of an XY pair of waves, see **areaXY**.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If *x1* or *x2* are not within the X range of *waveName*, area limits them to the nearest X range limit of *waveName*.

If any values in the X range are NaN, area returns NaN.

Reversing the order of *x1* and *x2* changes the sign of the returned area.

The area operation is intended to work on 1D waves only.

Examples

```
Make/O/N=100 data;SetScale/I x 0,Pi,data
data=sin(x)
Print area(data,0,Pi)          // the entire X range, and no more
Print area(data)               // same as -infinity to +infinity
Print area(data,Inf,-Inf)      // +infinity to -infinity
```

The following is printed to the history area:

```
•Print area(data,0,Pi)         // the entire X range, and no more
  1.99983
•Print Print area(data)        // same as -infinity to +infinity
  1.99983
•Print area(data,Inf,-Inf)     // +infinity to -infinity
  -1.99983
```

The `-Inf` value was limited to 0 and `Inf` was limited to `Pi` to keep them within the X range of data.

See Also

The figure "Comparison of area, faverage and mean functions over interval (12.75,13.32)", in the **Details** section of the **faverage** function.

The **Integrate** operation.

For XY data, see **areaXY**.

areaXY

areaXY(*XWaveName, YWaveName* [, *x1, x2*])

The areaXY function returns the signed area between the named *YWaveName* and the line y=0 from x=*x1* to x=*x2* using trapezoidal integration with X values supplied by *XWaveName*.

This function is identical to the **area** function except that it works on an XY wave pair.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If *x1* or *x2* are outside the X range of *XWaveName*, areaXY limits them to the nearest X range limit of *XWaveName*.

If any Y values in the specified X range are NaN, areaXY returns NaN.

Reversing the order of *x1* and *x2* changes the sign of the returned area.

If *x1* or *x2* are not found in *XWaveName*, a Y value is found by linear interpolation based on the two bracketing X values and the corresponding values from *YWaveName*.

The values in *XWaveName* may be increasing or decreasing. AreaXY assumes that the values in *XWaveName* are monotonic. If they are not monotonic, Igor does not complain, but the result is not meaningful. If any X values are NaN, the result is NaN.

See the figure "Comparison of area, faverage and mean functions over interval (12.75,13.32)", in the **Details** section of the **faverage** function.

The areaXY operation is intended to work on 1D waves only.

Examples

```
Make/O/N=101 Xdata, Ydata
Xdata = x*pi/100
Ydata = sin(Xdata[p])
Print areaXY(Xdata, Ydata,0,Pi)         // the entire X range, and no more
Print areaXY(Xdata, Ydata)              // same as -infinity to +infinity
Print areaXY(Xdata, Ydata,Inf,-Inf)     // +infinity to -infinity
```

The following is printed to the history area:

```
•Print areaXY(Xdata, Ydata,0,Pi)        // the entire X range, and no more
  1.99984
•Print areaXY(Xdata, Ydata)             // same as -infinity to +infinity
  1.99984
•Print areaXY(Xdata, Ydata,Inf,-Inf)    // +infinity to -infinity
  -1.99984
```

The `-Inf` value was limited to `0`, and `Inf` was limited to `Pi` to stay within the X range of data.

See Also

The **Integrate** operation and the **area**, **faverage**, **faverageXY**, and **PolygonArea** functions.

asin

asin(*num*)

The asin function returns the inverse sine of num in radians in the range [-π/2,π/2].

In complex expressions, *num* is complex, and asin returns a complex value.

asinh

asinh(*num*)

The asinh function returns the inverse hyperbolic sine of *num*. In complex expressions, *num* is complex, and asinh returns a complex value.

atan

atan(*num*)

The atan function returns the inverse tangent of *num* in radians. In complex expressions, *num* is complex, and atan returns a complex value. Results are in the range -π/2 to π/2.

atan2

atan2(*y1, x1*)

The atan2 function returns the angle in radians whose tangent is *y1*/*x1*. Results are in the range -π to π.

atanh

atanh(*num*)

The atanh function returns the inverse hyperbolic tangent of *num*. In complex expressions, *num* is complex, and atanh returns a complex value.

AutoPositionWindow

AutoPositionWindow [**/E/M=*m*/R=*relWindow***] [***windowName***]

The AutoPositionWindow operation positions the window specified by *windowName* relative to the next lower window of the same kind or relative to the window given by the /R flag. If *windowName* is not specified, AutoPositionWindow acts on the target window.

Flags

/E — Uses entire area of the monitor. Otherwise, it takes into account the command window.

/M=*m* — Specifies the window positioning method.

- *m*=0: Positions *windowName* to the right of the other window, if possible. If there is no room, then it positions *windowName* just below the other window but at the left edge of the display area. If that is not possible, then the position is not affected.
- *m*=1: Positions *windowName* just under the other window lined up on the left edge, if possible. If there is no room, then it positions *windowName* just to the right of the other window lined up on the bottom edges. If neither are possible then it positions *windowName* as far to the bottom and right as it will go.

/R=*relWindow* — Positions *windowName* relative to *relWindow*.

AxisInfo

AxisInfo(*graphNameStr, axisNameStr*)

The AxisInfo function returns a string containing a semicolon-separated list of information about the named axis in the named graph window or subwindow.

Parameters

graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

axisNameStr is the name of the graph axis.

Details

The string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with a semicolon. The keywords are:

Keyword	Information Following Keyword
AXFLAG	Flag used to select the axis in any of the operations that display waves (Display, AppendMatrixContour, AppendImage, etc.).
AXTYPE	Axis type, such as “left”, “right”, “top”, or “bottom”.
CATWAVE	Wave supplying the categories for the axis if this is a category plot.
CATWAVEDF	Full path to data folder containing category wave.
CWAVE	Name of wave controlling named axis.
CWAVEDF	Full path to data folder containing controlling wave.
HOOK	Name set by ModifyFreeAxis with hook keyword.
ISCAT	Truth that this is a category axis (used in a category plot).

Keyword	Information Following Keyword
ISTFREE	Truth that this is truly free axis (created via NewFreeAxis).
MASTERAXIS	Name set by ModifyFreeAxis with master keyword.
RECREATION	List of keyword commands as used by ModifyGraph command. The format of these keyword commands is: *keyword*(x)=*modifyParameters*;
SETAXISCMD	Full SetAxis command.
SETAXISFLAGS	Flags that would be used with the SetAxis function to set the particular auto-scaling behavior that the axis uses. If the axis uses a manual axis range, SETAXISFLAGS is blank.
UNITS	Axis units, if any.

The format of the RECREATION information is designed so that you can extract a keyword command from the keyword up to the ";", prepend "ModifyGraph", replace the "x" with the name of an actual axis and then **Execute** the resultant string as a command.

Examples

```
Make/O data=x;Display data
Print StringByKey("CWAVE", AxisInfo("","left"))      // prints data
```

See Also

The **StringByKey** and **NumberByKey** functions.

The **GetAxis** and **SetAxis** operations.

The #include <Readback ModifyStr> procedures are useful for parsing strings returned by AxisInfo.

AxisList

AxisList(*graphNameStr*)

The AxisList function returns a semicolon-separated list of axis names from the named graph window or subwindow.

Parameters

graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Examples

```
Make/O data=x;Display/L/T data
Print AxisList("")               // prints left;top;
```

AxisValFromPixel

AxisValFromPixel(*graphNameStr, axNameStr, pixel*)

The AxisValFromPixel function returns an axis value corresponding to the local graph pixel coordinate in the graph window or subwindow.

Parameters

graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

If the specified axis is not found and if the name is "left" or "bottom" then the first vertical or horizontal axis will be used. Sources for *pixel* value may be the GetWindow operation or a user window hook with the mousemoved and mousedown event messages (see the **SetWindow** operation).

If *graphNameStr* references a subwindow, *pixel* is relative to top left corner of base window, not the subwindow.

See Also
The **PixelFromAxisVal** and **TraceFromPixel** functions; the **GetWindow** and **SetWindow** operations.

BackgroundInfo

BackgroundInfo

The BackgroundInfo operation returns information about the current unnamed background task.

BackgroundInfo works only with the unnamed background task. New code should used named background tasks instead. See **Background Tasks** on page IV-279 for details.

Details

Information is returned via the following variables:

V_flag	0:	No background task is defined.
	1:	Background task is defined, but not running (is idle).
	2:	Background task is defined and is running.
V_period	DeltaTicks value set by CtrlBackground. This is how often the background task runs.	
V_nextRun	Ticks value when the task will run again. 0 if the task is not scheduled to run again.	
S_value	Text of the numeric expression that the background task executes, as set by SetBackground.	

See Also

The **SetBackground**, **CtrlBackground**, **CtrlNamedBackground**, **KillBackground**, and **SetProcessSleep** operations, and the **ticks** function. See **Background Tasks** on page IV-279 for usage details.

Beep

Beep

The Beep operation plays the current alert sound (*Macintosh*) or the system beep sound (*Windows*).

Besseli

Besseli(*n*,*z*)

The Besseli function returns the modified Bessel function of the first kind, I*n*(*z*), of order *n* and argument *z*. Replaces the bessI function, which is supported for backwards compatibility only.

If *z* is real, Besseli returns a real value, which means that if *z* is also negative, it returns NaN unless *n* is an integer.

For complex *z* a complex value is returned, and there are no restrictions on *z* except for possible overflow.

Details

The calculation is performed using the SLATEC library. The function supports fractional and negative orders *n*, as well as real or complex arguments *z*.

See Also

The **Besselj**, **Besselk**, and **Bessely** functions.

Besselj

Besselj(*n*,*z*)

The Besselj function returns the Bessel function of the first kind, J*n* (*z*), of order *n* and argument *z*. Replaces the bessJ function, which is supported for backwards compatibility only.

If *z* is real, Besselj returns a real value, which means that if *z* is also negative, it returns NaN unless *n* is an integer.

For complex *z* a complex value is returned, and there are no restrictions on *z* except for possible overflow.

Details

The calculation is performed using the SLATEC library. The function supports fractional and negative orders *n*, as well as real or complex arguments *z*.

See Also

The **Besseli**, **Besselk**, and **Bessely** functions.

Besselk

Besselk(*n*,*z*)

The Besselk function returns the modified Bessel function of the second kind, K*n*(*z*), of order *n* and argument *z*. Replaces the bessK function, which is supported for backwards compatibility only.

If *z* is real, Besselk returns a real value, which means that if *z* is also negative, it returns NaN unless *n* is an integer.

For complex *z* a complex value is returned, and there are no restrictions on *z* except for possible overflow.

Details

The calculation is performed using the SLATEC library. The function supports fractional orders *n*, as well as real or complex arguments *z*.

See Also

The **Besseli**, **Besselj**, and **Bessely** functions.

Bessely

Bessely(*n*,*z*)

The Bessely function returns the Bessel function of the second kind, Y*n*(*z*), of order *n* and argument *z*. Replaces the bessY function, which is supported for backwards compatibility only.

If *z* is real, Bessely returns a real value, which means that if *z* is also negative, it returns NaN unless *n* is an integer.

For complex *z* a complex value is returned, and there are no restrictions on *z* except for possible overflow.

Details

The calculation is performed using the SLATEC library. The function supports fractional and negative orders *n*, as well as real or complex arguments *z*.

See Also

The **Besseli**, **Besselj**, and **Besselk** functions.

bessI

bessI(*n*, *x* [, *algorithm* [, *accuracy*]])

Obsolete — use **Besseli**.

The bessI function returns the modified Bessel function of the first kind, I*n*(*x*) of order *n* and argument *x*.

For real *x*, the optional parameter *algorithm* selects between a faster, less accurate calculation method and slower, more accurate methods. In addition, when *algorithm* is zero or absent, the order *n* is truncated to an integer.

When *algorithm* is included and is 1, *accuracy* can be used to specify the desired fractional accuracy. See Details about algorithms.

If *x* is complex, a complex result is returned. In this case, *algorithm* and *accuracy* are ignored. The order *n* can be fractional, and must be real.

Details

The *algorithm* parameter has three options, each selecting a different calculation method:

Algorithm	What You Get
0 (default)	Uses a calculation method that has fractional accuracy better than 10^{-6} everywhere and is generally better than 10^{-8}. This method does not handle fractional order *n*; the order is truncated to an integer before the calculation is performed. Algorithm 0 is fastest by a large margin.
1	Allows fractional order. The calculation is performed using methods described in *Numerical Recipes in C*, 2nd edition, pp. 240-245. Using algorithm 1, *accuracy* specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 1e-7 (that is, 10^{-7}), that means that you wish that the absolute value of $(f_{actual} - f_{returned})/f_{actual}$ be better than 10^{-7}. Asking for less accuracy gives some increase in speed.

Algorithm	What You Get
	You pay a heavy price for higher accuracy or fractional order. When *algorithm* is nonzero, calculation time is increased by an order of magnitude for small *x*; at larger *x* the penalty is even greater. If accuracy is greater than 10^{-8} and *n* is an integer, algorithm 0 is used. The algorithm calculates bessI and bessK simultaneously. Both values are stored, and if a call to bessI is followed by a call to bessK (or bessK is followed by bessI) with the same *n*, *x*, and *accuracy* the previously-stored value is returned, making the second call very fast.
2	Fractional order is allowed. The calculation is performed using code from the SLATEC library. The accuracy achievable is often better than algorithm 1, but not always. Algorithm 2 is 1.5 to 3 times faster than algorithm 1, but still slower than algorithm 0. The accuracy parameter is ignored.

The achievable accuracy of algorithms 1 and 2 is a complicated function *n* and *x*. To see a summary of achievable accuracies choose File→Example Experiments→Testing and Misc→Bessel Accuracy menu item.

bessJ

bessJ(*n*, *x* [, *algorithm* [, *accuracy*]])

Obsolete — use **Besselj**.

The bessJ function returns the Bessel function of the first kind, J*n*(*x*) of order *n* and argument *x*.

For real *x*, the optional parameter *algorithm* selects between a faster, less accurate calculation method and slower, more accurate methods. In addition, when *algorithm* is zero or absent, the order *n* is truncated to an integer.

When *algorithm* is included and is 1, *accuracy* can be used to specify the desired fractional accuracy. See Details about algorithms.

If *x* is complex, a complex result is returned. In this case, *algorithm* and *accuracy* are ignored. The order *n* can be fractional, and must be real.

Details

See the **bessI** function for details on algorithms, accuracy and speed of execution.

When *algorithm* is 1, pairs of values for bessJ and bessY are calculated simultaneously. The values are stored, and a subsequent call to bessY after a call to bessJ (or vice versa) with the same *n*, *x*, and *accuracy* will be very fast.

bessK

bessK(*n*, *x* [, *algorithm* [, *accuracy*]])

Obsolete — use **Besselk**.

The bessK function returns the modified Bessel function of the second kind, K*n*(*x*) of order *n* and argument *x*.

For real *x*, the optional parameter *algorithm* selects between a faster, less accurate calculation method and slower, more accurate methods. In addition, when *algorithm* is zero or absent, the order *n* is truncated to an integer.

When *algorithm* is included and is 1, *accuracy* can be used to specify the desired fractional accuracy. See Details about algorithms.

If *x* is complex, a complex result is returned. In this case, *algorithm* and *accuracy* are ignored. The order *n* can be fractional, and must be real.

Details

See the **bessI** function for details on algorithms, accuracy and speed of execution.

When *algorithm* is 1, pairs of values for bessJ and bessY are calculated simultaneously. The values are stored, and a subsequent call to bessY after a call to bessJ (or vice versa) with the same *n*, *x*, and *accuracy* will be very fast.

bessY

bessY(*n*, *x* [, *algorithm* [, *accuracy*]])

Obsolete — use **Bessely**.

The bessY function returns the Bessel function of the second kind, Y*n*(*x*) of order *n* and argument *x*.

For real *x*, the optional parameter *algorithm* selects between a faster, less accurate calculation method and slower, more accurate methods. In addition, when *algorithm* is zero or absent, the order *n* is truncated to an integer.

When *algorithm* is included and is 1, *accuracy* can be used to specify the desired fractional accuracy. See Details about algorithms.

If *x* is complex, a complex result is returned. In this case, *algorithm* and *accuracy* are ignored. The order *n* can be fractional, and must be real.

Details

See the **bessI** function for details on algorithms, accuracy and speed of execution.

When *algorithm* is 1, pairs of values for bessJ and bessY are calculated simultaneously. The values are stored, and a subsequent call to bessY after a call to bessJ (or vice versa) with the same *n*, *x*, and *accuracy* will be very fast.

beta

beta(*a*, *b*)

The beta function returns for real or complex arguments Gamma(*a*)Gamma(*b*)/Gamma(*a* + *b*) with *Re*(*a*), *Re*(*b*)>0.

See Also

The **gamma** function.

betai

betai(*a*, *b*, *x* [, *accuracy*])

The betai function returns the regularized incomplete beta function I*x*(*a*,*b*), where *a*,*b* > 0, and 0 ≤ *x* ≤ 1. Optionally, *accuracy* can be used to specify the desired fractional accuracy.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of $(f_{actual} - f_{returned})/f_{actual}$ be less than 10^{-7}.

Larger values of *accuracy* (poorer accuracy) result in evaluation of fewer terms of a series, which means the function executes somewhat faster.

A single-precision level of accuracy is about $3x10^{-7}$, double-precision is about $2x10^{-16}$. The betai function will return full double-precision accuracy for small values of *a* and *b*. Achievable accuracy declines as *a* and *b* increase:

a	*b*	*x*	betai	Accuracy Achievable
1	1.5	0.5	0.646447	$2x10^{-16}$ (full double precision)
8	10	0.5	0.685470	$6x10^{-16}$
20	21	0.5	0.562685	$2x10^{-15}$
20	21	0.1	$1.87186x10^{-10}$	$5x10^{-15}$

BinarySearch

BinarySearch(*waveName*, *val*)

The BinarySearch function performs a binary search of the one-dimensional *waveName* for the value *val*. BinarySearch returns an integer point number p such that *waveName*[p] and *waveName*[p+1] bracket *val*. If *val* is in *waveName*, then *waveName*[p]==*val*.

Details

BinarySearch is useful for finding the point in an XY pair that corresponds to a particular X coordinate.

WaveName must contain monotonically increasing or decreasing values.

BinarySearch returns -1 if *val* is not within the range of values in the wave, but would numerically be placed before the first value in the wave.

BinarySearch returns -2 if *val* is not within the range of values in the wave, but would fall after the last value in the wave.

BinarySearch returns -3 if the wave has zero points.

Examples

```
Make/O data = {1, 2, 3.3, 4.9}       // Monotonic increasing
Print BinarySearch(data,3)           // Prints 1
// BinarySearch returns 1 because data[1] <= 3 < data[2].
Make/O data = {9, 4, 3, -6}          // Monotonic decreasing
Print BinarySearch(data,2.5)         // Prints 2
// BinarySearch returns 2 because data[2] >= 2.5 > data[3].
Print BinarySearch(data,10)          // Prints -1, precedes first value
Print BinarySearch(data,-99)         // Prints -2, beyond last value
```

See Also

The **BinarySearchInterp** and **FindLevel** operations. See **Indexing and Subranges** on page II-95.

BinarySearchInterp

BinarySearchInterp(*waveName, val*)

The BinarySearchInterp function performs a binary interpolated search of the named wave for the value *val*. The returned value, pt, is a floating-point point index into the named wave such that *waveName*[pt] == *val*.

Details

BinarySearchInterp is useful for finding the point in an XY pair that corresponds to a particular X coordinate.

WaveName must contain monotonically increasing or decreasing values.

When the named wave does not actually contain the value *val*, BinarySearchInterp locates a value below *val* and a value above *val* and uses reverse linear interpolation to figure out where *val* would fall if a straight line were drawn between them. It includes that fractional amount in the resulting point index.

BinarySearchInterp returns NaN if *val* is not within the range of values in the wave.

Examples

```
Make/O data = {1, 2, 3.3, 4.9}          // Monotonic increasing
Print BinarySearchInterp(data,3)        // Prints 1.76923
Print data[1.76923]                     // Prints 3
Make/O data = {9, 4, 3, 1}              // Monotonic decreasing
Print BinarySearchInterp(data,2.5)      // Prints 2.25
Print data[2.25]                        // Prints 2.5
```

See Also

The **BinarySearch** and **FindLevel** operations. See **Indexing and Subranges** on page II-95.

binomial

binomial(*n, k*)

The binomial function returns the ratio:

$$\frac{n!}{k!(n-k)!}.$$

It is assumed that n and k are integers and $0 \le k \le n$ and ! denotes the factorial function.

Note that although the binomial function is an integer-valued function, a double-precision number has 53 bits for the mantissa. This means that numbers over 2^{52} (about 4.5×10^{15}) will be accurate to about one part in 2×10^{16}.

binomialln

binomialln(*a, b*)

The binomialln function returns the natural log of the binomial coefficient for *a* and *b*.

$$\text{binomialln}(a,b) = \ln(a!) - \ln(b!) - \ln((a-b)!)$$

See Also

Chapter III-12, **Statistics** for an overview of the various functions and operations; **binomial**, **StatsBinomialPDF**, **StatsBinomialCDF**, and **StatsInvBinomialCDF**.

binomialNoise

binomialNoise(*n, p*)

The binomialNoise function returns a pseudo-random value from the binomial distribution

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}, \qquad \begin{matrix} 0 \le p \le 1 \\ x = 1,2,...n \end{matrix}$$

whose mean is *np* and variance is *np*(1-*p*).

When *n* is large such that p^n is zero to machine accuracy the function returns NaN. When *n* is large such that *np*(1-*p*)>5 and 0.1<*p*<0.9 you can replace the binomial variate with a normal variate with mean *np* and standard deviation sqrt(*n***p**(1-*p*)).

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat the same sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-332.

Chapter III-12, **Statistics** for an overview of the various functions and operations.

boundingBall

boundingBall [/F/Z] *tripletWave*

The boundingBall operation calculates a bounding sphere for a set of X, Y, Z triplets. This operation accepts waves that have three or more columns; data in any additional columns are ignored.

Parameters

tripletWave is a three-column two-dimensional wave with X coordinates in column 0, Y in column 1, and Z in column 2.

Flags

/F Uses the algorithm from "An Efficient Bounding Sphere" by Jack Ritter in *Graphics Gems*. This algorithm is less accurate but it produces a ball that is sufficiently large to contain all the points.

/Z No error reporting.

Details

The center and radius of the bounding sphere are stored in the variables: V_CenterX, V_CenterY, V_CenterZ, and V_Radius.

If you are not using the /F flag, the operation also accepts a 2 column wave consisting of X, Y pairs for calculating the center and radius of a bounding circle in the plane.

References

Glassner, Andrew S., (Ed.), *Graphics Gems*, 833 pp., Academic Press, San Diego, 1990.

break

break

The break flow control keyword immediately terminates execution of a loop, switch, or strswitch. Execution then continues with code following the loop, switch, or strswitch.

See Also

Break Statement on page IV-37, **Switch Statements** on page IV-34, and **Loops** on page IV-36 for usage details.

BrowseURL

BrowseURL [**/Z**] ***urlStr***

The BrowseURL operation opens the Web browser or FTP browser on your computer and asks it to display a particular Web page or to connect to an FTP server.

BrowseURL sets a variable named V_flag to zero if the operation succeeds and to nonzero if it fails. This, in conjunction with the /Z flag, can be used to allow procedures to continue to execute if an error occurs.

Parameters

urlStr specifies a Web page or FTP server directory to be browsed. It is constructed of a naming scheme (e.g., "http://" or "ftp://"), a computer name (e.g., "www.wavemetrics.com" or "ftp.wavemetrics.com" or "38.170.234.2"), and a path (e.g., "/Test/TestFile1.txt"). See **Examples** for sample usage.

Flags

/Z Errors are not fatal. Will not abort procedure execution if the URL is bad or if the server is down. Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded or nonzero if it failed.

Syntactic errors, such as omitting the URL altogether or omitting quotes, are still fatal.

Examples

```
// Browse a Web page.
    String url = "http://www.wavemetrics.com/News/index.html"
    BrowseURL url

// Browse an FTP server.
    String url = "ftp://ftp.wavemetrics.com/pub/test"
    BrowseURL url
```

BuildMenu

BuildMenu ***menuNameStr***

The BuildMenu operation rebuilds the user-defined menu items in the specified menu the next time the user clicks in the menu bar.

Parameters

menuNameStr is a string expression containing a menu name or "All".

Details

Call BuildMenu when you've defined a custom menu using string variables for the menu items. After you change the string variables, call BuildMenu to update the menu.

In Igor 6.22 or later, BuildMenu "All" rebuilds all the menu items and titles and updates the menu bar. In earlier versions of Igor it just rebuilds all user-defined menu items.

Under the current implementation, if menuNameStr is not "All", Igor will rebuild *all* user-defined menu items if BuildMenu is called for *any* user-defined menu.

See Also

Dynamic Menu Items on page IV-109.

Button

Button [**/Z**] ***ctrlName*** [***keyword = value*** [**,** ***keyword = value*** …]]

The Button operation creates or modifies the named button control.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

name is the name of the Button control to be created or changed.

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind=default:Appearance determined by **DefaultGUIControls**.

kind=native:Creates standard-looking controls for the current computer platform.

kind=os9:Igor Pro 5 appearance (quasi-Macintosh OS 9 controls that look the same on Macintosh and Windows).

platform=Mac:Changes the appearance of controls only on Macintosh; affects the experiment whenever it is used on Macintosh.

platform=Win:Changes the appearance of controls only on Windows; affects the experiment whenever it is used on Windows.

platform=All:Changes the appearance on both Macintosh and Windows computers.

disable=*d*

Sets the state of the control. *d* is a bit field: bit 0 (the least significant bit) is set when the control is hidden. Bit 1 is set when the control is disabled:

d=0: Normal (visible), enabled.

d=1: Hidden.

d=2: Visible and disabled. Drawn in grayed state, also disables action procedure.

d=3: Hidden and disabled.

See the **ModifyControl** example for setting the bits individually.

fColor=(*r*,*g*,*b*)

Sets color of the button. *r*, *g*, and *b* are integers from 0 to 65535. To set the color of the title text, use escape sequences as described below for title. fColor defaults to black (0,0,0). To set the color of the title text, see valueColor.

font="*fontName*"

Sets button font, e.g., `font="Helvetica"`.

fsize=*s*

Sets font size.

fstyle=*fs*

Specifies the font style. *fs* is a binary coded number with each bit controlling one aspect of the font style:

bit 0: Sold.

bit 1: Italic.

bit 2: Underline.

bit 3: Outline (*Macintosh only*).

bit 4: Shadow (*Macintosh only*).

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

help={*helpStr*}

Specifies the help for the control. Help text is limited to a total of 255 characters. On Macintosh, help appears if you turn Igor Tips on. On Windows, help for the first 127 characters or up to the first line break appears in the status line. If you press F1 while the cursor is over the control, you will see the entire help text. You can insert a line break by putting "`\r`" in a quoted string.

noproc

No procedure is executed when clicking the button.

picture= *pict*

Draws the button using the named picture. The picture is taken to be three side-by-side frames that show the control appearance in the normal state, when the mouse is down, and in the disabled state. The picture may be either a global (imported) picture or a Proc Picture (see **Proc Pictures** on page IV-43). The size keyword is ignored when a picture is used.

pos={*left*,*top*}

Sets the position of the button in pixels.

pos+={*dx*,*dy*}

Offsets the position of the button in pixels.

proc=*procName*

Names the procedure to execute when clicking the button.

rename=*newName*

Gives the button a new name.

size={*width*,*height*}

Sets *width* and *height* of button in pixels.

title=*titleStr*

Sets title of button (text that appears in the button) to the specified string expression. If not given then title will be "New". If you use `""` the button will contain no text.

titleStr can contain formatting escape codes in order to create fancy, styled results. The escape codes are the same as used by the **TextBox** operation. The easiest way to generate fancy text is to create a dummy TextBox, set up the text as desired, click the To Cmd Line button, and then edit the TextBox command for use with the control.

userdata(*UDName*)=*UDStr*

Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*
: Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

valueColor=(*r*,*g*,*b*)
: Sets initial color of the button's text (title). *r*, *g*, and *b* range from 0 to 65535. valueColor defaults to black (0,0,0). To further change the color of the title text, use escape sequences as described for title=*titleStr*.

win=*winName*
: Specifies which window or subwindow contains the named button control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

The target window must be a graph or panel.

The action procedure, which may be a function *or* a macro, has the format:

```
Function procName(ctrlName) : ButtonControl
   String ctrlName
   …
End
```

The "`: ButtonControl`" designation tells Igor to include this procedure in the Procedure pop-up menu in the Button Control dialog.

The action procedure for a Button control can also use a predefined structure `WMButtonAction` as a parameter to the function. The control will use this more efficient method when the function properly matches the structure prototype for a Button control, otherwise it will use the old-style method.

A Button action procedure using a structure has the format:

```
Function newActionProcName(B_Struct) : ButtonControl
   STRUCT WMButtonAction &B_Struct
   …
End
```

For a Button control, the `WMButtonAction` structure has members as follows:

`WMButtonAction` Structure Members

Member	Description
`char ctrlName[MAX_OBJ_NAME+1]`	Control name.
`char win[MAX_WIN_PATH+1]`	Host (sub)window.
`STRUCT Rect winRect`	Local coordinates of host window.
`STRUCT Rect ctrlRect`	Enclosing rectangle of the control.
`STRUCT Point mouseLoc`	Mouse location.
`Int32 eventCode`	Event that executed the procedure.

`eventCode`	Event
-1	Control being killed
1	Mouse down
2	Mouse up
3	Mouse up outside control
4	Mouse moved
5	Mouse enter
6	Mouse leave

Events 2 and 3 only happen after event 1. Events 4, 5, and 6 happen only when mouse is over the control but happen regardless of the mouse button state.

WMButtonAction Structure Members

Member	Description
`Int32 eventMod`	Bitfield of modifiers. See **Control Structure eventMod Field** on page III-385.
`String userData`	Primary (unnamed) user data. If this changes, it is written back automatically.
`Int32 blockReentry`	Prevents reentry of control action procedure. See **Control Structure blockReentry Field** on page III-386.

Action functions should respond only to documented `eventCode` values. Other event codes may be added along with more fields. Although the return value is not currently used, action functions should always return zero.

The constants used to specify the size of structure `char` arrays are internal to Igor Pro and may change.

See Also

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

ButtonControl

ButtonControl

ButtonControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined button control. See **Procedure Subtypes** on page IV-179 for details. See **Button** for details on creating a button control.

cabs

cabs(*z*)

The cabs function returns the real-valued absolute value of complex number *z*.

See Also

The **magsqr** function.

CaptureHistory

CaptureHistory(*refnum, stopCapturing*)

The CaptureHistory function returns a string containing text from the History window since a matching call to the **CaptureHistoryStart** function.

Parameters

refnum is a number returned from a call to CaptureHistoryStart. It identifies the starting point in the history for the returned string.

Set *stopCapturing* to nonzero to indicate that no more history should be captured for the given refnum. Subsequent calls to CaptureHistory with the same *refnum* will result in an error.

Set *stopCapturing* to zero to retrieve history text captured so far. Further calls to CaptureHistory with the same reference number will return this text, plus any additional history text added subsequently.

Details

You can have multiple captures active at one time. Each call to CaptureHistoryStart will return a unique reference number identifying a start point in the history. The capture corresponding to each reference number can be terminated at any time, regardless of the order of the CaptureHistoryStart calls.

CaptureHistoryStart

CaptureHistoryStart()

The CaptureHistoryStart function returns a reference number to identify a starting point in the History window text. Subsequently, the CaptureHistory function can be used to retrieve captured history text. See **CaptureHistory** for details.

catch

catch

The catch flow control keyword defines the beginning of code in a try-catch-entry flow control construct for handling any abort conditions.

See Also

The **try-catch-endtry** flow control statement for details.

cd

cd *dataFolderSpec*

The cd operation sets the current data folder to the specified data folder. It is identical to the longer-named SetDataFolder operation.

cd is named after the UNIX "change directory" command.

See Also

SetDataFolder, **pwd**, **Dir**, **Data Folders** on page II-121

ceil

ceil(*num*)

The ceil function returns the closest integer greater than or equal to *num*.

See Also

The **round**, **floor**, and **trunc** functions.

cequal

cequal(*z1, z2*)

The cequal function determines the equality of two complex numbers *z1* and *z2*. It returns 1 if they are equal, or 0 if not.

This is in contrast to the == operator, which compares only the real components of *z1* and *z2*, ignoring the imaginary components.

Examples

```
Function TestComplexEqualities()
   Variable/C z1= cmplx(1,2), z2= cmplx(1,-2)
   // This test compares only the real parts of z1 and z2:
   if( z1 == z2 )
      Print "== match"
   else
      Print "no == match"
   endif
   // This test compares both real and imaginary parts of z1 and z2:
   if( cequal(z1,z2) )
      Print "cequal match"
   else
      Print "no cequal match"
   endif
End
```

```
•TestComplexEqualities()
  == match
  no cequal match
```

See Also

The **imag**, **real**, and **cmplx** functions.

char2num

char2num(*str*)

The char2num function returns a number which is the numeric representation of the first byte in the string expression *str*.

The char2num function treats *str* as a string of signed bytes. Consequently it returns a negative number for bytes that have bit 7 set. You can obtain the byte value as a positive number by ANDing with 0xFF.

Examples

```
Print char2num("A")           // Prints 65
Print char2num("ABC")         // Prints 65
Print char2num("•")           // Prints character code as negative number
Print char2num("•") & 0xFF    // Prints character code as positive number
Printf "%02X\r", char2num("•") & 0xFF   // Prints as hexadecimal
```

See Also

The **num2char**, **str2num** and **num2str** functions.

Chart

Chart [**/Z**] ***ctrlName*** [***keyword = value*** [**,** ***keyword = value*** **…**]]

The Chart operation creates or modifies a chart control. Charts are generally used in conjunction with data acquisition. Charts do not have to be connected to a FIFO, but they are not useful until they are.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the Chart control to be created or changed.

The following keyword=value parameters are supported:

chans={*ch#*, *ch#*,…} List of FIFO channel numbers that Chart is to monitor.

color(*ch#*)=(*r*,*g*,*b*) Sets the color of the specified trace. *r*, *g*, and *b* specify the amount of red, green, and blue in the color as an integer from 0 to 65535.

ctab=*colortableName* When a channel is connected to an image strip FIFO channel, the data is displayed as an image using this built-in color table. Valid names are the same as used in images. Invalid name will result in the default Grays color table being used.

disable=*d* Sets user editability of the control.

- *d*=0: Normal.
- *d*=1: Hide.
- *d*=2: Disable user input. Charts do not change appearance because they are read-only. When disabled, the hand cursor is not shown.

fbkRGB=(*r*,*g*,*b*) Sets frame background color. *r*, *g* and *b* are integers from 0 to 65535.

fgRGB=(*r*,*g*,*b*) Sets foreground color (text, etc.). *r*, *g* and *b* are integers from 0 to 65535

fifo=*FIFOName* Sets which named FIFO the chart will monitor. See the **NewFIFO** operation.

font="*fontName*" Sets the font used in the chart, e.g., `font="Helvetica"`.

fsize=*s* Sets font size for chart.

fstyle=*fs* Specifies the font style. *fs* is a binary coded number with each bit controlling one aspect of the font style as follows:

- bit 0: Bold.
- bit 1: Italic.
- bit 2: Underline.
- bit 3: Outline (*Macintosh only*).
- bit 4: Shadow (*Macintosh only*).

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

gain(*ch#*)=*g* Sets the display gain *g* of the specified channel relative to nominal. Values greater than unity expand the display.

gridRGB=(*r*,*g*,*b*) Sets grid color. *r*, *g*, and *b* are integers from 0 to 65535.

help={*helpStr*} Specifies help for the control. Help text is limited to a total of 255 characters. On Macintosh, help appears if you turn Igor Tips on. On Windows, help for the first 127 characters or up to the first line break appears in the status line. If you press F1 while the cursor is over the control, you will see the entire help text. You can insert a line break by putting "\r" in a quoted string.

jumpTo=*p* Jumps to point number *p*. This works in review mode only.

lineMode(*ch#*)=*lm* Sets the display line mode for the given channel.

lm=0: Dots mode. Draws values as dots. However, if the number of dots in a strip exceeds maxDots then Igor draws a vertical line from the min to the max of the values packed into the strip.

lm=1: Lines mode. Draws a vertical line encompassing the min and the max of the points in a given strip along with the last point of the preceding strip. Since which strip is the preceding strip depends on the direction of motion then the appearance may slightly shift depending on which direction the chart is moving.

lm=2: Sticks to zero mode. Draws a vertical line encompassing the min and the max of the points in a given strip along with a value of zero.

mass=*m* Sets the "feel" of the chart paper when you move it with the mouse. The larger the mass *m*, the slower the chart responds. Odd values cause the movement of the paper to stop the instant the mouse is clicked while even values continue with the illusion of mass.

maxDots=*md* Controls whether points in a given vertical strip of the chart are displayed as dots or as a solid line. See lineMode above. Default is 20.

offset(*ch#*)=*o* Sets the display offset of the specified channel. The offset value *o* is subtracted from the data before the gain is applied.

oMode=*om* Chart operation mode.

om=0: Live mode.

om=1: Review mode.

pbkRGB=(*r*,*g*,*b*) Sets plot area background color. *r*, *g*, and *b* are integers from 0 to 65535.

ppStrip=*pps* Number of data points packed into each vertical strip of the chart.

rSize(*ch#*)=*rs* Sets the relative vertical size allocated to the given channel. Nominal is unity. If the value of *rs* is zero then this channel shares space with the previous channel.

sMode=*sm* Status line mode.

sm=0: Turns off fancy status line and positioning bar.

sm=1: Normal mode.

sm=2: Uses alternate style for bar.

sRate=*sr* Sets the scroll rate (vertical strips/second). If the chart control is in review mode negative numbers scroll in reverse.

title=*titleStr* Specifies the chart title. Use "" for no title.

uMode=*um* Selects the update mode.

um=1: Fast update with no bells and whistles.

um=2: Status line and positioning bar.

um=3: Status line, positioning bar, and animated pens.

win=*winName* Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

The target window must be a graph or panel.

The action of some of the Chart keywords depends on whether or not data acquisition is taking place. If the chart is in review mode then all keywords cause the chart to be redrawn. If data acquisition is taking place and the chart is in live mode then some keywords affect new data but do not attempt to update the part of the "paper" that has already been drawn. The following keywords affect only new data during live mode:

```
ppStrip  maxDots  gain  offset  color  lineMode
```

See Also

Charts on page III-360 and **FIFOs and Charts** on page IV-276.

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

chebyshev

chebyshev(*n*, *x*)

The chebyshev function returns the Chebyshev polynomial of the first kind and of degree *n*.

The Chebyshev polynomials satisfy the recurrence relation:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \text{ with: } T_0(x) = 1 \quad .$$
$$T_1(x) = x$$
$$T_2(x) = 2x^2 - 1$$

The orthogonality of the polynomial is expressed by the integral:

$$\int_{-1}^{1} \frac{T_n(x)T_m(x)}{\sqrt{1-x^2}}dx = \begin{cases} 0 & m \neq n \\ \pi/2 & m = n \neq 0 \\ \pi & m = n = 0 \end{cases} .$$

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

See Also

chebyshevU.

chebyshevU

chebyshevU(*n*, *x*)

The chebyshevU function returns the Chebyshev polynomial of the second kind, degree *n* and argument *x*.

The Chebyshev polynomial of the second kind satisfies the recurrence relation

```
U(n+1,x)=2xU(n,x)-U(n-1,x)
```

which is also the recurrence relation of the Chebyshev polynomials of the first kind.

The first 10 polynomials of the second kind are:

```
U(0,x)=1
U(1,x)=2x
U(2,x)=4x^2-1
U(3,x)=8x^3-4x
U(4,x)=16x^4-12x^2+1
U(5,x)=32x^5-32x^3+6x
U(6,x)=64x^6-80x^4+24x-1
U(7,x)=128x^7-192x^5+80x^3-8x
U(8,x)=256x^8-448x^6+240x^4-40x^2+1
U(9,x)512x^9-1024x^^7+672x^5-160x^3+10x
```

See Also

The **chebyshev** function.

CheckBox

CheckBox [**/Z**] ***ctrlName*** [***keyword = value*** [**,** ***keyword = value*** **…**]]

The CheckBox operation creates or modifies a checkbox, radio button or disclosure triangle in the target or named window, which must be a graph or control panel.

ctrlName is the name of the checkbox.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the CheckBox control to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}
: Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

  kind can be one of `default`, `native`, or `os9`.

  platform can be one of `Mac`, `Win`, or *All*.

  See **Button** and **DefaultGUIControls** for more appearance details.

disable=*d*
: Sets user editability of the control.

  d=0: Normal.

  d=1: Hide.

  d=2: Disable user input.

fsize=*s*
: Sets font size for checkbox.

fColor=(*r*,*g*,*b*)
: Sets the initial color of the title. *r*, *g*, and *b* range from 0 to 65535. fColor defaults to black (0,0,0). To further change the color of the title text, use escape sequences as described for title=*titleStr*.

help={*helpStr*}
: Sets the help for the control. The help text is limited to a total of 255 characters. On Macintosh, the help appears if you turn Igor Tips on. On Windows, the help for the first 127 characters or up to the first line break appears in the status line. If you press F1 while the cursor is over the control, you will see the entire help text. You can insert a line break by putting "`\r`" in a quoted string.

mode=*m*
: Specifies checkbox appearance.

  m=0: Default checkbox appearance.

  m=1: Display as a radio button control.

  m=2: Display as a disclosure triangle (*Macintosh*) or treeview expansion node (*Windows*).

noproc
: Specifies that no procedure is to execute when clicking the checkbox.

picture= *pict*
: Draws the checkbox using the named picture. The picture is taken to be six side-by-side frames which show the control appearance in the normal state, when the mouse is down, and in the disabled state. The first three frames are used when the checked state is false and the next three show the true state. The picture may be either a global (imported) picture or a Proc Picture (see **Proc Pictures** on page IV-43).

pos={*left*,*top*}
: Sets the position of the checkbox in pixels.

pos+={*dx*,*dy*}
: Offsets the position of the checkbox in pixels.

proc=*procName*
: Specifies the procedure to execute when the checkbox is clicked.

rename=*newName*
: Renames the checkbox to *newName*.

side=s
: s =0: Checkbox is on the left, title is on the right (default).

  s =1: Checkbox is on the right, title is on the left.

size={*width*,*height*}
: Sets checkbox size in pixels.

title=*titleStr*
: Sets title of checkbox to the specified string expression. The title is the text that appears in the checkbox. If not given or if `""` then the title will be "New".

  titleStr can contain formatting escape codes in order to create fancy, styled results. The escape codes are the same as used by the **TextBox** operation. The easiest way to

generate fancy text is to make selections from the Insert popup in the Checkbox Control dialog.

userdata(*UDName*)=*UDStr*
: Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*
: Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

value=*v*
: Specifies whether the checkbox is selected (*v*=1) or not (*v*=0).

variable= *varName*
: Specifies a global numeric variable to be set to the current state of a checkbox whenever it is clicked or when it is set by the value parameter. The variable is two-way: setting the variable also changes the state of the checkbox.

win=*winName*
: Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

  When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

The target window must be a graph or panel.

The action procedure, which may be a function *or* a macro, has the format:

```
Function procName(ctrlName,checked) : CheckBoxControl
   String ctrlName
   Variable checked       // 1 if selected, 0 if not
   …
End
```

The "`: CheckboxControl`" designation tells Igor to include this procedure in the Procedure pop-up menu in the Checkbox Control dialog.

The action procedure for a CheckBox control can also use a predefined structure `WMCheckboxAction` as a parameter to the function. The control will use this more efficient method when the function properly matches the structure prototype for a CheckBox control, otherwise it will use the old-style method.

A CheckBox action procedure using a structure has the format:

```
Function newActionProcName(CB_Struct) : CheckBoxControl
   STRUCT WMCheckboxAction &CB_Struct
   …
End
```

For a CheckBox control, the `WMCheckboxAction` structure has members as described in the following table:

`WMCheckboxAction` Structure Members

Member	Description
`char ctrlName[MAX_OBJ_NAME+1]`	Control name.
`char win[MAX_WIN_PATH+1]`	Host (sub)window.
`STRUCT Rect winRect`	Local coordinates of host window.
`STRUCT Rect ctrlRect`	Enclosing rectangle of the control.
`STRUCT Point mouseLoc`	Mouse location.
`Int32 eventCode`	Event that executed the procedure.

`eventCode`	Event
-1	Control being killed
2	Mouse up

WMCheckboxAction Structure Members

Member	Description
Int32 eventMod	Bitfield of modifiers. See **Control Structure eventMod Field** on page III-385.
String userData	Primary (unnamed) user data. If this changes, it is written back automatically.
Int32 blockReentry	Prevents reentry of control action procedure. See **Control Structure blockReentry Field** on page III-386.
Int32 checked	Checkbox state.

Action functions should respond only to documented eventCode values. Other event codes may be added along with more fields. Although the return value is not currently used, action functions should always return zero.

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

When using radio button controls, it is the responsibility of the Igor programmer to turn off other radio buttons when one of a group of radio buttons is pressed.

Examples

The following code is an example of how to program such a group. (Copy the following code into the procedure window of a new experiment and then bring up the panel.)

```
Window Panel0() : Panel
   PauseUpdate; Silent 1          // building window …
   NewPanel /W=(150,50,353,212)
   Variable/G gRadioVal= 1
   CheckBox check0,pos={52,25},size={78,15},title="Radio 1"
   CheckBox check0,value=1,mode=1,proc=MyCheckProc
   CheckBox check1,pos={52,45},size={78,15},title="Radio 2"
   CheckBox check1,value=0,mode=1,proc=MyCheckProc
   CheckBox check2,pos={52,65},size={78,15},title="Radio 3"
   CheckBox check2,value= 0,mode=1,proc=MyCheckProc
EndMacro

Function MyCheckProc(name,value)
   String name
   Variable value

   NVAR gRadioVal= root:gRadioVal

   strswitch (name)
      case "check0":
         gRadioVal= 1
         break
      case "check1":
         gRadioVal= 2
         break
      case "check2":
         gRadioVal= 3
         break
   endswitch
   CheckBox check0,value= gRadioVal==1
   CheckBox check1,value= gRadioVal==2
   CheckBox check2,value= gRadioVal==3
End
```

See Also

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

CheckBoxControl

CheckBoxControl

CheckBoxControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined checkbox control. See **Procedure Subtypes** on page IV-179 for details. See **CheckBox** for details on creating a checkbox control.

CheckDisplayed

CheckDisplayed [/A/W] *waveName* [, *waveName*]…

The CheckDisplayed operation determines if named waves are displayed in a host window or subwindow.

Flags

/A	Checks all graph and table windows
/W=*winName*	Checks only the named graph or table window When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

If neither /A nor /W are used, CheckDisplayed checks only the top graph or table.

CheckDisplayed sets a bit in the variable V_flag for each wave that is displayed.

Examples

```
CheckDisplayed/W=Graph0 aWave,bWave,cWave
```

Checks Graph0 to see if aWave, bWave, and cWave are displayed in it. If aWave is displayed, CheckDisplayed sets bit 0 of V_flag (V_flag=1). If bWave is displayed, sets bit 1 (V_flag=2). If cWave is displayed, sets bit 2 (V_flag=4). If all three waves are displayed, V_flag=7.

See Also

Setting Bit Parameters on page IV-12 for information about bit settings.

CheckName

CheckName(*nameStr*, *objectType* [, *windowNameStr*])

The CheckName function returns a number which indicates if the specified name is legal and unique among objects in the namespace of the specified object type.

Waves, global numeric variables, and global string variables are all in the same namespace and need to be unique only within the data folder containing them. However, they also need to be distinct from names of Igor operations and functions and from names of user-defined procedures.

Data folders are in their own namespace and need to be unique only among other data folders at the same level of the data folder hierarchy.

windowNameStr is optional. If missing, it is taken to be the top graph, panel, layout, or notebook according to the value of *objectType*.

Details

A result of zero indicates that the name is legal and unique within its namespace. Any nonzero result indicates that the name is illegal or not unique. You can use the **CleanupName** and **UniqueName** functions to guarantee legality and uniqueness.

nameStr should contain an unquoted name (i.e., no single quotes for liberal names), such as you might receive from the user through a dialog or control panel.

objectType is one of the following:

1: Wave.
2: Reserved.
3: Global numeric variable.
4: Global string variable.
5: XOP target window (e.g., surface plot).
6: Graph window.
7: Table window.
8: Layout window.
9: Control panel window.
10: Notebook window.
11: Data folder.
12: Symbolic path.
13: Picture.
14: Annotation in the named or topmost graph or layout.
15: Control in the named topmost graph or panel.
16: Notebook action character in the named or topmost notebook.

The *windowNameStr* argument is used only with *objectTypes* 14, 15, and 16. The *nameStr* is checked for uniqueness only within the named window (other windows might have objects with the given name). If a named window is given but does not exist, any valid *nameStr* is permitted

Examples

```
Variable waveNameIsOK = CheckName(proposedWaveName, 1) == 0
Variable annotationNameIsOK = CheckName("text0", 14, "Graph0")
```

See Also

CleanupName and **UniqueName** functions.

ChildWindowList

ChildWindowList(*hostNameStr*)

The ChildWindowList function returns a string containing a semicolon-separated list of immediate subwindow window names of the specified host window or subwindow.

Parameters

hostNameStr is a string or string expression containing the name of an existing host window or subwindow.

When identifying a subwindow with *hostNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

Error if the host does not exist or if it is not an allowed host type.

See Also

WinList and **WinType** functions.

ChooseColor

ChooseColor [**/C=(*r*,*g*,*b*)**]

The ChooseColor operation displays a dialog for choosing a color.

The color initially shown is black unless you specify a different color with /C.

Flags

/C=(*r*,*g*,*b*) *r*, *g*, and *b* specify the amount of red, green, and blue in the color initially displayed in the dialog as an integer from 0 to 65535.

Details

ChooseColor sets the variable V_flag to 1 if the user clicks OK in the dialog or to 0 otherwise.

If V_flag is 1 then V_Red, V_Green, and V_Blue are set to the selected color as integers from 0 to 65535.

See Also

ImageTransform rgb2hsl and **hsl2rgb**.

CleanupName

CleanupName(*nameStr, beLiberal*)

The CleanupName function returns the input name string, possibly altered to make it a legal object name.

Details

nameStr should contain an unquoted (i.e., no single quotes for liberal names) name, such as you might receive from the user through a dialog or control panel.

beLiberal is 0 to use strict name rules or 1 to use liberal name rules. Strict rules allow only letters, digits and the underscore character. Liberal rules allow other characters such as spaces and dots. Liberal rules were introduced with Igor Pro 3.0 and are allowed for names of waves and data folders only.

Note that a cleaned up name is not necessarily unique. Call **CheckName** to check for uniqueness or **UniqueName** to ensure uniqueness.

Examples

```
String cleanStrVarName = CleanupName(proposedStrVarName, 0)
```

See Also

CheckName and **UniqueName** functions.

Close

Close [/A] *fileRefNum*

The Close operation closes a file previously opened by the **Open** operation or closes all such files if /A is used.

Parameters

fileRefNum is the file reference number of the file to close. This number comes from the Open operation. If /A is used, *fileRefNum* should be omitted.

Flags

/A Closes all files. Mainly useful for cleaning up after an error during procedure execution occurs so that the normal Close operation is never executed.

CloseMovie

CloseMovie

The CloseMovie operation closes the currently open movie. You must close a movie before you can play it.

See Also

The **NewMovie** operation.

CloseProc

CloseProc /NAME=*procNameStr* [*flags*]

CloseProc /FILE=*fileNameStr* [*flags*]

The CloseProc operation closes a procedure window. You cannot call CloseProc on the main Procedure window.

CloseProc provides a way to programmatically create and alter procedure files. You might do this in order to make a user-defined menu-bar menu with contents that change.

Note: CloseProc alters procedure windows so it cannot be called while functions or macros are running. If you want to call it from a function or macro, use **Execute/P**.

Warning: If you close a procedure window that has no source file or without specifying a destination file, the window contents will be permanently lost.

Flags

/COMP[=*compile*] Specifies whether procedures should be compiled after closing the procedure window.

compile=1: Compiles procedures (same as /COMP only).

compile=0: Leaves procedures in an uncompiled state.

/D[=*delete*] Specifies whether the procedure file should be deleted after closing the procedure window.

delete=1: Deletes procedure file (same as /D only). **Warning**: You cannot recover any file deleted this way.

delete=0: Leaves any associated file unaffected.

/FILE=*fileNameStr* Identifies the procedure window to close using the file name and path to the file given by *fileNameStr*. The string can be just the file name if /P is used to specify a symbolic path name of the enclosing folder. It can be a partial path if /P points to a folder enclosing the start of the partial path. It can also be a full path the file.

/NAME=*procNameStr*

Identifies the procedure window to close with the string expression *procNameStr*. This is the same text that appears in the window title. If the procedure window is associated with a file, it will be the file name and extension.

/P=*pathname* Specifies the folder to look in for the file specified by /FILE. *pathName* is the name of an existing symbolic path.

/SAVE[=*savePathStr*]

Saves the procedure before closing the window. If the flag is used with no argument, it saves any changes to the procedure window to its source file before closing it. If *savePathStr* is present, it must be a full path naming a file in which to save the procedure window contents. The /P flag is not used with *savePathStr* so it must be a full path.

Details

Specify which window to close using either the /NAME or /FILE flag. You must use one or the other. Usually you would use /NAME, as it is usually more convenient. If by some chance two procedures have the same name, /FILE can be used to distinguish between them.

You cannot call CloseProc on a nonmain procedure window that someone has had the bad taste to call "Procedure".

See Also

Chapter III-13, **Procedure Windows**.

The **Execute/P** operation.

cmplx

cmplx(*realPart*, *imagPart*)

The cmplx function returns a complex number whose real component is *realPart* and whose imaginary component is *imagPart*.

Use this to assign a value to a complex variable or complex wave.

Examples

Assume wave1 is complex. Then:

```
wave1(0) = cmplx(1,2)
```

sets the Y value of wave1 at x=0 such that its real component is 1 and its imaginary component is 2.

Assuming wave2 and wave3 are real, then:

```
wave1 = cmplx(wave2,wave3)
```

sets the real component of wave1 equal to the contents of wave2 and the imaginary component of wave1 equal to the contents of wave3.

You may get unexpected results if the number of points in wave2 or wave3 differs from the number of points in wave1. If wave2 or wave3 are shorter than wave1, the last element of the short wave is copied repeatedly to fill wave1.

See Also

conj, **imag**, **magsqr**, **p2rect**, **r2polar**, and **real** functions.

cmpstr

cmpstr(*str1*, *str2* [, *flags*])

The cmpstr function returns -1, 0 or 1 depending on how string *str1* compares alphabetically to string *str2*.

Details

cmpstr returns the following values:

-1:	*str1* is alphabetically before *str2*.
0:	*str1* and *str2* are equal.
1:	*str1* is alphabetically after *str2*.

If *flags* is not present, or if *flags* is zero, case (upper or lower) is not significant. Set *flags* to 1 for a case-sensitive comparison.

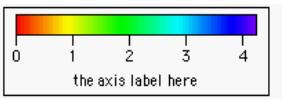
See Also

The **ReplaceString** function.

ColorScale

ColorScale [*flags*] [, *keyword = value*, …] [*axisLabelStr*]

The ColorScale operation puts a color scale (or "color legend") annotation on the top graph or page layout.



The ColorScale operation can be executed with no flags and no parameters.

When a graph is the top window the color scale represents the colors and values associated with the first image plot that was added to the graph.

If there is no image plot in the graph, the color scale represents the first contour plot or first f(z) trace added to the graph, one of which must exist for the command to execute without error when executed without parameters.

Executing ColorScale (with no parameters) when a layout is the top window displays a color bar as if the `ctab={0,100,Rainbow}` parameters had been specified.

Flags

Use the /W=*winName* flag to specify a specific graph or layout window. When used on the command line or in a Macro or Proc, /W must precede all other flags.

To change a color scale, use the /C/N=*name* flags. Annotations are automatically named "text0", "text1", etc. if no name is specified when it is created, and you must use that name to modify an existing annotation or else a new one will be created.

For explanations of all flags see the **TextBox** operation.

Parameters

The following keyword-value pairs are the important parameters to the ColorScale operation, because they specify what object the color scale is representing.

Note: For use with page layouts, the keyword ={*graphName*,…} form is required.

For graphs it is simpler to use the image=*imageInstanceName* form (omitting graphName), though you can use `$""` to mean the top graph, or specify the name of another graph with the long form. See the **Examples**.

axisLabelStr Contains the text printed beside the color scale's main axis. This text is interpreted the same way as for the **Label** operation. It may contain escape codes to alter the text color or display subscripts, for example. You can use **AppendText** or **ReplaceText** to modify this axis label string. The default value for axisLabelStr is `""`.

cindex=*cindexMatrixWave*

The colors shown are those in the named color index wave with axis values derived from the wave's X (row) scaling and units.

The image colors are determined by doing a lookup in the specified matrix wave. See the **ModifyImage cindex** keyword.

contour=*contourInstanceName*
contour={*graphName*,*contourInstanceName*}

The colors show the named contour plot's colors and associated contour (Z) values and contour data units. All of the image plot's characteristics are represented, including color table, cindex, and fixed colors.

ctab={*zMin*,*zMax*,*ctName*, *mode*}

The color table specified by *ctName* is drawn in the color legend. *ctName* can be any of those returned by the **CTabList** function, such as Grays or Rainbow. Also see **Color Tables** on page II-349.

The color table name can be omitted if you want to leave it unchanged. *zMin* and *zMax* set the range of Z values to map. Set parameter *mode* to 1 to reverse the color table; zero or missing value does not reverse the color table.

image=*imageInstanceName*
image={*graphName*,*imageInstanceName*}

The colors show the named image plot's colors and associated image (Z) values and image data units. All of the image plot's characteristics are represented, including color table, cindex, lookup wave, eval colors, and NaN transparency. **Note**: only false-color image plots can be used with ColorScale (see **Indexed Color Details** on page II-356).

lookup= *waveName* Specifies an optional 1D wave used to modify the mapping of scaled Z values into the color table specified with the ctab keyword. Values should range from 0.0 to 1.0. A linear ramp from 0 to 1 would have no effect while a ramp from 1 to 0 would reverse the image. Used to apply gamma correction to grayscale images or for special effects. Use `lookup=$""` to remove option.

This keyword is not needed with the image keyword, even if the image plot uses a lookup wave. The image plot's lookup wave is used instead of the ColorScale lookup wave.

trace=*traceInstanceName*
trace={*graphName*,*traceInstanceName*}

The colors show the color(s) of the named trace. This is useful when the trace has its

color set by a "Z wave" using the `ModifyGraph zColor(traceName)=...` feature. In the Modify Trace Appearance dialog this is selected in the "Set as f(z)" subdialog. The color scale's main axis shows the range of values in the Z wave, and displays any data units the wave may have.

Size Parameters

The following keyword-value parameters modify the size of the color scale annotation. These keywords are similar to those used by the **Slider** control. The size of the annotation is indirectly controlled by setting the size of the "color bar" and the various axis parameters. The annotation sizes itself to accommodate the color bar, tick labels, and axis label(s).

height=*h*
: Sets the height of the color bar in points, overriding any heightPct setting. The default height is 75% of the plot area height if the color scale is vertical, or a constant of 15 points if the color scale is horizontal. The default is restored by specifying `height=0`. Specifying a heightPct value resets height to this default.

heightPct=*hpct*
: Sets height as a percentage of the graph's plot area, overriding any height setting. The default height is 75% of the plot area height if the color scale is vertical, or a constant of 15 points if the color scale is horizontal. The default height is restored by setting heightPct=0. Specifying a height value resets heightPct to this default.

side=*s*
: Selects on which axis to draw main axis ticks.

  s=1: Right of the color bar if vert=1, or below if vert=0.

  s=2: Left of the color bar if vert=1, or above if vert=0.

vert=*v*
: Specifies color scale orientation.

  v=0: Horizontal.

  v=1: Vertical (default).

width=*w*
: Sets the width of the color bar in points, overriding any widthPct setting. The default width is a constant 15 points if the color scale is vertical, or 75% of the plot area width if the color scale is horizontal. The default is restored by specifying `width=0`. Specifying a widthPct value resets width to this default.

widthPct=*wpct*
: Sets width as a percentage of the graph's plot area, overriding any width setting. The default width is a constant 15 points if the color scale is vertical, or 75% of the plot area width if the color scale is horizontal. The default is restored by setting `widthPct=0`. Specifying a width value resets widthPct to this default.

Color Bar Parameters

The following keyword-value parameters modify the appearance of the color scale color bar.

colorBoxesFrame=*on*
: Draws frames surrounding up to 99 swatches of colors in the color bar (*on*=1).

  When specifying more than 99 colors in the color bar (such as the Rainbow color table, which has 100 colors), the boxes aren't framed. Framing color boxes is effective only for small numbers of colors. Set the width of the frame with the frame keyword.

  Use *on*=0 to turn off color box frames.

frame=*f*
: Specifies the thickness of the frame drawn around the color bar in points (*f* can range from 0 to 5 points). The default is 1 point. Fractional values are permitted.

  Turn frames off with *f*=0. Values less that 0.5 do not display on screen, but the thin frame will print.

frameRGB=(*r*, *g*, *b*) or 0
: Sets the color of the frame around the color bar. *r*, *g*, and *b* specify the amount of red, green, and blue as an integer from 0 to 65535. The frame includes the individual color bar colors when colorBoxesFrame=1.

  The frame will use the colorscale foreground color, as set by the /G flag, when frameRGB=0.

Axis Parameters

The following keyword-value parameters modify the appearance of the color scale axes. These keywords are based on the **ModifyGraph** Axis keywords because they modify the main or secondary color scale axes.

axisRange={*zMin*, *zMax*}
Sets the color bar axis range to values specified by *zMin* and *zMax*. Use * to use the default axis range for either or both values.

Omit *zMin* or *zMax* to leave that end of the range unchanged. For example, use {*zMin*, } to change *zMin* and leave *zMax* alone, or use { ,*} to set only the axis maximum value to the default value.

dateInfo={*sd*,*tm*,*dt*} Controls formatting of date/time axes.

sd=0: Show date in the date&time format.

sd=1: Suppress date.

tm=0: 12 hour (AM/PM) time.

tm=1: 24 hour (military) time.

tm=2: Elapsed time.

dt=0: Short dates (2/22/90).

dt=1: Long dates (Thursday, February 22, 1990).

dt=2: Abbreviated dates (Thurs, Feb 22, 1990).

These have no effect unless the axis is controlled by a wave with 'dat' data units.

For an f(z) color scale:

```
SetScale d, 0,0, "dat", fOfZWave
```

For a contour plot or image plot color scale:

```
SetScale d, 0,0, "dat", ZorXYZorImageWave
```

See **Date/Time Axes** on page II-276 and **Date, Time, and Date&Time Units** on page II-85 for details on how date/time axes work.

font=*fontNameStr* Name of font as string expression. If the font does not exist, the default font is used. Specifying "default" has the same effect. Unlike ModifyGraph, the *fontNameStr* is evaluated at runtime, and its absence from the system is not an error.

fsize=*s* Sets the font size in points.

s=0: Use the graph font size for tick labels and axis labels (default).

fstyle=*fs* Sets the font style. *fs* is a binary coded number with each bit controlling one aspect of the font style for the tick mark labels:

bit 0: Bold.

bit 1: Italic.

bit 2: Underline.

bit 3: Outline (*Macintosh only*).

bit 4: Shadow (*Macintosh only*).

bit 5: Condensed (*Macintosh only*).

bit 6: Extended (*Macintosh only*).

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

highTrip=*h* If the extrema of an axis are between its lowTrip and its highTrip then tick mark labels use fixed point notation. Otherwise they use exponential (scientific or engineering) notation. The default highTrip is 100,000.

lblLatPos=*p* Sets a lateral offset for the main axis label. This is an offset parallel to the axis. *p* is in points. Positive is down for vertical axes and to the right for horizontal axes. The default is 0.

lblMargin=*m* Moves the main axis label by *m* points (default is 0) from the normal position. The default value is -5, which brings the axis label closer to the axis. Use more positive values to move the axis label away from the axis.

lblRot=*r* Rotates the axis label by *r* degrees. *r* is a value from -360 to 360. Rotation is counterclockwise and starts from the label's normal orientation.

log=*l* Specifies the axis type.

l=0: Linear (default).

l=1: Log base 10.

l=2: Log base 2.

logHTrip=*h* Same as highTrip but for log axes. The default is 10,000.

logLTrip=*l* Same as lowTrip but for log axes. The default is 0.0001.

logTicks=*t* Specifies the maximum number of decades in log axis before minor ticks are suppressed.

lowTrip=*l* If the axis extrema are between its lowTrip and its highTrip, then tick mark labels use fixed point notation. Otherwise, they use exponential (scientific or engineering) notation. The default lowTrip is 0.1.

minor=*m*
- *m*=0: Disables minor ticks (default).
- *m*=1: Enables minor ticks.

notation=*n*
- *n*=0: Engineering notation (default).
- *n*=1: Scientific notation.

nticks=*n* Specifies the approximate number of ticks to be distributed along the main axis. Ticks are labelled using the same automatic algorithm used for graph axes. The default is 5.
- *n*=0: No ticks.

prescaleExp=*exp* Multiplies axis range by 10*exp* for tick labeling and *exp* is subtracted from the axis label exponent. In other words, the exponent is moved from the tick labels to the axis label. The default is 0 (no modification). See the discussion in the **ModifyGraph (axes) Details** section.

tickExp=*te*
- *te*=1: Forces tick labels to exponential notation when labels have units with a prefix.
- *te*=0: Turns off exponential notation.

tickLen=*t* Sets the length of the ticks. *t* is the major tick mark length in points. This value must be between -100 and 50.
- *t*=0 to 50: Draws tick marks between the tick labels and the colors box.
- *t*=-1: Default; auto tick length, equal to 70% of the tick label font size. Draws tick marks between the tick labels and the colors box.
- *t*=-2 to -50: Draws tick marks crossing the edge of the colors box nearest the tick labels. The actual total tick mark length is -*t*.
- *t*=-100 to -51: Draws tick marks inside the edge of the colors box nearest the tick labels. Actual tick mark length is -(*t*+50). For example, -58 makes in an inside tick mark that is 8 points long.

tickThick=*t* Sets the tick mark thickness in points (from 0 to 5 points). The default is 1 point. Fractional values are permitted.
- *t*=0: Turns tick marks off, but not the tick labels.

tickUnit=*tu* Turns on (*tu*=0) or off (*tu*=1) units labels attached to tick marks.

userTicks={*tvWave*,*tlblWave*}
Supplies user defined tick positions and labels for the main axis. *tvWave* contains the numeric tick positions while text wave *tlblWave* contains the corresponding labels.

The tick mark labels can be multiline and use styled text. For more details, see **Fancy Tick Mark Labels** on page II-312. Overrides normal ticking specified by nticks.

ZisZ=*z* *z* =1 labels the zero tick mark (if any) with the single digit "0" regardless of the number of digits used for other labels. Default is *z*=0.

Secondary Axis Parameters

axisLabel2=*axisLabelString2*
Axis label for the secondary axis. This axis label is drawn only if userTicks2 is in effect. Text after any `\r` character is ignored, as is the `\r` character. The default is `""`.

lblLatPos2=*p* Sets lateral offset for secondary axis labels. This is an offset parallel to the axis. *p* is in points. Positive is down for vertical axes and to the right for horizontal axes. The default is 0.

lblMargin2=*m* Specifies the distance in points (default 0) to move the secondary axis label from the position that would be normal for a graph. The default is value is -5, which brings the axis label closer to the axis. Use more positive values to move the axis label away from the axis.

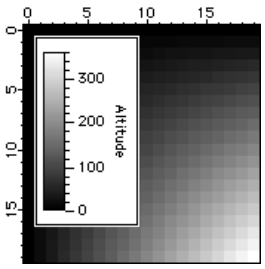
lblRot2=*r* Rotates the secondary axis label by *r* degrees counterclockwise starting from the normal label orientation. r is a value from -360 to 360.

userTicks2={*tvWave*,*tlblWave*}
Supplies user defined tick positions and labels for a second axis which is always on the

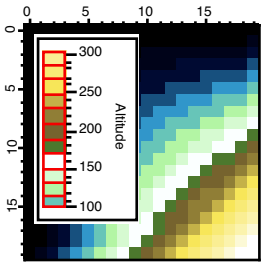
opposite side of the color bar from the main axis. The tick mark labels can be multiline and use styled text. For more details, see **Fancy Tick Mark Labels** on page II-312. This is the only way to draw a second axis.

Examples

```
Make/O/N=(20,20) img=p*q; NewImage img      // Make and display an image
ColorScale                                  // Create default color scale
// First annotation is text0
ColorScale/C/N=text0 nticks=3,minor=1,"Altitude"
```




```
ModifyImage img ctab= {*,*,Relief19,0}          // 19-color color table
ColorScale/C/N=text0 axisRange={100,300}        // Detail for 100-300 range
ColorScale/C/N=text0 colorBoxesFrame=1          // Frame the color boxes
ColorScale/C/N=text0 frameRGB=(65535,0,0)       // Red frame
```



See Also

For all other flags see the **TextBox** and **AppendText** operations.

ColorTab2Wave

ColorTab2Wave ***colorTableName***

The ColorTab2Wave operation extracts colors from the built-in color table and places them in an Nx3 matrix of red, green, and blue columns named M_colors. Values are unsigned 16-bit integers and range from 0 to 65535.

N will typically be 100 but may be as little as 9 and as large as 476. Use

```
Variable N= DimSize(M_colors,0)
```

to determine the actual number of colors.

The wave M_colors is created in the current data folder. Red is in column 0, green is in column 1, and blue in column 2.

Parameters

colorTableName can be any of those returned by **CTabList**, such as Grays or Rainbow.

colorTableName can also be `Igor` or `IgorRecent`, to return either the 128 standard or 0-32 user-selected colors from Igor's color menu.

Details

See **Color Tables** on page II-349.

Concatenate

Concatenate [***type flags***][***flags***] ***waveListStr, destWave***

Concatenate [***type flags***][***flags***] **{*wave1, wave2, wave3,…*}, *destWave***

The Concatenate operation combines data from the source waves into *destWave*, which is created if it does not already exist. If *destWave* does exists and overwrite is not specified, the source waves' data is concatenated with the existing data in the destination wave.

By default the concatenation increases the dimensionality of the destination wave if possible. For example, if you concatenate two 1D waves of the same length you get a 2D wave with two columns. The destination wave is said to be "promoted" to a higher dimensionality.

If you use the /NP (no promotion) flag, the dimensionality of the destination wave is not changed. For example, if you concatenate two 1D waves of the same length using /NP you get a 1D wave whose length is the sum of the lengths of the source waves.

If the source waves are of different lengths, no promotion is done whether /NP is used or not.

Parameters

waveListStr is a string expression containing a list of wave names separated by semicolons. The list must be terminated with a semicolon. The alternate syntax using {*wave1*, *wave2*, …} is limited to 100 waves or less, but there is no limit when using *waveListStr*.

destWave is the name of a new or existing wave that will contain the concatenation result.

Flags

/DL	Sets dimension labels. For promotion, it uses source wave names as new dimension labels otherwise it uses existing labels.
/KILL	Kills source waves.
/NP	Prevents promotion to higher dimension.
/NP=*dim*	Prevents promotion and appends data along the specified dimension (0= rows, 1= columns, 2=layers, 3=chunks). All dimensions other than the one specified by *dim* must be the same in all waves. Requires Igor Pro 6.12 or later.
/O	Overwrites *destWave*.

Type Flags *(used only in functions)*

Concatenate also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-58 and **WAVE Reference Type Flags** on page IV-58 for a complete list of type flags and further details.

Details

If *destWave* does not already exist or, if the /O flag is used, *destWave* is created by duplication of the first source wave. Waves are concatenated in order through the list of source waves. If *destWave* exists and the /O flag is not used, then the concatenation starts with *destWave*.

destWave cannot be used in the source wave list.

Source waves must be either all numeric or all text.

If promotion is allowed, the number of low-order dimensions that all waves share in common determines the dimensionality of *destWave* so that the dimensionality of *destWave* will then be one greater. The default behaviors will vary according to the source wave sizes. Concatenating 1D waves that are all the same length will produce a 2D wave, whereas concatenating 1D waves of differing lengths will produce a 1D wave. Similarly, concatenating 2D waves of the same size will produce a 3D wave; but if the 2D source waves have differing numbers of columns then *destWave* will be a 2D wave, or if the 2D waves have differing numbers of rows then *destWave* will be a 1D wave. Concatenating 1D and 2D waves that have the same number of rows will produce a 2D wave, but when the number of rows differs, *destWave* will be a 1D wave. See the examples.

Use the /NP flag to suppress dimension promotion and keep the dimensionality of *destWave* the same as the input waves.

Examples

```
// Given the following waves:
Make/N=10 w1,w2,w3
Make/N=11 w4
Make/N=(10,7) m1,m2,m3
Make/N=(10,8) m4
Make/N=(9,8) m5

// Concatenate 1D waves
Concatenate/O {w1,w2,w3},wdest          // wdest is a 10x3 matrix
Concatenate {w1,w2,w3},wdest            // wdest is a 10x6 matrix
Concatenate/NP/O {w1,w2,w3},wdest       // wdest is a 30-point 1D wave
Concatenate/O {w1,w2,w3,w4},wdest       // wdest is a 41-point 1D wave

// Concatenate 2D waves
Concatenate/O {m1,m2,m3},wdest          // wdest is a 10x7x3 volume
Concatenate/NP/O {m1,m2,m3},wdest       // wdest is a 10x21 matrix
Concatenate/O {m1,m2,m3,m4},wdest       // wdest is a 10x29 matrix
Concatenate/O {m4,m5},wdest             // wdest is a 152-point 1D wave
Concatenate/O/NP=0 {m4,m5},wdest        // wdest is a 19x8 matrix

// Concatenate 1D and 2D waves
Concatenate/O {w1,m1},wdest             // wdest is a 10x8 matrix
Concatenate/O {w4,m1},wdest             // wdest is a 81-point 1D wave

// Append rows to 2D wave
Make/O/N=(3,2) m6, m7
Concatenate/NP=0 {m6}, m7               // m7 is a 6x2 matrix

// Append columns to 2D wave
Make/O/N=(3,2) m6, m7
Concatenate/NP=1 {m6}, m7               // m7 is a 3x4 matrix

// Append layer to 2D wave
Make/O/N=(3,2) m6, m7
Concatenate/NP=2 {m6}, m7               // m7 is a 3x2x2 volume
// The last command has the same effect as:
// Concatenate {m6}, m7
// Both versions extend add a third dimension to m7
```

See Also

The **Duplicate** and **Redimension** operations.

conj

conj(*z*)

The conj function returns the complex conjugate of the complex value *z*.

See Also

cmplx, **imag**, **magsqr**, **p2rect**, **r2polar**, and **real** functions.

Constant

Constant *kName = literalNumber*

The Constant declaration defines the number *literalNumber* under the name *kName* for use by other code, such as in a switch construct.

See Also

The **Strconstant** keyword for string types, **Constants** on page IV-40 and **Switch Statements** on page IV-34.

continue

continue

The continue flow control keyword returns execution to the beginning of a loop, bypassing the remainder of the loop's code.

See Also

Continue Statement on page IV-38 and **Loops** on page IV-36 for usage details.

ContourInfo

ContourInfo(*graphNameStr, contourWaveNameStr, instanceNumber*)

The ContourInfo function returns a string containing a semicolon-separated list of information about the specified contour plot in the named graph.

Parameters

graphNameStr can be "" to refer to the top graph.

contourWaveNameStr is a string containing either the name of a wave displayed as a contour plot in the named graph, or a contour instance name (wave name with "#n" appended to distinguish the nth contour plot of the wave in the graph). You might get a contour instance name from the **ContourNameList** function.

If *contourWaveNameStr* contains a wave name, *instanceNumber* identifies which instance you want information about. *instanceNumber* is usually 0 because there is normally only one instance of a wave displayed as a contour plot in a graph. Set *instanceNumber* to 1 for information about the second contour plot of the wave, etc. If *contourWaveNameStr* is "", then information is returned on the *instanceNumber*th contour plot in the graph.

If *contourWaveNameStr* contains an instance name, and *instanceNumber* is zero, the instance is taken from *contourWaveNameStr*. If *instanceNumber* is greater than zero, the wave name is extracted from *contourWaveNameStr*, and information is returned concerning the *instanceNumber*th instance of the wave.

Details

The string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon. The keywords are as follows:

Keyword	Information Following Keyword
AXISFLAGS	Flags used to specify the axes. Usually blank because /L and /B (left and bottom axes) are the defaults.
DATAFORMAT	Either XYZ or Matrix.
LEVELS	A comma-separated list of the contour levels, including the final automatic levels, (or manual or from-wave levels), and the "more levels", all sorted into ascending Z order.
RECREATION	List of keyword commands as used by **ModifyContour** command. The format of these keyword commands is: ***keyword (x)=modifyParameters;***
TRACESFORMAT	The format string used to name the contour traces (see **AppendMatrixContour** or **AppendXYZContour**).
XAXIS	X axis name.
XWAVE	X wave name if any, else blank.
XWAVEDF	Full path to the data folder containing the X wave or blank if there is no X wave.
YAXIS	Y axis name.
YWAVE	Y wave name if any, else blank.
YWAVEDF	Full path to the data folder containing the Y wave or blank if there is no Y wave.
ZWAVE	Name of wave containing Z data from which the contour plot was calculated.
ZWAVEDF	Full path to the data folder containing the Z data wave.

The format of the RECREATION information is designed so that you can extract a keyword command from the keyword and colon up to the ";", prepend "ModifyContour", replace the "x" with the name of a contour plot ("data#1" for instance) and then **Execute** the resultant string as a command.

Examples

The following command lines create a very unlikely contour display. If you did this, you would most likely want to put each contour plot on different axes, and arrange the axes such that they don't overlap. That would greatly complicate the example.

```
Make/O/N=(20,20) jack
Display;AppendMatrixContour jack
AppendMatrixContour/T/R jack         // Second instance of jack
```

This example accesses the contour information for the second contour plot of the wave "jack" (which has an instance number of 1) displayed in the top graph:

```
Print StringByKey("ZWAVE", ContourInfo("","jack",1))     // prints jack
```

See Also

The **Execute** and **ModifyContour** operations.

ContourNameList

ContourNameList(*graphNameStr, separatorStr*)

The ContourNameList function returns a string containing a list of contours in the graph window or subwindow identified by *graphNameStr*.

Parameters

graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

The parameter *separatorStr* should contain a single character such as "," or ";" to separate the names.

A contour name is defined as the name of the wave containing the data from which a contour plot is calculated, with an optional #n suffix that distinguishes between two or more contour plots in the same graph window that have the same wave name. Since the contour name has to be parsed, it is quoted if necessary.

Examples

The following command lines create a very unlikely contour display. If you did this, you would most likely want to put each contour plot on different axes, and arrange the axes such that they don't overlap. That would greatly complicate the example.

```
Make/O/N=(20,20) jack,'jack # 2';
Display;AppendMatrixContour jack
AppendMatrixContour/T/R jack
AppendMatrixContour 'jack # 2'
AppendMatrixContour/T/R 'jack # 2'
Print ContourNameList("",";")
```

prints `jack;jack#1;'jack # 2';'jack # 2'#1;`

See Also

Another command related to contour plots and waves: **ContourNameToWaveRef**.

For commands referencing other waves in a graph: **TraceNameList**, **WaveRefIndexed**, **XWaveRefFromTrace**, **TraceNameToWaveRef**, **CsrWaveRef**, **CsrXWaveRef**, **ImageNameList**, and **ImageNameToWaveRef**.

ContourNameToWaveRef

ContourNameToWaveRef(*graphNameStr, contourNameStr*)

Returns a wave reference to the wave corresponding to the given contour name in the graph window or subwindow named by *graphNameStr*.

Parameters

graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

The contour name is identified by the string in *contourNameStr*, which could be a string determined using ContourNameList. Note that the same contour name can refer to different waves in different graphs, if the waves are in different data folders.

See Also

The **ContourNameList** function.

For a discussion of wave reference functions, see **Wave Reference Functions** on page IV-173.

ContourZ

ContourZ(*graphNameStr, contourInstanceNameStr, x, y* [,*pointFindingTolerance*])

The ContourZ function returns the interpolated Z value of the named contour plot data displayed in the named graph.

For gridded contour data, ContourZ returns the bilinear interpolation of the four surrounding XYZ values.

For XYZ triplet contour data, ContourZ returns the value interpolated from the three surrounding XYZ values identified by the Delaunay triangulation.

Parameters

graphNameStr can be "" to specify the topmost graph.

contourNameStr is a string containing either the name of the wave displayed as a contour plot in the named graph, or a contour instance name (wave name with "#n" appended to distinguish the nth contour plot of the wave in the graph). You might get a contour instance name from the **ContourNameList** function.

If *contourNameStr* contains a wave name, *instance* identifies which contour plot of *contourNameStr* you want information about. *instance* is usually 0 because there is normally only one instance of a wave displayed as a contour plot in a graph. Set *instance* to 1 for information about the second contour plot of *contourNameStr*, etc. If *contourNameStr* is "", then information is returned on the *instance*th contour plot in the graph.

If *contourNameStr* contains an instance name, and *instance* is zero, the instance is taken from *contourNameStr*. If *instance* is greater than zero, the wave name is extracted from *contourNameStr*, and information is returned concerning the *instance*th instance of the wave.

x and *y* specify the X and Y coordinates of the value to be returned. This may or may not be the location of a data point in the wave selected by *contourNameStr* and *instance*.

Set *pointFindingTolerance* =1e-5 to overcome the effects of perturbation (see the perturbation keyword of the **ModifyContour** operation).

The default value is 1e-15 to account for rounding errors created by the triangulation scaling (see ModifyContour's equalVoronoiDistances keyword), which works well ModifyContour perturbation=0.

A value of 0 would require an exact match between the scaled x/y coordinate and the scaled and possibly perturbed coordinates to return the original z value; that is an unlikely outcome.

Details

For gridded contour data, ContourZ returns NaN if *x* or *y* falls outside the XY domain of the contour data. If *x* and *y* fall on the contour data grid, the corresponding Z value is returned.

For XYZ triplet contour data, ContourZ returns the null value if *x* or *y* falls outside the XY domain of the contour data. You can set the null value to *v* with this command:

```
ModifyContour contourName nullValue=v
```

If *x* and *y* match one of the XYZ triplet values, the corresponding Z value from the triplet usually won't be returned because Igor uses the Watson contouring algorithm which perturbs the x and y values by a small random amount. This also means that normally x and y coordinates on the boundary will return a null value about half the time if perturbation is on and *pointFindingTolerance* is greater than 1e-5.

Examples

Because ContourZ can interpolate the Z value of the contour data at any X and Y coordinates, you can use ContourZ to convert XYZ triplet data into gridded data:

```
// Make example XYZ triplet contour data
Make/O/N=50 wx,wy,wz
wx= enoise(2)           // x = -2 to 2
wy= enoise(2)           // y = -2 to 2
wz= exp(-(wx[p]*wx[p] + wy[p]*wy[p]))      // XY gaussian, z= 0 to 1
```

```
// ContourZ requires a displayed contour data set
Display; AppendXYZContour wz vs {wx,wy};DelayUpdate
ModifyContour wz autolevels={*,*,0}         // no contour levels are needed
ModifyContour wz xymarkers=1                // show the X and Y locations

// Set the null (out-of-XY domain) value
ModifyContour wz nullValue=NaN              // default is min(wz) - 1

// Convert to grid: Make matrix that spans X and Y
Make/O/N=(30,30) matrix
SetScale/I x, -2, 2, "", matrix
SetScale/I y, -2, 2, "", matrix
matrix= ContourZ("","wz",0,x,y)             // or = ContourZ("","",0,x,y)
AppendImage matrix
```

See Also

The **AppendMatrixContour**, **AppendXYZContour**, and **ModifyContour** operations. The **zcsr** and **ContourInfo** functions.

References

Watson, David F., *Contouring: A Guide To The Analysis and Display of Spatial Data*, Pergamon, 1992.

ControlBar

ControlBar [***flags***] ***barHeight***

The ControlBar operation sets the height and location of the control bar in a graph.

Parameters

barHeight is in pixels. Setting *barHeight* to zero removes the control bar.

Flags

/L/R/B/T	Designates whether to use the Left, Right, Bottom, or Top (default) window edge, respectively, for the control bar location.
/W=*graphName*	Specifies the name of a particular graph containing a control bar.

Details

The control bar is an area at the top of graphs reserved for controls such as buttons, checkboxes and pop-up menus. A line is drawn between this area and the graph area. The control bar may be assigned a separate background color by pressing Control (*Macintosh*) or Ctrl (*Windows*) and clicking in the area, by right-clicking it (*Windows*), or with the **ModifyGraph** operation. You can not use draw tools in this area.

For graphs with no controls you do not need to use this operation.

Examples

```
Display myData
ControlBar 35       // 35 pixels high
Button button0,pos={56,8},size={90,20},title="My Button"
```

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

ControlInfo

ControlInfo [**/W=*winName***] ***controlName***

The ControlInfo operation returns information about the state or status of the named control in a graph or control panel window or subwindow.

Flags

/G [=*doGlobal*]	If *doGlobal* is non-zero or absent, the position returned via V_top and V_left is in global screen coordinates rather relative to the window containing the control.
/W=*winName*	Looks for the control in the named graph or panel window or subwindow. If /W is omitted, ControlInfo looks in the top graph or panel window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Parameters

controlName is the name of the control in *winName* or in the top graph or panel window. *controlName* may also be the keyword kwBackgroundColor to set V_Red, V_Green, and V_Blue, the keyword kwControlBar to set V_Height, or the keyword kwSelectedControl to set S_value and V_flag.

Details

Information for all controls is returned via the following string and numeric variables:

S_recreation Commands to recreate the named control.

V_disable Disable state of control:

- 0: Normal (enabled, visible).
- 1: Hidden.
- 2: Disabled, visible.

V_Height, V_Width, V_top, V_left
Dimensions and position of the named control in pixels.

The kind of control is returned in V_flag as a positive or negative integer. A negative value indicates the control is incomplete or not active. If V_flag is zero, then the named control does not exist. Information returned for specific control types is as follows:

Button
- V_flag: 1
- V_value: Tick count of last mouse up.
- S_UserData: Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
- S_recreation, V_disable, V_Height, V_Width, V_top, V_left: See descriptions of these at the beginning of this section.

Chart
- V_flag: 6 or -6
- V_value: Current point number.
- S_value: Keyword-packed information string. See **S_value for Chart Details** for more keyword information.
- S_recreation, V_disable, V_Height, V_Width, V_top, V_left: See descriptions of these at the beginning of this section.

CheckBox
- V_flag: 2
- V_value: 0 if it is deselected or 1 if it is selected.
- S_UserData: Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
- S_recreation, V_disable, V_Height, V_Width, V_top, V_left: See descriptions of these at the beginning of this section.

CustomControl
- V_flag: 12
- V_value: Tick count of last mouse up.
- S_UserData: Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
- S_value: Name of the picture used to define the control appearance.
- S_recreation, V_disable, V_Height, V_Width, V_top, V_left: See descriptions of these at the beginning of this section.

GroupBox
- V_flag: 9
- S_value: Title text.
- S_recreation, V_disable, V_Height, V_Width, V_top, V_left: See descriptions of these at the beginning of this section.

ListBox
- V_flag: 11
- V_value: Currently selected row (valid for mode 1 or 2 or modes 5 and 6 when no selWave is used). If no list row is selected, then it is set to -1.

	V_selCol	Currently selected column (valid for modes 5 and 6 when no selWave is used).
	V_horizScroll	Number of pixels the list has been scrolled horizontally to the right.
	V_vertScroll	Number of pixels the list has been scrolled vertically downwards.
	V_rowHeight	Height of a row in pixels.
	V_startRow	The current top visible row.
	S_columnWidths	A comma-separated list of column widths in pixels.
	S_dataFolder	Full path to listWave (if any).
	S_UserData	Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
	S_value	Name of listWave (if any).
	S_recreation, V_disable, V_Height, V_Width, V_top, V_left	See descriptions of these at the beginning of this section.
PopupMenu	V_flag	3 or -3
	V_Red, V_Green, V_Blue	For color array pop-up menus, these are the encoded color values.
	V_value	Current item number (counting from one).
	S_UserData	Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
	S_value	Text of the current item. If PopupMenu is a color array then it contains color values encoded as (*r*,*g*,*b*) where *r*, *g*, and *b* are integers from 0 to 65535.
	S_recreation, V_disable, V_Height, V_Width, V_top, V_left	See descriptions of these at the beginning of this section.
SetVariable	V_flag	5 or -5
	V_value	Value of the variable. If the SetVariable is used with a string variable, then it is the interpretation of the string as a number, which will be NaN if conversion fails.
	S_dataFolder	Full path to the variable.
	S_UserData	Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
	S_value	Name of the variable or, if the value was set using _STR: syntax, the string value itself.
	S_recreation, V_disable, V_Height, V_Width, V_top, V_left	See descriptions of these at the beginning of this section.
Slider	V_flag	7
	V_value	Numeric value of the variable.
	S_dataFolder	Full path to the variable.
	S_UserData	Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
	S_value	Name of the variable.
	S_recreation, V_disable, V_Height, V_Width, V_top, V_left	See descriptions of these at the beginning of this section.
TabControl	V_flag	8
	V_value	Number of the current tab.
	S_UserData	Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
	S_value	Tab text.
	S_recreation, V_disable, V_Height, V_Width, V_top, V_left	See descriptions of these at the beginning of this section.

TitleBox
V_flag 10
S_dataFolder Full path if text is from a string variable.
S_value Name if text is from a string variable.
S_recreation, V_disable, V_Height, V_Width, V_top, V_left
See descriptions of these at the beginning of this section.

ValDisplay
V_flag 4 or -4
V_value Displayed value.
S_value Text of expression that ValDisplay evaluates.
S_recreation, V_disable, V_Height, V_Width, V_top, V_left
See descriptions of these at the beginning of this section.

kwBackgroundColor
V_Red, V_Green, V_Blue
If *controlName* is kwBackgroundColor then this is the color of the control panel background. This color is usually the default user interface background color, as set by the Appearance control panel on the Macintosh or by the Appearance tab of the Display Properties on Windows, until changed by `ModifyPanel cbRGB`.

kwControlBar
V_Height If *controlName* is kwControlBar then this is the height (in pixels) of the control bar area in a graph or of an entire panel.

kwSelectedControl
V_flag If *controlName* is kwSelectedControl then V_flag is 1 if a control is selected or 0 if not. (**SetVariable** and **ListBox** controls can be selected, most other controls cannot.)
S_value Name of selected control (if any) or `""`.

S_value for Chart Details

The following applies *only* to the keyword-packed information string returned in S_value for a chart. S_value will consist of a sequence of sections with the format: "*keyword***:***value***;**" You can pick a value out of a keyword-packed string using the **NumberByKey** and **StringByKey** functions. Here are the S_value keywords:

Keyword	Type	Meaning
FNAME	string	Name of the FIFO chart is monitoring.
LHSAMP	number	Left hand sample number.
NCHANS	number	Number of channels displayed in chart.
PPSTRIP	number	The chart's points per strip value.
RHSAMP	number	Right hand sample number (same as V_value).

In addition, ControlInfo writes fields to S_value for each channel in the chart. The keyword for the field is a combination of a name and a number that identify the field and the channel to which it refers. For example, if channel 4 is named "Pressure" then the following would appear in the S_value string: "CHNAME4:Pressure". In the following table, the channel's number is represented by #:

Keyword	Type	Meaning
CHCTAB#	number	Channel's color table value as set by Chart ctab keyword.
CHGAIN#	number	Channel's gain value as set by Chart gain keyword.
CHNAME#	string	Name of channel defined by FIFO.
CHOFFSET#	number	Channel's offset value as set by Chart offset keyword.

Examples

```
ControlInfo myChart; Print S_value
```

Prints the following to the history area:

```
FNAME:myFIFO;NCHANS:1;PPSTRIP:1100;RHSAMP:271;LHSAMP:-126229;
```

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

ControlNameList

ControlNameList(*winNameStr* [, *listSepStr* [, *matchStr*]])

The ControlNameList function returns a string containing a list of control names in the graph or panel window or subwindow identified by *winNameStr*.

Parameters

winNameStr can be "" to refer to the top graph or panel window.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

The optional parameter *listSepStr* should contain a single character such as "," or ";" to separate the names; the default value is ";".

The optional parameter *matchStr* is some combination of normal characters and the asterisk wildcard character that matches anything. To use *matchStr*, *listSepStr* must also be used. See **stringmatch** for wildcard details.

Only control names that satisfy the match expression are returned. For example, "*_tab0" matches all control names that end with "_tab0". The default is "*", which matches all control names.

Examples

```
NewPanel
Button myButton
Checkbox myCheck
Print ControlNameList("")                   // prints "myButton;myCheck;"
Print ControlNameList("", ";", "*Check")    // prints "myCheck;"
```

See Also

The **ListMatch**, **StringFromList** and **stringmatch** functions, and the **ControlInfo** and **ModifyControlList** operations. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

ControlUpdate

ControlUpdate [**/A/W=*winName***][***controlName***]

The ControlUpdate operation updates the named control or all controls in a window, which can be the top graph or control panel or the named graph or control panel if you use /W.

Flags

/A	Updates all controls in the window. You must omit *controlName*.
/W=*winName*	Specifies the window or subwindow containing the control. If you omit *winName* it will use the top graph or control panel window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

ControlUpdate is useful for forcing a pop-up menu to rebuild, to update a ValDisplay control, or to forcibly accept a SetVariable's currently-being-edited value.

Normally, a pop-up menu rebuilds only when the user clicks on it. If you set up a pop-up menu so that its contents depend on a global string variable, on a user-defined string function or on an Igor function (e.g., **WaveList**), you may want to force the pop-up menu to be updated at your command.

Usually, a ValDisplay control displays the value of a global variable or of an expression involving a global variable. If the global variable changes, the ValDisplay will automatically update. However, you can create a ValDisplay that displays a value that does not depend on a global variable. For example, it might display

the result of an external function. In a case like this, the ValDisplay will not automatically update. You can update it by calling ControlUpdate.

When a SetVariable control is being edited, the text the user types isn't "accepted" (or processed) until the user presses Return or Enter. ControlUpdate effectively causes the named control to act as though the user has pressed one of those keys. If /A is specified, the currently active SetVariable control (if any) is affected this way. The motivation here is that the user may have typed a new value without having yet pressed return, and then may click a button in a different panel which runs a routine that uses the SetVariable value as input. The user expected the typed value to have been accepted but the variable has not yet been set. Calling ControlUpdate/A on the first panel will read the typed value in the variable, avoiding a discrepancy between the visible value of the SetVariable control and the actual value of the variable.

Examples

```
NewPanel;DoWindow/C PanelX
String/G popupList="First;Second;Third"
PopupMenu oneOfThree value=popupList        // popup shows "First"
popupList="1;2;3"                           // popup is unchanged
ControlUpdate/W=PanelX oneOfThree           // popup shows "1"
```

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **ValDisplay** and **WaveList** operations.

ConvexHull

convexHull [***flags***]***xwave, ywave***

convexHull [***flags***] ***tripletWave***

The ConvexHull operation calculates the convex hull in either 2 or 3 dimensions. The dimensionality is deduced from the input wave(s). If the input consists of two 1D waves of the same length, the number of dimensions is assumed to be 2. If the input consists of a single triplet wave (a wave of 3 columns), then the number of dimensions is 3.

In 2D cases the operation calculates the convex hull and produces the result in a pair of x and y waves, W_XHull and W_YHull.

In 3D cases the operation calculates the convex hull and stores it in a triplet wave M_Hull that describes ordered facets of the convex hull.

ConvexHull returns an error if the input waves have fewer than 3 data points.

Flags

/C	(2D convex hull only) adds the first point to the end of the W_XHull and W_YHull waves so that the first and the last points are the same.
/E	(3D case only) if you use this flag the operation also creates a wave that lists the indices of the vertices which are not part of the convex hull, i.e., vertices which are interior to the hull. The output is in the wave W_HullExcluded.
/I	(3D convex hull only) use this flag to get the corresponding index of the vertex as the fourth column in the M_Hull wave.
/S	(3D convex hull only) use this flag if you want the resulting M_Hull to have NaN lines separating each triangle.
/T=*tolerance*	(3D case only) default tolerance for measuring if a point is inside or outside the convex hull is 1.0×10^{-20}. You can use any other positive value.
/V	(3D case only) if you use this flag the operation also creates a wave containing the output in a list of vertex indices. The wave M_HullVertices contains a row per triangle where each entry on a row corresponds to the index of the input vertex.
/Z	No error reporting.

Examples

```
Make/O/N=33 xxx=gnoise(5),yyy=gnoise(7)
Convexhull/c xxx,yyy
Display W_Yhull vs W_Xhull
Appendtograph yyy vs xxx
ModifyGraph mode(yyy)=3,marker(yyy)=8,rgb(W_YHull)=(0,15872,65280)
```

See Also
Triangulate3D

Convolve

Convolve [**/A/C**] ***srcWaveName, destWaveName*** [**, *destWaveName***]…

The Convolve operation convolves *srcWaveName* with each destination wave, putting the result of each convolution in the corresponding destination wave.

Convolve is not multidimensional aware. Some multidimensional convolutions are covered by the **MatrixConvolve**, **MatrixFilter**, and **MatrixOp** operations

Flags

/A	Acausal linear convolution.
/C	Circular convolution.

Details

Convolve performs linear convolution unless the /C or /A flag is used. See the diagrams in the examples below.

Depending on the type of convolution, the destination waves' lengths may increase. *srcWaveName* is not altered unless it also appears as a destination wave.

If *srcWaveName* is real-valued, each destination wave must be real-valued, and if *srcWaveName* is complex, each destination wave must be complex, too. Double and single precision waves may be freely intermixed; calculations are performed in the higher precision.

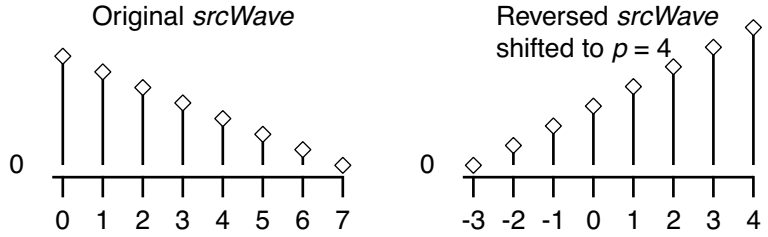
The linear convolution equation is:

$$destWaveOut[p] = \sum_{m=0}^{N-1} destWaveIn[m] \cdot srcWave[p-m]$$

where *N* is the number of points in the longer of *destWaveIn* and *srcWave*. For circular convolution, the index [*p -m*] is wrapped around when it exceeds the range of [0,numpnts(*srcWave*)-1]. For acausal convolution, when [*p -m*] exceeds the range a zero value is substituted for *srcWave* [*p -m*]. Similar operations are applied to *destWaveIn* [*m*].

Another way of looking at this equation is that, for all *p*, *destWaveOut*[*p*] equals the sum of the point-by-point products from 0 to *p* of the destination wave and an end-to-end reversed copy of the source wave that has been shifted to the right by *p*.

The following diagram shows the reversed/shifted *srcWave* that would be combined with *destWaveIn*. The points numbered 0 through 4 of the reversed *srcWave* would be multiplied with *destWaveIn*[0…4] and summed to produce *destWaveOut*[4]:



For linear and acausal convolution, the destination wave is first zero-padded by one less than the length of the source wave. This prevents the "wrap-around" effect that occurs in circular convolution. The zero-padded points are removed after acausal convolution, and retained after linear convolution. The X scaling of the waves is ignored.

The convolutions are performed by transforming the source and destination waves with the Fast Fourier Transform, multiplying them in the frequency domain, and then inverse-transforming them into the destination wave(s).

The convolution is performed in segments if the resulting wave has more than 256 points and the destination wave has twice as many points as the source wave. For acausal convolution, the length of the resulting wave is considered to be (numpnts(*srcWaveName*) +numpnts(*destWaveName*)-1) for this calculation.

Applications

The usual application of convolution is to compute the response of a linear system defined by its impulse response to an input signal. *srcWaveName* would contain the impulse response, and the destination wave would initially contain the input signal. After the Convolve operation has completed, the destination wave contains the output signal.

Use linear convolution when the source wave contains an impulse response (or filter coefficients) where the first point of *srcWave* corresponds to no delay ($t = 0$).

Use circular convolution for the case where the data in *srcWaveName* and *destWaveName* are considered to endlessly repeat (or "wrap around" from the end back to the start), which means no zero padding is needed.

Use acausal convolution when the source wave contains an impulse response where the middle point of *srcWave* corresponds to no delay ($t = 0$).

See Also

Convolution on page III-249 for illustrated examples. **MatrixOp**.

References

A very complete explanation of circular and linear convolution can be found in sections 2.23 and 2.24 of Rabiner and Gold, *Theory and Application of Digital Signal Processing*, Prentice Hall, 1975.

CopyFile

CopyFile [***flags***][***srcFileStr***] [**as** ***destFileOrFolderStr***]

The CopyFile operation copies a file on disk.

Parameters

srcFileStr can be a full path to the file to be copied (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*.

If Igor can not determine the location of the source file from *srcFileStr* and *pathName*, it displays a dialog allowing you to specify the source file.

destFileOrFolderStr is interpreted as the name of (or path to) an existing folder when /D is specified, otherwise it is interpreted as the name of (or path to) a possibly existing file.

If *destFileOrFolderStr* is a partial path, it is relative to the folder associated with pathName.

If /D is specified, the source file is copied inside the folder using the source file's name.

If Igor can not determine the location of the destination file from *pathName*, *srcFileStr*, and *destFileOrFolderStr*, it displays a Save File dialog allowing you to specify the destination file (and folder).

If you use a full or partial path for either *srcFileStr* or *destFileOrFolderStr*, see **Path Separators** on page III-398 for details on forming the path.

Folder paths should not end with single Path Separators. See the **Details** section for **MoveFolder**.

Flags

/D — Interprets *destFileOrFolderStr* as the name of (or path to) an existing folder (or "directory"). Without /D, *destFileOrFolderStr* is interpreted as the name of (or path to) a file.

If *destFileOrFolderStr* is not a full path to a folder, it is relative to the folder associated with pathName.

/I [=*i*] — Specifies the level of user interactivity.

- /I=0: Interactive only if one or *srcFileStr* or *destFileOrFolderStr* is not specified or if the source file is missing. (Same as if /I was not specified.)
- /I=1: Interactive even if *srcFileStr* is specified and the source file exists.
- /I=2: Interactive even if *destFileOrFolderStr* is specified.
- /I=3: Interactive even if *srcFileStr* is specified, the source file exists, and *destFileOrFolderStr* is specified. Same as /I only.

/M=*messageStr*	Specifies the prompt message in the Open File dialog. If /S is not used, then *messageStr* will be used for both Open File and for Save File dialogs.
/O	Overwrites any existing destination file.
/P=*pathName*	Specifies the folder to look in for the source file, and the folder into which the file is copied. *pathName* is the name of an existing symbolic path. Using /P means that both *srcFileStr* and *destFileOrFolderStr* must be either simple file or folder names, or paths relative to the folder specified by *pathName*.
/S=*saveMessageStr*	Specifies the prompt message in the Save File dialog.
/Z [=*z*]	Prevents procedure execution from aborting if it attempts to copy a file that does not exist. Use /Z if you want to handle this case in your procedures rather than aborting execution. /Z=0: Same as no /Z. /Z=1: Copies a file only if it exists. /Z alone has the same effect as /Z=1. /Z=2: Copies a file if it exists or displays a dialog if it does not exist.

Variables

The CopyFile operation returns information in the following variables:

`V_flag`	Set to zero if the file was copied, to -1 if the user cancelled either the Open File or Save File dialogs, and to some nonzero value if an error occurred, such as the specified file does not exist.
`S_fileName`	Stores the full path to the file that was copied. If an error occurred or if the user cancelled, it is set to an empty string.
`S_path`	Stores the full path to the file copy. If an error occurred or if the user cancelled, it is set to an empty string.

Examples

Copy a file within the same folder using a new name:

```
CopyFile/P=myPath "afile.txt" as "destFile.txt"
```

Copy a file into subfolder using the original name (using /P):

```
CopyFile/D/P=myPath "afile.txt" as ":subfolder"
Print S_Path     // prints "Macintosh HD:folder:subfolder:afile.txt"
```

Copy file into subfolder using the original name (using full paths):

```
CopyFile/D "Macintosh HD:folder:afile.txt" as "Server:archive"
```

Copy a file from one folder to another, assigning the copy a new name:

```
CopyFile "Macintosh HD:folder:afile.txt" as "Server:archive:destFile.txt"
```

Copy user-selected file in any folder as destFile.txt in myPath folder (prompt to save even if destFile.txt doesn't exist):

```
CopyFile/I=2/P=myPath as "destFile.txt"
```

Copy user-selected file in any folder as destFile.txt in any folder:

```
CopyFile as "destFile.txt"
```

See Also

The **Open**, **MoveFile**, **DeleteFile**, and **CopyFolder** operations. The **IndexedFile** function. **Symbolic Paths** on page II-34.

CopyFolder

`CopyFolder` `[`***`flags`***`]`[***`srcFolderStr`***`]` `[`**`as`** ***`destFolderStr`***`]`

The CopyFolder operation copies a folder (and its contents) on disk.

Warning: *The CopyFolder command can destroy data* by overwriting another folder and contents!

When overwriting an existing folder on disk, CopyFolder will do so only if permission is granted by the user. The default behavior is to display a dialog asking for permission. The user can alter this behavior via the Miscellaneous Settings dialog's Misc category. For further details see **Misc Settings** on page III-414.

If permission is denied, the folder will not be copied and V_Flag will return 1088 (Command is disabled) or 1275 (You denied permission to overwrite a folder). Command execution will cease unless the /Z flag is specified.

Parameters

srcFolderStr can be a full path to the folder to be copied (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a folder inside the folder associated with *pathName*.

If Igor can not determine the location of the folder from *srcFolderStr* and *pathName*, it displays a dialog allowing you to specify the source folder.

If /P=*pathName* is given, but *srcFolderStr* is not, then the folder associated with *pathName* is copied.

destFolderStr can be a full path to the output (destination) folder (in which case /P is not needed), or a partial path relative to the folder associated with *pathName*.

An error is returned if the destination folder would be inside the source folder.

If Igor can not determine the location of the destination folder from *destFolderStr* and *pathName*, it displays a dialog allowing you to specify or create the destination folder.

If you use a full or partial path for either folder, see **Path Separators** on page III-398 for details on forming the path.

Flags

/D — Interprets *destFolderStr* as the name of (or path to) an existing folder (or directory) to copy the source folder into. Without /D, *destFolderStr* is interpreted as the name of (or path to) the copied folder.

If *destFolderStr* is not a full path to a folder, it is relative to the folder associated with *pathName*.

/I [=*i*] — Specifies the level of user interactivity.

- /I=0: Interactive only if the source or destination folder is not specified or if the source folder is missing. (Same as if /I was not specified.)
- /I=1: Interactive even if the source folder is specified and it exists.
- /I=2: Interactive even if *destFolderStr* is specified.
- /I=3: Interactive even if the source folder is specified, the source file exists, and *destFolderStr* is specified. Same as /I only.

/M=*messageStr* — Specifies the prompt message in the Select (source) Folder dialog. If /S is not used, then *messageStr* will be used for the Select Folder dialog and for the Create Folder dialog.

/O — Overwrite existing destination folder, if any.

On Macintosh, a Macintosh-style overwrite-move is performed in which the source folder completely replaces the destination folder.

On Windows, a Windows-style mix-in move is performed in which the contents of the source folder are moved into the destination folder, replacing any same-named files but leaving other files in place.

/P=*pathName* — Specifies the folder to look in for the source folder. *pathName* is the name of an existing symbolic path.

If *srcFolderStr* is not specified, the folder associated with *pathName* is copied.

Using /P means that *srcFolderStr* (if specified) and *destFolderStr* must be either simple folder names or paths relative to the folder specified by *pathName*.

/S=*saveMessageStr* — Specifies the prompt message in the Create Folder dialog.

/Z [=*z*] — Prevents procedure execution from aborting if it attempts to copy a file that does not exist. Use /Z if you want to handle this case in your procedures rather than aborting execution.

- /Z=0: Same as no /Z.
- /Z=1: Copies a folder only if it exists. /Z alone has the same effect as /Z=1.
- /Z=2: Copies a folder if it exists or displays a dialog if it does not exist.

Variables

The CopyFolder operation returns information in the following variables:

V_flag	Set to zero if the folder was copied, to -1 if the user cancelled either the Select Folder or Create Folder dialogs, and to some nonzero value if an error occurred, such as the specified file does not exist.
S_fileName	Stores the full path to the folder that was copied, with a trailing semicolon. If an error occurred or if the user cancelled, it is set to an empty string.
S_path	Stores the full path to the folder copy, with a trailing semicolon. If an error occurred or if the user cancelled, it is set to an empty string.

Details

You can use only /P=*pathName* (without *srcFolderStr*) to specify the source folder to be copied.

Folder paths should not end with single Path Separators. See the **Details** section for **MoveFolder**.

Examples

Copy the folder that the current experiment is stored in:

```
CopyFile/P=home as "HD:Copy Of Folder Experiment Is In"
```

Copy the Igor Extensions Folder to the Windows desktop:

```
CopyFile/D/P=Igor ":Igor Extensions" as "C:WINDOWS:Desktop"
```

Ask the user to select a folder, starting with the Igor folder, and then make a copy of that folder in the Igor Pro folder:

```
CopyFile/I=2/P=Igor as "::Folder Copy"
```

Copy an entire disk inside a folder:

```
CopyFolder/O/D "Floppy" as "HD:Desktop Folder:Copy Into Here"
```

See Also

Open, **MoveFile**, **DeleteFile**, **MoveFolder**, **NewPath**, and **IndexedDir** operations, and **Symbolic Paths** on page II-34.

CopyScales

CopyScales [**/I/P**] ***srcWaveName, waveName*** [**, *waveName***]…

The CopyScales operation copies the x, y, z, and t scaling, x, y, z, and t units, the data Full Scale and data units from *srcWaveName* to the other waves.

Flags

/I	Copies the x, y, z, and t scaling in inclusive format.
/P	Copies the x, y, z, and t scaling in slope/intercept format (x0, dx format).

Details

Normally the x, y, z, and t (dimension) scaling is copied in min/max format. However, if you use /P, the dimension scaling is copied in slope/intercept format so that if *srcWaveName* and the other waves have differing dimension size (number of points if the wave is a 1D wave), then their dimension values will still match for the points they have in common. Similarly, /I uses the inclusive variant of the min/max format. See **SetScale** for a discussion of these dimension scaling formats.

If a wave has only one point, /I mode reverts to /P mode.

CopyScales copies scales only for those dimensions that *srcWaveName* and *waveName* have in common.

See Also

x, **y**, **z**, and **t** scaling functions.

Correlate

Correlate [**/AUTO/C/NODC**] ***srcWaveName, destWaveName*** [**, *destWaveName***]…

The Correlate operation correlates *srcWaveName* with each destination wave, putting the result of each correlation in the corresponding destination wave.

Flags

/AUTO	Auto-correlation scaling. This forces the X scaling of the destination wave's center point to be x=0, and divides the destination wave by the center point's value so that the center value is exactly 1.0.

If srcWaveName and destWaveName do not have the same number of points, this flag is ignored.

/AUTO is not compatible with /C.

/C Circular correlation. (See **Compatibility Note**.)

/NODC Removes the mean from the source and destination waves before computing the correlations. Removing the mean results in the un-normalized auto- or cross-covariance.

"DC" is an abbrevation of "direct current", an electronics term for the non-varying average value component of a signal.

Details

Note: To compute a single-value correlation number use the **StatsCorrelation** function which returns the Pearson's correlation coefficient of two same-length waves.

Correlate performs linear correlation unless the /C flag is used.

Depending on the type of correlation, the length of the destination may increase. *srcWaveName* is not altered unless it also appears as a destination wave.

If the source wave is real-valued, each destination wave must be real-valued and if the source wave is complex, each destination wave must be complex, too. Double and single precision waves may be freely intermixed; calculations are performed in the higher precision.

The linear correlation equation is:

$$destWaveOut[p] = \sum_{m=0}^{N-1} srcWave[m] \cdot destWaveIn[p+m]$$

where N is the number of points in the longer of *destWaveIn* and *srcWave*.

For circular correlation, the index [$p+m$] is wrapped around when it exceeds the range of `[0,numpnts(destWaveIn)-1]`. For linear correlation, when [$p+m$] exceeds the range a zero value is substituted for *destWaveIn*[$p+m$]. When m exceeds `numpnts(srcWave)-1`, 0 is used instead of *srcWave*[m].

Comparing this with the **Convolve** operation, which is the linear convolution:

$$destWaveOut[p] = \sum_{m=0}^{N-1} destWaveIn[m] \cdot srcWave[p-m]$$

you can see that the only difference is that for correlation the source wave is *not* reversed before shifting and combining with the destination wave.

The Correlate operation is not multidimensional aware. For details, see **Analysis on Multidimensional Waves** on page II-110 and in particular **Analysis on Multidimensional Waves** on page II-110.

Compatibility Note

Prior to Igor Pro 5, Correlate/C scaled and rotated the results improperly (the result was often rotated left by one and the X scaling was entirely negative).

Now the destination wave's X scaling is unaltered and it does not rotate the result. You can force the old behavior for compatibility with old procedures that depend on the old behavior by setting `root:V_oldCorrelationScaling=1`.

A better way to get identical Correlate/C results with all versions of Igor Pro is to use this code, which rotates the result so that x=0 is always the first point in *destWave*, no matter which Igor Pro version runs this code (currently, it doesn't change anything and runs extremely quickly because it does no rotation):

```
Correlate/C srcWave, destWave
Variable pointAtXEqualZero= x2pnt(destWave,0)     // 0 for Igor Pro 5
Rotate -pointAtXEqualZero,destWave
SetScale/P x, 0, DimDelta(destWave,0), "", destWave
```

Applications

A common application of correlation is to measure the similarity of two input signals as they are shifted by one another.

Often it is desirable to normalize the correlation result to 1.0 at the maximum value where the two inputs are most similar. To normalize *destWaveOut*, compute the RMS values of the input waves and the number of points in each wave:

```
WaveStats/Q srcWave
Variable srcRMS = V_rms
Variable srcLen = numpnts(srcWave)

WaveStats/Q destWave
Variable destRMS = V_rms
Variable destLen = numpnts(destWave)

Correlate srcWave, destWave                // overwrites destWave

// now normalize to max of 1.0
destWave /= (srcRMS * sqrt(srcLen) * destRMS * sqrt(destLen))
```

Another common application is using autocorrelation (where *srcWaveName* and *destWaveName* are the same) to determine Power Spectral Density. In this case it better to use the **DSPPeriodogram** operation which provides more options.

See Also

Convolution on page III-249 and **Correlation** on page III-251 for illustrated examples. See the **Convolve** operation for algorithm implementation details, which are identical except for the lack of source wave reversal, and the lack of the /A (acausal) flag.

The **MatrixOp**, **StatsCorrelation**, **StatsCircularCorrelationTest**, **StatsLinearCorrelationTest**, and **DSPPeriodogram** operations.

References

An explanation of autocorrelation and Power Spectral Density (PSD) can be found in Chapter 12 of Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

WaveMetrics provides Igor Technical Note 006, "DSP Support Macros" that computes the PSD with options such as windowing and segmenting. See the Technical Notes folder. Some of the techniques discussed there are available as Igor procedure files in the "WaveMetrics Procedures:Analysis:" folder.

Wikipedia: http://en.wikipedia.org/wiki/Correlation

Wikipedia: http://en.wikipedia.org/wiki/Cross_covariance

Wikipedia: http://en.wikipedia.org/wiki/Autocorrelation_function

cos

cos(*angle*)

The cos function returns the cosine of *angle* which is in radians.

In complex expressions, *angle* is complex, and `cos(`*angle*`)` returns a complex value:

$$\cos(x+iy) = \cos(x)\cosh(y) - i\sin(x)\sinh(y).$$

See Also

sin, **tan**, **sec**, **csc**, **cot**

cosh

cosh(*num*)

The cosh function returns the hyperbolic cosine of *num*:

$$\cosh(x) = \frac{e^x + e^{-x}}{2}.$$

In complex expressions, *num* is complex, and `cosh(`*num*`)` returns a complex value.

See Also

sinh, **tanh**, **coth**

cot

cot(*angle*)

The cot function returns the cotangent of *angle* which is in radians.

In complex expressions, *angle* is complex, and cot(*angle*) returns a complex value.

See Also

sin, **cos**, **tan**, **sec**, **csc**

coth

coth(*num*)

The coth function returns the hyperbolic cotangent of *num*:

$$\coth(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}}.$$

In complex expressions, *num* is complex, and coth(*num*) returns a complex value.

See Also

sinh, **cosh**, **tanh**

CountObjects

CountObjects(*sourceFolderStr, objectType*)

The CountObjects function returns the number of objects of the specified type in the data folder specified by the string expression.

For Igor Pro 6.1 or later, **CountObjectsDFR** is preferred.

Parameters

sourceFolderStr can be either ":" or "" to specify the current data folder. You can also use a full or partial data folder path. *objectType* should be one of the following values:

objectType	**What CountObjects Counts**
1	Waves
2	Numeric variables
3	String variables
4	Data folders

See Also

Chapter II-8, **Data Folders**, and the **GetIndexedObjName** function.

CountObjectsDFR

CountObjectsDFR(*dfr,objectType*)

The CountObjectsDFR function returns the number of objects of the specified type in the data folder specified by the data folder reference *dfr*.

Requires Igor Pro 6.1 or later.

CountObjectsDFR is the same as CountObjects except the first parameter, *dfr*, is a data folder reference instead of a string containing a path.

Parameters

objectType is one of the following values:

objectType	What CountObjectsDFR Counts
1	Waves
2	Numeric variables
3	String variables
4	Data folders

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-61.

GetIndexedObjNameDFR

cpowi

cpowi(*num, ipow*)

This function is obsolete as the exponentiation operator ^ handles complex expressions with any combination of real, integer and complex arguments. See **Operators** on page IV-5. The cpowi function returns a complex number resulting from raising complex *num* to integer-valued power *ipow*. *ipow* can be positive or negative, but if it is not an integer cpowi returns (NaN, NaN).

CreateAliasShortcut

CreateAliasShortcut [*flags*][*targetFileDirStr*] [as *aliasFileStr*]

The CreateAliasShortcut operation creates an alias (*Macintosh*) or shortcut (*Windows*) file on disk. The alias can point to either a file or a folder. The file or folder pointed to is called the "target" of the alias or shortcut.

Parameters

targetFileDirStr can be a full path to the file or folder to make an alias or shortcut for, a partial path relative to the folder associated with /P=*pathName*, or the name of a file or folder in the folder associated with *pathName*.

If Igor can not determine the location of the file or folder from *targetFileDirStr* and *pathName*, it displays a dialog allowing you to specify a target file. Use /D to select a folder as the alias target, instead.

aliasFileStr can be a full path to the created alias file, a partial path relative to the folder associated with *pathName* if specified, or the name of a file in the folder associated with *pathName*.

If Igor can not determine the location of the alias or shortcut file from *aliasFileStr* and *pathName*, it displays a File Save dialog allowing you to create the file.

If you use a full or partial path for either *targetFileDirStr* or *aliasFileStr*, see **Path Separators** on page III-398 for details on forming the path.

Folder paths should not end with single path separators. See the **MoveFolder Details** section.

Flags

/D — Uses the Select Folder dialog rather than Open File dialog when *targetFileDirStr* is not fully specified.

/I [=*i*] — Specifies the level of user interactivity.

- *i*=0: Interactive only if one or *targetFileDirStr* or *aliasFileStr* is not specified or if the target file is missing. (Same as if /I was not specified.)
- *i*=1: Interactive even if *targetFileDirStr* is fully specified and the target file exists.
- *i*=2: Interactive even if *targetFileDirStr* is specified.
- *i*=3: Interactive even if *targetFileDirStr* is specified and the target file exists. Same as /I only.

/M=*messageStr* — Specifies the prompt message in the Open File or Select Folder dialog. If /S is not specified, then *messageStr* will be used for Open File (or Select Folder) and for Save File dialogs.

/O — Overwrites any existing file with the alias or shortcut file.

/P=*pathName* — Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

/S=*saveMessageStr* Specifies the prompt message in the Save File dialog when creating the alias or shortcut file.

/Z[=*z*] Prevents procedure execution from aborting if the procedure tries to copy a file that does not exist. Use /Z if you want to handle this case in your procedures rather than aborting execution.

- /Z=0: Same as no /Z.
- /Z=1: Creates an alias to a file or folder only if it exists. /Z alone has the same effect as /Z=1.
- /Z=2: Creates an alias to a file or folder only if it exists and displaying a dialog if it does not exist.

Variables

The CreateAliasShortcut operation returns information in the following variables:

`V_flag`
- 0: Created an alias or shortcut file.
- -1: User cancelled any of the Open File, Select Folder, or Save File dialogs.
- Other: An error occurred, such as the target file does not exist.

`S_path` Full path to the target file or folder. If an error occurred or if the user cancelled, it is an empty string.

`S_fileName` Full path to the created alias or shortcut file. If an error occurred or if the user cancelled, it is an empty string.

Examples

Create a shortcut (*Windows*) to the current experiment, on the desktop:

```
String target= Igorinfo(1)+".pxp" // experiments are usually .pxp on Windows
CreateAliasShortcut/O/P=home target as "C:WINDOWS:Desktop:"+target
```

Create an alias (*Macintosh*) to the VDT XOP in the Igor Extensions folder:

```
String target= ":More Extensions:Data Acquisition:VDT"
CreateAliasShortcut/O/P=Igor target as ":Igor Extensions:VDT alias"
```

Create an alias to the "HD 2" disk. Put the alias on the desktop:

```
CreateAliasShortcut/D/O "HD 2" as "HD:Desktop Folder:Alias to HD 2"
```

See Also

Symbolic Paths on page II-34.

The **Open**, **MoveFile**, **DeleteFile**, and **GetFileFolderInfo** operations. The **IgorInfo** and **ParseFilePath** functions.

CreationDate

`CreationDate(`*`waveName`*`)`

Returns creation date of wave as an Igor date/time value, which is the number of seconds from 1/1/1904.

The returned value is valid for waves created with Igor Pro 3.0 or later. For waves created in earlier versions, it returns 0.

See Also

modDate.

Cross

`Cross [/T/Z] `*`vectorA, vectorB`*` [, `*`vectorC`*`]`

The Cross operation computes the cross products *vectorA* x *vectorB* and *vectorA* x (*vectorB* x *vectorC*). Each vector is a 1D real wave containing 3 rows. Stores the result in the wave W_Cross in the current data folder.

Flags

/T Stores output in a row instead of a column in W_Cross.

/Z Generates no errors for any unsuitable inputs.

csc

csc(*angle*)

The csc function returns the cosecant of *angle* which is in radians.

$$\csc(x) = \frac{1}{\sin(x)}.$$

In complex expressions, *angle* is complex, and csc(*angle*) returns a complex value.

See Also

sin, **cos**, **tan**, **sec**, **cot**

CsrInfo

CsrInfo(*cursorName* [, *graphNameStr*])

The CsrInfo function returns a keyword-value pair list of information about the specified cursor (*cursorName* is A through J) in the top graph or graph specified by *graphNameStr*. It returns "" if the cursor is not in the graph.

Details

The returned string contains information about the cursor in the following format:

```
TNAME:traceName; ISFREE:freeNum;POINT:xPointNumber;[YPOINT:yPointNumber;]
RECREATION:command;
```

The *traceName* value is the name of the graph trace or image to which it is attached or which supplies the x (and y) values even if the cursor isn't attached to it.

If TNAME is empty, fields POINT, ISFREE, and YPOINT are not present.

The *freeNum* value is 1 if the cursor is not attached to anything, 0 if attached to a trace or image.

The POINT value is the same value **pcsr** returns.

The YPOINT keyword and value are present only when the cursor is attached to a two-dimensional item such as an image, contour, or waterfall plot or when the cursor is free. It's value is the same as returned by **qcsr**.

If cursor is free, POINT and YPOINT values are fractional relative positions (see description in the **Cursor** command).

The RECREATION keyword contains the Cursor commands (including /W) necessary to regenerate the current settings.

Examples

```
Variable aExists= strlen(CsrInfo(A)) > 0   // A is a name, not a string
Variable bIsFree= NumberByKey("ISFREE",CsrInfo(B,"Graph0"))
```

See Also

The **Cursor** operation and the **hcsr**, **pcsr**, **qcsr**, **vcsr**, **xcsr**, and **zcsr** functions. **Cursors — Moving Cursor Calls Function** on page IV-294.

CsrWave

CsrWave(*cursorName* [, *graphNameStr* [, *wantTraceName*]])

The CsrWave function returns a string containing the name of the wave the specified cursor (A through J) is on in the top (or named) graph. If the optional *wantTraceName* is nonzero, the trace name is returned. A trace name is the wave name with optional instance notation (see **ModifyGraph (traces)**).

Details

The name of a wave by itself is not sufficient to identify the wave because it does not specify what data folder contains the wave. Thus, if you are calling CsrWave for the purpose of passing the wave name to other procedures, you should use the **CsrWaveRef** function instead. Use CsrWave if you want the name of the wave to use in an annotation or a notebook.

Examples

```
String waveCursorAIsOn = CsrWave(A)            // not CsrWave("A")
String waveCursorBIsOn = CsrWave(B,"Graph0")   // in specified graph
String traceCursorBIsOn = CsrWave(B,"",1)      // trace name in top graph
```

CsrWaveRef

CsrWaveRef(*cursorName* [, *graphNameStr*])

The CsrWaveRef function returns a wave reference to the wave the specified cursor (A through J) is on in the top (or named) graph.

Details

The wave reference can be used anywhere Igor is expecting the name of a wave (not a string containing the name of a wave).

CsrWaveRef should be used in place of the CsrWave() string function to work properly with data folders.

Examples

```
Print CsrWaveRef(A)[50]                 // not CsrWaveRef("A")
Print CsrWaveRef(B,"Graph0")[50]        // in specified graph
```

See Also

See **Wave Reference Functions** on page IV-173.

CsrXWave

CsrXWave(*cursorName* [, *graphNameStr*])

The CsrXWave function returns a string containing the name of the wave supplying the X coordinates for an XY plot of the Y wave the specified cursor (A through J) is attached to in the top (or named) graph.

Details

CsrXWave returns an empty string ("") if the wave the cursor is on is not plotted versus another wave providing the X coordinates (that is, if the wave was not plotted with a command such as `Display theWave vs anotherWave`).

The name of a wave by itself is not sufficient to identify the wave because it does not specify what data folder contains the wave. Thus, if you are calling CsrXWave for the purpose of passing the wave name to other Igor procedures, you should use the **CsrXWaveRef** function instead. Use CsrXWave if you want the name of the wave to use in an annotation or a notebook.

Examples

```
Display ywave vs xwave
```

ywave supplies the Y coordinates and xwave supplies the X coordinates for this XY plot.

```
Cursor A ywave,0
Print CsrXWave(A)       // prints xwave
```

CsrXWaveRef

CsrXWaveRef(*cursorName* [, *graphNameStr*])

The CsrXWaveRef function returns a wave reference to the wave supplying the X coordinates for an XY plot of the Y wave the specified cursor (A through J) is attached to in the top (or named) graph.

Details

The wave reference can be used anywhere Igor is expecting the name of a wave (not a string containing the name of a wave).

CsrXWaveRef returns a null reference (see **WaveExists**) if the wave the cursor is on is not plotted versus another wave providing the X coordinates (that is, if the wave was not plotted with a command such as `Display theWave vs anotherWave`). CsrXWaveRef should be used in place of the CsrXWave string function to work properly with data folders.

Examples

```
Display ywave vs xwave
```

ywave supplies the Y coordinates and xwave supplies the X coordinates for this XY plot.

```
Cursor A ywave,0
Print CsrXWaveRef(A)[50]      // prints value of xwave at point #50
```

See Also

See **Wave Reference Functions** on page IV-173.

CTabList

CTabList()

The CTabList string function returns a semicolon-separated list of the names of built-in color tables. This can be useful when creating pop-up menus in control panels.

Color tables available through version 4:

Grays	Rainbow	YellowHot	BlueHot	BlueRedGreen
RedWhiteBlue	PlanetEarth	Terrain		

Additional color tables added for version 5:

Grays256	Rainbow256	YellowHot256	BlueHot256	BlueRedGreen256
RedWhiteBlue256	PlanetEarth256	Terrain256	Grays16	Rainbow16
Red	Green	Blue	Cyan	Magenta
Yellow	Copper	Gold	CyanMagenta	RedWhiteGreen
BlueBlackRed	Geo	Geo32	LandAndSea	LandAndSea8
Relief	Relief19	PastelsMap	PastelsMap20	Bathymetry9
BlackBody	Spectrum	SpectrumBlack	Cycles	Fiddle
Pastels				

Additional color tables added for version 6:

RainbowCycle	Rainbow4Cycles	GreenMagenta16	dBZ14	dBZ21
Web216	BlueGreenOrange	BrownViolet	ColdWarm	Mocha
VioletOrangeYellow	SeaLandAndFire			

Additional color tables added for version 6.2:

Mud	Classification

See Also

See **Color Tables** on page II-349 and **ColorTab2Wave**.

CtrlBackground

CtrlBackground [***key*** [**= *value***]]…

The CtrlBackground operation controls the unnamed background task.

CtrlBackground works only with the unnamed background task. New code should used named background tasks instead. See **Background Tasks** on page IV-279 for details.

Parameters

dialogsOK=1 *or* 0	If 1, your task will be allowed to run while an Igor dialog is present. This can potentially cause crashes unless your task is well-behaved.
noBurst=1 *or* 0	Normally (or noBurst=0), your task will be called at maximum rate if a delay causes normal run times to be missed. Using noBurst=1, will suppress this burst catch up mode.
period=*deltaTicks*	Sets the minimum number of ticks that must pass between invocations of the background task.
start[=*startTicks*]	Starts the background task (designated by SetBackground) when the tick count reaches *startTicks*. If you omit *startTicks* the task starts immediately.
stop	Stops the background task.

See Also

The **BackgroundInfo**, **SetBackground**, **CtrlNamedBackground**, **KillBackground**, and **SetProcessSleep** operations, and **Background Tasks** on page IV-279.

CtrlNamedBackground

CtrlNamedBackground *taskName*, *keyword* = *value* [, *keyword* = *value* …]

The CtrlNamedBackground operation creates and controls named background tasks.

We recommend that you see **Background Tasks** on page IV-279 for an orientation before working with background tasks.

Important: Unlike the unnamed background task, by default named tasks run when a dialog window is active. This can cause a crash if the background task does things the dialog does not expect. See **Background Tasks and Dialogs** on page IV-281 for details.

Parameters

taskName	*taskName* is the name of the background task or _all_ to control all named background tasks. You can use any valid standard Igor object name as the background task name.
burst [= *b*]	Enable burst catch up mode (off by default, *b*=0). When on (*b*=1), the task is called at the maximum rate if a delay misses normal run times.
dialogsOK [= *d*]	Use dialogsOK=0 to prevent the background task from running when a dialog window is active. By default, dialogsOK=1 is in effect. See **Background Tasks and Dialogs** on page IV-281 for details.
kill [= *k*]	Stops and releases task memory for reuse (*k*=1; default) or continues (*k*=0).
period=*deltaTicks*	Sets the minimum number of ticks (*deltaTicks*) that must pass between background task invocations. *deltaTicks* is truncated to an integer and clipped to a value greater than zero. See **Background Task Period** on page IV-280 for details.
proc=*funcName*	Specifies name of a background user function (see **Details**).
start [=*startTicks*]	Starts when the tick count reaches *startTicks*. A task starts immediately without *startTicks*.
status	Returns background task information in the S_info string variable.
stop [= *s*]	Stops the background task (*s*=1; default) or continues (*s*=0).

Details

The user function you specify via the proc keyword must have the following format:

```
Function myFunc(s)
   STRUCT WMBackgroundStruct &s
   …
```

The members of the `WMBackgroundStruct` are:

Base `WMBackgroundStruct` Structure Members

Member	Description
`char name[MAX_OBJ_NAME+1]`	Background task name.
`uint32 curRunTicks`	Tick count when task was called.
`int32 started`	TRUE when CtrlNamedBackground start is issued. You may clear or set to desired value.
`uint32 nextRunTicks`	Precomputed value for next run but user functions may change this.

You may also specify a user function that takes a user-defined STRUCT as long as the first elements of the structure match the `WMBackgroundStruct` or, preferably, if the first element is an instance of `WMBackgroundStruct`. Use the started field to determine when to initialize the additional fields. Your structure may not include any String, WAVE, NVAR, DFREF or other fields that reference memory that is not part of the structure itself.

If you specify a user-defined structure that matches the first fields rather than containing an instance of `WMBackgroundStruct`, then your function will fail if, in the future, the size of the built-in structure changes. The value of `MAX_OBJ_NAME` is 31 but this may also change.

Your function should return zero unless it wants to stop in which case it should return 1.

You can call CtrlNamedBackground within your background function. You can even switch to a different function if desired.

Use the status keyword to obtain background task information via the S_info variable, which has the format:

`NAME:`*`name`*`;PROC:`*`fname`*`;RUN:`*`r`*`;PERIOD:`*`p`*`;NEXT:`*`n`*`;QUIT:`*`q`*`;FUNCERR:`*`e`*`;`

When parsing S_info, do not rely on the number of key-value pairs or their order. RUN, QUIT, and FUNCERR values are 1 or 0, NEXT is the tick count for the next firing of the task. QUIT is set to 1 when your function returns a nonzero value and FUNCERR is set to 1 if your function could not be used for some reason.

See Also

See **Background Tasks** on page IV-279 for examples.

Also see the Background Task Demo experiment (choose File→Example Experiments→Programming→Background Task Demo).

CtrlFIFO

`CtrlFIFO` ***`FIFOName`*** `[,` ***`key = value`***`]…`

The CtrlFIFO operation controls various aspects of the named FIFO.

Parameters

close	Closes the FIFO's output or review file (if any).
deltaT=*dt*	Documents the data acquisition rate.
doffset=*dataOffset*	Used only with rdfile. Offset to data. If not provided offset is zero.
dsize=*dataSize*	Used only with rdfile. Size of data in bytes. If not provided, then data size is assumed to be the remainder of file. If this assumption is not valid then unexpected results may be observed.
flush	New data in FIFO is flushed to disk immediately.
file=*oRefNum*	File reference number for the FIFO's output file. You obtain this reference number from the **Open** operation used to create the file.
note=*noteStr*	Stores the note string in the file header. It is limited to 255 characters.
rdfile=*rRefNum*	Like rfile but for review of raw data (use Open/R command). Channel data must match raw data in file. Offset from start of file to start of data can be provided using doffset given in same command. If data does not extend all the way to the end of the file, then the number of bytes of data can be provided using dsize in the same command.
rfile=*rRefNum*	File reference number for the FIFO's review file. Use a review file when you are using a FIFO to review existing data. Obtain the reference number from the Open/R operation used to open the file. File may be either unified header/data or a split format where the header contains the name of a file containing the raw data.
size=*s*	Sets number of chunks in the FIFO. The default is 10000. A chunk of data consists of a single data point from each of the FIFO's channels.
start	Starts the FIFO running by setting the time/date in the FIFO header, writing the header to the output file and marking the FIFO active.
stop	Stops the FIFO by flushing data to disk and marking the FIFO as inactive.
swap	Used only with rdfile. Indicates that the raw data file requires byte-swapping when it is read. This would be the case if you are running on a Macintosh, reading a binary file from a PC, or vice versa.

Details

Once start has been issued, the FIFO can accept no further commands except stop.

The FIFO must be in the valid state for you to access its data (using a chart control or using the **FIFO2Wave** operation). When you create a FIFO, using **NewFIFO**, it is initially invalid. It becomes valid when you issue the start command via the CtrlFIFO operation. It remains valid until you change a FIFO parameter using CtrlFIFO.

FIFOs are used for data acquisition.

See Also
The **NewFIFO** and **FIFO2Wave** operations, and **FIFOs and Charts** on page IV-276.

Cursor

```
Cursor [flags] cursorName traceName x_value
Cursor /F[flags] cursorName traceName x_value, y_value
Cursor /K[/W=graphName] cursorName
Cursor /I[/F][flags] cursorName imageName x_value, y_value
Cursor /M[flags] cursorName
```

The Cursor operation moves the cursor specified by *cursorName* onto the named trace at the point whose X value is *x_value*. or the coordinates of an image pixel or free cursor position at *x_value* and *y_value*.

Parameters
cursorName is one of ten cursors A through J.

Flags

/A=*a* — Activates (*a*=1) or deactivates (*a*=0) the cursor. Active cursors move with arrow keys or the cursor panel.

/C=(*r*,*g*,*b*) — Sets the cursor color (default is black). *r*, *g*, and *b* specify the amount of red, green, and blue in the color of the waves as an integer from 0 to 65535.

/F — Cursor roams free. The trace or image provides the axis pair that defines x and y coordinates for the setting and readout. Use /P to set in relative coordinates, where 0,0 is the top left corner of the rectangle defined by the axes and 1,1 is the right bottom corner.

/H=*h* — Specifies crosshairs on cursors.

- *h*=0 Full crosshairs off.
- *h*=1 Full crosshairs on.
- *h*=2 Vertical hairline.
- *h*=3 Horizontal hairline.

/I — Places cursor on specified image.

/K — Removes the named cursor from the top graph.

/L=*lStyle* — Line style for crosshairs (full or small).

- *lStyle*=0: Solid lines.
- *lStyle*=1: Alternating color dash.

/M — Modifies properties without having to specify trace or image coordinates. Does not work with the /F or /I flags.

/P — Interpret *xNum* as a point number rather than an X value.

/S=*s* — Sets cursor style.

- *s*=0: Original square or circle.
- *s*=1: Small crosshair with letter.
- *s*=2: Small crosshair without letter.

/W=*graphName* — Specifies a particular named graph window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *graphName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details
Usually *traceName* is the same as the name of the wave displayed by that trace, but it could be a name in instance notation. See **ModifyGraph (traces)** and **Instance Notation** on page IV-16 for discussions of trace names and instance notation.

A string containing *traceName* can be used with the $ operator to specify the trace name.

x_value is an X value in terms of the X scaling of the wave displayed by *traceName*. If *traceName* is graphed as an XY pair, then *x_value* is *not* the same as the X axis coordinate. Since the X scaling is ignored when displaying an XY pair in a graph, we recommend you use the /P flag and use a point number for *x_value*.

cursorName is a name, *not* a string.

To get a cursor readout, choose ShowInfo from the Graph menu.

Moving a cursor in a macro or function does not immediately erase the old cursor. DoUpdate has to be explicitly called.

Examples

```
Display myWave                          // X coordinates from X scaling of myWave
Cursor A, myWave, leftx(myWave)         //cursor A on first point of myWave

AppendToGraph yWave vs xWave            //X coordinates from xWave, not X scaling
Cursor/P B,yWave,numpnts(yWave)-1       //cursor B on last point of yWave
DoUpdate                                // erase any old A or B cursors
```

See Also

The **DoUpdate** and **ShowInfo** operations and the **CsrInfo**, **CsrWave**, **CsrWaveRef**, **CsrXWave**, **CsrXWaveRef**, **hcsr**, **pcsr**, **qcsr**, **vcsr**, **xcsr**, and **zcsr** functions. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

CursorStyle

CursorStyle

CursorStyle is a procedure subtype keyword that puts the name of the procedure in the "Style function" submenu of the Cursor Info pop-up menu. It is automatically used when Igor creates a cursor style function. To create a cursor style function, choose "Save style function" in the "Style function" submenu of the Cursor Info pop-up menu.

CurveFit

CurveFit [*flags*] *fitType*, [**kwCWave=***coefWaveName*,] *waveName* [*flag parameters*]

The CurveFit operation fits one of several built-in functions to your data (for user-defined fits, see the **FuncFit** operation). When with CurveFit and built-in fit functions, automatic initial guesses will provide a good starting point in most cases.

The results of the fit are returned in a wave, by default W_coef. In addition, the results are put into the system variables K0, K1 … K*n* but the use of the system variables is limited and considered obsolete

You can specify your own wave for the coefficient wave instead of W_coef using the kwCWave keyword.

Virtually all waves specified to the CurveFit operation can be a sub-range of a larger wave using the same sub-range syntax as the Display operation uses for graphing. See **Wave Subrange Details** on page V-91.

See Chapter III-8, **Curve Fitting** for detailed information including the use of the Curve Fit dialog.

CurveFit operation parameters are grouped in the following categories: flags, fit type, parameters (kwCWave=*coefWaveName* and *waveName*), and flag parameters. The sections below correspond to these categories. Note that flags must precede the fit type and flag parameters must follow *waveName*.

Flags

/B=*pointsPerCycle*	Used when *type* is sin; *pointsPerCycle* is the estimated number of data points per sine wave cycle. This helps provide initial guesses for the fit. You may need to try a few different values on either side of your estimated points/cycle.
/C	Makes constraint matrix and vector. This only applies if you use the /C=*constraintSpec* parameter to specify constraints (see below). Creates the M_FitConstraint matrix and the W_FitConstraint vector. For more information, see **Fitting with Constraints** on page III-197.
/G	Use values in variables K0, K1 … K*n* as starting guesses for a fit. If you specify a coefficient wave with the kwCWave keyword, the starting guesses will be read from the coefficient wave.
/H="*hhh…*"	Specifies coefficients to hold constant. *h* is 1 for coefficients to hold, 0 for coefficients vary. For example, `/H="100"` holds K0 constant, varies K1 and K2.
/K={*constants*}	Sets values of constants (not fit coefficients) in certain fitting functions. For instance, the exp_XOffset function contains an X offset constant. Built-in functions will set the constant automatically, but the automatic value can be overridden using this flag.

constants is a list of constant values, e.g., /K={1,2,3}. The length of the list must match the number of constants used by the chosen fit function.

This flag is not currently supported by the Curve Fit dialog. Use the To Cmd button and add the flag on the command line.

/L=*destLen* Sets the length of the wave created by the AutoTrace feature, that is, /D without destination wave (see the /D parameter above). The length of the wave fit_*waveName* will be set to *destLen*. This keyword also sets the lengths of waves created for confidence and prediction bands.

/M Generates the covariance matrix, the waves CM_K*n*, where *n* is from 0 (for K0) to the number of coefficients minus one.

/M=*doMat* Generates the covariance matrix. If *doMat* =2, the covariance matrix is put into a 2D matrix wave called M_Covar. If *doMat* =1 or is missing, the covariance matrix is generated as the 1D waves CM_K*n*, where *n* is from 0 (for K0) to the number of coefficients minus one. If *doMat* =0, the covariance matrix is not generated. *doMat* =1 is included for compatibility with previous versions; it is better to use *doMat* =2.

/N[=*dontUpdate*] If *dontUpdate* = 1, suppresses updates during the fit. This can make the curve fit go much faster; all graphs, tables, etc. will be updated when the fit finishes. /N is the same as /N=1.

/NTHR = *nthreads* Uses multithreaded code in *nthreads* threads for built-in and standard user-defined fitting functions. Values for *nthreads* are:

0: Selects Auto mode, which uses a number of threads equal to the number of processors in your computer (see **ThreadProcessorCount**).

1: Uses one thread, that is, it is not multithreaded. Overhead for multithreaded code makes it pretty much useless for built-in functions. For user-defined functions you can get almost double the speed on a two-processor computer; the benefit will depend on how fast your fit function is, and how many data points you are fitting.

/O Generates only initial guesses; doesn't actually do the fit.

/ODR=*fitMethod* Selects a fitting method. Values for *fitMethod* are:

0: Default Levenberg-Marquardt least-squares method using old code.

1: Trust-region Levenberg-Marquardt ordinary least-squares method implemented using ODRPACK95 code. See **Curve Fitting References** on page III-231.

2: Trust-region Levenberg-Marquardt least orthogonal distance method implemented using ODRPACK95 code. This method is appropriate for fitting when there are measurement errors in the independent variables, sometimes called "errors in variables fitting", "random regressor models," or "measurement error models".

3: Implicit fit. No dependent variable is specified; instead fitting attempts to adjust the fit coefficients such that the fit function returns zero for all dependent variables.

Implicit fitting will be of almost no use with the built-in fitting functions. Instead, use **FuncFit** and a user-defined fit function designed for an implicit fit.

/Q[=*quiet*] If quiet = 1, prevents results from being printed in history. /Q is the same as /Q=1.

/TBOX = *textboxSpec* Adds an annotation to the graph containing the fit data (see the **TextBox** operation, or Chapter III-2, **Annotations**). The textbox contains a customizable set of information about the fit. The argument *textboxSpec* is a bitfield to select various elements to be included in the textbox:

textboxSpec	**Selects This Element**
1	Title "Curve Fit Results"
2	Date
4	Time

textboxSpec	Selects This Element
8	Fit Type (Least Squares, ODR, etc.)
16	Fit function name
32	Model Wave, the autodestination wave (includes a symbol for the trace if appropriate)
64	Y Wave, with trace symbol
128	X Wave
256	Coefficient value report
512	Include errors in the coefficient value report

Request inclusion of various parts by adding up the values for each part you want.

Setting *textboxSpec* to zero will remove the textbox. Default is *textboxSpec* = 0.

/X Sets the X scaling of the auto-trace destination wave to match the appropriate X axis on the graph when the Y data wave is on the top graph. This is useful when you want to extrapolate the curve outside the range of X data being fit.

/W=*wait* Specifies behavior for the curve fit results window.

wait=1 Wait till user clicks OK button before dismissing curve fit results window. This is the default behavior from the command line or dialog.

wait =0 Do not wait. This is the default behavior from a procedure.

wait =2 Do not display the curve fit results window at all. Use this when you are doing many curve fits in a loop. Requires Igor Pro 6.21 or later.

Fit Types

fitType is one of the built-in curve fit function types:

gauss Gaussian peak: $y = K_0 + K_1 \exp\left[-\left(\frac{x-K_2}{K_3}\right)^2\right]$.

lor Lorentzian peak: $y = K_0 + \frac{K_1}{(x-K_2)^2 + K_3}$.

exp Exponential: $y = K_0 + K_1 \exp(-K_2 x)$.

dblexp Double exponential: $y = K_0 + K_1 \exp(-K_2 x) + K_3 \exp(-K_4 x)$.

sin Sinusoid: $y = K_0 + K_1 \sin(K_2 x + K_3)$.

line Line: $y = K_0 + K_1 x$.

poly *n* Polynomial: $y = k_0 + K_1 x + K_2 x^2 + \ldots$.

n is from 3 to 20. *n* is the number of terms or the degree plus one.

poly_XOffset n Polynomial: y = K0+K1*(x-x_0)+K2*(x-x_0)^2+...

n is from 3 to 20. *n* is the number of terms or the degree plus one.

x_0 is a constant; by default it is set to the minimum X value involved in the fit. Inclusion of x_0 prevents problems with floating-point roundoff errors when you have large values of X in your data set.

hillequation Hill's Equation: $y = K_0 + \frac{(K_1 - K_0)}{1 + (K_3/x)^{K_2}}$.

This is a sigmoidal function. Note that X values must be greater than 0.

sigmoid $y = K_0 + \frac{K_1}{1 + \exp(K_2 - x/K_3)}$.

power — Power law: $y = K_0 + K_1 x^{K_2}$. Note that X values must be greater than 0.

lognormal — Log normal: $y = K_0 + K_1 \exp\left[-\left(\frac{\ln(x/K_2)}{K_3}\right)^2\right]$. X values must be greater than 0.

gauss2D — Two-dimensional Gaussian:

$$z = K_0 + K_1 \exp\left[\frac{-1}{2(1-K_6^2)}\left(\left(\frac{x-K_2}{K_3}\right)^2 + \left(\frac{y-K_4}{K_5}\right)^2 - \frac{2K_6(x-K_2)(y-K_4)}{K_3 K_5}\right)\right].$$

The cross-correlation coefficient (K_6) must be between -1 and 1. This coefficient is automatically constrained to lie in that range. If you are confident that the correlation is zero, it may greatly speed the fit to hold it at zero.

poly2D *n* — Two-dimensional polynomial: $z = K_0 + K_1 x + K_2 y + K_3 x^2 + K_4 xy + K_5 y^2 + \ldots$

where *n* is the degree of the polynomial. All terms up to degree *n* are included, including cross terms. For instance, degree 3 terms are x^3, x^2y, xy^2, and y^3.

exp_XOffset — Exponential: $y = K_0 + K_1 \exp(-(x - x_0)/K_2)$.

x_0 is a constant; by default it is set to the minimum *x* value involved in the fit. Inclusion of x_0 prevents problems with floating-point roundoff errors that can afflict the exp function.

dblexp_XOffset — Double exponential:

$$y = K_0 + K_1 \exp(-(x - x_0)/K_2) + K_3 \exp(-(x - x_0)/K_{4(2)}).$$

x_0 is a constant; by default it is set to the minimum *x* value involved in the fit. Inclusion of x_0 prevents problems with floating-point roundoff errors that can afflict the exp function.

Parameters

kwCWave=*coefWaveName* specifies an optional coefficient wave. If present, the specified coefficient wave is set to the final coefficients determined by the curve fit. If absent, a wave named W_coef is created and is set to the final coefficients determined by the curve fit.

If you use kwCWave=*coefWaveName* and you include the /G flag, initial guesses are taken from the specified coefficient wave.

waveName is the wave containing the Y data to be fit to the selected function *type*. You can fit to a subrange of the wave by supplying (*startX*,*endX*) after the wave name. Though not shown in the syntax description, you can also specify the subrange in points by supplying [*startP*,*endP*] after the wave name. See **Wave Subrange Details** on page V-91 for more information on subranges of waves in curve fitting.

If you are using one of the two-dimensional fit functions (gauss2D or poly2D) either *waveName* must name a matrix wave or you must supply a list of X waves via the /X flag.

Flag Parameters

These flag parameters must follow *waveName*.

/A=*appendResid* — *appendResid* =1 (default) appends the automatically-generated residual to the graph and *appendResid* =0 prevents appending (see /R[=*residwaveName*]). With *appendResid* =0, the wave is generated and filled with residual values, but not appended to the graph.

/AD[=*doAutoDest*] — If *doAutoDest* is 1, it is the same as /D alone. /AD is the same as /AD=1.

/C=*constraintSpec* — Applies linear constraints during curve fitting. Constraints can be in the form of a text wave containing constraint expressions (/C=*textWaveName*) or a suitable matrix and vector (/C={*constraintMatrix*, *constraintVector*}). See **Fitting with Constraints** on page III-197. **Note**: Constraints are not available for the built-in line, poly and poly2D fit functions. To apply constraints to these fit functions you must create a user-defined fit function.

/D [=*destwaveName*] — *destwaveName* is evaluated based on the equation resulting from the fit. *destwaveName* must have the same length as *waveName*.

If only /D is specified, an automatically named wave is created. The name is based on the *waveName* with "fit_" as a prefix. This automatically named wave will be appended (if necessary) to the top graph if *waveName* is graphed there. The X scaling of the fit_ wave is set from the range of x data used during the fit.

By default the length of the automatically-created wave is 200 points (or 2 points for a straight line fit). This can be changed with the /L flag.

If *waveName* is a 1D wave displayed on a logarithmic X axis, Igor also creates an X wave with values exponentially spaced. The name is based on *waveName* with "fitX_" as a prefix.

/F={*confLevel*, *confType* [, *confStyleKey* [, *waveName*…]]}

Calculates confidence intervals for a confidence level of *confLevel*. The value of *confLevel* must be between 0 and 1 corresponding to confidence levels of 0 to 100 per cent.

confType selects what to calculate:

1: Confidence bands for the model.

2: Prediction bands for the model.

4: Confidence intervals for the fit coefficients.

These values can be added together to select multiple options. That is, to select both a confidence band and fit coefficient confidence intervals, set *confType* to 5.

Confidence and prediction bands can be shown as waves contouring a given confidence level (use "Contour" for *confStyleKey*) or as error bars (use "ErrorBar" for *confStyleKey*). The default is Contour.

If no waves are specified, waves to contain the results are automatically generated and appended to the top graph (if the top graph contains the fitted data). See **Confidence Band Details** for details on the waves for confidence bands.

Note: Confidence bands and prediction bands are not available for multivariate curve fits.

/I [=*weightType*]

If *weightType* is 1, the weighting wave (see /W parameter) contains standard deviations. If *weightType* is 0, the weighting wave contains reciprocal of the standard deviation. If the /I parameter is not present, the default is /I=0.

/M=*maskWaveName*

Specifies that you want to use the wave named *maskWaveName* to select points to be fit. The mask wave must match the dependent variable wave in number of points and dimensions. Setting a point in the mask wave to zero or NaN (blank in a table) eliminates that point from the fit.

/R [=*residwaveName*]

Calculates elements of *residwaveName* by subtracting model values from the data values. *residwaveName* must have the same length as *waveName*.

If only /R is specified, an automatically named wave is created with the same number of points as *waveName*. The name is based on *waveName* with "Res_" as a prefix.

The automatically created residual wave will be appended (if necessary) to the top graph if *waveName* is graphed there. The residual wave is appended to a new free axis named by prepending "Res_" to the name of the vertical axis used for plotting *waveName*. To the extent possible, the new free axis is formatted nicely.

If the graph containing the data to be fit has very complex formatting, you may not wish to automatically append the residual to the graph. In this case, use /A=0.

/AR=*doAutoResid*

If *doAutoResid* is 1, it is the same as /R alone. /AR is the same as /AR=1.

/W=*wghtwaveName*

wghtwaveName contains weighting values applied during the fit, and must have the same length as *waveName*. These weighting values can be either the reciprocal of the standard errors, or the standard errors. See the /I parameter above for details.

/X=*xwaveName*

The X values for the data to fit come from *xwaveName*, which must have the same length and type as *waveName*.

If you are fitting to one of the two-dimensional fit functions and *waveName* is a matrix wave, *xwaveName* supplies independent variable data for the X dimension. In this case, *xwaveName* must name a 1D wave with the same number of rows as *waveName*.

/X={*xwave1*, *xwave2*}

For fitting to one of the two-dimensional fit functions when *waveName* is a 1D wave. *xwave1* and *xwave2* must have the same length as *waveName*.

/Y=*ywaveName*

For fitting using one of the 2D fit functions if *waveName* is a matrix wave. *ywaveName* must be a 1D wave with length equal to the number of columns in *waveName*.

/NWOK

Allowed in user-defined functions only. When present, certain waves may be set to null wave references. Passing a null wave reference to CurveFit is normally treated as an error. By using /NWOK, you are telling CurveFit that a null wave

reference is not an error but rather signifies that the corresponding flag should be ignored. This makes it easier to write function code that calls CurveFit with optional waves.

The waves affected are the X wave or waves (/X), weight wave (/W), mask wave (/M) and constraint text wave (/C). The destination wave (/D=wave) and residual wave (/R=wave) are also affected, but the situation is more complicated because of the dual use of /D and /R to mean "do autodestination" and "do autoresidual". See /AR and /AD.

If you don't need the choice, it is better not to include this flag, as it disables useful error messages when a mistake or run-time situation causes a wave to be missing unexpectedly.

Note: To work properly this flag must be the last one in the command.

Flag Parameters for Nonzero /ODR

/XW=*xWeightWave*

/XW={*xWeight1*, *xWeight2*}

/ODR=2 or 3 only.

Specifies weighting values for the independent variables using *xWeightWave*, which must have the same length as *waveName*. When fitting to one of the multivariate fit functions such as poly2D or Gauss2D, you must supply a weight wave for each independent variable using the second form.

Weighting values can be either the reciprocal of the standard errors, or the standard errors. The choice of standard error or reciprocal standard error must be the same for both /W and /XW. See /I for details.

/XHLD=*holdWave*

/XHLD={*holdWave1*, *holdWave2*}

/ODR=2 or 3 only.

Specifies a wave or waves to hold the values of the independent variables fixed during orthogonal distance regression. The waves must match the input X data; a one in a wave element fixes the value of the corresponding X value.

/CMAG=*scaleWave* Specifies a wave that indicates the expected scale of the fit coefficients at the solution. If different coefficients have very different orders of magnitude of expected values, this can improve the efficiency and accuracy of the fit.

/XD=*xDestWave*

/XD={*xDestWave1*, *xDestWave2*}

/ODR=2 or 3 only.

Specifies a wave or waves to receive the fitted values of the independent variables during a least orthogonal distance regression.

/XR=*xResidWave*

/XR={*xResidWave1*, *xResidWave2*}

/ODR=2 or 3 only.

Specifies a wave or waves to receive the differences between fitted values of the independent variables and the starting values during a least orthogonal distance regression. That is, they will be filled with the X residuals.

Details

CurveFit gets initial guesses from the K*n* system variables when user guesses (/G) are specified, unless a coefficient wave is specified using the kwCWave keyword. Final curve fit parameters are written into a wave name W_coef, unless you specify a coefficient wave with the kwCWave keyword.

Other output waves are M_Covar (see the /M flag), M_FitConstraint and W_FitConstraint (see /C parameter and **Fitting with Constraints** on page III-197) and W_sigma.

For compatibility with earlier versions of Igor, the parameters are also stored in the system variables Kn. This can be a source of confusion. We suggest you think of W_coef as the **output** coefficients and Kn as **input** coefficients that get overwritten.

Other output waves are M_Covar (see the /M flag), M_FitConstraint and W_FitConstraint (see /C parameter and **Fitting with Constraints** on page III-197), W_sigma. If you have selected coefficient confidence limits

using the /F parameter, a wave called W_ParamConfidenceInterval is created with the confidence intervals for the fit coefficients.

CurveFit stores other curve fitting statistics in variables whose names begin with "V_". CurveFit also looks for certain V_ variables which you can use to modify its behavior. These are discussed in **Special Variables for Curve Fitting** on page III-202.

When fitting with /ODR=nonzero, fitting with constraints is limited to simple "bound constraints." That is, you can constrain a fit coefficient to be greater than or less than some value. Constraints involving combinations of fit coefficients are supported only with /ODR=0. The constraints are entered in the same way, using an expression like `K0>1`.

Wave Subrange Details

Almost any wave you specify to CurveFit can be a subrange of a wave. The syntax for wave subranges is the same as for the Display command (see **Subrange Display Syntax** on page II-288 for details). However, the Display command allows only one dimension to have a range (multiple elements from the dimension); if a multidimensional wave is appropriate for CurveFit, you may use a range for more than one dimension.

Some waves must have the same number of points as other waves. For instance, a one-dimensional Y wave must have the same number of points as any X waves. Thus, if you use a subrange for an X wave, the number of points in the subrange must match the number of points being used in the Y wave (but see **Subrange Backward Compatibility** on page V-92 for a complication to this rule).

A common use of wave subranges might be to package all your data into a single multicolumn wave, along with the residuals and model values. For a univariate fit, you might need X and Y waves, plus a destination (model) wave and a residual wave. You can achieve all of that using a four-column wave. For example:

```
Make/D/N=(100, 4) Data
... fill column zero with X data and column one with Y data ...
CurveFit poly 3, Data[][1] /X=Data[][0]/D=Data[][2]/R=Data[][3]
```

Note that because all the waves are full columns from a single multicolumn wave, the number of points is guaranteed to be the same.

The number of points used for X waves (*xwaveName* or {*xwave1*, *xwave2*, …}), weighting wave (*wghtwaveName*), mask wave (`maskWaveName`), destination wave (*destwaveName*) and residual wave (*residwaveName*) must be the same as the number of points used for the Y wave (*waveName*). If you specify your own confidence band waves (/F flag) they must match the Y wave; you cannot use subranges with confidence band waves. If you set /ODR = nonzero, the X weight, hold, destination and residuals waves must match the Y wave.

The total number of points in each wave does not need to match other waves, just the number of points in the specified subrange.

When fitting to a univariate fit function (that includes almost all the fit types) the Y wave must have effectively one dimension. That means the Y wave must either be a 1D wave, or it must have a subrange that makes the data being used one dimensional. For instance:

```
Make/N=(100,100) Ydata           // 2D wave
CurveFit gauss Ydata[][0]        // OK- a single column is one-dimensional
CurveFit gauss Ydata[2][]        // OK- s single row is one-dimensional
CurveFit gauss Ydata             // not OK- Ydata is two-dimensional
CurveFit gauss Ydata[][0,1]      // not OK- two columns makes 2D subrange
```

When fitting a multivariate function (**poly2D** or **Gauss2D**) you have the choice of making the Y data either one-dimensional or two-dimensional. If it is one-dimensional, then you must be fitting XYZ (or Y,X1,X2) triplets. In that case, you must provide a one-dimensional Y wave and two one-dimensional X waves, or 2 columns from a multicolumn wave. For instance:

These are OK:

```
Make/N=(100,3) myData
CurveFit Gauss2D myData[][0] /X={myData[][1],myData[][2]}
CurveFit Gauss2D myData[][0] /X=myData[][1,2]
```

These are not OK:

```
CurveFit Gauss2D myData /X={myData[][1],myData[][2]}// 2D Y wave with 1D X waves
CurveFit Gauss2D myData[][0] /X=myData            // too many X columns
```

If you use a 2D Y wave, the X1 and X2 data can come from the grid positions and the Y wave's X and Y index scaling, or you can use one-dimensional waves or wave subranges to specify the X1 and X2 positions of the grid:

```
Make/N=(20,30) yData
CurveFit Gauss2D yData        //OK- 2D Y data, X1 and X2 from scaling
Make/N=20 x1Data
Make/N=30 x2Data
// OK: 2D effective Y data, matching 1D X and Y flags
CurveFit Gauss2D yData[0,9][0,19] /X=x1Data[0,9]/Y=x2data[10,29]
// OK: effective 2D Y data
Make/N=(10,20,3) Y data
CurveFit Gauss2D yData[][][0]
```

There are, of course, lots of possible combinations, too numerous to enumerate.

Subrange Backward Compatibility

Historically, a Y wave could have a subrange. The same subrange applied to all other waves. For backward compatibility, if you use a subrange with the Y wave only, and other waves lack a subrange, these other waves must have either: 1) The same total number of points as the total number of points in the Y wave in which case the Y wave subrange will be applied; or 2) The same total number of points as the Y wave's subrange.

In addition, the Y wave can take a subrange in parentheses to indicate that the subrange refers to the Y wave's scaled indices (X scaling). If you use parentheses to specify an X range, you must satisfy the old subrange rules: All waves must have the same number of points. Subrange is allowed for the Y wave only. The Y wave subrange is applied to all other waves.

Confidence Band Details

Automatic generation of confidence and prediction bands occurs if the /F={…} parameter is used with no wave names. One to four waves are generated, or you can specify one to four wave names yourself depending on the *confKind* and *confStyle* settings.

Waves auto-generated by /F={*confLevel*, *confKind*, *confStyle*}:

confKind	*confStyle*	What You Get	Auto Wave Names
1	"Contour"	upper and lower confidence contours	UC_*dataName*, LC_*dataName*
2	"Contour"	upper and lower prediction contours	UP_*dataName*, LP_*dataName*
3	"Contour"	upper and lower confidence contours and prediction contours	UC_*dataName*, LC_*dataName*, UP_*dataName*, LP_*dataName*
1	"ErrorBar"	confidence interval wave	CI_*dataName*
2	"ErrorBar"	prediction interval wave	PI_*dataName*
3	"ErrorBar"	confidence and prediction interval waves	CI_*dataName*, PI_*dataName*

Note that *confKind* may have 4 added to it if you want coefficient confidence limits calculated as well.

The contour waves are appended to the top graph as traces if the data wave is displayed in the top graph. The wave names have *dataName* replaced with the name of the wave containing the Y data for the fit.

Waves you must supply for /F={*confLevel*, *confKind*, *confStyle*, *wave*, *wave*…}:

confKind	*confStyle*	You Supply
1	"Contour"	2 waves to receive upper and lower confidence contours.
2	"Contour"	2 waves to receive upper and lower prediction contours.
3	"Contour"	4 waves to receive upper and lower confidence and upper and lower prediction contours.
1	"ErrorBar"	1 wave to receive values of confidence band width.
2	"ErrorBar"	1 wave to receive values of prediction band width.
3	"ErrorBar"	2 waves to receive values of confidence and prediction band widths.

The waves you supply must have the same number of points as the dependent variable data wave. The band intervals will be calculated at the X values of the input data. These waves are not automatically appended to a graph; it is expected that you will display the contour waves as traces or use the error bar waves to make error bars on the model fit wave.

Residual Details

Residuals are calculated only for elements corresponding to elements of waveName that are included in the fit. Thus, you can calculate residuals automatically for a piecewise fit done in several steps.

The automatic residual wave will be appended to the top graph if the graph displays the Y data. It is appended to a new free axis positioned directly above the axis used to display the Y data, making a stacked graph. Other axes are shortened as necessary to make room for the new axis. You can alter the axis formatting later. See **Creating Stacked Plots** on page II-293 for details.

While Igor will go to some lengths to make a nicely formatted stacked graph, the changes made to the graph formatting may be undesirable in certain cases. Use /A=0 to suppress the automatic append to the graph. The automatic residual wave will be created and filled with residual values, but not appended to the graph.

See Also

Inputs and Outputs for Built-In Fits on page III-186 and **Special Variables for Curve Fitting** on page III-202 as well as **Accessing Variables Used by Igor Operations** on page IV-103.

When fitting to a user-specified function, see **FuncFit**. For multivariate user-specified fitting functions, see **FuncFit** and **FuncFitMD**. See **Confidence Bands and Coefficient Confidence Intervals** on page III-194 for a detailed description of confidence and prediction bands.

References

An explanation of the Levenberg-Marquardt nonlinear least squares optimization can be found in Chapter 14.4 of Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

CustomControl

`CustomControl [/Z] ctrlName [keyword = value [, keyword = value …]]`

The CustomControl operation creates or modifies a custom control in the target window. A CustomControl starts out as a generic button, but you can customize both its appearance and its action.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the CustomControl to be created or changed. See **Button** for standard default parameters.

The following keyword=value parameters are supported:

fColor=(*r*,*g*,*b*) Sets color of the button only when picture is not used and frame=1.

r, *g*, and *b* can range from 0 to 65535.

frame= f Sets frame style used only when picture is not used:

- f =0: No frame (only the title is drawn).
- f =1: Default, a button is drawn with a centered title. Set fColor to something other than black to colorize the button.
- f =2: Simple box.
- f =3: 3D sunken frame. On Macintosh, when "native GUI appearance" is enabled for the control, the frame is filled with the proper operating system color.
- f =4: 3D raised frame.
- f =5: Text well.

labelBack=(*r*,*g*,*b*) or 0

Sets background color for the control only when a picture is not used and frame is not 1 and is not 3 on Macintosh.

r, *g* and *b* specify the amount of red, green and blue in the color as an integer from 0 to 65535. If not set (or labelBack=0), then background is transparent (not erased).

mode=*m* Notifies the control that something has happened. Can be used for any purpose. See **Details** discussion of the `kCCE_mode` event.

noproc Specifies that no procedure will execute when clicking the custom control.

picture= *pict* Uses the named Proc Pictures to draw the control. The picture is taken to be three side-by-side frames, which show the control appearance in the normal state, when the mouse is down, and in the disabled state.

The control action function can overwrite the picture number using the picture={*pict*,*n*} syntax.

The picture size overrides the size keyword.

picture={*pict*,*n*} Uses the specified Proc Picture to draw the control. The picture is *n* side-by-side frames instead of the default three frames.

pos={*left*,*top*} Sets the postion of the control in pixels.

pos+={*dx*,*dy*} Offsets the position of the control in pixels.

proc=*procName* Specifies the name of the action function for the control. The function must not kill the control or the window.

size={*width*,*height*} Sets size of the control in pixels but only when not using a Proc Picture.

title=*titleStr* Specifies text that appears in the control.

titleStr can contain formatting escape codes in order to create fancy, styled results. The escape codes are the same as used by the **TextBox** operation. The easiest way to generate fancy text is to create a dummy TextBox, set up the text as desired, click the To Cmd Line button, and then edit the TextBox command for use with the control.

userdata(*UDName*)=*UDStr*
Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*
Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

value=*varName* Sets the numeric variable, string variable, or wave that is associated with the control. With a wave, specify a point using the standard bracket notation with either a point number (`value=awave[4]`) or a row label (`value=awave[%alabel]`).

valueColor=(*r*,*g*,*b*) Sets initial color of the title for the button drawn only when picture is not used and frame=1.

r, *g*, and *b* range from 0 to 65535. *valueColor* defaults to black (0,0,0). To further change the color of the title text, use escape sequences as described for title=*titleStr*.

Flags

/Z No error reporting.

Details

When you create a custom control, your action procedure will need to get information about the state of the control using the `WMCustomControlAction` structure, which is a predefined structure passed to your function. All of the various members of the `WMCustomControlAction` structure are as described in the following tables:

Base `WMCustomControlAction` Structure Members

Member	Description
`char ctrlName[MAX_OBJ_NAME+1]`	Control name.
`char win[MAX_WIN_PATH+1]`	Host (sub)window.
`STRUCT Rect winRect`	Local coordinates of host window.
`STRUCT Rect ctrlRect`	Enclosing rectangle of the control.
`STRUCT Point mouseLoc`	Mouse location.
`Int32 eventCode`	Event that caused the procedure to execute.
`Int32 eventMod`	Bitfield of modifiers. See **Control Structure eventMod Field** on page III-385.
`String userData`	Primary (unnamed) user data. When this is changed, it is automatically written back.
`Int32 blockReentry`	Prevents reentry of control action procedure. See **Control Structure blockReentry Field** on page III-386.

Base WMCustomControlAction Structure Members

Member	Description
Int32 missedEvents	TRUE when events occurred but the user function was not available for action.
Int32 mode	General purpose.
Int32 curFrame	Input and output, used with kCCE_frame event.
Int32 needAction	Action meaning depends on the event: Events kCCE_mousemoved, kCCE_enter, and kCCE_leave set to TRUE to force redraw, which is normally not done for these events. Event kCCE_tab and kCCE_mousedown set to TRUE to request keyboard focus (and get kCCE_char events). Event kCCE_idle set to TRUE to request redraw.

Members of WMCustomControlAction Structure with value=*varName*

Member	Description
Int32 isVariable	TRUE if *varName* is a numeric variable or a string variable.
Int32 isWave	TRUE if *varName* referenced a wave.
Int32 isString	TRUE if *varName* is a String type.
NVAR nVal	If isVariable and not isString.
SVAR sVal	If isVariable and isString.
WAVE nWave	If isWave and not isString.
WAVE/T sWave	If isWave and not isString.
Int32 rowIndex	If isWave, this is the row index if rowLabel is empty.
char rowLabel[MAX_OBJ_NAME+1]	Wave row label.

Members of WMCustomControlAction Structure with kCCE_char

Member	Description
Int32 kbChar	Keyboard key character code.
Int32 kbMods	Keyboard key modifiers bit field: bit 0: Command (*Macintosh*). bit 1: Shift. bit 2: Alpha Lock. bit 3: Option (*Macintosh*) or Alt (*Windows*). bit 4: Control.

When determining the state of the eventCode member in the WMCustomControlAction structure, the various code values you use are defined in this section.

When determining the state of the eventCode member in the WMCustomControlAction structure, the various values you use are specified below. You can define them as static constants in your procedure file or, in Igor Pro 6.20 or later, define them by adding this include statement to your procedure file:

```
#include <CustomControl Definitions>
```

Event Code	Description
kCCE_mousedown = 1	Mouse down in control.
kCCE_mouseup = 2	Mouse up in control.
kCCE_mouseup_out = 3	Mouse up outside control.
kCCE_mousemoved = 4	Mouse moved (happens only when mouse is over the control).
kCCE_enter = 5	Mouse entered control.
kCCE_leave = 6	Mouse left control.
kCCE_draw = 10	Time to draw custom content.
kCCE_mode = 11	Sent when executing CustomControl *name*, mode=*m*.
kCCE_frame = 12	Sent before drawing a subframe of a custom picture.
kCCE_dispose = 13	Sent as the control is killed.
kCCE_modernize = 14	Sent when dependency (variable or wave set by value=*varName* parameter) fires. It will also get draw events, which probably don't need a response.
kCCE_tab = 15	Sent when user tabs into the control. If you want keystrokes (kCCE_char), then set needAction.
kCCE_char = 16	Sent on keyboard events. Stores the keyboard character in kbChar and modifiers bit field is stored in kbMods. Sets needAction if key event was used and requires a redraw.
kCCE_drawOSBM = 17	Called after drawing *pict* from picture parameter into an offscreen bitmap. You can draw custom content here.
kCCE_idle = 18	Idle event typically used to blink insertion points etc. Set needAction to force the control to redraw. Sent only when the host window is topmost.

When you call a function with the kCCE_draw event, the basic button picture (custom or default) will already have been drawn. You can use standard draw commands such as **DrawLine** to draw on top of the basic picture. Unlike the normal situation when draw commands merely add to a draw list, which only later is drawn, kCCE_draw event draw commands are executed directly. The coordinate system, which you can not change, is pixels with (0,0) being the top left corner of the control. Most drawing commands are legal but because of the immediate nature of drawing, the /A (append) flag of **DrawPoly** is not allowed.

The kCCE_mode event can be used for any purpose, but it mainly serves as a notification to the control that something has happened. For example, to send information to a control, you can set a named (or the unnamed) userdata and then set the mode to indicate that the control should examine the userdata. For this signaling purpose, you should use a mode value of 0 because this value will not become part of the recreation macro.

The kCCE_frame event is sent just before drawing one of the *pict* frames, as set by the picture parameter. On input, the curFrame field is set to 0 (normal or mouse down outside button), to 1 (mouse down in button), or to 2 (disable). You may modify curFrame as desired but your value will be clipped to a valid value.

When you specify a *pict* with the picture parameter, you will get a kCCE_drawOSBM event when that *pict* is drawn into an offscreen bitmap. Once it is created, all updates use the offscreen bitmap until you specify a new picture parameter. Thus the custom drawing done at this event is static, unlike drawing done during the kCCE_draw event, which can be different each time the control is drawn. Because the *pict* can be contain multiple side-by-side frames, the width of the offscreen bitmap is the width derived from the ctrlRect field multiplied by the number of frames.

Because the action function is called in the middle of various control events, it must not kill the control or the window. Doing so will almost certainly cause a crash.

Examples

See **CustomControl** on page III-371 for some examples of custom controls.

For a demonstration of custom controls, see the Custom Control Demo.pxp example experiment, which is located in your Igor Pro Folder in the Examples:Testing & Misc: folder.

See Also

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Proc Pictures on page IV-43.

The **TextBox**, **DrawPoly** and **DefaultGUIControls** operations.

CWT

CWT [***flags***] ***srcWave***

The CWT operation computes the continuous wavelet transform (CWT) of a 1D real-valued input wave (*srcWave*). The input can be of any numeric type. The computed CWT is stored in the wave M_CWT in the current data folder. M_CWT is a double precision 2D wave which, depending on your choice of mother wavelet and output format, may also be complex. The dimensionality of M_CWT is determined by the specifications of offsets and scales. The operation sets the variable V_flag to zero if successful or to a nonzero number if it fails for any reason.

Flags

/ENDM=*method* Selects the method used to handle the two ends of the data array with direct integration (/M=1).

method=0: Padded on both sides by zeros.

method=1: Reflected at both the start and end.

method=2: Entered with cyclical repetition.

/FSCL Use correction factor to the wave scaling of the second dimension of the output wave so that the numbers are more closely related to Fourier wavelength. See **References** for more information on the calculation of these correction factors. This flag does not affect the output from the Haar wavelet.

/M=*method* Specifies the CWT computation method.

method=0: Fast method uses FFT (default).

method=1: Slower method using direct integration.

You should mostly use the more efficient FFT method. The direct method should be reserved to situations where the FFT is not producing optimal results. Theoretically, when the FFT method fails, the direct method should also be fairly inaccurate, e.g., in the case of undersampled signal. The main advantage in the direct method is that you can use it to investigate edge effects.

/OUT=*format* Sets the format of the output wave M_CWT:

format=1: Complex.

format=2: Real valued.

format=4: Real and contains the magnitude.

Depending on the method of calculation and the choice of mother wavelet, the "native" output of the transform may be real or complex. You can force the output to have a desired format using this flag.

/Q quiet mode; no results printed to the history.

/R1={*startOffset*, *delta1*, *numOffsets*}

Specifies offsets for the CWT. Offsets are the first dimension in a CWT. Normally you will calculate the CWT for the full range of offsets implied by *srcWave* so you will not need to use this flag. However, when using the slow method, this flag restricts the output range of offsets and save some computation time. *startOffset* (integer) is the point number of the first offset in *srcWave*. *delta1* is the interval between two consecutive CWT offsets. It is expressed in terms of the number *srcWave* points. *numOffsets* is the number of offsets for which the CWT is computed.

By default *startOffset*=0, *delta1*=1, and *numOffsets* is the number of points in *srcWave*. If you want to specify just the *startOffset* and *delta1*, you can set *numOffsets*=0 to use the same number of points as the source wave.

/R2={*startScale*, *scaleStepSize*, *numScales*}

Specifies the range of scales for the CWT is computation. Scales are the second dimension in the output wave. Note however that there are limitations on the minimum and maximum scales having to do with the sampling of your data. Because there is a rough correspondence between a Fourier spatial frequency and CWT scale it should be understood that there is also a maximum theoretical scale. This is obvious if you compute the CWT using an FFT but it also applies to the slow method. If you specify a range outside the allowed limits, the corresponding CWT values are set to NaN.

Use NaN if you want to use the default value for any parameter.

The default value for *startScale* is determined by sampling of the source wave and the wavelet parameter or order.

At a minimum you must specify either *scaleStepSize* or *numScales*.

/SMP1=*offsetMode* Determines computation of consecutive offsets. Currently supporting only *offsetMode*=1 for linear, user-provided partial offset limits (see /R1 flag): `val1=startOffset+numOffsets*delta1`.

/SMP2=*scaleMode* Determines computation of consecutive scales. *scaleMode* is 1 by default if you specify the /R2 flag.

scaleMode=1: Linear:

```
theScale=startScale+index*scaleStepSize
```

scaleMode=2: User-provided scaling wave.

scaleMode=4: Power of 2 scaling interval:

```
theScale=startScale*2.^(index*scaleStepSize)
```

When using *scaleMode*=4 the operation saves the consecutive scale values in the wave W_CWTScaling. Note also that if you use *scaleMode*=4 without specifying a corresponding /R2 flag, the default *scaleStepSize* of 1 and 64 scale values gives rise to scale values that quickly exceed the allowed limits.

(See /R2 flag for details about the different parameters used in the equations above.)

/SW2=*sWave* Provides specific scale values at which the transform is evaluated. Use instead of /R2 flag. It is your responsibility to make sure that the entries in the wave are appropriate for the sampling density of *srcWave*.

/WBI1={*Wavelet* [, *order*]}

Specifies the built-in wavelet (mother) function. *Wavelet* is the name of a wavelet function: Morlet (default), MorletC (complex), Haar, MexHat, DOG, and Paul.

Morlet: $\Psi_0(x) = \frac{1}{\Pi^{1/4}}\cos(\omega x)e^{-x^2/2}$.

By default, ω=5. Use the /WPR1 flag to specify other values for ω.

MorletC: $\Psi_0(x) = \frac{1}{\Pi^{1/4}}e^{i\omega x}e^{-x^2/2}$.

By default, ω=5. Use the /WPR1 flag to specify other values for ω.

Haar: $\Psi_0(x) = \begin{cases} 1 & 0 \le x < 0.5 \\ -1 & 0.5 \le x < 1 \end{cases}$.

DOG: $\Psi_0(m,x) = \frac{(-1)^{m+1}}{\sqrt{\Gamma\left(m+\frac{1}{2}\right)}}\ \frac{\mathrm{d}^m}{\mathrm{d}x^m}(e^{-x^2/2})$.

MexHat: special case of DOG with *m*=2.

Paul: $\Psi_0(m,x) = \frac{2^m i^m m!}{\sqrt{\pi(2m)!}}\ (1-ix)^{-(m+1)}$.

order applies to DOG and Paul wavelets only and specifies *m*, the particular member of the wavelet family.

The default wavelet is the Morlet.

/WPR1={*param1*} *param1* is a wavelet-specific parameter for the wavelet function selected by /WBI1. For example, use /WPR1={6} to change the Morlet frequency from the default (5).

/Z No error reporting. If an error occurs, sets V_flag to -1 but does not halt function execution.

Details

The CWT can be computed directly from its defining integral or by taking advantage of the fact that the integral represents a convolution which in turn can be calculated efficiently using the fast Fourier transform (FFT).

When using the FFT method one encounters the typical sampling problems and edge effects. Edge effects are also evident when using the slow method but they only significant in high scales.

From sampling considerations it can be shown that the maximum frequency of a discrete input signal is 1/2dt where dt is the time interval between two samples. It follows that the smallest CWT scale is 2dt and the largest scale is Ndt where N is the total number of samples in the input wave.

The transform in M_CWT is saved with the wave scaling. *startOffset* and *delta1* are used for the X-scaling. Both *startOffset* and *delta1* are either specified by the /R1 flag or copied from *srcWave*. The Y-scaling of M_CWT depends on your choice of /SMP2. If the CWT scaling is linear then the wave scaling is based on *startScale* and *scaleStepSize*. If you are using power of 2 scaling interval then the Y wave scaling of M_CWT has a *start*=0 and *delta*=1 and the wave W_CWTScaling contains the actual scale values for each column of M_CWT. Note that W_CWTScaling has one extra data point to make it suitable for display using an operation like:

```
AppendImage M_CWT vs {*, W_CWTScaling}
```

We have encountered two different definitions for the Morlet wavelet in the literature. The first is a complex function (MorletC) and the second is real (Morlet). Instead of choosing one of these definitions we implemented both so you may choose the appropriate wavelet.

See Also

For discrete wavelet transforms use the **DWT** operation. The **WignerTransform** and **FFT** operations.

For further discussion and examples see **Continuous Wavelet Transform** on page III-246.

References

Torrence, C., and G.P. Compo, A Practical Guide to Wavelet Analysis, *Bulletin of the American Meteorological Society*, *79*, 61-78, 1998.

The Torrence and Compo paper is also online at:
<http://paos.colorado.edu/research/wavelets/>.

DataFolderDir

DataFolderDir(*mode* [, *dfr*])

The DataFolderDir function returns a string containing a listing of some or all of the objects contained in the current data folder or in the data folder referenced by *dfr*. The *dfr* parameter was added in Igor Pro 6.20.

Parameters

mode is a bitwise flag for each type of object. Use -1 for all types. Use a sum of the bit values for multiple types.

Desired Type	Bit Number	Bit Value
All		-1
Data folders	0	1
Waves	1	2
Numeric variables	2	4
String variables	3	8

dfr is a data folder reference.

Details

The returned string has the following format:

FOLDERS:*name*,*name*,…;<CR>

WAVES:*name*,*name*,…;<CR>

VARIABLES:*name*,*name*,…;<CR>

STRINGS:*name*,*name*,…;<CR>

Where <CR> represents the carriage return character.

Tip

This function is mostly useful during debugging, used in a **Print** command. For finding the contents of a data folder programmatically, it will be easier to use the functions **CountObjects** and **GetIndexedObjName**.

Examples

```
Print DataFolderDir(8+4)      // prints variables and strings
Print DataFolderDir(-1)       // prints all objects
```

See Also

Chapter II-8, **Data Folders**.

Setting Bit Parameters on page IV-12 for information about bit settings.

DataFolderExists

DataFolderExists(*folderNameStr*)

The DataFolderExists function returns the truth that the specified data folder exists.

You can use a full or partial path to the data folder.

See Also

Chapter II-8, **Data Folders**.

DataFolderRefsEqual

DataFolderRefsEqual(*dfr1, dfr2*)

The DataFolderRefsEqual function returns the truth the two data folder references are the same.

Requires Igor Pro 6.20 or later.

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-61.

The **DataFolderRefStatus** function.

DataFolderRefStatus

DataFolderRefStatus(*dfr*)

The DataFolderRefStatus function returns the status of a data folder reference.

Requires Igor Pro 6.1 or later.

Details

DataFolderRefStatus returns zero if the data folder reference is invalid or non-zero if it is valid.

DataFolderRefStatus returns a bitwise result with bit 0 indicating if the reference is valid and bit 1 indicating if the reference data folder is free. Therefore the returned values are:

0: The data folder reference is invalid.

1: The data folder reference refers to a regular global data folder.

3: The data folder reference refers to a free data folder.

A data folder reference is invalid if it was never assigned a value or if it is assigned an invalid value. For example:

```
DFREF dfr                                   // dfr is invalid
DFREF dfr = root:                           // dfr is valid
DFREF dfr = root:NonExistentDataFolder      // dfr is invalid
```

A data folder reference can be valid and yet point to a non-existent data folder:

```
NewDataFolder/O root:MyDataFolder
DFREF dfr = root:MyDataFolder               // dfr is valid
KillDataFolder root:MyDataFolder            // dfr is still valid
```

After the KillDataFolder, dfr is still a valid data folder reference but points to a non-existent data folder.

You should use DataFolderRefStatus to test any DFREF variables that might not be valid, such as after assigning a reference when you are not sure that the referenced data folder exists. For historical reasons, an invalid DFREF variable will often act like root.

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-61.

dateToJulian

dateToJulian(*year, month, day*)

The dateToJulian function returns the Julian day number for the specified date. The Julian day starts at noon. Use negative number for BC years and positive numbers for AD years. To exclude any ambiguity, there is no year zero in this calendar. For general orientation, Julian day 2450000 corresponds to October 9, 1995.

See Also

The **JulianToDate** function.

For more information about the Julian calendar see: <http://www.tondering.dk/claus/cal/>.

date

date()

The date function returns a string containing the current date.

Formatting of dates depends on your operating system and on your preferences entered in the Date & Time control panel (*Macintosh*) or the Regional Settings control panel (*Windows*).

Examples

```
Print date()        // Prints Mon, Mar 15, 1993
```

See Also

The **Secs2Date**, **Secs2Time**, and **time** functions.

date2secs

date2secs(*year, month, day*)

The date2secs function returns the number of seconds from midnight on 1/1/1904 to the specified date.

Note that the month and day parameters are one-based, so these series start at one.

If *year*, *month*, and *day* are all -1 then date2secs returns the offset in seconds from the local time to the UTC (Universal Time Coordinate) time.

Examples

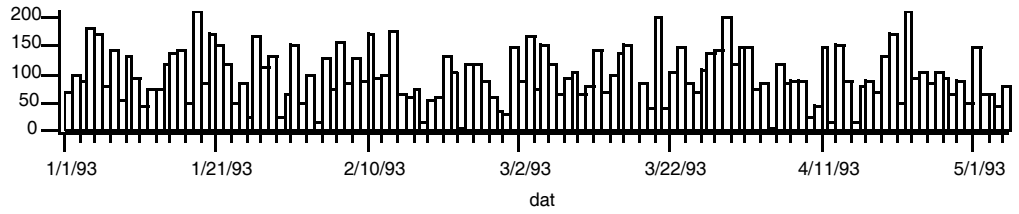
```
Print Secs2Date(date2secs(1993,3,15),1)        // Ides of March, 1993
```

Prints the following, depending on your system's date settings, in the history area:

```
  Monday, March 15, 1993
```

This next example sets the X scaling of a wave to 1 day per point, starting January 1, 1993:

```
Make/N=125 myData = 100 + gnoise(50)
SetScale/P x,date2secs(1993,1,1),24*60*60,"dat",myData
Display myData;ModifyGraph mode=5
```



See Also

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-102.

The **Secs2Date**, **Secs2Time**, and **time** functions.

DateTime

DateTime

The DateTime function returns number of seconds from 1/1/1904 to current date and time.

Unlike most Igor functions, DateTime is used without parentheses.

Examples

```
Variable now = DateTime
```

See Also

The **Secs2Date**, **Secs2Time** and **time** functions.

dawson

dawson(*x*)

The dawson function returns the value of the Dawson integral:

$$F(x) = \exp(-x^2)\int_0^x \exp(t^2)dt.$$

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 298 pp., Dover, New York, 1972.

DDEExecute

DDEExecute(*refNum*, *cmdStr* [, *timeout*])

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDEExecute function sends a string of commands, *cmdStr*, to the server for execution. The format of the commands depends on the server application.

refNum is a DDE session reference number returned by DDEInitiate to start a particular session.

It returns an error code from the server or zero if there was no error.

DDEExecute returns -1 if the server gave an error but the error code was zero. Returns -2 if the server did not reply before the timeout period. Returns -3 if the *refNum* is invalid and -4 for other errors.

The optional *timeout* value is in seconds. The default *timeout* is 60 sec.

In some situations, you may want to use a zero *timeout* and then use DDEStatus to monitor when the server is finished. You would do this if the server might take a long time to accomplish the commands and you want Igor to continue working at the same time.

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDEInitiate

DDEInitiate(*serverName, topicName*)

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDEInitiate function opens a DDE session and returns a reference number for use by the rest of the DDE routines. Returns zero if failure.

If the desired application is not running, DDEInitiate will return zero. If this occurs, you can use the ExecuteScriptText operation to start the server application.

Refer to your application's documentation for the *serverName* and *topicName*. See the following example for usage with Excel.

Examples

```
Variable ch= DDEInitiate("excel","book1")
```

See Also

The **ExecuteScriptText** operation. For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDEPokeString

DDEPokeString(*refNum, itemString, string* [*, timeout*])

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDEPokeString function sends *string* to the server.

itemString is a string specifying the server application's DDE item name for the location into which to store the string. For example, `"R1C1"` specifies the first cell in an Excel spreadsheet.

refNum is the DDE session reference number returned by DDEInitiate to start a particular session.

It returns an error code from the server or zero if there was no error.

DDEPokeString returns -1 if the server gave an error but the error code was zero. Returns -2 if the server did not reply before the timeout period. Returns -3 if the *refNum* is invalid and -4 for other errors.

The optional *timeout* value is in seconds. The default *timeout* is 60 sec.

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDEPokeWave

DDEPokeWave(*refNum, itemString, wave* [*, timeout* [*, format*]])

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDEPokeWave function sends data from a *wave* to the server.

itemString is a string specifying the server application's DDE item name for the location into which to store the string. For example, `"R1C1:R10C10"` specifies a 10x10 block of cells in an Excel spreadsheet.

refNum is a DDE session reference number returned by DDEInitiate to start a particular session.

It returns an error code from the server or zero if there was no error.

DDEPokeWave returns -1 if the server gave an error but the error code was zero. Returns -2 if the server did not reply before the *timeout* period. Returns -3 if the *refNum* is invalid and -4 for other errors.

The optional *format* parameter can be 0 to send the data as tab-delimited text (default) or can be 1 to specify Microsoft's XLTable (excel) format. To specify *format* without specifying *timeout*, the latter may be completely missing (`,,`) or a `*` symbol may be used:

```
err= DDEPokeWave(ch,"R1C1:R10C10",*,1)
```

The optional *timeout* value is in seconds. The default *timeout* is 60 sec.

Examples

A session using Microsoft Excel:

```
Variable ch,err1,err2
ch= DDEInitiate("excel","book1")
```

```
Make/O/N=(5,5) jack= P+10*Q
err1= DDEPokeWave(ch,"R1C1:R5C5",jack)
err2= DDETerminate(ch)
```

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDERequestString

DDERequestString(*refNum, itemString* [*, timeout*])

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDERequestString string function returns a string of requested data or null handle in case of failure.

itemString is a string specifying the server application's DDE item name for the data being requested. For example, `"R1C1"` specifies the first cell in an Excel spreadsheet.

refNum is a DDE session reference number returned by DDEInitiate to start a particular session.

The optional *timeout* value is in seconds. The default *timeout* is 60 sec.

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDERequestWave

DDERequestWave(*refNum, itemString, destWave* [*, timeout*])

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDERequestWave function loads a preexisting wave with data from the server. The provided destination wave, *destWave*, can be either text or numeric. Data from the server must be a tab delimited array. It is analyzed to determine the dimensions of the wave but the numeric type (or string type) of the wave is not changed.

itemString is a string specifying the server application's DDE item name for the requested data. For example, `"R1C1:R10C10"` specifies a 10x10 block of cells in an Excel spreadsheet.

refNum is a DDE session reference number returned by DDEInitiate to start a particular session.

It returns an error code from the server or zero if there was no error.

DDERequestWave returns -1 if the server gave an error but the error code was zero. Returns -2 if the server did not reply before the timeout period. Returns -3 if the *refNum* is invalid and -4 for other errors.

The optional *timeout* value is in seconds. The default *timeout* is 60 sec.

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDEStatus

DDEStatus(*refNum*)

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDEStatus function returns the status of the DDE session defined by *refNum* from a previous DDEInitiate.

Returns zero if *refNum* is not valid or if the session has been closed.

Returns nonzero if session is valid where the individual bits have the following meanings:

Bit 0: Set if the session is valid and not busy.

Bit 1: Set if waiting for an ack from the server for a previous DDEPoke or DDEExecute that timed out. (bit 0 and 1 are exclusive).

Bit 2: Set if the ack from the server for a previous poke or execute command was negative. Only applies to timed out commands.

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDETerminate

DDETerminate(*refNum*)

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDETerminate function closes the DDE session defined by *refNum* from a previous DDEInitiate. Returns truth session was valid.

Pass zero to terminate all client sessions.

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

Debugger

Debugger

The Debugger operation breaks into the debugger if it is enabled.

See Also

The Debugger on page IV-184 and the **DebuggerOptions** operation.

DebuggerOptions

DebuggerOptions [**enable=*en*, debugOnError=*doe*, NVAR_SVAR_WAVE_Checking=*nvwc***]

The DebuggerOptions operation programmatically changes the user-level debugger settings. These are the same three settings that are available in the Procedure menu (and the debugger source pane contextual menu)

Parameters

All parameters are optional. If none are specified, no action is taken, but the output variables are still set.

enable=*en* Turns the debugger on (*en*=1) or off (*en*=0).

If the debugger is disabled then the other settings are cleared even if other settings are on.

debugOnError=*doe* Turns Debugging On Error on or off.

doe=0: Disables Debugging On Error (see **Debugging on Error** on page IV-185).

doe=1: Enables Debugging On Error and also enables the debugger (implies enable=1).

NVAR_SVAR_WAVE_Checking=*nvwc*

Turns NVAR, SVAR, and WAVE checking on or off.

nvwc=0: Disables "NVAR SVAR WAVE Checking". See **Accessing Global Variables and Waves** on page IV-50 for more details.

nvwc=1: Enables this checking and also enables the debugger (implies enable=1).

Details

DebuggerOptions sets the following variables to indicate the Debugger settings that are in effect *after* the command is executed. A value of zero means the setting is off, nonzero means the setting is on.

```
V_enable            V_debugOnError      V_NVAR_SVAR_WAVE_Checking
```

See Also

The Debugger on page IV-184 and the **Debugger** operation.

default

default:

The default flow control keyword is used in switch and strswitch statements. When none of the case labels in the switch or strswitch match the evaluation expression, execution will continue with code following the default label, if it is present.

See Also

Switch Statements on page IV-34.

DefaultFont

DefaultFont [**/U**] "***fontName***"

The DefaultFont operation sets the default font to be used in graphs for axis labels, tick mark labels and annotations, and in page layouts for annotations.

Parameters

"fontName" should be a font name, optionally in quotes. The quotes are not required if the font name is one word.

Flags

/U Updates existing graphs and page layouts immediately to use the new default font.

DefaultGUIControls

DefaultGUIControls [**/Mac/W=*winName*/Win**] [***appearance***]

The DefaultGUIControls operation changes the appearance of user-defined controls.

Note: The recommended way to change the appearance of user-defined controls is to use the Miscellaneous Settings dialog's Native GUI Appearance for Controls checkbox in the Compatibility tab, which is equivalent to `DefaultGUIControls native` when checked, and to `DefaultGUIControls os9` when unchecked.

Use `DefaultGUIControls/W=`*winName* to override that setting for individual windows.

Parameters

appearance may be one of the following:

native Creates standard-looking controls for the current computer platform. This is the default in Igor Pro 6.02 or later.

os9 Igor Pro 5 appearance (quasi-Macintosh OS 9 controls that look the same on Macintosh and Windows).

default Inherits the window appearance from either a parent window or the experiment-wide default (only valid with /W).

Flags

/Mac Changes the appearance of controls only on Macintosh, and it affects the experiment whenever it is used on Macintosh.

/W=*winName* Affects the named window or subwindow. When omitted, sets an experiment-wide default. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/Win Changes the appearance of controls only on Windows, and it affects the experiment whenever it is used on Windows.

Details

If *appearance* is not specified, nothing is changed. The current value for appearance is returned in S_value.

If *appearance* is specified the previous appearance value for the window- or experiment-wide default is returned in S_value.

With /W, the control appearance applies only to the specified window (Graph or Panel). If it is not used, then the settings are global to experiments on the current computer. **Tip**: Use `/W=#` to refer to the current active subwindow.

The /Mac and /Win flags specify the affected computer platform. If the current platform other than specified, then the settings are not used, but (if native or OS9) are remembered for use in window recreation

macros or experiment recreation. This means you can create an experiment that with different appearances depending on the current platform.

If neither /Mac nor /Win are used, it is implied by the current platform. To set native appearance on both platforms, use two commands:

```
DefaultGUIControls/W=Panel0/Mac native
DefaultGUIControls/W=Panel0/Win native
```

Note: The setting for DefaultGUIControls without /W is not stored in the experiment file; it is a user preference set by the Miscellaneous Settings dialog's Native GUI Appearance for Controls checkbox in the Compatibility tab. If you use DefaultGUIControls native or DefaultGUIControls os9 commands, the checkbox will not show the current state of the experiment-wide setting. Clicking Save Settings in the Miscellaneous Settings dialog will overwrite the DefaultGUIControls setting (but not the per-window settings).

In addition to the experiment-wide appearance setting and the window-specific appearance setting, an individual control's appearance can be set with the appropriate control command's appearance keyword (or a ModifyControl appearance keyword). A control-specific appearance setting overrides a window-specific appearance, which in turn overrides the experiment-wide appearance setting.

Although meant to be used before controls are created, calling DefaultGUIControls will update all open windows.

DefaultGUIControls does not change control fonts or font sizes, which means you can create controls that look "native-ish" without having to readjust their positions to avoid avoid shifting or overlap. However, the smooth font rendering that the Native GUI uses on Macintosh does change the length of text slightly, so some shifting will occur that affects mostly controls that were aligned on their right sides.

The native appearance affects the way that controls are drawn in **TabControl** and **GroupBox** controls.

TabControl Background Details

Unlike the os9 appearance which draws only an outline to define the tab region (leaving the center alone) the native tab appearance fills the tab region. Fortunately, TabControls are drawn before all other kinds of controls which allows enclosed controls to be drawn on top of a tab control regardless of the order in which the buttons are defined in the window recreation macro.

However the drawing order of native TabControls does matter: the top-most TabControls draws over other TabControls. (The top-most TabControl is listed last in the window recreation macro.) The os9 appearance allows a smaller (nested) TabControl to be underneath the later (enclosing) TabControl because tabs normally aren't filled. Converting these tabs to native appearance will cause nested tab to be hidden.

To fix the drawing order problem in an existing panel, turn on the drawing tools, select the arrow tool, right-click the enclosing TabControl, and choose Send to Back to correct this situation. If the TabControl itself is inside another TabControl, select that enclosing TabControl and also choose Send to Back, etc.

To fix the window recreation macro or function that created the panel, arrange the enclosing TabControl commands to execute before the commands that create the enclosed TabControls.

A natively-drawn TabControl draws any drawing objects that are entirely enclosed by the tab region so that it behaves the same as an os9 unfilled TabControl with drawing objects inside.

Groupbox Control Background Details

GroupBox controls, unlike TabControls, are not drawn before all other controls, so the drawing order always matters if the GroupBox specifies a background (fill) color and it contains other controls.

You may find that enabling native appearance hides some controls inside the GroupBox. They are probably underneath (before) the GroupBox in the drawing order.

To fix this in an existing panel, turn on the drawing tools, right-click on the GroupBox and choose Send to Back. To fix the window recreation macro or function that created the panel, arrange the GroupBox commands to execute before the commands that create the enclosed controls.

A natively-drawn GroupBox draws any drawing objects that are entirely enclosed by the box; an os9 filled GroupBox does not.

See Also

The **DefaultGUIFont**, **ModifyControl**, **Button**, **GroupBox**, and **TabControl** operations.

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

DefaultGUIFont

DefaultGUIFont [**/W=*winName* /Mac/Win**] ***group* = {*fNameStr*,*fSize*,*fStyle*}** [**,…**]

The DefaultGUIFont operation changes the default font for user-defined controls and other Graphical User Interface elements.

Parameters

fNameStr is the name of a font, *fSize* is the font size, and *fStyle* is a binary coded number with each bit controlling one aspect of the font style. See **Button** for details about these parameters.

group may be one of the following:

all	All controls
button	Button and default CustomControl
checkbox	CheckBox controls
tabcontrol	TabControl controls
popup	Affects the icon (not the title) of a PopupMenu control. The text in the popped state is set by the system and can not be changed. The title of a PopupMenu is affected by the all group but the icon text is not.
panel	Draw text in a panel.
graph	Overlay graphs. Size is used only if `ModifyGraph gfSize= -1`; style is not used.
table	Overlay tables.

Flags

/Mac	Changes control fonts only on Macintosh, and it affects the experiment whenever it is used on Macintosh.
/W=*winName*	Affects the named window or subwindow. When omitted, sets an experiment-wide default. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.
/Win	Changes control fonts only on Windows, and it affects the experiment whenever it is used on Windows.

Details

Although designed to be used before controls are created, calling DefaultGUIFont will update all affected windows with controls. This makes it easy to experiment with fonts. Keep in mind that fonts can cause compatibility problems when moving between machines or platforms.

The /Mac and /Win flags indicate the platform on which the fonts are to be used. If the current platform is not the one specified then the settings are not used but are remembered for use in window recreation macros or experiment recreation. This allows a user to create an experiment that will use different fonts depending on the current platform.

If the /W flag is used then the font settings apply only to the specified window (Graph or Panel.) If the /W flag is not used, then the settings are global to the experiment. Tip: Use `/W=#` to refer to the current active subwindow.

fNameStr may be an empty string (`""`) to clear a group. Setting the font name to `"_IgorSmall"`, `"_IgorMedium"`, or `"_IgorLarge"` will use Igor's own defaults. The standard defaults for controls are the equivalent to setting all to `"_IgorSmall"`, tabcontrol to `"_IgorMedium"`, and button to `"_IgorLarge"`. Use a *fSize* of zero to also get the standard default for size. On Windows, the three default fonts and sizes are all the same.

Although designed to be used before controls are created, calling DefaultGUIFont will update all affected windows with controls. This makes it easy to experiment with fonts. Keep in mind that fonts can cause compatibility problems when moving between machines or platforms.

To read back settings, use `DefaultGUIFont [/W=`*winName*`/Mac/Win/OVR] `*group* to return the current font name in S_name, the size in V_value, and the style in V_flag. With /OVR or if /Mac or /Win is not current, it returns only override values. Otherwise, values include Igor built-in defaults. If S_name is zero length, values are not defined.

Default Fonts and Sizes

The standard defaults for controls is the equivalent to setting all to `"_IgorSmall"`, tabcontrol to `"_IgorMedium"`, and button to `"_IgorLarge"`. Use a *fSize* of zero to also get the standard default for size. On Windows, the three default fonts and sizes are all the same.

	Macintosh		Windows	
Control	**Font**	**Font Size**	**Font**	**Font Size**
Button	Lucida Grande	13	MS Shell Dlg*	12
Checkbox	Geneva	9	MS Shell Dlg	12
GroupBox	Geneva	9	MS Shell Dlg	12
ListBox	Geneva	9	MS Shell Dlg	12
PopupMenu†	Geneva	9	MS Shell Dlg	12
SetVariable	Geneva	9	MS Shell Dlg	12
Slider	Geneva	9	MS Shell Dlg	12
TabControl	Geneva	12	MS Shell Dlg	12
TitleBox	Geneva	9	MS Shell Dlg	12
ValDisplay	Geneva	9	MS Shell Dlg	12

* MS Shell Dlg is a "virtual font name", which usually maps to Tahoma on Windows.

† PopupMenu font for the title is Geneva 9 on Macintosh, for the popup menu itself the font is Lucida Grande 12. On Windows, both fonts are MS Shell Dlg 12.

Examples

```
DefaultGUIFont/Mac all={"Zapf Chancery",12,0},panel={"geneva",12,3}
DefaultGUIFont/Win all={"Century Gothic",12,0},panel={"arial",12,3}
NewPanel
Button b0
DrawText 40,43,"Some text"
```

See Also

The **DefaultGUIControls** operation. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

The example experiment: All Controls Demo.pxp.

defined

defined(*symbol*)

The defined function returns 1 if the symbol is defined 0 if the symbol is not defined.

symbol is a symbol possibly created by a #define statement or by SetIgorOption poundDefine=*symbol*.

symbol is a name, not a string. However you can use $ to convert a string expression to a name.

Details

The defined function can be used in three ways:

Outside of a procedure using a #if statement

Inside a procedure using a #if statement

Inside a procedure using an if statement

For example:

```
#define DEBUG

#if defined(DEBUG)                    // Outside of a function with #if
    Constant kSomeConstant = 100
#else
    Constant kSomeConstant = 50
```

```
#endif

Function Test1()                        // Inside a function with #if
    #if defined(DEBUG)
        Print "Debugging"
    #else
        Print "Not debugging"
    #endif
End

Function Test1()                        // Inside a function with if
    if (defined(DEBUG))
        Print "Debugging"
    else
        Print "Not debugging"
    endif
End
```

In these examples, we could have just as well used #ifdef instead of the defined function. For logical combinations of conditions however, only defined will do:

```
#if (defined(SYMBOL1) && defined(SYMBOL2)
    . . .
#endif
```

When used in a procedure window, defined(symbol) returns 1 if symbol is defined at the time the line is compiled. In a given procedure file, only the following symbols are visible:

Symbols defined earlier in that procedure file *

Symbols defined in the built-in procedure window †

Predefined symbols (see **Predefined Global Symbols** on page IV-87)

Symbols defined by **SetIgorOption** poundDefine=*symbol*

* When used in the body of a procedure, as opposed to outside of a procedure, a symbol defined anywhere in a given procedure window is visible. However, to avoid depending on this confusing exception, you should define all symbols before they are referenced in a procedure file.

† Symbols defined in the built-in procedure window are not available to independent modules.

When the defined function is used from the command line, only symbols defined in the built-in procedure window, predefined symbols, and symbols defined using SetIgorOption are visible.

The defined function was added for Igor 6.20.

See Also

#**define**, **Conditional Compilation** on page IV-86, **Predefined Global Symbols** on page IV-87

DefineGuide

DefineGuide [**/W=** ***winName***] ***newGuideName*** **=** **{**[***guideName1***, ***val*** [**,** ***guideName2***]]**}** [**,**…]

The DefineGuide operation creates or overwrites a user-defined guide line in the target or named window or subwindow. Guide lines help with the positioning of subwindows in a host window.

Parameters

newGuideName is the name for the newly created guide. When it is the name of an existing guide, the guide will be moved to the new position.

guideName1, *guideName2*, etc., must be the names of existing guides.

The meaning of *val* depends on the form of the command syntax. When using only one guide name, *val* is an absolute distance offset from to the guide. The directionality of *val* is to the right or below the guide for positive values. The units of measure are points except in panels where they are in pixels. When using two guide names, *val* is the fractional distance between the two guides.

Flags

/W=winName Defines guides in the named window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

The names for the built-in guides are as defined in the following table:

	Left	Right	Top	Bottom
Host Window Frame	FL	FR	FT	FB
Host Graph Rectangle	GL	GR	GT	GB
Inner Graph Plot Rectangle	PL	PR	PT	PB

The frame guides apply to all window and subwindow types. The graph rectangle and plot rectangle guide types apply only to graph windows and subwindows.

To delete a guide use *guideName*={}.

See Also

The **Display**, **Edit**, **NewPanel**, **NewImage**, and **NewWaterfall** operations.

The **GuideInfo** function.

DelayUpdate

DelayUpdate

The DelayUpdate operation delays the updating of graphs and tables while executing a macro.

Details

Use DelayUpdate at the end of a line in a macro if you want the next line in the macro to run before graphs or tables are updated.

This has no effect in user-defined functions. During execution of a user-defined function, windows update only when you explicitly call the **DoUpdate** operation.

See Also

The **DoUpdate**, **PauseUpdate**, and **ResumeUpdate** operations.

DeleteFile

DeleteFile [*flags*] [*fileNameStr*]

The DeleteFile operation deletes a file on disk.

Parameters

fileNameStr can be a full path to the file to be deleted (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*.

If Igor can not locate the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file to be deleted.

If you use a full or partial path for either file, see **Path Separators** on page III-398 for details on forming the path.

Flags

/I — Interactive mode displays the Open File dialog even if *fileNameStr* is specified and the file exists.

/M=*messageStr* — Specifies the prompt message for the Open File dialog.

/P=*pathName* — Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

/Z[=*z*] — Prevents procedure execution from aborting if it attempts to delete a file that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort.

- /Z=0: Same as no /Z.
- /Z=1: Deletes a file only if it exists. /Z alone has the same effect as /Z=1.
- /Z=2: Deletes a file if it exists or displays a dialog if it does not exist.

Variables

The DeleteFile operation returns information in the following variables:

V_flag Set to zero if the file was deleted, to -1 if the user cancelled the Open File dialog, and to some nonzero value if an error occurred, such as the specified file does not exist.

S_path Stores the full path to the file that was deleted. If an error occurred or if the user cancelled, it is set to an empty string.

See Also

DeleteFolder, **MoveFile**, **CopyFile**, **NewPath**, and **Symbolic Paths** on page II-34.

DeleteFolder

DeleteFolder [***flags***] [***folderNameStr***]

The DeleteFolder operation deletes a disk folder and all of its contents.

Warning: *The DeleteFolder command destroys data!* The deleted folder and the contents are not moved to the Trash or Recycle Bin.

DeleteFolder will delete a folder only if permission is granted by the user. The default behavior is to display a dialog asking for permission. The user can alter this behavior via the Miscellaneous Settings dialog's Misc category. For further details see **Misc Settings** on page III-414.

If permission is denied, the folder will not be deleted and V_Flag will return 1088 (Command is disabled) or 1276 (You denied permission to delete a folder). Command execution will cease unless the /Z flag is specified.

Parameters

folderNameStr can be a full path to the folder to be deleted, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a folder within the folder associated with *pathName*.

If Igor can not determine the location of the folder from *folderNameStr* and *pathName*, it displays a Select Folder dialog allowing you to specify the folder to be deleted.

If /P=*pathName* is given, but *folderNameStr* is not, then the folder associated with *pathName* is deleted.

If you use a full or partial path for either folder, see **Path Separators** on page III-398 for details on forming the path.

Folder paths should not end with single Path Separators. See the **MoveFolder Details** section.

Flags

/I Interactive mode displays a Select Folder dialog even if *folderNameStr* is specified and the folder exists.

/M=*messageStr* Specifies the prompt message for the Select Folder dialog.

/P=*pathName* Specifies the folder to look in for the folder. *pathName* is the name of an existing symbolic path.

/Z[=*z*] Prevents procedure execution from aborting if it attempts to delete a folder that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort.

- /Z=0: Same as no /Z.
- /Z=1: Deletes a folder only if it exists. /Z alone has the same effect as /Z=1.
- /Z=2: Deletes a folder if it exists or displays a dialog if it does not exist.

Variables

The DeleteFolder operation returns information in the following variables:

V_flag Set to zero if the folder was deleted, to -1 if the user cancelled the Select Folder dialog, and to some nonzero value if an error occurred, such as the specified folder does not exist.

S_path Stores the full path to the folder that was deleted, with a trailing semicolon. If an error occurred or if the user cancelled, it is set to an empty string.

Details

You can use only /P=*pathName* (without *folderNameStr*) to specify the source folder to be deleted.

Folder paths should not end with single Path Separators. See the **Details** section for **MoveFolder**.

See Also

The **DeleteFile**, **MoveFolder**, **CopyFolder**, **NewPath**, and **IndexedDir** operations. **Symbolic Paths** on page II-34.

DeletePoints

DeletePoints [**/M=***dim*] ***startElement, numElements, waveName***
[**, *waveName***]…

The DeletePoints operation deletes *numElements* elements from the named waves starting from element *startElement*.

Flags

/M=*dim* *dim* specifies the dimension from which elements are to be deleted. Values are:

0: Rows.

1: Columns.

2: Layers.

3: Chunks.

If /M is omitted, DeletePoints deletes from the rows dimension.

Details

Removing all but one element from the highest dimension of a wave reduces the dimensionality of the wave by one.

A wave may have any number of points, including zero. Removing all elements from any dimension removes all points from the wave, leaving a 1D wave with zero points.

See Also

The **Redimension** operation.

deltax

deltax(*waveName*)

The deltax function returns the named wave's dx value. deltax works with 1D waves only.

Details

This is equal to the difference of the X value of point 1 minus the X value of point 0.

See Also

The **leftx** and **rightx** functions.

When working with multidimensional waves, use the **DimDelta** function.

For an explanation of waves and wave scaling, see **Changing Dimension and Data Scaling** on page II-83.

DFREF

DFREF *localName* [= *path* or *dfr*], [*localName1* [= *path* or *dfr*]]

DFREF is used to define a local data folder reference variable or input parameter in a user-defined function.

Requires Igor Pro 6.1 or later.

The syntax of the DFREF is:

```
DFREF localName [= path or dfr ][, localName1 [= path or dfr ]]...
```

where *dfr* stands for "data folder reference". The optional assignment part is used only in the body of a function, not in a parameter declaration.

Unlike the **WAVE** reference, a DFREF in the body without the assignment part does not do any lookup. It simply creates a variable whose value is null.

Examples

```
Function Test(dfr)
   DFREF dfr

   Variable dfrStatus = DataFolderRefStatus(dfr)
```

```
    if (dfrStatus == 0)
        Print "Invalid data folder reference"
        return -1
    endif

    if (dfrStatus & 2)            // Bit 1 set means free data folder
        Print "Data folder reference refers to a free data folder"
    endif

    if (dfrStatus == 1)
        Print "Data folder reference refers a global data folder"
        DFREF dfSav = GetDataFolderDFR()
        Print GetDataFolder(1)     // Print data folder path
        SetDataFolder dfSav
    endif

    Make/O dfr:jack=sin(x/8)      // Make a wave in the referenced data folder

    return 0
End
```

See Also

For information on programming with data folder references, see **Data Folder References** on page IV-61.

Differentiate

Differentiate [*type* ***flags***][***flags***] ***yWaveA*** [**/X = *xWaveA***]
[**/D = *destWaveA***][, ***yWaveB*** [**/X = *xWaveB***][**/D = *destWaveB***][, …]]

The Differentiate operation calculates the 1D numerical derivative of a wave.

Flags

/DIM=*d* Specifies the wave dimension along which to differentiate when *yWave* is multi-dimensional.

d=-1: Treats entire wave as 1D (default).

For *d*=0, 1, 2, 3, Differentiate operates along rows, columns, layers or chunks.

For example, for a 2D wave, /DIM=0 differentiates each row and /DIM=1 differentiates each column.

/EP=*e* Controls end point handling.

e=0: Replaces undefined points with an approximation (default).

e=1: Deletes the point(s).

/METH=*m* Sets the differentiation method.

m=0: Central difference (default).

m=1: Forward difference.

m=2: Backward difference.

/P Forces point scaling.

Type Flags (*used only in functions***)**

Differentiate also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-58 and **WAVE Reference Type Flags** on page IV-58 for a complete list of type flags and further details.

Wave Parameters

Note: ***All*** wave parameters must follow *yWave* in the command. All wave parameter flags and type flags must appear immediately after the operation name.

/D=*destWave* Specifies the name of the wave to hold the differentiated data. It creates *destWave* if it does not already exist or overwrites it if it exists.

/X=*xWave* Specifies the name of the corresponding X wave.

Details

If the optional /D = *destWave* flag is omitted, then the wave is differentiated in place overwriting the original data.

When using a method that deletes points (/EP=1) with a multidimensional wave, deletion is not done if no dimension is specified.

When using an X wave, the X wave must match the Y wave data type (excluding the complex type flag) and it must be 1D with the number points matching the size of the dimension being differentiated. X waves are not used with integer source waves.

`Differentiate/METH=1/EP=1` is the inverse of `Integrate/METH=2`, but `Integrate/METH=2` is the inverse of `Differentiate/METH=1/EP=1` only if the original first data point is added to the output wave.

Differentiate applied to an XY pair of waves does not check the ordering of the X values and doesn't care about it. However, it is usually the case that your X values should be monotonic. If your X values are not monotonic, you should be aware that the X values will be taken from your X wave in the order they are found, which will result in random X intervals for the X differences. It is usually best to sort the X and Y waves first (see **Sort**).

See Also

The **Integrate** operation.

digamma

digamma(*x*)

The digamma function returns the digamma, or psi function of *x*. This is the logarithmic derivative of the gamma function:

$$\psi(z) \equiv \frac{\mathrm{d}}{\mathrm{d}z}\ln\Gamma(z) = \frac{\Gamma'(z)}{\Gamma(z)}.$$

In complex expressions, *x* is complex, and `digamma(x)` returns a complex value.

Limited testing indicates that the accuracy is approximately 1 part in 10^{16}, at least for moderately-sized values of *x*.

DimDelta

DimDelta(*waveName, dimNumber*)

The DimDelta function returns the scale factor delta of the given dimension.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers and 3 for chunks. If *dimNumber*=0 this is identical to `deltax(waveName)`.

See Also

For an explanation of waves and wave scaling, see **Changing Dimension and Data Scaling** on page II-83.

DimOffset

DimOffset(*waveName, dimNumber*)

The DimOffset function returns the scaling offset of the given dimension.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers, and 3 for chunks. If *dimNumber*=0 this is identical to `leftx(waveName)`.

See Also

For an explanation of waves and wave scaling, see **Changing Dimension and Data Scaling** on page II-83.

DimSize

DimSize(*waveName, dimNumber*)

The DimSize function returns the size of the given dimension.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers, and 3 for chunks. For a 1D wave, `DimSize(waveName,0)` is identical to `numpnts(waveName)`.

Dir

Dir [*dataFolderSpec*]

The Dir operation returns a listing of all the objects in the specified data folder.

Parameters

If you omit *dataFolderSpec* then the current data folder is used.

If present, *dataFolderSpec* can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

Details

The format of the printed information is the same as the format used by the string function **DataFolderDir**. Igor programmers may find it more convenient to use **CountObjects** and **GetIndexedObjName**.

Usually it is easier to use the Data Browser (Data menu). However, **Dir** is useful when you want to copy a name into the command line or when you want to document the current state of the folder in the history.

See Also

Chapter II-8, **Data Folders**.

Display

Display [*flags*] [*waveName* [, *waveName*]…[**vs** *xwaveName*]]
[**as** *titleStr*]

The Display operation creates a new graph window or subwindow, and appends the named waves, if any. Waves are displayed as 1D traces.

By default, waves are plotted versus the left and bottom axes. Use the /L, /B, /R, and /T flags to plot the waves against other axes.

Parameters

Up to 100 *waveName*s may be specified, subject to the 400 character command limit. If no wave names are specified, a blank graph is created and the axis flags are ignored.

If you specify "vs *xwaveName*", the Y values of the named waves are plotted versus the Y values of *xwaveName*. If you don't specify "vs *xwaveName*", the Y values of each *waveName* are plotted versus its own X values.

If *xwaveName* is a text wave, the resulting plot is a category plot. Each element of *waveName* is plotted by default in bars mode (`ModifyGraph mode=5`) against a category labeled with the text of the corresponding element of *xwaveName*.

The Y waves for a category plot should have point scaling (see **Changing Dimension and Data Scaling** on page II-83); this is how category plots were intended to work. However, if all the Y waves have the same scaling, it will work correctly.

titleStr is a string expression containing the graph's title. If not specified, Igor will provide one which identifies the waves displayed in the graph.

Subsets of data, including individual rows or columns from a matrix, may be specified using **Subrange Display Syntax** on page II-288.

You can provide a custom name for a trace by appending /TN=traceName to the waveName specification. This may be useful when displaying waves with the same name but from different data folders. See **User-defined Trace Names** on page IV-71 for more information. This feature was added in Igor Pro 6.20.

Flags

/B[=*axisName*] Plots X coordinates versus the standard or named bottom axis.

/FG=(*gLeft*, *gTop*, *gRight*, *gBottom*)
Specifies the frame guide to which the outer frame of the subwindow is attached inside the host window.

The standard frame guide names are `FL`, `FR`, `FT`, and `FB`, for the left, right, top, and bottom frame guides, respectively, or user-defined guide names as defined by the host. Use `*` to specify a default guide name.

Guides may override the numeric positioning set by /W.

/HIDE=*h* Hides (h = 1) or shows (h = 0, default) the window.

/HOST=*hcSpec*
: Embeds the new graph in the specified host window or subwindow *hcSpec*.

  When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/I
: Specifies that /W coordinates are in inches.

/K=*k*
: Specifies window behavior when the user attempts to close it.

  k=0: Normal with dialog (default).

  k=1: Kills with no dialog.

  k=2: Disables killing.

  k=3: Hides the window.

  If you use /K=2 or /K=3, the only way to kill the window is via the DoWindow/K operation.

/L[=*axisName*]
: Plots Y coordinates versus the standard or named left axis.

/M
: Specifies that /W coordinates are in centimeters.

/N=*name*
: Requests that the created graph have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen graph name. Use `DoWindow/K name` to ensure that *name* is available.

/PG=(*gLeft*, *gTop*, *gRight*, *gBottom*)
: Specifies the inner plot rectangle of the graph subwindow inside its host window.

  The standard plot rectangle guide names are `PL`, `PR`, `PT`, and `PB`, for the left, right, top, and bottom plot rectangle guides, respectively, or user-defined guide names as defined by the host. Use `*` to specify a default guide name.

  Guides may override the numeric positioning set by /W.

/R[=*axisName*]
: Plots Y coordinates versus the standard or named right axis.

/T[=*axisName*]
: Plots Y coordinates versus the standard or named top axis.

/W=(*left*,*top*,*right*,*bottom*)
: Gives the graph a specific location and size on the screen. Coordinates for /W are in points unless /I or /M are specified before /W.

  When used with the /HOST flag, the specified location coordinates of the sides can have one of two possible meanings:

  1) When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.
  2) When any value is greater than 1, coordinates are taken to be fixed locations measured in points relative to the top left corner of the host frame.

  When the subwindow position is fully specified using guides (using the /HOST, /FG, or /PG flags), the /W flag may still be used although it is not needed.

Details

If /N is not used, Display automatically assigns to the graph a name of the form "Graph*n*", where *n* is some integer. In a function or macro, the assigned name is stored in the S_name string. This is the name you can use to refer to the graph from a procedure. Use the **RenameWindow** operation to rename the graph.

Examples

To make a contour plot, use:

```
Display; AppendMatrixContour waveName
```

or

```
Display; AppendXYZContour waveName
```

To display an image, use:

```
Display; AppendImage waveName
```

or

```
NewImage waveName
```

See Also

The **AppendToGraph** operation.

The operations **AppendImage**, **AppendMatrixContour**, **AppendXYZContour**, and **NewImage**. For more information on Category Plots, see Chapter II-13, **Category Plots**.

The operations **ModifyGraph**, **ModifyContour**, and **ModifyImage** for changing the characteristics of graphs.

The **DoWindow** operation for changing aspects of the graph window.

DisplayHelpTopic

DisplayHelpTopic [**/K=*k* /Z**] ***TopicString***

The DisplayHelpTopic operation displays a help topic as if a help link had been clicked in an Igor help file.

Parameters

TopicString is string expression containing the topic. It may be in one of three forms: <topic name>, <subtopic name>, <topic name>[<subtopic name>]. These forms are illustrated by the examples.

Make sure that your topic string is specific to minimize the likelihood that Igor will find the topic in a help file other than the one you intended. To avoid this problem, it is best to use the <topic name>[<subtopic name>] form if possible.

Flags

/K=*k* *k*=0: Leaves the help file open indefinitely (default). Use this if the help topic may be of interest in any experiment.

k=1: If the found topic is in a closed help file, the help file closes with the current experiment. Use this if the help topic is tightly associated with the current experiment.

/Z Ignore errors. If /Z is used, DisplayHelpTopic sets V_flag to 0 if the help topic was found or to a nonzero error code if it was not found. V_flag is set only when /Z is used.

Details

DisplayHelpTopic first searches for the specified topic in the open help files. If the topic is not found, it then searches all help files in the Igor Pro folder and subfolders.

If the topic is still not found, it then searches all help files in the current experiment's home folder, but not in subfolders. This puts a help file that is specific to a particular experiment in the experiment's home folder.

If the topic is still not found and if DisplayHelpTopic was called from a procedure and if the procedure resides in a stand-alone file on disk (i.e., it is not in the built-in procedure window or in a packed procedure file), Igor then searches all help files in the procedure file's folder, but not in subfolders. This puts a help file that is specific to a particular set of procedures in the same folder as the procedure file.

If Igor finds the topic, it displays it. If Igor can not find the topic, it displays an error message, unless /Z is used.

Examples

```
// This example uses the topic only.
DisplayHelpTopic "Waves"

// This example uses the subtopic only.
DisplayHelpTopic "Waveform Arithmetic and Assignment"

// This example uses the topic[subtopic] form.
DisplayHelpTopic "Waves[Waveform Arithmetic and Assignment]"
```

See Also

Chapter II-1, **Getting Help** for information about Igor help files and formats.

DisplayProcedure

DisplayProcedure [**/B=*winTitleOrName* /W=*procWinTitle***] [***functionOrMacroNameStr***]

The DisplayProcedure operation displays the named function or macro by bringing the procedure window it is defined in to the front with the function or macro highlighted.

Parameters

functionOrMacroNameStr is string expression containing the function or macro name. If omitted, you must use /W. *functionOrMacroNameStr* may also contain independent module and/or module name prefixes to display static functions.

Flags

/B=*winTitleOrName* Brings up the procedure window just behind the window with this name or title.

/W=*procWinTitle* Searches in the procedure window with this title. If omitted, it searches all open (nonindependent module) procedure windows.

Details

If a procedure window has syntax errors that prevent Igor from determining where functions and macros start and end, then DisplayProcedure may not be able to locate the procedure.

winTitleOrName is not a string; it is a name. To position the found procedure window behind a window whose title has a space in the name, use the $ operator as in the second example, below.

If *winTitleOrName* does not match any window, then the found procedure window is placed behind the top target window.

procWinTitle is also a name. Use `/W=$"New Polar Graph.ipf"` to search for the function or macro in only that procedure file.

Advanced Details

If `SetIgorOption IndependentModuleDev=1`, *procWinTitle* can also be a title followed by a space and, in brackets, an independent module name. In such cases searches for the function or macro are in the specified procedure window and independent module. (See **Independent Modules** on page IV-214 for independent module details.)

For example, if any procedure file contains these statements:

```
#pragma IndependentModule=myIM
#include <Axis Utilities>
```

The command

```
DisplayProcedure/W=$"Axis Utilities.ipf [myIM]" "HVAxisList"
```

opens the procedure window that contains the `HVAxisList` function, which is in the Axis Utilities.ipf file and the independent module `myIM`. The command uses the `$""` syntax because space and bracket characters interfere with command parsing.

Similarly, if `SetIgorOption IndependentModuleDev=1` then *functionOrMacroNameStr* may also contain an independent module prefix followed by the # character. The preceding command can be rewritten as:

```
DisplayProcedure/W=$"Axis Utilities.ipf" "myIM#HVAxisList"
```

or more simply

```
DisplayProcedure "myIM#HVAxisList"
```

You can use the same syntax to display a static function in a non-independent module procedure file using a module name instead of (or in addition to) the independent module name.s

procWinTitle can also be just an independent module name in brackets to retrieve the text from *any* procedure window that belongs to named independent module:

```
DisplayProcedure/W=$"[myIM]" "HVAxisList"
```

Examples

```
DisplayProcedure "Graph0"
DisplayProcedure/B=$"Strings as Lists.ipf" "MyOwnUserFunction"
DisplayProcedure/W=Procedure         // Shows the main Procedure window
DisplayProcedure/W=$"Strings as Lists.ipf"
```

```
DisplayProcedure "moduleName#myStaticFunctionName"
SetIgorOption IndependentModuleDev=1
```

```
DisplayProcedure "WMGP#GizmoBoxAxes#DrawAxis"
```

See Also

Independent Modules on page IV-214.

MacroList, **FunctionList**, and **ProcedureText**, **HideProcedures**, **DoWindow**.

do-while

```
do
   <loop body>
while(<expression>)
```

A do-while loop executes *loop body* until *expression* evaluates as FALSE (zero) or until a break statement is executed.

See Also

Do-While Loop on page IV-36 and **break** for more usage details.

DoAlert

```
DoAlert [/T=titleStr] alertType, promptStr
```

The DoAlert operation displays an alert dialog and waits for user to click button.

Parameters

alertType= 0: Dialog with an OK button.
1: Dialog with Yes button and No buttons.
2: Dialog with Yes, No, and Cancel buttons.

promptStr text that is displayed in the alert dialog.

Flags

/T=*titleStr* Changes the title of the dialog window from the default title.

Details

DoAlert sets the variable V_flag as follows:

V_flag= 1: Yes clicked.
2: No clicked.
3: Cancel clicked.

See Also

The **Abort** operation.

DoIgorMenu

```
DoIgorMenu [/C] MenuNameStr, MenuItemStr
```

The DoIgorMenu operation allows an Igor programmer to invoke Igor's built-in menu items. This is useful for bringing up Igor's built-in dialogs under program control.

Parameters

MenuNameStr The name of an Igor menu, like "File", "Graph", or "Load Waves".

MenuItemStr The text of an Igor menu item, like "Copy" (in the Edit menu) or "New Graph" (in the Windows menu).

Flags

/C Just Checking. The menu item is not invoked, but V_flag is set to 1 if the item was enabled or to 0 if it was not enabled.

Details

All menu names and menu item text are in English to ensure that code developed for a localized version of Igor Pro will run on all versions. Note that no trailing "…" is used in *MenuItemStr*.

V_flag is set to 1 if the corresponding menu item was enabled, which usually means the menu item was successfully selected. Otherwise V_flag is 0. V_flag does not reflect the success or failure of the resulting dialog, if any.

If the menu item selection displays a dialog that generates a command, clicking the Do It button executes the command immediately without using the command line as if Execute/Z operation had been used. Clicking the To Cmd Line button appends the command to the command line rather than inserting the command at the front.

The DoIgorMenu operation will not attempt to select a menu during curve fitting. Doubtless there are other times during which using DoIgorMenu would be unwise.

The text of some items in the File menu changes depending on the type of the active window. In these cases you must pass generic text as the *MenuItemStr* parameter. Use "Save Window", "Save Window As", "Save Window Copy", "Adopt Window", and "Revert Window" instead of "Save Notebook" or "Save Procedure", etc. Use "Page Setup" instead of "Page Setup For All Graphs", etc. Use "Print" instead of "Print Graph", etc.

See Also

The **SetIgorMenuMode** and **Execute** operations.

DoPrompt

DoPrompt [**/HELP=*helpStr***] ***dialogTitleStr, variable*** [**, *variable***]…

The DoPrompt command in a function invokes the simple input dialog. A DoPrompt specifies the title for the simple input dialog and which input variables are to be included in the dialog.

Flags

/HELP=*helpStr* Sets the help topic or help text that appears when the dialog's Help button is pressed.

helpStr can be a help topic and subtopic such as is used by `DisplayHelpTopic/K=1` *`helpStr`*, or it can be text (255 characters max) that is displayed in a subdialog just as if `DoAlert 0,` *`helpStr`* had been called, or *helpStr* can be `""` to remove the Help button.

Parameters

variable is the name of a dialog input variable, which can be real or complex numeric local variable or local string variable, defined by a Prompt statement. You can specify as many as 10 variables.

dialogTitleStr is a string or string expression containing the text for the title of the simple input dialog.

Details

Prompt statements are required to define what variables are to be used and the text for any string expression to accompany or describe the input variable in the dialog. When a DoPrompt variable is missing a Prompt statement, you will get a compilation error. Pop-up string data can not be continued across multiple lines as can be done using Prompt in macros. See **Prompt** for further usage details.

Prompt statements for the input variables used by DoPrompt must come before the DoPrompt statement itself, otherwise, they may be used anywhere within the body of a function. The variables are not required to be input parameters for the function (as is the case for Prompt in macros) and they may be declared within the function body. DoPrompt can accept as many as 10 variables.

Functions can use multiple DoPrompt statements, and Prompt statements can be reused or redefined.

When the user clicks the Cancel button, any new input parameter values are not stored in the variables.

DoPrompt sets the variable V_flag as follows:

`V_flag=` 0: Continue button clicked.

1: Cancel button clicked.

See Also

The Simple Input Dialog on page IV-122, the **Prompt** keyword, and **DisplayHelpTopic**.

DoUpdate

DoUpdate [**/E=*e* /W=*targWin* /SPIN=*ticks***]

The DoUpdate operation updates windows and dependent objects.

Flags

/E=*e* Used with /W, /E=1 marks window as a progress window that can accept mouse events while user code is executing. Currently, only control panel windows can be used as a progress window.

/W=*targWin* Updates only the specified window. Does not update dependencies or do any other updating. V_Flag is set to the truth the window exists. Currently, only graph and panel windows honor the /W flag.

/SPIN=*ticks* Sets the delay between the start of a control procedure and the spinning beachball. *ticks* is the delay in ticks (60th of a second.) Unless used with the /W flag, /SPIN just sets the delay and an update is not done.

Details

Call DoUpdate from an Igor procedure to force Igor to update any objects that need updating. Igor updates any graphs, tables or page layouts that need to be updated and also any objects (string variables, numeric variables, waves, controls) that depend on other objects that have changed since the last update.

Igor performs updates automatically when:

- No user-procedure is running.
- An interpreted procedure (Macro, Proc, Window type procedures) is running and PauseUpdate or DelayUpdate is not in effect.

An automatic DoUpdate is not done while a user-defined function is running. You can call DoUpdate from a user-defined function to force an update.

See Also

The **DelayUpdate**, **PauseUpdate**, and **ResumeUpdate** operations, **Progress Windows** on page IV-134.

DoWindow

DoWindow [***flags***] [***windowName***]

The DoWindow operation controls various window parameters and aspects. There are additional forms for DoWindow when the /S or /T flags are used; see the following DoWindow entries.

Parameters

windowName is the name of a graph, table, page layout, notebook, panel or XOP target window.

A window's name is *not* the same as its title. The title is shown in the window's title bar. The name is used to manipulate the window from Igor commands. You can check both the name and the title using the Window Control dialog (in the Arrange submenu of the Window menu).

Flags

/B[=*bname*] Moves the specified window to the back (to the bottom of desktop) or behind window *bname*.

/C Changes the name of the target window to the specified name. The specified name must not be used for any other object except that it can be the name of an existing window macro.

/C/N Changes the target window name and creates a new window macro for it. However, /N does nothing if a macro or function is running. /N is not applicable to notebooks.

/D Deletes the file associated with window, if any (for notebooks only).

/F Brings the window with the given name to the front (top of desktop).

/H Specifies the command/history window as the target of the operation. When using /H, *windowName* must not be specified and only the /B and /HIDE flags are honored.

Use /H to bring the command window to the front (top of desktop).

Use /H/B to send the history/command window to the bottom of the desktop.

Use /H/HIDE to hide or show the command window.

/HIDE=*h* Sets hidden state of a window.

h=0: Visible.

h=1: Hidden.

h=?: Sets the variable V_flag as follows:

0: The window does not exist.

1: The window is visible.

2: The window is hidden.

You can also read the hidden state using **GetWindow** and set it using **SetWindow**.

/K Kills the window with the given name.

/N Creates a new window macro for the window with the given name. However, /N does nothing if a macro or function is running. /N is not applicable to notebooks.

/R Replaces (updates) the window macro for the named window or creates it if it does not yet exist. However, /R does nothing if a macro or function is running. /R is not applicable to notebooks.

/R/K — Replaces (updates) the window macro for the named window or creates it if it does not yet exist and then kills the window. However, /R does nothing if a macro or function is running. /R is not applicable to notebooks.

/W=*targWin* — Designates *targWin* as the target window; it also requires that you specify *windowName*. Use this mainly with floating panels, which are always on top. You can use a subwindow specification of an external subwindow only with the /T flag or without any flags.

Details

DoWindow sets the variable V_flag to 1 if there was a window with the specified name after DoWindow executed, to 0 if there was no such window, or to 2 if the window is hidden.

Call DoWindow with no flags to check if a window exists.

When used with the /N flag, *windowName* must not conflict with the name of any other object. When used with the /C flag, *windowName* must not conflict with the name of any other object except that it can be the name of an existing window macro.

As of Igor Pro 3.13, the /R and /N flags do nothing when executed while a macro or function is running. This is necessary because changing procedures while they are executing causes unpredictable and undesirable results. However you can use the Execute/P operation to cause the DoWindow command to be executed after procedures are finished running. For example:

```
Function SaveWindowMacro(windowName)
   String windowName                    // "" for top graph or table
   if (strlen(windowName) == 0)
      windowName = WinName(0, 3)        // Name of top graph or table
   endif
   String cmd
   sprintf cmd, "DoWindow/R %s", windowName
   Execute/P cmd
End
```

You can use the /D flag in conjunction with the /K flag to kill a notebook window and delete its associated file, if any. /D has no effect on any other type of window and has no effect if the /K flag is not present.

Examples

```
DoWindow Graph0            // Set V_flag to 1 if Graph0 window exists.
DoWindow/F Graph0          // Make Graph0 the top/target window.
DoWindow/C MyGraph         // Target window (Graph0) renamed MyGraph.
DoWindow/K Panel0          // Kill the Panel0 window.
DoWindow/H/B               // Put the history/command window in back.
DoWindow/D/K Notebook2     // Kill Notebook2, delete its file.
```

See Also

The **RenameWindow** operation for renaming windows or subwindows.

The **IgorInfo** function, the **HideProcedures** operation, and the following DoWindow sections.

DoWindow/T

DoWindow /T *windowName, windowTitleStr*

The DoWindow/T operation sets the window title for the named window to the specified title.

Details

The title is shown in the window's title bar, and listed in the appropriate Windows submenu. The window *name* is still used to manipulate the window, so, for example, the window name (if *windowName* is a graph or table) is listed in the New Layout dialog; not the title.

You can check both the name and the title using the Window Control dialog (in the Control submenu of the Windows menu).

windowName is the name of the window or a special keyword, kwTopWin or kwFrame.

If *windowName* is kwTopWin, DoWindow retitles the top target window.

If *windowName* is kwFrame, DoWindow retitles the "frame" or "application" window that Igor has only under Windows. This is the window that contains Igor's menus and status bar. On Macintosh, kwFrame is allowed, but the command does nothing.

The Window Control dialog does not support kwFrame. The frame title persists until Igor quits or until it is restored as shown in the example. Setting *windowTitleStr* to "" will restore the normal frame title.

Examples

```
DoWindow/T MyGraph, "My Really Neat Graph"
DoWindow/T kwFrame, "My Igor-based Application"
DoWindow/T kwFrame, ""           // restore normal frame title
```

DoWindow/S

DoWindow /N/S=*styleMacroName windowName*

DoWindow /R/S=*styleMacroName windowName*

The DoWindow/S operation creates a new "style macro" for the named window, using the specified style macro name. Does not create or replace the window macro for the specified window.

Flags

/N/S=*styleMacroName* Creates a new style macro with the given name based on the named window.

/R/S=*styleMacroName* Creates or replaces the style macro with the given name based on the named window.

Details

The /R or /N flag must appear before the /S flag.

If the /S flag is present, the DoWindow operations does *not* create or replace the window macro for the specified window.

As of Igor Pro 3.13, the /R and /N flags do nothing when executed while a macro or function is running. This is necessary because changing procedures while they are executing causes unpredictable and undesirable results.

DoXOPIdle

DoXOPIdle

The DoXOPIdle operation sends an IDLE event to all open XOPs. This operation is very specialized. Generally, only the author of an XOP will need to use this operation.

Details

Some XOPs (External OPeration code modules) require IDLE events to perform certain tasks. For example, the SoundInput XOP on the Macintosh transfers sound data from an interrupt buffer to an Igor Named FIFO only during an IDLE event.

Igor does not automatically send IDLE events to XOPs while an Igor program is running. You can call DoXOPIdle from a user-defined program to force Igor to send the event.

DrawAction

DrawAction [**/L=*layerName*/W=*winName***] ***keyword* = *value*** [**, *keyword* = *value*** …]

The DrawAction operation deletes, inserts, and reads back a named drawing object group or the entire draw layer.

Parameters

DrawAction accepts multiple *keyword* = *value* parameters on one line.

beginInsert [=*index*] Inserts draw commands before or at *index* position or at position specified by getgroup or delete parameters; position otherwise is zero.

commands [=*start*,*stop*] Stores commands in S_recreation for draw objects between *start* and *stop* index values, range defined by getgroup, or entire layer otherwise.

delete [=*start*,*stop*] Deletes draw objects between *start* and *stop* index values, range defined by getgroup, or entire layer otherwise.

extractOutline [=*start*,*stop*] Stores polygon outline between *start* and *stop* index values, range defined by getgroup, or entire layer otherwise. Waves W_PolyX and W_PolyY contain coordinates with NaN separators. V_npnts contains the number of objects. Coordinates are for the first object encountered.

endInsert Terminates insert mode.

getgroup=*name* Stores first and last index of named group in V_startPos and V_endPos. Use `_all_` to specify the entire layer. Sets V_flag to truth group exists.

Flags

/L=*layerName* Specifies the drawing layer on which to act. *layerName* is one of the drawing layers as specified in **SetDrawLayer**.

/W=*winName* Sets the named window or subwindow for drawing. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

Commands stored in S_recreation are the same as those that would be generated for the range of objects in the recreation macro for the window but also have comment lines preceding each object of the form:

```
// ;ITEMNO:n;
```

where *n* is the item number of the draw object.

Examples

Create a drawing with a named group:

```
NewPanel /W=(455,124,936,413)
SetDrawEnv fillfgc= (65535,0,0)
DrawRect 58,45,132,103
SetDrawEnv gstart,gname= fred
SetDrawEnv fillfgc= (65535,43690,0)
DrawRect 79,62,154,120
SetDrawEnv arrow= 1
DrawLine 139,70,219,70
SetDrawEnv gstop
SetDrawEnv fillfgc= (0,65535,65535)
DrawRect 95,77,175,138
SetDrawEnv fillfgc= (0,0,65535)
DrawRect 111,91,191,156
```

Get and print commands for the “fred” group:

```
DrawAction getgroup=fred,commands
Print S_recreation
```

prints:

```
// ;ITEMNO:3;
SetDrawEnv gstart,gname= fred
// ;ITEMNO:4;
SetDrawEnv fillfgc= (65535,43690,0)
// ;ITEMNO:5;
DrawRect 79,62,154,120
// ;ITEMNO:6;
SetDrawEnv arrow= 1
// ;ITEMNO:7;
DrawLine 139,70,219,70
// ;ITEMNO:8;
SetDrawEnv gstop
```

Replace group fred (the orange rectangle and the arrow) with a different object. First delete the group and enter insert mode:

```
DrawAction getgroup=fred, delete, begininsert
```

Next draw the replacement:

```
SetDrawEnv gstart,gname= fred
SetDrawEnv fillfgc= (65535,65535,0)
DrawOval 82,62,161,123
SetDrawEnv gstop
```

Lastly exit insert mode:

```
DrawAction endinsert
```

See Also

The **SetDrawEnv** operation and Chapter III-3, **Drawing**.

DrawArc

DrawArc [**/W=*winName*/X/Y**] ***xOrg*, *yOrg*, *arcRadius*, *startAngle*, *stopAngle***

The DrawArc operation draws a circular counterclockwise arc with center at *xOrg* and *yOrg*.

Parameters

(*xOrg*, *yOrg*) defines the center point for the arc in the currently active coordinate system.

Angles are measured in degrees increasing in a counterclockwise direction. The *startAngle* specifies the starting angle for the arc and *stopAngle* specifies the end. If *stopAngle* is equal to *startAngle*, 360° is added to *stopAngle*. Thus, a circle can be drawn using *startAngle* = *stopAngle*.

The *arcRadius* is the radial distance measured in points from (*xOrg*, *yOrg*).

Flags

/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.
/X	Measures *arcRadius* using the current X-coordinate system. If /Y is also used, the arc may be elliptical.
/Y	Measures *arcRadius* using the current Y-coordinate system. If /X is also used, the arc may be elliptical.

Details

Arcs honor the current dash pattern and arrowhead setting in the same way as polygons and Beziers. In fact, arcs are implemented using Bezier curves.

Normally, you would create arcs programmatically. If you need to sketch an arc-like object, you should probably use a Bezier curve because it is more flexible and easier to adjust. However, there is one handy feature of arcs that make them useful for manual drawing: the origin can be in any of the supported coordinate systems and the radius is in points.

To draw an arc interactively, see **Arcs and Circles** on page III-72 for instructions.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv** and **SetDrawLayer** operations.

The **DrawBezier**, **DrawOval** and **DrawAction** operations.

DrawBezier

DrawBezier [**/W=*winName* /ABS**] ***xOrg*, *yOrg*, *hScaling*, *vScaling*, {x_0,y_0,x_1,y_1 …}**

DrawBezier [**/W=*winName* /ABS**] ***xOrg*, *yOrg*, *hScaling*, *vScaling*, *xWaveName*, *yWaveName***

DrawBezier/A [**/W=*winName***] {x_n, y_n, x_{n+1}, y_{n+1} …}

The DrawBezier operation draws a Bezier curve with first point of the curve positioned at *xOrg* and *yOrg*.

Parameters

(*xOrg*, *yOrg*) defines the starting point for the Bezier curve in the currently active coordinate system.

hScaling and *vScaling* set the horizontal and vertical scale factors about the origin, with 1 meaning 100%.

The *xWaveName*, *yWaveName* version of DrawBezier gets data from the named X and Y waves. This connection is maintained so that any changes to either wave will result in updates to the Bezier curve.

To use the version of DrawBezier that takes a literal list of vertices, you place as many vertices as you like on the first line and then use as many /A versions as necessary to define all the vertices.

Flags

/A	Appends the given vertices to the currently open Bezier (freshly drawn or current selection).
/ABS	Suppresses the default subtraction of the first point from the rest of the data.

/W=*winName* Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

Data waves defining Bezier curves must have 1+3**n* data points. Every third data point is an anchor point and lies on the curve; intervening points are control points that define the direction of the curve relative to the adjacent anchor point.

Normally, you should create and edit a Bezier curve using drawing tools, and not calculate values. See **Polygon Tool** on page III-73 for instructions.

As of Igor Pro 5.02, you can specify the /ABS flag to suppress the default subtraction of the first point. Also, you can now insert NaN values to break a bezier into pieces.

To change just the origin and scale without respecifying the data use:

```
DrawBezier xOrg, yOrg, hScaling, vScaling,{}
```

To delete an anchor point, press Option (*Macintosh*) or Alt (*Windows*) and then click on the anchor. To insert a new anchor, click on the curve between anchor points.

Example

Create a half circle approximation from three anchor points starting at the 12 o'clock position, with an anchor at the 3 o'clock position, and the last at the 6 o'clock position using explicit values:

```
// Set plot relative coords, 0-1, no fill
SetDrawEnv xcoord=prel, ycoord=prel, fillpat= 0, save

Variable len= 0.275       // control point length = 0.55 * radius for a circle
// Starting anchor point has only a trailing control point
Variable anchor0x= 0.5, anchor0y=1          // starting point at 6 o'clock
Variable t0x= 0.5+len, t0y= 1               // trailing control point

// second anchor point has both leading and trailing control points
Variable l1x=1, l1y = 0.5+len               // leading control point
Variable anchor1x= 1, anchor1y= 0.5         // 3 o'clock anchor
Variable t1x=1, t1y = 0.5-len               // trailing control point

// Last (3rd) anchor point has only a leading control point
Variable l2x=0.5+len, l2y = 0               // leading control point
Variable anchor2x= 0.5, anchor2y= 0         // 6 o'clock

// One command per anchor for clarity
DrawBezier anchor0x, anchor0y, 1,1, {anchor0x, anchor0y, t0x, t0y}
DrawBezier/A {l1x, l1y, anchor1x, anchor1y, t1x, t1y}
DrawBezier/A {l2x, l2y, anchor2x, anchor2y}
```

To draw using waves:

```
Make/O bezierX= {anchor0x, t0x, l1x, anchor1x, t1x, l2x, anchor2x }
Make/O bezierY= {anchor0y, t0y, l1y, anchor1y, t1y, l2y, anchor2y }
DrawBezier anchor0x, anchor0y, 1,1, bezierX, bezierY
```

See Also

Chapter III-3, **Drawing**.

Polygon Tool on page III-73 for discussion on creating Beziers. **DrawPoly and DrawBezier** on page III-81 and the **SetDrawEnv** and **SetDrawLayer** operations.

The **DrawArc**, **DrawPoly** and **DrawAction** operations.

DrawLine

DrawLine [**/W=*winName***] x_0, y_0, x_1, y_1

The DrawLine operation draws a line in the target graph, layout or control panel from (*x0*,*y0*) to (*x1*,*y1*).

Flags

/W=*winName* Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

The coordinate system as well as the line's thickness, color, dash pattern and other properties are determined by the current drawing environment. The line is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawOval

DrawOval [**/W=*winName***] ***left, top, right, bottom***

The DrawOval operation draws an oval in the target graph, layout or control panel within the rectangle defined by *left*, *top*, *right,* and *bottom*.

Flags

/W=*winName* Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

The coordinate system as well as the oval's thickness, color, dash pattern and other properties are determined by the current drawing environment (note that you cannot draw dashed ovals). The oval is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawPICT

DrawPICT [**/W=*winName***][**/RABS**] ***left, top, hScaling, vScaling, pictName***

The DrawPICT operation draws the named picture in the target graph, layout or control panel. The *left* and *top* parameters set the position of the top/left corner of the picture. *hScaling* and *vScaling* set the horizontal and vertical scale factors with 1 meaning 100%.

Flags

/RABS Draws the named picture using absolute scaling. In this mode, it draws the picture in the rectangle defined by *left* and *top* for point (x0,y0), and by *hScaling* and *vScaling* for point (x1,y1), respectively.

/W=*winName* Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

The coordinate system for the left and top parameters is determined by the current drawing environment. The PICT is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawPoly

DrawPoly [**/W=*winName* /ABS**] ***xorg, yorg, hScaling, vScaling, xWaveName, yWaveName***
DrawPoly [**/W=*winName* /ABS**] ***xorg, yorg, hScaling, vScaling,*** $\{x_0,y_0,x_1,y_1 \ldots\}$
DrawPoly/A [**/W=*winName***] $\{x_n, y_n, x_{n+1}, y_{n+1} \ldots\}$

The DrawPoly operation draws a polygon in the target graph, layout or control panel.

Parameters

(*xorg*, *yorg*) defines the starting point for the polygon in the currently active coordinate system.

hScaling and *vScaling* set the horizontal and vertical scale factors, with 1 meaning 100%.

The *xWaveName*, *yWaveName* version of DrawPoly gets data from those X and Y waves. This connection is maintained so that changes to either wave will update the polygon.

The DrawPoly operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

To use the version of DrawPoly that takes a literal list of vertices, you place as many vertices as you like on the first line and then use as many /A versions as necessary to define all the vertices.

Flags

/A	Appends the given vertices to the currently open polygon (freshly drawn or current selection).
/ABS	Suppresses the default subtraction of the first point from the rest of the data.
/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

Because *xorg* and *yorg* define the location of the starting vertex of the poly, adding or subtracting a constant from the vertices will have no effect. The first XY pair in the {*x0*, *y0*, *x1*, *y1*,…} vertex list will appear at (*xorg*,*yorg*) *regardless* of the value of *x0* and *y0*. *x0* and *y0* merely serve to set a reference point for the list of vertices. Subsequent vertices are relative to (*x0*,*y0*).

To keep your mental health intact, we recommend that you specify (*x0*,*y0*) as (0,0) so that all the following vertices are offsets from that origin. Then (*xorg*,*yorg*) sets the position of the polygon and all of the vertices in the list are relative to that origin.

An alternate method is to use the same values for (*x0*,*y0*) as for (*xorg*,*yorg*) if you consider the vertices to be "absolute" coordinates.

As of Igor Pro 5.02, you can specify the /ABS flag to suppress the subtraction of the first point. Also, you can now insert NaN values to break a polygon into pieces.

To change just the origin and scale of the currently open polygon — without having to respecify the data — use:

```
DrawPoly xorg, yorg, hScaling, vScaling,{}
```

The coordinate system as well as the polygon's thickness, color, dash pattern and other properties are determined by the current drawing environment. The polygon is drawn in the current draw layer for the window, as determined by SetDrawLayer.

Examples

Here are some commands to draw some small triangles using absolute drawing coordinates (see **SetDrawEnv**).

```
Display             // make a new empty graph
//Draw one triangle, starting at 50,50 at 100% scaling
SetDrawEnv xcoord= abs,ycoord= abs
DrawPoly 50,50,1,1, {0,0,10,10,-10,10,0,0}
//Draw second triangle below and to the right, same size and shape
SetDrawEnv xcoord= abs,ycoord= abs
DrawPoly 100,100,1,1, {0,0,10,10,-10,10,0,0}
```

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv** and **SetDrawLayer** operations, **DrawPoly and DrawBezier** on page III-81, and Chapter III-3, **Drawing** and **DrawAction**.

DrawRect

DrawRect [**/W=*winName***] ***left, top, right, bottom***

The DrawRect operation draws a rectangle in the target graph, layout or control panel within the rectangle defined by *left*, *top*, *right*, and *bottom*.

Flags

/W=*winName* Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

The coordinate system as well as the rectangle's thickness, color, dash pattern and other properties are determined by the current drawing environment. The rectangle is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawRRect

DrawRRect [**/W=*winName***] ***left, top, right, bottom***

The DrawRRect operation draws a rounded rectangle in the target graph, layout or control panel within the rectangle defined by *left*, *top*, *right*, and *bottom*.

Flags

/W=*winName* Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

The coordinate system as well as the rectangle's rounding, thickness, color, dash pattern and other properties are determined by the current drawing environment. The rounded rectangle is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawText

DrawText [**/W=*winName***] ***x_0, y_0, textStr***

The DrawText operation draws the specified text in the target graph, layout or control panel. The position of the text is defined by (*x0*, *y0*) along with the current environment settings of textxjust, textyjust and textrot parameters of **SetDrawEnv**.

Flags

/W=*winName* Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

The coordinate system as well as the text's font, size, style and other properties are determined by the current drawing environment. The text is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DSPDetrend

DSPDetrend [***flags***] ***srcWave***

The DSPDetrend operation removes from *srcWave* a trend defined by the best fit of the specified function to the data in *srcWave*.

Flags

/F= *function*	*function* is the name of a built-in curve fitting function: gauss, lor, exp, dblexp, sin, line, poly (requires /P flag), hillEquation, sigmoid, power, lognormal, poly2d (requires /P flag), gauss2d. If *function* is unspecified, the defaults are line if *srcWave* is 1D or poly2d if *srcWave* is 2D.
/M=*maskWave*	Detrending will only affect points that are nonzero in *maskWave*. Note that *maskWave* must have the same dimensionality as *srcWave*.
/P=*polyOrder*	Specifies polynomial order for poly or poly2d functions (see **CurveFit** for details). By default *polyOrder* =3 for the 1D case and *polyOrder* =1 for the 2D case.
/Q	Quiet mode; no error reporting.

Details

DSPDetrend sets V_flag to zero when the operation succeeds, otherwise it will be set to -1 or will contain an error code from the curve fitting routines. Results are saved in the wave W_Detrend (for 1D input) or M_Detrend (for 2D input) in the current data folder. If a wave by that name already exists in the current data folder it will be overwritten.

See Also

CurveFit for more information about V_FitQuitReason and the built-in fitting functions.

DSPPeriodogram

DSPPeriodogram [***flags***] ***srcWave*** [***srcWave2***]

The DSPPeriodogram operation calculates the periodogram, cross-spectral density or the degree of coherence of the input waves. The result of the operation is stored in the wave W_Periodogram in the current data folder.

To compute the cross-spectral density or the degree of coherence, you need to specify the second wave using the optional *srcWave2* parameter. In this case, W_Periodogram will be complex and the /dB and /dBR flags do not apply.

Flags

/dB	Expresses results in dB using the maximum value as reference.
/dBR=*ref*	Express the results in dB using the specified *ref* value.
/COHR	Computes the degree of coherence. This flag applies when the input consists of two waves.
/DLSG	When computing the periodogram, cross-spectral density or the degree of coherence using multiple segments the operation by default pads the last segment with zeros as necessary. If you specify this flag, an incomplete last segment is dropped and not included in the calculation.
/NODC=*val*	Suppresses the DC term *val*=1: Removes the DC by subtracting the average value of the signal before processing and before applying any window function (see /Win below). *val*=2: Suppresses the DC term by setting it equal to the second term in the FFT array.

val=0: Computes the DC term using the FFT (default).

/NOR=*N* Sets the normalization, N, in the periodogram equation. By default, it is the number of data points times the square norm of the window function (if any).

N=0 or 1: Skips default normalization.

Any other value of *N* is used as the only normalization.

/Q Quiet mode; suppresses printing in the history area.

/SEGN={*ptsPerSegment*, *overlapPts*}

Use this flag to compute the periodogram, cross-spectral density or degree of coherence by averaging over multiple segments taken from the input waves. The size of each interval is *ptsPerSegment*. *overlapPts* determines the number of points at the end of each interval that are included in the next segment.

/R=[*startPt*, *endPt*] Calculates the periodogram for a limited range of the wave. *startPt* and *endPt* are expressed in terms of point numbers in *srcWave*.

/R=(*startX*, *endX*) Calculates the periodogram for a limited range of the wave. *startX* and *endX* are expressed in terms of x-values. Note that this option will convert your x-specifications to point numbers and some roundoff may occur.

/WIN=*windowKind* Specifies the window type. If you omit the /W flag, DSPPeriodogram uses a rectangular window for the full wave or the range of data selected by the /R flag.

Choices for *windowKind* are:

Bartlett, Blackman367, Blackman361, Blackman492, Blackman474, Cos1, Cos2, Cos3, Cos4, Hamming, Hanning, KaiserBessel20, KaiserBessel25, KaiserBessel30, Parzen, Poisson2, Poisson3, Poisson4, and Riemann.

See **FFT** for window definition details.

/Z Do not report errors. When an error occurs, V_flag is set to -1.

Details

The default periodogram is defined as

$$Periodogram = \frac{|F(s)|^2}{N},$$

where F (s) is the Fourier transform of the signal s and N is the number of points.

In most practical situations you need to account for using a window function (when computing the Fourier transform) which takes the form

$$Periodogram = \frac{|F(s \cdot w)|^2}{N_p N_w},$$

where w is the window function, N_p is the number of points and N_w is the normalization of the window function.

If you compute the periodogram by subdividing the signal into multiple segments (with any overlap) and averaging the results over all segments, the expression for the periodogram is

$$Periodogram = \frac{\sum_{i=1}^{M} |F(s_i \cdot w)|^2}{M N_s N_w},$$

where si is the ith segment s, Ns is the number of points per segment and M is the number of segments.

When calculating the cross-spectral density (csd) of two waves s_1 and s_2, the operation results in a complex valued wave

$$csd = \frac{F(s_A)[F(s_B)]^*}{N},$$

which contains the normalized product of the Fourier transform of the first wave S_A with the complex conjugate of the Fourier transform of the second wave S_B. The extension of the csd calculation to segment averaging has the form

$$csd = \frac{\sum_{i=0}^{M} F(s_{Ai})[F(s_{Bi})]^*}{MN_sN_w},$$

where S_{Ai} is the ith segment of the first wave, M is the number of segments and N_s is the number of points in a segment.

The degree of coherence is a normalized version of the cross-spectral density. It is given by

$$\gamma = \frac{\sum_{i=0}^{M} F(s_{Ai})[F(s_{Bi})]^*}{\sqrt{\sum_{i=0}^{M} F(s_{Ai})[F(s_{Ai})]^* \sum_{i=0}^{M} F(s_{Bi})[F(s_{Bi})]^*}}.$$

The bias in the degree of coherence is calculated using the approximation

$$B = \frac{1}{M}\left[1 - |\gamma|^2\right]^2.$$

The bias is stored in the wave W_Bias.

If you use the /SEGN flag the actual number of segments is reported in the variable V_numSegments.

Note that DSPPeriodogram does not test the dimensionality of the wave; it treats the wave as 1D. When you compute the cross-spectral density or the degree of coherence the number-type, dimensionality and the scaling of the two waves must agree.

See Also

The **ImageWindow** operation for 2D windowing applications. **FFT** for window equations and details.

The **Hanning**, **LombPeriodogram** and **MatrixOp** operations.

References

For more information about the use of window functions see:

Harris, F.J., On the use of windows for harmonic analysis with the discrete Fourier Transform, *Proc, IEEE, 66*, 51-83, 1978.

G.C. Carter, C.H. Knapp and A.H. Nuttall, The Estimation of the Magnitude-squared Coherence Function Via Overlapped Fast Fourier Transform Processing, *IEEE Trans. Audio and Electroacoustics*, V. AU-21, (4) 1973.

Duplicate

Duplicate [***flags***][***type flags***] ***srcWaveName, destWaveName*** [**,** ***destWaveName***]…

The Duplicate operation creates new waves, the names of which are specified by *destWaveName*s and the contents, data type and scaling of which are identical to *srcWaveName*.

Parameters

srcWaveName must be the name of an existing wave.

The *destWaveNames* should be wave names not currently in use unless the /O flag is used to overwrite existing waves.

Flags

/FREE — Creates a free wave (see **Free Waves** on page IV-71).
/FREE is allowed only in functions and only if a simple name or structure field is specified as the destination wave name.
Requires Igor Pro 6.1 or later. For advanced programmers only.

/O — Overwrites existing waves with the same name as *destWaveName*.

/R=(*startX*,*endX*) — Specifies an X range in the source wave from which the destination wave is created. See Details for further discussion of /R.

/R=(*startX*,*endX*)(*startY*,*endY*)
Specifies both X and Y range. Further dimensions are constructed analogously. See Details for further discussion of /R.

/R=[*startP*,*endP*] — Specifies a row range in the source wave from which the destination wave is created. Further dimensions are constructed just like the scaled dimension ranges. See Details for further discussion of /R.

Type Flags *(used only in functions)*

When used in user-defined functions, Duplicate can also take the /B, /C, /D, /I, /S, /U, /W, /T, /DF and /WAVE flags. This does not affect the result of the Duplicate operation - these flags are used only to identify what kind of wave is expected at runtime.

This information is used if, later in the function, you create a wave assignment statement using a duplicated wave as the destination:

```
Function DupIt(wv)
    Wave/C wv                   //complex wave

    Duplicate/O/C wv,dupWv      //tell Igor that dupWv is complex
    dupWv[0]=cmplx(5.0,1.0)     //no error, because dupWv known complex
    …
```

If Duplicate did not have the /C flag, Igor would complain with a "function not available for this number type" message when it tried to compile the assignment of dupWv to the result of the cmplx function.

These type flags do not need to be used except when it needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-58 and **WAVE Reference Type Flags** on page IV-58 for a complete list of type flags and further details.

Details

If /R is omitted, the entire wave is duplicated.

In the range specifications used with /R, a `*` for the end means duplicate to the end. You can also simply leave out the end specification. To include all of a given dimension, use `/R=[]`. If you leave off higher dimensions, all those dimensions are duplicated. That is, `/R=[1,5]` for a 2D wave is equivalent to `/R=[1,5][]`.

Warning: Under some circumstances, such as in loops in user-defined functions, Duplicate may exhibit undesired behavior.

When you use

```
Duplicate/O srcWave, DestWaveName
```

in a user-defined function, it creates a local WAVE variable named *DestWaveName* at compile time. At runtime, if the WAVE variable is NULL, it creates a wave of the same name in the current data folder. If, however, the WAVE variable is not NULL, as it would be in a loop, then the referenced wave will be overwritten no matter where it is located. If the desired behavior is to create (or overwrite) a wave in the current data folder, you should use one of the following two methods:

```
Duplicate/O srcWave, $"DestWaveName"
WAVE DestWaveName   // only if you need to reference dest wave
```

or

```
Duplicate/O srcWave, DestWaveName
// then after you are finished using DestWaveName…
WAVE DestWaveName=$""
```

See Also

The **Rename** operation.

DuplicateDataFolder

DuplicateDataFolder *sourceDataFolderSpec, destDataFolderSpec*

The DuplicateDataFolder operation makes a copy of the source data folder and everything in it and places the copy at the specified location with the specified name.

Parameters

sourceDataFolderSpec can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

destDataFolderSpec can be just a data folder name, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name. If just a data folder name is used then the new data folder is created in the current data folder.

Details

An error is issued if the destination is contained within the source data folder.

Examples

Create a copy of foo named foo2 in bar:

```
DuplicateDataFolder foo,:bar:foo2
```

See Also

See the **MoveDataFolder** operation. Chapter II-8, **Data Folders**.

DWT

DWT [***flags***] ***srcWaveName, destWaveName***

The DWT operation performs discrete wavelet transform on the input wave *srcWaveName*. The operation works on one or more dimensions only as long as the number of elements in each dimension is a power of 2 or when the /P flag is specified

Flags

/D Denoises the source wave. Performs the specified wavelet transform in the forward direction. It then zeros all transform coefficients whose magnitude fall below a given percentage (specified by the /V flag) of the maximum magnitude of the transform. It then performs the inverse transform placing the result in *destWaveName*. The /I flag is incompatible with the /D flag.

/I Perform the inverse wavelet transform. The /S and /D flags are incompatible with the /I flag.

/N=*num* Specifies the number of wavelet coefficients. See /T flag for supported combinations.

/P=*num* *num*=1: Adds zero padding to the end of the dimension up to nearest power of 2 when the number of data elements in a given dimension of *srcWaveName* is not a power of 2.

num=2: Uses zero padding to compute the transform, but the resulting wave is truncated to the length of the input wave.

/S Smooths the source wave. This performs the specified wavelet transform in the forward direction. It then zeros all transform coefficients except those between 0 and the cut-off value (specified in % by /V flag). It then performs the inverse transform placing the result in *destWaveName*. The /I flag is incompatible with the /S flag.

/T=*type* Performs the wavelet transform specified by *type*. The following table gives the transform name with the *type* code for the transform and the allowed values of the *num* parameter used with the /N flag. "NA" means that the /N flag is not applicable to the corresponding transform.

Wavelet Transform	***type***	***num***
Daubechies	1 (default)	4, 6, 8, 10, 12, 20
Haar	2	NA
Battle-Lemarie	4	NA
Burt-Adelson	8	NA

Wavelet Transform	*type*	*num*
Coifman	16	2, 4, 6
Pseudo-Coifman	32	NA
splines	64	1 (2-2), 2 (2-4), 3 (3-3), 4 (3-7)

/V=*value* Specifies the degree of smoothing with the /S and /D flags only.

For /S, *value* gives the cutoff as a percentage of data points above which coefficients are set to zero. For /D, *value* specifies the percentage of the maximum magnitude of the transform such that coefficients smaller than this value are set to zero.

Details

If *destWaveName* exists, DWT overwrites it; if it does not exist, DWT creates it.

When used in a function, the DWT operation automatically creates a wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-56 for details.

If *destWaveName* is not specified, the DWT operation stores the results in W_DWT for 1D waves and M_DWT for higher dimensions.

When working with 1D waves, the transform results are packed such that the higher half of each array contains the detail components and the lower half contains the smooth components and each successive scale is packed in the lower elements. For example, if the source wave contains 128 points then the lowest scale results are stored in elements 64-127, the next scale (power of 2) are stored from 32-63, the following scale from 16-31 etc.

Example

```
Make/O/N=1024 testData=sin(x/100)+gnoise(0.05)
DWT /S/N=20/V=25 testData, smoothedData
```

See Also

For continuous wavelet transforms use the **CWT** operation. See the **FFT** operation.

For further discussion and examples see **Discrete Wavelet Transform** on page III-247.

e

e

The e function returns the base of the natural logarithm system (2.7182818…).

EdgeStats

EdgeStats [***flags***] ***waveName***

The EdgeStats operation produces simple statistics on a region of a wave that is expected to contain a single edge. If more than one edge exists, EdgeStats works on the first one found.

Flags

/A=*avgPts* Determines *startLevel* and *endLevel* automatically by averaging *avgPts* points at centered at *startX* and *endX*. Default is /A=1.

/B=*box* Sets box size for sliding average. This should be an odd number. If /B=*box* is omitted or *box* equals 1, no averaging is done.

/F=*frac* Specifies levels 1, 2 and 3 as a fraction of (*endLevel-startLevel*):

level1 = *frac** (*endLevel-startLevel*) + *startLevel*

level2 = 0.5 * (*endLevel-startLevel*) + *startLevel*

level3 = (1-*frac*) * (*endLevel-startLevel*) + *startLevel*

The default value for *frac* is 0.1 which makes level1 the 10% level, level2 the 50% level and level3 the 90% level.

frac must be between 0 and 0.5.

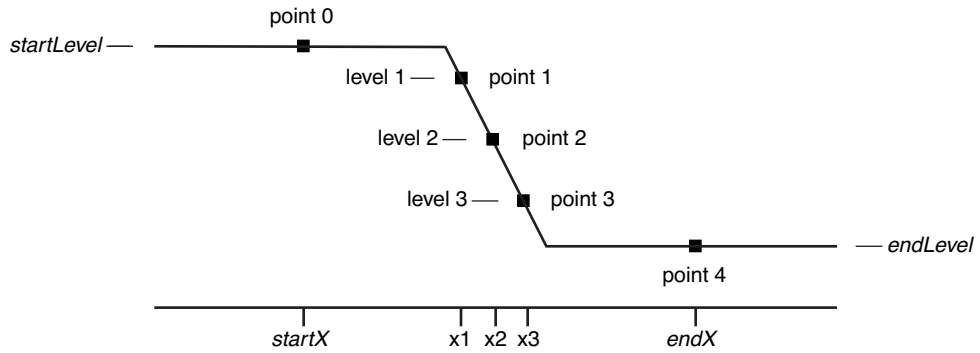
/L=(*startLevel*, *endLevel*)

Sets *startLevel* and *endLevel* explicitly. If omitted, they are determined automatically. See /A.

/P	Output edge locations (see **Details**) are returned as point numbers. If /P is omitted, edge locations are returned as X values.
/Q	Prevents results from being printed in history and prevents error if edge is not found.
/R=(*startX*,*endX*)	Specifies an X range of the wave to search. You may exchange *startX* and *endX* to reverse the search direction.
/R=[*startP*,*endP*]	Specifies a point range of the wave to search. You may exchange *startP* and *endP* to reverse the search direction. If /R is omitted, the entire wave is searched.
/T=*dx*	Forces search in two directions for a possibly more accurate result. *dx* controls where the second search starts.

Details

The /B=*box*, /T=*dx*, /P, and /Q flags behave the same as for the **FindLevel** operation.



EdgeStats considers a region of the input wave between two X locations, called *startX* and *endX*. *startX* and *endX* are set by the /R=(*startX*,*endX*) flag. If this flag is missing, *startX* and *endX* default to the start and end of the entire wave. *startX* can be greater than *endX* so that the search for an edge can proceed from the "right" to the "left".

The diagram above shows the default search direction, from the "left" (lower point numbers) of the wave toward the "right" (higher point numbers).

The *startLevel* and *endLevel* values define the base levels of the edge. You can explicitly set these levels with the /L=(*startLevel*, *endLevel*) flag or you can let EdgeStats find the base levels for you by using the /A=*avgPts* flag which averages points around *startX* and *endX*.

Given *startLevel* and *endLevel* and a *frac* value (see the /F=*frac* flag) EdgeStats defines level1, level2 and level3 as shown in the diagram above. With the default *frac* value of 0.1, level1 is the 10% point, level2 is the 50% point and level3 is the 90% point.

With these levels defined, EdgeStats searches the wave from *startX* to *endX* looking for level2. Having found it, it then searches for level1 and level3. It returns results via variables described below.

EdgeStats sets the following variables:

`V_flag`	0: All three level crossings were found. 1: One or two level crossings were found. 2: No level crossings were found.
`V_EdgeLoc1`	X location of level1.
`V_EdgeLoc2`	X location of level2.
`V_EdgeLoc3`	X location of level3.
`V_EdgeLvl0`	*startLevel* value.
`V_EdgeLvl1`	level1 value.
`V_EdgeLvl2`	level2 value.
`V_EdgeLvl3`	level3 value.
`V_EdgeLvl4`	*endLevel* value.
`V_EdgeAmp4_0`	Edge amplitude (*endLevel - startLevel*).
`V_EdgeDLoc3_1`	Edge width (x distance between point 1 and point 3).

V_EdgeSlope3_1 Edge slope (straight line slope from point 1 and point 3).

These X locations and distances are in terms of the X scaling of the named wave unless you use the /P flag, in which case they are in terms of point number.

The EdgeStats operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

See Also
The **FindLevel** operation for use of the /B=*box*, /T=*dx*, /P and /Q flags, and **PulseStats**.

Edit

Edit [***flags***] [***columnSpec*** [**,** ***columnSpec***]…][**as** ***titleStr***]
The Edit operation creates a table window or subwindow containing the specified columns.

Parameters
columnSpec is usually just the name of a wave. If no *columnSpec*s are given, Edit creates an empty table.

Column specifications are wave names optionally followed by one of the suffixes:

Suffix	Meaning
.i	Index values.
.l	Dimension labels.
.d	Data values.
.id	Index and data values.
.ld	Dimension labels and data values.

If the wave is complex, the wave names may be followed by .real or .imag suffixes. However, as of Igor Pro 3.0, both the real and imaginary columns are added to the table together — you can not add one without the other — so using these suffixes is discouraged.

Historical Note:
Prior to Igor Pro 3.0, only 1D waves were supported. We called index values "X values" and used the suffix ".x" instead of ".i". We called data values "Y values" and used the suffix ".y" instead of ".d". For backward compatibility, Igor accepts ".x" in place of ".i" and ".y" in place of ".d".

titleStr is a string expression containing the table's title. If not specified, Igor will provide one which identifies the columns displayed in the table.

Flags

/HIDE=*h* Hides (h = 1) or shows (h = 0, default) the window.

/HOST=*hcSpec* Embeds the new table in the specified host window or subwindow *hcSpec*.
When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/I Specifies that /W coordinates are in inches.

/K=*k* Specifies window behavior when the user attempts to close it.
k=0: Normal with dialog (default).
k=1: Kills with no dialog.
k=2: Disables killing.
k=3: Hides the window.
If you use /K=2 or /K=3, the only way to kill the window is via the DoWindow/K operation.

/M Specifies that /W coordinates are in centimeters.

/N=*name* Requests that the created table have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen table name. Use `DoWindow/K name` to ensure that *name* is available.

/W=(*left*,*top*,*right*,*bottom*)

Gives the table a specific location and size on the screen. Coordinates for /W are in points unless /I or /M are specified before /W.

When used with the /HOST flag, the specified location coordinates of the sides can have one of two possible meanings:

1) When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.
2) When any value is greater than 1, coordinates are taken to be fixed locations measured in points relative to the top left corner of the host frame.

Details

You can not change dimension index values shown in a table. Use the Change Wave Scaling dialog or the **SetScale** operation.

If /N is not used, Edit automatically assigns to the table window a name of the form "Table*n*", where *n* is some integer. In a function or macro, the assigned name is stored in the S_name string. This is the name you can use to refer to the table from a procedure. Use the **RenameWindow** operation to rename the graph.

Examples

These examples assume that the waves are 1D.

```
Edit myWave,otherWave      // 2 columns: data values from each wave
Edit myWave.id             // 2 columns: x and data values
Edit cmplxWave             // 2 columns: real and imaginary data values
Edit cmplxWave.i           // One column: x values
```

The following examples illustrates the use of column name suffixes in procedures when the name of the wave is in a string variable.

```
Macro TestEdit()
   String w = "wave0"
   Edit $w                 // edit data values
   Edit $w.i               // show index values
   Edit $w.id              // index and data values
End
```

Note that the suffix, if any, must not be stored in the string. In a user-defined function, the syntax would be slightly different:

```
Function TestEditFunction()
   Wave w = $"wave0"
   Edit w                  // no $, because w is name, not string
   Edit w.i                // show index values
   Edit w.id               // index and data values
End
```

See Also

The **DoWindow** operation. For a description of how tables are used, see Chapter II-11, **Tables**.

ei

ei(*x*)

The ei function returns the value of the exponential integral *Ei*(*x*):

$$E(i) = P\int_{-\infty}^{x}\frac{e^t}{t}dt \qquad x>0$$ where *P* denotes the principal value of the integral.

See Also

The **expInt** function.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 228 pp., Dover, New York, 1972.

End

End

The End keyword marks the end of a macro, user function, or user menu definition.

See Also
The **Function** and **Macro** keywords.

EndMacro

EndMacro
The EndMacro keyword marks the end of a macro. You can also use End to end a macro.

See Also
The **Macro** and **Window** keywords.

EndStructure

EndStructure
The EndStructure keyword marks the end of a Structure definition.

See Also
The **Structure** keyword.

endtry

endtry
The endtry flow control keyword defines the end of a try-catch-entry flow control construct.

See Also
The **try-catch-endtry** flow control statement for details.

enoise

enoise(*num* [, *RNG*])
The enoise function returns a random value drawn from a uniform distribution having a range of [-*num*, *num*).

The random number generator is initialized using the system clock when you start Igor, virtually guaranteeing that you will never repeat the same sequence. If you want repeatable "random" numbers, use **SetRandomSeed**.

The optional parameter *RNG* selects one of two different pseudo-random number generators. If omitted, the default is 1. The *RNG*'s are:

RNG	Description
1	Linear Congruential generator by L'Ecuyer with added Bayes-Durham shuffle. The algorithm is described in *Numerical Recipes* as the function `ran2()`. This option has nearly 23^2 distinct values and the sequence of random numbers has a period in excess of 10^{18}.
2	Mersenne Twister by Matsumoto and Nishimura. It is claimed to have better distribution properties and period of 2^{19937}-1.

See Also
The **SetRandomSeed** operation and the **gnoise** function.

Noise Functions on page III-332.

References
Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

Details about the Mersene Twister are in:

Matsumoto, M., and T. Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, *ACM Trans. on Modeling and Computer Simulation*, *8*, 3-30, 1998.

More information is available online at: <http://en.wikipedia.org/wiki/Mersenne_twister>

EqualWaves

EqualWaves(*waveA, waveB, selector* [, *tolerance*])

TheEqualWaves function compares *waveA* to *waveB*. Each wave can be of any data type. It returns 1 for equality and zero otherwise.

Use the *selector* parameter to determine which aspects of the wave are compared. You can add *selector* values to test more than one field at a time or pass -1 to compare all aspects.

selector	Field Compared
1	Wave data
2	Wave data type
4	Wave scaling
8	Data units
16	Dimension units
32	Dimension labels
64	Wave note
128	Wave lock state
256	Data full scale
512	Dimension sizes

If you use the selectors for wave data, wave scaling, dimension units, dimension labels or dimension sizes, EqualWaves will return zero if the waves have unequal dimension sizes. The other selectors do not require equal dimension sizes.

Details

If you are testing for equality of wave data and if the *tolerance* is specified, it must be a positive number. The function returns 1 for equality if the data satisfies:

$$\sum_i \left(waveA[i] - waveB[i]\right)^2 < tolerance.$$

If *tolerance* is not specified, it defaults to 10^{-8}.

If *tolerance* is set to zero and *selector* is set to 1 then the data in the two waves undergo a binary comparison (byte-by-byte).

If *tolerance* is non-zero then the presence of NaNs at a given point in both waves does not contribute to the sum shown in the equation above when both waves contain NaNs at the same point. A NaN entry that is present in only one of the waves is sufficient to flag inequality. Similarly, INF entries are excluded from the tolerance calculation when they appear in both waves at the same position and have the same signs.

If you are comparing wave data (selector =1) and both waves contain zero points the function returns 1.

See Also

The **MatrixOp** operation equal keyword.

erf

erf(*num* [, *accuracy*])

The erf function returns the error function of *num*.

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

Optionally, *accuracy* can be used to specify the desired fractional accuracy.

In complex expressions the error function is

$$erf(z) = \frac{2z}{\sqrt{\pi}}{}_1F_1(\frac{1}{2},\frac{3}{2},-z^2) \text{ where } {}_1F_1(\frac{1}{2},\frac{3}{2},-z^2)$$

is the confluent hypergeometric function of the first kind HyperG1F1. In this case the accuracy parameter is ignored.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of $(f_{actual} - f_{returned})/f_{actual}$ be less than 10^{-7}.

For backwards compatibility, in the absence of *accuracy* an alternate calculation method is used that achieves fractional accuracy better than about $2x10^{-7}$.

If *accuracy* is present, erf can achieve fractional accuracy better than $8x10^{-16}$ for *num* as small as 10^{-3}. For smaller *num* fractional accuracy is better than $5x10^{-15}$.

Higher accuracy takes somewhat longer to calculate. With *accuracy* set to 10^{-16} erfc takes about 50% more time than with *accuracy* set to 10^{-7}.

See Also

The **erfc**, **erfcw**, **dawson**, **inverseErf**, and **inverseErfc** functions.

erfc

erfc(*num* [, *accuracy*])

The erfc function returns the complementary error function of *num* (erfc(x) = 1 - erf(x)). Optionally, *accuracy* can be used to specify the desired fractional accuracy.

In complex expressions the complementary error function is

$$erfc(z) = 1 - erfc(z) = 1 - \frac{2z}{\sqrt{\pi}}{}_1F_1(\frac{1}{2},\frac{3}{2},-z^2) \text{ where } {}_1F_1(\frac{1}{2},\frac{3}{2},-z^2)$$

is the confluent hypergeometric function of the first kind HyperG1F1. In this case the accuracy parameter is ignored.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of $(f_{actual} - f_{returned})/f_{actual}$ be less than 10^{-7}.

For backwards compatibility, in the absence of *accuracy* an alternate calculation method is used that achieves fractional accuracy better than $2x10^{-7}$.

If *accuracy* is present, erfc can achieve fractional accuracy better than $2x10^{-16}$ for *num* up to 1. From *num* = 1 to 10 fractional accuracy is better than $2x10^{-15}$.

Higher accuracy takes somewhat longer to calculate. With *accuracy* set to 10^{-16} erfc takes about 50% more time than with *accuracy* set to 10^{-7}.

See Also

The **erf**, **erfcw**, **inverseErfc**, and **dawson** functions.

erfcw

erfcw(z)

The erfcw is a complex form of the error function defined by

$$\text{erfcw}(z) = \exp[-z^2]\text{erfc}(-iz),$$

where

$$\mathrm{erfc}(z) = \frac{2}{\sqrt{\pi}} \int_{z}^{\infty} \exp[-t^2]dt.$$

The function is computed with accuracy of 0.5e-10. It is particularly useful for large |z| where the computation of erfc(*z*) starts encountering numerical instability.

References

1. http://en.wikipedia.org/wiki/Error_function

2. W. Gautschi, "*Efficient Computation of the Complex Error Function*", SIAM J. Numer. Anal. Vol. 7, No. 1, March 1970.

See Also

The **erf**, **erfc**, **inverseErfc**, and **dawson** functions.

ErrorBars

ErrorBars [***flags***] ***traceName, mode*** [***errorSpecification***]

The ErrorBars operation adds or removes error bars to or from the named trace in the specified graph.

The "error bars" are lines that extend from each data point to "caps". The length of the line (or "bar") is usually used to bracket a measured value by the amount of uncertainty, or "error" in the measurement.

Parameters

traceName is usually the name of a wave. If a wave is displayed more than once in a graph, the instance number can be appended to identify which instance to apply error bars to. For instance, wave0#2 refers to the third instance of wave0 displayed in the top graph (wave0, or wave0#0, is the first instance).

A string containing *traceName* can be used with the $ operator to specify *traceName*.

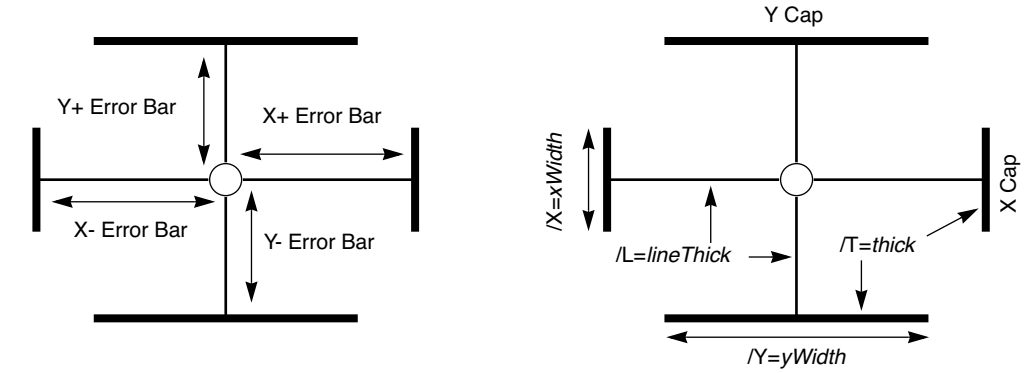
mode is one of the following keywords:

OFF	No error bars.
X	Horizontal error bars only.
Y	Vertical error bars only.
XY	Horizontal and vertical error bars.
BOX	Box error bars.

For any *mode* other than OFF, there is an *errorSpecification* whose format is ***keyword*** [**=** ***value***]. The *errorSpecification* keywords are:

pct	Percent.
sqrt	Square root.
const	Constant.
wave	Arbitrary error values. You can use subranges; see **Subrange Display Syntax** on page II-288.

See the examples for the values that correspond to these *mode* and *errorSpecification* keywords. See the diagram below. *mode* and *errorSpecification* control only the lengths of the horizontal and vertical lines (the "bars") to the "caps". All other sizes and thicknesses are controlled by the flags.



XY *mode* Error Bars

Flags

/L=*lineThick*	Specifies the thickness of both the X and Y error bars drawn from the point on the wave to the caps.
/T=*thick*	Specifies the thickness of both the X and Y error bar "caps".
/W=*winName*	Changes error bars in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.
/X=*xWidth*	Specifies the width (height, actually) of the caps to the left or right of the point.
/Y=*yWidth*	Specifies the width of the caps above or below the point.

The thicknesses and widths are in units of points. The thickness parameters need not be integers. Although only integral thicknesses can be displayed exactly on the screen, nonintegral thicknesses are produced properly on high resolution hard copy devices. Use /T=0 to completely suppress the caps and /L=0 to completely suppress the lines to the caps.

Details

If a point in *traceName* is not within the graph's axes (because the graph has been expanded) then that point's error bars are not shown. If a wave specifying error values for *traceName* is shorter than the wave displayed by *traceName* then the last value of the error wave is used for the unavailable points. If a point in an error wave contains NaN (Not a Number) then the half-bar associated with that point is not shown.

Examples

`ErrorBars wave1,XY pct=10,pct=5`	X and Y error bars, X is 10% of `wave1`, Y is 5%
`ErrorBars wave1,X sqrt`	X error bars only, square root of `wave1`
`ErrorBars wave1,Y const=4.3`	Y error bars only, constant error value = 4.3
`ErrorBars wave1,BOX pct=10,pct=5`	error box, 10% in horizontal direction 5% in vertical direction
`ErrorBars wave1,Y wave=(w1,w2)`	Y error bars only, arbitrary error values wave `w1` = errors for upper (Y+) bars wave `w2` = errors for lower (Y-) bars
`ErrorBars wave1,Y wave=(,w2)`	Y error bars only, no upper (Y+) error bars wave `w2` = errors for lower bars
`ErrorBars wave1,OFF`	turns error bars for wave1 off

Execute

Execute [**/Q/Z**] ***cmdStr***

The Execute operation executes the contents of *cmdStr* as if it had been typed in the command line.

The most common use of Execute is to call a macro or an external operation from a user-defined function. This is necessary because Igor does not allow you to make such calls directly.

When the /Z flag is used, an error code is placed in V_flag.The error code will be -1 if a missing parameter style macro is called and the user clicks Quit Macro, or zero if there was no error.

Flags

/Q Command is not printed in the command line or history area.

/Z Errors are not fatal and error dialogs are suppressed.

Details

Because the command line and command buffer are limited to 400 characters on a single line, *cmdStr* is likewise limited to a maximum of 400 executable characters.

Do not reference local variables in *cmdStr*. The command is not executed in the local environment provided by a macro or user-defined function.

Execute can accept a string expression containing a macro. The string must start with Macro, Proc, or Window, and must follow the normal rules for macros. All lines must be terminated with carriage returns including the last line. The name of the macro is not important but must exist. Errors will be reported except when using the /Z flag, which will assign V_Flag a nonzero number in an error condition.

Examples

It is a good idea to compose the command to be executed in a local string variable and then pass that string to the Execute operation. This prints the string to the history for debugging:

```
String cmd
sprintf cmd, "GBLoadWave/P=%s/S=%d \"%s\"", pathName, skipCount, fileName
Print cmd       // For debugging
Execute cmd
```

Execute with a macro:

```
Execute "Macro junk(a,b)\rvariable a=1,b=2\r\rprint \"hello from macro\",a,b\rEnd\r"
```

See Also

The Execute Operation on page IV-176 for other uses.

Execute/P

Execute/P [**/Q/Z**] ***cmdStr***

Execute/P is similar to Execute except the command string, *cmdStr*, is not immediately executed but rather is posted to an operation queue. Items in the operation queue execute only when nothing else is happening. Macros and functions must not be running and the command line must be empty.

Flags

/Q Command is not printed in the command line or history area.

/Z No error reporting.

See Also

Operation Queue on page IV-250 for more details on using Execute/P with the operation queue.

ExecuteScriptText

ExecuteScriptText [***flags***] ***textStr***

The ExecuteScriptText operation passes your text to Apple's scripting subsystem for compilation and execution or to the Windows command line.

If the /Z flag is used then a variable named V_flag is created and is set to a nonzero value if an error was generated by the script or zero if no error. The error is not reported to Igor if the /Z flag is used.

On Macintosh: Any results, including error messages, are placed in a string variable named S_value. The /B flag is ignored. The /W flag is ignored and ExecuteScriptText does not return until the script is finished.

On Windows: *textStr* contains the name of the executable file with optional Windows-style path and optional arguments:

[*path*]*executableName* [.exe] [*arg1*]…

If *path* is not given and *executableName* ends with ".exe" or with no extension, Igor locates the executable by searching first the registry and then along the PATH environment variable. If not found, then the Igor Pro Folder directory is assumed.

```
ExecuteScriptText "calc"  // calc.exe, calculator
```

When calling a batch file or other non-*.exe file, supply the full path. If the path (or file name) contains spaces put quotes in the string:

```
ExecuteScriptText "\"C:\\Program Files\\my.bat\""
```

If you use the /W=*waitTime* flag with a positive value for *waitTime*, ExecuteScriptText waits up to that many seconds after submitting the command for the process started by the command to terminate. If the process fails to terminate within that period of time, ExecuteScriptText returns an error.

If you omit the /W flag or if you pass zero for *waitTime*, for GUI programs, ExecuteScriptText returns when the program begins processing messages. For non-GUI programs, ExecuteScriptText returns as soon as the OS returns control to Igor after Igor submits the script text to the OS.

Igor currently can not write to a console application's standard input nor read from its standard output.

Use the /B flag to run the command in the background, keeping Igor as the active application.

S_value is always set to "".

Parameters

textStr must contain a valid AppleScript program or Windows command line.

Flags

/B	Execute Windows command line as a background task.
/W=*waitTime*	This flag is accepted on any platform but has an effect only on Windows. See the description above.
/Z	Script errors are not fatal.

Examples

```
// Macintosh: Convert file.PICT to file.GIF:
String ae = "tell application \"clip2gif\" "
ae += "to save file \"HD:file.PICT\"\r"
ExecuteScriptText/Z ae

// Macintosh: Execute a Unix shell command:
Function/S DemoUnixShellCommand()
    // Paths must be POSIX paths (using /).
    // Paths containing spaces or other nonstandard characters must be single-quoted.
    // See Apple Techical Note TN2065 for more on shell scripting via AppleScript.
    String unixCmd
    unixCmd = "ls '/Applications/Igor Pro Folder'"

    String igorCmd

    sprintf igorCmd, "do shell script \"%s\"", unixCmd
    Print igorCmd                       // For debugging only.

       ExecuteScriptText igorCmd
    Print S_value                       // For debugging only.
    return S_value
End

// Windows: Open MatLab in the background:
ExecuteScriptText/B "C:\\Matlab\\bin\\matlab.exe myFile.m"

// Windows: Pass a script to Windows Script Host:
ExecuteScriptText/W=5 "WScript.exe \"C:\\Test Script.vbs\""

// Windows: Execute a batch file and leave the command window open
ExecuteScriptText "cmd.exe /K \"C:\\mybatch.bat\""
```

See Also
See **AppleScript** on page IV-234.

exists

exists(*objNameStr*)

The exists function returns a number which indicates if *objNameStr* contains the name of an Igor object, function or operation.

Details

objNameStr can optionally include a full or partial path to the object. If the name does not include a path, exists checks for waves, strings and variables in the current data folder.

objNameStr can optionally include a module name or independent module name prefix such as "ProcGlobal#" to check for the existence of functions. As of Igor 6.20 this works for macros as well.

The return values are:

0: Name not in use, or does not conflict with a wave, numeric variable or string variable in the specified data folder.
1: Name of a wave in the specified data folder.
2: Name of a numeric or string variable in the specified data folder.
3: Function name.
4: Operation name.
5: Macro name.
6: User-defined function name.

exists is not aware of local variables or parameters in user-defined functions, however it is aware of local variables and parameters in macros.

objNameStr is a string or string expression, *not* a name.

Examples

```
// Prints 2 if V_flag exists as a global variable in the current data folder:
Print exists("V_Flag")

// Prints 5 if a macro named Graph0 exists. Requires Igor Pro 6.20 or later.
Print exists("ProcGlobal#Graph0")
```

See Also

The **DataFolderExists** and **WaveExists** functions and the **WinType** operation.

exp

exp(*num*)

The exp function returns e^{num}. In complex expressions, *num* is complex, and exp(*num*) returns a complex value.

ExperimentModified

ExperimentModified [*newModifiedState*]

The ExperimentModified operation gets and optionally sets the modified (save) state of the current experiment.

Use this command to prevent Igor from asking you to save the current experiment after you have made changes you do not need to save or, conversely, to force Igor to ask about saving the experiment even though Igor would not normally do so.

The variable V_flag is always set to the experiment-modified state that was in effect before the ExperimentModified command executed: 1 for modified, 0 for not modified.

Parameters

If *newModifiedState* is present, it sets the experiment-modified state as follows:

newModifiedState = 0: Igor will not ask to save the experiment before quitting or opening another experiment, and the Save Experiment menu item will be disabled.

newModifiedState = 1: Igor will ask to save the experiment before quitting or opening another experiment, and the Save Experiment menu item will be enabled.

If *newModifiedState* is omitted, the state of experiment-modified state is not changed.

Details

Executing ExperimentModified 0 on the command line will not work because the command will be echoed to the history area, marking the experiment as modifed. Use the command in a function or macro that does not echo text to the history area.

Examples

The /Q flag is vital: it suppresses printing into the history area which would mark the experiment as modified again.

```
Menu "File"
    "Mark Experiment Modified",/Q,ExperimentModified 1    // Enables "Save Experiment"
    "Mark Experiment Saved",/Q,ExperimentModified 0       // Disables "Save Experiment"
End
```

See Also

The **SaveExperiment** operation, **Menu Definition Syntax** on page IV-107.

expInt

expInt(*n, x*)

The expInt function returns the value of the exponential integral $E_n(x)$:

$$E_n(x) = P\int_1^\infty \frac{e^{-xt}}{t^n}dt \qquad x > 0;\ n = 0, 1, 2, \ldots$$

See Also

The **ei** function.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

expnoise

expnoise(*b*)

The expnoise function returns a pseudo-random value from an exponential distribution whose average and standard deviation are *b* and the probability distribution function is

$$f(x) = \frac{1}{b}\exp\left(-\frac{x}{b}\right).$$

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-332.

Chapter III-12, **Statistics** for a function and operation overview.

Extract

Extract [*type flags*] [**/INDX/O**] *srcWave, destWave, LogicalExpression*

The Extract operation finds data in *srcWave* wherever *LogicalExpression* evaluates to TRUE and stores the matching data sequentially in *destWave*, which will be created if it does not already exist.

Parameters

srcWave is the name of a wave.

destWave is the name of a new or existing wave that will contain the result.

LogicalExpression can use any comparison or logical operator in the expression.

Flags

/FREE Creates a free destWave (see **Free Waves** on page IV-71).

/FREE is allowed only in functions and only if a simple name or structure field is specified for destWave.

Requires Igor Pro 6.1 or later. For advanced programmers only.

/INDX Stores the index in *destWave* instead of data from *srcWave*.

/O Overwrites *destWave*.

Type Flags *(used only in functions)*

Extract also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when it needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-58 and **WAVE Reference Type Flags** on page IV-58 for a complete list of type flags and further details.

Details

srcWave may be of any type including text.

destWave has the same type as *srcWave*, but it is always one dimensional. With /INDX, the *destWave* type is set to unsigned 32-bit integer and the values represent a linear index into *srcWave* regardless of its dimensionality.

Example

```
Make/O source= x
Extract/O source,dest,source>10 && source<20
print dest
```

Prints the following to the history area:

```
  dest[0]= {11,12,13,14,15,16,17,18,19}
```

See Also

The **Duplicate** operation.

factorial

factorial(*n*)

The factorial function returns *n*!, where *n* is assumed to be a positive integer.

Note that while factorial is an integer-valued function, a double-precision number has 53 bits for the mantissa. This means that numbers over 2^{52} will be accurate to about one part in about $2x10^{16}$. Values of *n* greater than 170 result in overflow and return Inf.

FakeData

FakeData(*waveName*)

The FakeData function puts fake data in the named wave, which must be single-precision float. This is useful for testing things that require changing data before you have the source for the eventual real data. FakeData can be useful in a background task expression.

The FakeData function is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

Examples

```
Make/N=200 wave0; Display wave0
SetBackground FakeData(wave0)        // define background task
CtrlBackground period=60, start      // start background task
// observe the graph for a while
CtrlBackground stop                  // stop the background task
```

FastGaussTransform

FastGaussTransform [*flags*] *srcLocationsWave, srcWeightsWave*

The FastGaussTransform operation implements an efficient algorithm for evaluating the discrete Gauss transform, which is given by

$$\boldsymbol{G}(\boldsymbol{y}) = \sum_{i=1}^{n} \boldsymbol{q}_i \exp\left[-\frac{\|\boldsymbol{y}-\boldsymbol{x}_i\|^2}{h}\right]$$

where $\boldsymbol{G}$ is an M-dimensional vector, $\boldsymbol{y}$ is an N-dimensional vector representing the observation position, $\{\boldsymbol{q}_i\}$ are the M-dimensional weights, $\{\boldsymbol{x}_i\}$ are N-dimensional vectors representing source locations, and h is the Gaussian width. The wave M_FGT contains the output in the current data folder.

Flags

/AERR=*aprxErr* Sets the approximate error, which determines how many terms of the Taylor expansion of the Gaussian are used by the calculation. Default value is 1e-5.

/WDTH=*h* Sets the Gaussian width. Default value is 1.

/OUTW=*locWave* Specifies the locations at which the output is computed. *locWave* must have the same number of columns as *srcLocationsWave*. The other /OUT flags are mutually exclusive; you should use only one at any time.

/OUT1={*x1*,*nx*,*x2*}
/OUT2={*x1*,*nx*,*x2*,*y1*,*ny*,*y2*}
/OUT3={*x1*,*nx*,*x2*,*y1*,*ny*,*y2*,*z1*,*nz*,*z2*}
Specifies gridded output of the required dimension. In each case you set the starting and ending values together with the number of intervals in that dimension. You cannot specify an output that does not match the dimensions of the input source.

/Q No results printed in the history area.

/RX=*rx* Sets the maximum radius of any cluster. The clustering algorithm terminates when the maximum radius is less than *rx*. Without /RX, the maximum radius is the same as the maximum radius encountered.

/RY=*ry* Sets the upper bound for the distance between an observation point and a cluster center for which the cluster contributes to the transform value. Default is 5*h*.

/TET=*nTerms* Sets the number of terms in the Taylor expansion. Use /TET to set the number of terms and bypass the default error estimate, which is estimated from the approximate error value (/AERR).

/Z No error reporting.

Details

The discrete Gauss transform can be computed as a direct sum. An exact calculation is practical only for moderate number of sources and observation points and for low spatial dimensionality. With increasing dimensionality and increasing number of sources it is more efficient to take advantage of some properties of the Gaussian function. The FastGaussianTransform operation does so in two ways: It first arranges the sources in N-dimensional spatial clusters so that it is not necessary to compute the contributions of all source points that belong to remote clusters (see **FPClustering**). The second component of the algorithm is an approximation that factorizes the sum into a factor that depends only on source points and a factor that depends only on observation points. The factor that depends only on source points is computed only once while the factor that depends on observation points is evaluated once for each observation point.

The trade-off between computation efficiency and accuracy can be adjusted using multiple parameters. By default, the operation calculates the number of terms it needs to use in the Taylor expansion of the Gaussian. You can modify the default approximate error value using /AERR or you can directly set the number of terms in the expansion using /TET.

FastGaussianTransform supports calculations in dimensions that may exceed the maximum allowed wave dimensionality. *srcLocationsWave* must be a 2D, real-valued single- or double-precision wave in which each row corresponds to a single source position and columns represent the components in each dimension (e.g., a triplet wave would represent 3D source locations). *srcWeightsWave* must have the same number of rows as *srcLocationsWave* and it must be a real-valued single- or double-precision wave. In most applications *srcWeightsWave* will have a single column so that the output $\boldsymbol{G}$ will be scalar. However, if *srcWeightsWave* has multiple columns than $\boldsymbol{G}$ is a vector. This can be handy if you need to test multiple sets of coefficients at one time. If you specify observation points using /OUTW then *locWave* must have the same number of columns as *srcLocationsWave* (the number of rows in the output is arbitrary). The operation does not support wave scaling.

See Also

The **CWT**, **FFT**, **ImageInterpolate**, **Loess**, and **FPClustering** operations.

References

Yang, C., R. Duraiswami, and L. Davis, Efficient Kernel Machines Using the Improved Fast Gauss Transform, *Advances in Neural Information Processing Systems 16*, 2004.

FastOp

FastOp [**/C**] ***destwave*** **=** ***prod1*** [**±** ***prod2*** [**±** ***prod3***]]

The FastOp operation can be used to get improved speed out of certain wave assignment statements. The syntax was designed so that you can simply insert `FastOp` in front of any wave assignment statement that meets the syntax requirements.

Parameters

destWave An existing destination wave for the assignment expression. An error will be reported at runtime if the waves are not all the same length or number type.

prod1, *prod2*, *prod3* Products with the following formats:

*constexpr*wave1*wave2*

or

*constexpr*wave1/wave2*

constexpr may be a literal numeric constant or a constant expression in parentheses. Such expressions are evaluated only once.

Any component in a prod expression may be omitted.

Flags

/C Specifies a complex expression. Only applicable to floating point waves.

Details

Certain combinations are evaluated using faster specific code rather than more general but slower generic code. The following specific formats are given special consideration:

Single or Double Precision Real	Integer
waved = C0	*waved = C0*
*waved = C0 *waveA +C1*	*waved = waveA +C1*
waved = waveA +C1	*waved = waveA +waveB +C2*

In the above, pluses may be minuses and the trailing constant (*C0, C1, C2*) may be omitted.

Note: Integer waves are evaluated using double precision intermediate values except for the aforementioned special cases which are evaluated using the native format.

Typically, FastOp will improve performance by 10 to 40 times. The speed increase will be dependent on the computer and on the length of the waves, with the greatest improvement for waves having 1000 to 100,000 points.

This operation replaces the obsolete FastWaveOps XOP. It has all the capabilities of the XOP and then some and has an easier to read syntax.

Examples

Valid expressions:

```
FastOp waved= 3
FastOp waved= waveA + waveB
FastOp waved= 0.5*waveA + 0.5*waveB
FastOp waved= waveA*waveB
FastOp waved= (2*3)*waveA + 6
FastOp waved= (locvar)*waveA
```

Expressions that are **not** valid:

```
FastOp waved= 3*4
FastOp waved= (waveA + waveB)
FastOp waved= waveA*0.5 + 0.5*waveB
FastOp waved= waveA*waveB/2
FastOp waved= 2*3*waveA + 6
FastOp waved= locvar*waveA
```

See Also
The **MatrixOp** operation for more efficient matrix operations.

faverage

faverage(*waveName* [, *x1, x2*])

The faverage function returns the trapezoidal average value of the named wave from x=*x1* to x=*x2*.

If your data are in the form of an XY pair of waves, see **faverageXY**.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If *x1* or *x2* are not within the X range of *waveName*, faverage limits them to the nearest X range limit of *waveName*.

faverage returns the area divided by (*x2*-*x1*). In other words, the X scaling of *waveName* is eliminated when computing the average.

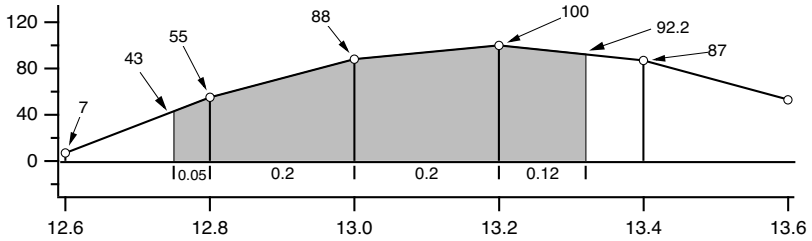
If any Y values in the specified X range are NaN, faverage returns NaN.

Unlike the **area** function, reversing the order of *x1* and *x2* does *not* change the sign of the returned value.

The faverage function is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

Examples

Comparison of area, faverage and mean functions over interval (12.75,13.32)



area(wave,12.75,13.32)	= 0.05 · (43+55) / 2	// first trapezoid
	+ 0.20 · (55+88) / 2	// second trapezoid
	+ 0.20 · (88+100) / 2	// third trapezoid
	+ 0.12 · (100+92.2) / 2	// fourth trapezoid
	= 47.082	
faverage(wave,12.75,13.32)	= area(wave,12.75,13.32) / (13.32-12.75)	
	= 47.082/0.57 = 82.6	
mean(wave,12.75,13.32)	= (55+88+100+87)/4 = 82.5	

See Also
Integrate operation and **area** function.

faverageXY

faverageXY(*XWaveName, YWaveName* [, *x1, x2*])

The faverageXY function returns the trapezoidal average value of *YWaveName* from x=*x1* to x=*x2*, using X values from *XWaveName*.

This function operates identically to faverage, except that it uses an XY pair of waves for X and Y values.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If *x1* or *x2* are not within the X range of *XWaveName*, faverageXY limits them to the nearest X range limit of *XWaveName*.

faverageXY returns the area divided by (*x2* -*x1*).

If any values in the X range are NaN, faverageXY returns NaN.

Reversing the order of *x1* and *x2* does not change the sign of the returned value.

The values in *XWaveName* may be increasing or decreasing. faverageXY assumes that the values in *XWaveName* are monotonic. If they are not monotonic, Igor does not complain, but the result is not meaningful. If any X values are NaN, the result is NaN.

The faverageXY function is not multidimensional aware. See Chapter II-6, **Multidimensional Waves** for details on multidimensional waves, particularly **Analysis on Multidimensional Waves** on page II-110.

See Also
See **faverage**, **areaXY**, **area**, and **PolygonArea**.

FBinRead

FBinRead [***flags***] ***refNum, objectName***

The FBinRead operation reads binary data from the file specified by *refNum* into the named object.

Parameters
refNum is a file reference number from the Open operation used to open the file.

objectName is the name of a wave, numeric variable, string variable, or structure.

Flags

/B[=*b*] Specifies file byte ordering.
- *b*=0: Native (same as no /B).
- *b*=1: Reversed (same as /B for compatibility with Igor Pro 3.0).
- *b*=2: Big-endian (Motorola/Macintosh).
- *b*=3: Little-endian (Intel/Windows).

/F=*f* Controls the number of bytes read and how the bytes are interpreted.
- *f*=0: Native binary format of the object (default).
- *f*=1: Signed one-byte integer.
- *f*=2: Signed 16-bit word; two bytes.
- *f*=3: Signed 32-bit word; four bytes.
- *f*=4: 32-bit IEEE floating point.
- *f*=5: 64-bit IEEE floating point.

/U Integer formats (/F=1, 2, or 3) are unsigned. If /U is omitted, integers are signed.

Details
If *objectName* is the name of a string variable then /F doesn't apply. The number of bytes read is the number of bytes in the string *before* the FBinRead operation is called. You can use the **PadString** function to set the size of a string.

The binary format that FBinRead uses for numeric variables or waves depends on the /F flag. If no /F flag is present, the native binary format of the named object is used.

Byte ordering refers to the order in which a multibyte datum is read from a file. For example, a 16-bit word (sometimes called a "short") consists of a high-order byte and a low-order byte. Under big-endian byte ordering, which is commonly used on Macintosh, the high-order byte is read from the file first. Under little-endian byte ordering, which is commonly used on Windows, the low-order byte is read from the file first.

FBinRead will read an entire structure from a disk file. The individual fields of the structure will be byte-swapped if the /B flag is designated.

The FBinRead operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

See Also
The **Open** operation. See **FSetPos** to set positions within files and **FStatus** for file information.

FBinWrite

FBinWrite [***flags***] ***refNum, objectName***

The FBinWrite operation writes the named object in binary to a file.

Parameters

refNum is a file reference number from the **Open** operation used to open the file.

objectName is the name of a wave, numeric variable, string variable, or structure.

Flags

/B[=*b*] Specifies file byte ordering.
- *b*=0: Native (same as no /B).
- *b*=1: Reversed (same as /B for compatibility with Igor Pro 3.0).
- *b*=2: Big-endian (Motorola/Macintosh).
- *b*=3: Little-endian (Intel/Windows).

/F=*f* Controls the number of bytes written and how the bytes are formatted.
- *f*=0: Native binary format of the object (default).
- *f*=1: Signed byte; one byte.
- *f*=2: Signed 16-bit word; two bytes.
- *f*=3: Signed 32-bit word; four bytes.
- *f*=4: 32-bit IEEE floating point.
- *f*=5: 64-bit IEEE floating point.

/P Adds an IgorBinPacket to the data. This is used for PPC or Apple event result packets (*refNum* = 0) and is not normally of use when writing to a file.

/U Integer formats (/F=1, 2, or 3) are unsigned. If /U is omitted, integers are signed.

Details

A zero value of *refNum* is used in conjunction with Program-to-Program Communication (PPC) or Apple events (*Macintosh*) or DDE (*Windows*). The data that would normally be written to a file is appended to the PPC or Apple event or DDE result packet.

If the object is a string variable then /F doesn't apply. The number of bytes written is the number of bytes in the string.

The binary format that FBinWrite uses for numeric variables or waves depends on the /F flag. If no /F flag is present, FBinWrite uses the native binary format of the named object.

Byte ordering refers to the order in which a multibyte datum is written to a file. For example, a 16-bit word (sometimes called a "short") consists of a high-order byte and a low-order byte. Under big-endian byte ordering, which is commonly used on Macintosh, the high-order byte is written to the file first. Under little-endian byte ordering, which is commonly used on Windows, the low-order byte is written to the file first.

FBinWrite will write an entire structure to a disk file. The individual fields of the structure will be byte-swapped if the /B flag is designated.

The FBinWrite operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

See Also

The **Open** operation and Chapter IV-10, **Advanced Programming**. See **FSetPos** to set positions within files and **FStatus** for file information.

FetchURL

FetchURL(*urlStr*)

The FetchURL function returns a string containing the server's response to a request to get the contents of the URL specified by *urlStr*. If *urlStr* contains a URL that uses the file:// scheme, the contents of the local file is returned.

Parameters

urlStr is a string containing the URL to retrieve. You can include a username, password, and server port number as part of the URL.

FetchURL expects that *urlStr* has been percent-encoded if it contains reserved characters. See **Percent Encoding** on page IV-239 for additional information on when percent-encoding is necessary and how to do it.

See **URLs** on page IV-239 for details about how to correctly specify the URL.

FetchURL supports only the http://, ftp://, and file:// schemes. See **Supported Network Schemes** on page IV-239 for details.

There are two special values of *urlStr* that can be used to get information about the network library that Igor uses. The keyword=value pairs returned when *urlStr* is "curl_version_info" may be useful to programmers in that the features and protocols available in the library are specified.

```
FetchURL("curl_version")
FetchURL("curl_version_info")
```

Details

If FetchURL encounters an error, the string it returns a NULL string. You should check for errors before using the returned string. In a user-defined function, use the **GetRTError** function.

```
String urlStr = "http://www.badserver"
String response = FetchURL(urlStr)
Variable error = GetRTError(1)       // Check for error before using response
if (error != 0)
    // FetchURL produced an error
    // so don't try to use the response.
endif
```

Limitations

It is possible for FetchURL to return a valid server response even though the URL you requested does not exist on the server or requires a username and password that you did not provide. In this situation, the response returned by the server will usually be a web page stating that the page was not found or another error message. You can check for this kind of error in your own code by examining the response.

FetchURL does not support advanced features such as network proxies, file or data uploads, or saving the server's response directly to a file. When using the http:// scheme, only the GET method is supported. This means that you cannot use FetchURL to submit form data to a web server that requires using the http POST method.

Encrypted connections using Secure Sockets Layer (SSL) using the https:// scheme are not supported.

Igor Pro is not capable of displaying the contents of a URL in a rendered form like a web browser.

Examples

```
// Retrieve the contents of the WaveMetrics home page.
String response
response = FetchURL("http://www.wavemetrics.com")

// Get a binary image file from a web server and then
// save the image to a file on the desktop.
String url = "http://www.wavemetrics.net/images/tbg.gif"
String imageBytes = FetchURL(url)
Variable error = GetRTError(1)
if (error != 0)
    Print "Error downloading image."
else
    Variable refNum
    String localPath = SpecialDirPath("Desktop", 0, 0, 0) + "tbg.gif"
    Open/T=".gif" refNum as localPath
    FBinWrite refNum, imageBytes
    Close refNum
endif
```

See Also

FTPDownload, **URLEncode**

Network Communications on page IV-238, **Network Connections From Multiple Threads** on page IV-242.

FFT

FFT [*flags*] *srcWave*

The FFT operation computes the Discrete Fourier Transform of *srcWave* using a multidimensional prime factor decomposition algorithm. By default, *srcWave* is overwritten by the FFT.

Output Wave Name

For compatibility with earlier versions of Igor, if you use FFT with no flags or with just the /Z flag, the operation overwrites *srcWave*.

As of Igor Pro 5, if you use any flag other than /Z, FFT uses default output wave names: W_FFT for a 1D FFT and M_FFT for a multidimensional FFT.

We recommend that you use the /DEST flag to make the output wave explicit and to prevent overwriting *srcWave*.

Flags

/COLS Computes the 1D FFT of 2D *srcWave* one column at a time, storing the results in the destination wave.

$$I[t_1][n] = \sum_{k=0}^{N-1} f[t_1][k]\exp\left(i2\pi kn / N\right).$$

You must specify a destination wave using the /DEST flag. No other flags are allowed with this flag. The number of rows must be even. If *srcWave* is a real (NxM) wave, the output matrix will be (1+N/2,M) in analogy with 1D FFT. To avoid changes in the number of points you can convert *srcWave* to complex data type. This flag applies only to 2D source waves. See also the /ROWS flag.

/DEST=*destWave* Specifies the output wave created by the FFT operation.

It is an error to attempt specify the same wave as both *srcWave* and *destWave*.

The default output wave name is W_FFT for a 1D FFT and M_FFT for a multidimensional FFT.

When used in a function, the FFT operation by default creates a complex wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-56 for details.

/HCC Hypercomplex transform (cosine). Computes the integral

$$I_c(\omega_1,\omega_2) = \int_{-\infty}^{\infty}\int f(t_1,t_2)\cos(t_1\omega_1)\exp(it_2\omega_2)dt_1dt_2$$

using the 2D FFT (see **Details**).

/HCS Hypercomplex transform (sine). Computes the integral

$$I_s(\omega_1,\omega_2) = \int_{-\infty}^{\infty}\int f(t_1,t_2)\sin(t_1\omega_1)\exp(it_2\omega_2)dt_1dt_2$$

using the 2D FFT (see **Details**).

/MAG Saves just the magnitude of the FFT in the output wave. See comments under /OUT.

/MAGS Saves the squared magnitude of the FFT in the output wave. See comments under /OUT.

/OUT=*mode* Sets the output wave format.

mode=1: Default for complex output.

mode=2: Real output.

mode=3: Magnitude.

mode=4: Magnitude square.

mode=5: Phase.

mode=6: Scaled magnitude.

mode=7: Scaled magnitude squared.

You can also identify modes 2-4 using the convenience flags /REAL, /MAG, and /MAGS. The convenience flags are mutually exclusive and are overridden by the /OUT flag.

The scaled quantities apply to transforms of real valued inputs where the output is normally folded in the first dimension (because of symmetry). The scaling applies a factor of 2 to the squared magnitude of all components except the DC. The scaled transforms should be used whenever Parseval's relation is expected to hold.

/PAD={*dim1* [, *dim2*, *dim3*, *dim4*]}

Converts *srcWave* into a padded wave of dimensions *dim1*, *dim2*…. The padded wave contains the original data at the start of the dimension and adds zero entries to each dimension up to the specified dimension size. The *dim1*… values must be greater than or equal to the corresponding dimension size of *srcWave*. If you need to pad just the lowest dimension(s) you can omit the remaining dimensions; for example, /Pad=*dim1* will set *dim2* and above to match the dimensions in *srcWave*.

/REAL Saves just the real part of the transform in the output wave. See comments under /OUT.

/ROWS Calculates the FFT of only the first dimension of a 2D *srcWave*. It thus computes the 1D FFT of one row at a time, storing the results in the destination wave.

$$N[n][t_2] = \sum_{k=0}^{M-1} f[k][t_2]\exp(i2\pi kn / M)$$

You must specify a destination wave using the /DEST flag. No other flags are allowed with this flag. The number of columns must be even. If *srcWave* is a real (NxM) wave, the output matrix will be (N,1+M/2) in analogy with 1D FFT. To avoid changes in the number of points you can convert *srcWave* to complex data type. See also /COLS flag.

/RP=[*startPoint*, *endPoint*]

/RX=(*startX*, *endX*) Defines a segment of a 1D *srcWave* that will be transformed. By default the operation transforms the whole wave. It is sometimes useful to take advantage of this feature in order to transform just the defined interval, which includes both end points. You can define the interval using wave point indexing with the /RP flag or using the X-values with the /RX flag. The interval must include at least four data points and the total number of points must be an even number.

/WINF=*windowKind*

Premultiplies a 1D *srcWave* with the selected window function.

In the following window definitions, $w(n)$ is the value of the window function that multiplies the signal, N is the number of points in the signal wave (or range if /R is specified), and n is the wave point index. With /R, n=0 for the first datum in the range.

Choices for *windowKind* are in bold (some windows, in italics, have multiple options for different constant values):

Bartlet: a synonym for Bartlett.

Bartlett: $$w(n) = \begin{cases} \dfrac{2n}{N} & n = 0,1,...\dfrac{N}{2} \\ 2 - \dfrac{2n}{N} & n = \dfrac{N}{2},...N-1 \end{cases}$$

Blackman: $w(n) = a_0 - a_1 \cos\left(\frac{2\pi}{N} n\right) + a_2 \cos\left(\frac{2\pi}{N} 2n\right) - a_3 \cos\left(\frac{2\pi}{N} 3n\right)$.

$n = 0,1,2...N-1.$

windowKind	a_0	a_1	a_2	a_3
Blackman367	0.42323	0.49755	0.07922	
Blackman361	0.44959	0.49364	0.05677	
Blackman492	0.35875	0.48829	0.14128	0.01168
Blackman474	0.40217	0.49703	0.09392	0.00183

Cos:

$$w(n) = \cos\left(\frac{n}{N}\pi\right)^{\alpha},$$

$$n = -\frac{N}{2},...,-1,0,1,...,\frac{N}{2}.$$

Cos1: $\alpha = 1.$
Cos2: $\alpha = 2.$
Cos3: $\alpha = 3.$
Cos4: $\alpha = 4.$

Hamming: $$w(n) = \begin{cases} 0.54 + 0.46\cos\left(\frac{2\pi n}{N}\right) & n = -\frac{N}{2},...,-1,0,1,...,\frac{N}{2} \\ 0.54 - 0.46\cos\left(\frac{2\pi n}{N}\right) & n = 0,1,2,...,N-1 \end{cases}$$

Hanning: $$w(n) = \begin{cases} \frac{1}{2}\left[1 + \cos\left(\frac{2\pi n}{N}\right)\right] & n = -\frac{N}{2},...,-1,0,1,...,\frac{N}{2} \\ \frac{1}{2}\left[1 - \cos\left(\frac{2\pi n}{N}\right)\right] & n = 0,1,2,...,N-1 \end{cases}$$

Kaiser Bessel: $$w(n) = \frac{I_0\left(\pi\alpha\sqrt{1-\left(\frac{2n}{N}\right)^2}\right)}{I_0(\pi\alpha)} \quad 0 \le |n| \le \frac{N}{2}.$$

where I_0 is the zero-order modified Bessel function of the first kind.

KaiserBessel20: $\alpha = 2.$
KaiserBessel25: $\alpha = 2.5.$
KaiserBessel30: $\alpha = 3.$

Parzen: $$w(n) = 1 - \left|\frac{2n}{N}\right|^2 \quad 0 \le |n| \le \frac{N}{2}.$$

Poisson: $$w(n) = \exp\left(-\alpha \frac{2|n|}{N}\right) \qquad 0 \le |n| \le \frac{N}{2}.$$

Poisson2: $\alpha = 2$.
Poisson3: $\alpha = 3$.
Poisson4: $\alpha = 4$.

Riemann: $$w(n) = \frac{\sin\left(\frac{2\pi n}{N}\right)}{\left(\frac{2\pi n}{N}\right)} \qquad 0 \le |n| \le \frac{N}{2}.$$

/Z Disables rotation of the FFT of a complex wave. Igor normally rotates the FFT result (which is also complex) by N/2 so that x=0 is at the center point (N/2). When /Z is specified, Igor does not perform this rotation and leaves x=0 at the first point (0).

Details

The data type of *srcWave* is arbitrary. The first dimension of *srcWave* must be an even number and the minimum length of *srcWave* is four points. When *srcWave* is a double precision wave, the FFT is computed in double precision. All other data types are transformed using single precision calculations. The result of the FFT operation is always a floating point number (single or double precision).

Depending on your choice of outputs, you may not be able to invert the transform in order to obtain the original *srcWave*.

srcWave or any of it's intervals must have at least four data points and must not contain NaNs or INFs.

The FFT algorithm is based on prime number decomposition, which decomposes the number of points in each dimension of the wave into a product of prime numbers. The FFT is optimized for primes < 5. In time consuming applications it is frequently worthwhile to pad the data so that the total number of points factors into small prime numbers.

The hypercomplex transforms are computed by writing the sine and cosine as a sum of two exponentials. Let the 2D Fourier transform of the input signal be

$$F[n_1][n_2] = \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} f[k_1][k_2] \exp\left(i2\pi k_1 \frac{n_1}{N_1}\right) \exp\left(i2\pi k_2 \frac{n_2}{N_2}\right)$$

then the two hypercomplex transforms are given by

$$I_c[n_1][n_2] = \frac{1}{2}\left(F[n_1][n2] + F[-n_1][n2]\right)$$

and

$$I_s[n_1][n_2] = \frac{1}{2i}\left(F[n_1][n2] - F[-n_1][n2]\right)$$

See Also

See **Fourier Transforms** on page III-235 for discussion. The inverse operation is **IFFT**.

Spectral Windowing on page III-240. For 2D windowing see **ImageWindow**. Also the **Hanning** window operation.

Also see the **DWT** operation for the discrete wavelet transform and the **CWT** operation for the continuous wavelet transform. The **HilbertTransform** and **WignerTransform** operations.

The **Unwrap**, **MatrixOp**, **DSPPeriodogram** and **LombPeriodogram** operations.

References

For more information about the use of window functions see:

Harris, F.J., On the use of windows for harmonic analysis with the discrete Fourier Transform, *Proc, IEEE*, *66*, 51-83, 1978.

FIFO2Wave

FIFO2Wave [**/R/S**] ***FIFOName, channelName, waveName***

The FIFO2Wave operation copies FIFO data from the specified channel of the named FIFO into the named wave. FIFOs are used for data acquisition.

Flags

/R=[*startPoint*,*endPoint*] Dumps the specified FIFO points into the wave.

/S=*s*
- *s*=0: Same as no /S.
- *s*=1: Sets the wave's X scaling x_0 value to the number of the first sample in the FIFO.
- *s*=2: Changes the wave's number type to match the FIFO channel's type.
- *s*=3: Combination of *s*=1 and *s*=2.

Details

The FIFO must be in the valid state for FIFO2Wave to work. When you create a FIFO, using NewFIFO, it is initially invalid. It becomes valid when you issue the start command via the CtrlFIFO operation. It remains valid until you change a FIFO parameter using CtrlFIFO.

If you specify a range of FIFO data points, using /R=[*startPoint*,*endPoint*] then FIFO2Wave dumps the specified FIFO points into the wave after clipping *startPoint* and *endPoint* to valid point numbers.

The valid point numbers depend on whether the FIFO is running and on whether or not it is attached to a file. If the FIFO is running then *startPoint* and *endPoint* are truncated to number of points in the FIFO. If the FIFO is buffering a file then the range can include the full extent of the file.

If you specify no range then FIFO2Wave transfers the most recently acquired FIFO data to the wave. The number of points transferred is the smaller of the number of points in the FIFO and number of points in the wave.

FIFO2Wave may or may not change the wave's X scaling and number type, depending on the current X scaling and on the /S flag.

Think of the wave's X scaling as being controlled by two values, x_0 and dx, where the X value of point p is x_0 + p*dx. FIFO2Wave always sets the wave's dx value equal to the FIFO's deltaT value (as set by the CtrlFIFO operation). If you use no /S flag, FIFO2Wave does not set the wave's x_0 value nor does it set the wave's number type.

If you are using FIFO2Wave to update a wave in a graph as quickly as possible, the /S=0 flag gives the highest update rate. The other /S values trigger more recalculation and slow down the updating.

If the wave's number type (possibly changed to match the FIFO channel) is a floating point type, FIFO2Wave scales the FIFO data before transferring it to the wave as follows:

```
scaled_value = (FIFO_value - offset) * gain
```

If the FIFO channel's gain is one and its offset is zero, the scaling would have no effect so FIFO2Wave skips it.

If the specified FIFO channel is an image strip channel (one defined using the optional vectPnts parameter to NewFIFOChan), then the resultant wave will be a matrix with the number of rows set by vectPnts and the number of columns set by the number of points described above for one-dimensional waves. To create an image plot that looks the same as the corresponding channel in a Chart, you will need to transpose the wave using **MatrixTranspose**.

See Also

The **NewFIFO** and **CtrlFIFO** operations, and **FIFOs and Charts** on page IV-276 for more information on FIFOs and data acquisition. For an explanation of waves and wave scaling, see **Changing Dimension and Data Scaling** on page II-83.

FIFOStatus

FIFOStatus [/Q] ***FIFOName***

The FIFOStatus operation returns miscellaneous information about a FIFO and its channels. FIFOs are used for data acquisition.

Flags

/Q Doesn't print in the history area.

Details

FIFOStatus sets the variable V_flag to nonzero if a FIFO of the given name exists. If the named FIFO does exist then FIFOStatus stores information about the FIFO in the following variables:

V_FIFORunning	Nonzero if FIFO is running.
V_FIFOChunks	Number of chunks of data placed in FIFO so far.
V_FIFOnchans	Number of channels in the FIFO.
S_Info	Keyword-packed information string.

The keyword-packed information string consists of a sequence of sections with the following form: *keyword*:*value*;

You can pick a value out of a keyword-packed string using the **NumberByKey** and **StringByKey** functions. Here are the keywords for S_Info:

Keyword	Type	Meaning
DATE	Number	The date/time when start was issued via CtrlFIFO.
DELTAT	Number	The FIFO's deltaT value as set by CtrlFIFO.
DISKTOT	Number	Current number of chunks written to the FIFO's file.
FILENUM	Number	The output file refNum or review file refNum as set by CtrlFIFO. This will be zero if the FIFO is connected to no file.
NOTE	String	The FIFO's note string as set by CtrlFIFO.
VALID	Number	Zero if FIFO is not valid.

In addition, FIFOStatus writes fields to S_Info for each channel in the FIFO. The keyword for the field is a combination of a name and a number that identify the field and the channel to which it refers. For example, if channel 4 is named "Pressure" then the following would appear in the S_Info string: NAME4:Pressure.

In the following table, the channel's number is represented by "#".

Keyword	Type	Meaning
FSMINUS#	Number	Channel's minus full scale value as set by NewFIFOChan.
FSPLUS#	Number	Channel's plus full scale value as set by NewFIFOChan.
GAIN#	Number	Channel's gain value as set by NewFIFOChan.
NAME#	String	Name of channel.
OFFSET#	Number	Channel's offset value as set by NewFIFOChan.
UNITS#	String	Channel's units as set by NewFIFOChan.

See Also

The **NewFIFO**, **CtrlFIFO**, and **NewFIFOChan** operations, **FIFOs and Charts** on page IV-276 for more information on FIFOs and data acquisition.

The **NumberByKey** and **StringByKey** functions for parsing keyword-value strings.

FilterFIR

FilterFIR [***flags***] ***waveName*** [, ***waveName***]…

The FilterFIR operation convolves each *waveName* with automatically-designed filter coefficients or with *coefsWaveName* using time-domain methods.

The automatically-designed filter coefficients are simple lowpass and highpass window-based filters or a maximally-flat notch filter. Multiple filter designs are combined into a composite filter. The filter can be optionally placed into the first *waveName* or just used to filter the data in *waveName*.

FilterFIR filters data faster than **Convolve** when there are many fewer filter coefficient values than data points in *waveName*.

Note: FilterFIR replaces the obsolete **SmoothCustom** operation.

Parameters

waveName is a destination wave that is overwritten by the convolution of itself and the filter.

waveName may be multidimensional, but only one dimension selected by /DIM is filtered (for two-dimensional filtering, see **MatrixFilter**).

Flags

/COEF [=*coefsWaveName*]

Replaces the first output *waveName* by the filter coefficients instead of the filtered results or, when *coefsWaveName* is specified, replaces the output wave(s) by the result of convolving *waveName* with coefficients in *coefsWaveName*.

coefsWaveName must not be one of the destination *waveName*s. It must be single- or double-precision numeric and one-dimensional.

To avoid shifting the output with respect to the input, *coefsWaveName* must have an odd length with the "center" coefficient in the middle of the wave.

The coefficients are usually symmetrical about the middle point, but FilterFIR does not enforce this.

/DIM=*d*

Specifies the wave dimension to filter.

d=-1: Treats entire wave as 1D (default).

For *d*=0, 1,…, operates along rows, columns, etc.

Use /DIM=0 to apply the filter to each individual column (each one a channel, say left and right) in a multidimensional *waveName* where each row comprises all of the sound samples at a particular time.

/E=*endEffect*

Determines how the ends of the wave (*w*) are handled when fabricating missing neighbor values. *endEffect* has values:

0: Bounce method (default). Uses w[*i*] in place of the missing w[-*i*] and w[*n*-*i*] in place of the missing w[*n*+*i*].

1: Wrap method. Uses w[*n*-*i*] in place of the missing w[-*i*] and vice versa.

2: Zero method. Uses 0 for any missing value.

3: Fill method. Uses w[0] in place of the missing w[-*i*] and w[*n*] in place of the missing w[*n*+*i*].

/HI={*f1*, *f2*, *n*}

Creates a high-pass filter based on the windowing method, using the Hanning window unless another window is specified by /WINF.

f1 and *f2* are filter design frequencies measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency).

f1 is the end of the reject band, and *f2* is the start of the pass band:

$0 < f1 < f2 < 0.5$

n is the number of FIR filter coefficients to generate. A larger number gives better stop-band rejection. A good number to start with is 101.

Use both /HI and /LO to create a bandpass filter.

/LO={*f1*, *f2*, *n*}

Creates a low-pass filter. *f1* is the end of the pass band, *f2* is the start of the reject band, and *n* is the number of FIR filter coefficients. See /HI for more details.

/NMF={*fc*, *fw* [, *eps*, *nMult*]}

Creates a maximally-flat notch filter centered at *fc* with a -3dB width of *fw*. *fc* and *fw* are filter design frequencies measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency).

The longest filter length allowed is 4001 points, which requires *fw* >= 0.0079 (1.58% of the sampling frequency).

Coefficients at the ends that are smaller than the optional *eps* parameter are removed, making the filter shorter (and faster), though less accurate. The default is 2^{-40}. Use 0 to retain all coefficients, no matter how small, even zero coefficients.

nMult specifies how much longer the filter may be to obtain the most accurate notch frequency. The default is 2 (potentially twice as many coefficients). Set *nMult* <= 1 to generate the shortest possible filter.

The maximally flat notch filter design is based on Zahradník and Vlcek, and uses arbitrary precision math (see **APMath**) to compute the coefficients.

/WINF=*windowKind*

Applies the named "window" to the filter coefficients. Windows alter the frequency response of the filter in obvious and subtle ways, enhancing the stop-band rejection or steepening the transition region between passed and rejected frequencies. They matter less when many filter coefficients are used.

If /WINF is not specified, the Hanning window is used. For no coefficient filtering, use /WINF=None.

Choices for *windowKind* are (see the **FFT** /WINF flag for window equations and details):

Bartlett, Blackman367, Blackman361, Blackman492, Blackman474, Cos1, Cos2, Cos3, Cos4, Hamming, Hanning, KaiserBessel20, KaiserBessel25, KaiserBessel30, Parzen, Poisson2, Poisson3, Poisson4, and Riemann.

Details

If *coefsWaveName* is specified, then /HI, /LO, and /NMF are ignored.

If more than one of /HI, /LO, and /NMF are specified, the filters are combined using linear convolution. The length of the combined filter is slightly less than the sum of the individual filter lengths.

The filtering convolution is performed in the time-domain. That is, the FFT is not employed to filter the data. For this reason the coefficients length should be small in comparison to the destination waves.

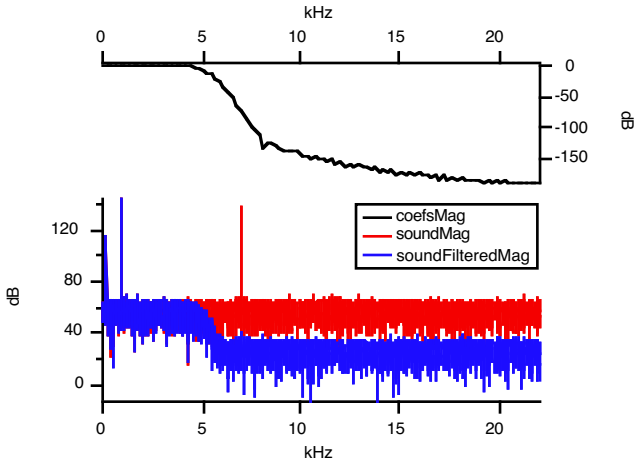
FilterFIR assumes that the middle point of *coefsWaveName* corresponds to the delay = 0 point. The "middle" point number = trunc(numpnts(*coefsWaveName* -1)/2). *coefsWaveName* usually contains the two-sided impulse response of a filter, and usually contains an odd number of points. This is the kind of coefficients data generated by /HI, /LO, and /NMF.

FilterFIR ignores the X scaling of all waves, except when /COEF creates a coefficients wave, which preserves the X scale deltax and alters the leftx value so that the zero-phase (center) coefficient is located at x=0.

Examples

```
// Make test sound from three sine waves
Variable/G fs= 44100                                // Sampling frequency
Variable/G seconds= 0.5                             // Duration
Variable/G n= 2*round(seconds*fs/2)
Make/O/W/N=(n) sound                                // 16-bit integer sound wave
SetScale/p x, 0, 1/fs, "s", sound
Variable/G f1= 200, f2= 1000, f3= 7000
Variable/G a1=100, a2=3000,a3=1500
sound= a1*sin(2*pi*f1*x)
sound += a2*sin(2*pi*f2*x)
sound += a3*sin(2*pi*f3*x)+gnoise(10)               // Add a noise floor

// Compute the sound's spectrum in dB
FFT/MAG/WINF=Hanning/DEST=soundMag sound
soundMag= 20*log(soundMag)
SetScale d, 0, 0, "dB", soundMag

// Apply a 5 kHz low-pass filter to the sound wave
Duplicate/O sound, soundFiltered
FilterFIR/E=3/LO={4000/fs, 6000/fs, 101} soundFiltered

// Compute the filtered sound's spectrum in dB
FFT/MAG/WINF=Hanning/DEST=soundFilteredMag soundFiltered
```

```
soundFilteredMag= 20*log(soundFilteredMag)
SetScale d, 0, 0, "dB", soundFilteredMag

// Compute the filter's frequency response in dB
Make/O/D/N=0 coefs                                    // Double precision is recommended
SetScale/p x, 0, 1/fs, "s", coefs
FilterFIR/COEF/LO={4000/fs, 6000/fs, 101} coefs
FFT/MAG/WINF=Hanning/PAD={(2*numpnts(coefs))}/DEST=coefsMag coefs
coefsMag= 20*log(coefsMag)
SetScale d, 0, 0, "dB", coefsMag

// Graph the frequency responses
Display/R/T coefsMag as "FIR Lowpass Example";DelayUpdate
AppendToGraph soundMag, soundFilteredMag;DelayUpdate
ModifyGraph axisEnab(left)={0,0.6}, axisEnab(right)={0.65,1}
ModifyGraph rgb(soundFilteredMag)=(0,0,65535), rgb(coefsMag)=(0,0,0)
Legend
```




```
// Graph the unfiltered and filtered sound time responses
Display/L=leftSound sound as "FIR Filtered Sound";DelayUpdate
AppendToGraph/L=leftFiltered soundFiltered;DelayUpdate
ModifyGraph axisEnab(leftSound)={0,0.45}, axisEnab(leftFiltered)={0.55,1}
ModifyGraph rgb(soundFiltered)=(0,0,65535)

// Listen to the sounds
PlaySound sound               // This has a very high frequency tone
PlaySound soundFiltered       // This doesn't
```

References

Zahradník, P., and M. Vlcek, Fast Analytical Design Algorithms for FIR Notch Filters, *IEEE Trans. on Circuits and Systems*, *51*, 608 - 623, 2004.

<http://euler.fd.cvut.cz/publikace/files/vlcek/notch.pdf>

See Also

Smoothing on page III-255; the **Smooth**, **Convolve**, **MatrixConvolve**, and **MatrixFilter** operations.

FilterIIR

FilterIIR [***flags***] [***waveName***,…]

The FilterIIR operation applies to each *waveName* either the automatically-designed IIR filter coefficients or the IIR filter coefficients in *coefsWaveName*. Multiple filter designs are combined into a composite filter. The filter can be optionally placed into the first *waveName* or just used to filter the data in *waveName*.

The automatically-designed filter coefficients are bilinear transforms of the Butterworth analog prototype with an optional variable-width notch filter.

To design more advanced IIR filters, see **Designing the IIR Coefficients**.

Parameters

waveName may be multidimensional, but only the one dimension selected by /DIM is filtered (for two-dimensional filtering, see **MatrixFilter**).

waveName may be omitted for the purpose of checking the format of *coefsWaveName*. If the format is detectably incorrect an error code will be returned in V_flag. Use /Z to prevent command execution from stopping.

Flags

/CASC — Specifies that *coefsWaveName* contains cascaded bi-quad filter coefficients. The cascade implementation is more stable and numerically accurate for high-order IIR filtering than Direct Form 1 filtering. See **Cascade Details**.

/COEF [=*coefsWaveName*] — Replaces the first output *waveName* by the filter coefficients instead of the filtered results or, when *coefsWaveName* is specified, replaces the output wave(s) by the result of filtering *waveName* with the IIR coefficients in *coefsWaveName*.

coefsWaveName must not be one of the destination *waveName*s. It must be single- or double-precision numeric and two-dimensional.

When used with /CASC, *coefsWaveName* must have 6 columns, containing real-valued coefficients for a product of ratios of second-order polynomials (cascaded bi-quad sections).

If /ZP is specified, it must be complex, otherwise it must be real.

See **Details** for the format of the coefficients in *coefsWaveName*.

/DIM=*d* — Specifies the wave dimension to filter.

d=-1: Treats entire wave as 1D (default).

For *d*=0, 1,…, operates along rows, columns, etc.

Use /DIM=0 to apply the filter to each individual column (each one a channel, say left and right) in a multidimensional *waveName* where each row comprises all of the sound samples at a particular time.

/HI=*fHigh* — Creates a high-pass Butterworth filter with the -3dB corner at *fHigh*. The order of the filter is controlled by the /ORD flag.

fHigh is a filter design frequency measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency).

/LO=*fLow* — Creates a low-pass Butterworth filter with the -3dB corner at *fLow*. The /ORD flag controls the order of the filter.

fLow is a filter design frequency measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency).

Create bandpass and bandreject filters by specifying both /HI and /LO. For a bandpass filter, set *fLow* > *fHigh*, and for a band reject filter, set *fLow* < *fHigh*.

/N={*fNotch*, *notchQ*} — Creates a notch filter with the center frequency at *fNotch* and a -3dB width of *fNotch*/*notchQ*.

fNotch is a filter design frequency measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency).

notchQ is a number greater than 1, typically 10 to 100. Large values produce a filter that "rings" a lot.

/ORD=*order* — Sets the order of the Butterworth filter(s) created by /HI and /LO. The default is 2 (second order), and the maximum is 100.

/Z=*z* — Prevents procedure execution from aborting when an error occurs. Use /Z=1 to handle this case in your procedures using **GetRTError**(1) rather than having execution abort. /Z=0 is the same as no /Z at all.

/ZP — Specifies that *coefsWaveName* contains complex z-domain zeros (in column 0) and poles (in column 1) or, if *coefsWaveName* is not specified, that the first output *waveName* is to be replaced by filter coefficients in the zero-pole format. See **Zeros and Poles Details**.

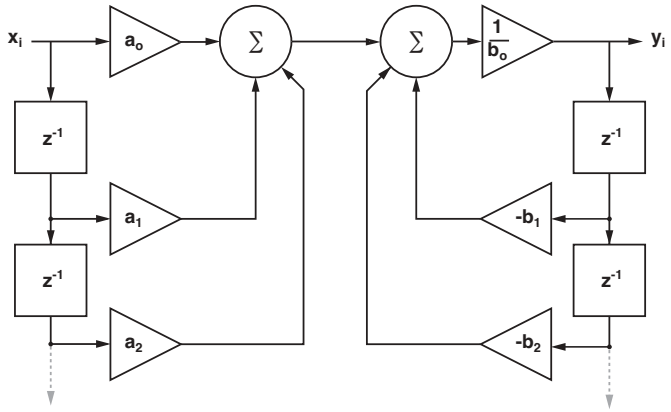
Details

FilterIIR sets V_flag to 0 on success or to an error code if an error occurred. Command execution stops if an error occurs unless the /Z flag is set. Omit /Z and call **GetRTError** and **GetRTErrMessage** under similar circumstances to see what the error code means.

Direct Form 1 Details

Unless /CASC or /ZP are specified, the coefficients in *coefsWaveName* describe a ratio of two polynomials of the Z transform:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2} + \ldots}{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots}$$



Direct Form I Implementation

$$y_i = \frac{a_0 x_i + a_1 x_{i-1} + a_2 x_{i-2} + \ldots - b_1 y_{i-1} - b_2 y_{i-2} + \ldots}{b_0}$$

where x is the input wave *waveName* and y is the output wave (either *waveName* again or *destWaveName*).

FilterIIR computes the filtered result using the Direct Form I implementation of $H(z)$.

The rational polynomial numerator (a_i) coefficients in are column 0 and denominator (b_i) coefficients in column 1 of *coefsWaveName*.

The coefficients in row 0 are the nondelayed coefficients a_0 (in column 0) and b_0 (in column 1).

The coefficients in row 1 are the z^{-1} coefficients, a_1 and b_1.

The coefficients in row n are the z^{-n} coefficients, a_n and b_n.

The number of coefficients for the numerator can differ from the number of coefficients for the denominator. In this case, specify 0 for unused coefficients.

Note: If all the coefficients of the denominator are 0 (b_i= 0 except b_0 = 1), then the filter is actually a causal FIR filter (Finite Impulse Response filter with delay of n-1). In this sense, FilterIIR implements a superset of the FilterFIR operation.

Alternate Direct Form 1 Notation

The designation of a_i, etc. as the numerator is at odds with many textbooks such as *Digital Signal Processing*, which uses b for the numerator coefficients of the rational function, a for the denominator coefficients with an implicit a_0 = 1, in addition to reversing the signs of the remaining denominator coefficients so that they can write $H(z)$ as:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{i=0}^{n} b_i z^{-i}}{1 - \sum_{i=1}^{n} a_i z^{-i}}.$$

Coefficients derived using this notation need their denominator coefficients sign-reversed before putting them into rows 1 through n of column 1 (the second column), and the "missing" nondelayed denominator coefficient of 1.0 placed in row 0, column 1.

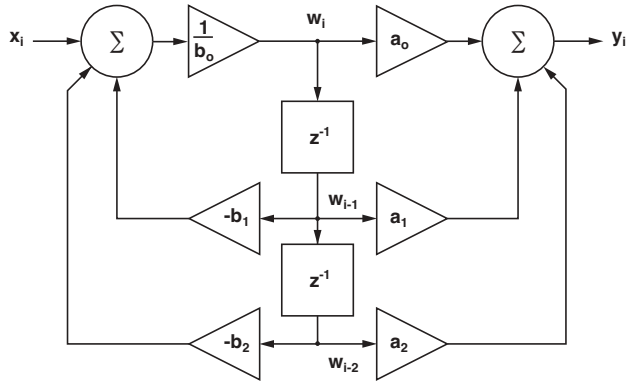
Cascade Details

When using /CASC, coefficients in *coefsWaveName* describe the product of one or more ratios of two quadratic polynomials of the Z transform:

$$H(z) = \frac{Y(z)}{X(z)} = \prod_{k=1}^{k} \frac{a_{0_k} + a_{1_k} z^{-1} + a_{2_k} z^{-2}}{b_{0_k} + b_{1_k} z^{-1} + b_{2_k} z^{-2}}.$$

Each product term implements a "cascaded bi-quad section", and $H(z)$ can be realized by feeding the output of one section to the next one.

The cascade coefficients filter the data using a Direct Form II cascade implementation:



$$w_i = \frac{x_i - b_1 w_{i-1} - b_2 w_{i-2}}{b_0}$$

$$y_i = a_0 w_i + a_1 w_{i-1} + a_2 w_{i-2}$$

Cascaded Bi-Quad Direct Form II Implementation

The cascade implementation is more stable and numerically accurate for high-order IIR filtering than Direct Form I filtering. Cascade IIR filtering is recommended when the filter order exceeds 16 (a 16th-order Direct Form I filter has 17 numerator coefficients and 17 denominator coefficients).

coefsWaveName must be a six-column real-valued numeric wave. Each row describes one bi-quad section. The coefficients for the second term (or "section") of the product (*k*=2) are in the following row, etc.:

k	Row	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5
1	0	a_{0_1}	a_{1_1}	a_{2_1}	b_{0_1}	b_{1_1}	b_{2_1}
2	1	a_{0_2}	a_{1_2}	a_{2_2}	b_{0_2}	b_{1_2}	b_{2_2}
	…						

The number of coefficients for the numerator (*a*'s) is allowed to differ from the number of coefficients for the denominator (*b*'s). In this case, specify 0 for unused coefficients.

For example, a third order filter (three poles and three zeros) cascade implementation is a single-order section combined with a second order section. The values for a_{2_k}, b_{2_k} for that section (*k*) would be 0. Here the second section is specified as the first-order section:

k	Row	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5
1	0	a_{0_1}	a_{1_1}	a_{2_1}	b_{0_1}	b_{1_1}	b_{2_1}
2	1	a_{0_2}	a_{1_2}	0	b_{0_2}	b_{1_2}	0

Alternate Cascade Notation

In the DSP literature, the b_{0_k} gain values are typically one and the $H(z)$ expression contains an overall gain value, usually *K*. Here each product term (or "section") has a user-settable gain value. Computing the correct gain values to control overflow in integer implementations is the responsibility of the user. For floating implementations, you might as well set all b_{0_k} values to one except, say, b_{0_1}, to control the overall gain.

Zeros and Poles Details

When using /ZP, coefficients in *coefsWaveName* contains complex zeros and poles in the (also complex) Z transform domain:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{(z-z_0)\bullet(z-z_1)\bullet(z-z_2)\bullet\ldots}{(z-p_0)\bullet(z-p_1)\bullet(z-p_2)\bullet\ldots}.$$

coefsWaveName must be a two-column complex wave with zero0, zero1,… zeroN in the first column of N+1 rows, and pole0, pole1,… poleN in the second column of those same rows:

k	Row	Col 0	Col 1
1	0	(zero0Real, zero0Imag)	(pole0Real, pole0Imag)
2	1	(zero1Real, zero1Imag)	(pole1Real, pole1Imag)
3	2	(zero2Real, zero2Imag)	(pole2Real, pole2Imag)
	…		

If a zero or pole has a nonzero imaginary component, the conjugate zero or pole must be included in *coefsWaveName*. For example, if a zero is placed at (0.7, 0.5), the conjugate is (0.7, -0.5), and that value must also appear in column 0. These two zeros form what is known as a "conjugate pair". The conjugate values must match within the greater of 1.0e-6 or 1.0e-6 * |zeroOrPole|.

Use (0,0) for unused poles or zeros, as a zero or pole at *z*= (0,0) has no effect on the filter frequency response.

The /ZP format for the coefficients is internally converted into the Direct Form 1 implementation, or into the Cascade Direct Form 2 implementation if /CASC is specified. There is no option for returning these implementation-dependent coefficients in a wave.

Designing the IIR Coefficients

Simple IIR filters can be used or created by specifying the /LO, /HI, /ORD, /N, /CASC, and /ZP flags. Use /COEF without *coefsWaveName* to put these simple IIR filter coefficients into the first *waveName*.

More advanced IIR filters (Bessel, Chebyshev) can be designed using the separate IFDL package. IFDL is a suite of extensions and macros that you use to design FIR (Finite Impulse Response) and IIR (Infinite Impulse Response) filters and to apply them to your data. The IIR design software creates IIR coefficients based on bilinear transforms of analog prototype filters such as Bessel, Butterworth, and Chebyshev. See the WaveMetrics web site for more about IFDL.

Even without IFDL, you can create custom IIR filters by manually placing poles and zeros in the Z plane using the Pole and Zero Filter Design procedures. Copy the following line to your Procedure window and click the Compile button at the bottom of the procedure window:

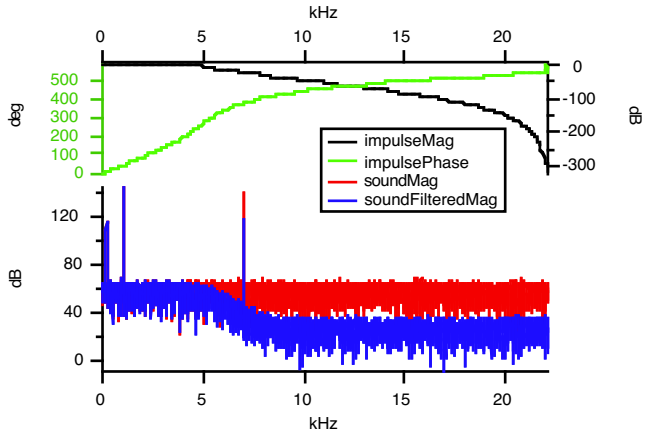
```
#include <Pole And Zero Filter Design>
```

Then choose Pole and Zero Filter Design from the Analysis menu.

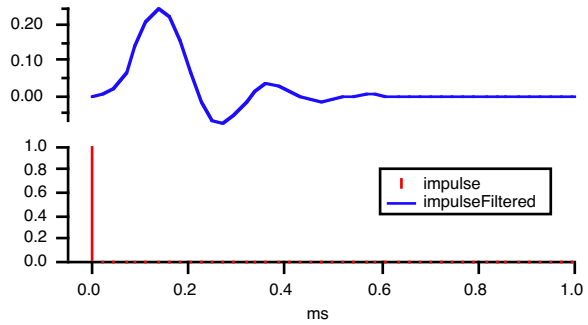
Examples

```
// Make test sound from three sine waves
Variable/G fs= 44100                          // Sampling frequency
Variable/G seconds= 0.5                       // Duration
Variable/G n= 2*round(seconds*fs/2)
Make/O/W/N=(n) sound                          // 16-bit integer sound wave
SetScale/p x, 0, 1/fs, "s", sound
Variable/G f1= 200, f2= 1000, f3= 7000
Variable/G a1=100, a2=3000,a3=1500
sound= a1*sin(2*pi*f1*x)
sound += a2*sin(2*pi*f2*x)
sound += a3*sin(2*pi*f3*x)+gnoise(10)             // Add a noise floor

// Compute the sound's spectrum in dB
FFT/MAG/WINF=Hanning/DEST=soundMag sound
soundMag= 20*log(soundMag)
SetScale d, 0, 0, "dB", soundMag

// Apply a 5 kHz, 6th order low-pass filter to the sound wave
Duplicate/O sound, soundFiltered
FilterIIR/LO=(5000/fs)/ORD=6 soundFiltered        // Second order by default
```

```
// Compute the filtered sound's spectrum in dB
FFT/MAG/WINF=Hanning/DEST=soundFilteredMag soundFiltered
soundFilteredMag= 20*log(soundFilteredMag)
SetScale d, 0, 0, "dB", soundFilteredMag

// Compute the filter's frequency and phase by filtering an impulse
Make/O/D/N=2048 impulse= p==0                    // Impulse at t==0
SetScale/P x, 0, 1/fs, "s", impulse
Duplicate/O impulse, impulseFiltered
FilterIIR/LO=(5000/fs)/ORD=6 impulseFiltered
FFT/MAG/DEST=impulseMag impulseFiltered
impulseMag= 20*log(impulseMag)
SetScale d, 0, 0, "dB", impulseMag
FFT/OUT=5/DEST=impulsePhase impulseFiltered
impulsePhase *= 180/pi                           // Convert to degrees
SetScale d, 0, 0, "deg", impulsePhase
Unwrap 360, impulsePhase                         // Continuous phase

// Graph the frequency responses
Display/R/T impulseMag as "IIR Lowpass Example"
AppendToGraph/L=phase/T impulsePhase
AppendToGraph soundMag, soundFilteredMag
ModifyGraph axisEnab(left)={0,0.6}
ModifyGraph axisEnab(right)={0.65,1}
ModifyGraph axisEnab(phase)={0.65,1}
ModifyGraph freePos=0, lblPos=60, rgb(soundFilteredMag)=(0,0,65535)
ModifyGraph rgb(impulseMag)=(0,0,0), rgb(impulsePhase)=(0,65535,0)
ModifyGraph axRGB(phase)=(3,52428,1), tlblRGB(phase)=(3,52428,1)
Legend
```




```
// Graph the unfiltered and filtered impulse time responses
Display/L=leftImpulse impulse as "IIR Filtered Impulse"
AppendToGraph/L=leftFiltered impulseFiltered
ModifyGraph axisEnab(leftImpulse)={0,0.45}, axisEnab(leftFiltered)={0.55,1}
ModifyGraph freePos=0, margin(left)=50
ModifyGraph mode(impulse)=1, rgb(impulseFiltered)=(0,0,65535)
SetAxis bottom -0.00005,0.001
Legend
```




```
// Listen to the sounds
PlaySound sound                  // This has a very high frequency tone
PlaySound soundFiltered          // This doesn't
```

References

Embree, P.M., and B. Kimble, *C Language Algorithms for Signal Processing*, 456 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Lynn, P.A., and W. Fuerst, *Introductory Digital Signal Processing with Computer Applications*, 479 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1998.

Oppenheim, A.V., and R.W. Schafer, *Digital Signal Processing*, 585 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1975.

Terrell, T.J., *Introduction to Digital Filters*, 2nd ed., 261 pp., John Wiley & Sons, New York, 1988.

See Also

Smoothing on page III-255; the **FFT** and **FilterFIR** operations.

FindDimLabel

FindDimLabel(*waveName, dimNumber, labelString*)

Returns the index value corresponding to the label for the given dimension. Returns -1 if the label is for the entire dimension. Returns -2 if the label is not found.

Use *dimNumber* =0 for rows, 1 for columns, 2 for layers, or 3 for chunks.

FindLevel

FindLevel [***flags***] ***waveName, level***

The FindLevel operation searches the named wave to find the X value at which the specified Y *level* is crossed.

Flags

/B=*box* Sets box size for sliding average. If /B=*box* is omitted or *box* equals 1, no averaging is done. If you specify an even box size then the next higher (odd) integer is used. If you use a box size greater than 1, FindLevel will be unable to find a level crossing that occurs in the first or last *box*/2 -1 points of the wave since these points don't have enough neighbors for computing the derived average wave values.

/EDGE=*e* Specifies searches for either increasing or decreasing level crossing.

- *e*=1: Searches only for crossing where Y values are increasing as *level* is crossed from wave start towards wave end.
- *e*=2: Searches only for crossing where the Y values are decreasing as *level* is crossed from wave start towards wave end.
- *e*=0: Same as no /EDGE flag (searches for either increasing and decreasing level crossing).

/P Computes the X crossing location in terms of point number. If /P is omitted, the level crossing location is computed in terms of X values.

/Q Don't print results in history and don't report error if *level* is not found.

/R=(*startX*,*endX*) Specifies an X range of the wave to search. You may exchange *startX* and *endX* to reverse the search direction.

/R=[*startP*,*endP*] Specifies a point range of the wave to search. You may exchange *startP* and *endP* to reverse the search direction. If you specify the range as /R=[*startP*] then the end of the range is taken as the end of the wave. If /R is omitted, the entire wave is searched.

/T=*dx* Search for two level crossings. *dx* must be less than *minWidthX*, so you must also specify /M if you use /T. (FindLevel limits *dx* so that second search start isn't beyond where the first search for next edge will be.)

/T=*dx* Performs a second search after finding the initial level crossing. The second search starts *dx* units beyond the initial level crossing and looks back in the direction of the initial crossing. If FindLevel finds a second level crossing, it sets V_LevelX to the average of the initial and second crossings. Otherwise, it sets V_LevelX to the initial crossing.

Details

FindLevel scans through the wave comparing *level* to values derived from the Y values of the wave. Each derived value is a sliding average of the Y values.

FindLevel searches for two derived wave values that straddle *level*. If it finds these values it computes the X value at which *level* is located by linearly interpolating between the straddling Y values.

Note: FindLevel does not locate values exactly equal to *level*; it locates transitions through *level*. See **BinarySearch** for one method of locating exact values.

FindLevel reports its results by setting these variables:

`V_flag`	0: *level* was found. 1: *level* was not found.
`V_LevelX`	Interpolated X value at which *level* was found, or the corresponding point number if /P is specified.
`V_rising`	0: Y values at the crossing are decreasing from wave start towards wave end. 1: Y values at the crossing are increasing.

If you do not use the /Q flag then FindLevel also reports its results by printing them in the history area, and if *level* is not found FindLevel generates an error which puts up an error alert and also halts execution of any command line or macro that is in progress.

These X locations and distances are in terms of the X scaling of the named wave unless you use the /P flag, in which case they are in terms of point number.

V_LevelX is returned in terms of the X scaling of the named wave unless you use the /P flag, in which case it is in terms of point number.

The FindLevel operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

See Also

The **EdgeStats**, **FindLevels**, **FindValue**, and **PulseStats** operations and the **BinarySearch** and **BinarySearchInterp** functions.

FindLevels

**`FindLevels [`*`flags`*`] `*`waveName, level`*

The FindLevels operation searches the named wave to find one or more X values at which the specified Y *level* is **crossed**.

To find where the wave is equal to a given value, use **FindValue** instead.

Flags

/B=*box*	Sets box size for sliding average. See the **FindLevel** operation.
/D=*destWaveName*	Specifies wave into which FindLevels is to store the level crossing values. If /D and /DEST are omitted, FindLevels creates a wave named W_FindLevels to store the level crossing values in.
/DEST=*destWaveName*	Same as /D. Added /DEST for Igor 6.13; now the named wave need not pre-exist. Both /D and /DEST create a real wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-56 for details.
/EDGE=*e*	Specifies searches for either increasing or decreasing level crossing. *e*=1: Searches only for crossings where the Y values are increasing as level is crossed from wave start towards wave end. *e*=2: Searches only for crossings where the Y values are decreasing as level is crossed from wave start towards wave end. *e*=0: Same as no /EDGE flag (searches for both increasing and decreasing level crossings).
/M=*minWidthX*	Sets the minimum X distance between level crossings. This determines where FindLevels searches for the next crossing after it has found a level crossing. The search starts *minWidthX* X units beyond the crossing. The default value for *minWidthX* is 0.

/N=*maxLevels*	Sets a maximum number of crossings that FindLevels is to find. The default value for *maxLevels* is the number of points in the specified range of *waveName*.
/P	Compute crossings in terms of points. See the **FindLevel** operation.
/Q	Doesn't print to history and doesn't abort if no levels are found.
/R=(*startX*,*endX*)	Specifies X range. See the **FindLevel** operation.
/R=[*startP*,*endP*]	Specifies point range. See the **FindLevel** operation.
/T=*dx*	Search for two level crossings. *dx* must be less than *minWidthX*, so you must also specify /M if you use /T. (FindLevels limits *dx* so that second search start isn't beyond where the first search for next edge will be.) See **FindLevel** for more about /T.

Details

The algorithm for finding a level crossing is the same one used by the **FindLevel** operation.

If FindLevels finds *maxLevels* crossings or can not find another level crossing, it stops searching.

FindLevels sets the following variables:

`V_flag`	0: *maxLevels* level crossings were found.
	1: At least one but less than *maxLevels* level crossings were found.
	2: No level crossings were found.
`V_LevelsFound`	Number of level crossings found.

Examples

```
// Prior to Igor 6.13, any /D wave had to exist:
Make/O/D/N=0 destWave
FindLevels/Q/D=destWave data, 5
if( V_LevelsFound )
   Print destWave[0]   // First crossing X location

// As of Igor 6.13, /D and /DEST create the destination wave if it does not exist:
FindLevels/Q/D=destWave data, 5
if( V_LevelsFound )
   Print destWave[0]   // First crossing X location
```

See Also

The **FindLevel** operation for details about the level crossing detection algorithm and the /B, /P, /Q, /R, and /T flag values.

The FindLevels operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

FindListItem

FindListItem(*itemStr, listStr* [, *listSepStr* [, *start* [, *matchCase*]]])

The FindListItem function returns a numeric offset into *listStr* where *itemStr* begins. *listStr* should contain items separated by the *listSepStr* character, such as "abc;def;".

Use FindListItem to locate the start of an item in a string containing a "wave0;wave1;" style list such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

Use WhichListItem to determine the index of an item in the list.

If *itemStr* is not found, if *listStr* is `""`, or if *start* is not within the range of 0 to strlen(*listStr*)-1, then -1 is returned.

listSepStr, *startIndex*, and *matchCase* are optional; their defaults are ";", 0, and 1 respectively.

Details

ItemStr may have any length.

listStr is searched for the first instance of the item string bound by a *listSepStr* on the left and a *listSepStr* on the right. The returned number is the character index where the first character of *itemStr* was found in *listSepStr*.

The search starts from the character position in *listStr* specified by *start*. A value of 0 starts with the first character in *listStr*, which is the default if *start* is not specified.

listString is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *listSepStr* are always case-sensitive. The comparison of *itemStr* to the contents of *listStr* is usually case-sensitive. Setting the optional *matchCase* parameter to 0 makes the comparison case insensitive.

Only the first character of *listSepStr* is used.

If *startIndex* is specified, then *listSepStr* must also be specified. If *matchCase* is specified, *startIndex* and *listSepStr* must be specified.

Examples

```
Print FindListItem("w1", "w0;w1;w2,")                // prints 3
Print FindListItem("v2", "v1,v2,v3,", ",")           // prints 3
Print FindListItem("v2", "v0,v2,v2,", ",", 4)        // prints 6
Print FindListItem("C", "a;c;C;")                    // prints 4
Print FindListItem("C", "a;c;C;", ";", 0, 0)         // prints 2
```

See Also

The **AddListItem**, **strsearch**, **StringFromList**, **RemoveFromList**, **ItemsInList**, **WhichListItem**, **WaveList**, **TraceNameList**, **StringList**, **VariableList**, and **FunctionList** functions.

FindPeak

FindPeak [***flags***] ***waveName***

The FindPeak operation searches for a minimum or maximum by analyzing the smoothed first and second derivatives of the named wave. Information about the peak position, amplitude, and width are returned in the output variables.

Flags

Some of the flags have the same meaning as for the FindLevel operation.

/B=*box*	Sets box size for sliding average.
/I	Modify the search criteria to accommodate impulses (peaks of one sample) by requiring only one value to exceed *minLevel*. The default criteria requires that two successive values exceed *minLevel* for a peak to be found (or two successive values be less than the /M level when searching for negative peaks). Impulses can also be found by omitting *minLevel*, in which case /I is superfluous.
/M=*minLevel*	Defines minimum level of a peak. /N changes this to maximum level (see **Details**).
/N	Searches for a negative peak (minimum) rather then a positive peak (maximum).
/P	Location output variables (see **Details**) are reported in terms of (floating point) point numbers. If /P is omitted, they are reported as X values.
/Q	Doesn't print to history and doesn't abort if no peak is found.
/R=(*startX*,*endX*)	Specifies X range and direction for search.
/R=[*startP*,*endP*]	Specifies point range and direction for search.

Details

FindPeak sets the following variables:

V_flag	Set only when using the /Q flag. 0: Peak was found. Any nonzero value means the peak was not found.
V_LeadingEdgeLoc	Interpolated location of the peak edge closest to *startX* or *startP*. If you use the /P flag, V_LeadingEdgeLoc is a point number rather than to an X value. If the edge was not found, this value is NaN.
V_PeakLoc	Interpolated X value at which the peak was found. If you use the /P flag, FindPeak sets V_PeakLoc to a point number rather than to an X value. Set to NaN if peak wasn't found.
V_PeakVal	The *approximate* Y value of the found peak. If the peak was not found, this value is NaN (Not a Number).
V_PeakWidth	Interpolated peak width. If you use the /P flag, V_PeakWidth is expressed in point numbers rather than as an X value. V_PeakWidth is never negative. If either peak edge was not found, this value is NaN.

V_TrailingEdgeLoc Interpolated location of the peak edge closest to *endX* or *endP*. If you use the /P flag, V_TrailingEdgeLoc is a point number rather than to an X value. If the edge was not found, this value is NaN.

FindPeak computes the sliding average of the input wave using the BoxSmooth algorithm with the *box* parameter. The peak center is found where the derivative of this smoothed result crosses zero. The peak edges are found where the second derivative of the smoothed result crosses zero. Linear interpolation of the derivatives is used to more precisely locate the center and edges. The peak value is simply the greater of the two unsmoothed values surrounding the peak center (if /N, then the lesser value).

FindPeak is not a high-accuracy measurement routine; it is intended as a simple peak-finder. Use the **PulseStats** operation for more precise statistics.

Without /M, a peak is found where the derivative crosses zero, regardless of the peak height.

If you use the /M=*minLevel* flag, FindPeak ignores peaks that are lower than *minLevel* (i.e., the Y value of a found peak will exceed *minLevel*) in the box-smoothed input wave. If /N is also specified (search for minimum), FindPeak ignores peaks whose amplitude is greater than *minLevel* (i.e., the Y value of a found peak will be *less* than *minLevel*).

Without /I, a peak must have two successive values that exceed *minLevel*. Use /I when you are searching for peaks that may have only one value exceeding *minLevel*.

The search for the peak begins at *startX* (or the first point of the wave if /R is not specified), and ends at *endX* (or the last point of the wave if no /R). Searching backwards is permitted, and exchanges the values of V_LeadingEdgeLoc and V_TrailingEdgeLoc.

A simple automatic peak-finder is implemented in the procedure file:

```
#include <Peak AutoFind>
```

one of the `#include <Multi-peak fitting 1.3>` procedures that provides support for Gaussian, Lorentzian, and Voigt fitting functions. See the "Multi-peak fit" example experiment for details (:Examples:Curve Fitting: folder).

The FindPeak operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

See Also

The **PulseStats** operation, the **FindLevel** operation for details about the /B, /P, /Q, and /R flag values.

FindPointsInPoly

FindPointsInPoly *xWaveName, yWaveName, xPolyWaveName, yPolyWaveName*

The FindPointsInPoly operation determines if points fall within a certain polygon. It can be used to write procedures that operate on a subset of data identified graphically in a graph.

Details

FindPointsInPoly determines which points in *yWaveName* vs *xWaveName* fall within the polygon defined by *yPolyWaveName* vs *xPolyWaveName*.

xWaveName must have the same number of points as *yWaveName* and *xPolyWaveName* must have the same number of points as *yPolyWaveName*.

FindPointsInPoly creates an output wave named W_inPoly with the same number of points as *xWaveName*. FindPointsInPoly indicates whether the point *yWaveName*[p] vs *xWaveName*[p] falls within the polygon by setting `W_inPoly[p]=1` if it is within the polygon, or `W_inPoly[p]=0` if it is not.

FindPointsInPoly uses integer arithmetic with a precision of about 1 part in 1000. This should be good enough for visually determined (hand-drawn) polygons but might not be sufficient for mathematically generated polygons.

The FindPointsInPoly operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

See Also

The **GraphWaveDraw** operation.

FindRoots

FindRoots [***flags***] ***funcspec, pWave*** [**,** ***funcspec, pwave*** [**, …**]]

FindRoots /P=*PolyCoefsWave*

The FindRoots operation determines roots or zeros of a specified nonlinear function or system of functions. The function or system of functions must be defined in the form of Igor user procedures.

Using the second form of the command, FindRoots finds all the complex roots of a polynomial with real coefficients. The polynomial coefficients are specified by *PolyCoefsWave*.

Flags for roots of nonlinear functions

/B [= *doBracket*] Specifies bracketing for roots of a single nonlinear function only.

doBracket=0: Skips an initial check of the root bracketing values and the possible search for bracketing values. This means that you must provide good bracketing values via the /L and /H flags. See /L and /H flags for details on bracketing of roots. /B alone is the same as /B=0.

doBracket=1: Uses default root bracketing.

/F=*trustRegion* Sets the expansion factor of the trust region for the search algorithm when finding roots of systems of functions. Smaller numbers will result in a more stable search, although for some functions larger values will allow the search to zero in on a root more rapidly. Default is 1.0; useful values are usually between 0.1 and 100.

/I=*maxIters* Sets the maximum number of iterations in searching for a root to *maxIters*. Default is 100.

/L=*lowBracket*
/H=*highBracket* /L and /H are used only when finding roots of a single nonlinear function. *lowBracket* and *highBracket* are X values that bracket a zero crossing of the function. A root is found between the bracketing values.

If *lowBracket* and *highBracket* are on the same side of zero, it will try to find a minimum or maximum between *lowBracket* and *highBracket*. If it is found, and it is on the other side of zero, Igor will find two roots.

If *lowBracket* and *highBracket* are on the same side of zero, but no suitable extreme point is found between, it will search outward from these values looking for a zero crossing. If it is found, Igor determines one root.

If *lowBracket* and *highBracket* are equal, it adds 1.0 to *highBracket* before looking for a zero crossing.

The default values for *lowBracket* and *highBracket* are zero. Thus, not using either *lowBracket* or *highBracket* is the same as /L=0/H=1.

/Q Suppresses printout of results in the history area. Ordinarily, the results of root searches are printed in the history.

/T=*tol* Sets the acceptable accuracy to *tol*. That is, the reported root should be within ±*tol* of the real root.

/X=*xWave*
/X={*x1*, *x2*, …} Sets the starting point for searching for a root of a system of functions. There must be as many X values as functions. The starting point can be specified with a wave having as many points as there are functions, or you can write out a list of X values in braces. If you are finding roots of a single function, use /L and /H instead.

If you specify a wave, this wave is also used to receive the result of the root search.

/Z=*yValue* Finds other solutions, that is, places where f(x) = *yValue*. FindRoots usually finds zeroes — places where f(x) = 0.

Flag for roots of polynomials

/P=*PolyCoefsWave* Specifies a wave containing real polynomial coefficients. With this flag, it finds polynomial roots and does not expect to find user function names on the command line.

The /P flag causes all other flags to be ignored.

Use of this flag is not permitted in a thread-safe function.

Parameters

func specifies the name of a user-defined function.

pwave gives the name of a parameter wave that will be passed to your function as the first parameter. It is not modified. It is intended for your private use to pass adjustable constants to your function.

These parameters occur in pairs. For a one-dimensional problem, use a single *func*, *pwave* pair. An N-dimensional problem requires N pairs unless you use the combined function form (see **Combined Format for Systems of Functions**).

Function Format for 1D Nonlinear Functions

Finding roots of a nonlinear function or system of functions requires that you realize the function in the form of an Igor user function of a certain form. In the FindRoots command you then specify the functions with one or more function names paired with parameter wave names. See **Finding Function Roots** on page III-283 for detailed examples.

The functions must have a particular form. If you are finding the roots of a single 1D function, it should look like this:

```
Function myFunc(w,x)
    Wave w
    Variable x

    return f(x)          // an expression …
End
```

Replace "`f(x)`" with an appropriate expression. The FindRoots command might then look like this:

```
FindRoots /L=0 /H=1 myFunc, cw       // cw is a parameter wave for myFunc
```

Function Format for Systems of Multivariate Functions

If you need to find the roots of a system of multidimensional functions, you can use either of two forms. In one form, you provide N functions with N independent variables. You must have a function for each independent variable. For instance, to find the roots of two 2D functions, the functions must have this form:

```
Function myFunc1(w, x1, x2)
    Wave w
    Variable x1, x2

    return f1(x1, x2)       // an expression …
End

Function myFunc2(w, x1, x2)
    Wave w
    Variable x1, x2

    return f2(x1, x2)       // an expression …
End
```

In this case, the FindRoots command might look like this (where `cw1` and `cw2` are parameter waves that must be made before executing FindRoots):

```
FindRoots /X={0,1} myFunc1, cw1, myFunc2, cw2
```

You can also use a wave to pass in the X values. Make sure you have the right number of points in the X wave — it must have N points for a system of N functions.

```
Function myFunc1(w, xW)
    Wave w, xW

    return f1(xW[0], xW[1])        // an expression …
End

Function myFunc2(w, xW)
    Wave w, xW

    return f2(xW[0], xW[1])        // an expression …
End
```

Combined Format for Systems of Functions

For large systems of equations it may get tedious to write a separate function for each equation, and the FindRoots command line will get very long. Instead, you can write it all in one function that returns N Y values through a Y wave. The X values are passed to the function through a wave with N elements. The parameter wave for such a function must have N columns, one column for each equation. The parameters for equation N are stored in column N-1. FindRoots will complain if any of these waves has other than N rows.

Here is an template for such a function:

```
Function myCombinedFunc(w, xW, yW)
    Wave w, xW, yW

    yW[0] = f1(w[0][...], xW[0], xW[1],..., xW[N-1])
    yW[1] = f2(w[1][...], xW[0], xW[1],..., xW[N-1])
```

```
    …
    yW[N-1] = fN(w[N-1][...], xW[0], xW[1],..., xW[N-1])
End
```

When you use this form, you only have one function and parameter wave specification in the FindRoots command:

```
Make/N=(nrows, nequations) paramWave
fill in paramWave with values
Make/N=(number of equations) guessWave
guessWave = {x0, x1, …, xN}
FindRoots /X=guessWave myCombinedFunc, paramWave
```

FindRoots has no idea how many actual equations you have in the function. If it doesn't match the number of rows in your waves, your results will not be what you expect!

Coefficients for Polynomials

To find the roots of a polynomial, you first create a wave with the correct number of points. For a polynomial of degree N, create a wave with N+1 points. For instance, to find roots of a cubic equation you need a four-point wave.

The first point (row zero) of the wave contains the constant coefficient, the second point contains the coefficient for X, the third for X^2, etc.

There is no hard limit on the maximum degree, but note that there are significant numerical problems associated with computations involving high-degree polynomials. Round-off error most likely limits reasonably accurate results to polynomials with degree limited to 20 to 30.

Ultimately, if you are willing to accept very limited accuracy, the numerical problems will result in a failure to converge. In limited testing, we found no failures to converge with polynomials up to at least degree 100. At degree 150, we found occasional failures. At degree 200 the failures were frequent, and at degree 500 we found no successes.

Note that you really can't evaluate a polynomial with such high degree, and we have no idea if the computed roots for a degree-100 polynomial have any practical relationship to the actual roots.

While FindRoots is a thread-safe operation, finding polynomial roots is not. Using FindRoots/P=polyWave in a ThreadSafe function results in a compile error.

Results for Nonlinear Functions and Systems of Functions

The FindRoots operation reports success or failure via the V_flag variable. A nonzero value of V_flag indicates the reason for failure:

`V_flag`	0:	Successful search for a root. Otherwise, the value indicates what went wrong:
	1:	User abort.
	3:	Exceeded maximum allowed iterations
	4:	/T=*tol* was too small. Reported by the root finder for systems of nonlinear functions.
	5:	The search algorithm wasn't making sufficient progress. It may mean that /T=*tol* was set to too low a value, or that the search algorithm has gotten trapped at a false root. Try restarting from a different starting point.
	6:	Unable to bracket a root. Reported when finding roots of single nonlinear functions.
	7:	Fewer roots than expected. Reported by the polynomial root finder. This may indicate that roots were successfully found, but some are doubled. Happens only rarely.
	8:	Decreased degree. Reported by the polynomial root finder. This indicates that one or more of the highest-order coefficients was zero, and a lower degree polynomial was solved.
	9:	Convergence failure or other numerical problem. Reported by the polynomial root finder. This indicates that a numerical problem was detected during the computation. The results are not valid.

The results of finding roots of a single 1D function are put into several variables:

`V_numRoots`	The number of roots found. Either 1 or 2.

V_Root	The root.
V_YatRoot	The Y value of the function at the root. *Always* check this; some discontinuous functions may give an indication of success, but the Y value at the found root isn't even close to zero.
V_Root2	Second root if FindRoots found two roots.
V_YatRoot2	The Y value at the second root.

Results for roots of a system of nonlinear functions are reported in waves:

W_Root	X values of the root of a system of nonlinear functions. If you used /X=*xWave*, the root is reported in your wave instead.
W_YatRoot	The Y values of the functions at the root of a system of nonlinear functions. Only one root is found during a single call to FindRoots.

Roots of a polynomial are reported in a wave:

W_polyRoots	A complex wave containing the roots of a polynomial. The number of roots should be equal to the degree of the polynomial, unless a root is doubled.

See Also

Finding Function Roots on page III-283.

The FindRoots operation uses the Jenkins-Traub algorithm for finding roots of polynomials:

Jenkins, M.A., Algorithm 493, Zeros of a Real Polynomial, *ACM Transactions on Mathematical Software*, *1*, 178-189, 1975. Used by permission of ACM (1998).

FindSequence

FindSequence [***flags***] ***srcWave***

The FindSequence operation finds the location of the specified sequence starting the search from the specified start point. The result of the search stored in V_value is the index of the entry in the wave where the first value is found or -1 if the sequence was not found.

Flags

/I=*wave*	Specifies an integer sequence wave for integer search.
/M=*val*	If there are repeating entries in the match sequence, *val* is a tolerance value that specifies the maximum difference between the number of repeats. So, for example, if the match sequence is aaabbccc and the *srcWave* contains a sequence aabbcc then the sequence will not be considered a match if *val*=0 but will be considered a match if *val*=1.
/S=*start*	Sets starting point of the search. If /S is not specified, start is 0.
/T=*tolerance*	Defines the tolerance (value ± *tolerance* will be accepted) when comparing floating point numbers.
/U=*uValue*	Specifies the match sequence wave in case of unsigned long range.
/V=*rValue*	Specifies the match sequence wave in the case of single/double precision numbers.
/Z	No error reporting.

Details

If the match sequence is specified via the /V flag, it is considered to be a floating point wave (i.e., single or double precision) in which case it is compared to data in the wave using a tolerance value. If the tolerance is not specified by the /T flag, the default value 1.0^{-7}.

If the match sequence is specified via the /I flag, the sequence is assumed to be an integer wave (this includes both signed and unsigned char, signed and unsigned short as well as long). In this case *srcWave* must also be of integer type and the operation searches for the sequence based on exact equality between the match sequence and entries in the wave as signed long integers.

If the match sequence is unsigned long wave use the /U flag to specify the value for an integer comparison.

You can also use this operation on waves of two or more dimensions. In this case you can calculate the rows, columns, etc. For example, in the case of a 2D wave:

```
col=floor(V_value/rowsInWave)
row=V_value-col*rowsInWave
```

See Also
The **FindValue** operation.

FindValue

FindValue [***flags***] ***srcWave***
FindValue [***flags***] ***txtWave***

This operation finds the location of the specified value starting the search from the specified start point. The result of the search stored in V_value is the index of the entry in the wave where the value is found or -1 if not found.

Flags

/I=*ivalue* Specifies an integer value for integer search.

/S=*start* Sets start of search in the wave. If /S is not specified, start is set to 0.

/T=*tolerance* Use this flag when comparing floating point numbers to define a non-negative tolerance such that the specified value ± *tolerance* will be accepted.

/TEXT=*templateString*
Specifies a template string that will be searched for in *txtWave*.

/TXOP=*txOptions* Specifies the search options using a combination of binary values

txOptions	Search Type
1	Case sensitive
2	Whole word
4	Whole wave element

/U=*uValue* Specifies the match value in case of unsigned long range.

/V=*rValue* Specifies the match value in the case of single/double precision numbers.

/Z No error reporting.

Details

If the match value is specified via the /V flag, it is considered to be a floating point value in which case it is compared to data in the wave using a tolerance value. If the tolerance is not specified by the /T flag, the value 10^{-7} is used.

If the match value is specified via the /I flag, the value is assumed to be an integer. In this case *srcWave* must be of integer type and the operation searches for the value based on exact equality between the match value and entries in the wave as signed long integers.

If the match value is unsigned long use the /U flag to specify the value for an integer comparison.

You can also use this operation on waves of two or more dimensions. In this case you can calculate the rows, columns, etc. For example, in the case of a 2D wave:

```
col=floor(V_value/rowsInWave)
row=V_value-col*rowsInWave
```

When searching for text in a text wave the operation creates the variable V_value as above but it also creates the variable V_startPos to specify the position of *templateString* from the start of the particular wave element.

See Also
The **FindSequence**, **FindLevel** and **FindLevels** operations.

FitFunc

FitFunc

Marks a user function as a user-defined curve fit function. By default, only functions marked with this keyword are displayed in the Function menu in the Curve Fit dialog.

If you wish other functions to be displayed in the Function menu, you can select the checkbox labelled "Show old-style functions (missing FitFunc keyword)".

See Also
User-Defined Fitting Function: Detailed Description on page III-217.

floor

floor(*num*)
The floor function returns the closest integer less than or equal to *num*.

See Also
The **round**, **ceil**, and **trunc** functions.

FontList

FontList(*separatorStr* [, *options*])
The FontList function returns a list of the installed fonts, separated by the characters in *separatorStr*.

Parameters
A maximum of 10 characters from *separatorStr* are appended to each font name as the output string is generated. *separatorStr* is usually ";".

Use *options* to limit the returned font list according to font type. It is restricted to returning only scalable fonts (TrueType, PostScript, or OpenType), which you can do with *options* = 1.

To get a list of nonscalable fonts (bitmap or raster), use:

```
String bitmapFontList = RemoveFromList(FontList(";",1), FontList(";"))
```

(Most Mac OS X fonts are scalable, so bitmapFontList may be empty.)

Examples

```
Function SetFont(fontName)
    String fontName
    Prompt fontName,"font name:",popup,FontList(";")+"default;"
    DoPrompt "Pick a Font", fontName

    Print fontName

    Variable type= WinType("")                  // target window type
    String windowName= WinName(0,127)
    if((type==1) || (type==3) || (type==7))     // graph, panel, layout
        Print "Setting drawing font for "+windowName
        Execute "SetDrawEnv fname=\""+fontName+"\""
    else
        if( type == 5 )                         // notebook
            Print "Setting font for selection in "+windowName
            Notebook $windowName font=fontName
        endif
    endif
End
```

Execute on the command line:

```
SetFont("")
```

See Also
The **FontSizeStringWidth**, **FontSizeHeight**, and **WinType** functions, and the **Execute**, **SetDrawEnv**, and **Notebook** Operations.

FontSizeHeight

FontSizeHeight(*fontNameStr, fontSize, fontstyle* [,*appearanceStr*])
The FontSizeHeight function returns the line height in pixels of any string when rendered with the named font and the given font style and size.

Parameters
fontNameStr is the name of the font, such as `"Helvetica"`.

fontSize is the size (height) of the font in pixels.

fontStyle is text style (bold, italic, etc.). Use 0 for plain text.

Details

The returned height is the sum of the font's ascent and descent heights. Variations in *fontStyle* and typeface design cause the actual font height to be different than *fontSize* would indicate. (Typically a font "height" refers to only the ascent height, so the total height will be slightly larger to accommodate letters that descend below the baseline, such as g, p, q, and y).

FontSize is in pixels. To obtain the height of a font specified in points, use the **ScreenResolution** function and the conversion factor of 72 points per inch (see Examples).

If the named font is not installed, FontSizeHeight returns NaN.

FontSizeHeight understands "default" to mean the current experiment's default font.

fontStyle is a binary coded integer with each bit controlling one aspect of the text style as follows:

bit 0:	Bold.
bit 1:	Italic.
bit 2:	Underline.
bit 3:	Outline (*Macintosh only*).
bit 4:	Shadow (*Macintosh only*).

To set bit 0 and bit 2 (bold, underline), use $2^0+2^2 = 1+4 = 5$ for *fontStyle*. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The optional *appearanceStr* parameter has no effect on Windows.

On Macintosh, the *appearanceStr* parameter is used for determining the height of a string drawn by a control. Set *appearanceStr* to "native" if you are measuring the height of a string drawn by a "native GUI" control or to "os9" if not.

Set *appearanceStr* to "default" to use the appearance set by the user in the Miscellaneous Settings dialog. "os9" is the default value.

Usually you will want to set *appearanceStr* to the S_Value output of **DefaultGUIControls**/W=winName when determining the height of a string drawn by a control.

Examples

```
Variable pixels= 12 * ScreenResolution/72         // convert 12 points to pixels
Variable pixelHeight= FontSizeHeight("Helvetica",pixels,0)
Print "Height in points= ", pixelHeight * 72/ScreenResolution
```

```
Function FontIsInstalled(fontName)
   String fontName
   if( numtype(FontSizeHeight(fontName,10,0)) == 2 )
      return 0            // NaN returned, font not installed
   else
      return 1
   endif
End
```

See Also

The **FontList**, **FontSizeStringWidth**, **numtype**, **ScreenResolution**, and **DefaultGUIControls** functions.

FontSizeStringWidth

FontSizeStringWidth(*fontNameStr, fontSize, fontstyle, theStr* [,*appearanceStr*])

The FontSizeStringWidth function returns the width of *theStr* in pixels, when rendered with the named font and the given font style and size.

Parameters

fontNameStr is the name of the font, such as `"Helvetica"`.

fontSize is the size (height) of the font in pixels.

fontStyle is text style (bold, italic, etc.). Use 0 for plain text.

theStr is the string whose width is being measured.

The optional *appearanceStr* parameter has no effect on Windows.

On Macintosh, the *appearanceStr* parameter is used for determining the width of a string drawn by a control. Set *appearanceStr* to "native" if you are measuring the width of a string drawn by a "native GUI" control or to "os9" if not.

Set *appearanceStr* to "default" to use the appearance set by the user in the Miscellaneous Settings dialog. "os9" is the default value.

Usually you will want to set *appearanceStr* to the S_Value output of **DefaultGUIControls**/W=winName when determining the width of a string drawn by a control.

Details

If the named font is not installed, FontSizeStringWidth returns NaN.

FontSizeStringWidth understands "default" to mean the current experiment's default font.

FontSize is in pixels. To obtain the width of a font specified in points, use the **ScreenResolution** function and the conversion factor of 72 points per inch (see Examples).

fontStyle is a binary coded integer with each bit controlling one aspect of the text style as follows:

bit 0:	Bold.
bit 1:	Italic.
bit 2:	Underline.
bit 3:	Outline (*Macintosh only*).
bit 4:	Shadow (*Macintosh only*).

To set bit 0 and bit 2 (bold, underline), use $2^0+2^2 = 1+4 = 5$ for *fontStyle*. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Example 1

```
Variable fsPix= 10 * ScreenResolution/72          // 10 point text in pixels
String text= "How long is this text?"
Variable WidthPix= FontSizeStringWidth("Helvetica",fsPix,0,text)
Print "width in inches= ", WidthPix / ScreenResolution
```

Example 2

```
Variable fsPix= 13 * ScreenResolution/72          // 13 point text in pixels
String text= "text for a control"
DefaultGUIControls/W=Panel0                       // Sets S_Value
Variable WidthPix= FontSizeStringWidth("Helvetica",fsPix,0,text,S_Value)
Print "width in points= ", WidthPix / ScreenResolution * 72
```

See Also

The **FontList**, **FontSizeHeight**, **ScreenResolution** and **DefaultGUIControls** functions.

for-endfor

```
for(<initialize>;<exit test>;<update>)
   <loop body>
endfor
```

A for-endfor loop executes *loop body* code until *exit test* evaluates as FALSE, zero, or until a break statement is executed within the body code. When the loop starts, the *initialize* expressions are evaluated once. For each iteration, the *exit test* is evaluated at the beginning, and the *update* expressions are evaluated at the end.

See Also

For Loop on page IV-37 and **break** for more usage details.

FPClustering

```
FPClustering [flags] srcWave
```

The FPClustering operation performs cluster analysis using the farthest-point clustering algorithm. The input for the operation *srcWave* defines M points in N-dimensional space. Outputs are the waves W_FPCenterIndex and W_FPClusterIndex.

Flags

/CAC	Computes all the clusters specified by /MAXC.
/CM	Computes the center of mass for each cluster. The results are stored in the wave M_clustersCM in the current data folder. Each row corresponds to a single cluster with columns providing the respective dimensional components.
/INCD	Computes the inter-cluster distances. The result is stored in the current data folder in the wave M_InterClusterDistance, a 2D wave in which the [*i*][*j*] element contains the distance between cluster *i* and cluster *j*.
/MAXC=*nClusters*	Terminates the calculation when the number of clusters reaches the specified value. Note that this termination condition is sufficient but not necessary, i.e., the operation can terminate earlier if the farthest distance of an element from a hub is less than the average distance.
/MAXR=*maxRad*	Terminates the calculation when the maximum distance is less than or equal to *maxRad*.
/NOR	Normalizes the data on a column by column basis. The normalization makes each columns of the input span the range [0,1] so that even when *srcWave* contains columns that may be different by several orders of magnitude, the algorithm is not biased by a larger implied cartesian distance.
/Q	Don't print information to the history window.
/SHUB=*sHub*	Specifies the row which is used as a starting hub number. By default the operation uses the first row in *srcWave*.
/Z	No error reporting.

Details

The input for FPClustering is a 2D wave *srcWave* which consists of M rows by N columns where each row represents a point in N-dimensional space. *srcWave* can contain only finite real numbers and must be of type SP or DP. The operation computes the clustering and produces the wave W_FPCenterIndex which contains the centers or "hubs" of the clusters. The hubs are specified by the zero-based row number in *srcWave* which contains the cluster center. In addition, the operation creates the wave W_FPClusterIndex where each entry maps the corresponding input point to a cluster index. By default, the operation continues to add clusters as long as the largest possible distance is greater than the average intercluster distance. You can also stop the processing when the operation has formed a specified number of clusters (see /MAXC).

The variable V_max contains the maximum distance between any element and its cluster hub.

It is possible that in some circumstances you can get slightly different clustering depending on your starting point. The default starting hub is row zero of *srcWave* but you can use the /SHUB flag to specify a different starting point.

FPClustering computes the Cartesian distance between points. As a result, if the scale of any dimension is significantly larger than other dimensions it might bias the clustering towards that dimension. To avoid this situation you can use the /NOR flag which normalizes each column to the range [0,1] and hence equalizes the weight of each dimension in the clustering process.

See Also

The **KMeans** operation.

References

Gonzalez, T., Clustering to minimize the maximum intercluster distance, *Theoretical Computer Science*, *38*, 293-306, 1985.

fprintf

fprintf *refNum*, *formatStr* [, *parameter*]…

The fprintf operation prints formatted output to a text file.

Parameters

refNum is a file reference number from the **Open** operation used to open the file.

formatStr is the format string, as used by the **printf** operation.

parameter varies depending on *formatStr*.

Details

If *refNum* is 1, fprintf will print to the history area instead of to a file, as if you used printf instead of fprintf. This useful for debugging purposes.

A zero value of *refNum* is used in conjunction with Program-to-Program Communication (PPC), Apple events (*Macintosh*) or DDE (*Windows*). Data that would normally be written to a file is appended to the PPC, Apple event or DDE result packet.

The fprintf operation supports numeric (real only) and string fields from structures. All other field types will cause a compile error.

See Also

The **printf** operation for complete format and parameter descriptions. The **Open** operation and **Creating Formatted Text** on page IV-230.

FReadLine

FReadLine [**/N/T**] ***refNum, stringVarName***

The FReadLine operation reads bytes from a file into the named string variable. The read starts at the current file position and continues until a terminator character is read, the end of the file is reached, or the maximum number of characters is read.

Parameters

refNum is a file reference number from the **Open** operation used to create the file.

Flags

/N=*n*	Specifies the maximum number of characters to read.
/T=*termcharStr*	Specifies the terminator character.
	`/T=(num2char(13))` specifies carriage return (CR, ASCII code 13).
	`/T=(num2char(10))` specifies linefeed (LF, ASCII code 10).
	`/T=";"` specifies the terminator as a semicolon.
	`/T=""` specifies the terminator as null (ASCII code 0).
	See **Details** for default behavior regarding the terminator.

Details

If /N is omitted, there is no maximum number of characters to read. When reading lines of text from a normal text file, you will not need to use /N. It may be of use in specialized cases, such as reading text embedded in a binary file.

If /T is omitted, FReadLine will terminate on any of the following: CR, LF, CRLF, LFCR. (Most Macintosh files use CR. Most Windows files use CRLF. Most UNIX files use LF. LFCR is an invalid terminator but some buggy programs generate files that use it.) FReadLine reads whichever of these appears in the file, terminates the read, and returns just a CR in the output string. This default behavior transparently handles files that use CR, LF, CRLF, or LFCR as the terminator and will be suitable for most cases.

If you use the /T flag, then FReadLine will terminate on the specified character only and will return the specified character in the output string.

Once you have read all of the characters in the file, FReadLine will return zero characters via *stringVarName*. The example below illustrates testing for this.

Example

```
Function PrintAllLinesInFile()
    Variable refNum
    Open/R refNum as ""           // Display dialog
    if (refNum == 0)
        return -1                 // User canceled
    endif

    Variable lineNumber, len
    String buffer
    lineNumber = 0
    do
        FReadLine refNum, buffer
        len = strlen(buffer)
        if (len == 0)
```

```
            break                   // No more lines to be read
        endif
        Printf "Line number %d: %s", lineNumber, buffer
        if (CmpStr(buffer[len-1],"\r") != 0)       // Last line has no CR ?
            Printf "\r"
        endif
    lineNumber += 1
    while (1)

    Close refNum
    return 0
End
```

See Also

The **Open** and **FBinRead** operations.

fresnelCos

fresnelCos(*x*)

The fresnelCos function returns the Fresnel cosine function $C(x)$.

$$C(x) = \int_0^x \cos\left(\frac{\pi}{2}v^2\right) dv .$$

See Also

The **fresnelSin** and **fresnelCS** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

fresnelCS

fresnelCS(*x*)

The fresnelCS function returns both the Fresnel cosine in the real part of the result and the Fresnel sine in the imaginary part of the result.

See Also

The **fresnelSin** and **fresnelCos** functions.

fresnelSin

fresnelSin(*x*)

The fresnelSin function returns the Fresnel sine function $S(x)$.

$$S(x) = \int_0^x \sin\left(\frac{\pi}{2}v^2\right) dv .$$

See Also

The **fresnelCos** and **fresnelCS** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

FSetPos

FSetPos *refNum, filePos*

The FSetPos operation attempts to set the current file position to the given position.

Parameters

refNum is a file reference number obtained from the **Open** operation when the file was opened.

filePos is the desired position of the file in bytes from the start of the file.

Details

FSetPos generates an error if *filePos* is greater than the number of bytes in the file. You can ascertain this limit with the **FStatus** operation.

When a file that is open for writing is closed, any bytes past the end of the current file position are deleted by the operating system. Therefore, if you use FSetPos, make sure to set the current file position properly before closing the file.

As of Igor Pro 6.1, FSetPos supports very big files theoretically up to about 4.5E15 bytes in length.

See Also

The **Open** and **FStatus** operations.

FStatus

FStatus *refNum*

The FStatus operation provides file status information for a file.

Parameters

refNum is a file reference number obtained from the **Open** operation.

Details

As of Igor Pro 6.1, FStatus supports very big files theoretically up to about 4.5E15 bytes in length.

FStatus sets the following variables:

V_flag	Nonzero (true) if *refNum* is valid, in which case FStatus sets the other variables as well.
V_filePos	Current file position for the file in bytes from the start.
V_logEOF	Total number of bytes in the file.
S_fileName	Name of the file.
S_path	Path from the volume to the folder containing the file. For example, "hd:Folder1:Folder2:". This is suitable for use as an input to the **NewPath** operation. Note that on the Windows operating system Igor uses a colon between folders instead of the Windows-standard backslash to avoid confusion with Igor's use of backslash to start an escape sequence (see **Escape Characters in Strings** on page IV-13).
S_info	Keyword-packed information string.

The keyword-packed information string for S_info consists of a sequence of sections with the following form: *keyword*:*value*; You can pick a value out of a keyword-packed string using the **NumberByKey** and **StringByKey** functions.

Here are the keywords for S_info:

Keyword	Type	Meaning
PATH	string	Name of the symbolic path in which the file is located. This will be empty if there is no such symbolic path.
WRITEABLE	number	1 if file can be written to, 0 if not.

See Also

The **Open** operation.

FTPCreateDirectory

FTPCreateDirectory [***flags***] ***urlStr***

The FTPCreateDirectory operation creates a directory on an FTP server on the Internet.

For background information on Igor's FTP capabilities and other important details, see **File Transfer Protocol (FTP)** on page IV-244.

FTPCreateDirectory sets V_flag to zero if the operation succeeds or to a non-zero error code if it fails.

If the directory specified by *urlStr* already exists on the server, the server contents are not touched and V_flag is set to -1. This is not treated as an error.

Parameters

urlStr specifies the directory to create. It consists of a naming scheme (always "ftp://"), a computer name (e.g., "ftp.wavemetrics.com" or "38.170.234.2"), and a path (e.g., "/test/newDirectory"). For example:

```
"ftp://ftp.wavemetrics.com/test/newDirectory"
```

urlStr must always end with a directory name, and must not end with a slash.

To indicate that *urlStr* contains an absolute path, insert an extra '/' character between the computer name and the path. For example:

```
ftp://ftp.wavemetrics.com//pub/test/newDirectory
```

If you do not specify that the path in *urlStr* is absolute, it is interpreted as relative to the FTP user's base directory. Since pub is the base directory for an anonymous user at wavemetrics.com, these URLs reference the same directory for an anonymous user:

```
ftp://ftp.wavemetrics.com//pub/test/newDirectory  // Absolute path
ftp://ftp.wavemetrics.com/test/newDirectory       // Relative to base directory
```

Special characters, such as punctuation, that are used in *urlStr* may be incorrectly interpreted by the operation. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. See **Percent Encoding** on page IV-239 for additional information.

Flags

/N=*portNumber*
: Specifies the server's TCP/IP port number to use (default is 21). In almost all cases, the default will be correct so you won't need to use the /N flag.

/U=*userNameStr*
: Specifies the user name to be used when logging in to the FTP server. If /U is omitted or if *userNameStr* is "", the login is done as an anonymous user. Use /U if you have an account on the FTP server.

/V=*diagnosticMode*
: Determines what kind of diagnostic messages FTPCreateDirectory will display in the history area. *diagnosticMode* is a bitwise parameter, with the bits defined as follows:

  Bit 0: Show basic diagnostics. Currently this just displays the URL in the history.

  Bit 1: Show errors. This displays additional information when errors occur.

  Bit 2: Show status. This displays commands sent to the server and the server's response.

  The default value for *diagnosticMode* is 3 (show basic and error diagnostics). If you are having difficulties, you can try using 7 to show the commands sent to the server and the server's response.

  See **FTP Troubleshooting** on page IV-247 for other troubleshooting tips.

/W=*passwordStr*
: Specifies the password to be used when logging in to the FTP server. Use /W if you have an account on the FTP server.

  If /W is omitted, the login is done using a default password that will work with most anonymous FTP servers.

  Note: See **Safe Handling of Passwords** on page IV-241 for information on handling sensitive passwords.

/Z
: Errors are not fatal. Will not abort procedure execution if an error occurs.

  Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded, -1 if the specified directory already exists, or another nonzero value if an error occurred.

Examples

```
// Create a directory.
String url = "ftp://ftp.wavemetrics.com/pub/test/newDirectory"
FTPCreateDirectory url
```

See Also

File Transfer Protocol (FTP) on page IV-244.

FTPDelete, **FTPDownload**, **FTPUpload**, **URLEncode**

FTPDelete

FTPDelete [***flags***] ***urlStr***

The FTPDelete operation deletes a file or a directory from an FTP server on the Internet.

Warning: If you delete a directory on an FTP server, all contents of that directory and any subdirectories are also deleted.

For background information on Igor's FTP capabilities and other important details, see **File Transfer Protocol (FTP)** on page IV-244.

FTPDelete sets V_flag to zero if the operation succeeds and to nonzero if it fails. This, in conjunction with the /Z flag, can be used to allow procedures to continue to execute if an FTP error occurs.

Parameters

urlStr specifies the file or directory to delete. It consists of a naming scheme (always "ftp://"), a computer name (e.g., "ftp.wavemetrics.com" or "38.170.234.2"), and a path (e.g., "/test/TestFile1.txt"). For example: "ftp://ftp.wavemetrics.com/test/TestFile1.txt"

urlStr must always end with a file name if you are deleting a file or with a directory name if you are deleting a directory. In the case of a directory, *urlStr* must not end with a slash.

To indicate that *urlStr* contains an absolute path, insert an extra '/' character between the computer name and the path. For example:

```
ftp://ftp.wavemetrics.com//pub/test
```

If you do not specify that the path in *urlStr* is absolute, it is interpreted as relative to the FTP user's base directory. Since pub is the base directory for an anonymous user at wavemetrics.com, these URLs reference the same directory for an anonymous user:

```
ftp://ftp.wavemetrics.com//pub/test
ftp://ftp.wavemetrics.com/test
```

Special characters such as punctuation that are used in *urlStr* may be incorrectly interpreted by the operation. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. See **Percent Encoding** on page IV-239 for additional information

Flags

/D — Deletes a complete directory and all its contents. Omit /D if you are deleting a file.

/N=*portNumber* — Specifies the server's TCP/IP port number to use (default is 21). In almost all cases, the default will be correct so you won't need to use the /N flag.

/U=*userNameStr* — Specifies the user name to be used when logging in to the FTP server. If /U is omitted or if userNameStr is "", the login is done as an anonymous user. Use /U if you have an account on the FTP server.

/V=*diagnosticMode* — Determines what kind of diagnostic messages FTPCreateDirectory will display in the history area. *diagnosticMode* is a bitwise parameter, with the bits defined as follows:

Bit 0: Show basic diagnostics. Currently this just displays the URL in the history.

Bit 1: Show errors. This displays additional information when errors occur.

Bit 2: Show status. This displays commands sent to the server and the server's response.

The default value for *diagnosticMode* is 3 (show basic and error diagnostics). If you are having difficulties, you can try using 7 to show the commands sent to the server and the server's response.

See **FTP Troubleshooting** on page IV-247 for other troubleshooting tips.

/W=*passwordStr* — Specifies the password to be used when logging in to the FTP server. Use /W if you have an account on the FTP server.

If /W is omitted, the login is done using a default password that will work with most anonymous FTP servers.

Note: See **Safe Handling of Passwords** on page IV-241 for information on handling sensitive passwords.

/Z — Errors are not fatal. Will not abort procedure execution if an error occurs.

Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded, or a nonzero value if an error occurred.

Examples

```
// Delete a file.
String url = "ftp://ftp.wavemetrics.com/test/TestFile1.txt"
FTPDelete url

// Delete a directory.
String url = "ftp://ftp.wavemetrics.com/test/TestDir1"
FTPDelete/D url
```

See Also

File Transfer Protocol (FTP) on page IV-244.

FTPCreateDirectory, **FTPDownload**, **FTPUpload**, **URLEncode**

FTPDownload

FTPDownload [*flags*] *urlStr*, *localPathStr*

The FTPDownload operation downloads a file or a directory from an FTP server on the Internet.

Warning: When you download a file or directory using the path and name of a file or directory that already exists on your local hard disk, all previous contents of the local file or directory are obliterated.

For background information on Igor's FTP capabilities and other important details, see **File Transfer Protocol (FTP)** on page IV-244.

FTPDownload sets a variable named V_flag to zero if the operation succeeds and to nonzero if it fails. This, in conjunction with the /Z flag, can be used to allow procedures to continue to execute if a FTP error occurs.

If the operation succeeds, FTPDownload sets a string named S_Filename to the full file path of the downloaded file or, if the /D flag was used, the full path to the base directory that was downloaded. This is useful in conjunction with the /I flag.

If the operation fails, S_Filename is set to "".

Parameters

urlStr specifies the file or directory to download. It consists of a naming scheme (always `"ftp://"`), a computer name (e.g., `"ftp.wavemetrics.com"` or `"38.170.234.2"`), and a path (e.g., `"/Test/TestFile1.txt"`). For example: `"ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"`.

urlStr must always end with a file name if you are downloading a file or with a directory name if you are downloading a directory. In the case of a directory, *urlStr* must not end with a slash.

To indicate that *urlStr* contains an absolute path, insert an extra '/' character between the computer name and the path. For example:

```
ftp://ftp.wavemetrics.com//pub/test
```

If you do not specify that the path in *urlStr* is an absolute path, it is interpreted as a path relative to the FTP user's base directory. Since pub is the base directory for an anonymous user, this URL references the same directory:

```
ftp://ftp.wavemetrics.com/test
```

Special characters such as punctuation that are used in *urlStr* may be incorrectly interpreted by the operation. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. See **Percent Encoding** on page IV-239 for additional information.

localPathStr and *pathName* specify the name to use for the file or directory that will be created on your hard disk. If you use a full or partial path for *localPathStr*, see **Path Separators** on page III-398 for details on forming the path.

localPathStr must always end with a file name if you are downloading a file or with a directory name if you are downloading a directory. In the case of a directory, *localPathStr* must not end with a colon or backslash.

FTPDownload displays a dialog through which you can identify the local file or directory in the following cases:

1. You have used the /I (interactive) flag.
2. You did not completely specify the location of the local file or directory via *pathName* and *localPathStr*.

3. There is an error in *localPathStr*. This can be either a syntactical error or a reference to a nonexistent file or directory.
4. The specified local file or directory exists and you have not used the /O (overwrite) flag.

See **Examples** for examples of constructing a URL and local path.

Flags

/D — Downloads a complete directory. Omit it if you are downloading a file.

/I — Interactive mode which will prompt you to specify the name and location of the file or directory to be created on the local hard disk.

/M=*messageStr* — Specifies the prompt message used by the dialog in which you specify the name and location of the file or directory to be created.

/N=*portNumber* — Specifies the server's TCP/IP port number to use (default is 21). In almost all cases, this will be correct so you won't need to use the /N flag.

/O[=*mode*] — Controls whether a local file or directory whose name is in conflict with the file or directory being downloaded is overwritten without prompting the user.

- 0: Prompts the user to allow the overwrite. This is the default behavior if /O is omitted.
- 1: Overwrites without prompting the user. If the /D flag is also used, all contents of the destination directory are deleted if it already exists. /O=1 is the same as /O.
- 2: Merges files and subdirectories downloaded with the contents of the destination directory. Unlike /O=1, the contents of the destination directory are not deleted, however files and directories downloaded from the server will overwrite existing files and directories of the same name. When downloading a file this mode is accepted but has the same effect as /O=1.

/P=*pathName* — Contributes to the specification of the file or directory to be created on your hard disk. *pathName* is the name of an existing symbolic path. See **Examples**.

/S=*showProgress* — Displays a progress dialog.

- 0: No progress dialog.
- 1: Show a progress dialog (default).

/T=*transferType* — Transfers in image binary mode.

- 0: Image (binary) transfer (default).
- 1: ASCII transfer.

See **FTP Transfer Types** on page IV-247 for more discussion.

/U=*userNameStr* — Specifies the user name to be used when logging in to the FTP server. If this flag is omitted or if *userNameStr* is `""`, you will be logged in as an anonymous user. Use this flag if you have an account on the FTP server.

/V=*diagnosticMode* — Determines what kind of diagnostic messages FTPDownload will display in the history area. *diagnosticMode* is a bitwise parameter, with the bits defined as follows:

- Bit 0: Show basic diagnostics. Currently, this displays the URL and the local path in the history.
- Bit 1: Show errors. This displays additional information when errors occur.
- Bit 2: Show status. This displays commands sent to the server and the server's response.

The default value for this parameter is 3 (show basic and error diagnostics). If you are having difficulties, you can try using 7 to show the commands sent to the server and the server's response.

See **FTP Troubleshooting** on page IV-247 for other troubleshooting tips.

/W=*passwordStr* — Specifies the password to be used when logging in to the FTP server. Use this flag if you have an account on the FTP server.

If this flag is omitted, "nopassword" will be used for the login password. This will work with most anonymous FTP servers. Some anonymous FTP servers request that you use your email address as a password. You can do this by including the /W="<your email address>" flag.

If /W is omitted, the login is done using a default password that will work with most anonymous FTP servers.

Note: See **Safe Handling of Passwords** on page IV-241 for information on handling sensitive passwords.

/Z — Errors are not fatal. Will not abort procedure execution if an error occurs.

Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded, -1 if the user canceled in an interactive dialog, or another nonzero value if an error occurred.

Examples

Download a file using a full local path:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String localPath = "hd:Test Folder:TestFile1.txt"
FTPDownload url, localPath
```

Download a file using a local symbolic path and file name:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String pathName = "Igor"      // Igor is the name of a symbolic path.
String fileName = "TestFile1.txt"
FTPDownload/P=$pathName url, fileName
```

Download a directory using a full local path:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestDir1"
String localPath = "hd:Test Folder:TestDir1"
FTPDownload/D url, localPath
```

See Also

File Transfer Protocol (FTP) on page IV-244.

FTPCreateDirectory, **FTPDelete**, **FTPUpload**, **URLEncode**, **FetchURL**.

FTPUpload

FTPUpload [***flags***] ***urlStr, localPathStr***

The FTPUpload operation uploads a file or a directory to an FTP server on the Internet.

Warning: When you upload a file or directory to an FTP server, all previous contents of the server file or directory are obliterated.

For background information on Igor's FTP capabilities and other important details, see **File Transfer Protocol (FTP)** on page IV-244.

FTPUpload sets a variable named V_flag to zero if the operation succeeds and to nonzero if it fails. This, in conjunction with the /Z flag, can be used to allow procedures to continue to execute if a FTP error occurs.

If the operation succeeds, FTPUpload sets a string named S_Filename to the full file path of the uploaded file or, if the /D flag was used, to the full path to the base directory that was uploaded. This is useful in conjunction with the /I flag.

If the operation fails, S_Filename is set to "".

Parameters

urlStr specifies the file or directory to create. It consists of a naming scheme (always `"ftp://"`), a computer name (e.g., `"ftp.wavemetrics.com"` or `"38.170.234.2"`), and a path (e.g., `"/Test/TestFile1.txt"`). For example: `"ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"`.

urlStr must always end with a file name if you are uploading a file or with a directory name if you are uploading a directory, in which case *urlStr* must not end with a slash.

To indicate that *urlStr* contains an absolute path, insert an extra '/' character between the computer name and the path. For example:

```
ftp://ftp.wavemetrics.com//pub/test
```

If you do not specify that the path in *urlStr* is an absolute path, it is interpreted as a path relative to the FTP user's base directory. Since pub is the base directory for an anonymous user, this URL references the same directory:

```
ftp://ftp.wavemetrics.com/test
```

Special characters such as punctuation that are used in *urlStr* may be incorrectly interpreted by the operation. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. See **Percent Encoding** on page IV-239 for additional information.

localPathStr and *pathName* specify the name and location on your hard disk of the local file to be uploaded. If you use a full or partial path for *localPathStr*, see **Path Separators** on page III-398 for details on forming the path.

localPathStr must always end with a file name if you are uploading a file or with a directory name if you are uploading a directory. In the case of a directory, *localPathStr* must not end with a colon or backslash.

FTPUpload displays a dialog that you can use to identify the file or directory to be uploaded in the following cases:

1. You used the /I (interactive) flag.
2. You did not completely specify the location of the file or folder to be uploaded via *pathName* and *localPathStr*.
3. There is an error in *localPathStr*. This can be either a syntactical error or a reference to a nonexistent directory.

See **Examples** for examples of constructing a URL and local path.

Flags

/D — Uploads a complete directory. Omit it if you are uploading a file.

/I — Interactive mode which displays a dialog for choosing the local file or directory to be uploaded.

/M=*messageStr* — Specifies the prompt message used by the dialog in which you choose the local file or directory to be uploaded.

/N=*portNumber* — Specifies the server's TCP/IP port number to use (default is 21). In almost all cases, this will be correct so you won't need to use the /N flag.

/O[=*mode*] — Overwrite. FTPUpload *always* overwrites the specified server file or directory, whether /O is used or not.

If /O=2 is *not* used, all files and subdirectories in the destination directory on the server are first deleted and then the local files and directories are uploaded to the server.

If /O=2 *is* used, the existing contents the contents of the local source directory are merged into the remote directory instead of completely overwriting it.

/P=*pathName* — Contributes to the specification of the file or directory to be uploaded. *pathName* is the name of an existing symbolic path. See **Examples**.

/S=*showProgress* — Displays a progress dialog.

0: No progress dialog.

1: Shows a progress dialog (default).

/T=*transferType* — Transfers in image binary mode.

0: Image (binary) transfer (default).

1: ASCII transfer.

See **FTP Transfer Types** on page IV-247 for more discussion.

/U=*userNameStr* — Specifies the user name to be used when logging in to the FTP server. If this flag is omitted or if *userNameStr* is `""`, you will be logged in as an anonymous user. Use this flag if you have an account on the FTP server.

/V=*diagnosticMode* — Determines what kind of diagnostic messages FTPUpload will display in the history area. *diagnosticMode* is a bitwise parameter, with the bits defined as follows:

Bit 0: Show basic diagnostics. Currently, this displays the URL and the local path in the history.

Bit 1: Show errors. This displays additional information when errors occur.

Bit 2: Show status. This displays status information returned by the operating system to FTPUpload.

The default value for this parameter is 3 (show basic and error diagnostics). If you are having difficulties, you can try using 7 to show the commands sent to the server and the server's response.

See **FTP Troubleshooting** on page IV-247 for other troubleshooting tips.

/W=*passwordStr* Specifies the password used when logging in to the FTP server. Use this flag if you have an account on the FTP server.

If this flag is omitted, "nopassword" will be used for the login password. This will work with most anonymous FTP servers. Some anonymous FTP servers request that you use your email address as a password. You can do this by including the /W="<your email address>" flag.

If /W is omitted, the login is done using a default password that will work with most anonymous FTP servers.

Note: See **Safe Handling of Passwords** on page IV-241 for information on handling sensitive passwords.

/Z Errors are not fatal. Will not abort procedure execution if an error occurs.

Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded, -1 if the user canceled in an interactive dialog, or another nonzero value if an error occurred.

Examples

Upload a file using a full local path:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String localPath = "hd:Test Folder:TestFile1.txt"
FTPUpload url, localPath
```

Upload a file using a local symbolic path and file name:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String pathName = "Igor"      // Igor is the name of a symbolic path.
String fileName = "TestFile1.txt"
FTPUpload/P=$pathName url, fileName
```

Upload a directory using a full local path:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestDir1"
String localPath = "hd:Test Folder:TestDir1"
FTPUpload/D url, localPath
```

See Also

File Transfer Protocol (FTP) on page IV-244.

FTPCreateDirectory, **FTPDelete**, **FTPDownload**, **URLEncode**.

FuncFit

FuncFit [***flags***] ***fitFuncName, cwaveName, waveName*** [***flag parameters***]

FuncFit [***flags***] {***fitFuncSpec***}, ***waveName*** [***flag parameters***]

The FuncFit operation performs a curve fit to a user defined function, or to a sum of fit functions using the second form (see **Fitting Sums of Fit Functions** on page V-195). Fitting can be done using any method that can be selected using the /ODR flag (see **CurveFit** for details).

FuncFit operation parameters are grouped in the following categories: flags, parameters (*fitFuncName*, *cwaveName*, *waveName* or {*fitFuncSpec* }, *waveName*), and flag parameters. The sections below correspond to these categories. Note that flags must precede the *fitFuncName* or `fitFuncSpec` and flag parameters must follow *waveName*.

Flags

See **CurveFit** for all available flags.

Parameters

fitFuncName The user-defined function to fit to, which can be a function taking multiple independent variables (see also **FuncFitMD**). Multivariate fitting with FuncFit *requires* /X=*xwaveSpec*.

cwaveName Wave containing the fitting coefficients.

waveName The wave containing the dependent variable data to be fit to the specified function. For functions of just one independent variable, the dependent variable data is often referred to as "Y data". You can fit to a subrange of the wave by supplying (*startX*,*endX*) or [*startP*,*endP*] after the wave name. See **Wave Subrange Details** below for more information on subranges of waves in curve fitting.

fitFuncSpec List of fit functions and coefficient waves, with some optional information. Using this format fits a model consisting of the sum of the listed fit functions. Intended for fitting multiple peaks, but probably useful for other applications as well. See **Fitting Sums of Fit Functions** on page V-195.

Flag Parameters

These flag parameters must follow *waveName*.

/*E=ewaveName* A wave containing the epsilon values for each parameter. Must be the same length as the coefficient wave.

/STRC=*structureInstance*
Used only with **Structure Fit Functions** on page III-226. When using a structure fit function, you must specify an instance of the structure to FuncFit. This will be an instance that has been initialized by a user-defined function that you write in order to invoke FuncFit.

/*X=xwaveSpec* An optional wave containing X values for each of the input data values. If the fitting function has more than one independent variable, *xwaveSpec* is required and must be either a 2D wave with a column for each independent variable, or a list of waves, one for each independent variable. A list must be in braces: /X={*xwave0*, *xwave1*,…}. There must be exactly one column or wave for each independent variable in the fitting function.

/NWOK Allowed in user-defined functions only. When present, certain waves may be set to null wave references. Passing a null wave reference to FuncFit is normally treated as an error. By using /NWOK, you are telling FuncFit that a null wave reference is not an error but rather signifies that the corresponding flag should be ignored. This makes it easier to write function code that calls FuncFit with optional waves.

The waves affected are the X wave or waves (/X), weight wave (/W), epsilon wave (/E) and mask wave (/M). The destination wave (/D=wave) and residual wave (/R=wave) are also affected, but the situation is more complicated because of the dual use of /D and /R to mean "do autodestination" and "do autoresidual". See /AR and /AD.

If you don't need the choice, it is better not to include this flag, as it disables useful error messages when a mistake or run-time situation causes a wave to be missing unexpectedly.

Note: To work properly this flag must be the last one in the command.

Other parameters are used as for the **CurveFit** operation, with some exceptions for multivariate fits.

Details for Multivariate Fits

The dependent variable data wave, *waveName*, must be a 1D wave even for multivariate fits. For fits to data in a multidimensional wave, see **FuncFitMD**.

For multivariate fits, the auto-residual (/R with no wave specified) is calculated and appended to the top graph if the dependent variable data wave is graphed in the top graph as a simple 1D trace. Auto residuals are calculated but not displayed if the data are displayed as a contour plot.

The autodest wave (/D with no wave specified) for multivariate fits has the same number of points as the data wave, with a model value calculated at the X values contained in the wave or waves specified with /*X=xwaveSpec*.

Confidence bands are not supported for multivariate fits.

Wave Subrange Details

Almost any wave you specify to FuncFit can be a subrange of a wave. The syntax for wave subranges is the same as for the Display command (see **Subrange Display Syntax** on page II-288 for details). See **Wave Subrange Details** on page V-91 for a discussion of the use of subranges in curve fitting.

The backwards compatibility rules for CurveFit apply to FuncFit as well.

In addition to the waves discussed in the CurveFit documentation, it is possible to use subranges when specifying the coefficient wave and the epsilon wave. Since the coefficient wave and epsilon wave must have the same number of points, it might make sense to make them two columns from a single multicolumn wave. For instance, here is an example in which the first column is used as the coefficient wave, the second is used as the epsilon wave, and the third is used to save a copy of the initial guesses for future reference:

```
Make/D/N=(5, 3) myCoefs
myCoefs[][0] = {1,2,3,4,5}              // hypothetical initial guess
```

```
myCoefs[][1] = 1e-6                        // reasonable epsilon values
myCoefs[][2] = myCoefs[p][0]               // save copy of initial guess
FuncFit myFitFunc, myCoefs[][0] myData /E=myCoefs[][1] …
```

You might have a fit function that uses a subset of the coefficients that are used by another. It might be useful to use a single wave for both. Here is an example in which a function that takes four coefficients is used to fit a subset of the coefficients, and then that solution is used as the initial guess for a function that takes six coefficients:

```
Make/D/N=6 Coefs6={1,2,3,4,5,6}
FuncFit Fit4Coefs, Coefs6[0,3] fitfunc4Coefs …
FuncFit Fit6Coefs, Coefs6 ...
```

Naturally, the two fit functions must be worked out carefully to allow this.

Fitting Sums of Fit Functions

If Igor encounters a left brace at the beginning of the fit function name, it expects a list of fit functions to be summed during the fit. This is useful for, for instance, fitting several peaks in a data set to a sum of peak functions.

The fit function specification includes at least the name of the fitting function and an associated coefficient wave. A sum of fit functions requires multiple coefficient waves, one for each fit function. Any coefficient wave-related options must be specified in the fit function specification via keyword-value pairs.

The syntax of the sum-of-fit-functions specification is as follows:

{{*func1, coef1, keyword=value*},{*func2, coef2, keyword=value*}, …}

or

{*string=fitSpecStr*}

Within outer braces, each fit function specification is enclosed within inner braces. You can use one or more fit function specifications, with no intrinsic limit on the number of fit functions.

The second format is available to overcome limitations on the length of a command line in Igor. This format is just like the first, but everything inside the outer braces is contained in a string expression (which may be just a single string variable).

You can use any fit function that can be used for ordinary fitting, including the built-in functions that are available using the CurveFit operation. If you should write a user-defined fitting function with the same name as a built-in fit function, the user-defined function will be used (this is strongly discouraged).

Every function specification must include an appropriate coefficient wave, pre-loaded with initial guesses.

The comma between each function specification is optional.

The keyword-value pairs are optional, and are used to communicate further options on a function-by-function basis. Available keywords are:

HOLD=*holdstr*
: Indicates that a fit coefficient should be held fixed during fitting. *holdstr* works just like the hold string specified via the /H flag for normal fitting, but applies only to the coefficient wave associated with the fit function it appears with.

  If you include HOLD in a string expression (the {`string=fitSpecStr`} syntax), you must escape the quotation marks around the hold string.

  If you use the command-line syntax {{`func1,coef1,HOLD=`*`holdStr`*}, `...`}, *holdStr* may be a reference to a global variable acquired using SVAR, or it may be a quoted literal string.

  If you use {`string=`*`fitSpecStr`*}, *fitSpecStr* is parsed at run-time outside the context of any running function. Consequently, you cannot use a general string expression. You can use either `HOLD="quotedLiteralString"` or `HOLD=root:globalString`.

CONST={*constants*}
: Sets the values of constants in the fitting function. So far, only two built-in functions take constants: exp_XOffset and dblexp_XOffset. They each take just one constant (the X offset), so you will have a "list" of one number inside the braces.

EPSW=*epsilonWave*
: Specifies a wave holding epsilon values. Use only with a user-defined fitting function to set the differencing interval used to calculate numerical estimates of derivatives of the fitting function.

STRC=*structureInstance*
: Specifies an instance of the structure to FuncFit when using a structure fit function. *structureInstance* is an instance that was initialized by a user-defined function that invokes FuncFit. This keyword (and structure fitting functions) can be used only

when calling FuncFit from within a user-defined function. See **Structure Fit Functions** on page III-226 for more details.

For more details, and for examples of sums of fit functions in use, **Fitting Sums of Fit Functions** on page III-213.

See Also

The **CurveFit** operation for parameter details. See also **FuncFitMD** for user-defined multivariate fits to data in a multidimensional wave.

The best way to create a user-defined fitting function is using the Curve Fitting dialog. See **Using the Curve Fitting Dialog** on page III-159, especially the section **Fitting to a User-Defined Function** on page III-171.

For details on the form of a user-defined function, see **User-Defined Fitting Function: Detailed Description** on page III-217.

FuncFitMD

FuncFitMD [***flags***] ***fitFuncName, cwaveName, waveName*** [***flag parameters***]

The FuncFitMD operation performs a curve fit to the specified multivariate user defined *fitFuncSpec*. FuncFitMD handles gridded data sets in multidimensional waves. Most parameters and flags are the same as for the **CurveFit** and **FuncFit** operations; differences are noted below.

cwaveName is a 1D wave containing the fitting coefficients, and *functionName* is the user-defined fitting function, which has 2 to 4 independent variables.

FuncFitMD operation parameters are grouped in the following categories: flags, parameters (*fitFuncName*, *cwaveName*, *waveName*), and flag parameters. The sections below correspond to these categories. Note that flags must precede the *fitFuncName* and flag parameters must follow *waveName*.

Flags

/L=*dimSize* Sets the dimension size of the wave created by the auto-trace feature, that is, /D without destination wave. The wave fit_*waveName* will be a multidimensional wave of the same dimensionality as *waveName* that has *dimSize* elements in each dimension. That is, if you are fitting to a matrix wave, fit_*waveName* will be a square matrix that has dimensions *dimSize* X*dimSize*. **Beware**: *dimSize* =100 requires 100 million points for a 4-dimensional wave!

Parameters

fitFuncName User-defined function to fit to, which must be a function taking 2 to 4 independent variables.

cwaveName 1D wave containing the fitting coefficients.

waveName The wave containing the dependent variable data to be fit to the specified function. For functions of just one independent variable, the dependent variable data is often referred to as "Y data". You can fit to a subrange of the wave by supplying (*startX*,*endX*) or [*startP*,*endP*] for each dimension after the wave name. See **Wave Subrange Details** below for more information on subranges of waves in curve fitting.

Flag Parameters

These flag parameters must follow *waveName*.

/E=*ewaveName* A wave containing the epsilon values for each parameter. Must be the same length as the coefficient wave.

/T=*twaveName* Like /X except for the T independent variable. This is a 1D wave having as many elements as *waveName* has chunks.

/X=*xwaveName* The X independent variable values for the data to fit come from *xwaveName* instead of from the X scaling of *waveName*. This is a 1D wave having as many elements as *waveName* has rows.

/Y=*ywaveName* Like /X except for the Y independent variable. This is a 1D wave having as many elements as *waveName* has columns.

/Z=*ywaveName* Like /X except for the Z independent variable. This is a 1D wave having as many elements as *waveName* has layers.

/NWOK Allowed in user-defined functions only. When present, certain waves may be set to null wave references. Passing a null wave reference to FuncFitMD is normally treated as an error. By using /NWOK, you are telling FuncFitMD that a null wave reference is

not an error but rather signifies that the corresponding flag should be ignored. This makes it easier to write function code that calls FuncFitMD with optional waves.

The waves affected are the X wave or waves (/X), the Y spacing wave (/Y), the Z spacking wave (/Z) the T spacing wave (/T), weight wave (/W), epsilon wave (/E) and mask wave (/M). The destination wave (/D=wave) and residual wave (/R=wave) are also affected, but the situation is more complicated because of the dual use of /D and /R to mean "do autodestination" and "do autoresidual". See /AR and /AD.

If you don't need the choice, it is better not to include this flag, as it disables useful error messages when a mistake or run-time situation causes a wave to be missing unexpectedly.

Note: To work properly this flag must be the last one in the command.

Details

Auto-residual (/R with no wave specified) and auto-trace (/D with no wave specified) for functions having two independent variables are plotted in a separate graph window if *waveName* is plotted as a contour or image in the top graph. An attempt is made to plot the model values and residuals in the same way as the input data.

By default the auto-trace and auto-residual waves are 50x50 or 25x25x25 or 15x15x15x15. Use /L=*dimSize* for other sizes. Make your own wave and use /D=*waveName* or /R=*waveName* if you want a wave that isn't square. In this case, the wave dimensions must be the same as the dependent data wave.

Confidence bands are not available for multivariate fits.

Wave Subrange Details

Almost any wave you specify to FuncFitMD can be a subrange of a wave. The syntax for wave subranges is the same as for the Display command; see **Subrange Display Syntax** on page II-288 for details. Note that the dependent variable data (*waveName*) must be a multidimensional wave; this requires an extension of the subrange syntax to allow a multidimensional subrange. See **Wave Subrange Details** on page V-194 for a discussion of the use of subranges in curve fitting.

The backwards compatibility rules for **CurveFit** apply to FuncFitMD as well.

A subrange could be used to pick a plane from a 3D wave for fitting using a fit function taking two independent variables:

```
Make/N=(100,100,3) DepData
FuncFitMD fitfunc2D, myCoefs, DepData[][][0] …
```

See Also

The **CurveFit** operation for parameter details.

The best way to create a user-defined fitting function is using the Curve Fitting dialog. See **Using the Curve Fitting Dialog** on page III-159, especially the section **Fitting to a User-Defined Function** on page III-171.

For details on the form of a user-defined function, see **User-Defined Fitting Function: Detailed Description** on page III-217.

FUNCREF

FUNCREF *protoFunc func* [**= *funcSpec***]

Within a user function, FUNCREF is a reference that creates a local reference to a function or a variable containing a function reference.

When passing a function as an input parameter to a user function, the syntax is:

`FUNCREF` *protoFunc func*

In this FUNCREF reference, *protoFunc* is a function that specifies the format of the function that can be passed by the FUNCREF, and *func* is a function reference used as an input parameter.

When you declare a function reference variable within a user function, the syntax is:

`FUNCREF` *protoFunc* `func =` *funcSpec*

Here, the local FUNCREF variable, `func`, is assigned a *funcSpec*, which can be a literal function name, a $ string expression that evaluates at runtime, or another FUNCREF variable.

See Also

Function References on page IV-84 for an example and further usage details.

FuncRefInfo

FuncRefInfo(*funcRef*)

The FuncRefInfo function returns information about a **FUNCREF**.

Parameters

funcRef is a function reference variable declared by a FUNCREF statement in a user-defined function.

Details

FuncRefInfo returns a semicolon-separated keyword/value string containing the following information:

Keyword	Information
NAME	The name of the reference function or "" if the FUNCREF variable has not been assigned to point to a function.
ISPROTO	0 if the FUNCREF variable has been assigned to point to a function.
	1 if it has not been assigned and therefore still points to the prototype function.
ISXFUNC	0 if it points to a user-defined function.
	1 if the FUNCREF points to an external function.

See Also

Function References on page IV-84 and **FUNCREF** on page V-197.

Function

Function [[/C /D /S /DF /WAVE] *functionName*([*parameters*])

The Function keyword introduces a user-defined function in a procedure window.

The optional flags specify the return value type, if any, for the function.

Flags

/C	Returns a complex number.
/D	Returns a double-precision number. Obsolete, accepted for backward compatibility.
/S	Returns a string.
/DF	Returns a data folder reference. See **Data Folder Reference Function Results** on page IV-64.
/WAVE	Returns a wave reference. See **Wave Reference Function Results** on page IV-60.

Details

If you omit all flags, the result is a scalar double-precision number.

The /D flag is not needed because all numeric return values are double-precision.

The /DF and /WAVE flags require Igor Pro 6.1 or later.

See Also

Chapter IV-3, **User-Defined Functions** and **Function Syntax** on page IV-29 for further information.

FunctionInfo

FunctionInfo(*functionNameStr* [, *procedureWinTitleStr*])

The FunctionInfo function returns a keyword-value pair list of information about the user-defined or external function name in *functionNameStr*.

Parameters

functionNameStr a string expression containing the name or multipart name of a user-defined or external function. *functionNameStr* is usually just the name of a function.

To return information about a static function, supply both the module name and the function name in MyModule#MyFunction format (see **Regular Modules** on page IV-212), or specify the function name and *procedureWinTitleStr* (see below).

To return information about a function in a different independent module, supply the independent module name in addition to any module name and function name (a double or triple name):

Name	What It Refers To
MyIndependentModule#MyFunction	Refers to a non-static function in an independent module.
MyIndependentModule#MyModule#MyFunction	Refers to a static function in a procedure file with `#pragma moduleName=MyModule` in an independent module.

(See **Independent Modules** on page IV-214 for details on independent modules.)

The optional *procedureWinTitleStr* can be the title of a procedure window (such as "Procedure" or "File Name Utilities.ipf") in which to search for the named user-defined function. The information about the named function in the specified procedure window is returned.

The *procedureWinTitleStr* parameter makes it possible to select one of several static functions with identical names among different procedure windows, even if they do not contain a `#pragma moduleName=myModule` statement.

If you execute this command:

```
SetIgorOption IndependentModuleDev=1
```

then *procedureWinTitleStr* can also be a title followed by an independent module name in brackets to return information about the named function in the procedure window of the given title that belongs to named independent module.

procedureWinTitleStr can also be just an independent module name in brackets to return information about the named nonstatic function in any procedure window that belongs to named independent module.

Details

The returned string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon. The keywords are as follows:

User-Defined and External Functions

Keyword	Information Following Keyword
`NAME`	The name of the function. Same as contents of *functionNameStr* in most cases. Just the function name if you use the `module#function` format.
`TYPE`	Value is "UserDefined" or "XFunc".
`RETURNTYPE`	Number giving the return type of the function. See the table **Return Type and Parameter Type Codes** on page V-200.
`N_PARAMS`	Number of parameters for this function.
`PARAM_`*n*`_TYPE`	Number encoding the type of each parameter. There will be N of these keywords, one for each parameter. The part shown as *n* will be a number from 0 to N.

See **Examples** for a method for decoding these keywords.

User-Defined Functions Only

Keyword	Information Following Keyword	
PROCWIN	Title of procedure window containing the function definition.	
MODULE	Module containing function definition (see **Regular Modules** on page IV-212).	
INDEPENDENTMODULE	Independent module containing function definition (see See **Independent Modules** on page IV-214).	
SPECIAL	The value part has one of three values:	
	no:	Not a "special" function.
	static:	Is a static function. Use the module#function format to get info about static functions.

User-Defined Functions Only

Keyword	Information Following Keyword
	override: Function is an override function. See **Function Overrides** on page IV-84.
SUBTYPE	The function subtype, for instance **FitFunc**. See **Procedure Subtypes** on page V-12 for others.
PROCLINE	Line number within the procedure window of the function definition.
VISIBLE	Either "Yes" or "No". Set to No in the unlikely event that the function is defined in an invisible file.
N_OPT_PARAMS	Number of optional parameters. Usually zero.

External Functions Only

Keyword	Information Following Keyword
XOP	Name of the XOP module containing the function.

Return Type and Parameter Type Codes

Type	Code	Code in Hex
Complex	1	0x1
Variable	4	0x4
Single Precision	2	0x2
Double Precision	4	0x4
Byte	8	0x8
16-bit Integer	16	0x10
32-bit Integer	32	0x20
Unsigned	64	0x40
Data folder reference	256	0x100
Structure	512	0x200
Function reference	1024	0x400
Pass by reference parameter	4096	0x1000
String	8192	0x2000
Wave	16384	0x4000

Functions can return either a numeric value or a string. A numeric value is always double precision and my be complex. Thus, the return type will always be one of the values 4 (numeric), 5 (4+1 for complex numeric) or 8192 (string).

These codes may be combined to indicate combinations of attributes. For instance, the code for a variable is 4 and the code for complex is 1. Consequently, the code for a complex variable is 5. The code for a complex variable passed by reference is (4+1+4096) = 4101.

Variables are always double-precision, hence the code of 4.

Waves may have a variety of codes. Numeric waves will combine with one of the number type codes such as 2 or 16. This does not reflect the numeric type of any actual wave, but rather any flag you may have used in the Wave reference. Thus, if the beginning of your function looks like

```
Function myFunc(w)
    Wave w
```

the code for the parameter w will be 16386 (16384 + 2) indicating a single-precision wave. You can use a numeric type flag with the Wave reference:

```
Function myFunc(w)
    Wave/I w
```

In this case, the code will be 16416 (16384 + 32).

Such codes are not very useful, as it is very rare to use a numeric type flag because the numeric type will be resolved correctly at runtime regardless of the flag.

A text wave has no numeric type, so its code is exactly 16384. Thus, the numeric type part of the code for a numeric wave serves to distinguish a numeric wave from a text wave.

Examples

This function formats function information nicely and prints it in the history window in an organized fashion. You can copy it into the Procedure window to try it out. It uses the function `interpretParamType()` below to print a human-readable version of the parameter and return types. To try `PrintFuncInfo()`, you will need to copy the code for `interpretParamType()` as well.

```
Function PrintFuncInfo(FunctionName)
    String FunctionName

    String infostr = FunctionInfo(FunctionName)
    if (strlen(infostr) == 0)
        print "The function "+FunctionName+" does not exist."
        return -1
    endif

    print "Name: ", StringByKey("NAME", infostr)

    String typeStr = StringByKey("TYPE", infostr)
    print "Function type: ", typeStr
    Variable IsUserDefined = CmpStr(typeStr, "UserDefined")==0

    // It's not really necessary to use an IF statement here;
    // it simply prevents lines with blank information being printed for an XFUNC.

    if (IsUserDefined)
        print "Module: ", StringByKey("MODULE", infostr)
        print "Procedure window: ", StringByKey("PROCWIN", infostr)
        print "Subtype: ", StringByKey("SUBTYPE", infostr)
        print "Special? ", StringByKey("SPECIAL", infostr)

        // Note use of NumberByKey to get a numeric key value
        print "Line number: ", NumberByKey("PROCLINE", infostr)
    endif

    // See function interpretParamType() below for example of
    // interpreting type information

    Variable dummy = NumberByKey("RETURNTYPE", infostr)
    String dummyStr = interpretParamType(dummy)
    print "Return type: ", dummy, dummyStr

    Variable nparams = NumberByKey("N_PARAMS", infostr)
    print "Number of Parameters: ", nparams

    Variable nOptParams = 0
    if (IsUserDefined)
        nOptParams = NumberByKey("N_OPT_PARAMS", infostr)
        print "Optional Parameters: ", nOptParams
    endif

    Variable i
    for (i = 0; i < nparams; i += 1)

        // Note how the PARAM_n_TYPE keyword string is constructed here
        String paramKeyStr = "PARAM_"+num2istr(i)+"_TYPE"
        Variable ptype = NumberByKey(paramKeyStr, infostr)

        String output = "Parameter "+num2istr(i)+"; "
        output += "type as number: "+num2istr(ptype)+"; "
        output += "type as string: "+interpretParamType(ptype)
        print output
    endfor
End
```

Function that creates a human-readable string with information about parameter and return types. Note that various attributes of the type info is tested using the bitwise AND operator (&) to test for individual bits. The

constants are expressed as hexadecimal values (prefixed with "0x") to make them more readable (at least to a programmer). Otherwise, 0x4000 would be 16384; at least, 0x4000 is clearly a single-bit constant.

```
Function/S interpretParamType(ptype)
    Variable ptype

    String typeStr = ""

    // flag to tell whether to print out "complex" and "real"
    // We don't want that information on strings and text waves

    Variable wantRC = 1

    if (ptype & 0x4000)              // test for WAVE bit set
        typeStr += "Wave "

        // If *only* wave bit set, its a text wave. A numeric
        // wave has some other information causing the value
        // to be different from 0x4000

        if (ptype == 0x4000)
            typeStr += "text "
            // don't want to print "real" and "complex" for text waves
            wantRC = 0
        endif

    elseif (ptype & 0x2000)          // test for STRING bit set
        typeStr += "String "
        // don't want to print "real" and "complex" for strings
        wantRC = 0
    elseif (ptype & 4)               // test for VARIABLE bit
        typeStr += "Variable "
    elseif (ptype & 0x200)           // test for STRUCTURE bit
        typeStr += "Struct "
        wantRC = 0
    elseif (ptype & 0x400)           // test for FUNCREF bit
        typeStr += "FuncRef "
        wantRC = 0
    endif

    // print "real" and "complex" for numeric objects only
    if (wantRC)
        if (ptype & 1)               // test for COMPLEX bit
            typeStr += "cmplx "
        else
            typeStr += "real "
        endif
    endif

    if (ptype & 0x1000)              // test for PASS BY REFERENCE bit
        typeStr += "reference "
    endif

    return typeStr
End
```

See Also

The **StringByKey** and **NumberByKey** functions.

StringByKey, **NumberByKey**, and **FunctionList** functions.

Regular Modules on page IV-212 and **Independent Modules** on page IV-214.

FunctionList

FunctionList(*matchStr*, *separatorStr*, *optionsStr*)

The FunctionList function returns a string containing a list of built-in or user-defined function names satisfying certain criteria. This is useful for making a string to list functions in a pop-up menu control. Note that if the procedures need to be compiled, then FunctionList will not list user-defined functions.

Parameters

Only functions having names that match *matchStr* string are listed. Use `"*"` to match all names. See **WaveList** for examples.

The first character of *separatorStr* is appended to each function name as the output string is generated. *separatorStr* is usually `";"` for list processing (See **Processing Lists of Waves** on page IV-174 for details on list processing).

Use *optionsStr* to further qualify the list of functions. *optionsStr* is a string containing keyword-value pairs separated by commas. Available options are:

KIND:*nk*

nk=1: List built-in functions.

nk=2: List normal and override user-defined functions.

nk=4: List external functions (defined by an XOP).

nk=8: List only curve fitting functions; must be summed with 1, 2, 4, or 16. For example, use 10 to list user-defined fitting functions.

nk=16: Include static user-defined functions (requires WIN: option).

SUBTYPE:*typeName*

Lists functions that have the type *typeName*. That is, you could use ButtonControl as *typeName* to list only functions that are action procedures for buttons.

VALTYPE:*nv*

Restricts list to functions whose return type is a certain kind.

nv=1: Real-valued functions.

nv=2: Complex-valued functions.

nv=4: String functions.

nv=8: WAVE functions

nv=16: DFREF functions

Use a sum of these values to include more than one type. The return type is not restricted if this option is omitted.

NPARAMS:*np*

Restricts the list to functions having exactly *np* parameters. Omitting this option lists functions having any number of parameters.

NINDVARS:*ni*

Restricts the list to fitting functions for exactly *ni* independent variables. NINDVARS is ignored if you have not elected to list curve fitting functions using the KIND option. Functions for any number of independent variables are listed if the NINDVARS option is omitted.

WIN:*windowTitle*

Lists functions that are defined in the procedure window with the given title. "Procedure" is the title of the built-in procedure window.

Note: Because the *optionsStr* keyword-value pairs are comma separated and procedure window names can have commas in them, the WIN:keyword must be the last one specified.

WIN:windowTitle [*independentModuleName*]

Lists functions that are defined in the named procedure window that belongs to the independent module *independentModuleName*. See **Independent Modules** on page IV-214 for details. Requires `SetIgorOption IndependentModuleDev=1`, otherwise no functions are listed.

Requires *independentModuleName*=ProcGlobal or SetIgorOption independentModuleDev=1, otherwise no functions are listed.

Note: The syntax is literal and strict: the window title must be followed by one space and a left bracket, followed directly by the independent module name and a closing right bracket.

WIN:[*independentModuleName*]

Lists functions that are defined in any procedure file that belongs to the named independent module.

Requires *independentModuleName*=ProcGlobal or SetIgorOption independentModuleDev=1, otherwise no functions are listed.

Note: The syntax is literal and strict: 'WIN:' must be followed by a left bracket, followed directly by the independent module name and a closing right bracket, like this:

```
FunctionList(...,"WIN:[myIndependentModuleName]")
```

Examples

To list user-defined fitting functions for two independent variables:

```
Print FunctionList("*",";","KIND:10,NINDVARS:2")
```

To list button-control functions that start with the letter *b* (note that button-control functions are user-defined):

```
Print FunctionList("b*",";","KIND:2,SUBTYPE:ButtonControl")
```

See Also

Independent Modules on page IV-214.

For details on procedure subtypes, see **Procedure Subtypes** on page IV-179, as well as **Button**, **CheckBox**, **SetVariable**, and **PopupMenu**.

The **DisplayProcedure** operation and the **MacroList**, **OperationList**, **StringFromList**, and **WinList** functions.

FunctionPath

FunctionPath(*functionNameStr*)

The FunctionPath function returns a path to the file containing the named function. This is useful in certain specialized cases, such as if a function needs access to a lookup table of a large number of values.

The most likely use for this is to find the path to the file containing the currently running function. This is done by passing `""` for *functionNameStr*, as illustrated in the example below.

The returned path uses Macintosh syntax regardless of the current platform. See **Path Separators** on page III-398 for details.

If the procedure file is a normal standalone procedure file, the returned path will be a full path to the file such as `"hd:Igor Pro Folder:WaveMetrics Procedures:Waves:Wave Lists.ipf"`.

If the function resides in the built-in procedure window the returned path will be `":Procedure"`. If the function resides in a packed procedure file, the returned path will be `":<packed procedure window title>"`.

If FunctionPath is called when procedures are in an uncompiled state, it returns ":".

Parameters

If *functionNameStr* is `""`, FunctionPath returns the path to the currently executing function or `""` if no function is executing.

Otherwise FunctionPath returns the path to the named function or `""` if no function by that name exists.

Examples

This example loads a lookup table into memory. The lookup table is stored as a wave in an Igor binary file.

```
Function LoadMyLookupTable()
   String path

   path = FunctionPath("")    // Path to file containing this function.
   if (CmpStr(path[0],":") == 0)
      // This is the built-in procedure window or a packed procedure
      // file, not a standalone file. Or procedures are not compiled.
      return -1
   endif

   // Create path to the lookup table file.
   path = ParseFilePath(1, path, ":", 0, 0) + "MyTable.ibw"

   String dfSave = GetDataFolder(1)

   // A previously-created place to store my private data.
   SetDataFolder root:Packages:MyData

   // Load the lookup table.
   LoadWave/O path

   SetDataFolder dfSave

   return 0
End
```

See Also

The **FunctionList** function.

GalleryGlobal

GalleryGlobal#*pictureName*

The GalleryGlobal keyword is used in an independent module to reference a picture in the global picture gallery which you can view by choosing Misc→Pictures.

See Also

See **Independent Modules and Pictures** on page IV-220.

gamma

gamma(*num*)

The gamma function returns the value of the gamma function of *num*. If *num* is complex, it returns a complex result. Note that the return value for *num* close to negative integers is NaN, not ±Inf.

See Also

The **gammln** function.

gammaInc

gammaInc(*a*, *x* [, *upperTail*])

The gammaInc function returns the value of the incomplete gamma function, defined by the integral

$$\Gamma(a,x) = \int_x^\infty e^{-t}t^{e-1}dt$$

If *upperTail* is zero, the limits of integration are 0 to x. If *upperTail* is absent, it defaults to 1, and the limits of integration are x to infinity, as shown. Note that `gammaInc(a, x) = gamma(a) - gammaInc(a, x, 0)`.

Defined for x > 0, a ≥ 0 (*upperTail* = zero or absent) or a > 0 (*upperTail* = 0).

See Also

The **gamma**, **gammp**, and **gammq** functions.

gammaNoise

gammaNoise(*a* [, *b*])

The gammaNoise function returns a pseudo-random value from the gamma distribution

$$f(x) = \frac{x^{a-1}\exp\left(-\frac{x}{b}\right)}{b^a\Gamma(a)}, \qquad x>0, a>0, b>0$$

whose mean is *ab* and variance is ab^2. For backward compatibility you can omit the parameter *b* in which case its value is set to 1. When $a \rightarrow 1$ gammaNoise reduces to **expnoise**.

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

References

Marsaglia, G., and W. W. Tsang, *ACM*, 26, 363-372, 2000.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-332.

Chapter III-12, **Statistics** for a function and operation overview.

gammln

gammln(*num* [, *accuracy*])

The gammln function returns the natural log of the gamma function of *num*, where *num* > 0. If *num* is complex, it returns a complex result. Optionally, *accuracy* can be used to specify the desired fractional accuracy. If *num* is complex, it returns a complex result. In this case, *accuracy* is ignored.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of $(f_{actual} - f_{returned})/f_{actual}$ be less than 10^{-7}.

For backward compatibility, if you don't include *accuracy*, gammln uses older code that achieves an accuracy of about $2x10^{-10}$.

With *accuracy*, newer code is used that is both faster and more accurate. The output has fractional accuracy better than $1x10^{-15}$ except for values near zero, where the absolute accuracy ($f_{actual} - f_{returned}$) is better than $2x10^{-16}$.

The speed of calculation depends only weakly on accuracy. Higher accuracy is significantly slower than lower accuracy only for *num* between 6 and about 10.

See Also

The **gamma** function.

gammp

gammp(*a, x* [, *accuracy*])

The gammp function returns the regularized incomplete gamma function P(*a,x*), where $a > 0$, $x \geq 0$. Optionally, *accuracy* can be used to specify the desired fractional accuracy. Same as `gammaInc(a, x, 0)/gamma(a)`.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of $(f_{actual} - f_{returned})/f_{actual}$ be less than 10^{-7}.

For backward compatibility, if you don't include *accuracy*, gammp uses older code that is slower for an equivalent accuracy, and cannot achieve as high accuracy.

The ability of gammp to return a value having full fractional accuracy is limited by double-precision calculations. This means that it will mostly have fractional accuracy better than about 10^{-15}, but this is not guaranteed, especially for extreme values of *a* and *x*.

See Also

The **gammaInc** and **gammq** functions.

gammq

gammq(*a, x* [, *accuracy*])

The gammq function returns the regularized incomplete gamma function 1-P(*a,x*), where $a > 0$, $x \geq 0$. Optionally, *accuracy* can be used to specify the desired fractional accuracy. Same as `gammaInc(a, x)/gamma(a)`.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of $(f_{actual} - f_{returned})/f_{actual}$ be less than 10^{-7}.

For backward compatibility, if you don't include *accuracy*, gammq uses older code that is slower for an equivalent accuracy, and cannot achieve as high accuracy.

The ability of gammq to return a value having full fractional accuracy is limited by double-precision calculations. This means that it will mostly have fractional accuracy better than about 10^{-15}, but this is not guaranteed, especially for extreme values of *a* and *x*.

See Also

The **gammaInc** and **gammp** functions.

Gauss

Gauss(x,*xc*,*wx* [,y,*yc*,*wy* [,z,*zc*,*wz* [,t,*tc*,*wt*]]])

The Gauss function returns a normalized Gaussian for the specified dimension.

$$\text{Gauss}(\underline{r}, \underline{c}, \underline{w}) = \prod_{i=1}^{n} \frac{1}{w_i \cdot \sqrt{2\pi}} \exp\left\{-\frac{1}{2}\left(\frac{r_i - c_i}{w_i}\right)^2\right\}$$, where n is the number of dimensions.

Parameters

xc, *yc*, *zc*, and *tc* are the centers of the Gaussian in the X, Y, Z, and T directions, respectively.

wx, *wy*, *wz*, and *wt* are the widths of the Gaussian in the X, Y, Z, and T directions, respectively.

Note that w_i here is the standard deviation of the Gaussian. This is different from the width parameter in the gauss curve fitting function, which is sqrt(2) times the standard deviation.

Note also that the Gauss function lacks the cross-correlation parameter that is included in the Gauss2D curve fitting function.

Examples

```
Make/n=100 eee=gauss(x,50,10)
Print area(eee,-inf,inf)
  0.999999
Make/n=(100,100) ddd=gauss(x,50,10,y,50,15)
Print area(ddd,-inf,inf)
  0.999137
```

Gauss1D

Gauss1D(*w, x*)

The Gauss1D function returns the value of a Gaussian peak defined by the coefficients in the wave *w*. The equation is the same as the Gauss curve fit:

$$w[0] = w[0] + w[1]\exp\left[-\left(\frac{x - w[2]}{w[3]}\right)^2\right].$$

Examples

Do a fit to a Gaussian peak in a portion of a wave, then extend the model trace to the rest of the X range:

```
Make/O/N=100 junkg                        // fake data wave
Setscale/I x -1,1,junkg
Display junkg
junkg = 1+2.5*exp(-((x-.5)/.3)^2)+gnoise(.1)
Duplicate/O junkg, junkgfit
junkgfit = NaN
AppendToGraph junkgfit
CurveFit gauss junkg[50,99] /D=junkgfit
// now extend the model trace
junkgfit = Gauss1D(w_coef, x)
```

See Also

The **CurveFit** operation.

Gauss2D

Gauss2D(*w, x, y*)

The Gauss2D function returns the value of a two-dimensional Gaussian peak defined by the coefficients in the wave *w*. The equation is the same as the Gauss2D curve fit:

$$w[0] + w[1]\exp\left[\frac{-1}{2(1 - w[6]^2)}\left(\left(\frac{x - w[2]}{w[3]}\right)^2 + \left(\frac{y - w[4]}{w[5]}\right)^2 - \frac{2w[6](x - w[2])(y - w[4])}{w[3]w[5]}\right)\right].$$

Examples

Do a fit to a Gaussian peak in a portion of a wave, then extend the model trace to the rest of the X range (watch out for the very long wave assignment to junkg2D):

```
Make/O/N=(100,100) junkg2D                        // fake data wave
Setscale/I x -1,1,junkg2D
Setscale/I y -1,1,junkg2D
Display; AppendImage junkg2D
//Caution! Next command wrapped to fit page:
junkg2D = -1 + 2.5*exp((-1/(2*(1-.4^2)))*(((x-.1)/.2)^2+((y+.2)/.35)^2+2*.4*
          ((x-.1)/.2)*((y+.2)/.35)))
junkg2D += gnoise(.01)
Duplicate/O junkg2D, junkg2Dfit
junkg2Dfit = NaN
AppendMatrixContour junkg2Dfit
CurveFit gauss2D junkg2D[20,80][10,70] /D=junkg2Dfit[20,80][10,70]
// now extend the model trace
junkg2Dfit = Gauss2D(w_coef, x, y)
```

See Also

The **CurveFit** operation.

gcd

gcd(*A, B*)

The gcd function calculates the greatest common divisor of *A* and *B*, which are both assumed to be integers.

Examples

Compute least common multiple (LCM) of two integers:

```
Function LCM(a,b)
   Variable a, b
   return((a*b)/gcd(a,b))
End
```

GetAxis

GetAxis [**/W=*winName* /Q**] ***axisName***

The GetAxis operation determines the axis range and sets the variables V_min and V_max to the minimum and maximum values of the named axis.

Parameters

axisName is usually `"left"`, `"right"`, `"top"` or `"bottom"`, though it may also be the name of a free axis such as `"VertCrossing"`.

Flags

/Q	Prevents values of V_flag, V_min, and V_max from being printed in the history area. The results are still stored in the variables.
/W=*winName*	Retrieves axis info from the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

GetAxis sets V_min according to the bottom of vertical axes or left of horizontal axes and V_max according to the top of vertical axes or right of horizontal axes. It also sets the variable V_flag to 0 if the specified axis is actually used in the graph, or to 1 if it is not.

See Also

The **AxisInfo** function.

GetDataFolder

GetDataFolder(*mode* [, *dfr*])

The GetDataFolder function returns a string containing the name of or full path to the current data folder or, if *dfr* is present, the specified data folder.

For Igor Pro 6.1 or later, **GetDataFolderDFR** is preferred.

Parameters

If *mode*=0, it returns just the name of the data folder.

If *mode*=1, GetDataFolder returns a string containing the full path to the data folder.

dfr, if present, specifies the data folder of interest. This parameter was added in Igor Pro 6.1.

Details

GetDataFolder can be used to save and restore the current data folder in a procedure. However, as of Igor Pro 6.1, GetDataFolderDFR is preferred for that purpose.

Examples

```
String savedDataFolder = GetDataFolder(1)          // Save
SetDataFolder root:
Variable/G gGlobalRootVar
SetDataFolder savedDataFolder                      // and restore
```

See Also

Chapter II-8, **Data Folders**.

The **SetDataFolder** operation and **GetDataFolderDFR** function.

GetDataFolderDFR

GetDataFolderDFR()

The GetDataFolderDFR function returns the data folder reference for the current data folder.

Requires Igor Pro 6.1 or later.

Details

GetDataFolderDFR can be used to save and restore the current data folder in a procedure. It is like GetDataFolder but returns a data folder reference rather than a string.

Example

```
DFREF saveDFR = GetDataFolderDFR()                // Save
SetDataFolder root:
Variable/G gGlobalRootVar
SetDataFolder saveDFR                             // and restore
```

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-61.

The **SetDataFolder** operation.

GetDefaultFont

GetDefaultFont(*winName*)

The GetDefaultFont function returns a string containing the name of the default font for the named window or subwindow.

Parameters

If *winName* is null (that is, `""`) returns the default font for the experiment.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

Only graph windows and the experiment as a whole have default fonts. If *winName* is the name of a window other than a graph (e.g., a layout), or if *winName* is not the name of any window, GetDefaultFont returns the experiment default font.

In user-defined functions, font names are usually evaluated at compile time. To use the output of GetDefaultFont in a user-defined function, you will usually need to build a command as a string expression and execute it with the **Execute** operation.

Examples

```
String fontName = GetDefaultFont("Graph0")
String command= "SetDrawEnv fname=\"" + fontName + "\", save"
Execute command
```

See Also

The **GetDefaultFontSize**, **GetDefaultFontStyle**, **FontSizeHeight**, and **FontSizeStringWidth** functions.

GetDefaultFontSize

GetDefaultFontSize(*graphNameStr, axisNameStr*)

The GetDefaultFontSize function returns the default font size of the graph or of the graph's axis (in points) in the specified window or subwindow.

Details

If *graphNameStr* is `""` the top graph is examined.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

If *axisNameStr* is `""`, the font size of the default font for the graph is returned.

If named axis exists, the default font size for the named axis in the graph is returned.

If named axis does not exist, NaN is returned.

See Also
The **GetDefaultFont**, **GetDefaultFontStyle**, **FontSizeHeight**, and **FontSizeStringWidth** functions.

GetDefaultFontStyle

GetDefaultFontStyle(*graphNameStr, axisNameStr*)
The GetDefaultFontStyle function returns the default font style of the graph or of the graph's axis in the specified window or subwindow.

Details
If *graphNameStr* is `""` the top graph is examined.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

If *axisNameStr* is `""`, the font style of the default font for the graph is returned.

If named axis exists, the default font style for the named axis in the graph is returned.

If named axis does not exist, NaN is returned.

The returned value is a binary coded number with each bit identifying one aspect of the font style as follows:

bit 0:	Bold.
bit 1:	Italic.
bit 2:	Underline.
bit 3:	Outline (*Macintosh only*).
bit 4:	Shadow (*Macintosh only*).
bit 5:	Condensed (*Macintosh only*).
bit 6:	Extended (*Macintosh only*).

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

See Also
The **GetDefaultFont**, **GetDefaultFontSize**, **FontSizeHeight**, and **FontSizeStringWidth** functions.

GetDimLabel

GetDimLabel(*waveName, dimNumber, dimIndex*)
The GetDimLabel function returns a string containing the label for the given dimension or dimension element.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers and 3 for chunks.

If *dimIndex* is -1, it returns the label for the entire dimension. If *dimIndex* is ≥ 0, it returns the dimension label for that element of the dimension.

See Also
The **SetDimLabel** function. **Dimension Labels** on page II-109 for further usage details and examples.

GetFileFolderInfo

GetFileFolderInfo [***flags***] [***fileOrFolderNameStr***]
The GetFileFolderInfo operation returns information about a file or folder.

Parameters
fileOrFolderNameStr specifies the file (or folder) for which information is returned. It is optional if /P=*pathName* and /D are specified, in which case information about the directory associated with *pathName* is returned.

If you use a full or partial path for *fileOrFolderNameStr*, see **Path Separators** on page III-398 for details on forming the path.

Folder paths should not end with single Path Separators. See the **MoveFolder Details** section.

If Igor can not determine the location of the file from *fileOrFolderNameStr* and *pathName*, it displays a dialog allowing you to specify the file to be examined. Use /D to select a folder.

Flags

/D — Uses the Select Folder dialog rather than Open File dialog when *pathName* and *fileOrFolderNameStr* do not specify an existing file or folder.

/P=*pathName* — Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

/Q — No information printed to the history window.

/Z[=*z*] — Prevents procedure execution from aborting if GetFileFolderInfo tries to get information about a file or folder that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort.

- /Z=0: Same as no /Z at all.
- /Z=1: Used for getting information for a file or folder only if it exists. /Z alone has the same effect as /Z=1.
- /Z=2: Used for getting information for a file or folder if it exists and displaying a dialog if it does not exist.

Variables

GetFileFolderInfo returns information in the following variables:

`V_flag`
- 0: File or folder was found.
- -1: User cancelled the Open File dialog.
- Other: An error occurred, such as the specified file or folder does not exist.

`S_path` — File system path of the selected file.

`V_isFile` — 1: *fileOrFolderNameStr* is a file.

`V_isFolder` — 1: *fileOrFolderNameStr* is a folder.

`V_isInvisible` — 1: File is invisible (*Macintosh*) or Hidden (*Windows*).

`V_isReadOnly` — Set if the file is locked (*Macintosh*) or is read-only (*Windows*).

On Windows, V_isReadonly is either 0 (unlocked) or 1 (locked). To set this manually, right-click the file and choose "Properties", then select or deselect the "read-only" checkbox.

On Macintosh, V_isReadonly is a bitmap of two settings:

- 0: Unlocked.
- 1: "Locked" with the Finder. Only files can be locked this way.

  You set this kind of locking with the Finder's Get Info window by selecting the "Locked" checkbox.
- 2: "File Locked" as with ResEdit. You can't set this kind of locking with the Finder, but applications can. See **SetFileFolderInfo**.
- 3: Both kinds of locking are set.

`V_creationDate` — Number of seconds since midnight on January 1, 1904 when the file or folder was first created. Use **Secs2Date** to get a text format date.

`V_modificationDate` — Number of seconds since midnight on January 1, 1904 when the file or folder was last modified. Use **Secs2Date** to get a text format date.

`V_isAliasShortcut` — 1: File is an alias (*Macintosh*) or a shortcut (*Windows*) and S_aliasPath is also set.

If *fileOrFolderNameStr* refers to a file (not a folder), GetFileFolderInfo returns additional information in the following variables:

`S_aliasPath` — Full path to the file or folder that is the source for an alias (*Macintosh*) or a shortcut (*Windows*).

When the source is a folder, S_aliasPath ends with a ":" character.

`V_isStationery` — 1: The stationery bit is set (*Macintosh*) or (*Windows*) the file type is one of the stationery file types (.pxt, .uxt, .ift).

S_fileType	Four-character file type code, such as 'TEXT' or 'IGsU' (packed experiment). On Windows, these codes are fabricated by translating from the equivalent file name extensions, such as .txt and .pxp.
S_creator	Four-character creator code, such as 'IGR0' (Igor Pro creator code). On Windows, S_creator is set to 'IGR0' if the file name extensions is one of those registered to Igor Pro, such as .pxp or .bwav (but not .txt). For other registered extensions, S_creator is set to the full file path of the registered application. Otherwise it is set to "".
V_logEOF	Number of bytes in the file data fork. For other forks, use **Open**/F and **FStatus**.
V_version	Version number of the file. On Macintosh, this is the value in the vers(1) resource. On Windows, a file version such as 3.10.2.1 is returned as 4.021: use S_fileVersion to avoid the problem of the second digit overflowing into the first digit. "0": File version can't be determined, or the file can't be examined because it is already open.
S_fileVersion	The file version as a string, added in Igor 5.02. On Macintosh, this is just a string representation of V_Version. On Windows, a file version such as 3.10.2.1 is returned as "3.10.2.1". "0": (*Macintosh*) file version can't be determined. "0.0.0.0": (*Windows*) file version can't be determined.

Details

You can change some of the file information by using **SetFileFolderInfo**.

Examples

Print the modification date of a file:

```
GetFileFolderInfo/Z "Macintosh HD:folder:afile.txt"
if( V_Flag == 0 && V_isFile )                    // file exists
   Print Secs2Date(V_modificationDate,0), Secs2Time(V_modificationDate,0)
endif
```

Determine if a folder exists (easier than creating a path with NewPath and then using PathInfo):

```
GetFileFolderInfo/Z "Macintosh HD:folder:subfolder"
if( V_Flag && V_isFolder )
   Print "Folder Exists!"
endif
```

Find the source for a shortcut or alias:

```
GetFileFolderInfo/Z "Macintosh HD:fileThatIsAlias"
if( V_Flag && V_isAliasShortcut )
   Print S_aliasPath
endif
```

See Also

The **SetFileFolderInfo**, **PathInfo**, **ImageFileInfo**, and **FStatus** operations. The **IndexedFile**, **Secs2Date**, and **ParseFilePath** functions.

GetErrMessage

GetErrMessage(*errorCode* [, *substitutionOption*])

GetErrMessage returns a string containing an explanation of the error associated with *errorCode*.

Details

errorCode is a value sometimes returned in V_Flag, as per the documentation of the an Igor function or operation; the **Execute** operation is an example.

For a few error codes, the corresponding error message is designed to be combined with "substituted" information available only immediately after the error occurs. An example is the "parameter out of range" error which produces an error message such as "expected number between *x* and *y*". To get the correct error message, you must call GetErrMessage immediately after calling the function or operation that generated the error and you must pass the appropriate value for *substitutionOption* as explained below.

Substitution

Igor maintains two environments which store the substitution information: one for macros created using the Macro, Proc and Window keywords and another for user-defined functions created with the Function keyword. The optional *substitutionOption* parameter gives you the ability to choose between those environments or to not substitute at all. Set *substitutionOption* to one of:

substitutionOption	GetErrMessage Action
0	Substitution values are filled in with "(not avail)". This is the default when *substitutionOption* is not specified.
1	Substitution values are blank.
2	Substitution is performed based on the presumption that the error was received while executing a macro or a command using Igor's command line. This includes a command executed via the Execute operation even from a user-defined function because such commands are executed as if entered in the command line.
3	Substitution is performed based on the presumption that the error was received while executing a user-defined function.

For most purposes you should pass 3 for *substitutionOption* when the error was generated in a user-defined function other than through the Execute operation and pass 2 otherwise.

Examples

```
// Macro, Execute or command line
Execute/Q/Z "Duplicate/O nonexistentWave, dup"
Print GetErrMessage(V_Flag,2)
```

Prints:

```
    expected wave name
```

```
// Function example
Function Test()
    Make/O/N=(2,2) data= 0
    FilterIIR/COEF=data/LO=999/Z data    // purposely wrong /LO value
    Print GetErrMessage(V_Flag)
    Print GetErrMessage(V_Flag,1)
    Print GetErrMessage(V_Flag,2)
    Print GetErrMessage(V_Flag,3)
End

Test()
```

Prints:

```
  expected (not avail) between (not avail) and (not avail)
  expected  between  and
  expected  between  and
  expected /LO frequency between 0 and 0.5
```

See Also

The **GetRTErrMessage** and **GetRTError** functions.

GetFormula

GetFormula(*objName*)

The GetFormula function returns a string containing the named object's dependency formula. The named object must be a wave, numeric variable or string variable.

Details

Normally an object will have an empty dependency formula and GetFormula will return an empty string (""). If you assign a expression to an object using the := operator or the SetFormula operation, the text on the right side of the := or the parameter to SetFormula is the object's dependency formula and this is what GetFormula will return.

Examples

```
Variable/G dependsOnIt
Make/O wave0 := dependsOnIt*2     //wave0 changes when dependsOnItdoes
Print GetFormula(wave0)
```

Prints the following in the history area:

```
  dependsOnIt*2
```

See Also

See **Dependency Formulas** on page IV-200, and the **SetFormula** operation.

GetIndependentModuleName

GetIndependentModuleName()

The GetIndependentModuleName function returns the name of the currently running Independent Module. If no independent module is running, it returns "ProcGlobal".

See Also

Independent Modules on page IV-214.

IndependentModuleList.

GetIndexedObjName

GetIndexedObjName(*sourceFolderStr, objectType, index*)

The GetIndexedObjName function returns a string containing the name of the indexth object of the specified type in the data folder specified by the string expression.

For Igor Pro 6.1 or later, **GetIndexedObjNameDFR** is preferred.

Parameters

sourceFolderStr can be either ":" or "" to specify the current data folder. You can also use a full or partial data folder path. *index* starts from zero. If no such object exists a zero length string ("") is returned. *objectType* is one of the following values:

objectType	What You Get
1	Waves
2	Numeric variables
3	String variables
4	Data folders

Examples

```
Function PrintAllWaveNames()
    String objName
    Variable index = 0
    do
        objName = GetIndexedObjName(":", 1, index)
        if (strlen(objName) == 0)
            break
        endif
        Print objName
        index += 1
    while(1)
End
```

See Also

The **CountObjects** function, and Chapter II-8, **Data Folders**.

GetIndexedObjNameDFR

GetIndexedObjNameDFR(*dfr, objectType, index*)

The GetIndexedObjNameDFR function returns a string containing the name of the indexth object of the specified type in the data folder referenced by *dfr*.

Requires Igor Pro 6.1 or later.

GetIndexedObjNameDFR is the same as GetIndexedObjName except the first parameter, *dfr*, is a data folder reference instead of a string containing a path.

Parameters

index starts from zero. If no such object exists a zero length string ("") is returned.

objectType is one of the following values:

objectType	What You Get
1	Waves
2	Numeric variables
3	String variables
4	Data folders

Examples

```
Function PrintAllWaveNames()
    String objName
    Variable index = 0
    DFREF dfr = GetDataFolderDFR()    // Reference to current data folder
    do
        objName = GetIndexedObjNameDFR(dfr, 1, index)
        if (strlen(objName) == 0)
            break
        endif
        Print objName
        index += 1
    while(1)
End
```

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-61.

The **CountObjectsDFR** function.

GetKeyState

GetKeyState(*flags*)

The GetKeyState function returns a bitwise numeric value that indicates the state of certain keyboard keys.

GetKeyState is normally called from a procedure that is invoked directly through a user-defined button or user-defined menu item. The procedure tests the state of one or more modifier keys and adjusts its behavior accordingly.

Another use for GetKeyState is to determine if Escape is pressed. This can be used to detect that the user wants to stop a procedure.

GetKeyState tests the keyboard at the time it is called. It does not tell you if keys were pressed between calls to the function. Consequently, when a procedure uses the escape key to break a loop, the user must press Escape until the running procedure gets around to calling the function.

Parameters

flags is reserved for future use and currently must be zero.

Details

When set, the return value is interpreted bitwise as follows:

Bit 0:	Command (*Macintosh*) or Ctrl (*Windows*) pressed.
Bit 1:	Option (*Macintosh*) or Alt (*Windows*) pressed.
Bit 2:	Shift pressed.
Bit 3:	Caps Lock pressed.
Bit 4:	Control pressed (*Macintosh only*).
Bit 5:	Escape pressed.

To test if a particular key is pressed, do a bitwise AND of the return value with the mask value 1, 2, 4, 8, 16, and 32 for bits 0 through 5 respectively. To test if a particular key and only that key is pressed compare the return value with the mask value.

On Macintosh, it is currently possible to define a keyboard shortcut for a user-defined menu item and then, in the procedure invoked by the keyboard shortcut, to use GetKeyState to test for modifier keys. This is not possible on Windows because the keyboard shortcut will not be activated if a modifier key not specified in the keyboard shortcut is pressed. It is also possible that this ability on Macintosh will be compromised by future operating system changes. On both operating systems, however, you can test for a modifier key when the user chooses a user-defined menu item or clicks a user-defined button using the mouse.

Examples

```
Function ShiftKeyExample()
    Variable keys = GetKeyState(0)

    if (keys == 0)
        Print "No modifier keys are pressed."
    endif

    if (keys & 4)
        if (keys == 4)
            Print "The Shift key and only the Shift key is pressed."
        else
            Print "The Shift key is pressed."
        endif
    endif
End

Function EscapeKeyExample()
    Variable keys

    do
        keys = GetKeyState(0)
        if ((keys & 32) != 0)          // User is pressing escape?
            break
        endif
    while(1)
End
```

See Also

Keyboard Shortcuts on page IV-118. **Setting Bit Parameters** on page IV-12 for details about bit settings.

GetLastUserMenuInfo

GetLastUserMenuInfo

The GetLastUserMenuInfo operation sets variables in the local scope to indicate the value of the last selected user-defined menu item.

Details

GetLastUserMenuInfo creates and sets these special variables:

`V_flag` The kind of menu that was selected:

V_flag	Menu Kind
0	Normal text menu item, including **Optional Menu Items** (see page IV-110) and **Multiple Menu Items** (see page IV-111).
3	"*FONT*"
6	"*LINESTYLEPOP*"

V_flag	Menu Kind
7	"*PATTERNPOP*"
8	"*MARKERPOP*"
9	"*CHARACTER*"
10	"*COLORPOP*"
13	"*COLORTABLEPOP*"

See **Specialized Menu Item Definitions** on page IV-112 for details about these special user-defined menus.

`V_value` Which menu item was selected. The value also depends on the kind of menu the item was selected from:

V_flag	V_value meaning
0	Text menu item number (the first menu item is number 1).
3	Font menu item number (use S_Value, instead).
6	Line style number (0 is solid line)
7	Pattern number (1 is the first selection, a SW-NE light diagonal).
8	Marker number (1 is the first selection, the X marker).
9	Character as an integer, = char2num(S_Value). Use S_Value instead.
10	Color menu item (use V_red, V_green, and V_blue instead).
13	Color table list menu item (use S_Value instead).

`S_value` The menu item text, depending on the kind of menu it was selected from:

V_flag	S_value meaning
0	Text menu item text.
3	Font name or “default”.
6	Name of the line style menu or submenu.
7	Name of the pattern menu or submenu.
8	Name of the marker menu or submenu.
9	Character as string.
10	Name of the color menu or submenu.
13	Color table name.

In the case of **Specialized Menu Item Definitions** (see page IV-112), S_value will be the title of the menu or submenu, etc.

`V_red`, `V_green`, `V_blue`

If a user-defined color menu ("*COLORPOP*" menu item) was chosen then these values hold the red, green, and blue values of the selected color. The values range from 0 to 65535.

Will be 0 if the last user-defined menu selection was not a color menu selection.

`S_graphName`, `S_traceName`

These are set only when any user-defined menu is chosen from a graph’s trace contextual menu. (Menu “TracePopup” or Menu “AllTracesPopup” definitions).

Initially `""` until a user-defined menu selection was made from one of these contextual menus, these are not reset for each user-defined menu selection.

S_graphName is the full host-child specification for the graph. If the graph is embedded into a host window, S_graphName might be something like "Panel0#G0". See **Subwindow Syntax** on page III-95.

S_traceName is name of the trace that was selected by the trace contextual menu, or "" if the AllTracesPopup menu was chosen. See **Subwindow Syntax** on page III-95.

Examples

A Multiple Menu Items menu definition:

```
Menu "Wave List", dynamic
    "Menu Item 1", <some command>
    "Menu Item 2", <some command>
    WaveList("*",";",""), DoSomethingWithWave()
End
```

If the user selects one of the (many) menu items created by the "Wave List" menu item definition, the DoSomethingWithWave user function can call GetLastUserMenuInfo to determine which wave was selected:

```
Function DoSomethingWithWave()
    GetLastUserMenuInfo
    WAVE/Z selectedWave = $S_value
    …use selectedWave for something…
End
```

A trivial user-defined color menu definition:

```
Menu "Color"
    "*COLORPOP*", DoSomethingWithColor()
End

Function DoSomethingWithColor()
    GetLastUserMenuInfo
    …do something with V_red, V_green, V_blue…
End
```

See **Specialized Menu Item Definitions** on page IV-112 for another color menu example.

A Trace contextual menu Items menu definition:

```
Menu "TracePopup", dynamic        // menu when a trace is right-clicked
    "-"                       // separator divides this from built-in menu items
    ExportTraceName(), ExportSelectedTrace()
End

Function/S ExportTraceName()
    GetLastUserMenuInfo     // only S_graphName, S_traceName are set.
    Return "Export "+S_traceName
End

Function ExportSelectedTrace()
    GetLastUserMenuInfo
    …do something with S_graphName, S_traceName…
End
```

See Also

Chapter IV-5, **User-Defined Menus** and especially the sections **Optional Menu Items** on page IV-110, **Multiple Menu Items** on page IV-111, and **Specialized Menu Item Definitions** on page IV-112.

GetMarquee

GetMarquee [**/K/W=*winName*/Z**] [***axisName*** [**, *axisName***]]

The GetMarquee operation provides a way for you to use the marquee as an input mechanism in graphs and page layout windows. It puts information about the marquee into variables.

Parameters

If you specify *axisName* (allowed only for graphs) the coordinates are in axis units. If you specify an axis that does not exist, Igor generates an error.

If you specify only one axis then Igor sets only the variables appropriate to that axis. For example, if you execute "GetMarquee left" then Igor sets the V_bottom and V_top variables but does *not* set V_left and V_right.

Flags

/K Kills the marquee. Usually you will want to kill the marquee when you call GetMarquee, so you should use the /K flag. This is modeled after what happens when you create a marquee in a graph and then choose Expand from the Marquee menu.

There may be some situations in which you want the marquee to persist. Igor also automatically kills the marquee anytime the window containing the marquee is deactivated, including when a dialog is summoned.

/W=*winName* Specifies the named window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/Z No runtime error generated if the target window isn't a graph or layout, but V_flag will be zero. /Z does not prevent other kinds of problems from generating a runtime error.

Details

GetMarquee is intended to be used in procedures invoked through user menu items added to the graph Marquee menu and the layout Marquee menu.

GetMarquee sets the following variables and strings:

`V_flag`	0: There was no marquee when GetMarquee was invoked.
	1: There was a marquee when GetMarquee was invoked.
`V_left`	Marquee left coordinate.
`V_right`	Marquee right coordinate.
`V_top`	Marquee top coordinate.
`V_bottom`	Marquee bottom coordinate.
`S_marqueeWin`	Name of window containing the marquee, or `""` if no marquee. If subwindow, subwindow syntax will be used.

When called from the command line, GetMarquee sets global variables and strings in the current data folder. When called from a procedure, it sets local variables and strings.

In addition, creating, adjusting, or removing a marquee may set additional marquee global variables (see the **Marquee Globals** section, below).

The target window must be a layout or a graph. Use /Z to avoid generating a runtime-error (V_flag will be 0 if the target window was not a layout or graph).

If the target is a layout then Igor sets the variables in units of points relative to the top/left corner of the paper.

If the target is a graph then Igor sets V_left and V_right based on the specified horizontal axis. If no horizontal axis was specified, V_left and V_right are set relative to the left edge of the base window in points.

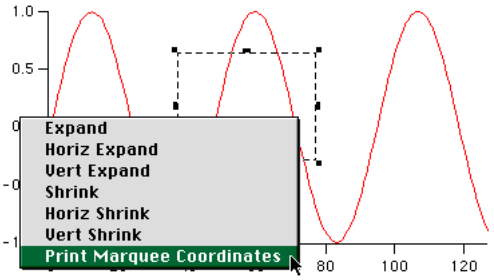
If the target is a graph then Igor sets V_bottom and V_top based on the specified vertical axis. If no vertical axis was specified, V_top and V_bottom are set relative to the top edge of the base window in points.

If there is no marquee when you invoke GetMarquee then Igor sets V_left, V_top, V_right, V_bottom based on the last time the marquee was active.

GetMarquee Example

```
Menu "GraphMarquee"
    "Print Marquee Coordinates", PrintMarqueeCoords()
End

Function PrintMarqueeCoords()
    GetMarquee left, bottom
    if (V_flag == 0)
        Print "There is no marquee"
    else
        printf "marquee left in bottom axis terms: %g\r", V_left
        printf "marquee right in bottom axis terms: %g\r", V_right
        printf "marquee top in left axis terms: %g\r", V_top
        printf "marquee bottom in left axis terms: %g\r", V_bottom
    endif
End
```

You can run this procedure by putting it into the procedure window, making a marquee in a graph, clicking in the marquee and choosing Print Marquee Coordinates:



The procedure calls GetMarquee to set the local marquee variables and then prints their values in the history area:

```
•PrintMarqueeCoords()
  marquee left in bottom axis terms: 32.1149
  marquee right in bottom axis terms: 64.7165
  marquee top in left axis terms: 0.724075
  marquee bottom in left axis terms: -0.131061
```

Marquee Globals

You can cause Igor to update global marquee variables whenever the user adjusts the marquee (without the need for you to invoke GetMarquee) by creating a global variable named V_marquee in the root data folder:

```
Variable/G root:V_marquee = 1     //Creates V_marquee and sets bit 0 only
```

When the user adjusts the marquee Igor checks to see if root:V_marquee exists and which bits are set, and updates (and creates if necessary) these globals:

`Variable/G root:V_left`	Marquee left coordinate.
`Variable/G root:V_right`	Marquee right coordinate.
`Variable/G root:V_top`	Marquee top coordinate.
`Variable/G root:V_bottom`	Marquee bottom coordinate.
`String/G root:S_marqueeWin`	Name of window that contains marquee, or `""` if no marquee. Set only if root:V_Marquee has bit 15 (0x8000) set.

Unlike the local variables, for graphs these global variables are never in points. Root:V_left and V_right will be axis coordinates based on the first bottom axis created for the graph (if none, then for the first top axis). The axis creation order is the same as is returned by **AxisList**. Similarly, root:V_top and root:V_bottom will be axis coordinates based on the first left axis or the first right axis.

Igor examines the global root:V_marquee for bitwise flags to decide which globals to update, and when:

root:V_marquee Bit Meaning	Bit Number	Bit Value
Update global variables for graph marquees	0	1
Update global variables for layout marquees	2	4
Update S_marqueeWin when updating global variables	15	0x8000

Marquee Globals Example

By creating the global variable root:V_marquee this way:

```
Variable/G root:V_marquee = 1 + 4 + 0x8000
```

whenever the user creates, adjusts, or removes a marquee in any graph or layout Igor will create and update the global root:V_left, etc. coordinate variables and set the global string root:S_marqueeWin to the name of the window which has the marquee in it. When the marquee is removed, root:S_marqueeWin will be set to `""`.

This mechanism does neat things by making a **ValDisplay** or **SetVariable** control depend on any of the globals. See the Marquee Demo experiment in the Examples:Feature Demos folder for an example.

You can also cause a function to run whenever the user creates, adjusts, or removes a marquee by setting up a dependency formula using **SetFormula** to bind one of the marquee globals to one of the function's input arguments:

```
Variable/G root:dependencyTarget
SetFormula root:dependencyTarget, "MyMarqueeFunction(root:S_marqueeWin)"
Function MyMarqueeFunction(marqueeWindow)
    String marqueeWindow          // this will be root:S_marqueeWin
    if( strlen(marqueeWindow) )
        NVAR V_left= root:V_left, V_right= root:V_right
        NVAR V_top= root:V_top, V_bottom= root:V_bottom
        Printf marqueeWindow + " has a marquee at: "
        Printf "%d, %d, %d, %d\r", V_left, V_right, V_top, V_bottom
    else
        Print "The marquee has disappeared."
    endif
    return 0                      // return value doesn't really matter
End
```

See Also

The **SetMarquee** and **SetFormula** operations. **Setting Bit Parameters** on page IV-12 for information about bit settings.

GetRTError

GetRTError(*flag*)

The GetRTError function returns information about the error state of Igor's user-defined function runtime execution environment.

If *flag* is 0, GetRTError returns an error code if an error has occurred or 0 if no error has occurred.

If *flag* is 1, GetRTError returns an error code if an error has occurred or 0 if no error has occurred and it clears the error state of Igor's runtime execution environment. Use this if you want to detect and handle runtime errors yourself.

If *flag* is 2, GetRTError returns the state of Igor's internal abort flag but does not clear it.

For *flag*=0 and *flag*=1, you can call **GetErrMessage** to obtain the error message associated with the returned error code, if any.

Example

```
// This illustrates how to detect and handle a runtime error
// rather than allowing it to cause Igor to abort execution.
Function Demo()
    <Call an Igor operation or a user-defined function>
    Variable err = GetRTError(0)
    if (err != 0)
        String message = GetErrMessage(err)
        Printf "Error in Demo: %s\r", message
        err = GetRTError(1)                    // Clear error state
        Print "Continuing execution"
    endif
    <Call an Igor operation or a user-defined function>
End
```

See also

The **GetErrMessage** and **GetRTErrMessage** functions.

GetRTErrMessage

GetRTErrMessage()

In a user function, GetRTErrMessage returns a string containing the name of the operation that caused the error, a semicolon, and an error message explaining the cause of the error. This is the same information that appears in the alert dialog displayed. If no error has occurred, the string will be of zero length. GetRTErrMessage must be called before the error is cleared by calling GetRTError with a nonzero argument.

See also

The **GetRTError** and **GetErrMessage** functions.

GetRTStackInfo

GetRTStackInfo(*selector*)

The GetRTStackInfo function returns information about "runtime stack" (the chain of macros and functions that are executing).

Details

If *selector* is 0, GetRTStackInfo returns a semicolon-separated list of the macros and procedures that are executing. This list is the same you would see in the debugger's stack list.

The currently executing macro or function is the last item in the list, the macro or function that started execution is the first item in the list.

If *selector* is 1, it returns the name of the currently executing function or macro.

If *selector* is 2, it returns the name of the calling function or macro.

If *selector* is 3, GetRTStackInfo returns a semicolon-separated list of routine names, procedure file names and line numbers. This is intended for advanced debugging by advanced programmers only.

For example, if RoutineA in procedure file ProcA.ipf calls RoutineB in procedure file ProcB.ipf, and RoutineB calls GetRTStackInfo(3), it will return:

```
RoutineA,ProcA.ipf,7;RoutineB,ProcB.ipf,12;
```

The numbers 7 and 12 would be the actual numbers of the lines that were executing in each routine. Line numbers are zero-based.

GetRTStackInfo does not work correctly with string macros executed via the Execute operation.

In future versions of Igor, *selector* may request other kinds of information.

Examples

```
Function Called()
   Print "Called by " + GetRTStackInfo(2) + "()"
   Print "Routines in calling chain: " + GetRTStackInfo(0)
End

Function Calling()
   Called()
End

Macro StartItUp()
   Calling()
End
```

Executing on the Command line:

```
•StartItUp()
  Called by Calling()
  Routines in calling chain: StartItUp;Calling;Called;
```

See Also

StringFromList, **ItemsInList**, and **GetRTError** functions. **The Stack and Variables Lists** on page IV-187.

GetScrapText

GetScrapText()

The GetScrapText function returns a string containing any plain text on the Clipboard (aka "scrap"). This is the text that would be pasted into a text document if you used Paste in the Edit menu.

See Also

The **PutScrapText** and **LoadPICT** operations.

GetSelection

GetSelection *winType, winName, bitflags*

The GetSelection operation returns information about the current selection in the specified window.

Parameters

winType is one of the following keywords:

```
graph, panel, table, layout, notebook, procedure
```

winName is the name of a window of the specified type.

When identifying a subwindow with winName, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

If *winType* is procedure then *winName* is actually a procedure window title inside a $"" wrapper, such as:

```
GetSelection procedure $"DemoLoader.ipf", 3
```

bitflags is a bitwise parameter that is used in different ways for different window types, as described in **Details**. You should use 0 for undefined bits. **Setting Bit Parameters** on page IV-12 for further details about bit settings.

Details

For all window types, GetSelection sets V_flag:

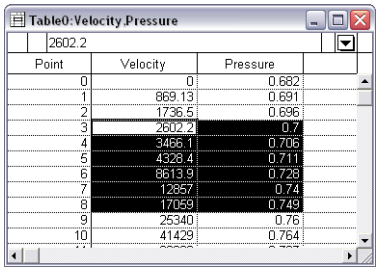
`V_flag`	0:	No selection when GetSelection was invoked.
	1:	There was a selection when GetSelection was invoked.

Here is a description of what GetSelection does for each window type:

winType	*bitFlags*	**Action**
`graph`		Does nothing.
`panel`		Does nothing.
`table`	1	Sets V_startRow, V_startCol, V_endRow, and V_endCol based on the selected cells in the table. The top/left cell, not including the Point column, is (0, 0).
	2	Sets S_selection to a semicolon-separated list of column names.
	4	Sets S_dataFolder to a semicolon-separated list of data folders, one for each column.
`layout`	2	Sets S_selection to a semicolon separated list of selected objects in the layout layer (not any drawing layers). S_selection will be `""` if no objects are selected.
`notebook`	1	Sets V_startParagraph, V_startPos, V_endParagraph, and V_endPos based on the selected text in the notebook.
	2	Sets S_selection to the selected text.
`procedure`	1	Sets V_startParagraph, V_startPos, V_endParagraph, V_endPos based on the selected text in the procedure window.
	2	Sets S_selection to the selected text.

Examples

Make a table named “Table0” with some columns, and select some combination of rows and columns:



and execute these commands in a procedure or in the command line:

```
GetSelection table, Table0, 3
Print V_flag, V_startRow, V_startCol, V_endRow, V_endCol
Print S_selection
```

This will print the following in the history area:

```
  1   3   0   8   1
  Velocity.d;Pressure.d;
```

GetUserData

GetUserData(*winName, objID, userdataName*)

The GetUserData function returns a string containing the user data for a window or subwindow. The return string will be empty if no user data exists.

Parameters

winName may specify a window or subwindow name. Use "" for the top window.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

objID is a string specifying the name of a control or graph trace. Use "" for a window or subwindow.

userdataName is the name of the user data or "" for the default unnamed user data.

See Also

The **ControlInfo**, **GetWindow**, and **SetWindow** operations.

GetWavesDataFolder

GetWavesDataFolder(*waveName, kind*)

The GetWavesDataFolder function returns a string containing the name of a data folder containing the wave named by *waveName*. Variations on the theme are selected by *kind*.

The most common use for this is in a procedure, when you want to create a wave or a global variable in the data folder containing a wave passed as a parameter.

For Igor Pro 6.1 or later, **GetWavesDataFolderDFR** is preferred.

Details

kind	GetWavesDataFolder Returns
0	Only the name of the data folder containing *waveName*.
1	Full path of data folder containing *waveName*, without wave name.
2	Full path of data folder containing *waveName*, including possibly quoted wave name.
3	Partial path from current data folder to the data folder containing *waveName*.
4	Partial path including possibly quoted wave name.

Kinds 2 and 4 are especially useful in creating command strings to be passed to **Execute**.

Examples

```
Function DuplicateWaveInDataFolder(w)
   Wave w
   String dfSav = GetDataFolder(1)
   SetDataFolder GetWavesDataFolder(w,1)
   Duplicate/O w, $(NameOfWave(w) + "_2")
   SetDataFolder dfSav
End
```

See Also

Chapter II-8, **Data Folders**.

GetWavesDataFolderDFR

GetWavesDataFolderDFR(*waveName*)

The GetWavesDataFolderDFR function returns a data folder reference for the data folder containing the specified wave.

Requires Igor Pro 6.1 or later.

GetWavesDataFolderDFR is the same as GetWavesDataFolder except that it returns a data folder reference instead of a string containing a path.

See Also
Chapter II-8, **Data Folders** and **Data Folder References** on page IV-61.

GetWindow

GetWindow [/Z] *winName, keyword*

The GetWindow operation provides information about the named window or subwindow. Information is returned in variables, strings, and waves.

Parameters

winName can be the name of graph, table, panel, page layout, notebook, or any subwindow. It can also be the title of a procedure window or one of these four special keywords:

kwTopWin	Specifies the topmost graph, table, panel, page layout, or notebook window.
kwCmdHist	Specifies the command history window.
kwFrameOuter	Specifies the "frame" or "application" window that Igor Pro has only under Windows. This is the window that contains Igor's menus and status bar.
kwFrameInner	Specifies the inside of the same "frame" window under Windows. This is the window that all other Igor windows are inside.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Only one of the following keywords may follow *winName*. The keyword chosen determines the information stored in the output variables:

active	Sets V_Value to 1 if the window is active or to 0 otherwise. Active usually means the window is the frontmost window.
activeSW	Stores the window "path" of currently active subwindow in S_Value. See **Subwindow Syntax** on page III-95 for details on the window hierarchy.
exterior	Sets V_value to 1 if the window is an exterior panel window or to 0 otherwise. Useful for window hook functions that must work for both regular windows and exterior panel windows, since exterior panels use their own hook function.
gsize	Reads graph outer dimensions into V_left, V_right, V_top, and V_bottom in local coordinates. This includes axes but not the tool bar, control bar, or info panel. Dimensions are in points.
gsizeDC	Same as gsize but dimensions are in device coordinates (pixels).
hide	Sets V_Value bit 0 if window or subwindow is hidden. Sets bit 1 if the host window is minimized.
hook	Copies name of window hook function to S_value. See **Unnamed Window Hook Functions** on page IV-271.
hook(*hName*)	For the given named hook *hName*, copies name of window hook function to S_value. See **Named Window Hook Functions** on page IV-265.
logicalpapersize	Returns logical paper size of the page setup associated with the named window into V_left, V_right, V_top, and V_bottom. Dimensions are in points. If the Page Setup dialog uses 100% scaling, these are also the physical dimensions of the page. V_left and V_top are 0 and correspond to the left top corner of the physical page. On the Macintosh, using a Scale of 50% multiplies all of these dimensions by 2.
logicalprintablesize	Returns logical printable size of the page setup associated with the named window into V_left, V_right, V_top, and V_bottom. Dimensions are in points. If the Page Setup dialog uses 100% scaling, these are also the physical dimensions of the page minus the margins. V_left and V_top are the number of points from the left top corner of the physical page to the left top corner of the printable area of page. On the Macintosh, using a page setup scale of 50% multiplies all of these dimensions by 2.
maximize	Sets V_Value to 1 if the window is maximized, 0 otherwise. On Macintosh, V_Value is always 0.
needUpdate	Sets V_Value to 1 if window or subwindow is marked as needing an update.

note	Copies window note to S_value.
psize	Reads graph plot area dimensions (where the traces are) into V_left, V_right, V_top, and V_bottom in local coordinates. Dimensions are in points.
psizeDC	Same as psize but dimensions are in device coordinates (pixels).
title	Gets the title (set by as *titleStr* with NewPanel, Display, etc., or by the Window Control dialog) and puts it into S_value. S_value is set to "" if *winName* specifies a subwindow. See also the wtitle keyword, below.
userdata	Returns the primary (unnamed) user data for a window in S_value. Use **GetUserData** to obtain any named user data.
wavelist	Creates a three-column text wave called W_WaveList containing a list of waves used in the graph in *winName*. Each wave occupies one row in W_WaveList. This list includes all waves that can be in a graph, including the data waves for contour plots and images.
wsize	Reads window dimensions into V_left, V_right, V_top, and V_bottom in points from the top left of the screen. For subwindows, values are local coordinates in the host.
wsizeDC	Same as wsize but dimensions are in local device coordinates (pixels). The origin is the top left corner of the host window's active rectangle.
wsizeOuter	Reads window dimensions into V_left, V_right, V_top, and V_bottom in points from the top left of the screen. Dimensions are for the entire window including any frame and title bar. For subwindows, values are for the host window.
wsizeOuterDC	Same as wsizeOuter but dimensions are in local device coordinates (pixels). The origin is the top left corner of the host window's active rectangle, so V_top will be negative for a window with a title bar. V_left will be negative for windows with a frame; windows on Macintosh OS X have no frame, so V_left will be zero.
wsizeRM	Generally the same as wsize, but these are the coordinates that would actually be used by a recreation macro except that the coordinates are in points even if the window is a panel. Also, if the window is minimized or maximized, the coordinates represent the window's restored location.
wtitle	Gets the actual window title displayed in the window's title bar, regardless of whether it was set by the user (see the title keyword above) or is the default title created by Igor, and puts it into S_value. S_value is set to "" if winName specifies a subwindow. If winName is kwFrameOuter or kwFrameInner, on Macintosh S_Value is set to the name of the Igor application. On Windows it is set to the full title of the application as seen on the frame's window, which can be altered using DoWindow/T kwFrame.

Flags

/Z	Suppresses error if, for instance, *winName* doesn't name an existing window. V_flag is set to zero if no error occurred or to a non-zero error code.

Details

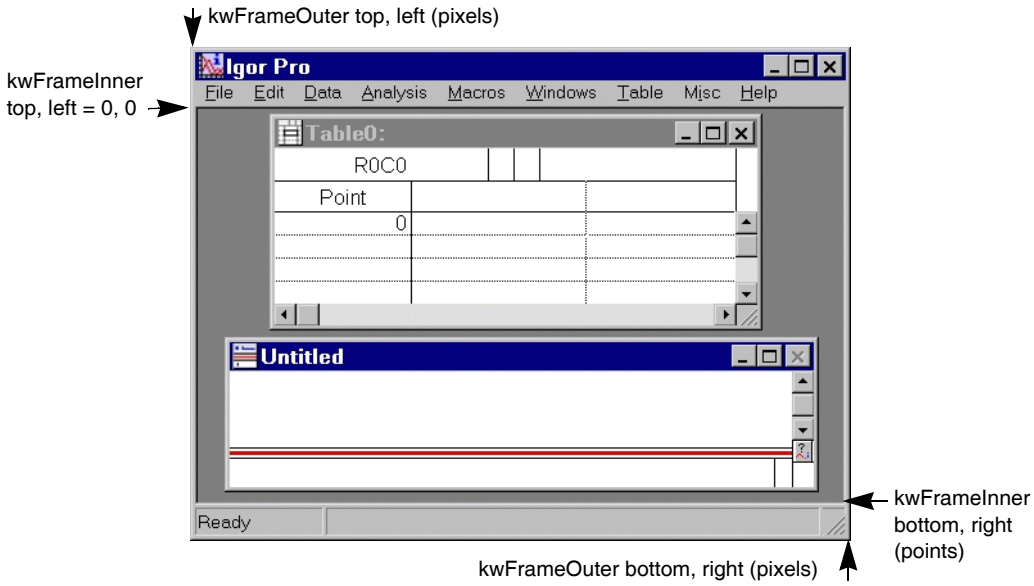
The wsize parameter is appropriate for all windows.

The gsize, psize, and wavelist parameters are appropriate only for graph windows.

The logicalpapersize and logicalprintablesize parameters are appropriate for all printable windows (not panels).

kwCmdHist, kwFrameInner, and kwFrameOuter may be used with only the wsize keyword.

On Windows computers, kwFrameInner and kwFrameOuter return coordinates into V_left, V_right, V_top, and V_bottom. On the Macintosh, they always return 0 (because Igor has no frame on the Macintosh).



kwFrameOuter coordinates are the location of the outer edges of Igor's application window, expressed in screen (pixel) coordinates suitable for use with `MoveWindow/F` to restore, minimize, or maximize the Igor application window.

If Igor is currently minimized, kwFrameOuter returns 0 for all values. If maximized, it returns 2 for all values. Otherwise, the screen (pixel) coordinates of the frame are returned in V_left, V_right, V_top, and V_bottom. This is consistent with `MoveWindow/F`.

kwFrameInner coordinates, however, are the location of the inner edges of the application window, expressed in Igor window coordinates (points) suitable for positioning graphs and other windows with **MoveWindow**.

If Igor is currently minimized, kwFrameInner returns the inner frame coordinates Igor would have if Igor were "restored" with `MoveWindow/F 1,1,1,1`.

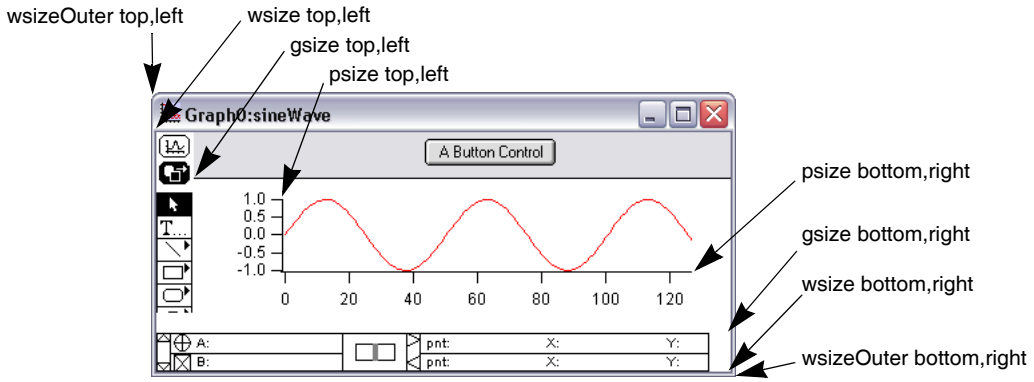
V_left and V_top will always both be 0, and V_Bottom and V_Right will be the maximum visible (or potentially visible) window (not screen) coordinates in points.

winName can be the title of a procedure window. If the title contains spaces, use:

```
GetWindow $"Title With Spaces" wsize
```

However, if another window has a name which matches the given procedure window title, that window's properties are returned instead of the procedure window.

"Local coordinates" are relative to the top left of the graph area, regardless of where that is on the screen or within the graph window. All dimensions are reported in units of points (1/72 inch) regardless of screen resolution. On the Macintosh, this is the same as screen pixels.



The format of W_WaveList, created with the wavelist keyword, is as follows:

Column 1	Column 2	Column 3
Wave name	partial path to the wave	special ID number

The wave name in column 1 is simply the name of the wave with no path. It may be the same as other waves in the list, if there are waves from different data folders.

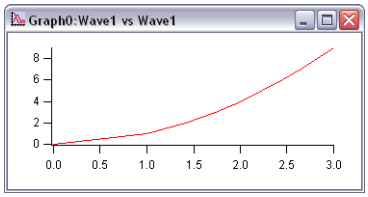
The partial path in column 2 includes the wave name and can be used with the $ operator to get access to the wave.

The special ID number in column 3 has the format ##<number>##. A version of the recreation macro for the graph can be generated that uses these ID numbers instead of wave names (see the **WinRecreation** function). This makes it relatively easy to find every occurrence of a particular wave using a function like **strsearch**.

For instance, executing these commands:

```
NewDataFolder/S Folder1
Make/N=10 Wave1=x
SetDataFolder ::
NewDataFolder/S Folder2
Make/N=10 Wave1=sqrt(x)
SetDataFolder ::
Display :Folder1:wave1 vs :Folder2:wave1
```

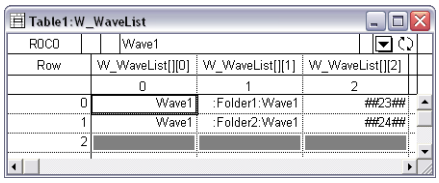
Makes a graph similar to this:



Executing these commands:

```
GetWindow kwTopWin, wavelist
Edit W_WaveList
```

makes a table similar to this:



Examples

```
// These commands draw a red foreground rectangle framing
// the printable area of a page layout window.
GetWindow Layout0 logicalpapersize
DoWindow/F Layout0
SetDrawLayer/K userFront
SetDrawEnv linefgc=(65535,0,0), fillpat=0         // Transparent fill
DrawRect V_left+1, V_top+1, V_right-1, V_bottom-1

// These commands demonstrate the difference between title and wtitle.
Make/O data=x
Display/N=MyGraph data
GetWindow MyGraph title;Print S_Value             // Prints nothing (S_Value = "")
GetWindow MyGraph wtitle;Print S_Value            // Prints "MyGraph:data"
DoWindow/T MyGraph, "My Title for My Graph"
GetWindow MyGraph title;Print S_Value             // Prints "My Title for My Graph"
GetWindow MyGraph wtitle;Print S_Value            // Prints "My Title for My Graph"
```

See Also

The **SetWindow**, **GetUserData**, **MoveWindow** and **DoWindow** operations.

The **IgorInfo** function.

gnoise

gnoise(*num* [, *RNG*])

The gnoise function returns a random value from a Gaussian distribution such that the standard deviation of an infinite number of such values would be *num*.

The random number generator is initialized using the system clock when you start Igor, virtually guaranteeing that you will never get the same sequence twice. If you want repeatable "random" numbers, use **SetRandomSeed**.

The Gaussian distribution is achieved using a Box-Muller transformation of uniform random numbers.

The optional parameter *RNG* selects one of two different pseudo-random number generators used to create the uniformly-distributed random numbers used as the input to the Box-Muller transformation. If omitted, the default is 1. The *RNG*'s are:

RNG	Description
1	Linear Congruential generator by L'Ecuyer with added Bayes-Durham shuffle. The algorithm is described in *Numerical Recipes* (2nd edition) as the function `ran2()`. This RNG has nearly 2^{32} distinct values and the sequence of random numbers has a period in excess of 10^{18}.
2	Mersenne Twister by Matsumoto and Nishimura. It is claimed to have better distribution properties and period of 2^{19937}-1.

See Also

The **SetRandomSeed** operation and the **enoise** function.

Noise Functions on page III-332.

References

Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

Details about the Mersene Twister are in:

Matsumoto, M., and T. Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, *ACM Trans. on Modeling and Computer Simulation*, *8*, 3-30, 1998.

More information is available online at: <http://en.wikipedia.org/wiki/Mersenne_twister>

Graph

Graph

Graph is a procedure subtype keyword that identifies a macro as being a graph recreation macro. It is automatically used when Igor creates a window recreation macro for a graph. See **Procedure Subtypes** on page IV-179 and **Saving and Recreating Graphs** on page II-300 for details.

GraphMarquee

GraphMarquee

GraphMarquee is a procedure subtype keyword that puts the name of the procedure in the graph Marquee menu. See **Marquee Menu as Input Device** on page IV-140 for details.

GraphNormal

GraphNormal [**/W=*winName***]

The GraphNormal operation returns the target or named graph to the normal mode, exiting any drawing mode that it may be in.

You would usually enter normal mode by choosing ShowTools from the Graph menu and clicking the crosshair tool.

Flags

/W=winName Reverts the named graph window. This must be the first flag specified when used in a Proc or Macro or on the command line.

See Also

The **GraphWaveDraw** and **GraphWaveEdit** operations.

GraphStyle

GraphStyle

GraphStyle is a procedure subtype keyword that puts the name of the procedure in the Style pop-up menu of the New Graph dialog and in the Graph Macros menu. See **Graph Style Macros** on page II-300 for details.

GraphWaveDraw

GraphWaveDraw [***flags***] [***yWaveName, xWaveName***]

The GraphWaveDraw operation initiates drawing a curve composed of *yWaveName* vs *xWaveName* in the target or named graph. The user draws the curve using the mouse, and the values are stored in a pair of waves as XY data.

Normally, you would initiate drawing by choosing ShowTools from the Graph menu and clicking in the appropriate tool rather than using GraphWaveDraw.

Parameters

yWaveName and *xWaveName* will contain the y and x values of the curve drawn by the user with the mouse.

If *yWaveName* and *xWaveName* do not already exist, they are created with two points which are initially set to NaN (Not a Number) and appended to the target.

If *yWaveName* and *xWaveName* already exist, an error is generated unless the /O (overwrite) flag is present. If /O is present, the waves are re-created — with two points which are initially set to NaN — and appended to the target if they are not already in it.

If *yWaveName* and *xWaveName* are omitted then waves called W_YPoly*n* and W_XPoly*n* are created with two points set to NaN and appended to the target (*n* is some digit, so Igor might create a wave named W_YPoly0, for example).

Flags

/F[=*f*]	Initiates freehand drawing. In normal drawing, you click where you want a data point. In freehand drawing, you click once and then draw with the mouse button held down. If present, *f* specifies the smoothing factor. Max value is 8 (which is really slow), min value is 0 (default). The drawing tools use a value of 3 which is the recommended value.
/L/R/B/T	Specifies which axes to use (Left, Right, Bottom, Top). Bottom and Left axes are used by default. Can specify free axes using /L=*axis name* type notation. See **AppendToGraph** for details. If necessary, the specified axes will be created. If an axis is created its range is set to -1 to 1.
/M	Specifies that the curve being edited must be monotonic in the X dimension. The user is not allowed drag points so that they cross horizontally.
/O	Overwrites *yWaveName* and *xWaveName* if they already exist.
/W=*winName*	Draws in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

Once drawing starts no other user actions are allowed.

In normal mode, drawing stops when you double-click or when you click the first point (in which case the last point is set equal to the first point). When drawing finishes, the edit mode is entered.

In freehand mode, drawing stops when the mouse is released or when 10000 points have been drawn.

If /O is used and the waves are already on the graph then the first instance on the graph will be used even if they use a different pair of axes than specified.

See Also

The **GraphNormal**, **GraphWaveEdit** and **DrawAction** operations.

GraphWaveEdit

GraphWaveEdit [***flags***] ***traceName***

The GraphWaveEdit operation initiates editing a wave trace in a graph. The wave trace must already be in the graph.

Normally, you would initiate editing by choosing ShowTools from the Graph menu and clicking in the appropriate tool rather than using GraphWaveEdit.

Parameters

traceName is a wave name, optionally followed by the # character and an instance number: "myWave#1" is the *second* instance of myWave appended to the graph ("myWave" is the first).

If *traceName* is omitted then you get to pick the wave trace to edit by clicking it.

Flags

/M	Specifies that the edited trace must be monotonic in the X dimension. You cannot drag points so that they cross horizontally.
/NI	Suppresses automatic new point insertion when clicking between points.
/T=*t*	Sets the trace mode. *t*=0: Lines and small square markers (default). *t*=1: User settings unchanged.
/W=*winName*	Edits traces in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

The GraphWaveEdit operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

See Also

The **GraphNormal**, **GraphWaveDraw** and **DrawAction** operations.

Grep

Grep [***flags***][***srcFileStr***][***srcTextWaveName***][**as** [***destFileOrFolderStr***] [***destTextWaveName***]]

The Grep operation copies lines matching a search expression from a file on disk, the Clipboard, or rows from a text wave to a new or existing file, an existing text wave, the History area, the Clipboard, or to S_value as a string list.

Source Parameters

The optional *srcFileStr* can be

- The full path to the file to copy lines from (in which case /P is not needed).
- The partial path relative to the folder associated with *pathName.*
- The name of a file in the folder associated with *pathName.*
- "Clipboard" to read lines of text from the Clipboard (in which case /P is ignored).

If Igor can not determine the location of the source file from *srcFileStr* and *pathName*, it displays a dialog allowing you to specify the source file.

The optional *srcTextWaveName* is the name or path to a text wave.

Only one of *srcFileStr* or *srcTextWaveName* may be specified. If neither is specified then an Open File dialog is presented allowing you to specify a source file.

Destination Parameters

The optional *destFileOrFolderStr* can be

- The name of (or path to) an existing folder when /D is specified.

- The name of (or path to) a possibly existing file.
- "Clipboard", in which case the matching lines are copied to the Clipboard (and /P and /D are ignored). The text can be retrieved with the **GetScrapText** function.

If *destFileOrFolderStr* is a partial path, it is relative to the folder associated with *pathName*.

If /D is specified, the source file is created inside the folder using the source file name.

If Igor can not determine the location of the destination file from *pathName*, *srcFileStr*, and *destFileOrFolderStr*, it displays a Save File dialog allowing you to specify the destination file (and folder).

The optional *destTextWaveName* is the name or path to an existing text wave. It may be the same wave as *srcTextWaveName*.

Only one of *destFileOrFolderStr* or *destTextWaveName* may be specified.

If no destination file or text wave is specified then matching lines are printed in the history area, unless the /Q flag is specified, in which case the matching lines aren't printed or copied anywhere (though the output variables are still set).

Use /LIST to set S_value to a string list containing the matching lines or rows.

Use /INDX to create a wave W_index containing the zero-based row or line number where matches were found.

Parameter Details

If you use a full or partial path for either *srcFileStr* or *destFileOrFolderStr*, see **Path Separators** on page III-398 for details on forming the path.

Folder paths should not end with single path separators. See **MoveFolder**'s Details section.

Flags

/A
: Appends matching lines to the destination file, creating it if necessary, appends text to the Clipboard if the *destFileOrFolderStr* is "Clipboard", or appends rows to the destination text wave. Has no effect on output to the History area.

/D
: Interprets *destFileOrFolderStr* as the name of (or path to) an existing folder (or directory). Without /D, *destFileOrFolderStr* is the name of (or path to) a file. It is ignored if the destination is a text wave, the Clipboard, or the History area.

: If *destFileOrFolderStr* is not a full path to a folder, it is relative to the folder associated with *pathName*.

/DCOL={*colNum*}
: Useful only when the destination is *destTextWaveName*.

: Copies matching lines of text from the source file, Clipboard, or *srcTextWaveName* to column *colNum* of *destTextWaveName*, with any terminator characters removed.

: The default when the source is a file or the Clipboard is `/DCOL={0}`, which copies matching lines into the first column of *destTextWaveName*.

: The default when the source is *srcTextWaveName* is to copy each column of a matched row to the corresponding column in *destTextWaveName*.

: /DCOL must be used with the /A flag, otherwise the destination wave will have only 1 column.

/DCOL={[*colNum*] [, *delimStr*], …}
: Useful only when the source is *srcTextWaveName* and the destination is a file, the Clipboard, History area, or S_value.

: Copies multiple columns in any order from matching rows of *srcTextWaveName* to the destination file, Clipboard, History area, or S_value.

: Construct the line by appending the contents of the numbered column and the *delimStr* parameters in the order specified. The output line is terminated as described in **Line Termination**.

/E=*regExprStr*
: Specifies the Perl-compatible regular expression string. A line must match the regular expression to be copied to the output file. See **Regular Expressions**.

: *Multiple /E flags may be specified*, in which case a line is copied only if it matches every regular expression.

/E={*regExprStr*, *reverse*}

Specifies the Perl-compatible regular expression string, *regExprStr*, for which the sense of the match can be changed by *reverse*.

reverse=1: Matching expressions are taken to **not** match, and vice versa. For example, use `/E={"CheckBox",1}` to list all lines that do not contain "CheckBox".

reverse=0: Same as /E=*regExptrStr*.

/GCOL=*grepCol*

Greps the specified column of *srcTextWaveName*, which is a two-dimensional text wave. The default search is on the first column (`grepCol=0`). Use `grepCol=-1` to match against any column of *srcTextWaveName*.

Does not apply if the source is a file or Clipboard.

/I

Requires interactive searching even if *srcFileStr* and *destFileOrFolderStr* are specified and if the source file exists. Same as /I=3.

/I=*i*

Specifies the degree of file search interactivity with the user.

i =0: Default; interactive only if *srcFileStr* is not specified or if the source file is missing. Same as if /I were not specified.

Note: This is different behavior than other commands such as **CopyFile**.

i =1: Interactive even if *srcFileStr* is specified and the source file exists.

i =2: Interactive even if *destFileOrFolderStr* is specified.

i =3: Interactive even if *srcFileStr* is specified, the source file exists, and *destFileOrFolderStr* is specified.

/INDX

Creates in the current data folder an output wave W_Index containing the line numbers (or row numbers) where matching lines were found. If this is the only output you need, also use the /Q flag.

/LIST[=*listSepStr*]

Creates an output string variable S_value containing a semicolon-separated list of the matching lines. If *listSepStr* is specified, then it is used to separate the list items. See **StringFromList** for details on string lists. If this is the only output you need, also use the /Q flag.

/M=*messageStr*

Specifies the prompt message in any Open File dialog. If /S is not specified, then *messageStr* will be used for both the Open File and Save File dialogs.

/O

Overwrites any existing destination file.

/P=*pathName*

Specifies the folder containing the source file or the folder into which the file is copied. *pathName* is the name of an existing symbolic path.

Both *srcFileStr* and *destFileOrFolderStr* must be either simple file or folder names, or paths relative to the folder specified by *pathName*.

/Q

Prevents printing results to an output file, text wave, History, or Clipboard. Use /Q to check for a match to *regExprStr* by testing the value of V_flag, V_value, S_value (/LIST), or W_Index (/INDX) without generating any other matching line output.

Note: When using /Q neither *destFileOrFolderStr* nor *destTextWaveName* may be specified.

/S=*saveMessageStr*

Specifies the prompt message in any Save File dialog.

/T=*termcharStr*

Specifies the terminator character.

termcharStr	**Terminator**
`/T=(num2char(13))`	Carriage return (CR, ASCII code 13).
`/T=(num2char(10))`	Linefeed (LF, ASCII code 10).
`/T=";"`	Semicolon.
`/T=""`	Null (ASCII code 0).

See **Line Termination** for the default behavior of the terminator character.

/Z[=*z*]

Prevents aborting of procedure execution when attempting to copy to a nonexistent file. Use /Z if you want to handle this case in your procedures rather than having execution abort.

z=0: Same as no /Z at all.

z=1: Copy file only if it exists. /Z alone is the same as /Z=1.

z=2: Copy file if it exists and display a dialog if it does not exist.

Line Termination

Line termination applies mostly to source and destination files. (The Clipboard and history area delimit lines with CR, and text waves have no line terminators.)

If /T is omitted, Grep will break file lines on any of the following: CR, LF, CRLF, LFCR. (Most Macintosh files use CR. Most Windows files use CRLF. Most UNIX files use LF. LFCR is an invalid terminator but some buggy programs generate files that use it.)

Grep reads whichever of these terminator(s) appear in the source file and use them to write lines to any output file.

The terminator(s) are removed before the line is matched against the regular expression.

For lines that match *regExprStr*, terminator(s) in the input file are transferred to the output file unless the output is the Clipboard or history area, in which case the output terminator is always only CR (like **LoadWave**). This means you can transparently handle files that use CR, LF, CRLF, or LFCR as the terminator, and omitting /T will be suitable for most cases.

If you use the /T flag, then Grep will terminate line-from-file reads on the specified character only and will output the specified character into any output file.

"Lines" in One-dimensional Text Waves

Grep considers each row of *srcTextWaveName* or *destTextWaveName* to be a "line" of input or output.

When the destination is a file, the Clipboard, or the History area, Grep copies all of the text in a matching row of *srcTextWaveName* to the file and terminates the line. See **Line Termination** for the rules on line terminators.

When the destination is a *destTextWaveName*, Grep simply copies all the text in a matching row to a row in *destTextWaveName*, without adding or omitting any terminators.

"Lines" and Columns in Two-Dimensional Text Waves

Grep by default *matches* against only the first column (column 0) of each row of *srcTextWaveName*. You can use the /GCOL=*grepCol* flag to specify a different column to match against. Use /GCOL=-1 to match against any column of *srcTextWaveName*.

When the source is a text wave and the destination is a file, the Clipboard, or the History area, Grep by default *copies* only the first column (column 0) to the destination.

Use the /DCOL={*colNum1, delimStr1, colNum2, delimStr2,...colNumN*} to print multiple columns (in any order) with delimiters after each column (the last column number need not be followed by a delimiter string). The output line is terminated with CR or *termcharStr* as described in **Line Termination**.

When both the source and destination are text waves and append (/A) is *not* specified, the destination text wave is redimensioned to have the same number of columns as the source text wave, and all columns of matching rows of *srcTextWaveName* are copied to *destTextWaveName*.

When both the source and destination are text waves and append /A is specified, then the number of columns in *destTextWaveName* is left unchanged, and each column of *srcTextWaveName* is copied to the corresponding column of *destTextWaveName*.

If the destination is a text wave and the source is a file or the Clipboard, each line (without the terminator) is copied to the first column of the destination text wave, or use /DCOL={*destColNum*} to put the text into a different column.

Output Variables

The Grep operation returns information in the following variables. When running in a user-defined function these are created as local variables. Otherwise they are created as global variables in the current data folder.

V_flag	0:	Output successfully generated.
	-1:	User cancelled either the Open File or Save File dialogs.
	Other:	An error occurred, such as the specified file does not exist.
V_value	The number of input lines that matched the regular expression.	
V_startParagraph	Zero-based line number into the file or Clipboard (or the row number of a source text wave) where the first regular expression was matched. Also see the **/INDX** flag.	

S_fileName	Full path to the source file, the source text wave, or "Clipboard". If an error occurred or if the user cancelled, it is an empty string.
S_path	Full path to the destination file or destination text wave.
	"Clipboard": If *destFileOrFolderStr* was the Clipboard.
	"History": If the output was printed to the History window.
	"": If an error occurred, if the user cancelled, or if /Q was specified.
S_value	Contains matching lines as a string list only if /LIST is specified.

Regular Expressions

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the "subject".

In the case of Grep, the "subject" is each line of the source file or Clipboard, or each row in the source text wave.

The regular expression syntax supported by Grep, **GrepList**, and **GrepString** is based on the "Perl-Compatible Regular Expression" (PCRE) library.

The syntax is similar to regular expressions supported by various UNIX and POSIX `egrep(1)` commands. See **Regular Expressions** on page IV-152 for more details. Igor's implementation does not support the Unicode (UTF-8) portion of PCRE.

As a trivial example, the pattern "Fred" as specified here:

```
Grep/P=myPath/E="Fred" "afile.txt" as "FredFile.txt"
```

matches lines that contain the string "Fred" anywhere on the line.

Character matching is *case-sensitive* by default, similar to strsearch. Prepend the Perl 5 modifier `(?i)` to match upper and lower-case versions of "Fred":

```
// Copy lines that contain "Fred", "fred", "FRED", "fREd", etc
Grep/P=myPath/E="(?i)fred" "afile.txt" as "AnyFredFile.txt"
```

To copy lines that do *not* match the regular expression, set the /E flag's reverse parameter:

```
// Copy lines that do NOT contain "Fred", "fred", "fREd", etc.
Grep/P=myPath/E={"(?i)fred",1} "afile.txt" as "NotFredFile.txt"
```

Note: Igor doesn't use the opening and closing regular expression delimiters that UNIX grep or Perl use: they would have used `"/Fred/"` and `"/(?i)fred/"`.

Regular expressions in Igor support the expected metacharacters and character classes that make the whole grep paradigm so useful. For example:

```
// Copy lines that START with a space or tab character
Grep/P=myPath/E="^[ \\t]" "afile.txt" as "LeadingTabsFile.txt"
```

For a complete description of regular expressions, see **Regular Expressions** on page IV-152, especially for a description of the many uses of the regular expression backslash character (see **Backslash in Regular Expressions** on page IV-155).

Note: Because Igor Pro also has special uses for backslash (see **Escape Characters in Strings** on page IV-13), you must double the number of backslashes you would normally use for a Perl or grep pattern. Each pair of backslashes identifies a single backslash for the Grep command.

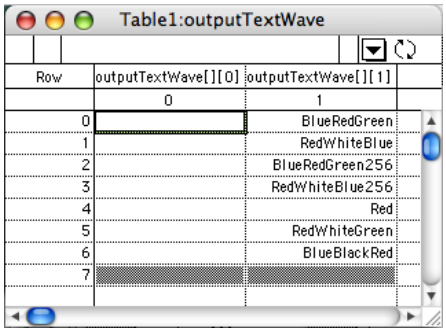
For example, to copy lines that contain "\z", the Perl pattern would be `\\z`, but the equivalent Grep expression would be `/E="\\\\z"`.

See **Backslash in Regular Expressions** on page IV-155 for a more complete description of backslash behavior in Igor Pro.

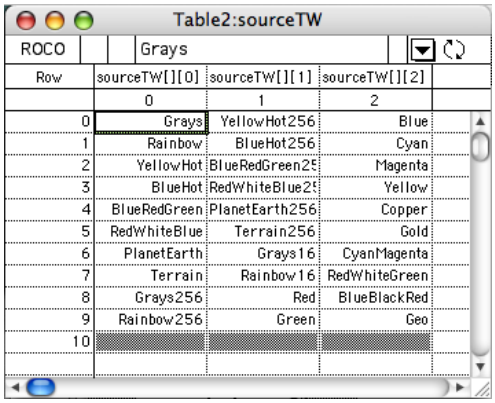
Examples

```
// Copy lines in afile.txt containing "Fred" (case sensitive)
// to an output file named "AnyFredFile.txt" in the same directory.
Grep/P=myPath/E="Fred" "afile.txt" as "AnyFredFile.txt"

// Copy lines in afile.txt containing "Fred" and "Wilma" (case-insensitive)
// to a text wave (which must exist andis overwritten):
Make/O/N=0/T outputTextWave
Grep/P=myPath/E="(?i)fred"/E="(?i)wilma" "afile.txt" as outputTextWave

// Print lines in afile.txt containing "Fred" and "Wilma" (case-insensitive)
// to the history area
Make/O/N=0/T outputTextWave
Grep/P=myPath/E="(?i)fred"/E="(?i)wilma" "afile.txt"
```

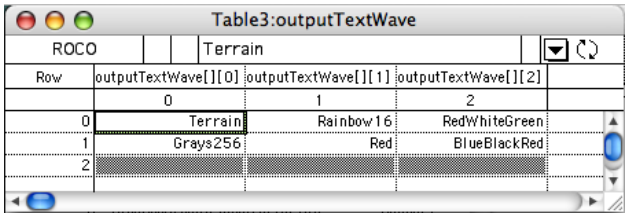
```
// Test whether afile.txt contains the word "boondoggle", and if so,
// on which line the first occurence was found, WITHOUT creating any output.
//
// Note: the \\b sequences limit matches to a word boundary before and after
// "boondoggle", so "boondoggles" and "aboondoggle" won't match.
//
Grep/P=myPath/Q/E="(?i)\\bBoondoggle\\b" "afile.txt"
if( V_value )                       // at least one instance was found
   Print "First instance of \"boondoggle\" was found on line", V_startParagraph
endif

// Create in S_value a string list of the lines as \r - separated list items:
Grep/P=myPath/LIST="\r"/Q/E="(?i)\\bBoondoggle\\b" "afile.txt"
if( V_Value )                       // some were found
   Print S_value
endif

// Create in W_index a list of the 0-based line numbers where "boondoggle"
// or "boondoggles", etc was found in afile.txt.
Grep/P=myPath/INDX/Q/E="(?i)boondoggle" "afile.txt"
if( V_flag )          // grep succeeded, perhaps none were found; let's see where
   WAVE W_Index       // needed if in a function
   Edit W_Index       // show line numbers in a table.
endif

// (Create a string list and text wave for the following examples.)
String list= CTabList()             // "Grays;Rainbow;YellowHot;..."
Variable items= ItemsInList(list)
Make/O/T/N=(items) textWave= StringFromList(p,list)

// Copy rows of textWave that contain "Red" (case sensitive)
// to the Clipboard as carriage-return separated lines.
Grep/E="Red" textWave as "Clipboard"

// Copy lines of the Clipboard that do NOT contain "Blue"
// (case in-sensitve) back to the Clipboard, overwriting what was there:
Grep/E={"(?i)blue",1} "Clipboard" as "Clipboard"

// Format matching text wave row to the history area
Grep/E=("Red")/DCOL={"prefix text --- ", 0, " --- suffix text"} textWave

// Printed output:
   prefix text --- BlueRedGreen --- suffix text
   prefix text --- RedWhiteBlue --- suffix text
   prefix text --- BlueRedGreen256 --- suffix text
   prefix text --- RedWhiteBlue256 --- suffix text
   prefix text --- Red --- suffix text
   prefix text --- RedWhiteGreen --- suffix text
   prefix text --- BlueBlackRed --- suffix text

// Create a 2-column text wave whose column 1 (the second column)
// contains the matching text from the Clipboard
Make/O/N=(0,2)/T outputTextWave
// Grep with /A to preserve 2 columns of outputTextWave
Grep/A/E="Red"/GCOL=1/DCOL={1} "Clipboard" as outputTextWave
```




```
// Examples with two-dimensional source text waves
Make/O/T/N=(10, 3) sourceTW= StringFromList(p+10*q,list)
Edit sourceTW
```




```
// Copy rows of textWave that contain "Red" in column 2 to outputTextWave.
Make/O/N=0/T outputTextWave
Grep/E="Red"/GCOL=2 sourceTW as outputTextWave
Edit outputTextWave
```




```
// Format matching text wave columns to the history area.
// Match lines that contain "Red" in any column of sourceTW:
Grep/E=("Red")/GCOL=-1/DCOL={0,", ",1,", ",2} sourceTW

// Printed output:
    YellowHot, BlueRedGreen256, Magenta
    BlueHot, RedWhiteBlue256, Yellow
    BlueRedGreen, PlanetEarth256, Copper
    RedWhiteBlue, Terrain256, Gold
    Terrain, Rainbow16, RedWhiteGreen
    Grays256, Red, BlueBlackRed
```

References

The regular expression syntax supported by Grep, **GrepString**, and **GrepList** is based on the *PCRE — Perl-Compatible Regular Expression Library* by Philip Hazel, University of Cambridge, Cambridge, England. The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5.

Visit <`http://pcre.org/`> for more information about the PCRE library, and <`http://www.perldoc.com/`> for more about Perl regular expressions. The description of regular expressions above is taken from the PCRE documentation.

A good book on regular expressions is: Friedl, Jeffrey E. F., *Mastering Regular Expressions*, 2nd ed., 492 pp., O'Reilly Media, 2002.

See Also

Regular Expressions on page IV-152 and **Symbolic Paths** on page II-34.

SplitString, **CopyFile**, **PutScrapText**, **LoadWave** operations. The **GrepString**, **GrepList**, **stringmatch**, and **cmpstr** functions.

GrepList

`GrepList(`*`listStr`*`, `*`regExprStr`*` [,`*`reverse`*` [, `*`listSepStr`*`]])`

The GrepList function returns each list item in *listStr* that matches the regular expression *regExprStr*.

ListStr should contain items separated by the *listSepStr* character, such as in "abc;def;".

regExprStr is a regular expression such as is used by the UNIX `grep(1)` command. It is much more powerful than the wildcard syntax used for **ListMatch**. See **Regular Expressions** on page IV-152 for *regExprStr* details.

reverse is optional. If missing, it is taken to be 0. If *reverse* is nonzero then the sense of the match is reversed. For example, if *regExprStr* is `"^abc"` and *reverse* is `1`, then all list items that do not start with "abc" are returned.

listSepStr is optional; the default is `";"`. In order to specify *listSepStr*, you must precede it with reverse.

Examples

To list ColorTables containing "Red", "red", or "RED" (etc.):

```
Print GrepList(CTabList(),"(?i)red")        // case-insensitive matching
```

To list window recreation commands starting with "\tCursor":

```
Print GrepList(WinRecreation("Graph0", 0), "^\tCursor", 0 , "\r")
```

See Also

Regular Expressions on page IV-152.

ListMatch, **StringFromList**, and **WhichListItem** functions and the **Grep** operation.

GrepString

GrepString(*string*, *regExprStr*)

The GrepString function tests *string* for a match to the regular expression *regExprStr*. Returns 1 to indicate a match, or 0 for no match.

Details

regExprStr is a regular expression such as is used by the UNIX grep(1) command. It is much more powerful than the wildcard syntax used for **stringmatch**. See **Regular Expressions** on page IV-152 for *regExprStr* details.

Character matching is case-sensitive by default, similar to **strsearch**. Prepend the Perl 5 modifier "(?i)" to match upper and lower-case text

Examples

Test for truth that the string contains at least one digit:

```
if( GrepString(str,"[0-9]+") )
```

Test for truth that the string contains at least one "abc", "Abc", "ABC", etc.:

```
if( GrepString(str,"(?i)abc") )             // case-insensitive test
```

See Also

Regular Expressions on page IV-152.

The **stringmatch**, **cmpstr**, **strsearch**, **ListMatch**, and **ReplaceString** functions and the **SplitString** and **sscanf** operations.

GridStyle

GridStyle

GridStyle is a procedure subtype keyword that puts the name of the procedure in the Grid->Style Function submenu of the mover pop-up menu in the drawing tool palette. You can have Igor automatically create a grid style function for you by choosing Save Style Function from that submenu.

GroupBox

GroupBox [**/Z**] ***ctrlName*** [***keyword* = *value*** [**, *keyword* = *value*** …]]

The GroupBox operation creates a box to surround and group related controls.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the GroupBox control to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **DefaultGUIControls Default Fonts and Sizes** for how enclosed controls are affected by native groupbox appearance.

See **Button** for more appearance details.

disable=*d* Sets user editability of the control.

d=0: Normal.

d=1: Hide.

d=2: Draw in gray state.

fColor=(*r*,*g*,*b*) Sets color of the title text. *r*, *g*, and *b* are integers from 0 to 65535.

font="*fontName*" Sets font used for the box title, e.g., `font="Helvetica"`.

frame=*f* Sets frame mode. If 1 (default), the frame has a 3D look. If 0, then a simple gray line is used. Generally, you should not use frame=0 with a title if you want to be in accordance with human interface guidelines.

fsize=*s* Sets font size for box title.

fstyle=fs Sets the font style of the title text. *fs* is a binary coded number with each bit controlling one aspect of the font style for the tick mark labels as follows:

bit 0: Bold.

bit 1: Italic.

bit 2: Underline.

bit 3: Outline (*Macintosh only*).

bit 4: Shadow (*Macintosh only*).

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

labelBack=(*r*,*g*,*b*) or 0 Sets fill color for the interior. *r*, *g*, and *b* are integers from 0 to 65535. If not set, then interior is transparent. Note that if a fill color is used, draw objects can not be used because they will be covered up. Also, you will have to make sure the GroupBox is drawn before any interior controls.

pos={*left*,*top*} Sets the postion of the box in pixels.

pos+={*dx*,*dy*} Offsets the position of the box in pixels.

size={*width*,*height*} Sets box size in pixels.

userdata(*UDName*)=*UDStr*

Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

title=*titleStr* Sets title to *titleStr*. Use `""` for no title.

win=*winName* Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

If no title is given and the width is less than 11 or height is specified as less than 6, then a vertical or horizontal separator line will be drawn rather than a box.

Note: Like TabControls, you need to click near the top of a GroupBox to select it.

See Also

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

GuideInfo

`GuideInfo(winNameStr, guideNameStr)`

The GuideInfo function returns a string containing a semicolon-separated list of information about the named guide line in the named host window or subwindow.

Parameters

winNameStr can be `""` to refer to the top host window.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

guideNameStr is the name of the guide line for which you want information.

Details

The returned string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon. The keywords are as follows:

Keyword	Information Following Keyword
NAME	Name of the guide.
WIN	Name of the window or subwindow containing the guide.
TYPE	The value associated with this keyword is either *User* or *Builtin*. A *User* type denotes a guide created by the DefineGuide operation, equivalent to dragging a new guide from an existing one.
HORIZONTAL	Either 0 for a vertical guide, or 1 for a horizontal guide.
POSITION	The position of the guide. This is the actual position relative to the left or bottom edge of the window, not the relative position specified to DefineGuide.

The following keywords will be present only for user-defined guides:

Keyword	Information Following Keyword
GUIDE1	The guide is positioned relative to GUIDE1.
GUIDE2	In some cases, the guide is positioned at a fractional position between GUIDE1 and GUIDE2. If the guide does not use GUIDE2, the value will be `""`.
RELPOSITION	The position relative to GUIDE1 (and GUIDE2 if applicable). This is the same as the *val* parameter in DefineGuide. May be a number of pixels if only GUIDE1 is used, or a fractional value if both GUIDE1 and GUIDE2 are used.

See Also

The **GuideNameList**,s **StringByKey** and **NumberByKey** functions; the **DefineGuide** operation.

GuideNameList

`GuideNameList(winNameStr, optionsStr)`

The GuideNameList function returns a string containing a semicolon-separated list of guide names from the named host window or subwindow.

Parameters

winNameStr can be "" to refer to the top host window.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

optionsStr is used to further qualify the list of guides. It is a string containing keyword-value pairs separated by commas. Use "" to list all guides. Available options are:

TYPE:type
: type = BuiltIn: List only built-in guides.
: type = User: List only user-defined guides, those created by the DefineGuide operation or by manually dragging a new guide from an existing one.

HORIZONTAL:h h = 0: List only non-horizontal (that is, vertical) guides.
h = 1: List only horizontal guides.

Example

```
String list = GuideNameList("Graph0", "TYPE:Builtin,HORIZONTAL:1")
```

See Also

The **DefineGuide** operation and the **GuideInfo** function.

Hanning

Hanning *waveName* [, *waveName*]…

Note: The **WindowFunction** operation has replaced the Hanning operation.

The Hanning operation multiplies the named waves by a Hanning window (which is a raised cosine function).

You can use Hanning in preparation for performing an FFT on a wave if the wave is not an integral number of cycles long.

The Hanning operation is not multidimensional aware. See Chapter II-6, **Multidimensional Waves**, particularly **Analysis on Multidimensional Waves** on page II-110 for details.

See Also

The **WindowFunction** operation implements the Hanning window as well as other forms such as Hamming, Parzen, and Bartlet (triangle).

The **ImageWindow** operation for windowing of images.

Hash

Hash(*inputStr, method*)

The Hash function returns a message digest string for *inputStr*. The length of the resulting hash string is fixed for each algorithm.

Parameters

inputStr is a string of arbitrary length.

method is 1 to use Secure Hash Algorithm-256.

See Also

For more about the SHA-256 algorithm see: <http://en.wikipedia.org/wiki/SHA-1>.

hcsr

hcsr(*cursorName* [, *graphNameStr*])

The hcsr function returns the horizontal coordinate of the named cursor (A through J) in the coordinate system of the top (or named) graph's X axis.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

The X axis used is the one that controls the trace on which the cursor is placed.

Examples

```
Variable xAxisValueAtCursorA = hcsr(A)      // not hcsr("A")
String str="A"
Variable xA= hcsr($str,"Graph0")            // $str is a name, too
```

See Also

The **Cursor** operation and the **CsrInfo**, **pcsr**, **qcsr**, **vcsr**, **xcsr**, and **zcsr** functions.

hermite

hermite(*n*, *x*)

The hermite function returns the Hermite polynomial of order *n*:

$$H_n(x) = (-1)^n \exp(x^2)\frac{d^n}{dx^n}\exp(-x^2).$$

The first few polynomials are:

$$1$$
$$2x$$
$$4x^2 - 2$$
$$8x^3 - 12x$$

.

See Also

The **hermiteGauss** function.

hermiteGauss

hermiteGauss(*n*, *x*)

The hermiteGauss function returns the normalized Hermite polynomial of order *n*:

$$H_n(x) = \frac{1}{\sqrt{\sqrt{\pi}2^n n!}}(-1)^n \exp(x^2)\frac{d^n}{dx^n}\exp(-x^2).$$

Here the normalization was chosen such that $\int_{-\infty}^{\infty} H_n(x)H_m(x)dx = \delta_{nm}$ where δ_{nm} is the Kronecker symbol.

See Also

The **hermite** function.

HideIgorMenus

HideIgorMenus [*MenuNameStr* [, *MenuNameStr*]…

The HideIgorMenus operation hides the named built-in menus or, if none are explicitly named, hides all built-in menus in the menu bar.

The effect of HideIgorMenus is lost when a new experiment is opened. The state of HideIgorMenus is saved with the experiment.

User-defined menus are not hidden by HideIgorMenus unless attached to built-in menus and the menu definition uses the hideable keyword.

Parameters

MenuNameStr The name of an Igor menu, like “File”, “Data”, or “Graph”.

Details

The optional menu names are in English and not abbreviated. This ensures that code developed for a localized version of Igor will run on all versions.

The built-in menus that can be shown or hidden (the Help menu can be hidden only on Windows) are those that appear in the menu bar:

File	Edit	Data	Analysis	Macros	Windows	Graph
Layout	Notebook	Panel	Procedure	Table	Misc	Help

Hiding a built-in menu to which a user-defined menu is attached results in a built-in menu with only the user-defined items. For example, if this menu definition attaches items to the built-in Graph menu:

```
Menu "Graph"
    "Do My Graph Thing", ThingFunction()
End
```

Calling `HideIgorMenus "Graph"` will still leave a Graph menu showing (when a Graph is the top-most target window) with only the user-defined menu(s) in it: in this example the one "Do My Graph Thing" item.

Hiding the Macros menu hides menus created from Macro definitions like:

```
Macro MyMacro()
   Print "Hello, world."
End
```

but does not hide normal user-defined "Macros" definitions like:

```
Menu "Macros"
   "Macro 1", MyMacro(1)
End
```

You can set user-defined menus to hide and show along with built-in menus by adding the optional hideable keyword to the menu definition:

```
Menu "Graph", hideable
   "Do My Graph Thing", ThingFunction()
End
```

Then `HideIgorMenus "Graph"` will hide those items, too. If all user-defined Graph menu definitions use the hideable keyword, then no Graph menu will appear in the menu bar.

Some WaveMetrics procedures use the `hideable` keyword so that only customer-defined menus remain when HideIgorMenus is executed.

See Also

Chapter IV-5, **User-Defined Menus**.

The **ShowIgorMenus**, **DoIgorMenu**, and **SetIgorMenuMode** operations.

HideInfo

HideInfo [**/W=*winName***]

The HideInfo operation removes the info box from a graph if it was previously shown by the **ShowInfo** operation.

Flags

/W=winName Hides the info box in the named window.

See Also

The **ShowInfo** operation.

HideProcedures

HideProcedures

The HideProcedures operation hides all procedure windows without closing or killing them.

See Also

The **DisplayProcedure** and **DoWindow** operations.

HideTools

HideTools [**/A/W=*winName***]

The HideTools operation hides the tool bar in the top graph or control panel if it was previously shown by the **ShowTools** operation.

Flags

/A Sizes the window automatically to make extra room for the tool palette. This preserves the proportion and size of the actual graph area.

/W=winName Hides the tool bar in the named window. This must be the first flag specified when used in a Proc or Macro or on the command line.

See Also

The **ShowTools** operation.

HilbertTransform

HilbertTransform [**/Z**][**/O**][**/DEST=*destWave*] *srcWave***

The HilbertTransform operation computes the Hilbert transformation of *srcWave*, which is a real or complex (single or double precision) wave of 1-3 dimensions. The result of the HilbertTransform is stored in *destWave*, or in the wave W_Hilbert (1D) or M_Hilbert in the current data folder.

Flags

/DEST=*destWave*	Creates a real wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-56 for details.
/O	Overwrites *srcWave* with the transform.
/Z	No error reporting.

Details

The Hilbert transform of a function $f(x)$ is defined by:

$$F(t) = \frac{1}{\pi}\int_{-\infty}^{\infty} \frac{f(x)}{t-x}dx .$$

Theoretically, the integral is evaluated as a Cauchy principal value. Computationally one can write the Hilbert transform as the convolution:

$$F(t) = \frac{1}{\pi t} \cdot f(t),$$

which by the convolution theorem of Fourier transforms, may be evaluated as the product of the transform of $f(x)$ with $-i$*sgn(x) where:

$$\mathrm{sgn}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases} .$$

Note that the Hilbert transform of a constant is zero. If you compute the Hilbert transform in more than one dimension and one of the dimensions does not vary (is a constant), the transform will be zero (or at least numerically close to zero).

There are various definitions for the extension of the Hilbert transform to more than one dimension. In two dimensions this operation computes the transform by multiplying the 2D Fourier transform of the input by the factor (-i)sgn(x)(-i)sgn(y) and then computing the inverse Fourier Transform. A similar procedure is used when the input is 3D.

Examples

Extract the instantaneous amplitude and frequency of a narrow-band signal.

```
Make/O/N=1000 w0,amp,phase
SetScale/I x 0,50,"", w0,amp,phase
w0 = exp(-x/10)*cos(2*pi*x)
HilbertTransform /DEST=w0h w0 // w0+i*w0h is the "analytic signal", i=cmplx(0,1)
amp = sqrt(w0^2 + w0h^2)   // extract the envelope
phase = atan2(-w0h,w0)     // extract the phase [SIGN CONVENTION?]
Unwrap 2*pi, phase         // eliminate the 2*pi phase jumps
Differentiate phase /D=freq   // would have less noise if fit to a line
                              // over interior points
freq /= 2*pi               // phase = 2*pi*freq*time
Display w0,amp   // original waveform and its envelope; note boundary effects
Display freq     // instantaneous frequency estimate, with boundary effects
```

See Also

The **FFT** operation.

References

Bracewell, R., *The Fourier Transform and Its Applications*, McGraw-Hill, 1965.

Histogram

Histogram [***flags***] ***srcWaveName, destWaveName***

The Histogram operation generates a histogram of the data in *srcWaveName* and puts the result in *destWaveName*.

Flags

/A — Accumulates the histogram result with the existing values in *destWaveName* instead of replacing the existing values with the result. Assumes /B=2 unless the /B flag is present.

Note: The result will be incorrect if you also use /P.

/B=mode

mode=1: Semiauto mode that sets the bin range based on the range of the Y values in *srcWaveName*. The number of bins is determined by the number of points in *destWaveName*.

mode=2: Uses the bin range and number of bins determined by the X scaling and number of points in *destWaveName*.

mode=3: Uses Sturges' method to determine optimal number of bins and to redimension *destWaveName* as necessary. By this method `numBins=1+log2(N)`, where N is the number of data points in *srcWaveName*. The bins will be distributed so that they include the minimum and maximum values.

mode=4: Uses a method due to Scott, which determines the optimal bin width as $\texttt{binWidth=3.49*s*N}^{-1/3}$, where N is the number of data points in *srcWaveName* and s is the standard deviation of the distribution. The bins will be distributed so that they include the minimum and maximum values.

/B={*binStart*,*binWidth*,*numBins*}

Sets the histogram bins from these parameters rather than from *destWaveName*. Changes the X scaling and length of *destWaveName*.

/C — Sets the X scaling so that X values are in the centers of the bins, which is required when you do a curve fit to the histogram output. Ordinarily, wave scaling of the output wave is set with X values at the left bin edges.

/CUM — Requests a cumulative histogram in which each bin is the sum of bins to the left. The last bin will contain the total number of input data points, or, with /P, 1.0.

When used with /A, the destination wave must be the result of a histogram created with /CUM.

Note that if you use a binning mode (/B flag) that sets a bin range that does not include the entire range of the input data, then the output will not count all of input points and the last bin will not contain the total number of input points. Input points whose values fall below the left edge of the first bin or above the right edge of the last bin will not be counted.

/N — Creates a wave (W_SqrtN) containing the square root of the number of counts in each bin. This is an appropriate wave to use as a weighting wave when doing a curve fit to the histogram results.

/P — Normalizes the histogram as a probability distribution function, and shifts wave scaling so that data correspond to the bin centers.

When using the results with **Integrate**, you must use /METH=0 or 1 because the trapezoidal approximation will give an error whose magnitude depends on the distribution function starting value.

/R=(*startX*,*endX*) — Specifies the range of X values of *srcWaveName* over which the histogram is to be computed.

/R=[*startP*,*endP*] — Specifies the range of points of *srcWaveName* over which the histogram is to be computed.

/W=*weightWave* — Creates a "weighted" histogram. In this case, instead of adding a single count to the appropriate bin, the corresponding value from *weightWave* is added to the bin. *weightWave* may be any number type, and it may be complex. If it is complex, then the destination wave will be complex.

Details

You must create a destination wave before doing the histogram. If you use /B={*binStart*, *binWidth*, *numBins*}, then the initial number of data points in the wave is unimportant since the Histogram operation changes the number of points.

Only one /B and only one /R flag is allowed.

If both /A and /B flags are missing, the bin range and number of bins is calculated as if /B=1 (auto-set) had been specified.

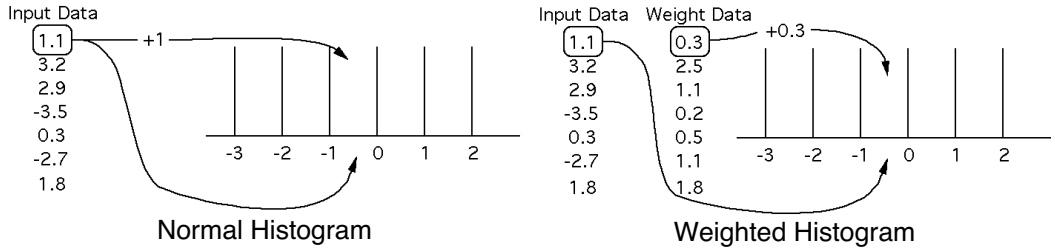
Typically, you will want to use /B={*binStart*,*binWidth*,*numBins*} for the first histogram, and /A for successive accumulations into the histogram.

The Histogram operation works on single precision floating point destination waves. If necessary, Histogram redimensions *destWaveName* to be single precision floating point. However, Histogram/A requires that *destWaveName* already be single precision floating point.

For a weighted histogram, the destination wave will be double-precision.

If you specify the range as `/R=(start)`, then the end of the range is taken as the end of *srcWaveName*.

In an ordinary histogram, input data is examined one data point at a time. The operation determines which bin a data value falls into and a single count is added to that bin. A weighted histogram works similarly, except that it adds to the bin a value from another wave in which each row corresponds to the same row in the input wave.



The Histogram operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details. In fact, the Histogram operation can be usefully applied to multidimensional waves, such as those that represent images. The /R flag will not work as expected, however.

Examples

```
// Create histogram of two sets of data.
Make/N=1000 data1=gnoise(1), data2=gnoise(1)
Make/N=1 histResult

// Sets bins, does histogram.
Histogram/B={-5,1,10} data1, histResult
Display histResult; ModifyGraph mode=5

// Accumulates into existing bins.
Histogram/A data2, histResult
```

See Also

Histograms on page III-126 and the **ImageHistogram** operation.

References

Sturges, H.A., The choice of a class-interval, *J. Amer. Statist. Assoc.*, *21*, 65-66, 1926.

Scott, D., On optimal and data-based histograms, *Biometrika*, *66*, 605-610, 1979.

hyperG0F1

hyperG0F1(*b*, *z*)

The hyperG0F1 function the confluent hypergeometric function

$${}_0F_1(b,z) = \sum_{i=0}^{\infty} \frac{z^i}{\Gamma(b+i)i!}$$ where $\Gamma(x)$ is the gamma function.

Note: The series evaluation may be computationally intensive. Exit the function by pressing Command-period (*Macintosh*) or Ctrl+Break (*Windows*).

See Also

The **hyperG1F1**, **hyperG2F1**, and **hyperGPFQ** functions.

References

The PFQ algorithm was developed by Warren F. Perger, Atul Bhalla, and Mark Nardin.

hyperG1F1

hyperG1F1(*a*, *b*, *z*)

The hyperG1F1 function returns the confluent hypergeometric function

$${}_1F_1(a,b,z) = \sum_{n=0}^{\infty} \frac{(a)_n z^n}{(b)_n n!}$$ where $(a)_n$ is the Pochhammer symbol $(a)_n = a(a+1)\ldots(a+n-1)$.

Note: The series evaluation may be computationally intensive. Exit the function by pressing Command-period (*Macintosh*) or Ctrl+Break (*Windows*).

See Also

The **hyperG0F1**, **hyperG2F1**, and **hyperGPFQ** functions.

References

The PFQ algorithm was developed by Warren F. Perger, Atul Bhalla, and Mark Nardin.

hyperG2F1

hyperG2F1(*a*, *b*, *c*, *z*)

The hyperG2F1 function returns the confluent hypergeometric function

$${}_2F_1(a,b,c,z) = \sum_{n=0}^{\infty} \frac{(a)_n (b)_n z^n}{(c)_n n!}$$ where $(a)_n$ is the Pochhammer symbol $(a)_n = a(a+1)\ldots(a+n-1)$.

Note: The series evaluation may be computationally intensive. Exit the function by pressing Command-period (*Macintosh*) or Ctrl+Break (*Windows*).

See Also

The **hyperG0F1**, **hyperG1F1**, and **hyperGPFQ** functions.

References

The PFQ algorithm was developed by Warren F. Perger, Atul Bhalla, and Mark Nardin.

hyperGNoise

hyperGNoise(*m*, *n*, *k*)

The hyperGNoise function returns a pseudo-random value from the hypergeometric distribution whose probability distribution function is

$$f(x;m,n,k) = \frac{\binom{n}{x}\binom{m-n}{k-x}}{\binom{m}{k}}$$

where *m* is the total number of items, *n* is the number of marked items, and *k* is the number of items in a sample.

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

SetRandomSeed, **StatsHyperGCDF**, and **StatsHyperGPDF**.

Chapter III-12, **Statistics** for a function and operation overview.

Noise Functions on page III-332.

hyperGPFQ

hyperGPFQ(*waveA, waveB, z*)

The hyperGPFQ function returns the generalized hypergeometric function

$${}_pF_q(\{a_1,\ldots a_p\},\{b_1,\ldots b_q\},z) = \sum_{n=0}^{\infty} \frac{(a_1)_n(a_2)_n\ldots(a_p)_n z^n}{(b_1)_n(b_2)_n\ldots(b_q)_n n!}$$

where $(a)_n$ is the Pochhammer symbol $(a)_n = a(a+1)\ldots(a+n-1)$.

Note: The series evaluation may be computationally intensive. Exit the function by pressing Command-period (*Macintosh*) or Ctrl+Break (*Windows*).

See Also

The **hyperG0F1**, **hyperG1F1**, and **hyperG2F1** functions.

References

The PFQ algorithm was developed by Warren F. Perger, Atul Bhalla, and Mark Nardin.

i

i

The i function returns the loop index of the inner most iterate loop in a macro. Not to be used in a function. iterate loops are archaic and should not be used.

if-elseif-endif

```
if ( <expression1> )
   <TRUE part 1>
elseif ( <expression2> )
   <TRUE part 2>
[…]
[else
   <FALSE part>]
endif
```

In an if-elseif-endif conditional statement, when an expression first evaluates as TRUE (nonzero), then only code corresponding to the TRUE part of that expression is executed, and then the conditional statement is exited. If all expressions evaluate as FALSE (zero) then *FALSE part* is executed when present. After executing code in any TRUE part or the FALSE part, execution will next continue with any code following the if-elseif-endif statement.

See Also

If-Elseif-Endif on page IV-32 for more usage details.

if-endif

```
if ( <expression> )
   <TRUE part>
[else
   <FALSE part>]
endif
```

An if-endif conditional statement evaluates *expression*. If *expression* is TRUE (nonzero) then the code in *TRUE part* is executed, or if FALSE (zero) then the optional *FALSE part* is executed.

See Also

If-Else-Endif on page IV-31 for more usage details.s

IFFT

IFFT [***flags***] ***srcWave***

The IFFT operation calculates the Inverse Discrete Fourier Transform of *srcWave* using a multidimensional fast prime factor decomposition algorithm. This operation is the inverse of the **FFT** operation.

Output Wave Name

For compatibility with earlier versions of Igor, if you use IFFT without /ROWS or /COLS, the operation overwrites *srcWave*.

If you use the /ROWS flag, IFFT uses the default output wave name M_RowFFT and if you use the /COLS flag, IFFT uses the default output wave name M_ColFFT.

We recommend that you use the /DEST flag to make the output wave explicit and to prevent overwriting *srcWave*.

Parameters

srcWave is a complex wave. The IFFT of *srcWave* is a either a real or complex wave, according to the length and flags.

Flags

/C Forces the result of the IFFT to be complex. Normally, the IFFT produces a real result unless certain special conditions are detected as described in **Details**.

/COLS Computes the 1D IFFT of 2D *srcWave* one column at a time, storing the results in the destination wave. You must specify a destination wave using the /DEST flag (no other flags are allowed). See the /ROWS flag and corresponding flags of the **FFT** operation.

/DEST=*destWave*

Specifies the output wave created by the IFFT operation.

It is an error to specify the same wave as both *srcWave* and *destWave*.

In a function, IFFT by default creates a real wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-56 for details.

/R Forces real output when, due to a power of 2 number of points, IFFT would otherwise automatically produce a complex result.

/ROWS Calculates the IFFT of only the first dimension of 2D *srcWave*. It computes the 1D FFT one row at a time. You must specify a destination wave using the /DEST flag (no other flags are allowed). See the /COLS flag and corresponding flags of the **FFT** operation.

/Z Will not rotate *srcWave* when computing the IDFT of a complex wave whose length is an integral power of 2.

This length indicates that the Inverse DFT result will also be a complex wave. When the result is complex, *and* the x scaling of *srcWave* is such that the first point is *not* x=0, it normally rotates *srcWave* by -N/2 points before performing the IFFT. This inverts the process of performing an FFT on a complex wave. However when /Z is specified, it does not perform this rotation.

Details

The data type of *srcWave* must be complex and must not be an integer type. You should be aware that an IFFT on a number of points that is prime can be slow.

By default, IFFT assumes you are performing an inverse transform on data that was originally real and therefore it produces a real result. However, for historical and compatibility reasons, IFFT detects the special conditions of a one-dimensional wave containing an integral power of 2 data points and automatically creates a complex result.

When the result is complex, the number of points (N) in the resulting wave will be of the same length. Otherwise the resulting wave will be real and of length (N-1)*2.

In either the complex or real case the X units of the output wave are changed to "s". The X scaling also is changed appropriately, cancelling out the adjustments made by the **FFT** operation. When the data is multidimensional, the same considerations apply to the additional dimensions. The scaling description and IDFT equation below pretend that the IFFT is not performed in-place. After computing the IFFT values, the X scaling of *waveOut* is changed as if Igor had executed these commands:

```
Variable points                    // time-domain points, N_timeDomain
if( waveIn was complex wave )
   points= numpnts(waveIn)
```

```
else                                    // waveIn was real wave
   points= (numpnts(waveIn) - 1) * 2
endif
Variable deltaT= 1 / (points*deltaX(waveIn))      // 1/(N_timeDomain dx)
SetScale/P waveOut 0,deltaT,"s"
```

The IDFT equation is:

$$waveOut[n] = \frac{1}{N} \cdot \sum_{k=0}^{N-1} waveIn[k] \cdot e^{-2\pi \cdot i \cdot k \cdot n/N}, \ where \ i = \sqrt{-1} \ .$$

See Also

The **FFT**, **DSPPeriodogram**, and **MatrixOp** operations.

IgorInfo

IgorInfo(*selector*)

The IgorInfo function returns information about the Igor application and the environment in which it is running.

Details

selector is a number from 0 to 5.

Always pass 0, 1, 2, 3, 4 or 5 as the input parameter. In future versions of Igor Pro, this parameter may request other kinds of information.

If *selector* is 0, IgorInfo returns a collection of assorted information. The result string contains five kinds of information. Each group is prefaced by a keyword and colon, and terminated with a semicolon. The keywords are IGORVERS, IGORKIND, FREEMEM, NSCREENS, and SCREEN*n* where *n* varies from 1 to the number of screens (monitors) currently attached to the computer and used for the desktop.

Keyword	Information Following Keyword For IgorInfo(0)
FREEMEM	The amount of free memory available to Igor. When running under virtual memory, such as on Windows, this is the amount of free virtual memory.
IGORKIND	The type of Igor application; "pro" means the full (nondemo) version of Igor Pro. "pro demo" is the demo version of Igor Pro. Currently there are no other values for this keyword.
IGORVERS	The version number of the Igor application. Also see IGORFILEVERSION returned by IgorInfo(3).
NSCREENS	Number of screens (monitors) currently attached to the computer and used for the desktop. In Igor Pro 4 this was erroneously " NSCREENS" (with a leading space).
SCREEN1	Description of the characteristics of screen 1. On Macintosh, the screen number corresponds to the number in the Monitors Control Panel; use the Identify button there to show the monitor (screen) numbers. SCREEN1 is not necessarily the monitor with the menu bar. On Windows, SCREEN1 is the main monitor (whose top left corner screen coordinate is 0, 0). Format of the SCREEN description is: SCREEN*n*:DEPTH=*bitsPerPixel*,RECT=*left*,*top*,*right*,*bottom*; *left*, *top*, *right*, and *bottom* are all in pixels.
SCREEN*n*	Description of the characteristics of the last screen. See NSCREENS, above.

If *selector* is 1, IgorInfo returns the name of the current Igor experiment.

If *selector* is 2, IgorInfo returns the name of the current platform: "Macintosh" or "Windows".

If *selector* is 3, IgorInfo returns a collection of more detailed information about the operating system, localization information, and the actual file version of the Igor executable. The keywords are OS, OSVERSION, LOCALE, and IGORFILEVERSION.

Keyword	Information Following Keyword For IgorInfo(3)
IGORFILEVERSION	The actual version number of the Igor application file.
	On the Macintosh, the version number is a floating point number with a possible suffix. Igor Pro 5.00, for example, returns "5.00". Igor Pro 5.02A returns "5.02A".
	As of Igor Pro 5.02, the Windows version format is a period-separated list of four numbers. Igor Pro 5.02 returns "5.0.2.0". A revision to Igor Pro 5.02 would be indicated in the last digit, such as "5.0.2.12".
	(For versions older than 5.02, the Windows version format is a floating point number similar to the Macintosh. For example, Igor 4.09A returns "4.091". We abandoned this representation because it limits each of the digits to 0-9.)
LOCALE	Country for which this version of Igor Pro is localized. "US" for most versions, "Japan" for the Japanese versions.
OS	On Mac OS X, this will be "Macintosh OS X".
	On Windows, this might be "Windows XP (Build 1234)". The actual build number and format of the text will vary with the operating system.
OSVERSION	Operating system number.
	On Macintosh, this might be "9.1.0", "10.1.2" etc.
	On Windows, this might be "5.1.2600".

If *selector* is 4, IgorInfo returns the name of the current processor architecture: "PowerPC" or "Intel".

If *selector* is 5, IgorInfo returns (as a string) the serial number of the program if it is registered or "_none_" if it isn't registered. Use **str2num** to store the result in a numeric variable. str2num will return NaN if the program isn't registered.

Examples

```
Print NumberByKey("NSCREENS", IgorInfo(0))     // number of active displays

Function RunningWindows()                      // returns 0 if Macintosh, 1 if Windows
   String platform= UpperStr(igorinfo(2))
   Variable pos= strsearch(platform,"WINDOWS",0)
   return pos >= 0
End
```

IgorVersion

#pragma IgorVersion = *versNum*

When a procedure file contains the directive, `#pragma IgorVersion=`*versNum*, an error will be generated if *versNum* is greater than the current Igor Pro version number. It prevents procedures that use new features added in later versions from running under older versions of Igor in which these features are missing. However, this version check is limited because it does not work with versions of Igor older than 4.0.

See Also

The **The IgorVersion Pragma** on page IV-42 and **#pragma**.

IgorVersion

The IgorVersion function returns version number of the Igor application as a floating point number. Igor 6.01 returns 6.01, as does Igor 6.01A.

Details

The IgorVersion function was introduced in Igor 6.1.

The returned value is identical to that returned by more cumbersome:

```
NumberByKey("IGORVERS", IgorInfo(0))
```

This older code is compatible with older versions of Igor.

Because floating point numbers are not precise, exact comparisons to floating point values often behave in unexpected ways. For example:

```
Variable result= 6 + 0.1
if( result == 6.1 )
    Print "result == 6.1"                // this is not printed!
else
    Print "difference = ", result - 6.1  // prints "difference =   -8.88178e-16"
endif
```

However, IgorVersion compensates for this so that the following will work as expected:

```
if (IgorVersion() == 6.1)
    Print "result == 6.1"// this is printed to the history area
endif
```

You can use IgorVersion in conditionally compile code expressions, which can be used to omit calls to new Igor features or to provide backwards compatibility code.

```
#if (IgorVersion() >= 6.1)
    [Code that compiles only on Igor 6.1 or later]
#else
    [Code that compiles only on earlier versions of Igor]
#endif
```

However, this will fail with older versions which precede the IgorVersion function. To work with versions older than 6.1 you must use this instead:

```
#if NumberByKey("IGORVERS", IgorInfo(0)) >= 6.1
    [Code that compiles only on Igor 6.1 or later]
#else
    [Code that compiles only on earlier versions of Igor]
#endif
```

If at all possible, it is better to require your users to use a later version of Igor rather than writing conditional code. Attempting this kind of backward-compatibility multiplies your testing requirements and the chances for bugs.

See Also

IgorInfo, **Conditional Compilation** on page IV-86, **The IgorVersion Pragma** on page IV-42

ilim

ilim

The ilim function returns the ending loop count for the inner most iterate loop Not to be used in a function. iterate loops are archaic and should not be used.

imag

imag(*z*)

The imag function returns the imaginary component of the complex number *z* as a real (not complex) number.

See Also

The **cmplx**, **conj**, **p2rect**, **r2polar**, and **real** functions.

ImageAnalyzeParticles

ImageAnalyzeParticles [***flags***] ***keyword imageMatrix***

The ImageAnalyzeParticles operation performs one of two particle analysis operations on a 2D or 3D source wave *imageMatrix*. The source image wave must be binary, i.e., an unsigned char format where the particles are designated by 0 and the background by 255 (the operation will produce erroneous results if your data uses the opposite designation). Note that all nonzero values in the source image will be considered part of the background. Grayscale images must be thresholded before invoking this operation (you may need to use the /I flag with the **ImageThreshold** operation).

Note: ImageAnalyzeParticles does not take into account wave scaling. All image metrics are in pixels and all pixels are assumed to be square.

Parameters

keyword is one of the following names:

mark — Creates a masking image for a single particle, which is specified by an internal (seed) pixel using the /L flag. The masking image is stored in the wave M_ParticleMarker, which is an unsigned char wave. All points in M_ParticleMarker are set to 64 (image operations on binary waves use the value 64 to designate the equivalent of NaN) except points in the particle which are set to the 0. This wave is designed to be used as an overlay on the original image (using the explicit=1 mode of ModifyImage). This keyword is superseded by the **ImageSeedFill** operation.

stats — Measures the particles in the image. Results of the measurements are reported for all particles whose area exceeds the *minArea* specified by the /A flag. The results of the measurements are:

V_NumParticles	Number of particles that exceed the *minArea* limit.
W_ImageObjArea	Area (in pixels) for each particle.
W_ImageObjPerimeter	Perimeter (in pixels) of each particle. The perimeter calculation involves estimates for 45-degree pixel edges resulting in noninteger values.
W_circularity	Ratio of the square of the perimeter to (4*π*objectArea). This value approaches 1 for a perfect circle.
W_rectangularity	Ratio of the area of the particle to the area of the inscribing (nonrotated) rectangle. This ratio is π/4 for a perfectly circular object and unity for a nonrotated rectangle.
W_SpotX and W_SpotY	Contain a single x, y point from each object. There is one entry per particle and the entries follow the same order as all other waves created by this operation. Each (x,y) point from these waves can used to define the position of a tag or annotation for a particle. Points can also be used as seed pixels for the associated *mark* method or for the ImageSeedFill operation.
W_xmin, W_xmax, W_ymin, W_ymax	Contain a single point for each particle defining an inscribing rectangular box with axes along the X and Y directions.

One of the following waves can be created depending on the /M specification. The waves are designed to be used as an overlay on the original image (using the explicit=1 mode of **ModifyImage**). **Note**: the additional time required to create these waves is negligible compared with the time it takes to generate the stats data.

M_ParticlePerimeter	Masking image of particle boundaries. It is an unsigned char wave that contains 0 values for the object boundaries and 64 (i.e., NaN) for all other points.
M_ParticleArea	Masking image of the area occupied by the particles. It is an unsigned char wave containing 0 values for the object boundaries and 64(=NaN) for all other points. It is also different from the input image in that particles smaller than the minimum size, specified by /A, are absent.
M_Particle	Image of both the area and the boundary of the particles. It is an unsigned char wave that contains the value 16 for object area, the value 18 for the object boundaries and the value 64(=NaN) for all other points.
M_rawMoments	Contains five columns. The first column is the raw sum of the x values for each particle, and the second column contains the sum of the y values. To obtain the average or "center" of a particle divide these values by the corresponding area. The third column contains the sum of x^2, the fourth column the sum of y^2, and the fifth column the sum of x*y. The entries of this wave are used in calculating a fit to an ellipse (using the /E flag).

When *imageMatrix* is a 3D wave, the different results are packed into a single 2D wave M_3DParticleInfo, which consists of one row and 11 columns for each particle. Columns are arranged in the following order: minRow, maxRow, minCol, maxCol, minLayer, maxLayer, xSeed, ySeed, zSeed, volume, and area. Use `Edit M_3DParticleInfo.ld` to display the results in a table with dimension labels describing the different columns.

Flags

/A=*minArea* Specifies a minimum area as a threshold that must be exceeded for a particle to be counted (e.g., use *minArea*=0 to find single pixel particles). The minimum area is measured in pixels; its default value is *minArea*=5.

Has no effect when used with the *mark* method.

/CIRC={*minCircularity*,*maxCircularity*}

Use this flag to filter the output so that only particles in the range of the specified circularity are counted.

/D=*dataWave* Specify a wave from which the minimum, maximum, and total particle intensity are sampled when used with the stats keyword. dataWave must be of the same dimensions as the input binary image imageMatrix. It can be of any real numeric type. Results are returned in the waves W_IntMax, W_IntMin, and W_IntAvg.

/E Calculates an ellipse that best fits each particle. The equivalent ellipse is calculated by first finding the moments of the particle (i.e., average x-value, average y-value, average x^2, average y^2, and average x*y), and then requiring that the area of the ellipse be equal to that of the particle. The resulting ellipses are saved in the wave M_Moments. When *imageMatrix* is a 2D wave, the results returned in M_Moments are the columns: the X-center of the ellipse, the Y-center of the ellipse, the major axis, the minor axis, and the angle (radians) that the major axis makes with the X-direction. When *imageMatrix* is a 3D wave, the results in M_Moments include the sum of the X, Y, and Z components as well as all second order permutations of their products. They are arranged in the order: sumX, sumY, sumZ, sumXX, sumYY, sumZZ, sumXY, sumXZ, and sumYZ.

/EBPC Use this flag to exclude from counting any particle that has one or more pixels on any boundary of the image.

/F Fills 2D particles having internal holes and adjusts their area measure for the removal of holes. Internal boundaries around the holes are also eliminated. When the boundary of the particle consists of thin elements that cannot be traversed as a single closed path which passes each boundary pixel only once, the particle will not be filled. Note that filling particles may increase execution time considerably and on some images it may require large amount of memory. It is likely that a more efficient approach would be to preprocess the binary image and remove holes using morphology operations. This flag is not supported when *imageMatrix* is a 3D wave.

/L= (*row*,*col*) Specifies a 2D particle location in connection with the mark method. (*row*, *col*) is a seed value corresponding to any pixel inside the particle. If the seed belongs to the particle boundary, the particle will not be filled. This flag is not supported when *imageMatrix* is a 3D wave.

/M=*markerVal* Use this flag with the stats mode for 2D images. See stats keyword for a full description of the following waves:

markerVal=0: No marker waves.

markerVal=1: M_ParticlePerimeter.

markerVal=2: M_ParticleArea.

markerVal=3: M_Particle.

This flag does not apply to 3D waves.

/MAXA=*maxArea* Specifies an upper limit of the area of an acceptable particle when used with the stats keyword. The area is measured in pixels and the default value of *maxArea* is the number of pixels in the image. In 3D the maximum value applies to the number of voxels.

/NSW Creates the marker wave (see /M flag) but not the particle statistics waves when used with the stats keyword. This should reduce execution time in images containing many particles.

/P=*plane* Specifies the plane when operating on a single layer of a 3D wave.

/Q Quiet flag, does not report the number of particles to the history window.

/R=*roiWave* Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u) that has the same number of rows and columns as *imageMatrix*. The ROI itself is defined by the entries or pixels in the *roiWave* with value of 0. Pixels outside the ROI may have any nonzero value. The ROI does not have to be contiguous. When *imageMatrix* is a 3D wave, *roiWave* can be either a 2D wave (matching the number of rows and columns in *imageMatrix*) or it can be a 3D wave that must have the same number of rows, columns

and layers as *imageMatrix*. When using a 2D *roiWave* with a 3D *imageMatrix* the ROI is understood to be defined by *roiWave* for each layer in the 3D wave.

See **ImageGenerateROIMask** for more information on creating 2D ROI waves.

/U — Saves the wave M_ParticleMarker as an 8-bit unsigned instead of the default 16-bit when used with the mark keyword.

/W — Creates boundary waves W_BoundaryX, W_BoundaryY, and W_BoundaryIndex for a 2D *imageMatrix* wave. W_BoundaryX and W_BoundaryY contain the pixels along the particle boundaries. The boundary of each particle ends with a NaN entry in both waves. Each entry in W_BoundaryIndex is the index to the start of a new particle in W_BoundaryX and W_BoundaryY, so that you can quickly locate the boundary of each particle.

When there are holes in particles, the entries in W_BoundaryX and W_BoundaryY start with the external boundary followed by all the internal boundaries for that particle. There are no index entries for internal boundaries.

This flag is not supported when *imageMatrix* is a 3D wave.

Details

Particle analysis is accomplished by first converting the data from its original format into a binary representation where the particle is designated by zero and the background by any nonzero value. The algorithm searches for the first pixel or voxel that belongs to a particle and then grows the particle from that seed while keeping count of the area, perimeter and count of pixels or voxels in the particle. If you use additional flags, the algorithm must compute additional quantities for each pixel or voxel belonging to the particle.

If your goal is to mask only the particle, a more efficient approach is to use the **ImageSeedFill** operation, which similarly follows the particle but does not spend processing time on computing unrelated particle properties. ImageSeedFill also has the additional advantage of not requiring that the input wave be binary, which will save time on performing the initial threshold and, in fact, may produce much better results with the adaptive/fuzzy features that are not available in ImageAnalyzeParticles.

Examples

Convert a grayscale image (blobs) into a proper binary input:

```
ImageThreshold/M=4/Q/I blobs
```

Get the statistics on the thresholded image of blobs and create an image mask output wave for the perimeter of the particles:

```
ImageAnalyzeParticles/M=1 stats M_ImageThresh
```

Display an image of the blobs with a red overlay of the perimeter image:

```
NewImage/F blobs; AppendImage M_ParticlePerimeter
ModifyImage M_ParticlePerimeter explicit=1, eval={0,65000,0,0}
```

See Also

The **ImageThreshold**, **ImageGenerateROIMask**, **ImageSeedFill**, and **ModifyImage** operations. For more usage details see **Particle Analysis** on page III-319.

ImageBlend

ImageBlend [**/A=*alpha* /W=*alphaWave***] ***srcWaveA, srcWaveB*** [**, *destWave***]

The ImageBlend operation takes two RGB images (3D waves) in *srcWaveA* and *srcWaveB* and computes the alpha blending so that

destWave = *srcWaveA* * (1 - *alpha*) + *srcWaveB* * *alpha*

for each color component. If *destWave* is not specified or does not already exist, the result is saved in the current data folder in the wave M_alphaBlend.

The source and destination waves must be of the same data types and the same dimensions. The *alphaWave*, if used, must be a single precision (SP) float wave and it must have the same number of rows and columns as the source waves.

Flags

/A=*alpha* — Specifies a single alpha value for the whole image

/W=*alphaWave* — Single precision wave that specifies an alpha value for each pixel.

ImageBoundaryToMask

ImageBoundaryToMask width=*w*, height=*h*, xwave=*xwavename*, ywave=*ywavename* [, scalingWave=*scalingWaveName*, [seedX=*xVal*, seedY=*yVal*]]

The ImageBoundaryToMask operation scan-converts a pair of XY waves into an ROI mask wave.

Parameters

height = *h* Specifies the mask height in pixels.

scalingWave = *scalingWaveName*
2D or 3D wave that provides scaling for the mask. If specified, the scaling of the first two dimensions of scalingWave are copied to M_ROIMask, and both the X and Y waves are assumed to describe pixels in the scaled domain.

seedX = *xVal* Specifies seed pixel location. The operation fills the region defined by the seed and the boundary with the value 1. Background pixels are set to zero. Requires seedY.

seedY = *yVal* Specifies seed pixel location. The operation fills the region defined by the seed and the boundary with the value 1. Background pixels are set to zero. Requires seedX.

width = *w* Specifies the mask width in pixels.

xwave = *xwavename* Name of X wave for mask region.

ywave = *ywavename* Name of Y wave for mask region.

Details

ImageBoundaryToMask generates an unsigned char 2D wave named M_ROIMask, of dimensions specified by width and height. The wave consists of a background pixels that are set to 0 and pixels representing the mask that are set to 1.

The x and y waves can be of any type. However, if the waves describe disjoint regions there must be at least one NaN entry in each wave corresponding to the discontinuity, which requires that you use either single or double precision waves. The values stored in the waves must correspond to zero-based integer pixel values.

If the x and y waves include a vertex that lies outside the mask rectangle, the offending vertex is moved to the boundary before the associated line segment is scan converted.

If you want to obtain a true ROI mask in which closed regions are filled, you can specify the seedX and seedY keywords. The ROI mask is set with zero outside the boundary of the domain and 1 everywhere inside the domain.

Examples

```
Make/O/N=(100,200) src=gnoise(5)            // create a test image
SetScale/P x 500,1,"", src;DelayUpdate      // give it some funny scaling
SetScale/P y 600,1,"", src
Display; AppendImage src
Make/O/N=201 xxx,yyy                 // create boundary waves
xxx=550+25*sin(p*pi/100)             // representing a close ellipse
yyy=700+35*cos(p*pi/100)
AppendToGraph yyy vs xxx
```

Now create a mask from the ellipse and scale it so that it will be appropriate for src:

```
ImageBoundaryToMask ywave=yyy,xwave=xxx,width=100,height=200,scalingwave=src
```

To generate an ROI masked filled with 1 in a region defined by a seed value and the boundary curves:

```
ImageBoundaryToMask
ywave=yyy,xwave=xxx,width=100,height=200,scalingwave=src,seedx=550,seedy=700
```

See Also

The **ImageAnalyzeParticles** and **ImageSeedFill** operations. For another example see **Converting Boundary to a Mask** on page III-322.

ImageEdgeDetection

ImageEdgeDetection [*flags*] *Method imageMatrix*

The ImageEdgeDetection operation performs one of several standard image edge detection operations on the source wave *imageMatrix*. Unless the /O flag is specified, the resulting image is saved in the wave M_ImageEdges. The edge detection methods produce binary images on output (the background is set to 0 and the edges to 255). This is due, in most cases to a thresholding performed in the final stage. Except for

the case of marr and shen detectors, you can use the /M flag to specify a method for automatic thresholding (see **ImageThreshold** /M flag).

Parameters

Method selects type of edge detection. *Method* is one of the following names:

canny	Canny edge detector uses smoothing before edge detection and thresholding. You can optionally specify the threshold using the /T flag and the smoothing factor using /S.
frei	Calculates the Frei-Chen edge operator (see Pratt p. 503) using only the row and column filters.
kirsch	Kirsch edge detector (see Pratt p. 509). Performs convolution with 8 masks calculating gradients.
marr	Marr-Hildreth edge detector. Performs two convolutions with Laplacian of Gaussian and then detects zero crossings. Use the /S flag to define the width of the convolution kernel.
prewitt	Calculates the Prewitt compass gradient filters. Returns the result for the largest filter response.
roberts	Calculates the square root of the magnitude squared of the convolution with the Robert's row and column edge detectors.
shen	Shen-Castan optimized edge detector. Supposed to be effective in the presence of noise. The flags that modify this operation are: /F for the threshold ratio (0.9 by default), /S for smoothness factor (0.9 by default), /W for window width (default is 10), /H for thinning factor which by default is 1.
sobel	Sobel edge detector using convolutions with row and column edge gradient masks (see Pratt p. 501).

Flags

/F=*fraction*	Determines the threshold value for the shen algorithm by starting from the histogram of the image and choosing a threshold such that *fraction* specifies the portion of the image pixels whose values are below the threshold. Valid values are in the interval (0 < *fraction* < 1).
/H=*thinning*	Thins edges when used with shen edge detector. By default the thinning value is 1. Higher values produce thinner edges.
/I	Inverts the output, i.e., sets the edges to 255 and the background to 0.
/M=*threshMethod*	See the **ImageThreshold** automatic methods for obtaining a threshold value. Methods 1, 2, 4 and 5 are supported in this operation. If you use *threshMethod* = -1, threshold is not applied. If you want to apply your own thresholding algorithm, use /M=6 to bypass the thresholding completely. The wave M_RawCanny contains the result regardless of any other flags you may have used.
/N	Sets the background level to 64 (i.e., NaN)
/O	Overwrites the source image with the output image.
/R=*roiSpec*	Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u). The ROI wave must have the same number of rows and columns as the image wave. The ROI itself is defined by entries/pixels whose values are 0. Pixels outside the ROI can be any nonzero value. The ROI does not have to be contiguous and can be any arbitrary shape. See **ImageGenerateROIMask** for more information on creating ROI waves. In general, the *roiSpec* has the form {*roiWaveName*, *roiFlag*}, where *roiFlag* can take the following values: *roiFlag*=0: Set pixels outside the ROI to 0. *roiFlag*=1: Set pixels outside the ROI as in original image. *roiFlag*=2: Set pixels outside the ROI to NaN (=64). By default *roiFlag* is set to 1 and it is then possible to use the /R flag using the abbreviated form /R=*roiWave*.
/S= *smoothVal*	Specifies the standard deviation or the width of the smoothing filter. By default the operation uses 1. Larger values require longer computation time. In the shen operation the default value is 0.9 and the valid range is (0 < *smoothVal*< 1).
/T=*thresh*	Sets a manual threshold for any method above that uses a single threshold. This is faster than using /M.
/W=*width*	Specifies window width when used in the shen operation. By default width is set to 10 and it is clipped to 49.

See Also

The **ImageGenerateROIMask** operation for creating ROIs and the **ImageThreshold** operation.

Edge Detectors on page III-309 for a number of examples.

References

Pratt, William K., *Digital Image Processing*, John Wiley, New York, 1991.

ImageFileInfo

ImageFileInfo [**/P=*pathName***] ***fileNameStr***

The ImageFileInfo operation supplies information about an image file without having to open the file and load the data into Igor.

ImageFileInfo works with the following file types: PICT, TIFF, GIF, JPEG, PNG, Targa, QuickTime, and BMP.

ImageFileInfo requires QuickTime. The operation will fail (V_flag=0) if you request information for a file format that is not supported by QuickTime.

Parameters

fileNameStr specifies the image file for which information is needed.

The file of interest is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

If you want to force a dialog to select the file, omit the *fileNameStr* parameter.

Flags

/P=*pathName* Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

Variables

ImageFileInfo returns information in the following variables:

S_path	File system path to the selected file.
V_BPP	Number of bits per pixel. A value of 40 means 8-bit grayscale; all other values are bits per pixel.
V_flag	1 if operation was successful, 0 otherwise.
V_frameCount	Number of movie frames.
V_numCols	Height of the image.
V_numImages	Number of images in the file. Only supports TIFF files that contain multiple images.
V_numRows	Width of the image.
V_quality	Image compression quality in hexadecimal: codecLosslessQuality = 0x00000400 codecMaxQuality = 0x000003FF codecMinQuality = 0x00000000 codecLowQuality = 0x00000100 codecNormalQuality = 0x00000200 codecHighQuality = 0x00000300

ImageFilter

ImageFilter [***flags***] ***Method dataMatrix***

The ImageFilter operation is identical to **MatrixFilter**, accepting the same parameters and flags, with the exception of the additional features described below.

Parameters

Method selects the filter type. *Method* is one of the following names:

avg3d *nxnxn* average filter for 3D waves.

gauss3d	*n*x*n*x*n* gaussian filter for 3D waves.
hybridmedian	Implements ranking pixel values between two groups of pixels in a 5x5 neighborhood. The first group includes horizontal and vertical lines through the center, the second group includes diagonal lines through the center, and both groups include the center pixel itself. The resulting median value is the ranked median of both groups and the center pixel.
max3d	*n*x*n*x*n* maximum rank filter for 3D waves.
median3d	*n*x*n*x*n* median filter for 3D waves where *n* must be of the form 3^r (integer *r*), e.g., 3x3x3, 9x9x9 etc. The filter does not change the value of the voxel it is centered on if any of the filter voxels lies outside the domain of the data.
min3d	*n*x*n*x*n* minimum rank filter for 3D waves.
point3d	*n*x*n*x*n* point finding filter using normalized `(`n^3`-1)*`*center-outer* for 3D waves.

Flags

/N=*n*	Specifies the filter size. By default *n* =3. In most situations it will be useful to set *n* to an odd number in order to preserve the symmetry in the filters.
/O	Overwrites the source image with the output image. Used only with the hybridmedian filter, which does not automatically overwrite the source wave.

Details

You can operate on 3D waves using the 3D filters listed above. These filters are extensions of the 2D filters available under MatrixFilter. The avg3d, gauss3d, and point3d filters are implemented by a 3D convolution that uses an averaging compensation at the edges.

This operation does not support complex waves.

See Also

MatrixFilter for descriptions of the other available parameters and flags.

MatrixConvolve for information about convolving your own 3D kernels.

References

Russ, J., *Image Processing Handbook*, CRC Press, 1998.

ImageFocus

ImageFocus [***flags***] ***stackWave***

The ImageFocus operation creates in focus image(s) from a stack of images that contain in and out of focus regions. It computes the variance in a small neighborhood around each pixel and then takes the pixel value from the plane in which the highest variance is found.

Flags

/ED=*edepth*	Sets the effective depth in planes. For example, an effective depth of one means that it computes the best focus for each plane using a stack of three planes, which includes the current plane and any one adjacent plane above and below it. Does not affect the default method (/METH=0).
/METH=*method*	Specifies the calculation method. *method*=0: Computes a single plane output for the stack (default). *method*=1: Computes the best image for each plane using /ED.
/Q	Quiet mode; no output to history window.
/Z	No error reporting.

See Also

Chapter III-11, **Image Processing** contains links to and descriptions of other image operations.

ImageGenerateROIMask

ImageGenerateROIMask [**/W=*winName*/E=*e*/I=*i***] ***imageInstance***

The ImageGenerateROIMask operation creates a Region Of Interest (ROI) mask for use with other ImageXXX commands. It assumes the top (or /W specified) graph contains an image and that the user has drawn shapes using Igor's drawing tools in a specific manner.

ImageGenerateROIMask creates an unsigned byte mask matrix with the same x and y dimensions and scaling as the specified image. The mask is initially filled with zeros. Then the drawing layer, progFront, in the graph is scanned for suitable fillable draw objects. The area inside each shape is filled with ones unless the fill mode for the shape is set to erase in which case the area is filled with zeros.

Flags

/E=*e* Changes value used for the exterior from the default zero values to *e*.

/I=*i* Changes value used for the interior from the default one values to *i*.

/W=*winName* Looks for the named graph window or subwindow containing appropriate image masks drawn by the user. If /W is omitted, ImageGenerateROIMask uses the top graph window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

Suitable objects are those that can be filled (rectangles, ovals, etc.) and which are plotted in axis coordinate mode specified using the same axes by which the specified image instance is displayed. Objects plotted in plot relative mode are also used, However, this is not recommended because it will give correct results only if the image exactly fills the plot rectangle. If you use axis coordinate mode then you can zoom in or out as desired and the resulting mask will still be correct.

Note that the shapes can have their fill mode set to none. This still results in a fill of ones. This is to allow the drawn ROI to be visible on the graph without obscuring the image. However cutouts (fills with erase mode) will obscure the image.

Note also that nonfill drawing objects are ignored. You can use this fact to create callouts and other annotations.

In a future version of Igor, we may create a new drawing layer in graphs dedicated to ROIs.

The mask generated is named M_ROIMask and is generated in the current data folder.

Variable V_flag is set to 1 if the top graph contained draw objects in the correct layer and 0 if not. If 0 then the M_ROIMask wave was not generated.

Examples

```
Make/O/N=(200,400) jack=x*y; NewImage jack; ShowTools
SetDrawLayer ProgFront
SetDrawEnv linefgc=(65535,65535,0),fillpat=0,xcoord=top,ycoord=left,save
DrawRect 63.5,79.5,140.5,191.5
DrawRRect 61.5,206.5,141.5,280.5
SetDrawEnv fillpat= -1
DrawOval 80.5,169.5,126.5,226.5
ImageGenerateROIMask jack
NewImage M_ROIMask
AutoPositionWindow/E
```

See Also

For another example see **Generating ROI Masks** on page III-322.

ImageHistModification

ImageHistModification [***flags***] ***imageMatrix***

The ImageHistModification operation performs a modification of the image histogram and saves the results in the wave M_ImageHistEq. If /W is not specified, the operation is a simple histogram equalization of *imageMatrix*. If /W is specified, the operation attempts to produce an image with a histogram close to *waveName*. If /A is specified, the operation performs an adaptive histogram equalization. *imageMatrix* is a wave of any noncomplex numeric type. Adaptive histogram equalization applies only to 2D waves and the other parts apply to both 2D and 3D waves.

Flags

/A Performs an adaptive histogram equalization by subdividing the image into a minimum of 4 rectangular domains and using interpolation to account for the boundaries between adjacent domains. When the /C flag is specified with contrast factor greater than 1, this operation amounts to contrast-limited adaptive histogram equalization. By default the operation divides the image into 8 horizontal and 8 vertical regions. See /H and /V.

/B=*bins* Specifies the number of *bins* used with the /A flag. If not specified, this value defaults to 128.

/C=*cFactor* Specifies a contrast factor (or clipping value) above which pixels are equally distributed over the whole range. *cFactor* must be greater than 1, in the limit as *cFactor* approaches 1 the operation is a regular adaptive histogram equalization. **Note**: this flag is used only with the /A flag.

/H=*hRegions* Specifies the number of horizontal subdivisions to be used with the /A feature. Note, the number of image pixels in the horizontal direction must be an integer multiple of *hRegions*.

/I Extends the standard histogram equalization by using 2^{16} bins instead of 2^8 when calculating histogram equalization. This feature does not apply to the adaptive histogram equalization (/A flag).

/O Overwrites the source image. If this flag is not specified, the resulting image is saved in the wave M_ImageHistEq.

/R=*roiSpec* Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u). The ROI wave must have the same number of rows and columns as *imageMatrix*. The ROI itself is defined by the entries whose values are 0. Regions outside the ROI can take any nonzero value. The ROI does not have to be contiguous and can take any arbitrary shape.

In general, the *roiSpec* has the form {*roiWaveName*, *roiFlag*}, where *roiFlag* can take the following values:

roiFlag=0: Set pixels outside the ROI to 0.

roiFlag=1: Set pixels outside the ROI as in original image (default).

roiFlag=2: Set pixels outside the ROI to NaN (=64).

By default *roiFlag* is set to 1 and it is then possible to use the /R flag with the abbreviated form /R=*roiWave*. When *imageMatrix* is a 3D wave, *roiWave* can be either a 2D wave (matching the number of rows and columns in *imageMatrix*) or it can be a 3D wave which must have the same number of rows, columns, and layers as *imageMatrix*. When using a 2D *roiWave* with a 3D *imageMatrix*, the ROI is understood to be defined by *roiWave* for each layer in the 3D wave.

See **ImageGenerateROIMask** for more information on creating ROI waves.

/V=*vRegions* Specifies the number of vertical subdivisions to be used with the /A flag. The number of image pixels in the horizontal direction must be an integer multiple of *vRegions*. If the image dimensions are not divisible by the number of regions that you want, you can pad the image using **ImageTransform padImage**.

/W=*waveName* Specifies a 256-point wave that provides the desired histogram. The operation will attempt to produce an image having approximately the desired histogram values. This flag does not apply to the adaptive histogram equalization (/A flag)

See Also

The **ImageGenerateROIMask** and **ImageTransform** operations for creating ROIs. For examples see **Histograms** on page III-316 and **Adaptive Histogram Equalization** on page III-299.

ImageHistogram

ImageHistogram [***flags***] ***imageMatrix***

The ImageHistogram operation calculates the histogram of *imageMatrix*. The results are saved in the wave W_ImageHist. If *imageMatrix* is an RGB image stored as a 3D wave, the resulting histograms for each color plane are saved in W_ImageHistR, W_ImageHistG, W_ImageHistB.

imageMatrix must be a real-valued numeric wave.

Flags

/I Calculates a histogram with 65536 bins evenly distributed between the minimum and maximum data values. The operation first finds the extrema and then calculates the bins and the resulting histogram. Data can be a 2D wave of any type including float or double.

/P=*plane* Restricts the calculation of the histogram to a specific plane when *imageMatrix* is a non RGB 3D wave.

/R=*roiWave* Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u) that has the same number of rows and columns as *imageMatrix*. The ROI itself is defined by the entries o pixels in the *roiWave* with value of 0. Pixels outside the ROI may have any nonzero value. The ROI does not have to be contiguous. When *imageMatrix* is a 3D wave, *roiWave* can be either a 2D wave (matching the number of rows and columns in *imageMatrix*) or it can be a 3D wave that must have the same number of rows, columns and layers as *imageMatrix*. When using a 2D *roiWave* with a 3D *imageMatrix* the ROI is understood to be defined by *roiWave* for each layer in the 3D wave.

See **ImageGenerateROIMask** for more information on creating 2D ROI waves.

/S Computes the histogram for a whole 3D wave possibly subject to 2D or 3D ROI masking. The /S and /P flags are mutually exclusive.

Details

The ImageHistogram operation works on images, but it handles both 2D and 3D waves of any data type. Unless you use one of the special features of this operation (e.g., ROI or /P or /I) you could alternatively use the **Histogram** operation, which computes the histogram for the full wave and includes additional options for controlling the number of bins.

If the data type of *imageMatrix* is single byte, the histogram will have 256 bins from 0 to 255. Otherwise, the 256 bins will be distributed between the minimum and maximum values encountered in the data. Use the /I flag to increase the number of bins to 65536, which may be useful for unsigned short (/W/U) data.

See Also

The **ImageHistModification** and **ImageGenerateROIMask** operations. For examples see **Histograms** on page III-316.

ImageInfo

ImageInfo(*graphNameStr, imageWaveNameStr, instanceNumber*)

The ImageInfo function returns a string containing a semicolon-separated list of information about the specified image in the named graph window or subwindow.

Parameters

graphNameStr can be "" to refer to the top graph.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

imageWaveNameStr contains either the name of a wave displayed as an image in the named graph, or an image instance name (wave name with "#n" appended to distinguish the nth image of the wave in the graph). You might get an image instance name from the **ImageNameList** function.

If *imageWaveNameStr* contains a wave name, *instanceNumber* identifies which instance you want information about. *instanceNumber* is usually 0 because there is normally only one instance of a wave displayed as an image in a graph. Set *instanceNumber* to 1 for information about the second image of the wave, etc. If *imageWaveNameStr* is "", then information is returned on the *instanceNumber*th image in the graph.

If *imageWaveNameStr* contains an instance name, and *instanceNumber* is zero, the instance is taken from *imageWaveNameStr*. If *instanceNumber* is greater than zero, the wave name is extracted from *imageWaveNameStr*, and information is returned concerning the *instanceNumber*th instance of the wave.

Details

The string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon for ease of use with **StringByKey**. The keywords are as follows:

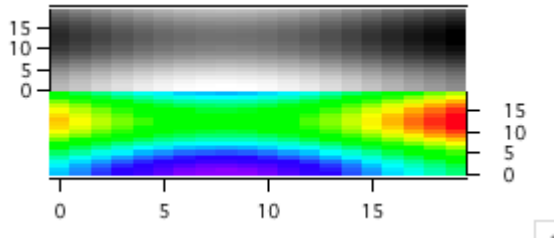
Keyword	Information Following Keyword
AXISFLAGS	Flags used to specify the axes. Usually blank because /L and /B (left and bottom axes) are the defaults.
COLORMODE	A number indicating how the image colors are derived: 1: Color table (see **Color Tables** on page II-349). 2: Scaled color index wave (see **Indexed Color Details** on page II-356). 3: Point-scaled color index (See **Example: Point-Scaled Color Index Wave** on page II-357). 4: Direct color (see **Direct Color Details** on page II-358). 5: Explicit Mode (See **ModifyImage** explicit keyword).
RECREATION	Semicolon-separated list of *keyword=modifyParameters* commands for the ModifyImage command.
XAXIS	X axis name.
XWAVE	X wave name if any, else blank.
XWAVEDF	The full path to the data folder containing the X wave or blank if there is no X wave.
YAXIS	Y axis name.
YWAVE	Y wave name if any, else blank.
YWAVEDF	The full path to the data folder containing the Y wave or blank if there is no Y wave.
ZWAVE	Name of wave containing Z data used to calculate the image plot.
ZWAVEDF	The full path to the data folder containing the Z data wave.

The format of the RECREATION information is designed so that you can extract a keyword command from the keyword and colon up to the "`;`", prepend "`ModifyImage `", replace the "`x`" with the name of a image plot ("`data#1`" for instance) and then **Execute** the resultant string as a command.

Example 1

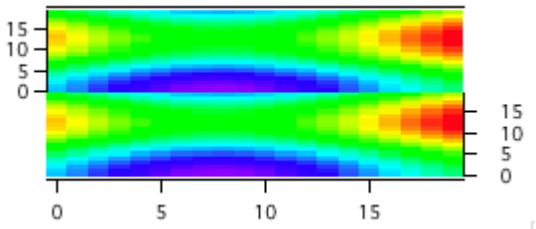
This example gets the image information for the second image plot of the wave "jack" (which has an instance number of 1) and applies its **ModifyImage** settings to the first image plot.

```
#include <Graph Utility Procs>, version>=6.1   // For WMGetRECREATIONFromInfo

// Make two image plots of the same data on different left and right axes
Make/O/N=(20,20) jack=sin(x/5)+cos(y/4)
Display;AppendImage jack                        // bottom and left axes
AppendImage/R jack                              // bottom and right axes

// Put image plot jack#0 above jack#1
ModifyGraph axisEnab(left)={0.5,1},axisEnab(right)={0,0.5}

// Set jack#1 to use the Rainbow color table instead of the default Grays
ModifyImage jack#1 ctab={*,*,Rainbow,0}
```



Now we peek at some of the image information for the second image plot of the wave "jack" (which has an instance number of 1) displayed in the top graph:

```
Print ImageInfo("","jack",1)[69,148]        // Just the interesting stuff
;ZWAVE:jack;ZWAVEDF:root:;COLORMODE:1;RECREATION:ctab= {*,*,Rainbow,0};plane= 0;
// Apply the color table, etc from jack#1 to jack:
String info= WMGetRECREATIONFromInfo(ImageInfo("","jack",1))
info= RemoveEnding(info)                    // Remove trailing semicolon
// Use comma instead of semicolon separators
String text = ReplaceString(";", info, ",")
Execute "ModifyImage jack " + text
```



Example 2

This example gets the full path to the wave containing the Z data from which the first image plot in the top graph was calculated.

```
String info= ImageInfo("","",0)      // 0 is index of first image plot
String pathToZ= StringByKey("ZWAVEDF",info)+StringByKey("ZWAVE",info)
Print pathToZ
   root:jack
```

See Also

The **ModifyImage**, **AppendImage**, **NewImage** and **Execute** operations.

How Images Are Displayed on page II-346.

Image Instance Names on page II-360.

ImageInterpolate

ImageInterpolate [*flags*] ***Method srcWave***

The ImageInterpolate operation interpolates the source *srcWave* and stores the results in the wave M_InterpolatedImage in the current data folder unless you specify a different destination wave using the /DEST flag.

Parameters

Method selects type of interpolation. *Method* is one of the following names:

Affine2D Performs an affine transformation on *srcWave* using parameters specified by the /APRM flag. The transformation applies to a general combination of rotation, scaling, and translation represented by a 3x3 matrix

$$M = \begin{bmatrix} r_{11} & r_{12} & tx \\ r_{21} & r_{22} & ty \\ 0 & 0 & w \end{bmatrix}$$

The upper 2x2 matrix is a composite of rotation and scaling, *tx* and *ty* are composite translations and *w* is usually 1. It computes the dimensions of the output wave and then uses the inverse transformation and bilinear interpolation to compute the value of each output pixel. When an output pixel does not map back into the source domain it is set to the user-specified background value. It supports 2D and 3D input waves. If *srcWave* is a 3D wave it applies on a layer by layer basis.

The output is stored in the wave M_Affine in the current data folder.

Bilinear Performs a bilinear interpolation subject to the specified flag. You can use either the /F or /S flag, but not both.

Kriging — Uses Kriging to generate an interpolated matrix from a sparse data set. Kriging calculates interpolated values for a rectangular domain specified by the /S flag. The Kriging parameters are specified via the /K flag.

Kriging is computed globally for a single user-selected variogram model. If there are significant spatial variances within the domain occupied by the data, you should consider subdividing the domain along natural boundaries and use a single variogram model in each subdivision.

If there are N data points, the algorithm first computes the NxN matrix containing the distances between the data and then inverts an associated matrix of similar size to compute the result for the selected variogram model. Because inversion of an NxN matrix can be computationally expensive, you should consider restricting the calculation to regions that are similar to the range implied by the variogram. Such an approach can also be justified in the sense that the local interpolation should not be affected by a remote datum.

Note: Kriging does not support data containing NaNs or INFs. Wave scaling has no effect.

Pixelate — Creates a lower resolution (pixelated image) of *srcWave* by averaging the pixels inside rectangles specified by /PXSZ flag. The results are saved in the wave M_PixelatedImage in the current data folder. The computed wave has the same numeric type as *srcWave* and the same number of layers. The number of rows and the number of columns of the new image are obtained by integer division of the original number by the respective size of the averaging rectangle and adding one more pixel for any remainder.

Resample — Computes a new image based on the selected interpolation function and transformation parameters. Set the interpolation function with the /FUNC flag. Use the /TRNS flag to specify transformation parameters for grayscale images, or /TRNR, /TRNG, and /TRNB for the red, green, and blue components, respectively, of RGB images. M_InterpolatedImage contains the output image in the current data folder.

There are currently two transformation functions: the first magnifies an image and the second applies a radial polynomial sampling. The radial polynomial affects pixels based on their position relative to the image center. A linear polynomial reproduces the same image. Any nonlinear terms contribute to distortion (or correction thereof).

Spline — Computes a 2D spline interpolation for 2D matrix data. The degree of the spline is specified by the /D flag.

Voronoi — Generates an interpolated matrix from a sparse data set (*srcWave* must be a triplet wave) using Voronoi polygons. It calculates interpolated values for a rectangular domain as specified by the /S flag. It first computes the Delaunay triangulation of X, Y locations in the Z=0 plane (assuming that X, Y positions occupy a convex domain in the plane). It then uses the Voronoi dual to interpolate the Z values for X and Y pairs from the grid defined by the /S flag. The computed grid may exceed the bounds of the convex domain defined by the triangulation. Interpolated values for points outside the convex domain are set to NaN or the value specified by the /E flag. Use the /I flag to iterate to finer triangulation by subdividing the original triangles into smaller domains. Each iteration increases computation time by approximately a factor of two, but improves the smoothness of the interpolation.

If you have multiple sets of data in which X,Y locations are unchanged, you can use the /STW flag to store one triangulation and then use the /PTW flag to apply the precomputed triangulation to a new interpolation. To use this option you should use the Voronoi keyword first with a triplet wave for *srcWave* and set xn = x0 and yn = y0. The operation creates the wave W_TriangulationData that you use in the next triangulation with a 1D wave as *srcWave*. For example:

```
ImageInterpolate/STW/S={0,1,0,1,1,1} Voronoi myTripletWave
ImageInterpolate/PTW=W_TriangulationData/S={0,1,0,1,1,1} Voronoi my1DZWave
```

Voronoi interpolation is similar to what can be accomplished with the **ContourZ** function except that it does not require an existing contour plot, it computes the whole output matrix in one call, and it has the option of controlling the subdivision iterations.

XYWaves — Performs bilinear interpolation on a matrix scaled using two X and Y 1D waves (specified by /W). The interpolation range is defined by /S. The data domain is defined between the centers of the first and last pixels (X in this example):

```
xmin=(xWave[0]+xWave[1])/2
xmax=(xWave[last]+xWave[last-1])/2
```

Values outside the domain of the data are set to NaN. The interpolation is contained in the M_InterpolatedImage wave, which is single precision floating point or double precision if *srcWave* is double precision.

Warp Performs image warping interpolation using a two step algorithm with three optional interpolation methods. The operation warps the image based on the relative positions of source and destination grids. The warped image has the same size as the source image. The source and destination grids are each specified by a pair of 2D X and Y waves where the rows and columns correspond to the relative location of the source grid. The smallest supported dimensions of grid waves are 2x2. All grid waves must be double-precision floating point and must have the same number of points corresponding to pixel positions within the image. Grid waves must not contain NaNs or INFs. Wave scaling is ignored.

Flags

/APRM={*r11*,*r12*,*tx*,*r21*,*r22*,*ty*,*w*,*background*}
Sets elements of the affine transformation matrix and the background value.

/D=*splineDeg* Specifies the spline degree with the Spline method. The default spline degree is 2. Supported values are 2, 3, 4, and 5.

/DEST=*destWave*

Specifies the wave to contain the output of the operation. If the specified wave already exists, it is overwritten.

Creates a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-56 for details.

/E=*outerValue* Assigns *outerValue* to all points outside the convex domain of the Delaunay triangulation. By default *outerValue* = NaN.

/F={*fx*,*fy*} Calculates a bilinear interpolation of all the source data. Here *fx* is the sampling factor for the X-direction and *fy* is the sampling factor in the Y-direction. The output number of points in a dimension is factor*(number of data intervals) +1. The number of data intervals is one less than the number of points in that dimension.

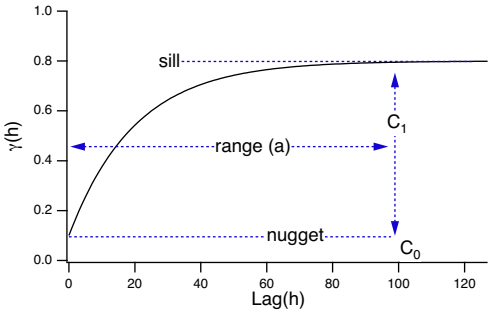
For example, if *srcWave* is a 2x2 matrix (you have a single data interval in each direction) and you use /F={2,2}, then the output wave is a 3x3 matrix (i.e., 2x2 intervals) which is a factor of 2 of the input. Sampling factors can be noninteger values.

/FUNC=*funcName* Specifies the interpolation function. *funcName* can be:

- nn Nearest neighbor interpolation uses the value of the nearest neighbor without interpolation. This is the fastest function.
- bilinear Bilinear interpolation uses the immediately surrounding pixels and computes a linear interpolation in each dimension. This is the second fastest function.
- cubic Cubic polynomial (photoshop-like) uses a 4x4 neighborhood value to compute the sampled pixel value.
- spline Spline smoothed sampled value uses a 4x4 neighborhood around the pixel.
- sinc Slowest function using a 16x16 neighborhood.

/I=*iterations* Specifies the number of times the original triangulation is subdivided with the Voronoi interpolation method. By default the Voronoi interpolation computes the original triangulation without subdivision.

/K={*model*, *nugget*, *sill*, *range*}
Specifies the variogram parameters for kriging using standard notation, models are expressed in terms of the nugget value C_0, sill value C_0+C_1, and range a.



model Selects the variogram model. Values and models are:

1: Spherical. $\gamma(h) = C_0 + C_1 \cdot (3h/2a - 0.5 \cdot h^2/a^2)$

2: Exponential. $\gamma(h) = C_0 + C_1 \cdot (1 - \exp[-3 \cdot h/a])$

3: Gaussian. $\gamma(h) = C_0 + C_1 \cdot (1 - \exp[-3 \cdot (h/a)^2])$

nugget Specifies the lowest value in the variogram.

sill Specifies the maximum (plateau) value in the variogram range the characteristic length of the different variogram models.

Wave scaling has no effect on kriging calculations.

/PTW=*tWave* Uses a previous triangulation wave with Voronoi interpolation. *tWave* will be the wave saved by the /STW flag. You can't use a triangulation wave that was computed and saved on a different computer platform.

/PXSZ={*nx*, *ny*} Specifies the size of the averaging rectangle in pixels. Here *nx* is the number of rows and *ny* is the number of columns that are averaged to yield a single output pixel.

/RESL={*nx*, *ny*} Specifies resampling the full input image to an output image having *nx* rows by *ny* columns.

/S={*x0*,*dx*,*xn*,*y0*,*dy*,*yn*}
Calculates a bilinear interpolation of a subset of the source data. Here *x0* is the starting point in the X-direction, *dx* is the sampling increment, *xn* is the end point in the X-direction and the corresponding values for the Y-direction. If you can set *x0* equal to *xn* the operation will compute the triangulation but not the interpolation

/STW Saves the triangulation information in the wave W_TriangulationData in the current data folder. W_TriangulationData can only be used on the computer platform where it was created.

/SV Saves the Voronoi interpolation in the 2D wave M_VoronoiEdges, which contains sequential edges of the Voronoi polygons. Edges are separated from each other by a row of NaNs. The outer most polygons share one or more edges with a large triangle containing the convex domain.

/TRNS={*transformFunc*,*p1*,*p2*,*p3*,*p4*}
Determines the mapping between a pixel in the destination image and the source pixel. *transformFunc* can be:

scaleShift Sets image scaling which could be anamorphic if the X and Y scaling are different.

radialPoly Corrects both color as well as barrel and pincushion distortion. In `radialPoly` the mapping from a destination pixel to a source pixel is a polynomial in the pixel's radius relative to the center of the image.

A source pixel, *sr*, satisfies the equation: $sr = a \cdot r + b \cdot r^2 + c \cdot r^3 + d \cdot r^4$, where *r* is the radius of a destination pixel having an origin at the center of the destination image.

The corresponding parameters are:

/U=*uniformScale* Calculates a bilinear interpolation of all the source data as with the /F flag but with two exceptions: A single uniform scale factor applies in both dimensions, and the scale factor applies to the number of points — not the intervals of the data.

transformFunc	*p1*	*p2*	*p3*	*p4*
scaleShift	xOffset	xScale	yOffset	yScale
radialPoly	a	b	c	d

/W={*xWave*, *yWave*} Provides the scaling waves for XYWaves interpolation. Both waves must be monotonic and must have one more point than the corresponding dimension in *srcWave*. The waves contain values corresponding to the edges of data points in *srcWave*, so that the X value at the first data point is equal to `(xWave[0]+xWave[1])/2`.

Flags for Warp

/dgrx=*wave* Sets the wave containing the destination grid X data.

/dgry=*wave* Sets the wave containing the destination grid Y data.

/sgrx=*wave* Sets the wave containing the source grid X data.

/sgry=*wave* Sets the wave containing the source grid Y data.

/WM=*im* Sets the interpolation method for warping an image.

I	Method
1	Fast selection of original data values.
2	Linear interpolation.
3	Smoothing interpolation (slow)

Details

When computing Bilinear or Spline interpolation *srcWave* can be a 2D or a 3D wave. When *srcWave* is a 3D wave the interpolation is computed on a layer by layer basis and the result is stored in a corresponding 3D wave. When the interpolation method is Kriging or Voronoi, *srcWave* is a 2D triplet wave (3-column wave) where each row specifies the X, Y, Z values of a datum. *srcWave* can be of any real data type. Results are stored in the wave M_InterpolatedImage. If *srcWave* is double precision so is M_InterpolatedImage; otherwise M_InterpolatedImage is a single precision wave.

See Also

The **interp**, **Interp3DPath**, **ImageRegistration**, and **Loess** operations. The **ContourZ** function. For examples see **Interpolation and Sampling** on page III-303.

References

Unser, M., A. Aldroubi, and M. Eden, B-Spline Signal Processing: Part I-Theory, *IEEE Transactions on Signal Processing*, *41*, 821-832, 1993.

Douglas B. Smythe, "A Two-Pass Mesh Warping Algorithm for Object Transformation and Image Interpolation" ILM Technical Memo #1030, Computer Graphics Department, Lucasfilm Ltd. 1990.

ImageLineProfile

ImageLineProfile [***flags***] **xWave=*xwave*, yWave=*ywave*, srcWave=*srcWave* [, width=*value*, widthWave=*wWave*]**

The ImageLineProfile operation provides sampling of a source image along an arbitrary path specified by the two waves: *xWave* and *yWave*. The arbitrary path is made of line segments between every two consecutive vertices of *xWave* and *yWave*. In each segment the profile is calculated at a number of points (profile points) equivalent to the sampling density of the original image (unless the /V flag is used). Both *xWave* and *yWave* should have the same scaling as *srcWave*. If *srcWave* does not have the same scaling in both dimensions you should remove the scaling to compute an accurate profile.

At each profile point, the profile value is calculated by averaging samples along the normal to the profile line segment. The number of samples in the average is determined by the keyword width. The operation actually averages the interpolated values at N equidistant points on the normal to profile line segment, with `N=2(width+0.5)`. Samples outside the domain of the source image do not contribute to the profile value.

The profile values are stored in the wave W_ImageLineProfile. The actual locations of the profile points are stored in the waves W_LineProfileX and W_LineProfileY. When the averaging width is greater than zero, the operation can also calculate at each profile point the standard deviation of the values sampled for that

point (see /S flag). The results are then stored in the wave W_LineProfileStdv. When using this operation on 3D RGB images, the profile values are stored in the three-column waves M_ImageLineProfile and M_LineProfileStdv respectively.

Parameters

srcWave=*srcWave* Specifies the image for which the line profile is evaluated. The image may be a 2D wave of any type or a 3D wave or RGB data.

xWave=*xwave* Specifies the wave containing the x coordinate of the line segments along the path.

yWave=*ywave* Specifies the wave containing the y coordinate of the line segments along the path.

width=*value* Specifies the width (diameter) in pixels (need not be an integer value) in a direction perpendicular to the path over which the data is interpolated and averaged for each path point. If you do not specify width or use width=0, only the interpolated value at the path point is used.

widthWave=*wWave* Specifies the width of the profile (see definition above) on a segment by segment basis. *wWave* should be a 1D wave that has the same number of entries as xWave and yWave. If you provide a widthWave any value assigned with the width keyword is ignored. All values in the wave must be positive and finite.

Flags

/P=*plane* Specifies which plane (layer) of a 3D wave is to be profiled. By default *plane* =-1 and the profiles are of either the single layer of a 2D wave or all three layers of a 3D RGB wave. Use *plane* =-2 if you want to profile all layers of a 3D wave.

/S Calculates standard deviations for each profile point.

/SC Saves W_LineProfileX and W_LineProfileY using the X and Y scaling of *srcWave*.

/V Calculate profile points only at the vertices of xWave and yWave.

Examples

```
Make/N=(50, 50) sampleData
sampleData = sin((x-25) / 10) * cos((y-25) / 10)
NewImage sampleData
Make/n=2 xTrace={0,50} ,yTrace={20,20}
ImageLineProfile srcWave=sampleData, xWave=xTrace, yWave=yTrace
AppendtoGraph/T yTrace vs xTrace
Display W_ImageLineProfile
```

See Also

For additional examples see **ImageLineProfile** on page III-316.

ImageLoad

ImageLoad [***flags***] [***fileNameStr***]

The ImageLoad operation loads an image file into an Igor matrix wave. See the /T flag for a list of supported image file types.

ImageLoad requires QuickTime to load certain types of images.

Parameters

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where pathName is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

On Macintosh, when loading a format that requires QuickTime, the name of the file to be loaded is limited to 31 characters because Igor calls Apple routines that have this limit.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

If you want to force a dialog to select the file, omit the *fileNameStr* parameter.

Flags

/C=*count* Specifies which images to load from a multi-image TIFF file. It loads and stores images as individual waves in the current data folder. By default, it loads only a single image

(i.e., /C=1). Use /C=-1 to load all images into a single 3D wave (images must be either 8-, 16-, or 32-bits/pixel for this option). If you specify a *count* that exceeds the number of images in the file, it will load all images beginning with the starting image. See also the /S flag and examples.

/G — Displays the loaded image.

/LR3D — Loads a partial range of a TIFF stack into a 3D wave (see also /C and /S).

/N=*baseName* — Stores the waves using *baseName* identically for the wave name. Only when *baseName* conflicts with an existing wave name will a numeric suffix be appended to the new wave names.

If you do not specify *baseName*, ImageLoad will use the name of the file as a base name.

/O — Overwrites an existing wave with the same name.

If /O is omitted and there is an existing wave with the same name, a numerical suffix will be appended to the image name.

/P=*pathName* — Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

/Q — Quiet mode; suppresses insertion of loading info into the history area.

/RAT — Read All Tags reads all of the tags in a TIFF file into one or more waves. It creates a data folder named "Tag*n*" (with a numeric suffix, *n*, starting from zero) for each loaded image. When reading multiple images from a stack TIFF file it will create a corresponding number of data folders.

Each data folder contains a text wave named T_Tags, which contains 5 columns. The first row contains the offset of the current Image File Directory (IFD) from the start of the file. The remaining rows describe the individual TIFF Tags as they appear in the IFD.

The first column contains the tag number, the second contains the tag description, the third contains the tag type, the fourth contains the tag length, and the fifth contains either the value of the tag or a statement identifying the name of the wave in which the data was stored. For example, a simple tag that contains a single value has the form:

Num	Desc	Type	Length	Value
256	IMAGEWIDTH	4	1	2560

A tag that contains more data, such as an array of values has the form:

Num	Desc	Type	Length	Value
273	STRIPOFFSETS	4	-120	tifTag273

Here the Length field is negative (-1*realLength) and the Value field contains the name of the wave tifTag273 which contains the array of strip offsets.

When the Value field consists of ASCII characters it is stored in the T_Tags wave itself. All other types are stored in a wave in the same Tag data folder.

Private tags are usually designated by negative tag numbers. If their data type is anything other than ASCII, they are saved in separate waves.

/RTIO — Reads tag information only from a TIFF file. (This flag is similar to the /RAT flag but it does not load the images.) If you are loading a stack of images you can use the /C and /S flags to obtain tags from a specific range of images.

/T=*type* — Identifies what kind of image file to load. *type* is one of the following image file formats:

type	Loads this Image Format
any	Any graphic file type.
bmp	PC bitmap file.
gif	GIF file.
jpeg	JPEG file.
photoshop	PhotoShop file.

type	Loads this Image Format
pict	Macintosh picture file.
png	PNG file.
rpng	Raw PNG file (see **Details**).
sgi	Silicon Graphics file.
sunraster	Sun Raster file.
targa	Targa file.
tiff	Any TIFF file (see also **Special Notes Regarding TIFF Files**).

If you do not specify *type*, Igor will make a guess based on the file type (*Macintosh*) or file name extension (*Windows*). An error will be reported if Igor is unable to guess the image type.

/S=*start* Specifies the starting image in TIFF files that contain multiple images. By default, /S=0, so if you want to load all images just use /C=-1.

/Z no error reporting.

Details

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "hd:Folder1:" or "C:\\Folder1\\". See **Symbolic Paths** on page II-34 for details.

ImageLoad sets the following variables:

`V_flag`	Set to 1 if the image was successfully loaded or to 0 otherwise.
`S_fileName`	Set to the name of the file that was loaded.
`V_numImages`	Set to the number of images in the file. Use this variable only with TIFF files.
`S_path`	Set to the file system path to the folder containing the file.
`S_waveNames`	Set to a semicolon-separated list of the names of loaded waves.

S_path uses Macintosh path syntax (e.g., "hd:FolderA:FolderB:"), even on Windows. It includes a trailing colon.

When loading a rpng format PNG image, the wave will be either 8- or 16-bit unsigned integer format with 1 to 4 planes. PNG images with physical units will produce waves with X and Y units of meters. If a PNG image has a color table, ImageLoad will create two waves: a main image wave with 1 plane and a color table wave of the same name but with a "_pal" suffix (if the name is too long, it will create a wave named *PNG_pal* instead).

Special Notes Regarding TIFF Files

The ImageLoad operation reads data from the named TIFF file into waves. The operation supports 1-, 8-, 16- and 24-bit TIFF files. In case of 1-bit/pixel the data is converted into a 1 byte/pixel (/B/U wave). Images of 16-bits/pixel are read into /W/U waves. We note that 16-bit format is not part of the TIFF 6.0 specification but it is used extensively by scientific camera manufacturers. Images of 24-bits/pixel are read into 3D RGB waves.

When a TIFF file contains multiple images (stack), you can read all images into waves or you can specify a particular image or a range of images that you would like to read. To read the complete stack specify -1 as the image count and all images will be saved into a single 3D wave where each image is a sequential plane. This is appropriate only for images that are 8- or 16-bits/pixel deep. If the depth of the image is other than 8- or 16-bits/pixel you should read the images into separate waves.

If you have a TIFF file that is not read correctly by this operation (e.g., 4-bits/pixel), you may be able to read the file using QuickTime. To do so, make sure that the file name does not contain ".tif" or ".tiff" suffix (and on the Macintosh change its file type from `'TIFF'` to `'????'`). You can then load the file using the "Any" file designation and the file will be read by QuickTime resulting in an RGB image wave.

Examples

Reading TIFF files containing a stack of images:

```
Imageload /C=8/S=10/T=Tiff // Creates 8 waves starting from image #10 (zero based)
ImageLoad/C=-1/S=10/T=TIFF // Loads all the images ignoring the /S flag.
                           // The images are stored in a single 3D wave.
ImageLoad/C=-1/T=TIFF/RTIO // Does not load the images but reads all tags.
NewDataFolder/O/S tmp      // Gets the number of images and cleans up
ImageLoad/C=-1/T=TIFF/RTIO
```

```
Print V_numImages
KillDataFolder :
```

See Also

See Chapter II-9, **Importing and Exporting Data**, for further information on loading waves, including loading multidimensional data from HDF files (see **Loading HDF Data** on page II-169).

For loading graphic objects, see **Pictures** on page III-421. The **ImageSave** operation for saving waves as image files.

ImageMorphology

ImageMorphology [***flags***] ***Method imageMatrix***

The ImageMorphology operation performs one of several standard image morphology operations on the source *imageMatrix*. Unless the /O flag is specified, the resulting image is saved in the wave M_ImageMorph. The operation applies only to waves of type unsigned byte. All ImageMorphology methods except for watershed use a structure element. The structure element may be one of the built-in elements (see /E flag) or a user specified element.

Erosion, Dilation, Opening, and Closing are the only methods supported for a 3D *imageMatrix*.

Parameters

Method is one of the following names:

BinaryErosion	Erodes the source binary image using a built-in or user specified structure element (see /E and /S flags).
BinaryDilation	Dilates the source binary image using a built-in or user specified structure element (see /E and /S flags).
Closing	Performs the closing operation (dilation followed by erosion). The same structure element is used in both erosion and dilation. Note that this operation is an idempotent, which means that there is no point of executing it more than once.
Dilation	Performs a dilation of the source grayscale image using either a built-in structure element or a user specified structure element. The operation supports only 8-bit gray images.
Erosion	Erodes the source grayscale image using either a built-in structure element or a user specified structure element. The operation supports only 8-bit gray images.
Opening	Performs an opening operation (erosion followed by dilation). The same structure element is used in both erosion and dilation. Note that this operation is an idempotent which means that there is no point of executing it more than once.
TopHat	Calculates the difference between the eroded image and dilated image using the same structure element.
Watershed	Calculates the watershed regions for grayscale or binary image. Use the /N flag to mark all nonwatershed lines as NaNs. The /L flag switches from using 4 neighboring pixels (default) to 8 neighboring pixels.

Flags

/E=*id* Uses a particular built in structure element. The following are the built-in structure element. The following are the built-in structure elements; make sure to use the appropriate id for the dimensionality of *imageMatrix*:

id	Element	Origin	Shape
1	2x2	(0,0)	square (default)
2	1x3	(1,1)	row (in 3x3 square)
3	3x1	(1,1)	column (in 3x3 square)
4	3x3	(1,1)	cross (in 3x3 square)
5	5x5	(2,2)	circle (in 5x5 square)
6	3x3	(1,1)	full 3x3 square
200	2x2x2	(1,1,1)	symmetric cube
202	2x2x2	(1,1,1)	2 voxel column in Y direction

id	Element	Origin	Shape
203	2x2x2	(1,1,1)	2 voxel column in X direction
204	2x2x2	(1,1,1)	2 voxel column in Z direction
205	2x2x2	(1,1,1)	XY plane
206	2x2x2	(1,1,1)	YZ plane
207	2x2x2	(1,1,1)	XZ plane
300	3x3x3	(1,1,1)	symmetric cube
301	3x3x3	(1,1,1)	symmetric ball
302	3x3x3	(1,1,1)	3 voxel column in Y direction
303	3x3x3	(1,1,1)	3 voxel column in X direction
304	3x3x3	(1,1,1)	3 voxel column in Z direction
305	3x3x3	(1,1,1)	XY plane
306	3x3x3	(1,1,1)	YZ plane
307	3x3x3	(1,1,1)	XY plane
500	5x5x5	(2,2,2)	symmetric cube
501	5x5x5	(2,2,2)	symmetric ball
700	7x7x7	(3,3,3)	symmetric cube
701	7x7x7	(3,3,3)	symmetric ball

Note that this flag has no effect on watershed calculations.

/I= *iterations* Repeats the operation the specified number of *iterations*.

/L Uses 8-connected neighbors instead of 4.

/N Sets the background level to 64 (= NaN).

/O Overwrites the source wave with the output.

/R=*roiSpec* Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u). The ROI wave must have the same number of rows and columns as the image wave. The ROI itself is defined by the entries/pixels whose values are 0. Pixels outside the ROI can take any nonzero value. The ROI does not have to be contiguous and can take any arbitrary shape. See **ImageGenerateROIMask** for more information on creating ROI waves.

In general, the *roiSpec* has the form {*roiWaveName*, *roiFlag*}, where *roiFlag* can take the following values:

roiFlag=0: Set pixels outside the ROI to 0.

roiFlag=1: Set pixels outside the ROI as in original image.

roiFlag=2: Set pixels outside the ROI to NaN (=64).

By default *roiFlag* is set to 1 and it is then possible to use the /R flag using the abbreviated form /R=*roiWave*.

/S= *seWave* Specifies your own structure element.

seWave must be of type unsigned byte with pixels that belong to the structure element set to 1 and background pixels set to 0.

There are no limitations on the size of the structure element and you can use the /X and /Y flags to specify the origin of your structure element.

/W= *whiteVal* Sets the white value in the binary image if it is different than 255. The black level is assumed to be zero.

/X= *xOrigin* Specifies the X-origin of a user-defined structure element starting at 0. If you do not use this flag Igor sets the origin to the center of the specified structure element.

/Y= *yOrigin* Specifies the Y-origin of a user defined structure element starting at 0. If you do not use this flag Igor sets the origin to the center of the specified structure element.

/Z= *zOrigin* Specifies the Z-origin of the element for 3D structure elements. If you do not use this flag Igor sets the origin to the center of the specified structure element.

Examples

If you would like to apply a morphological operation to a wave whose data type is not an unsigned byte and you wish to retain the wave's dynamic range, you can use the following approach:

```
Function ScaledErosion(inWave)
   Wave inWave

   WaveStats/Q inWave
   Variable nor=255/(V_max-V_min)
   MatrixOp/O tmp=nor*(inWave-V_min)
   Redimension/B/U tmp
   ImageMorphology/E=5 Erosion tmp
   Wave M_ImageMorph
   MatrixOp/O inWave=(M_ImageMorph/nor)+V_min
   KillWaves/Z tmp,M_ImageMorph
End
```

See Also

The **ImageGenerateROIMask** operation for creating ROIs. For details and usage examples see **Morphological Operations** on page III-312 and **Particle Analysis** on page III-319.

ImageNameList

ImageNameList(*graphNameStr, separatorStr*)

The ImageNameList function returns a string containing a list of image names in the graph window or subwindow identified by *graphNameStr*.

Parameters

graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

separatorStr should contain a single character such as "," or ";" to separate the names.

An image name is defined as the name of the 2D wave that defines the image with an optional #ddd suffix that distinguishes between two or more images that have the same wave name. Since the image name has to be parsed, it is quoted if necessary.

Examples

The following command lines create a very unlikely image display. If you did this, you would want to put each image on different axes, and arrange the axes such that they don't overlap. That would greatly complicate the example.

```
Make/O/N=(20,20) jack,'jack # 2';
Display;AppendImage jack
AppendImage/T/R jack
AppendImage 'jack # 2'
AppendImage/T/R 'jack # 2'
Print ImageNameList("",";")
```

prints jack;jack#1;'jack # 2';'jack # 2'#1;

See Also

Another command related to images and waves: **ImageNameToWaveRef**.

For commands referencing other waves in a graph: **TraceNameList**, **WaveRefIndexed**, **XWaveRefFromTrace**, **TraceNameToWaveRef**, **CsrWaveRef**, **CsrXWaveRef**, **ContourNameList**, and **ContourNameToWaveRef**.

ImageNameToWaveRef

ImageNameToWaveRef(*graphNameStr, imageNameStr*)

The ImageNameToWaveRef function returns a wave reference to the 2D wave corresponding to the given image name in the graph window or subwindow named by *graphNameStr*.

Parameters

graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

The image is identified by the string in *imageNameStr*, which could be a string determined using **ImageNameList**. Note that the same image name can refer to different waves in different graphs.

See Also

The **ImageNameList** function.

For a discussion of wave references, see **Wave Reference Functions** on page IV-173.

ImageRegistration

ImageRegistration [*flags*] **[testMask=***testMaskWave***] [refMask=***refMaskWave***] testWave=***imageWave1***, refWave=***imageWave2***

The ImageRegistration operation adjusts the test image wave, testWave, to match the reference image wave, refWave, possibly subject to auxiliary mask waves. The registration may involve offset, rotation, scaling or skewing.

Image data may be in two or three dimensions.

ImageRegistration is designed to find accurate registration for relatively small variation on the order of a few degrees of rotation and a few pixels offset between the reference and test images.

All the input waves are expected to be single precision float (SP) so you may have to redimension your images before using ImageRegistration.

ImageRegistration does not tolerate NaNs or INFs; use the masks if you need to exclude pixels from the registration process.

Parameters

refMask=*refMaskWave* Specifies an optional ROI wave used to mask refWave. Omit refMask to use all of the refWave.

The wave must have the same dimensions as refWave. refMask is a single precision floating point wave with nonzero entries marking the "ON" state. Note that the operation modifies this wave and that you should not use the same wave for both the reference and the test masks.

refWave=*imageWave2* Specifies the name of the reference image wave used to adjust testWave.

testMask=*testMaskWave* Specifies an optional ROI wave used to mask testWave. Omit testMask to use all of the testWave.

The wave must have the same dimensions as refWave. testMask is a single precision floating point wave with nonzero entries marking the "ON" state. Note that the operation modifies this wave and that you should not use the same wave for both the reference and the test masks.

testWave=*imageWave1* Specifies the name of the image wave that will be adjusted to match refWave.

testMaskWave and *refMaskWave* are optional ROI waves. The waves must have the same dimensions as testWave and refWave respectively. They must be single precision floating point waves with nonzero entries marking the "ON" state. If you need to include the whole region described by testWave or the whole region described by refWave you can omit the respective mask wave

Flags

/ASTP=*val* Sets the adaptation step for the Levenberg-Marquardt algorithm. Default value is 4.

/BVAL=*val* Enables clipping and sets the background values to which masked out voxels of the test data will be set.

/CONV=*val* Sets the convergence method.

val=0: Gravity, use if the difference between the images is only in translation. This option is frequently useful as a first step when the test and reference data are too far apart for accurate registration. The result of this registration is then passed to a subsequent ImageRegistration with /CONV=1.

val=1: Marquardt.

/CSNR=*val* Determines if the operation calculates the signal to noise ratio (SNR).

val=0: The SNR is not calculated.

val=1: The SNR is calculated (default).

Skipping the SNR calculation saves time and may be particularly useful when performing the registration on a stack of images.

/FLVL=*val* Specifies the finest level on which the optimization is to be performed.

If this is the same as /PRDL, then only the coarsest registration calculation is done.

If /FLVL=1 (default), then the full multiresolution pyramid is processed. You can use this flag to terminate the computation at a specified coarseness level greater than 1.

/GRYM Optimizes the gray level scaling factor.

It is sometimes dangerous to let the program adjust for gray levels because in some situations it might result in a null image.

/GRYR Renders output using the gray scaling parameter. This is more meaningful if the operation computes the optimal gray scaling (see /GRYM).

/GWDT={*sx*,*sy*,*sz*} Sets the three fields to the half-width of a Gaussian window that is used to smooth the data when computing the default masks. Defaults are {1,1,1}. See /REFM and /TSTM for more details.

/INTR=*val* Sets the interpolation method.

val=0: Nearest neighbor. Used when registering the center of gravity of the test and reference images.

val=1: Trilinear.

val=2: Tricubic (default).

/ISOS Optimizes the isometric scaling. This option is inappropriate if voxels are not cubic.

/ISR Computes the multiresolution pyramid with isotropic size reduction.

If the flag is not specified, the size reduction is in the XY plane only.

/MING=*val* Sets the minimum gain at which the computations will stop. Default value is zero, but you can use a slightly larger value to stop the iterations earlier.

/MSKC=*val* Sets mask combination value. During computation the masks for the test data and the mask for the reference data are also transformed. This flag determines how the two masks are to be combined. The registration criteria are computed for the combination of the two masks.

val=0: or.

val=1: nor.

val=2: and (default).

val=3: nand.

val=4: xor.

val=5: nxor.

/PRDL=*depth* Specifies the depth of the multiresolution pyramid. The finest level is *depth*=1. Each level of the pyramid decreases the resolution by a factor of 2. By default, the pyramid *depth*=4, which corresponds to a resolution reduction by a factor of $2^{(depth-1)}=8$.

The algorithm starts by computing the first registration on large scale features in the image (deepest level of the pyramid). It then makes small corrections to the registration at each consecutive pyramid level.

For best results, the coarsest representation the data should be between 30 and 60 pixels on a side. For example, for an image that is *H* by *V* pixels, you should choose the depth such that $H/2^{(depth-1)} \approx 30$.

/PSTK When performing registration of a stack of images, use this flag to apply the registration parameters of the previous layer as the initial guess for the registration of each layer after the first in the 3D stack.

/Q Quiet mode; no messages printed in the history window.

/REFM=*val* Sets the reference mask.

val=0: To leave blank and then every pixel is taken into account.

val=1: Value will be set if a valid reference mask is provided.

val=2: The reference mask is computed (default).

When computing the reference mask it is assumed that brighter features are more important. This is done by using a low pass filter on the data (using the parameters in /GWDT) which is then converted into a binary mask. Note that you do not need to specify /REFM=1 if you are providing a reference mask wave. See also /TSTM.

/ROT={*rotX*,*rotY*,*rotZ*}

Determines if optimization will take into account rotation about the X, Y, or Z axes. A value of one allows optimization and zero prevents optimization from affecting the corresponding rotation parameter. Defaults are {0,0,1}, which are the appropriate for rotating images.

/SKEW={*skewX*,*skewY*,*skewZ*}

Determines if optimization will take into account skewness about the X, Y, or Z axes. A value of one allows optimization and zero prevents optimization from affecting the corresponding skewness parameter. Defaults are {0,0,0}. Note that skewing and rotation or isometric scaling are mutually exclusive operations.

/STCK Use /STCK to perform the registration between a 2D reference image and each of the layers in a 3D image. The number of rows and columns of the refWave must match exactly the number of rows and columns in testWave. The transformation parameters are saved in the wave M_RegParams where each column contains the parameters for the corresponding layer in testWave.

/STRT=*val* Sets the first value of the adaptation parameter in the Levenberg-Marquardt algorithm.

The default value of this parameter is 1.

/TRNS={*transX*,*transY*,*transZ*}

Determines if optimization will take into account translation about the X, Y, or Z axes. A value of one allows optimization and zero prevents optimization from affecting the corresponding translation parameter. Defaults are {1,1,0}, which are appropriate for finding the X and Y translations of an image.

/TSTM=*val* Sets the test mask.

val=0: To leave blank and then every pixel is taken into account.

val=1: This value will be set if a valid reference mask is provided.

val=2: The reference mask is computed. This is the default value.

The test image mask is computed in the same way as the reference image mask (see /REFM) using the same set of smoothing parameters. Note that you do not need to specify /TSTM=1 if you are providing a test mask wave.

/USER=*pWave* Provides a user transformation that will be applied to the input testWave in order to create the trasnsformed image. *pWave* must be a double precision wave which contains the same number of rows as W_RegParams.

Note: If you use a previously created W_RegParams make sure to change its name as it is overwritten by the operation.

If *pWave* has only one column and testWave contains multiple layers, then the same transformation applies to all layers. If *pWave* contains more than one column, then each layer of testWave is processed with corresponding column. If there are more layers than columns the first column is used in place of the missing columns.

/ZMAN Modifies the test and reference data by subtracting their mean values prior to optimization.

Details

ImageRegistration will register images that have sufficiently similar features. It will not work if key features are too different. For example, ImageRegistration can handle two images that are rotated relative to each other by a few degrees, but cannot register images if the relative rotation is as large as 45 degrees. The algorithm is capable of subpixel resolution but it does not handle large variations between the test image and the reference image. If the centers of the two images are too far from each other, you should first try ImageRegistration using /CON=0 to remove the translation offset before proceeding with a finer registration of details.

The algorithm is based on an iterative processing that proceeds from coarse to fine detail. Optimization uses a modified Levenberg-Marquardt algorithm and produces an affine transformation for the relative rotation

and translation as well as for isometric scaling and contrast adjustment. The algorithm is most effective with square images where the center of rotation is not far from the center of the image.

When using gravity for convergence, skew parameters can't be evaluated (only translation is supported). Skew and isoscaling are mutually exclusive options. Mask waves are defined to have zero entries for pixels outside the region of interest and nonzero entries otherwise. If a mask is not provided, every pixel is used.

ImageRegistration creates the waves M_RegOut and M_RegMaskOut, which are both single precision waves. In addition, the operation creates the wave W_RegParams which stores 20 double precision registration parameters. M_RegOut contains the transformed (registered) test image and M_RegMaskOut contains the transformed mask (which is not affected by mask combination). ImageRegistration ignores wave scaling; images are compared and registered based on pixel values only.

The results printed in the history include:

dx, dy, dz	translation offsets measured in pixels.
a*ij*	Elements in the skewing transformation matrix.
phi	Rotation angle in degrees about the X-axis. Zero for 2D waves.
tht	Rotation angle in degrees about the Y-axis. Zero for 2D waves.
psi	Rotation angle in degrees about the Z-axis.
det	Absolute value of determinant of the skewing matrix (a*ij*).
err	Mean square error defined as $\frac{1}{N}\sum(x_i - y_i)^2$, where x_i is the original pixel value, y_i the computed value, and N is the number of pixels.
snr	Signal to noise ratio in dB. It is given by: $10 \cdot \log\left(\frac{\sum inVal^2}{\sum(x_i - y_i)^2}\right)$.

These parameters are stored in the wave W_RegParams (or M_RegParams in the case of registering a stack). Dimension labels are used to describe the contents of each row of the output wave. Each column of the wave consists of the following rows (also indicated by dimension labels):

Point	Contents	Point	Contents	Point	Contents	Point	Content
0	dx	6	a21	12	gamma	17	origin_x
1	dy	7	a22	13	phi	18	origin_y
2	dz	8	a23	14	theta	19	origin_z
3	a11	9	a31	15	psi	21	MSE
4	a12	10	a32	16	lambda	21	SNR
5	a13	11	a33				

You can view the output waves with dimension labels by executing:

```
Edit W_RegParams.ld
```

See Also

The **ImageInterpolate Warp** operation.

References

The ImageRegistration operation is based on an algorithm described by:

Thévenaz, P., and M. Unser, A Pyramid Approach to Subpixel Registration Based on Intensity, *IEEE Transactions on Image Processing*, *7*, 27-41, 1998.

ImageRemoveBackground

ImageRemoveBackground /R=*roiWave* [*flags*] *srcWave*

The ImageRemoveBackground operation removes a general background level, described by a polynomial of a specified order, from the image in *srcWave*. The result of the operation are stored in the wave M_RemovedBackground.

Flags

/F — Computes only the background surface fit. Will only store the resulting fit in M_RemovedBackground. This will not subtract the fit from the image.

/O — overwrites the original wave.

/P=*polynomial order* — Specifies the order of the polynomial fit to the background surface. If omitted, the order is assumed to be 1.

/R=*roiWave* — Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u), which has the same number of rows and columns as the image wave. The ROI itself is defined by the entries/pixels whose value are 0. Pixels outside the ROI can take any nonzero value. The ROI does not have to be contiguous. See **ImageGenerateROIMask** for more information on creating ROI waves.

/W — Specifies that polynomial coefficients are to be saved in the wave W_BackgroundCoeff.

Details

The identification of the background is done via the ROI wave. The ROI waves should consist of the value 1 where the user wants to designate a pixel as a background and 0 otherwise.

The operation first performs a polynomial fit to the points designated by the ROI wave using the specified polynomial order. A polynomial of order N corresponds to the function:

$$F_N(x,y) = \sum_{m=0}^{N} \sum_{n=0}^{m} c_{nm} x^{m-n} y^{n} .$$

Using the polynomial fit, a surface corresponding to the polynomial is subtracted from the source wave and the result is saved in M_RemovedBackground, unless the /O flag is used, in which case the original wave is overwritten.

Use the /W flag if you want polynomial coefficients to be saved in the W_BackgroundCoeff wave. Coefficients are stored in the same order as the terms in the sums above.

If you do not specify the polynomial order using the /P flag, the default order is 1, which means that the operation subtracts a plane (fitted to the ROI data) from the source image.

Note, if the image is stored as a wave of unsigned byte, short, or long, you might consider converting it into single precision (using `Redimension/S`) before removing the background. To see why this is important, consider an image containing a region of pixels equal to zero and subtracting a background plane corresponding to a nonconstant value. After subtraction, at least some of the pixels in the zero region should become negative, but because they are stored as unsigned quantities, they appear incorrectly as large values.

See Also

The **ImageGenerateROIMask** operation for creating ROIs. For examples see **Background Removal** on page III-323.

ImageRestore

ImageRestore [*flags*] srcWave=*wSrc*, psfWave=*wPSF* [, relaxationGamma=*h*, startingImage=*wRecon*]

The ImageRestore operation performs the Richardson-Lucy iterative image restoration.

Flags

/DEST=*destWave* — Specifies the desired output wave.
If /DEST is omitted, the output from the operation is stored in the wave M_Reconstructed in the current data folder.

/ITER=*iterations* — Specifies the number of iterations. The default number of iterations is 100.

/Z Do not report errors.

Parameters

psfWave=*wPSF* Specifies a known point spread function. *wPSF* must be a 2D (square NxN) wave of the same numeric type as *wSRC*. N must be an odd number greater than 1.

relaxationGamma=*h* Specifies positive power gamma of in the relaxation mapping (see Details).

startingImage=*wRecon* Use this keyword to specify a starting image that could be for example the output from a previous call to this operation. *wRecon* must have the same dimensions as *wSRC* and the same numeric type.

You must make sure that *wRecon* is not the user-specified or the default destination wave of the operation.

srcWave=*wSrc* Specifies the degraded image which must be a 2D single-precision or double-precision real wave.

Details

ImageRestore performs the Richardson-Lucy iteration solution to the deconvolution of an image. The input consists of the degraded image and point spread function as well as the desired number of iterations.

The operation allows you to apply additional iterations by setting the starting image to the restored output wave from a previous call to ImageRestore using the startingImage keyword. If startingImage is omitted, the starting image is created by ImageRestore with each pixel set to the value 1.

In the case of stellar images it may be useful to apply a relaxation step that involves scaling the correction evaluated at each iteration by

$$factor(v) = \sin\left(\frac{\pi}{2}\frac{v - v_{\min}}{v_{\max} - v_{\min}}\right)^{\gamma},$$

where v is pixel value, vmax and vmin are the maximum and minimum level pixels in the image and gamma is the user-specified relaxationGamma.

References

W.H. Richardson, "Bayesian-Based Iterative Method of Image Restoration". *JOSA 62, 1*: 55-59, 1972.

L.B. Lucy, "An iterative technique for the rectification of observed distributions", *Astronomical Journal 79, 6*: 745-754, 1974.

ImageRotate

**`ImageRotate [`*`flags`*`] `*`imageMatrix`*

The ImageRotate operation rotates the image clockwise by *angle* (degrees) or counter-clockwise if /W is used.

The resulting image is saved in the wave M_RotatedImage unless the /O flag is specified. The size of the resulting image depends on the angle of rotation.

The portions of the image corresponding to points outside the domain of the original image are set to the default value 64 or the value specified by the /E flag.

You can apply ImageRotate to 2D and 3D waves of any data type.

Flags

/A=*angle* Specifies the rotation angle measured in degrees in a clockwise direction. For rotations by exactly 90 degrees use /C or /W instead.

/C Specifies clockwise rotation.

Clockwise is the default direction so the rotation will be clockwise whether you use /C or not, so long as you do not use /W.

/E= *val* Specifies the value for pixels that are outside the domain of the original image. By default pixels are set to 64. If you specify `/E=(NaN)` and your data is of type char, short, or long, the operation sets the external values to 64.

/F Rotates image by 180 degrees.

/H	Flip the image horizontally.
/O	Overwrites the original image with the rotated image.
/Q	Quiet mode. Without this flag the operation writes warnings in the history window.
/S	Uses source image wave scaling to preserve scaling and relative locations of objects in the image for rotation angles that are multiples of 90 degrees.
/V	Flip the image vertically.
/W	Specifies counter-clockwise rotation.
/Z	Ignore errors.

See Also

The **MatrixTranspose** operation.

ImageSave

ImageSave [***flags***] ***waveName*** [[**as**]***fileNameStr***]

The ImageSave operation saves the named wave as an image file.

ImageSave requires that QuickTime be installed when saving formats other than TIFF and raw PNG.

To make your command more portable, we recommend using the /IGOR flag which tells ImageSave to use internal Igor routines rather than QuickTime, if possible. Currently this flag affects the TIFF format only.

Parameters

The file to be written is specified by *fileNameStr* and /P=*pathName* where pathName is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

On Macintosh, when saving in a format that requires QuickTime, the name of the file to be written is limited to 31 characters because Igor calls Apple routines that have this limit.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

If you want to force a dialog to select the file, omit the *fileNameStr* parameter or specify "" for *fileNameStr* or use the /I flag. The "as" keyword before *fileNameStr* is optional.

Flags

/D=*depth*	Specifies color depth in bits-per-pixel. Integer values of 1, 8, 16, 24, 32, and 40 are supported. A *depth* of 40 specifies 8-bit grayscale; a *depth* of 8 saves the file as 8-bits/pixel with a color lookup table. If /D is omitted, it acts like /D=8. Saving with a color table may cause some loss of fidelity. See also the discussion of saving TIFF files below.
/F	Saves the wave as single precision float. The data is not normalized. Applies only to TIFF files.
/I	Interactive mode. Forces ImageSave to display a Save File dialog, even if the file and folder location are fully specified.
/IGOR	Tells ImageSave to use Igor's internal code if possible. This flag is recommended to create a command that works if QuickTime is not available. /IGOR currently affects saving as TIFF only. The /S, /U and /WT flags also force the use of Igor's internal code.
/O	Overwrites the specified file if it exists. If /O is omitted and the file exists, ImageSave displays a Save File dialog.
/P=*pathName*	Specifies the folder in which to save the image file. *pathName* is the name of an existing symbolic path.
/Q=*quality*	Specifies the quality of the compressed image. *quality* is a number between 0 and 1, with 1 being the highest quality. This is only applicable when saving in formats with lossy compression like JPEG.
/S	Saves as stack. Applies only to 3D or 4D source waves saved as TIFF files.

/T=*fileTypeStr* Specifies the type of file to be saved.

fileTypeStr	Saved Image Format
"photoshop"	PhotoShop file
"bmp"	PC bitmap file
"jpeg"	JPEG file
"pict"	Macintosh picture file
"png"	PNG file
"rpng"	raw PNG file (see **Saving as Raw PNG**)
"sgi"	Silicon Graphics file
"targa"	Targa file
"tiff"	TIFF file

The default file types, used if /T is omitted, are "pict" on the Macintosh and "bmp" on Windows.

/U Prevents normalization. This works when saving TIFF only. See **Normalization** below.

/WT=*tagWave* Saves TIFF files with file tags as specified by *tagWave*, which is a text wave consisting of a row and 5 columns for each tag (see description of the /RAT flag under **ImageLoad**). It ignores any information in the second column but will write the tags sequentially (only in the first Image File Directory (IFD) if there is more than one image). If the fourth column contains a negative number, there must be a wave, whose name is given in the fifth column of *tagWave*, in the same data folder as *tagWave*. You must make sure that: (1) tag numbers are legal and do not conflict with any existing tags; (2) the data type and data size are consistent with the amount of data saved in external waves.

/Z No error reporting.

Saving as Raw PNG

The rpng image format requires a wave in 8- or 16-bit unsigned integer format with 1 to 4 layers. Use one layer for grayscale, 3 layers for rgb color, and the extra layer for an alpha channel. If X or Y scaling is not unity, they both must be valid and must be either inches per pixel or meters per pixel. If the units are not inches they are taken to be meters.

Normalization

Depending on the data type of your wave and the depth of the image file being created, ImageSave may save a "normalized" version of your data. The normalized version is scaled to fit in the range of the image file data type. For example, if you save a 16-bit Igor wave containing pixel values from -1000 to +1000 in an 8-bit grayscale TIFF file, ImageSave will map the wave values -1000 and +1000 to the file values 0 and 255 respectively.

When saving as greater than 8 bits, Igor normalizes to 65535 as the full-scale value.

When saving as floating point, no normalization is done. Normalization is also not done when saving 8-bit wave data to an 8-bit image file.

The /U flag disables normalization but works only when saving as TIFF.

If you use the /U flag, Igor will save unnormalized data. If the wave data exceeds the range supported by the file data format, the saved file data will be invalid.

Saving as TIFF

Igor can create four different categories of TIFF files:

Grayscale image

Grayscale image with color table (requires QuickTime)

RGB image (red, green and blue components for each pixel)

Stack of grayscale images

Which ImageSave flags you need to use depends on the nature of your data and on your goal, as described in the following sections.

Saving a Simple 2D Wave as TIFF

If your Igor data is in the form of a simple 2D wave, you can save it as a grayscale image with or without a color table.

Use /D=8 to save as 8 bits with a color table. Normalization is done except if the wave data is 8 bits. This requires QuickTime - do not use the /IGOR flag. The colors in the color table will all be shades of gray.

Use /D=40 to save as 8 bits without a color table. Normalization is done except if the wave data is 8 bits.

Use /D=16 to save as 16 bits with normalization.

Use /F to save as single-precision floating point without normalization. Many programs can not read this format.

Use /U to prevent normalization.

Use /IGOR to guarantee that Igor's internal TIFF routines will be used rather than QuickTime. Igor's internal routines do not save color tables.

Saving an Igor RGB Wave as TIFF

If your Igor data is an RGB wave (a 3D wave with exactly 3 layers), you can save it as an RGB image, a grayscale image or a grayscale image with a color table.

When saving as RGB, ImageSave does not perform normalization. However, if the wave data type is greater than 8 bits and you are saving to 8 bits, ImageSave divides the wave data by 256. No other normalization or scaling is done.

Use /D=8 to save as a grayscale image with a color table. Normalization is done except if the wave data is 8 bits. This requires QuickTime - do not use the /IGOR flag.

Use /D=40 to save as grayscale image without a color table. Normalization is done except if the wave data is 8 bits.

Without /D=8 or /D=40, the data is saved as RGB (three components per pixel - one for red, one for green and one for blue). The rest of this section applies when saving as RGB only.

Use /D=24 or /D=32 to save as 24 bits-per-pixel (8 bits-per-component) without normalization.

Use /D=16 to save as 48 bits-per-pixel (16 bits-per-component) without normalization.

Use /F to save as 96 bits-per-pixel (32 bits-per-component) without normalization. Many programs can not read this format.

Use /U to prevent normalization.

Saving 3D or 4D Waves as a Stack of TIFF Images

If your Igor data is a 3D wave other than an RGB wave or a 4D wave, you can save it as a stack of grayscale images without a color map.

Use /S to indicate that you want to save a stack of images rather than a single image from the first layer of the wave.

Use /D=8 to save as 8 bits. Normalization is done except if the wave data is 8 bits.

Use /D=16 to save as 16 bits with normalization.

Use /F to save as single-precision floating point without normalization. Many programs can not read this format.

Use /U to prevent normalization.

Stacked images are normalized on a layer by layer basis. If you want to have uniform scaling and normalization you should convert your wave to the file data type before executing ImageSave.

See Also

The **ImageLoad** operation for loading image files into waves.

ImageSeedFill

ImageSeedFill [*flags*] [*keyword*], **seedX=***xLoc*, **seedY=***yLoc*, **target=***setValue*, **srcWave=***srcImage*

The ImageSeedFill operation takes a seed pixel and fills a contiguous region with the target value, storing the result in the wave M_SeedFill. The filled region is defined by all contiguous pixels that include the seed

pixel and whose pixel values lie between the specified minimum and maximum values (inclusive). ImageSeedFill works on 2D and 3D waves.

Parameters

keyword is one of the following names:

adaptive=*factor* Invokes the adaptive algorithm where a pixel or voxel is accepted if, in addition, its value satisfies: $|val - avg| < factor \cdot stdv$.

Here *val* is the value of the pixel or voxel in question, *avg* is the average value of the pixels or voxels in the neighborhood and *stdv* is the standard deviation of these values. By choosing a small *factor* you can constrain the acceptable values to be very close to the neighborhood average. A large *factor* allows for more deviation assuming that the *stdv* is greater than zero.

This requirement means that a connected pixel has to be between the specified minimum and maximum value **and** satisfy the adaptive relationship. In most situations it is best to set wide limits on the minimum and maximum values and allow the adaptive parameter to control the local connectivity.

fillNumber=*num* Specifies the number, in the range 1 to 26, of voxels in each 3x3x3 cube that belong to the set. If fillNumber is exceeded, the operation fills the remaining members of the cube. If you do not specify this keyword, the operation does not fill the cube. Used only in the fuzzy algorithm.

fuzzyCenter=*fcVal* Specifies the center value for the fuzzy probability with the fuzzy algorithm (see **Details**). The default value is 0.25. Its standard range is 0 to 0.5, although interesting results might be obtained outside this range.

fuzzyProb=*fpVal* Specifies a probability threshold that must be met by a voxel to be accepted to the seeded set. The value must be in the range 0 to 1. The default value is 0.75.

fuzzyScale=*fsVal* Determines if a voxel is to be considered in a second stage using fuzzy probability. *fsVal* must be nonzero in order to invoke the fuzzy algorithm. The scale is used in comparing the value of the voxel to the value of the seed voxel. The scale should normally be in the range 0.5 to 2.0.

fuzzyWidth=*fwVal* Defines the width of the fuzzy probability distribution with the fuzzy algorithm (see **Details**). In most situations you should not need to specify this parameter. The default value is 1.

min=*minval* Specifies the minimum value that is accepted in the seeded set. Not needed when using fuzzy algorithm.

max=*maxval* Specifies the maximum value that is accepted to the seeded set. Not needed when using the fuzzy algorithm.

seedP=row Specifies the integer row location of the seed pixel or voxel. This avoids roundoff issues when srcWave has wave scaling. You must provide either seedP or seedX with all algorithms. It is sometimes convenient to use this with cursors e.g., `seedP=pcsr(a)`.

seedQ=col Specifies the integer column location of the seed pixel or voxel. This avoids roundoff difficulties when srcWave has wave scaling. You must provide either seedQ or seedY with all algorithms.

seedR=layer Specifies the integer layer position of the seed voxel. When srcWave is a 3D wave you must use either seedR or seedZ.

seedX=*xLoc* Specifies the pixel or voxel index. If srcWave has wave scaling, seedX must be expressed in terms of the scaled coordinate. This keyword or seedP is required with all algorithms.

seedY=*yLoc* Specifies the pixel or voxel index. If srcWave has wave scaling, seedY must be expressed in terms of the scaled coordinate. This keyword or seedQ is required with all algorithms.

seedZ=*zLoc* Specifies the voxel index. If srcWave has wave scaling, seedZ must be expressed in terms of the scaled coordinate. You must use this keyword or seedR whenever srcWave is 3D.

srcWave=*srcImage* Specifies the source image wave.

target=*val* Sets the value assigned to pixels or voxels that belonging to the seeded set.

Flags

/B=*bValue* Specifies the value assigned to pixels or voxels that do not belong to the fuzzy set.

/C	Uses 8-connectivity where a pixel can be connected to any one of its neighbors and with which it shares as little as a single boundary point. The default setting is 4-connectivity where pixels can be connected if they are neighbors along a row or a column. This has no effect in 3D, where 26-connectivity is the only option.
/K=*killCount*	Terminates the fill operation after *killCount* elements have been accepted.
/O	Overwrites the source wave with the output (2D only).
/R=*roiWave*	Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u), that has the same number of rows and columns and layers as the image wave. The ROI itself is defined by the entries/pixels whose value are 0. Pixels outside the ROI can take any nonzero value. The ROI does not have to be contiguous. See ImageGenerateROIMask for more information on creating ROI waves.

Details

In two dimensions, the operation takes a seed pixel, optional minimum and maximum pixel values and optional adaptive coefficient. It then fills a contiguous region (in a copy of the source image) with the target value. There are two algorithms for 2D seed fill. In direct seed fill (only min, max, seedX and seedY are specified) the filled region is defined by all contiguous pixels that include the seed pixel and whose pixel values lie between the specified minimum and maximum values (inclusive). In adaptive fill, there is an additional condition for the pixel or voxel to be selected, which requires that the pixel value must be within the standard deviation of the average in the 3x3 (2D) or 3x3x3 (3D) nearest neighbors. If you do not specify the minimum and maximum values then the operation selects only values identical to that of the seed pixel.

In 3D, there are three available algorithms. The direct seed fill algorithm uses the limits specified by the user to fill the seeded domain. In adaptive seed fill the algorithm requires the limits as well as the adaptive parameter. It fills the domain by accepting only voxels that lie within the adaptive factor times the standard deviation of the immediate voxel neighborhood. To invoke the third algorithm you must set fuzzyScale to a nonzero value. The fuzzy seed fill uses two steps to determine if a voxel should be in the filled domain. In the first step the value of the voxel is compared to the seed value using the fuzzy scale. If accepted, it passes to the second stage where a fuzzy probability is calculated based on the number of voxels in the 3x3x3 cell which passed the first step together with the user-specified probability center (fuzzyCenter) and width (fuzzyWidth). If the result is greater than fuzzyProb, the voxel is set to belong to the filled domain.

If the /O flag is not specified, the result is stored in the wave M_SeedFill.

If you specify a background value with the /B flag, the resulting image consists of the background value and the target value in the area corresponding to the seed fill. Although the wave is now bi-level, it retains the same number type as the source image.

ImageSeedFill returns a "bad seed pixel specification" if the seed pixel location derived from the various keywords above satisfies one or more of the following conditions:

1. The computed integer pixel/voxel is outside the image.
2. The value stored in the computed integer pixel/voxel location does not satisfy the min/max or fuzzy conditions. This is the most common condition when srcWave has wave scaling. To avoid this difficulty you should use the keywords seedP, seedQ, and seedR.

Examples

Using Cursor A position and value to supply parameter inputs for a 2D seedFill (**Warning**: command wrapped over two lines):

```
ImageSeedFill
seedP=pcsr(a),seedQ=qcsr(a),min=zcsr(a),max=zcsr(a),target=0,srcwave=image0
```

Using the fuzzy algorithm for a 3D wave (**Warning**: command wrapped over two lines):

```
ImageSeedFill seedX=232,seedY=175,seedZ=42,target=1,fillnumber=10,fuzzycenter=.25,
fuzzywidth=1,threshold=1,fuzzyprob=0.4,srcWave=ddd
```

See Also

For an additional example see **Seed Fill** on page III-321. To display the result of the operation for 3D waves it is useful to convert the 3D wave M_SeedFill into an array of quads. See **ImageTransform vol2surf**.

ImageSnake

ImageSnake [***flags***] ***srcWave***

The ImageSnake operation creates or modifies an active contour/snake in the grayscale image srcWave. The operation iterates to find the "lowest total energy" snake. The energy is defined by a range of optional flags,

each corresponding to an individual term in the total energy expression. Iterations terminate by reaching the maximum number of iterations, when the snake does not move between iterations or when the user aborts the operation.

Flags

/ALPH=*alpha* Sets the coefficient of the energy term arising from the "tightness" of the snake.

/BETA=*beta* Sets the coefficient of the energy term corresponding to curvature of the snake. A high value for beta makes the snake more rounded.

/DELT=*delta* Sets the coefficient of the repulsion energy. A high value of *delta* keeps nonconsecutive snake points far from each other.

/EPS=*num* Sets the maximum number of vertices which are allowed to move in one iteration. If the number of vertices which move during an iteration is smaller than *num* then iterations terminate.

/EXEF=*eta* Sets the coefficient of the optional external energy component. By default this value is set to zero and there is no external energy contribution to the snakes energy. Note, this component is referred to as "external" because it is completely up to the user to specify both its coefficient and the value associated with each pixel. It should not be confused with what is called external snake energy in the literature, which usually applies to energy proportional to the gradient image (see /GAMM and /GRDI).

/EXEN=*wave* Specify a wave that contains energy values that will be added to the snakes energy calculation. The wave must have the same dimensions as *srcWave* and must be single precision float. Each pixel value corresponds to user defined energy which will be multiplied by the /EXEF coefficient and added to the sum which the snake minimizes. Note that when /EXEF is set to zero this component is ignored. An external energy wave may be useful, for example, if you want to attract the snake to the picture boundaries. In that case you can set:

```
Duplicate/O srcWave,extWave
Redimension/S extWave
Variable rows=DimSize(srcWave,0)-1
Variable cols=DimSize(srcWave,1)-1
extWave=(p==0 || q==0 || p==rows || q==cols) ? 0:1
```

/GAMM=*gamma* Sets the coefficient of the energy term corresponding to the gradient. A high value of gamma makes the snake follow lines of high image gradient.

/GRDI=*gWave* Specify the gradient image. This wave must have the same dimensions as *srcWave* and it must be single precision float. The wave corresponds to the quantity `abs(grad(gauss**srcWave))`, where grad is the gradient operator and ** denotes convolution of the source wave with a Gaussian kernel. It is best to run the operation the first time without specifying this wave. When the operation executes, it creates the wave M_GradImage which can then be used in subsequent executions of this operation. If you want to modify the wave to express some other form of energy that you want the operation to minimize, you should use the /EXEN and /EXEF flags.

/ITER=*iterations* Sets the maximum number of iterations. Convergence can be achieved if the value specified by /EPS is met. You can also terminate the process earlier via a user abort (Command-'.' *Macintosh* or Ctrl-Break *Windows*).

/N=*snakePts* Specify the number of vertices in the snake or the number of snake points. Note that if you are providing snake waves in /SX and /SY, you do not need to specify this flag. If you do not specify this flag the default value is 40.

/SIG=*sigma* Sets the size of the Gaussian kernel that is used to convolve the input image when creating the gradient image. Note that you do not need to use this flag if you provide a gradient image. *sigma* is 3 by default. You can use larger odd integers for larger Gaussian kernels which would correspond to a stronger blur.

/STRT={*centerX*, *centerY*, *radius*}
Sets the starting snake to be a circle with the given center and radius. If you use this flag you should also provide the number of snake points using the /N flag.

/STEP=*pixels* Sets the maximum radius of search. By default the radius of the search is 6 pixels and the search follows a clockwise pattern from radius of 1 pixel to maximum radius specified by this flag. Note: the search radius should be smaller than the dimension of a typical feature in the image. If the radius is larger the snake may encompass more

than one object. Larger radius is also less efficient because many of the pixels in that range would result in a snake that crosses itself and hence get rejected in the process.

/SX=*xSnake* Provide an X-wave for the starting snake. You must also provide an appropriate Y-wave using /SY.

/SY=*ySnake* Provide a Y-Wave for the starting snake. Must work in combination with /SX.

/UPDM=*mode* Sets the update mode using any combination of the following:

Value	Update
0	Once when the operation terminates.
1	Once at the end of every iteration.
2	Once after every snake vertex moves.
4	Once for every search position.

/Q Quiet mode; don't print information in the history.

/Z Don't report any errors.

Details

A snake is a two-dimensional, usually closed, path drawn over an image. The snake is described by a pair of XY waves consisting of N vertices (sometimes called "snake elements" or "snaksels"). In this implementation it is assumed that the snake is closed so that the last point in the snake is connected to the first. Snakes are used in image segmentation, when you want to automatically select a contiguous portion of the plane based on some criteria. Unlike the classic contours, snakes do not have to follow a constant level. Their structure (or path) is found by associating the concept of energy with every snake configuration and attempting to find the configuration for which the associated energy is a minimum. The search for a minimum energy configuration is usually time consuming and it strongly depends on the format of the energy function and the initial conditions (as defined by the starting snake). The operation computes the energy as a sum of the following 5 terms:

1. The coefficient alpha times a sum of absolute deviations from the average snake segment length. This term tends to distribute the vertices of the snake at even intervals.
2. The coefficient beta times a sum of energies associated with the curvature of the snake at each vertex.
3. The coefficient gamma times a sum of energies computed from the negative magnitude of the gradient of a Gaussian kernel convolved with the image. This term is usually referred to in the literature as the external energy and usually drives the snake to follow the direction of high image gradients.
4. The coefficient delta times a repulsion energy. Repulsion is computed as an inverse square law by adding contributions from all vertices except the two that are immediately connected to each vertex. This energy term is designed to make sure that the snake does not fold itself into "valleys".
5. The coefficient eta times the sum of values corresponding to the positions of all snake vertices in the wave you provide in /EXEN.

The energy calculation skips all terms for which the coefficient is zero. In addition there is a built-in scan which adds a very high penalty for configurations in which the snake crosses itself.

ImageStats

ImageStats [**/BRXY={*xWave,yWave*}** ***imageWave***

ImageStats [**/M=*val*/P=*planeNumber***]**/R=*roiWave* *imageWave***

ImageStats [**/M=*val*/P=*planeNumber***]**/G={*startP,endP,startQ,endQ*}** ***imageWave***

ImageStats [**/M=*val*/P=*planeNumber***]**/GS={*sMinRow,sMaxRow,sMinCol,sMaxCol*}** ***imageWave***

ImageStats [**/M=*val*/P=*planeNumber***]** ***imageWave***

The ImageStats operation calculates wave statistics for specified regions of interest in a matrix wave. The operation applies to image pixels whose corresponding pixels in the ROI wave are set to zero. It does not print any results in the history area.

Flags

/BEAM Computes the average, minimum, and maximum pixel values in each layer of a 3D wave and 2D ROI. Output is to waves W_ISBeamAvg, W_ISBeamMax, and W_ISBeamMin in the current data folder. Use /RECT to improve efficiency for simple ROI domains. V_ variable results correspond to the last evaluated layer of the 3D wave. Do not use /G, /GS, or /P with this flag. Set /M=1 for maximum efficiency.

/BRXY={*xWave*, *yWave*}

Use this option with a 3D imageWave. It provides a more efficient method for computing average, minimum and maximum values when the set of points of interest is much smaller than the dimensions of an image.

Here *xWave* and *yWave* are 1D waves with the same number of points containing XY integer pixel locations specifying arbitrary pixels for which the statistics are calculated on a plane by plane basis as follows:

$$\text{W\_ISBeamAvg[k]}=\frac{1}{\text{n}}\sum_{i=1}^{n}\text{Im}\,age[xWave[i]][yWave[i]].$$

Pixel locations are zero-based; non-integer entries may produce unpredictable results.

The calculated statistics for each plane are stored in the current data folder in the waves W_ISBeamAvg, W_ISBeamMax and W_ISBeamMin.

Note: This flag is not compatible with any other flag except /BEAM.

/G={*startP*, *endP*, *startQ*, *endQ*}

Specifies the corners of a rectangular ROI. When this flag is used an ROI wave is not required. This flag requires that *startP* ≤ *endP* and *startQ* ≤ *endQ*. When the parameters extend beyond the image area, the command will not execute and V_flag will be set to -1. You should therefore verify that V_flag=0 before using the results of this operation.

/GS={*sMinRow*,*sMaxRow*,*sMinCol*,*sMaxCol*}

Specifies a rectangular region of interest in terms of the scaled image coordinates. Each one of the 4 values will be translated to an integer pixel using truncation.

This flag, /G, and an ROI specification are mutually exclusive.

/M=*val* Calculates the average and locates the minimum and the maximum in the ROI when /M=1. This will save you the computation time associated with the higher order statistical moments.

/P=*planeNumber* Restricts the calculation to a particular layer of a 3D wave. By default, *planeNumber*= -1 and only the first layer of the wave is processed.

/R=*roiWave* Specifies a region of interest (ROI) in the image. The ROI is defined by a wave of type unsigned byte (/b/u), which has the same number of rows and columns as the image wave. The ROI itself is defined by the entries/pixels whose value are 0. Pixels outside the ROI can take any nonzero value. The ROI does not have to be contiguous. See **ImageGenerateROIMask** for more information on creating ROI waves.

/RECT={*minRow*, *maxRow*, *minCol*, *maxCol*}

Limits the range of the ROI to a rectangular pixel range with /BEAM. It does not affect complex-valued wave statistics.

Details

The image statistics are returned via the following variables:

`V_adev` Average deviation of pixel values.

`V_avg` Average of pixel values.

`V_kurt` Kurtosis of pixel values.

`V_min` Minimum pixel value.

`V_minColLoc` Specifies the location of the column in which the minimum pixel value was found or the first eligible column if no single column was found.

`V_minRowLoc` Specifies the location of the row in which the minimum pixel value was found or the first eligible row if no single minimum was found.

`V_max` Maximum pixel value.

V_maxColLoc	Specifies the location of the column in which the maximum pixel value was found or the first eligible column if no single column was found.
V_maxRowLoc	Specifies the location of the row in which the maximum pixel value was found or the first eligible row if no single maximum was found.
V_npnts	Number of points in the ROI.
V_rms	Root mean squared of pixel values.
V_sdev	Standard deviation of pixel values.
V_skew	Skewness of pixel values.

Most of these statistical results are similarly defined as for the WaveStats operation. WaveStats will be more convenient to use when calculating statistics for an entire wave.

If *imageWave* is 4D it is often useful to use the reversible conversion

```
Redimension/N=(rows,cols,layers*chunks) ImageWave
```

which allows you to obtain the statistics for each layer and all chunks of the wave. To convert back to 4D, execute:

```
Redimension/N=(rows,cols,layers,chunks) ImageWave
```

See Also

The **ImageGenerateROIMask** and **WaveStats** operations. **ImageStats** on page III-315.

ImageThreshold

ImageThreshold [***flags***] ***imageMatrix***

The ImageThreshold operation converts a grayscale *imageMatrix* into a binary image. The operation supports all data types. However, the source wave must be a 2D matrix. If *imageMatrix* contains NaNs, the pixels corresponding to NaN values are mapped into the value 64. The values for the On and Off pixels are 255 and 0 respectively. The resulting image is stored in the wave M_ImageThresh.

Flags

/C — Calculates the correlation coefficient between the original image and the image generated by the threshold operation. The correlation value is printed to the history window (unless the /Q flag is specified), it is also stored in the variable V_correlation.

/I — Inverts values written to the image, i.e., sets to zero all pixels above threshold.

/M= *method* — Specifies the thresholding method. The calculated value will be printed to the history window (unless /Q is specified) and stored in the variable V_threshold.

- *method*= 1: Automatically calculate a threshold value using an iterative method.
- *method*=2: Image histogram is a simple bimodal distribution.
- *method*=3: Adaptive thresholding. Evaluates threshold based on the last 8 pixels in each row, using alternating rows. Note that this method is not supported when used as part of the operation **ImageEdgeDetection**.
- *method*=4: Fuzzy thresholding using entropy as the measure for "fuzziness".
- *method*=5: Fuzzy thresholding using a method that minimizes a "fuzziness" measure involving the mean gray level in the object and background.

By default *method*=0, in which case you must use the /T flag to specify a manually selected threshold.

/N — Sets the background level to 64 (i.e., NaN).

/O — Overwrites the original image with the calculated threshold image.

If you do not specify the /O flag, the threshold image is written into the wave M_ImageThresh.

/Q — Suppresses printing calculated correlation coefficients (/C) and calculated thresholds (/M) to the history window.

/R=*roiSpec* — Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u). The ROI wave must have the same number of rows and columns as the image wave. The ROI itself is defined by the entries/pixels whose values are 0. Pixels outside the ROI can take any nonzero value. The ROI does not have to be contiguous and can

take any arbitrary shape. See **ImageGenerateROIMask** for more information on creating ROI waves.

In general, the *roiSpec* has the form {*roiWaveName*, *roiFlag*}, where *roiFlag* can take the following values:

roiFlag=0: Set pixels outside the ROI to 0.

roiFlag=1: Set pixels outside the ROI as in original image.

roiFlag=2: Set pixels outside the ROI to NaN (=64).

By default *roiFlag* is set to 1 and it is then possible to use the /R flag using the abbreviated form /R=*roiWave*.

/T=*thresh* Sets the threshold value.

/W= *Twave* Sets the threshold intervals. Each interval is specified by a pair of values in the wave *Twave*. The first element in each pair is the low value and the second element is the high value. Pixel values that lie outside all the specified intervals are set to 0.

References

The automatic thresholding method (/M=1) is described in: T. W. Ridler and S. Calvard, *IEEE Transactions on Systems, Man and Cybernetics*, SMC-8, 630-632, 1978.

See Also

For usage examples see **Threshold Examples** on page III-300. The **ImageGenerateROIMask** and **ImageEdgeDetection** operations.

ImageTransform

ImageTransform [***flags***] ***Method imageMatrix***

The ImageTransform operation performs one of a number of transformations on *imageMatrix*. The result of using most keywords is a new wave stored in the current data folder. Most flags in this operation are exclusive to the keywords in which they are mentioned

Parameters

Method selects type of transform. It is one of the following names:

averageImage Computes an average image for a stack of images contained in 3D *imageMatrix*. The average image is stored in the wave M_AveImage and the standard deviation for each pixel is stored in the wave M_StdvImage in the current data folder. You can use this keyword together with the optional /R flag where a region of interest is defined by zero value points in a ROI wave. The operation sets to NaN the entries in M_AveImage and M_StdvImage that correspond to pixels outside the ROI. *imageMatrix* must have at least three layers.

backProjection Reconstructs the source image from a projection slice and stores the result in the wave M_BackProjection. The projection slice should either be a wave produced by the projectionSlice keyword of this operation or a wave that would be similarly scaled. The input must be a single or double precision real 2D wave. Row scaling must range symmetrically about zero. For example, if the reconstructed image is expected to have 256 rows then the row scaling of the input should be from -128 to 127. Similarly, the column scaling of the input should range between zero and π. An equivalent implementation as a user function is provided in the demo experiment. You can use this implementation as a starting point if you want to develop filtered back projection.

See the projectionSlice keyword and the RadonTransformDemo experiment. For algorithm details see the chapter "Reconstruction of cross-sections and the projection-slice theorem of computerized tomography" in Born and Wolf, 1999.

ccsubdivision Performs a Catmull-Clark recursive generation of B-spline surfaces. There are two valid inputs: triangular meshes or quad meshes.

Quad meshes are assumed to be in the form of a 3D wave where the first plane contains the X-values, the second the Y-values and the third the Z-values.

Triangle meshes are much more complicated to convert into face-edge-vertex arrays so they are less desirable. They are stored in a three column (triplet) wave where the first column corresponds to the X coordinate, the second to the Y coordinate and the third to the Z coordinate. Each triangle is described by three rows in the wave and common

	vertices have to be repeated so that each sequential three rows in the triplet wave correspond to a single triangle. You can also associate a scalar value with each vertex and it will be suitably interpolated as new vertices are computed and old ones are shifted. In this case the input source wave contains one more column in the case of a triplet wave or one more plane in the case of a quad wave. The scalar value is added everywhere as an additional dimension to the spatial part of any point calculation. You can specify the number of iterations using the /I flag. By default the operation executes a single iteration. The output is saved in a quad wave M_CCBSplines that consists of 4 columns. Each row corresponds to a Quad where the 3 planes contain the X, Y, and Z components. If you are using an optional scalar in the input, the scalar result is stored in the wave M_CCBScalar.
cmap2rgb	Converts an image and its associated colormap wave (specified using the /C flag) into an RGB image stored in a 3D wave M_RGBOut.
CMYK2RGB	Converts a CMYK image, stored as 4 layer unsigned byte wave, into a 3 layer, standard RGB image wave. The output wave is M_CMYK2RGB that is stored in the current data folder.
compress	Compresses the data in the *imageWave* using a nonlossy algorithm and stores it in the wave W_Compressed in the current data folder. The compressed wave includes all data associated with *imageWave* including its units and wavenote. Use the decompress keyword to recover the original wave. The operation supports all numeric data types.
convert2gray	Converts an arbitrary 2D wave into an 8-bit normalized 2D wave. The default output wave name is M_Image2Gray.
decompress	Decompresses a compressed wave. It saves a copy of the decompressed wave under the name W_DeCompressed in the current data folder.
extractSurface	Extracts values corresponding to a plane that intersects a 3D volume wave (*imageMatrix*). You must specify the extraction parameters using the /X flag. The volume is defined by the wave scaling of the 3D wave. The result, obtained by trilinear interpolation, is stored in the wave M_ExtractedSurface and is of the type NT_FP64. Points in the plane that lie outside the volume are set to NaN.
fht	Performs a Fast Hartley Transform subject to the /T flag. The source wave must be a 2D real matrix with a power of 2 number of rows and columns. Default output is saved in the double-precision wave M_Hartley in the current data folder. If you use the /O flag the result overwrites *imageMatrix* without changing the numeric type. If *imageMatrix* is single-precision float the conversion is straightforward. All other numeric types are scaled. Single- and double-byte types are scaled to the full dynamic range. 32 bit integers are scaled to the range of the equivalent 16 bit types (i.e., unsigned int is scaled to unsigned short range etc.). It does not support wave scaling or NaN entries.
fillImage	Fills a 2D target image wave with data from a 1D image wave (specified using /D). Both waves must be the same data type, and the number of data points in the target wave must match the number of points in the data wave. There are four fill modes that are specified via the /M flag. The operation supports all noncomplex numeric data types.
findLakes	Originally intended to identify lakes in geographical data, this operation creates a mask for a 2D wave for all the contiguous points whose values are close to each other. You can determine the minimum number of pixels within a contiguous region using the /LARA flag (default is 100). You can determine how close values must be in order to belong to a contiguous region using the /LTOL=*tolerance* flag (default is zero). You can also limit the search region using the /LRCT flag. Use the flag /LTAR=*target* to set the value of the masked regions. By default, the algorithm uses 4-connectivity when looking at adjacent pixels. You can set it to 8-connectivity using the /LCVT flag. The result of the operation is saved in the wave M_LakeFill. It has the same data type as the source wave and contains all the source values outside the masked pixels.
flipCols	Rearrange pixels by exchanging columns symmetrically about a center column or the center of the image (if the number of columns is even). The exchange is performed in place and can be reverted by repeating the operation. When working with 3D waves, use the /P flag to specify the plane that you want to operate on.
flipRows	Rearrange pixels in the image by exchanging rows symmetrically about the middle row or the middle of the image (if the image has an even number of rows). The exchange is

	performed in place and can be reverted by repeating the operation. When working with 3D waves, use the /P flag to specify the plane that you want to operate on.
flipPlanes	Rearrange data in a 3D wave by exchanging planes symmetrically about the middle plane. The operation is performed in place and can be reverted by repeating the operation.
fuzzyClassify	Segments grayscale and color images using fuzzy logic. Iteration stops when it reaches convergence defined by /TOL or the maximum number of iterations specified by /I. It is a good practice to specify the tolerance and the number of iterations. If the number of classes is small, the operation prints the class values in the history. Use /Q to eliminate printing and increase performance. Use /CLAS to specify the number of classes and optionally use /CLAM to modify the fuzzy probability values. Use /SEG to generate the segmentation image. The classes are stored in the wave W_FuzzyClasses in the current data folder and it will be overwritten if it already exists. When *imageMatrix* is a grayscale image, each class is a single wave entry. When *imageMatrix* is a RGB image, classes are stored consecutively in the wave W_FuzzyClasses. If you request more classes than are present in *imageMatrix*, you will likely find a degeneracy where the space of a data class is spanned by more than one class. It is a good idea to compute the Euclidean distance between every possible pair of classes and eliminate degeneracies when the distance falls below some threshold. Any real data type is allowed but values in the range [0,255] are optimal. You can segment 3D waves of more than 3 layers in which a class will be a vector of dimensionality equal to the number of layers in *imageMatrix*. For examples see Examples/Imaging/fuzzyClassifyDemo.pxp.
getBeam	Extracts a beam from a 3D wave. A "beam" is a 1D array in the Z-direction. If a row is a 1D array in the first dimension and a column is a 1D array in the second dimension then a beam is a 1D array in the third dimension. The number of points in a beam is equal to the number of layers in *imageWave*. Specify the beam with the /Beam={*row*,*column*} flag. It stores the result in the wave W_Beam in the current data folder. W_Beam has the same numeric type as *imageWave*. Use setBeam to set beam values. (See also, **MatrixOp** beam.)
getChunk	Extracts the chunk specified by chunk index /CHIX from *imageMatrix* and stores it in the wave M_Chunk in the current data folder. For example, if *imageMatrix* has the dimensions (10,20,3,10), the resulting M_Chunk has the dimensions (10,20,3). See also setChunk and insertChunk.
getCol	Extracts a 1D wave, named W_ExtractedCol, from any type of 2D or 3D wave. You specify the column using the /G flag. For a 3D source wave, it will use the first plane unless you specify a plane using the /P flag. *imageMatrix* can be real or complex. (See also putCol keyword, **MatrixOp** col.)
getPlane	Creates a new wave, named M_ImagePlane, that contains data in the plane specified by the /P flag. The new wave is of the same data type as the source wave. You can specify the type of plane using the /PTYP flag. (See also setPlane keyword, **MatrixOp**.)
getRow	Extracts a 1D wave, named W_ExtractedRow, from any type of 2D or 3D wave. You specify the row using the /G flag. For a 3D source wave, it will use the first plane unless you specify a plane using the /P flag. *imageMatrix* can be real or complex. (See also setRow keyword, **MatrixOp** row.)
Hough	Performs the Hough transform on the input wave. The result is saved to a 2D wave M_Hough in which the rows are the angle domain and the columns are the radial domain. It is assumed that the input wave is binary with background value of 0. See **Hough Transform** on page III-308.
hsl2rgb	Transforms a 3-plane HSL wave into a 3-plane RGB wave. If the source wave for this operation is of any type other than byte or unsigned byte, the HSL values are expected to be between 0 and 65535. For all source wave types the resulting RGB wave is of type unsigned short. The result of the operation is the wave M_HSL2RGB (of type unsigned word), where the RGB values are in the range 0 to 65535.
hslSegment	Creates a binary image of the same dimensions as the source image, in which all pixels that belong to all three of the specified Hue, Saturation, and Lightness ranges are set to 255 and the others to zero. You can specify the HSL ranges using the /H, /S, and /L

flags. Each flag takes as an argument either a pair of values or a wave containing pairs of values. You must specify the /H flag but you can omit the /S and /L flags in which case the default values (corresponding to full range 0 to 1) are used. *imageMatrix* is assumed to be an RGB image.

imageToTexture

Transforms a 2D or 3D image wave into a 1D wave of contiguous pixel components. The transformation is useful for creating an OpenGL texture (for Gizmo) or for saving a color image in a format requiring either RGB or RGBA sequences.

The /O flag does not apply to imageToTexture.

Use the /TEXT flag to specify the type of transformation. *imageMatrix* must be an unsigned byte wave. A 1D unsigned byte wave named W_Texture is created in the current data folder.

W_Texture's wave note is set to a semicolon-separated list of keyword -value pairs that can be parsed using **StringByKey** and **NumberByKey**:

Keyword	Information Following Keyword
WIDTHPIXELS	**DimSize**(imageMatrix,0) or truncated to nearest power of 2 if /TEXT value is odd
HEIGHTPIXELS	**DimSize**(imageMatrix,1) or truncated to nearest power of 2
LAYERS	**DimSize**(imageMatrix,2)
TEXTUREMODE	val parameter from /TEXT flag
SOURCEWAVE	**GetWavesDataFolder**(imageMatrix, 2)

insertChunk

Inserts a chunk (a 3D wave specified by the /D flag) into *imageMatrix* at chunk index specified by the /CHIX flag. The dimensions of the inserted chunk must match the first three dimensions of *imageMatrix*. The wave must also have the same numeric type. The 4th dimension of *imageMatrix* is incremented by 1 to accommodate the new data. See also getChunk and setChunk.

insertImage

Inserts the image specified by the flag /INSI into *imageMatrix* starting at the position specified by the flags /INSX and /INSY. If the *imageMatrix* is a 3D wave then it inserts the image in the layer specified by the /P flag. The inserted image and *imageMatrix* must be the same data type. The inserted data is clipped to the boundaries of *imageMatrix*.

insertXplane

Inserts a 2D wave as a new plane perpendicular to the X-axis in a 3D wave. The /P flag specifies the insertion point and the /INSW flag specifies the inserted wave. The 2D wave must be the same numeric data type as the 3D wave and its dimensions must be cols x layers of the 3D wave. For example, if the 3D wave has the dimensions (10x20x30) the 2D wave must be 20x30. If you do not use the /O flag, it stores the result in the wave M_InsertedWave in the current data folder.

insertYplane

Inserts a 2D wave as a new plane perpendicular to the Y-axis in a 3D wave. The /P flag specifies the insertion point and the /INSW flag specifies the inserted wave. The 2D wave must be the same numeric data type as the 3D wave and its dimensions must be rows x layers of the 3D wave. For example, if the 3D wave has the dimensions (10x20x30) the 2D wave must be 10x30. If you do not use the /O flag, it stores the result in the wave M_InsertedWave in the current data folder.

insertZplane

Inserts a 2D wave as a new plane perpendicular to the Z-axis in a 3D wave. The /P flag specifies the insertion point and the /INSW flag specifies the inserted wave. The 2D wave must be the same numeric data type as the 3D wave and its dimensions must be rows x cols of the 3D wave. For example, if the 3D wave has the dimensions (10x20x30) the 2D wave must be 10x20. If you do not use the /O flag, it stores the result in the wave M_InsertedWave in the current data folder. This keyword is included for completeness. You can accomplish the same task using InsertPoints.

Invert

Converts pixel values using the formula `newValue=255-oldValue`. Works on waves of any dimension, but only on waves of type unsigned byte. The result is stored in the wave M_Inverted unless specifying the /O flag.

indexWave — Creates a 1D wave W_IndexedValues in the current data folder containing values from *imageMatrix* that are pointed to by the index wave (see /IWAV). Each row in the index wave corresponds to a single value of *imageMatrix*. If any row does not point to a valid index (within the dimensions of *imageMatrix*), the corresponding value is set to zero and the operation returns an error. Indices are zero based integers; the operation does not support interpolation and ignores wave scaling.

JPEGQ — Generates a lossy JPEG-compressed image wave when used together with the /J flag. The compressed image wave is stored in the wave M_JPEGQ and the size of the equivalent image file is stored in the variable V_value.

matchPlanes — Finds pixels that match test conditions in all layers of a 3D wave. It creates a 2D unsigned byte output wave, M_matchPlanes, that is set to the values 0 and 255. A value of 255 indicates that the corresponding pixel has satisfied test conditions in all layers of the wave for which conditions were provided. Otherwise the pixel value is 0.

Test conditions are entered as a 2D wave using the /D flag. The condition wave must be double precision and it must contain the same number of columns as the number of layers in the 3D source wave. A condition for layer j of the source wave is specified by two rows in column j of the condition wave. The first row entry, say *A*, and the second row entry, say *B*, imply a condition on pixels in layer *j* such that $A \le x < B$. You can have more than one condition for a given layer by adding pairs of rows to the condition wave. For example, if you add in consecutive rows the values *C* and *D*, this implies the test:

$$\left(A \le x \le B\right) \| \left(C \le x \le D\right).$$

If you do not have any conditions for some layer, set its corresponding condition column to NaN. Similarly, if you have two conditions for the first layer and one condition for the second layer, pad the bottom of column 1 in the condition wave with NaNs. See Examples for use of this keyword to perform hue/saturation segmentation.

offsetImage — Shifts an image in the XY plane by dx, dy pixels (specified by the /IOFF flag). Pixels outside the shifted image will be set to the specified background value. The operation works on 2D waves or on 3D waves with the optional /P flag. When shifting a 3D wave with no specified plane, it creates a 3D wave with all planes offset by the same amount. The wave M_OffsetImage contains the result in the current data folder.

The /O flag is not supported with offsetImage.

padImage — Resizes the source image. When enlarged, values from the last row and column fill in the new area. The /N flag specifies the new image size in terms of the rows and columns change. The /W flag specifies whether data should be wrapped when padding the image. Unless you use the /O flag, the result is stored in the wave M_PaddedImage in the current data folder.

projectionSlice — Computes a projection slice for a parallel fan of rays going through the image at various angles. For every ray in the fan the operation computes a line integral through the image (equivalent to the sum of the line profile along the ray). The operation computes the line integrals for multiple fans defined by the number and position of the rays as well as the angle that they make with the positive X-direction. Use the /PSL flag to specify the projection parameters. The projection slice itself is stored in a 2D wave M_ProjectionSlice where the rows correspond to the rays and the columns correspond to the selected range of angles. The operation does not support wave scaling. If the source wave is 3D the projection slice currently supports slices that are perpendicular to the z-axis and specified by their plane number.

See the backProjection keyword and the RadonTransformDemo experiment. For algorithm details see the chapter "Reconstruction of cross-sections and the projection-slice theorem of computerized tomography" in Born and Wolf, 1999.

putCol — Sets a column of *imageMatrix* to the values in the wave specified by the /D flag. Use the /G flag to specify column number and the /P flag to specify the plane. Note that if there is a mismatch in the number of entries between the specified waves, the operation uses the smaller number. See also getCol keyword.

putRow — Sets a row of *imageMatrix* to the values in the wave specified by the /D flag. Use the /G flag to specify column number and the /P flag to specify the plane. Note that if there is

a mismatch in the number of entries between the specified waves, the operation uses the smaller number. See also getRow keyword.

rgb2cmap — Computes a default color map of 256 colors to represent the input RGB image. The colors are computed by clustering the input pixels in RGB space. The resulting color map is stored in the wave M_ColorIndex in the current data folder. The operation also saves the wave M_IndexImage which contains an index into the colormap that can be used to display the image using the commands:

```
NewImage M_IndexImage
ModifyImage M_IndexImage cindex= M_ColorIndex
```

To change the default number of colors use the /NCLR flag. When the number of colors are greater than 256, M_IndexImage will be a 16-bit unsigned integer or a 32 bit integer wave depending on the number. rgb2cmap supports input images in the form of 3D waves of type unsigned byte or single precision float. The floating point option may be used to input images in colorspaces that use signed numeric data.

rgb2gray — When the input *imageMatrix* is a 3D RGB wave, rgb2gray produces a 2D wave of type unsigned byte containing the grayscale representation of the input. By default, the operation stores the output in the wave M_RGB2Gray in the current data folder. The RGB values are converted into the luminance Y of the YIQ standard using:

```
Y = 0.299R + 0.587G + 0.114B
```

When the input imageMatrix is a 4D wave containing multiple (3 layer) RGB chunks, the conversion produces a 3D wave where each layer corresponds to the grayscale conversion of the corresponding chunk in the input wave. In this case the numeric type of the output is the same as that of the input but the conversion formula is the same. The /O flag is not supported when transforming a 4D RGB wave.

rgb2hsl — Converts an RGB image stored in a 3D wave into another 3D wave in which the three planes correspond to Hue, Saturation and Lightness in the HSL color model. Values of all components are normalized to the range 0 to 255 unless the /U flag is used or if the source wave is not 8-bit, in which case the range is 0 to 65535. The default output wave name is M_RGB2HSL.

rgb2i123 — Performs a colorspace conversion of an RGB image into the following quantities:

$$I_1 = |R-G|D$$
$$I_2 = |R-B|D \quad \text{where} \quad D = \frac{255}{|R-G|+|R-B|+|G-B|}.$$
$$I_3 = |G-B|D,$$

*I*1, *I*2, and *I*3 are stored in the wave M_I123 (using the same data type as the original RGB wave) in the current data folder. *I*1 is stored in the first layer, *I*2 in the second and *I*3 in the third. This color transformation is said to have useful applications in machine vision.

For more information see: Gevers and Smeulders (1999).

removeXplane — Removes one or more planes perpendicular to the X-axis from a 3D wave. The /P flag specifies the starting position. By default, it removes a single plane but you can remove more planes with the /NP flag. If you do not use the /O flag, it saves the result in the wave M_ReducedWave in the current data folder.

removeYplane — Removes one or more planes perpendicular to the Y-axis from a 3D wave. The /P flag specifies the starting position. By default, it removes a single plane but you can remove more planes with the /NP flag. If you do not use the /O flag, it saves the result in the wave M_ReducedWave in the current data folder.

removeZplane — Removes one or more planes perpendicular to the Z-axis from a 3D wave. The /P flag specifies the starting position. By default, it removes a single plane but you can remove more planes with the /NP flag. If you do not use the /O flag, it saves the result in the wave M_ReducedWave in the current data folder.

rgb2xyz — converts a 3D RGB image wave into a 3D wave containing the XYZ color space equivalent. The conversion is based on the D65 white point and uses the following transformation:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

The XYZ values are stored in a wave named "M_RGB2XYZ" unless the /O flag is used, in which case the source image is overwritten and converted into single precision wave (NT_FP32).

roiTo1D

Copies all pixels in an ROI and saves them sequentially in a 1D wave. The ROI is specified by /R. The ROI wave must have the same dimensions as *imageMatrix*. If *imageMatrix* is a 3D wave, the ROI must have as many layers as *imageMatrix*. The wave W_roi_to_1d contains the output in the current data folder, has the same numeric type as *imageMatrix*, and contains the selected pixels in a column-major order.

rotateCols

Rotates rows in place. This operation is analogous to the **Rotate** operation except that it works on images and rotates an integer number of rows. The number of rows is specified by the /G flag.

When *imageMatrix* contains multiple layers you can use the /P flag to specify the layer of the wave that will undergo rotation. By default, if you do not specify the /P flag and if *imageMatrix* consists of three layers (RGB), then all three layers are rotated. Otherwise the operation rotates only the first layer of the wave.

rotateRows

Rotates columns in place. This operation is analogous to the **Rotate** operation except that it works on images and rotates an integer number of columns. The number of columns is specified by the /G flag.

When *imageMatrix* contains multiple layers you can use the /P flag to specify the layer of the wave that will undergo rotation. By default, if you do not specify the /P flag and if *imageMatrix* consists of three layers (RGB), then all three layers are rotated. Otherwise the operation rotates only the first layer of the wave.

scalePlanes

Scales each plane of the 3D wave, *imageMatrix*, by a constant taken from the corresponding entry in the 1D wave specified by the /D flag. The result is stored in the wave M_ScaledPlanes unless the /O flag is specified, in which case scaling is done in place.

When using /O, first redimension the wave to a different data type to make sure there are no artifacts due to type clipping.

If *imageMatrix* is double precision, M_ScaledPlanes is double precision. Otherwise M_ScaledPlanes is single precision.

This operation also supports the optional flag.

Note that when you display M_ScaledPlanes, which has three planes that originated from scaling byte data, you will have to multiply the wave by 255 to see the image because the RGB format for single and double precision data requires values in the range 0 to 65535.

selectColor

Creates a mask for the image in which pixel values depend on the proximity of the color of the image to a given central color. The central color, the tolerance and a grayscale indicator must be specified using the /E flag.

For example, `/E={174,187,75,10,1}` specifies an RGB of (174,187,75), a tolerance level of 10 and a requested grayscale output.

RGB values must be provided in a range appropriate for the source image. If the source wave type is not byte or unsigned byte, then the range of the RGB components should be 0 to 65535.

The color proximity is calculated in the nonuniform RGB space and the tolerance applies to the maximum component difference from the corresponding component of the central color.

The tolerance, just like the central color, should be appropriate to the type of the source wave.

The generated mask is stored in the wave M_SelectColor in the current data folder. If a wave by that name exists prior to the execution of this operation, it is overwritten.

	You can also use the /R flag with this operation to limit the color selection to pixels in the ROI wave whose value is zero.
setBeam	Sets the data of a particular beam in *imageMatrix*. A "beam" is a 1D array in the Z-direction. If a row is a 1D array in the first dimension and a column is a 1D array in the second dimension then a beam is a 1D array in the third dimension. Specify the beam with the /Beam={*row*,*column*} flag and the 1D beam data wave with the /D flag. The beam data wave must have the same number of elements as the number of layers and same numeric type as *imageMatrix*. Use getBeam to extract the beam.
setChunk	Overwrites the data in the wave *imageMatrix* at chunk index specified by /CHIX with the data contained in a 3D wave specified by the /D flag. The assigned data must be contained in a wave that matches the first three dimensions of *imageMatrix* and must have the same number type. See also getChunk and insertChunk.
setPlane	Sets a plane (given by the /P flag) in the designated image with the data in a wave specified by the /D flag. It is designed as a complement of the getPlane keyword to provide an easier (faster) way to create multiplane images. Note that the operation supports setting a plane when the source data is smaller than the destination plane in which case the source data is placed in memory contiguously starting from the corner pixel of the destination plane. If the source data is larger than the destination plane it is clipped to the appropriate rows and columns. If you are setting all planes in the destination wave using algorithmically named source waves you could use the stackImages keyword instead. See also getPlane keyword.
shading	Calculates relative reflectance of a surface for a light source position defined by the /A flag. The operation estimates the slope of the surface and then computes a relative reflectance defined as the dot product of the direction of the light and the normal to the surface at the point of interest. Reflectivity is scaled using the expression: *outPixel* = *shadingA* * (*sunDirection* · *surfaceNormal*) + *shadingB* By default *shadingA*=1, *shadingB*=0. The result is stored in the wave M_ShadedImage, which has the same data type as the source wave. If the source wave is any integer type, and the value of *shadingA*=1 the operation sets that value to 255. The smallest supported wave size is 4x4 elements. Values along the boundary (1 pixel wide) are arbitrary because there are no derivatives calculable for those pixels, so these pixels are filled with duplicates of the inner rows and columns.
shrinkRect	Shrinks *imageMatrix* to include only the minimum rectangle that contains all the pixels whose value is different from an outer value. The outer value is specified with the /F flag. This is useful in situations where **ImageSeedFill** has set the pixels around the object of interest to some outer value and it is desired to extract the smallest rectangle that contains interesting data. The output is stored in the wave M_Shrunk.
stackImages	Creates a 3D or 4D stack from individual image waves in the current data folder. The waves should be of the form *baseNameN*, where *N* is a numeric suffix specifying the sequence order. *imageMatrix* should be the name of the first wave that you want to add to the stack. You can use the /NP flag to specify the number of waves that you want to add to the stack The result is a 3D or 4D wave named M_Stack, which overwrites any existing wave of that name in the current data folder. With /K, it kills all waves copied into the stack.
sumAllCols	Creates a wave W_sumCols in which every entry is the sum of the pixels on the corresponding image column. For a 3D wave, unless you specify a plane using the /P flag it will use the first plane by default.
sumAllRows	Creates a wave W_sumRows in which every entry is the sum of the pixels on the corresponding image row. For a 3D wave, unless you specify a plane using the /P flag it will use the first plane by default.

sumCol — Stores in the variable V_value the sum of the elements in the column specified by /G flag and optionally the /P flag.

sumPlane — Stores in the variable V_value the sum of the elements in the plane specified by the /P flag.

sumPlanes — Creates a 2D wave M_SumPlanes which contains the same number of rows and columns as the 3D source wave. Each entry in M_SumPlanes is the sum of the corresponding pixels in all the planes of the source wave. M_SumPlanes is a double precision wave if the source wave is double precision. Otherwise it is a single precision wave.

sumRow — Stores in the variable V_value the sum of the elements of a row specified by /G flag and optionally the /P flag.

swap — Swaps image data following a 2D FFT. The transform swaps diagonal quadrants of the image in one or more planes. This keyword does not support any flags. The swapping is done in place and it overwrites the source wave.

swap3D — Swaps data following a 3D FFT. The transform swaps diagonal quadrants of the data. This keyword does not support any flags. The swapping is done in place and the source wave is overwritten.

transposeVol — Transposes a 3D wave. The transposed wave is stored in M_VolumeTranspose. The /O flag does not apply. The operation supports the following 5 transpose modes which are specified using the /G flag:

mode	**Equivalent Command**
1	`M_VolumeTranspose=`*`imageMatrix`*`[p][r][q]`
2	`M_VolumeTranspose=`*`imageMatrix`*`[r][p][q]`
3	`M_VolumeTranspose=`*`imageMatrix`*`[r][q][p]`
4	`M_VolumeTranspose=`*`imageMatrix`*`[q][r][p]`
5	`M_VolumeTranspose=`*`imageMatrix`*`[q][p][r]`

vol2surf — Creates a quad-wave output (appropriate for display in Gizmo) that wraps around 3D "particles". A particle is defined as a region of nonzero value voxels in a 3D wave. The algorithm effectively computes a box at the resolution of the input wave which completely encloses the data. The output wave M_Boxy is a 2D single precision wave of 12 columns where each row corresponds to one disjoint quad and the columns provide the sequential X, Y, and Z coordinates of the quad vertices.

voronoi — Computes the voronoi tesselation of a convex domain defined by the X, Y positions of the input wave. *imageMatrix* must be a triplet wave where the first column contains the X-values, the second column contains the Y-values and the third column is an arbitrary (zero is recommended) constant. The result of the operation is stored in the two column wave M_VoronoiEdges which contains sequential edges of the Voronoi polygons. Edges are separated from each other by a row of NaNs. The outer most polygons share one or more edges with a large triangle which contains the convex domain.

xProjection — Computes the projection in the X-direction and stores the result in the wave M_xProjection. See zProjection for more information.

xyz2rgb — Converts a 3D single precision XYZ color-space data into RGB based on the D65 white point. The transformation used is:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}.$$

If you do not specify the /O flag, the results are saved in a single precision 3D wave (NT_FP32) "M_XYZ2RGB".

Note that not all XYZ values map into positive RGB triplets (consider colors that reside outside the RGB triangle in the XYZ diagram). This operation gives you the following choices: by default, the output wave is a single precision wave that will include

possible negative RGB values. If you specify the /U flag, for unsigned short output wave, the operation will set to zero all negative components and scale the remaining ones in the range 0 to 65535.

yProjection — Computes the projection in the Y-direction and stores the result in the wave M_yProjection. See zProjection for more information.

zDot — Computes the dot product of a beam in *srcWave* with a 1D zVector wave (specified with the /D flag). This will convert stacked images of spectral scans into RGB or XYZ depending on the scaling in zVector. The *srcWave* and zVector must be the same data type (float or double). The wave M_StackDot contains the result in the current data folder.

zProjection — Computes the projection in the Z-direction and stores the result in the wave M_zProjection. The source wave must be a 3D wave of arbitrary data type. The value of the projection depends on the method specified via the /METH flag.

Flags

/A={*azimuth*, *elevation* [, *shadingA*, *shadingB*]}

Specifies parameters for shade. Position of the light source is given by *azimuth* (measured in degrees counter-clockwise) and *elevation* (measured in degrees above the horizon).

The parameters *shadingA* and *shadingB* are optional. By default their values are 1 and 0, respectively.

/Beam={*row*,*column*}

Designates a beam in a 3D wave; both *row* and *column* are zero based.

/BPJ={*width*,*height*} Specifies the backProjection parameters: *width* and *height* are the width and height of the reconstructed image and should be equal to the size of the original wave.

/C=*CMapWave* Specifies the colormap wave for cmap2rgb keyword. The *CMapWave* is expected to be a 2D wave consisting of three columns corresponding to the RGB entries.

/CHIX=*chunkIndex* Identifies the chunk index for getting, inserting or setting a chunk of data in a 4D wave. *chunkIndex* ranges from 0 to the number of chunks in *imageMatrix*.

/CLAM=*fuzzy* Sets the value used to compute the fuzzy probability in fuzzyClassify. It must satisfy *fuzzy* > 1 (default is 2).

/CLAS=*num* Sets the number of requested classes in fuzzyClassify. If you don't know the number of expected classes and *num* is too high, fuzzyClassify will likely produce some degenerate classes.

/D=*waveName* Specifies a data wave. Check the appropriate keyword documentation for more information about this wave.

/F=*value* Increases the sampling in the angle domain when used with the Hough keyword. By default *value*=1 and the operation results in single degree increments in the interval 0 to 180, and if *value*=1.5 there will be 180*1.5 rows in the transform.

Specifies the outer pixel value surrounding the region of interest when used with shrinkRect keyword.

/G=*colNumber* Specifies either the row or column number used in connection with getRow or getCol keywords. This flag also specifies the transpose *mode* with the transposeVol keyword.

/H={*minHue*, *maxHue*}

/H=*hueWave* Specifies the range of hue values for selecting pixels. The hue values are specified in degrees in the range 0 to 360. Hue intervals that contain the zero point should be specified with the higher value first, e.g., `/H={330,10}`.

Use *hueWave* when you have more than one pair of hue values that bracket the pixels that you want to select.

See **HSL Segmentation** on page III-318 for an example.

/I=*iterations* Sets the number of iterations in ccsubdivision and in fuzzyClassify.

/INSI=*imageWave* Specifies the wave, *imageWave*, to be used with the insertImage keyword. *imageWave* is a 2D wave of the same numeric data type as *imageMatrix*.

/INSW=*wave* Specifies the 2D wave to be inserted into a 3D wave using the keywords: insertXplane, insertYplane, or insertZplane.

/INSX=*xPos* Specifies the pixel position at which the first row is inserted. Ignores wave scaling.

/INSY=*yPos* Specifies the pixel position at which the first column is inserted. Ignores wave scaling.

/IOFF={*dx*,*dy*,*bgValue*}
Specifies the amount of positive or negative integer offset with *dx* and *dy* and the new background value, *bgValue*, with the offsetImage keyword.

/IWAV=*wave* Specifies the wave which provides the indices when used with the keyword indexWave. The wave should have as many columns as the dimensions of *imageMatrix* (2, 3, or 4). For example, to specify indices for pixels in an image, the wave should have two columns. The first column corresponds to the row designation and the second to the column designation of the pixel. The wave can be of any number type (other than complex) and entries are assumed to be integer indices; there is no support for interpolation or for wave scaling.

/J=*quality* Specifies the quality of JPEG image compression. *quality* can have values ranging from 0 (lowest) to 100 (highest). This flag is used with the JPEGQ keyword.

/L={*minLight, maxLight*}
/L=*lightnessWave* Specifies the range of lightness for selecting pixels. The lightness values are in the range 0-1. If you do not use the /L flag than the default full range is used.

Use *lightnessWave* when you have more than one pair of lightness values corresponding to the pixels that you want to select. For each pair, values should be arranged so that the smaller one is first and the larger is second. There is no restriction on the order of pairs in the wave except that they match the other waves used by the operation.

/LARA=*minPixels*
Specifies the minimum number of pixels required for an area to be masked by the findLakes keyword. If you do not specify this flag, the default value used is 100.

/LCVT use 8-connectivity instead of 4-connectivity.

/LRCT={*minX*,*minY*,*maxX*,*maxY*}
Sets the rectangular region of interest for the findLakes keyword. The operation will not affect the original data outside the specified rectangle. The X and Y values are the scaled values (i.e., using wave scaling).

/LTAR=*target* Set the target value for the masked region in the findLakes keyword.

/LTOL=*tol* Specifies the tolerance for the findLakes keyword. By default the tolerance is zero. The tolerance must be a positive number. The operation uses the tolerance by requiring neighboring pixels to have a value between that of the current pixel V and V+*tol*.

/M=*n* Specifies the method by which a 2D target image is filled with data from a 1D wave using the fillImage keyword.

n =0: Straight column fill, which you can also accomplish by a redimension operation.

n =1: Straight row fill.

n =2: Serpentine column fill. The points from the data wave are sequentially loaded onto the first column and continue from the last to the first point of the second column, and then sequentially through the third column, etc.

n =3: Serpentine row fill.

/METH=*method* Determines the values of the projected pixels for xProjection, yProjection, and zProjection keywords.

method =1: Pixel (i,j) in M_zProjection is assigned the maximum value that (i,j,*) takes among all layers of *imageMatrix* (default).

method =2: Pixel (i,j) in M_zProjection is assigned the average value that (i,j,*) takes among all layers of *imageMatrix*.

method =3: Pixel (i,j) in M_zProjection is assigned the minimum value that (i,j,*) takes among all layers of *imageMatrix*.

/N={*rowsToAdd, colsToAdd*}
Creates an image that is larger or smaller by *rowsToAdd*, *colsToAdd*. The additional pixels are set by duplicating the values in the last row and the last column of the source image.

/NCLR=*M* Specifies the maximum number of colors to find with the rgb2cmap keyword. *M* must be a positive number; the default value is 256 colors.

The result of the operation is saved in the wave M_paddedImage.

/NP=*numPlanes* Specifies the number of planes to remove from a 3D wave when using the removeXplane, removeYplane, or removeZplane keywords. Specifies the number of waves to be added to the stack with the stackImages keyword.

/O Overwrites the input wave with the result except in the cases of Hough transform and cmap2rgb. Does not apply to the transposeVol parameter.

/P=*planeNum* Specifies the plane on which you want to operate with the rgb2gray or getPlane keywords. Also used for getRow or getCol if the source wave is 3D.

/PSL={*xStart*,*dx*,*Nx*,*aStart*,*da*,*Na*}
Specifies projection slice parameters. *xStart* is the first offset of the parallel rays measured from the center of the image. *dx* is the directed offset to the next ray in the fan and *Nx* is the number of rays in the fan. *aStart* is the first angle for which the projection is calculated. The angle is measured between the positive X-direction and the direction of the ray. *da* is the offset to the next angle at which the fan of rays is rotated and *Na* is the total number of angles for which the projection is computed.

/PTYP=*num* Specifies the plane to use with the getPlane keyword.

num=0: XY plane.

num=1: XZ plane.

num=2: YZ plane.

/Q Quiet flag. When used with the Hough transform, it suppresses report to the history of the angle corresponding to the maximum.

/R=*roiWave* Specifies a region of interest (ROI) defined by *roiWave*. For use with the keywords: averageImage, scalePlanes and selectColor.

/S={*minSat, maxSat*}
/S=*saturationWave* Specifies the range of saturation values for selecting pixels. The saturation values are in the range 0 to 1. If you do not use the /S flag, the default value is the full saturation range.

Use *saturationWave* when you have more than one pair of saturation values. If you use a saturation wave you must also use a lightness wave (see /L). *saturationWave* should consist of pairs of values where the first point is the lower saturation value and the second point is the higher saturation value. There is no restriction on the order of pairs within the wave.

/SEG Computes the segmentation image for fuzzyClassify. The image is stored in the 2D wave M_FuzzySegments. The value of each pixel is 255**classIndex*/*number of classes*. Here *classIndex* is the index of the class to which the pixel belongs with the highest probability.

/T=*flag* Use one or more of the following flags.

1: Swaps the data so that the DC is at the center of the image.

2: Calculates the power defined as: $P(f) = 0.5 \cdot (H(f)^2 + H(-f)^2)$.

/TEXT=*val* Specifies the type of texture to create with the imageToTexture keyword. *val* is a binary flag that can be a combination of the following values.

val	Texture
1	Truncates each dimension to the nearest power of 2, which is required for OpenGL textures.
2	Creates a 1D texture (all other textures are for 2D applications).
4	Creates a single channel texture suitable for alpha or luminance channels.
8	Creates a RGB texture from a 3 (or more) layer data.
16	Creates a RGBA texture. If *imageMatrix* does not have a 4th layer, alpha is set to 255.

/TOL=*tolerance* Sets the tolerance for iteration convergence with fuzzyClassify. Convergence is satisfied when the sum of the squared differences of all classes drops below *tolerance*, which must not be negative.

/U Creates an HSL wave of type unsigned short that contains values between 0 and 65535 when used with rgb2hsl.

/W Pads the image by wrapping the data. If you are adding more rows or more columns than are available in the source wave, the operation cycles through the source data as many times as necessary.

/X={*Nx,Ny,x1,y1,z1,x2,y2,z2,x3,y3,z3*}
Nx and *Ny* are the rows and columns of the wave M_ExtractedSurface. The remaining parameters specify three 3D points on the extracted plane. The three points must be chosen at the vertices of the plane and entered in clock-wise order without skipping a vertex.

/Z Ignores errors.

Examples

If you want to insert a 2D (M x N) wave, plane0, into plane number 0 of an (M x N x 3) wave, rgbWave:

```
ImageTransform /P=0/D=plane0 setPlane rgbWave
```

If your source wave is 100 rows by 100 columns and you want to create a montage of this image use:

```
ImageTransform /W/N={200,200} padImage srcWaveName
```

An example of hue and saturation segmentation on an HSL wave.

```
Function hueSatSegment(hslW,lowH,highH,lowS,highS)
    Wave hslW
    Variable lowH,highH,lowS,highS

    Make/D/O/N=(2,3) conditionW
    conditionW={{lowH,highH},{lowS,highS},{NaN,NaN}}
    ImageTransform/D=conditionW matchPlanes hslW
    KillWaves/Z conditionW
End
```

An example of voronoi tesselation.

```
Make/O/N=(33,3) ddd=gnoise(4)
ImageTransform voronoi ddd
Display ddd[][1] vs ddd[][0]
ModifyGraph mode=3,marker=19,msize=1,rgb=(0,0,65535)
Appendtograph M_VoronoiEdges[][1] vs M_VoronoiEdges[][0]
SetAxis left -15,15
SetAxis bottom -5,10
```

See Also

Chapter III-11, **Image Processing**, for many examples. In particular see: **Color Transforms** on page III-297, **Handling Color** on page III-323, and **General Utilities: ImageTransform Operation** on page III-325. The **MatrixOp** operation.

References

Born, Max, and Emil Wolf, *Principles of Optics*, 7th ed., Cambridge University Press, 1999.

Details about the rgb2i123 transform:

Gevers, T., and A.W.M. Smeulders, Color Based Object Recognition, *Pattern Recognition*, *32*, 453-464, 1999.

ImageUnwrapPhase

ImageUnwrapPhase [***flags***][**qualityWave=*qWave*,**] **srcwave=*waveName***

The ImageUnwrapPhase operation unwraps the 2D phase in srcWave and stores the result in the wave M_UnwrappedPhase in the current data folder. srcWave must be a real valued wave of single or double precision. Phase is measured in cycles (units of 2π).

Parameters

qualityWave=*qWave* Specifies a wave, *qWave,* containing numbers that rate the quality of the phase stored in the pixels. *qWave* is 2D wave of the same dimensions as srcWave that can be any real data type and values can have an arbitrary scale. If used with /M=1 the quality values determine the order of phase unwrapping subject to branch cuts, with higher quality unwrapped first. If used with /M=2 the unwrapping is guided by the quality values only. This wave must not contain any NaNs or INFs.

srcwave=*waveName* Specifies a real-valued SP or DP wave that may contain NaNs or INFs but is otherwise assumed to contain the phase modulo 1.

Flags

/E — Eliminate dipoles. Only applies to Goldstein's method (/M=1). Dipoles are a pair of a positive and negative residues that are side by side. They are eliminated from the unwrapping process by replacing them with a branch cut. The variable V_numResidues contains the number of residues remaining after removal of the dipoles.

/L — Saves the lookup table(LUT) used in the analysis with /M=1. This information may be useful in analyzing your results. The LUT is saved as a 2D unsigned byte wave M_PhaseLUT in the current data folder. Each entry consists of 8-bit fields:

bit 0: Positive residue.

bit 1: Negative residue.

bit 2: Branch cut.

bit 3: Image boundary exclusion.

Other bits are reserved and subject to change. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/M=*method* — Determines the method for computing the unwrapped phase:

method=0: Modified Itoh's algorithm, which assumes that there are no residues in the phase. The phase is unwrapped in a contiguous way subject only to the ROI or singularities in the data (e.g., NaNs or INFs). You will get wrong results for the unwrapped phase if you use this method and your data contains residues.

method=1: Modified Goldstein's algorithm. Creates the variables V_numResidues and V_numRegions. Optional *qWave* can determine order of unwrapping around the branch cuts.

method=2: Uses a quality map to decide the unwrapping path priority. The quality map is a 2D wave that has the same dimensions as the source wave but could have an arbitrary data type. The phase is unwrapped starting from the largest value in the quality map.

/MAX=*len* — Specifies the maximum length of a branch cut. Only applicable to Goldstein's method (/M=1). By default this is set to the largest of rows or columns.

/Q — Suppresses messages to the history.

/R=*roiWave* — Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u) that has the same number of rows and columns as *waveName*. The ROI itself is defined by entries or pixels in the *roiWave* with value of 1. Pixels outside the ROI should be set to zero. The ROI does not have to be contiguous but it is best if you choose a convex ROI in order to make sure that any branch cuts computed by the algorithm lie completely inside the ROI domain.

/REST = *threshold* — Sets the threshold value for evaluating a residue. The residue is evaluated by the equivalent of a closed path integral. If the path integral value exceeds the threshold value, the top-left corner of the quad is taken to be a positive residue. If the path integral is less than -*threshold*, it is a negative residue.

Details

Phase unwrapping in two dimensions is difficult because the result of the operation needs to be such that any path integral over a closed contour will vanish. In many practical situations, certain points in the plane have the property that a path integral around them is not zero. These nonzero points are residues. We use the definition that when a counterclockwise path integral leads to a positive value the residue is called a positive residue.

ImageUnwrapPhase uses the modified Itoh's method by default. Phase is unwrapped with an offset equal to the first element that is allowed by the ROI starting at (0,0) and scanning by rows. If there are no residues or if you unwrap the phase using Itoh's algorithm, then the phase is unwrapped only subject to the optional ROI using a seed-fill type algorithm that unwraps by growing a region outward from the seed pixel. Each time that the region growing is terminated by boundaries (external or due to the ROI), the algorithm returns to the row scanning to find a new starting point.

If there are residues and you choose Goldstein's method, the residues are first mapped into a lookup table (LUT) and branch-cuts are determined between residues and boundaries. It is also possible to remove some residues (dipoles) using the /E flag. Phase is then unwrapped in regions bounded by branch cuts using a seed-fill type algorithm that does not cross branch cuts. With a quality wave, the algorithm follows the same

seed-fill approach except that it gives priority to pixels with high quality level. The phase on the branch cuts themselves is subsequently calculated.

The output wave M_UnwrappedPhase has the same wave scaling and dimension units as srcWave. The unwrapped phase is units of cycles; you will have to multiply it by 2π if you need the results in radians.

The operation creates two variables:

V_numResidues	Number of residues encountered(if using /M=1).
V_numRegions	Number of independent phase regions. In Goldstein's method the regions are bounded by branch cuts, but in Itoh's method they depend on the content of the ROI wave.

Examples

To unwrap a complex wave wCmplx:

```
Make/O/N=(DimSize(wCmplx,0),DimSize(wCmplx,1) phaseWave
phaseWave=atan2(imag(wCmplx),real(wCmplx))/2*pi
ImageUnwrapPhase/N=1 srcWave=phaseWave
```

To find the locations of positive residues in the phase:

```
ImageUnwrapPhase/N=1/L srcWave=phaseWave
Duplicate/O M_PhaseLUT ee
ee=M_PhaseLUT&3 ? 1:0
```

To find the branch cuts:

```
Duplicate/O M_PhaseLUT bc
bc=M_PhaseLUT&4 ? 1:0
```

See Also

The **Unwrap** operation and the **mod** function.

References

The following reference is an excellent text containing in-depth theory and detailed explanation of many two-dimensional phase unwrapping algorithms:

Ghiglia, Dennis C., and Mark D. Pritt, *Two Dimensional Phase Unwrapping — Theory, Algorithms and Software*, Wiley, 1998.

ImageWindow

ImageWindow [**/I/O/P=*param***] ***method srcWave***

The ImageWindow operation multiplies the named waves by the specified windowing method.

ImageWindow is useful in preparing an image for FFT analysis by reducing FFT artifacts produced at the image boundaries.

Parameters

srcWave Two-dimensional wave of any numerical type. See **WindowFunction** for windowing one-dimensional data.

method Selects the type of windowing filter, which is one of the following:

Hanning: $w(n) = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi n}{L-1}\right)\right] \qquad 0 \le n \le L-1$.

Hamming: $w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{L-1}\right) \qquad 0 \le n \le L-1$.

Bartlet: Synonym for Bartlett.

Bartlett: $w(n) = \begin{cases} \frac{2n}{L-1} & 0 \le n \le \frac{L-1}{2} \\ 2 - \frac{2n}{L-1} & \frac{L-1}{2} \le n \le L-1 \end{cases}$.

Blackman: $w(n) = 0.42 - 0.5\cos\left(\frac{2\pi n}{L-1}\right) + 0.08\cos\left(\frac{4\pi n}{L-1}\right) \qquad 0 \le n \le L-1$.

Kaiser: $$\frac{I_0\left\{\omega_a\left[((L-1)/2)^2-[n-((L-1)/2)]^2\right]^{1/2}\right\}}{I_0\{\omega_a((L-1)/2)\}} \qquad 0 \le n \le L-1 .$$

where $I_0\{\ldots\}$ is the zeroth-order Bessel function of the first kind and ω_a is the design parameter specified by /P=*param*.

KaiserBessel20: $\alpha = 2.$

KaiserBessel25: $\alpha = 2.5.$

KaiserBessel30: $\alpha = 3.$

$$w(n) = \frac{I_0\left(\pi\alpha\sqrt{1-\left(\frac{n}{(L)/2}\right)^2}\right)}{I_0(\pi\alpha)} \qquad 0 \le |n| \le \frac{L}{2} .$$

$$I_0(X) = \sum_{k=0}^{\infty}\left(\frac{(x/2)^2}{k!}\right)^2$$

In all equations, L is the array width and n is the pixel number.

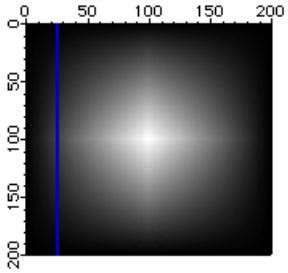
Flags

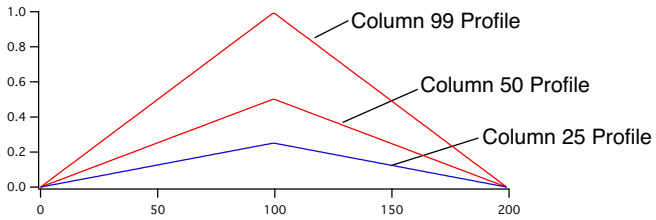
/I	Creates only the output wave containing the windowing filter values that are used to multiply each pixel in *srcWave*. It does not filter the source image.
/O	Overwrites the source image with the output image. If /O is not used then the operation creates the M_WindowedImage wave containing the filtered source image.
/P=*param*	Specifies the design parameter for the Kaiser window.

Details

The 1-dimensional window for each column is multiplied by the value of the corresponding row's window value. In other words, each point is multiplied by the both the row-oriented and column-oriented window value.

This means that all four edges of the image are decreased while the center remains at or near its original value. For example, applying the Bartlett window to an image whose values are all equal results in a trapezoidal pyramid of values:





The default output wave is created with the same data type as the source image. Therefore, if the source image is of type unsigned byte (/b/u) the result of using /I will be identically zero (except possibly for the middle-most pixel). If you keep in mind that you need to convert the source image to a wave type of single or double precision in order to perform the FFT, it is best if you convert your source image (e.g., `Redimension/S srcImage`) before using the ImageWindow operation.

The windowed output is in the M_WindowedImage wave unless the source is overwritten using the /O flag.

The necessary normalization value (equals to the average squared window factor) is stored in V_value.

Examples

To see what one of the windowing filters looks like:

```
Make/N=(80,80) wShape                  // Make a matrix
ImageWindow/I/O Blackman wShape        // Replace with windowing filter
Display;AppendImage wShape                  // Display windowing filter
Make/N=2 xTrace={0,79},yTrace={39,39}       // Prepare for 1D section
AppendToGraph yTrace vs xTrace
ImageLineProfile srcWave=wShape, xWave=xTrace, yWave=yTrace
Display W_ImageLineProfile             // Display 1D section of filter
```

See Also

The **WindowFunction** operation for information about 1D applications.

Spectral Windowing on page III-240. Chapter III-11, **Image Processing** contains links to and descriptions of other image operations.

See **FFT** operation for other 1D windowing functions for use with FFTs; **DSPPeriodogram** uses the same window functions. See **Correlations** on page III-306.

References

For further windowing information, see page 243 of:

Pratt, William K., *Digital Image Processing*, John Wiley, New York, 1991.

IndependentModule

```
#pragma IndependentModule = imName
```

The IndependentModule pragma designates groups of one or more procedure files that are compiled and linked separately. Once compiled and linked, the code remains in place and is usable even though other procedures may fail to compile. This allows functioning control panels and menus to continue to work regardless of user programming errors.

See Also

Independent Modules on page IV-214, **The IndependentModule Pragma** on page IV-43 and **#pragma**.

IndependentModuleList

```
IndependentModuleList(listSepStr)
```

The IndependentModuleList function returns a string containing a list of independent module names separated by listSepStr.

Use **StringFromList** to access individual names.

Parameters

listSepStr contains the character, usually ";", to be used to to separate the names in the returned list.

Details

Only the first character of *listSepStr* is used.

ProcGlobal is not in the returned list, and the order of returned names is not defined.

See Also

Independent Modules on page IV-214.

GetIndependentModuleName, **StringFromList**, **FunctionList**.

IndexedDir

```
IndexedDir(pathName, index, flags)
```

The IndexedDir function returns a string containing the name of or the full path to the *index*th folder in the folder referenced by *pathName*.

Parameters

pathName is the name of an Igor symbolic path pointing to the parent directory.

index is the index number of the directory (within the parent directory) of interest starting from 0. If *index* is -1, IndexedDir will return the name of *all* of the folders in the parent, separated by semicolons.

flags is a bitwise parameter:

Bit 0:	Set if you want a full path. Cleared if you want just the directory name.

All other bits are reserved and should be cleared.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Details

You create the symbolic path identifying the parent directory using the **NewPath** operation or the New Path dialog (Misc menu).

Prior to Igor Pro 3.1, IndexedDir was an external function and took a string as the first parameter rather than a name. The *pathName* parameter can now be either a name or a string containing a name. Any of the following will work:

```
String str = "IGOR"
Print IndexedDir(IGOR, 0, 0)         // First parameter is a name.
Print IndexedDir($str, 0, 0)         // First parameter is a name.
Print IndexedDir("IGOR", 0, 0)       // First parameter is a string.
Print IndexedDir(str, 0, 0)          // First parameter is a string.
```

The acceptance of a string is for backward compatibility only. New code should be written using a name.

The returned path uses the native conventions of the OS under which Igor is running.

Examples

Here is an example for heavy-duty Igor Pro programmers. It is an Igor Pro user-defined function that prints the paths of all of the files and folders in a given folder with or without recursion. You can rework this to do something with each file instead of just printing its path.

To try the function, copy and paste it into the Procedure window. Then execute the example shown in the comments.

```
// PrintFoldersAndFiles(pathName, extension, recurse, level)
// Shows how to recursively find all files in a folder and subfolders.
// pathName is the name of an Igor symbolic path that you created
// using NewPath or the Misc->New Path menu item.
// extension is a file name extension like ".txt" or "????" for all files.
// recurse is 1 to recurse or 0 to list just the top-level folder.
// level is the recursion level - pass 0 when calling PrintFoldersAndFiles.
// Example: PrintFoldersAndFiles("Igor", ".ihf", 1, 0)
Function PrintFoldersAndFiles(pathName, extension, recurse, level)
   String pathName     // Name of symbolic path in which to look for folders and files.
   String extension    // File name extension (e.g., ".txt") or "????" for all files.
   Variable recurse    // True to recurse (do it for subfolders too).
   Variable level      // Recursion level. Pass 0 for the top level.

   Variable folderIndex, fileIndex
   String prefix

   // Build a prefix (a number of tabs to indicate the folder level by indentation)
   prefix = ""
   folderIndex = 0
   do
      if (folderIndex >= level)
         break
      endif
      prefix += "\t"      // Indent one more tab
      folderIndex += 1
   while(1)

   // Print folder
   String path
   PathInfo $pathName     // Sets S_path
   path = S_path
   Printf "%s%s\r", prefix, path

   // Print files
   fileIndex = 0
   do
      String fileName
      fileName = IndexedFile($pathName, fileIndex, extension)
      if (strlen(fileName) == 0)
         break
      endif
      Printf "%s\t%s%s\r", prefix, path, fileName
      fileIndex += 1
   while(1)

   if (recurse)           // Do we want to go into subfolder?
      folderIndex = 0
      do
         path = IndexedDir($pathName, folderIndex, 1)
         if (strlen(path) == 0)
            break        // No more folders
         endif
```

```
            String subFolderPathName = "tempPrintFoldersPath_" + num2istr(level+1)

            // Now we get the path to the new parent folder
            String subFolderPath
            subFolderPath = path

            NewPath/Q/O $subFolderPathName, subFolderPath
            PrintFoldersAndFiles(subFolderPathName, extension, recurse, level+1)
            KillPath/Z $subFolderPathName

            folderIndex += 1
        while(1)
    endif
End
```

IndexedFile

IndexedFile(*pathName*, *index*, *fileTypeOrExtStr* [, *creatorStr*])

The IndexedFile function returns a string containing the name of the *index*th file in the folder specified by *pathName* which matches the file type or extension specified by *fileTypeOrExtStr*.

Details

IndexedFile returns an empty string ("") if there is no such file.

pathName is the *name* of an Igor symbolic path. It is *not* a string.

index normally starts from zero. However, if *index* is -1, IndexedFile returns a string containing a semicolon-separated list of the names of all files in the folder associated with the specified symbolic path which match *fileTypeOrExtStr*.

fileTypeOrExtStr is either:

- A string starting with ".", such as ".txt", ".bwav", or ".c". Only files with a matching file name extension are indexed. Set *fileTypeOrExtStr* to "." to index file names that end with "." such as "myFileNameEndsWithThisDot."
- A string containing exactly four characters, such as "TEXT" or "IGBW". Only files of the specified Macintosh file type are indexed. However, if *fileTypeOrExtStr* is "????", files of any type are indexed.

  On Windows, Igor considers files with ".txt" extensions to be of type TEXT. It does similar mappings for other extensions. See **File Types and Extensions** on page III-404 for details.

creatorStr is an optional string argument containing four characters such as "IGR0". Only files with the specified Macintosh creator code are indexed. Set *creatorStr* to "????" to index all files (or omit the argument altogether). This argument is ignored on Windows systems.

Treatment of Macintosh Dot-Underscore Files

As of Igor Pro 6.10, IndexedFile ignores "dot-underscore" files unless fileTypeOrExtStr is "????".

A dot-underscore file is a file created by Macintosh when it writes to a non-HFS volume, for example, when it writes to a Windows volume via SMB file sharing. The dot-underscore file stores Macintosh HFS-specific data such as the file's type and creator codes, and the file's resource fork, if it has one.

For example, if a file named "wave0.ibw" is copied via SMB to a Windows volume, Mac OS X creates two files on the Windows volume: "wave0.ibw" and "._wave0.ibw". Mac OS X makes these two files appear as one to Macintosh applications. However, Windows does not do this. As a consequence, when a Windows program sees "._wave0.ibw", it expects it to be a valid .ibw file, but it is not. This causes problems.

By ignoring dot-underscore files, IndexedFile prevents this type of problem. However, if fileTypeOrExtStr is "????", IndexedFile will return dot-underscore files on Windows.

Examples

```
NewPath/O myPath "MyDisk:MyFolder:"
Print IndexedFile(myPath,-1,"TEXT")         // all text-type files
Print IndexedFile(myPath,0,"TEXT")          // only the first text file
Print IndexedFile(myPath,-1,".dat")         // *.dat
Print IndexedFile(myPath,-1,"TEXT","IGR0")  // all Igor text files
Print IndexedFile(myPath,-1,"????")         // all files, all creators
```

See **IndexedDir** for another example using IndexedFile.

See Also
The **TextFile** and **IndexedDir** functions.

IndexSort

IndexSort [**/DIML**] ***indexWaveName, sortedWaveName*** [**, *sortedWaveName***]…

The IndexSort operation sorts the values in each *sortedWaveName* wave according to the Y values of *indexWaveName*.

Flags

/DIML Moves the dimension labels with the values (keeps any row dimension label with the row's value).

Details

indexWaveName can not be complex. *indexWaveName* is presumed to have been the destination of a previous **MakeIndex** operation.

This has the effect of putting the *sortedWaveName* waves in the same order as the wave from which the index values in *indexWaveName* was made.

All of the *sortedWaveName* waves must be of equal length.

See Also
MakeIndex and IndexSort Operations on page III-137.

Inf

Inf

The Inf function returns the "infinity" value.

InsertPoints

InsertPoints [**/M=*dim***] ***beforePoint, numPoints, waveName*** [**, *waveName***]…

The InsertPoints operation inserts *numPoints* points in front of point *beforePoint* in each *waveName*. The new points have the value zero.

Flags

/M=*dim* Specifies the dimension into which elements are to be inserted. Values are 0 for rows, 1 for columns, 2 for layers, 3 for chunks. If /M is omitted, InsertPoints inserts in the rows dimension.

Details

Trying to insert points into any but the rows of a zero-point wave results in a zero-point wave. You must first make the number of rows nonzero before anything else has an effect.

See Also
Lists of Values on page II-96.

Integrate

Integrate [***type flags***][***flags***] ***yWaveA*** [**/X = *xWaveA***][**/D = *destWaveA***]
[, ***yWaveB*** [**/X = *xWaveB***][**/D = *destWaveB***][**, …**]]

The Integrate operation calculates the 1D numeric integral of a wave. X values may be supplied by the X-scaling of the source wave or by an optional X wave. Rectangular integration is used by default.

Flags

/DIM= *d* Specifies the wave dimension along which to differentiate when yWave is multi-dimensional.

d=-1: Treats entire wave as 1D (default).

For *d*=0, 1, 2, 3, operates along rows, columns, layers or chunks.

For example, for a 2D wave, /DIM=0 integrates each row and /DIM=1 integrates each column.

/METH=*m* Sets the integration method.

m=0: Rectangular integration (default). Results at a point are stored at the same point (rather than at the next point as for /METH=2). This method keeps the dimension size the same.

m=1: Trapezoidal integration.

m=2: Rectangular integration. Results at a point are stored at the next point (rather than at the same point as for /METH=0). This method increases the dimension size by one to provide a place for the last bin.

/P Forces point scaling.

/T Trapezoidal integration. Same as /METH=1.

Type Flags *(used only in functions)*

Integrate also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when it needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-58 and **WAVE Reference Type Flags** on page IV-58 for a complete list of type flags and further details.

Wave Parameters

Note: ***All*** wave parameters must follow *yWave* in the command. All wave parameter flags and type flags must appear immediately after the operation name (`Integrate`).

/D=*destWave* Specifies the name of the wave to hold the integrated data. It creates *destWave* if it does not already exist or overwrites it if it exists.

/X=*xWave* Specifies the name of corresponding X wave. For rectangular integration, the number of points in the X wave must be one greater than the number of elements in the Y wave dimension being integrated.

Details

The computation equation for rectangular integration using /METH=0 is:

$$waveOut[p] = \sum_{i=0}^{p} waveIn[i] \cdot [x_{i+1} - x_i] .$$

The computation equation for rectangular integration using /METH=2 is:

$$waveOut[0] = 0$$

$$waveOut[p+1] = \sum_{i=0}^{p} waveIn[i] \cdot [x_{i+1} - x_i] .$$

The inverse of this rectangular integration is the backwards difference.

Trapezoidal integration (/METH=1) is a more accurate method of computing the integral than rectangular integration. The computation equation is:

$$waveOut[0] = 0$$

$$waveOut[p] = waveOut[p-1] + \frac{waveIn[p-1] + waveIn[p]}{2} \cdot [x_p - x_{p-1}] .$$

If the optional /D = *destWave* flag is omitted, then the wave is integrated in place overwriting the source wave.

When using an X wave, the X wave must be a 1D wave with data type matching the Y wave (excluding the complex type flag). Rectangular integration (/METH=0 or 2) requires an X wave having one more point than the number of elements in the dimension of the Y wave being integrated. X waves with number points plus one are allowed for rectangular integration with methods needing only the number of points. X waves are not used with integer source waves.

Although it is mathematically suspect, rectangular integration using /METH=0 would be correct if the X scaling of the output wave is offset by ΔX.

`Differentiate/METH=1/EP=1` is the inverse of `Integrate/METH=2`, but `Integrate/METH=2` is the inverse of `Differentiate/METH=1/EP=1` only if the original first data point is added to the output wave.

Integrate applied to an XY pair of waves does not check the ordering of the X values and doesn't care about it. However, it is usually the case that your X values should be monotonic. If your X values are not monotonic, you should be aware that the X values will be taken from your X wave in the order they are found, which will result in random X intervals for the X differences. It is usually best to sort the X and Y waves first (see **Sort**).

See Also

The **Differentiate** operation. The **Integrate1D**, **area** and **areaXY** functions.

Integrate1D

Integrate1D(*UserFunctionName*, *min_x*, *max_x* [, *options* [, *count*]])

The Integrate1D function performs numerical integration of a user function between the specified limits (*min_x* and *max_x*).

Parameters

UserFunctionName must have the following format:

```
Function UserFunctionName(inX)
    Variable inX
    ... do something
    return result
End
```

options is one of the following:

0: Trapezoidal integration (default).

1: Romberg integration.

2: Gaussian Quadrature integration.

By default, *options* is 0 and the function performs trapezoidal integration. In this case Igor evaluates the integral iteratively. In each iteration the number of points where Igor evaluates the function increases by a factor of 2. The iterations terminate at convergence to tolerance or when the number of evaluations is 2^{23}.

The *count* parameter specifies the number of subintervals in which the integral is evaluated. If you specify 0 or a negative number for count, the function performs an adaptive Gaussian Quadrature integration in which Igor bisects the interval and performs a recursive refining of the integration only in parts of the interval where the integral does not converge to tolerance.

Details

You can integrate complex-valued functions using a function with the format:

```
Function/C complexUserFunction(inX)
    Variable inX
    Variable/C result
    //… do something
    return result
End
```

The syntax used to invoke the function is:

```
Variable/C cIntegralResult=Integrate1D(complexUserFunction,min_x,max_x…)
```

You can also use Integrate1D to perform a higher dimensional integrals. For example, consider the function: $F(x,y) = 2x + 3y + xy$.

In this case, the integral $h = \int dy \int f(x,y)dx$ can be performed by establishing two user functions:

```
Function Do2dIntegration(xmin,xmax,ymin,ymax)
    Variable xmin,xmax,ymin,ymax

    Variable/G globalXmin=xmin
    Variable/G globalXmax=xmax
    Variable/G globalY

    return Integrate1d(userFunction2,ymin,ymax,1)  // Romberg integration
End

Function UserFunction1(inX)
    Variable inX

    NVAR globalY=globalY

    return (3*inX+2*globalY+inX*globalY)
End
```

```
Function UserFunction2(inY)
   Variable inY

   NVAR globalY=globalY
   globalY=inY
   NVAR globalXmin=globalXmin
   NVAR globalXmax=globalXmax

   // Romberg integration
   return Integrate1D(userFunction1,globalXmin,globalXmax,1)
End
```

This method can be extended to higher dimensions.

If the integration fails to converge or if the integrand diverges, Integrate1D returns NaN. When a function fails to converge it is a good idea to try another integration method or to use a user-defined number of intervals (as specified by the count parameter). Note that the trapezoidal method is prone to convergence problems when the absolute value of the integral is very small.

See Also

The **Integrate** operation.

IntegrateODE

IntegrateODE [***flags***] ***derivFunc, cwaveName, ywaveSpec***

The IntegrateODE operation calculates a numerical solution to a set of coupled ordinary differential equations by integrating derivatives. The derivatives are user-specified via a user-defined function, *derivFunc*. The equations must be a set of first-order equations; a single second-order equation must be recast as two first-order equations, a third-order equation to three first order equations, etc. For more details on how to write the function, see **Solving Differential Equations** on page III-268.

IntegrateODE offers two ways to specify the values of the independent variable (commonly called X or t) at which output Y values are recorded. You can specify the X values or you can request a "free-run" mode.

The algorithms used by IntegrateODE calculate results at intervals that vary according to the characteristics of the ODE system and the required accuracy. You can set specific X values where you need output (see the /X flag below) and arrangements will be made to get values at those specific X values. In between those values, IntegrateODE will calculate whatever spacing is needed, but intermediate values will not be output to you.

If you specify free-run mode, IntegrateODE will simply output all steps taken regardless of the spacing of the X values that results.

Parameters

cwaveName Name of wave containing constant coefficients to be passed to *derivFunc*.

derivFunc Name of user function that calculates derivatives. For details on the form of the function, see **Solving Differential Equations** on page III-268.

ywaveSpec Specifies a wave or waves to receive calculated results. The waves also contain initial conditions. The *ywaveSpec* can have either of two forms:

ywaveName: *ywaveName* is a single, multicolumn wave having one column for each equation in your equation set (if you have just one equation, the wave will be a simple 1D wave).

{*ywave0*, *ywave1*, …}: The *ywaveSpec* is a list of 1D waves, one wave for each equation. The ordering is important — it must correspond to the elements of the y wave and dydx wave passed to *derivFunc*.

Unless you use the /R flag to alter the start point, the solution to the equations is calculated at each point in the waves, starting with row 1. You must store the initial conditions in row 0.

Flags

/CVOP={*solver*, *jacobian*, *extendedErrors* [, *maxStep*]}

Selects options that affect how the Adams-Moulton and BDF integration schemes operate. This flag applies only when using /M = 2 or /M = 3. These methods are based on the CVODE package, hence the flag letters "CV".

The *solver* parameter selects a solver method for each step. The values of *solver* can be:

solver=0: Select the default for the integration method. That is functional for /M=2 or Newton for /M=3.

solver=1: Functional solver.

solver=2: Newton solver.

The *jacobian* parameter selects the method used to approximate the jacobian matrix (matrix of df/dy_i where f is the derivative function).

jacobian=0: Full jacobian matrix.

jacobian=1: Diagonal approximation.

In both cases, the derivatives are approximated by finite differences.

In our experience, *jacobian* = 1 causes the integration to proceed by much smaller steps. It might decrease overall integration time by reducing the computation required to approximate the jacobian matrix.

If the *extendedErrors* parameter is nonzero, extra error information will be printed to the history window in case of an error during integration using /M=2 or /M=3. This extra information is mostly of the sort that will be meaningful only to WaveMetrics software engineers, but may occasionally help you to solve problems. It is printed out as a bug message (`BUG: . . .`) regardless of whether it is our bug or yours.

If *maxStep* is present and greater than zero, this option sets the maximum internal step size that the CVODE package is allowed to take. This is particularly useful with /M=3, as the BDF method is capable of taking extremely large steps if the derivatives don't change much. Use of this option may be necessary to make sure that the CVODE package doesn't step right over a brief excursion in, say, a forcing function. If you have something in your derivative function that may be step-like and brief, set *maxStep* to something smaller than the duration of the excursion.

If you want to set *maxStep* only, set the other three options to zero.

/E=*eps* Adjusts the step size used in the calculations by comparing an estimate of the truncation error against a fraction of a scaled number. The fraction is *eps*. For instance, to achieve error less than one part in a thousand, set *eps* to 0.001. The number itself is set by a combination of the /F flag and possibly the wave specified with the /S flag. See **Solving Differential Equations** on page III-268 for details.

If you do not use the /E flag, *eps* is set to 10^{-6}.

For details, see **Error Monitoring** on page III-278.

/F=*errMethod* Adjusts the step size used in the calculations by comparing an estimate of the truncation error against a scaled number. *errMethod* is a bitwise value that specifies what to include in that number:

bit 0: Add a constant from the error scaling wave set by the /S flag.

bit 1: Add the current value of the results.

bit 2: Add the current value of the derivatives.

bit 3: Multiply by the current step size (/M=0 or /M=1 only).

Each bit that you set of bits 0, 1, or 2 adds a term to the number; setting bit 3 multiplies the sum by the current step size to achieve a global error limit. Note that bit 3 has no effect if you use the Adams or BDF integrators (/M=2 or /M=3). See **Setting Bit Parameters** on page IV-12 for further details about bit settings.

If you don't include the /F flag, a constant is used. Unless you use the /S flag, that constant is set to 1.0.

For details, see **Error Monitoring** on page III-278.

/M=*m* Specifies the method to use in calculating the solution.

m=0: Fifth-order Runge-Kutta-Fehlberg (default).

m=1: Bulirsch-Stoer method using Richardson extrapolation.

m=2: Adams-Moulton method.

m=3: BDF (Backwards Differentiation Formula, or Gear method). This method is the preferred method for stiff problems.

If you don't specify a method, the default is the Runge-Kutta method (*m*=0). Bulirsch-Stoer (*m*=1) should be faster than Runge-Kutta for problems with smooth solutions, but we find that this is often not the case. Simple experiments indicate that Adams-

Moulton (*m*=2 may be fastest for nonstiff problems. BDF (*m*=3) is definitely the preferred one for stiff problems. Runge-Kutta is a robust method that may work on problems that fail with other methods.

/Q [= *quiet*]
quiet = 1 or simply /Q sets quiet mode. In quiet mode, no messages are printed in the history, and errors do not cause macro abort. The variable V_flag returns an error code. See **Details** for the meanings of the V_flag error codes.

/R=(*startX*,*endX*)
Specifies an X range of the waves in *ywaveSpec*.

/R=[*startP*,*endP*]
Specifies a point range in *ywaveSpec*.

If you specify the range as /R=[*startP*] then the end of the range is taken as the end of the wave. If /R is omitted, the entire wave is evaluated. If you specify only the end point (/R = [,*endP*]) the start point is taken as point 0.

You must store initial conditions in *startP*. The first point is *startP*+1.

/S=*errScaleWaveName*
If you set bit 0 of *errMethod* using the /F flag, or if you don't include the /F flag, a constant is required for scaling the estimated error for each differential equation. By default, the constants are simply set to 1.0.

You provide custom values of the constants via the /S flag and a wave. Make a wave having one element for each derivative, set a reasonable scale factor for the corresponding equation, and set *errScaleWaveName* to the name of that wave.

If you don't use the /S flag, the constants are all set to 1.0.

/STOP = {*stopWave*, *mode*}
Requests that IntegrateODE stop when certain conditions are met on either the solution values (Y values) or the derivatives.

stopWave contains information that IntegrateODE uses to determine when to stop.

mode = 0: OR mode. If *stopWave* contains more than one condition, any one condition will stop the integration when it is satisfied.

mode = 1: AND mode. If *stopWave* contains more than one condition, all conditions must be satisfied to cause the integration to stop.

See Details, below, for more information.

/U=*u*
Update the display every *u* points. By default, it will update the display every 10 points. To disable updates, set *u* to a very large number.

/X=*xvaluespec*
Specifies the values of the independent variable (commonly called x or t) at which values are to be calculated (see parameter *ywaveSpec*).

You can provide a wave or *x0* and *deltaX*:

/X = *xWaveName*

Use this form to provide a list of values for the independent variable. They can have arbitrary spacing and may increase or decrease, but should be monotonic.

If you use the /XRUN flag to specify free-run mode, /X = *xWaveName* is required. In this case, the X wave becomes and output wave and any contents are overwritten. See the description of /XRUN for details.

xValues = {*x0, deltaX*}

If you use this form, *x0* is the initial value of the independent variable. This is the value at which the initial conditions apply. It will calculate the first result at *x0*+*deltaX*, and subsequent results with spacing of *deltaX*.

deltaX can be negative.

If you do not use the xValues keyword, it reads independent variable values from the X scaling of the results wave (see *ywaveSpec* parameter).

/XRUN={*dx0, Xmax*}
If *dx0* is nonzero, the output is generated in a free-running mode. That is, the output values are generated at whatever values if the independent variable (*x* or *t*) the integration method requires to achieve the requested accuracy. Thus, you will get solution points at variably-spaced X values.

The parameter *dx0* sets the step size for the first integration step. If this is smaller than necessary, the step size will increase rapidly. If it is too large for the requested accuracy, the integration method will decrease the step size as necessary.

If *dx0* is set to zero, free-run mode is not used; this is the same as if the XRUN flag is not used.

When using free-run mode, you must provide an X wave using /X = *xWaveName*. Set the first value of the wave (this is usually point zero, but may not be if you use the /R flag) to the initial value of X.

As the integration proceeds, the X value reached for each output point is written into the X wave. The integration stops when the latest step taken goes beyond *Xmax* or when the output waves are filled.

Details

The various waves you may use with the IntegrateODE operation must meet certain criteria. The wave to receive the results (*ywaveSpec*), and which contains the initial conditions, must have exactly one column for each equation in your system of equations, or you must supply a list of waves containing one wave for each equation. Because IntegrateODE can't determine how many equations there are from your function, it uses the number of columns or the number of waves in the list to determine the number of equations.

If you supply a list of waves for *ywaveSpec*, all the waves must have the same number of rows. If you supply a wave containing values of the independent variable or to receive X values in free-run mode (using /X=*waveName*) the wave must have the same number of rows as the *ywaveSpec* waves.

The wave you provide for error scaling via the /S flag must have one point for each equation. That is, one point for each *ywaveSpec* wave, or one point for each column of a multicolumn *ywaveSpec*.

By default, the display will update after each tenth point is calculated. If you display one of your *ywaveSpec* waves in a graph, you can watch the progress of the integration.

The display update may slow down the calculation considerably. Use the /U flag to change the interval between updates. To disable the updates entirely, set the update interval to a number larger than the length of the waves in *ywaveSpec*.

In free-run mode, it is impossible to predict how many output values you will get. IntegrateODE will stop when either your waves are filled, or when the X value exceeds *Xmax* set in the /XRUN flag. The best strategy is to make the waves quite large; unused rows in the waves will not be touched. To avoid having "funny" traces on a graph, you can prefill your waves with NaN. Make sure that you don't set the initial condition row and initial X value row to Nan!

Stopping IntegrateODE

In some circumstances it is useful to be able to stop the integration early, before the full number of output values has been computed. You can do this two ways: using the /STOP flag to put conditions on the solution, or by returning 1 from your derivative function.

When using /STOP={*stopWave*, *mode*}, *stopWave* must have one column for each equation in your system or, equivalently, a number of columns equal to the order of your system. Each column represents a condition on either the solution value or the derivatives for a given equation in your system.

Row 0 of *stopWave* contains a flag telling what sort of condition to apply to the solution values. If the flag is zero, that value is ignored. If the flag is 1, the integration is stopped if the solution value exceeds the value you put in row 1. If the flag is -1, integration is stopped when the solution value is less than the value in row 1.

Rows 2 and 3 work just like rows 0 and 1, but the conditions are applied to the derivatives rather than to the solution values.

If *stopWave* has two rows, only the solution values are checked. If *stopWave* has four rows, you can specify conditions on both solution values and derivatives.

You can set more than one flag value non-zero. If you do that, then *mode* determines how the multiple conditions are applied. If *mode* is 0, then any one condition can stop integration when it is satisfied. If *mode* is 1, all conditions with a non-zero flag value must be satisfied at the same time. If row 0 and row 2 have nothing but zeroes, then *stopWave* is ignored.

For further discussion, see **Stopping IntegrateODE on a Condition** on page III-281.

Output Variables

The IntegrateODE operation sets a variety of variables to give you information about the integration. These variables are updated at the same time as the display so you can monitor an integration in progress. They are:

V_ODEStepCompleted	Point number of the last result calculated.
V_ODEStepSize	Size of the last step in the calculation.
V_ODETotalSteps	Total number of steps required to arrive at the current result. In free-run mode, this is the same as V_ODEStepCompleted.
V_ODEMinStep	Minimum step size used during the entire calculation.
V_ODEFunctionCalls	The total number of calls made to your derivative function.

In Quiet mode (see the /Q flag) an V_Flag is set when IntegrateODE stops to indicate why it stopped:

V_flag — Set only in Quiet mode. Value indicates reason that IntegrateODE stopped. The values are:

- 0: Finished normally.
- 1: User aborted the integration.
- 2: Integration stopped because the step size became too small. That is, dX was so small that X + dX = X.
- 3: IntegrateODE ran out of memory.
- 4: In /M=2 or /M=3, the integrator received illegal inputs. Please report this to WaveMetrics (see **Technical Support** on page II-15 for contact details).
- 5: In /M=2 or /M=3, the integrator stopped with a failure in the step solver. The method chosen may not be suitable to the problem.
- 6: Indicates a bug in IntegrateODE. Please report this to WaveMetrics (see **Technical Support** on page II-15 for contact details).
- 7: An error scaling factor was zero (see /S and /F flags).
- 8: IntegrateODE stopped because the conditions specified by the /STOP flag were met.
- 9: IntegrateODE stopped because the derivative function returned a value requesting the stop.

See Also

Solving Differential Equations on page III-268 gives the form of the derivative function, details on the error specifications and what they mean, along with several examples.

References

The Runge-Kutta (/M=0) and Bulirsh-Stoer (/M=1) methods are based on routines in Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992, and are used by permission.

The Adams-Moulton (/M=2) and BDF methods (/M=3) are based on the CVODE package developed at Lawrence Livermore National Laboratory:

Cohen, Scott D., and Alan C. Hindmarsh, *CVODE User Guide*, LLNL Report UCRL-MA-118618, September 1994.

The CVODE package was derived in part from the VODE package. The parts used in Igor are described in this paper:

Brown, P.N., G. D. Byrne, and A. C. Hindmarsh, VODE, a Variable-Coefficient ODE Solver, *SIAM J. Sci. Stat. Comput.*, *10*, 1038-1051, 1989.

interp

interp(*x1, xwaveName, ywaveName*)

The interp function returns a linearly interpolated value at the location x = *x1* of a curve whose X components come from the Y values of *xwaveName* and whose y components come from the Y values of *ywaveName*.

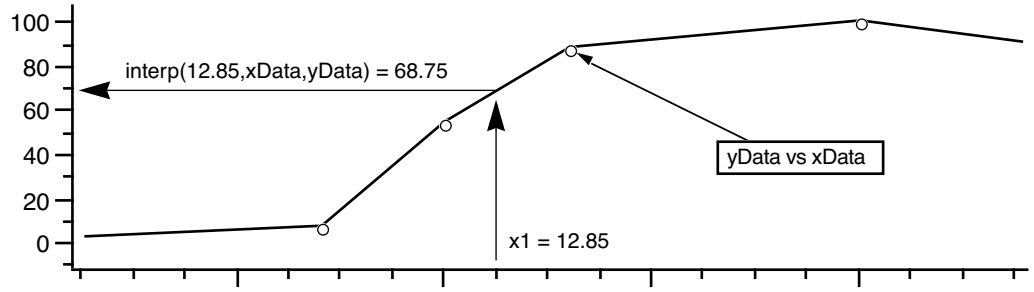
Details

interp returns nonsense if the waves are complex or if *xwaveName* is not monotonic.

The interp function is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

Examples

Examples



See Also

The Interpolate XOP can also do linear as well as spline interpolation.

The **Loess**, **ImageInterpolate**, **Interpolate3D**, and **Interp3DPath** operations.

The **Interp2D**, **Interp3D** and **ContourZ** functions.

Interp2D

Interp2D(*srcWaveName, xValue, yValue*)

The Interp2D function returns a double precision number as the interpolated value for the *xValue*, *yValue* point in the source wave. Returns NaN if the point is outside the source wave domain or if the source wave is complex.

Parameters

srcWaveName is the name of a 2D wave. The wave can not be complex.

xValue is the X-location of the interpolated point.

yValue is the Y-location of the interpolated point.

See Also

The **ImageInterpolate** operation. **Interpolation** on page III-121.

Interp3D

Interp3D(*srcWave, x, y, z* [, *triangulationWave*])

The Interp3D function returns an interpolated value for location P=(*x*, *y*, *z*) in a 3D scalar distribution *srcWave*.

If *srcWave* is a 3D wave containing a scalar distribution sampled on a regular lattice, the function returns a linearly interpolated value for any P=(*x*, *y*, *z*) location within the domain of *srcWave*. If P is outside the domain, the function returns NaN.

To interpolate a 3D scalar distribution that is not sampled on a regular lattice, *srcWave* is a four column 2D wave where the columns correspond to *x*, *y*, *z*, f(*z*, *y*, *z*), respectively. You must also use a "triangulation" wave for *srcWave* (use `Triangulate3D/out=1` to obtain the triangulation wave). If P falls within the convex domain defined by the tetrahedra in *triangulationWave*, the function returns the barycentric linear interpolation for P using the tetrahedron where P is found. If P is outside the convex domain the function returns NaN.

Examples

```
Make/O/N=(10,20,30) ddd=gnoise(10)
Print interp3D(ddd,1,0,0)
Print interp3D(ddd,1,1,1)
```

```
Make/O/N=(10,4) ddd=gnoise(10)
Triangulate3D/OUT=1 ddd
Print interp3D(ddd,1,0,0,M_3DVertexList)
Print interp3D(ddd,1,1,1,M_3DVertexList)
```

See Also

The **Interpolate3D** operation. **Interpolation** on page III-121.

Interp3DPath

Interp3DPath *3dWave tripletPathWave*

The Interp3DPath operation computes the trilinear interpolated values of *3dWave* for each position specified by a row of in *tripletPathWave*, which is a three-column wave in which the first column represents the X coordinate, the second represents the Y coordinate and the third represents the Z coordinate. Interp3DPath stores the resulting interpolated values in the wave W_Interpolated. Interp3DPath is equivalent to calling the Interp3D() function for each row in *tripletPathWave* but it is computationally more efficient.

If the position specified by the *tripletPathWave* is outside the definition of the *3dWave* or if it contains a NaN, the operation stores a NaN in the corresponding output entry.

Both *3dWave* and *tripletPathWave* can be of any numeric type. W_Interpolated is always of type NT_FP64.

See Also

The **ImageInterpolate** operation and the **Interp3D** and **interp** functions. **Interpolation** on page III-121.

Interpolate3D

Interpolate3D [/Z] /RNGX={*x0,dx,nx*}/RNGY={*y0,dy,ny*}/RNGZ={*z0,dz,nz*}
/DEST=*dataFolderAndName*, triangulationWave=*tWave*, srcWave=*sWave*

The Interpolate3D operation uses a precomputed triangulation of *sWave* (see **Triangulate3D**) to calculate regularly spaced interpolated values from an irregularly spaced source. The interpolated values are calculated for a lattice defined by the range flags /RNGX, /RNGY, and /RNGZ. *sWave* is a 4 column wave where the first three columns contain the spatial coordinates and the fourth column contains the associated scalar value. Interpolate3D is essentially equivalent to calling the **Interp3D** function for each interpolated point in the range but it is much more efficient.

Parameters

triangulationWave=*tWave*
: Specifies a 2D index wave, *tWave*, in which each row corresponds to one tetrahedron and each column (tetrahedron vertex) is represented by an index of a row in *sWave*. Use **Triangulate3D** with /OUT=1 to obtain *tWave*.

srcWave=*sWave*
: Specifies a real-valued 4 column 2D source wave, *sWave*, in which columns correspond to *x*, *y*, *z*, f(*x*, *y*, *z*). Requires that the domain occupied by the set of {*x*, *y*, *z*} be convex.

Flags

/DEST=*dataFolderAndName*
: Saves the result in the specified destination wave. The destination wave will be created or overwritten if it already exists. *dataFolderAndName* can include a full or partial path with the wave name.

/RNGX={*x0*,*dx*,*nx*}
: Specifies the range along the X-axis. The interpolated values start at *x0*. There are *nx* equally spaced interpolated values where the last value is at *x0*+(*nx*-1)*dx*. If you would like to interpolate the data for a single plane you can set the appropriate number of values to 1. For example, a YZ plane would have *nx*=1.

/RNGY={*y0*,*dy*,*ny*}
: Specifies the range along the Y-axis. The interpolated values start at *y0*. There are *nx* equally spaced interpolated values where the last value is at *y0*+(*ny*-1)*dy*. If you would like to interpolate the data for a single plane you can set the appropriate number of values to 1. For example, a XZ plane would have *ny*=1.

/RNGZ={*z0*,*dz*,*nz*}
: Specifies the range along the Z-axis. The interpolated values start at *z0*. There are *nz* equally spaced interpolated values where the last value is at *z0*+(*nz*-1)*dz*. If you would like to interpolate the data for a single plane you can set the appropriate number of values to 1. For example, a XY plane would have *nz*=1.

/Z
: No error reporting.

Details

The triangulation wave defines a set of tetrahedra that spans the convex source domain. If the requested range consists of points outside the domain, the interpolated values will be set to NaN. The interpolation process for points inside the convex domain consists of first finding the tetrahedron in which the point resides and then linearly interpolating the scalar value using the barycentric coordinate of the interpolated point.

In some cases the interpolation may result in NaN values for points that are clearly inside the convex domain. This may happen when the preceding Triangulate3D results in tetrahedra that are too thin. You can try using

Triangulate3D with the flag /OUT=4 to get more specific information about the triangulation. Alternatively you can introduce a slight random perturbation to the input source wave before the triangulation.

Example

```
Function Interpolate3DDemo()
    Make/O/N=(50,4) ddd=gnoise(20)    // First 3 columns store XYZ coordinates
    ddd[][3]=ddd[p][2]                // Fourth column stores a scalar which is set to z
    Triangulate3D ddd                 // Perform the triangulation
    Wave M_3dVertexList
    Interpolate3D /RNGX={-30,1,80}/RNGY={-40,1,80}/RNGZ={-40,1,80}
          /DEST=W_Interp triangulationWave=M_3dVertexList,srcWave=ddd
End
```

See Also

The **Triangulate3D** operation and the **Interp3D** function. **Interpolation** on page III-121.

References

Schneider, P.J., and D. H. Eberly, *Geometric Tools for Computer Graphics*, Morgan Kaufmann, 2003.

inverseErf

inverseErf(*x*)

The inverseErf function returns the inverse of the error function.

Details

The function is calculated using rational approximations in several regions followed by one iteration of Halley's algorithm.

See Also

The **erf**, **erfc**, **dawson**, and **inverseErfc** functions.

inverseErfc

inverseErfc(x)

The inverseErfc function returns the inverse of the complementary error function.

Details

The function is calculated using rational approximations in several regions followed by one iteration of Halley's algorithm.

See Also

The **erf**, **erfc**, **erfcw**, **dawson**, and **inverseErf** functions.

ItemsInList

ItemsInList(*listStr* [, *listSepStr*])

The ItemsInList function returns the number of items in *listStr*. *listStr* should contain items separated by the *listSepStr* character, such as "abc;def;".

Use ItemsInList to count the number of items in a string containing a list of items separated by a single character, such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

If *listStr* is `""` then 0 is returned.

listSepStr is optional. If missing, *listSepStr* is presumed to be ";".

Details

listStr is searched for item strings bound by *listSepStr* on the left and right.

An item can be empty. The lists `"abc;def;;ghi"` and `";abc;def;;ghi;"` have four items (the third item is `""`).

listStr is treated as if it ends with a *listSepStr* even if it doesn't. The search is case-sensitive.

Only the first character of *listSepStr* is used.

Examples

```
Print ItemsInList("wave0;wave1;wave1#1;")          // prints 3
Print ItemsInList("key1=val1,key2=val2", ",")      // prints 2
Print ItemsInList("1 \t 2 \t", "\t")               // prints 2
Print ItemsInList(";")                             // prints 1
Print ItemsInList(";;")                            // prints 2
Print ItemsInList(";a;")                           // prints 2
Print ItemsInList(";;;")                           // prints 3
```

See Also

The **AddListItem**, **StringFromList**, **FindListItem**, **RemoveFromList**, **WaveList**, **TraceNameList**, **StringList**, **VariableList**, and **FunctionList** functions.

j

j

The j function returns the loop index of the 2nd innermost iterate loop in a macro. Not to be used in a function. iterate loops are archaic and should not be used.

jlim

jlim

The jlim function returns the ending loop count for the 2nd inner most iterate loop. Not to be used in a function. iterate loops are archaic and should not be used.

JulianToDate

JulianToDate(*julianDay*, *format*)

The JulianToDate function returns a date string containing the day, month, and year. The input *julianDay* is truncated to an integer.

Parameters

julianDay is the Julian day to be converted.

format specifies the format of the returned date string.

format	Date String
0	mm/dd/year
1	dd/mm/year
2	Tuesday November 15, 2002
3	year mm dd
4	year/mm/dd

See Also

The **dateToJulian** function.

For more information about the Julian calendar see: <http://www.tondering.dk/claus/cal/>.

KillBackground

KillBackground

The KillBackground operation kills the unnamed background task.

KillBackground works only with the unnamed background task. New code should used named background tasks instead. See **Background Tasks** on page IV-279 for details.

Details

You can not call KillBackground from within the background function itself. However, if you return 1 from the background function, instead of the normal 0, Igor will terminate the background task.

See Also

The **BackgroundInfo**, **CtrlBackground**, **CtrlNamedBackground**, **SetBackground**, and **SetProcessSleep** operations; and **Background Tasks** on page IV-279.

KillControl

KillControl [**/W=*winName***] ***controlName***

The KillControl operation kills the named control in the top or specified graph or panel window or subwindow.

If the named control does not exist, KillControl does not complain.

Flags

/W=*winName* Looks for the control in the named graph or panel window or subwindow. If /W is omitted, KillControl looks in the top graph or panel window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

KillDataFolder

KillDataFolder [**/Z**] ***dataFolderSpec***

The KillDataFolder operation kills the specified data folder and everything in it including other data folders.

However, if *dataFolderSpec* is the name of a data folder reference variable that refers to a free data folder, the variable is cleared and the data folder is killed only if this is the last reference to that free data folder.

Flags

/Z No error reporting (except for setting V_flag). Does not halt function execution.

Parameters

dataFolderSpec can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

Details

If specified data folder is the current data folder or contains the current data folder then Igor makes its parent the new current data folder.

For legacy reasons, a null data folder is taken to be the current data folder. This can happen when using a $ expression where the string might possibly evaluate to "".

It is legal to kill the root data folder. In this case the root data folder itself is not killed but everything in it is killed.

KillDataFolder generates an error if any of the waves involved are in use. In this case, nothing is killed.

KillDataFolder generates an error if any of the waves involved are in use. In this case, nothing is killed. Execution ceases unless /Z is specified.

The variable V_flag is set to 0 when there is no error, otherwise it is an error code.

Examples

```
KillDataFolder foo          // Kills foo in the current data folder.
KillDataFolder :bar:foo     // Kills foo in bar in current data folder.
String str= "root:foo"
KillDataFolder $str         // Kills foo in the root data folder.
```

See Also

Chapter II-8, **Data Folders** and the **KillStrings**, **KillVariables**, and **KillWaves** operations.

KillFIFO

KillFIFO ***FIFOName***

The KillFIFO operation discards the named FIFO.

Details

FIFOs are used for data acquisition.

If there is an output or review file associated with the FIFO, KillFIFO closes the file. If the FIFO is used by an XOP, you should call the XOP to release the FIFO before killing it.

See Also

See **FIFOs and Charts** on page IV-276 for information about FIFOs and data acquisition.

KillFreeAxis

KillFreeAxis [**/W=*winName***] ***axisName***

The KillFreeAxis operation removes a free axis specified by *axisName* from a graph window or subwindow.

Flags

/W=*winName* Kills the free axis in the named graph window or subwindow. If /W is omitted, it acts on the top graph window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

Only an axis created by **NewFreeAxis** can be killed and only if no traces or images are attached to the axis.

See Also

The **NewFreeAxis** operation.

KillPath

KillPath [**/A/Z**] ***pathName***

The KillPath operation removes a path from the list of symbolic paths. KillPath is a newer name for the **RemovePath** operation.

Flags

/A Kills all symbolic paths in the experiment except for the built-in paths. Omit *pathName* if you use /A.

/Z Does not generate an error if a path to be killed is a built-in path or does not exist. To kill all paths in the experiment, use `KillPath/A/Z`.

Details

You can't kill the built-in paths "home" and "Igor".

See Also

The **NewPath** operation.

KillPICTs

KillPICTs [**/A/Z**] [***PICTName*** [**,** ***PICTName***]…]

The KillPICTs operation removes one or more named pictures from the current Igor experiment.

Flags

/A Kills all pictures in the experiment.

/Z Does not generate an error if a picture to be killed is in use or does not exist. To kill all pictures in the experiment, use `KillPICTs/A/Z`.

Details

You can not kill a picture that is used in a graph or page layout.

Warning: You *can* kill a picture that is referenced from a graph or layout recreation macro. If you do, the graph or layout can not be completely recreated. Use the Find dialog (Edit menu) to locate references in the procedure window to a named picture you want to kill.

See Also

See **Pictures** on page III-421 for general information on how Igor handles pictures.

KillStrings

KillStrings [**/A/Z**] [***stringName*** [**, *stringName***]…]

The KillStrings operation discards the named global strings.

Flags

/A Kills all global strings in the current data folder. If you use /A, omit *stringName*.

/Z Does not generate an error if a global string to be killed does not exist. To kill all global strings in the current data folder, use `KillStrings/A/Z`.

KillVariables

KillVariables [**/A/Z**] [***variableName*** [**, *variableName***]…]

The KillVariables operation discards the named global numeric variables.

Flags

/A Kills all global variables in the current data folder. If you use /A, omit *variableName*.

/Z Does not generate an error if a global variable to be killed does not exist. To kill all global variables in the current data folder, use `KillVariables/A/Z`.

KillWaves

KillWaves [***flags***] ***waveName*** [**, *waveName***]…

The KillWaves operation destroys the named waves.

Flags

/A Kills all waves in the current data folder. If you use /A, omit *waveName*s.

/F Deletes the Igor binary file from which *waveName* was loaded.

/Z Does not generate an error if a wave to be killed is in use or does not exist.

Details

The memory the waves occupied becomes available for other uses. You can't kill a wave used in a graph, table or user defined function, or which is reserved by an XOP.

XOPs reserve a wave by sending the OBJINUSE message.

Examples

```
KillWaves/A/Z       // kill waves not in use in current data folder
```

KillWindow

KillWindow *winName*

The KillWindow operation kills or closes a specified window or subwindow without saving a recreation macro.

Parameters

winName is the name of an existing window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

See Also

The **DoWindow** operation.

KMeans

KMeans [***flags***] ***populationWave***

The KMeans operation analyzes the clustering of data in *populationWave* using an iterative algorithm. The result of KMeans is a specification of the classes which is saved in the wave M_KMClasses in the current data folder. Optional results include the distribution of class members (W_KMMembers) and the inter-class distances. *populationWave* is a 2D wave in which columns correspond to members of the population and rows contain the dimensional information.

Flags

/CAN
Analyzes the clustering by computing Euclidean distances between the means of the resulting classes. The resulting distances are stored in an NxN square matrix where N is the number of classes. Self distances (along the diagonal) or distances involving classes that did not survive the iterations are filled with NaN. Also saves the wave W_KMDispersion, which contains the sum of the distances between the center of each class and all its members. Distances are evaluated using the method specified by /DIST.

/DEAD=*method*
Specifies how the algorithm should handle "dead" classes, which are those that lose all members in a given iteration.

method=1: Remove the class if it looses all members.

method=2: Default; keeps the last value of the mean vector in case the class might get new members in a subsequent iteration.

method=3: Assigns the class a random mean vector.

/DIST=*mode*
Specifies how the class distances are evaluated.

mode=1: Distance is evaluated as the sum of the absolute values (also known as Manhattan distance).

mode=2: Default; distance is evaluated as Euclidean distance.

/INIT=*method*
Specifies the initialization method.

method=1: Random assignment of members of the population to a class.

method=2: User-specified mean values (/INW).

method=3: Default; initialize classes using values of a random selection from the population.

/INW=*iWave*
Sets the initial classes. The number of rows of *iWave* equals the dimensionality of the class and the number of columns of *iWave* is the number of classes. For example, if we want to initialize 5 classes in a problem that involves position in two dimensions then *iWave* must have 2 rows and 5 columns. The number of rows must also match the number of rows in *populationWave*.

/NCLS=*num*
Sets the number of classes in the data. If the initialization method uses specific means (/INIT=2) then the number of columns of *iWave* (see /INW) must match *num*. The default number of classes is 2.

/OUT=*format*
Specifies the format for the results.

format=1: Output only the specification of the classes in the 2D wave M_KMClasses (default). Each column in M_KMClasses represents a class. The number of rows in M_KMClasses is equal to the number of rows in `populationWave+1`. The last row contains the number of class members. The remaining rows represent the center of the class. For example, if *populationWave* has two rows then the dimensionality of the problem is 2 and M_KMClasses has 3 rows with the first row containing the first components of each class center, the second row containing the second components of each class center and the third row containing the number of elements in each class.

format=2: Output (in addition to M_KMClasses) the class membership in the wave W_KMMembers. The rows in this 1D wave correspond to sequential members of *populationWave* and the entries correspond to the (zero based) column number in M_KMClasses.

/SEED=*val*
Sets the seed for a new sequence in the pseudo-random number generator that is used by the operation. *val* must be an integer greater than zero.

By changing the sequence you may be able to find new solutions or just make the process converge at a different rate.

/TER=*method*
Determines when the iterations stop.

method=1: User-specified number of iterations (/TERN).

method=2: Default; continue iterating until no more than a fixed number of elements change classes in one iteration (TERN).

/TERN=*num*
Specifies the termination number. The meaning of the number is determined by /TER above. By default, the termination *method*=2 and the default value of the maximum number of elements that change classes in one iteration is 5% of the size of the population.

/Z No error reporting. If an error occurs, sets V_flag to -1 but does not halt function execution.

Details

KMeans uses an iterative algorithm to analyze the clustering of data. The algorithm is not guaranteed to find a global optimum (maximum likelihood solution) so the operation provides various flags to control when the iterations terminate. You can determine if the operation iterates a fixed number of times or loops until at most a specified maximum number of elements change class membership in a single iteration. If you are computing KMeans in more than one dimension you should pay attention to the relative magnitudes of the data in each dimension. For example, if your data is distributed on the interval [0,1] in the first dimension and on the interval [0,1e7] in the second dimension, the operation will be biased by the much larger magnitude of values in the second dimension.

Examples

Create data with 3 classes:

```
Make/O/N=(1,128) jack=4+gnoise(1)
jack[0][15,50]+=10
jack[0][60,]+=20
```

Perform KMeans looking for 5 classes:

```
KMeans/init=1/out=1/ter=1/dead=1/tern=1000/ncls=5 jack
Print M_KMClasses
  M_KMClasses[0][0]= {24.1439,68}
  M_KMClasses[0][1]= {14.1026,36}
  M_KMClasses[0][2]= {4.01537,24}
```

See Also

The **FPClustering** function.

References

A nice overview of k-means classification can be found at:
<http://www.ece.neu.edu/groups/rpl/projects/kmeans/index.html>.

Label

Label [**/W=*winName*/Z**] ***axisName, labelStr***

The Label operation labels the named axis with *labelStr*.

Parameters

axisName is the name of an existing axis in the top graph. It is usually one of “left”, “right”, “top” or “bottom”, though it may also be the name of a free axis such as “VertCrossing”.

labelStr contains the text that labels the axis.

Flags

/W=*winName* Adds axis label in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/Z No errors generated if the named axis doesn’t exist. Used for style macros.

Details

labelStr can contain the following escape codes which affect subsequent characters in the string:

\B Use subscript (in smaller type).

\F'*fontName*' Use specified font (e.g., `\F'Helvetica'`).

\f*dd* *dd* is a binary coded number with each bit controlling one aspect of the column’s font style as follows:

bit 0: Bold.

bit 1: Italic.

bit 2: Underline.

bit 3: Outline (*Macintosh only*).

bit 4: Shadow (*Macintosh only*).

For example, bold underline is $2^0 + 2^2 = 1 + 4 = 5$. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

\K(*r*,*g*,*b*)	Use specified color for text. *r*, *g*, and *b* are integers from 0 to 65535.
\M	Use normal script (reverts to main line and font size).
).	
\S	Use superscript (in smaller type).
\Z*nn*	Use font size *nn*, which must be exactly two digits.

labelStr can also contain the following special codes that insert dynamically computed substrings:

\c	Inserts the name of the wave that controls the axis. This is the first wave graphed against that axis.
\E	Inserts power of 10 scaling with leading x. This can be ambiguous; we recommend that you use either \U or \u.
\e	Like \E but inverts the sign of the exponent. Even more ambiguous than \E.
\s(*traceName*)	Inserts a wave symbol for the named trace.
\U	Inserts units with automatic prefixes.
\u	Inserts power of 10 scaling but without the leading x as used by \E. No action if axis is not scaled. Use in front of custom or compound unit strings. Example label: "`Field Strength (\\u Volts/Meter)`" will produce something like "Field Strength (10^6 Volts/Meter)".
\u#1	This is a variant of \u that inserts the inverse of \u (e.g., 10^{-6} instead of 10^6).
\u#2	Prevents automatic insertion of any units or scaling. Normally, if you set a wave's units and scaling, using the Change Wave Scaling dialog or **SetScale** operation, and if you do not explicitly specify an axis label, Igor will automatically generate an axis label from the units and scaling. \u#2 provides a way to suppress this behavior when it gets in the way.

Note that escape codes are case sensitive; \u and \U insert different substrings.

labelStr can also contain other infrequently-used escape sequences as documented for the **TextBox** operation.

Each backslash character should be preceded with another backslash. This is because backslash is itself a special escape character for strings. Future Igor Pro versions may require this double-backslash syntax; currently either the single or double backslash are acceptable synonyms for inserting a single backslash into a string. For example:

```
String myStr = "\\"
Print myStr             // Prints \ to the history area
```

myStr was set to a *single* backslash character. See **Escape Characters in Strings** on page IV-13 for more about the backslash character. This behavior affects how escape sequences in *labelStr* should be written:

```
Label top, "\Z14Stuff"     // no: In future, may come out as "Z14Stuff"
Label top, "\\Z14Stuff"    // yes
```

You can see how *labelStr* should be constructed by using the Axis Label tab and observing the command it creates. You will observe many double backslash sequences. Single backslash sequences are deprecated, and are currently accepted for backward compatibility reasons.

The characters "<??>" in an axis label indicate that you specified an invalid escape code or used a font that is not available.

See Also

See **About Text Info Variables** on page III-64 for *labelStr* escape codes that manipulate text info variables. See **Escape Characters in Strings** on page IV-13 for more about the backslash character. See the **Legend** operation about wave symbols.

laguerre

laguerre(*n*, *x*)

The laguerre function returns the Laguerre polynomial of degree *n* (positive integer) and argument *x*. The polynomials satisfy the recurrence relation:

$$(n+1)Laguerre(n+1,x) = (2n+1-x)Laguerre(n,x) - nLaguerre(n-1,x)$$

with the initial conditions $Laguerre(0,x) = 1$ and $Laguerre(1,x) = 1-x$.

See Also

The **chebyshev**, **chebyshevU**, **hermite**, **hermiteGauss**, and **legendreA** functions.

laguerreA

laguerreA(*n*, *k*, *x*)

The laguerreA function returns the associated Laguerre polynomial of degree *n* (positive integer), index *k* (non-negative integer) and argument *x*. The associated Laguerre polynomials are defined by

$$L_n^k(x) = (-1)^k \frac{d^k}{dx^k}[L_{n+k}(x)]$$ where $L_{n+k}(x)$ is the Laguerre polynomial.

See Also

The **laguerre** and **laguerreGauss** functions.

References

Arfken, G., *Mathematical Methods for Physicists*, Academic Press, New York, 1985.

laguerreGauss

laguerreGauss(*p*, *m*, *r*)

The laguerreGauss function returns the normalized product of the associated Laguerre polynomials and a Gaussian. This function is typically encountered in solutions to physical problems where it represents the radial solution with an additional factor exp(i*m*ϕ) which is not included in this case. The LaguerreGauss is given by

$$U_{pm}(r) = \left[\frac{2p!}{\pi(m+p)!}\right]^{1/2} (\sqrt{2}r)^m L^m{}_p(2r^2)\exp(-r^2)$$

See Also

The **laguerre**, **laguerreA**, and **hermiteGauss** functions.

Layout

Layout [*flags*] [*objectSpec* [, *objectSpec*]…] [**as** *titleStr*]

The Layout operation creates a page layout.

Note: The Layout operation is antiquated and can not be used in user-defined functions. For new programming, use the **NewLayout** operation instead.

Parameters

All of the parameters are optional.

Each *objectSpec* parameter identifies a graph, table, textbox or picture to be added to the layout. An object specification can also specify the location and size of the object, whether the object should have a frame or not, whether it should be transparent or opaque, and whether it should be displayed in high fidelity or not. See **Details**.

titleStr is a string expression containing the layout's title. If not specified, Igor will provide one which identifies the objects displayed in the graph.

Flags

/A=(*rows*,*cols*)	Specifies rows and columns for tiling or stacking.
/B=(*r*,*g*,*b*)	Specifies the background color for the layout. *r*, *g*, and *b* are integers from 0 to 65535. Defaults to white (65535,65535, 65535).
/C=*colorOnScreen*	Obsolete. Prior to Igor Pro 5, this flag switched the screen display of the layout between black and white and color. Now layouts are always displayed in color. This flag has no effect but is still accepted.
/G=*g*	Specifies grout, the spacing between tiled objects. Units are points unless /I, /M, or /R are specified.
/HIDE=*h*	Hides (h = 1) or shows (h = 0, default) the window.

/I — Specifies that coordinates are in inches. This affects subsequent /G, /W, and *objectSpec* coordinates. Coordinates are relative to the top/left corner of the paper.

/K=*k* — Specifies window behavior when the user attempts to close it.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

k=3: Hides the window.

If you use /K=2 or /K=3, the only way to kill the window is via the DoWindow/K operation.

/M — Specifies that coordinates are in centimeters. This affects subsequent /G, /W, and *objectSpec* coordinates. Coordinates are relative to the top/left corner of the paper.

/P=*orientation* — *orientation* is either Portrait or Landscape (*e.g.*, `Layout/P= Landscape`). This controls the orientation of the page in the layout. See **Details**.

If you use the /P flag, you should make it the first flag in the Layout operation. This is necessary because the orientation of the page affects the behavior of other flags, such as /T and /G.

/R — Specifies that coordinates are in percent. This affects subsequent /G, /W, and *objectSpec* coordinates. For /W, coordinates are as a percent of the main screen. For /G and *objectSpec*, coordinates are relative to the top/left corner of the printing part of the page.

/S — Stacks objects.

/T — Tiles objects.

/W=(*left*, *top*, *right*, *bottom*) — Gives the layout window a specific location and size on the screen. Coordinates for /W are in points unless /I or /M are specified.

Details

The orientation of the page is controlled by the page setup record associated with the layout. When you create a brand new layout window, the page setup record comes from your preferred page setup (which you specify via the Capture Layout Prefs dialog) or from a default page setup that Igor creates by calling the current printer driver. When you recreate a layout window using a Window macro, Igor reuses the page setup originally used for the layout window.

If you use the /P flag, you should make it the first flag in the Layout operation. This is necessary because the orientation of the page affects the behavior of other flags, such as /T and /G.

The form of an *objectSpec* is:

```
objectName [(objLeft, objTop, objRight, objBottom)][/O=objType][/F=frame]
[/T=trans][/D=fidelity]
```

objectName can be the name of an existing graph, table or picture. It can also be the name of an object that does not yet exist. In this case it is called a "dummy object".

objectSpec can be specified using a string by using the $ operator, but the entire *objectSpec* must be in the string.

Here are some examples of valid usage:

```
Layout Graph0

Layout/I Graph0(1, 1, 6, 5)/F=1

String s = "Graph0"
Layout/I $s

String s = "Graph0(1, 1, 6, 5)/F=1"
Layout/I $s         // Entire object spec is in string.
```

The object's coordinates are determined as follows:

- If *objectName* is followed by a coordinates specification in (*objLeft*, *objTop*, *objRight*, *objBottom*) form then this sets the object's coordinates. The units for the coordinates are points unless the /I or /M flag was present in which case the units are inches or centimeters respectively.
- If the object coordinates are not specified explicitly but the Layout/S flag was present then the object is stacked. If the Layout/T flag was present then the object is tiled, and if the Layout/A=(*rows*,*cols*) flag is present, tiling is performed using that number of rows and columns.

- If the object's coordinates are not determined by these rules then the object is set to a default size and is stacked.

Each object has a type (graph, table, textbox or picture) determined as follows:

If the *objectName*/O=*objType* flag is present then it determines the object's type:

/O=*objType*	*objType*=1:	Graph.
	objType=2:	Table.
	objType=8:	Picture.
	objType=32:	Textbox.

If there is no /O flag and *objectName* is the name of an existing graph, table or picture, then the object type is graph, table or picture.

If the object's type is not determined by the above rules and *objectName* contains "Table", "PICT", or "TextBox", then the object type is table, picture or textbox.

If the object's type is not specified by any of the above rules, it is taken to be a graph type object.

The remaining flags have the following meanings:

/D=*fidelity*	*fidelity*=0:	Low fidelity display.
	fidelity=1:	High fidelity display (default).
/F=*frame*	*frame*=0:	No frame.
	frame=1:	Single frame (default).
	frame=2:	Double frame.
	frame=3:	Triple frame.
	frame=4:	Shadow frame.
/T=*trans*	*trans*=0:	Opaque (default).
	trans=1:	Transparent. For this to be effective, the object itself must also be transparent. Annotations have their own transparent/opaque settings. Graphs are transparent only if their backgrounds are white. Pictures may have been created transparent or opaque, and Igor cannot make an inherently opaque picture transparent.

See Also

The **NewLayout** and **LayoutInfo** operations. See Chapter II-16, **Page Layouts**.

Layout

Layout

Layout is a procedure subtype keyword that identifies a macro as being a page layout recreation macro. It is automatically used when Igor creates a window recreation macro for a layout. See **Procedure Subtypes** on page IV-179 and **Killing and Recreating a Layout** on page II-369 for details.

See Also

See Chapter II-16, **Page Layouts**.

LayoutInfo

LayoutInfo(*winNameStr, itemNameStr*)

The LayoutInfo function returns a string containing a semicolon-separated list of keywords and values that describe an object in a page layout or overall properties of the page layout. The main purpose of LayoutInfo is to allow an advanced Igor programmer to write a procedure which formats or arranges objects.

winNameStr is the name of an existing page layout window or "" to refer to the top layout.

itemNameStr is a string expression containing one of the following:

1. The name (e.g., "Graph0") of a layout object to get information about that object.
2. An object instance (e.g., "Graph0#0" or "Graph0#1") to get information about a particular instance of an object. This is of use only in the unusual situation when the same object appears in the layout multiple times. "Graph0#0" is equivalent to "Graph0". "Graph0#1" is the second occurrence of Graph0 in the layout.

3. An integer object index starting from zero to get information about an object referenced by its position in the layout. Zero refers to the first object going from back to front in the layout.
4. The word "Layout" to get overall information about the layout.

Details

In cases 1, 2 and 3 above, where *itemNameStr* references an object, the returned string contains the following keywords, with a semicolon after each keyword-value pair.

Keyword	Information Following Keyword
FIDELITY	Object fidelity expressed as a code usable in a ModifyLayout fidelity command.
FRAME	Object frame expressed as a code usable in a ModifyLayout frame command.
HEIGHT	Object height in points.
INDEX	Object position in the back-to-front order of the layout, starting from zero.
LEFT	Object left position in points.
NAME	The name of the object.
SELECTED	Zero if the object is not selected or nonzero if it is selected. You can identify the first-selected object by examining the SELECTED code of all objects. The one with the smallest nonzero selected code is the object that was first selected.
TOP	Object top position in points.
TRANS	Object transparency expressed as a code usable in a ModifyLayout trans command.
TYPE	Object type which is one of: Graph, Table, Picture, or Textbox.
WIDTH	Object width in points.

In case 4 above, where *itemNameStr* is "Layout", the returned string contains the following keywords, with a semicolon after each keyword-value pair.

Keyword	Information Following Keyword
BGRGB	Layout background color expressed as <red>, <green>, <blue> where each color is a value from 0 to 65535.
MAG	Layout magnification: 0.25, 0.5, 1.0, or 2.0.
NUMOJBECTS	Total number of objects in the layout.
NUMSELECTED	Number of selected objects.
PAGE	A rectangle defining the part of the paper that is inside the margins, expressed in points. The format is <left>, <top>, <right>, <bottom>.
PAPER	A rectangle defining the bounds of the paper, expressed in points. The format is <left>, <top>, <right>, <bottom>.
SELECTED	A comma-separated list of the names of selected objects.
UNITS	Units used to display object locations and sizes. This will be one of the following: 0 for points, 1 for inches, 2 for centimeters.

LayoutInfo returns `""` in the following situations:

winNameStr is `""` and there are no layout windows.

winNameStr is a name but there are no layout windows with that name.

itemNameStr is not "Layout" and is not the name or index of an existing object.

Examples

This example sets the background color of all selected graphs in a particular page layout to the color specified by `red`, `green`, and `blue`, which are numbers from 0 to 65535.

```
Function SetLayoutGraphsBackgroundColor(layoutName,red,green,blue)
   String layoutName      // Name of layout or "" for top layout.
   Variable red, green, blue

   Variable index
   String info
   Variable selected
   String indexStr
   String objectTypeStr
   String graphNameStr

   index = 0
   do
      sprintf indexStr, "%d", index
      info = LayoutInfo(layoutName, indexStr)
      if (strlen(info) == 0)
         break     // No more objects
      endif

      selected = NumberByKey("SELECTED", info)
      if (selected)
         objectTypeStr = StringByKey("TYPE", info)
         if (CmpStr(objectTypeStr,"Graph") == 0)// This is a graph?
            graphNameStr = StringByKey("NAME", info)
            ModifyGraph/W=$graphNameStr wbRGB=(red,green,blue)
            ModifyGraph/W=$graphNameStr gbRGB=(red,green,blue)
         endif
      endif

      index += 1
   while(1)
End
```

See Also

The **Layout** operation. See Chapter II-16, **Page Layouts**.

LayoutMarquee

LayoutMarquee

LayoutMarquee is a procedure subtype keyword that puts the name of the procedure in the layout Marquee menu. See **Marquee Menu as Input Device** on page IV-140 for details.

See Also

See Chapter II-16, **Page Layouts**.

LayoutStyle

LayoutStyle

LayoutStyle is a procedure subtype keyword that puts the name of the procedure in the Style pop-up menu of the New Layout dialog and in the Layout Macros menu. See **Layout Style Macros** on page II-389 for details.

See Also

See Chapter II-16, **Page Layouts** and **Layout Style Macros** on page II-389.

leftx

leftx(*waveName*)

The leftx function returns the X value of point 0 (the first point) of the named 1D wave. The leftx function is not multidimensional aware. The multidimensional equivalent of this function is **DimOffset**.

Details

Point 0 contains a wave's *first* value, which is usually the leftmost point when displayed in a graph. Leftx returns the value elsewhere called *x0*. The function DimOffset returns any of x0, y0, z0, or t0, for dimensions 0, 1, 2, or 3.

See Also

The **deltax** and **rightx** functions.

For multidimensional waves, see **DimDelta**, **DimOffset**, and **DimSize**.

For an explanation of waves and X scaling, see **Changing Dimension and Data Scaling** on page II-83.

Legend

Legend [*flags*] [*legendStr*]

The Legend operation puts a legend on a graph or page layout.

Parameters

legendStr contains the text that is printed in the legend.

If *legendStr* is missing or is an empty string (""), the text needed for a default legend is automatically generated. Legends are automatically updated when waves are appended to or removed from the graph or when you rename a wave in the graph.

See **Legend Text** on page III-52 for a discussion of what *legendStr* may contain.

Flags

/H=*legendSymbolWidth*

Sets the width in points of the area in which to draw the wave symbols. A value of 0 means "default". This results in a width that is based on the text size in effect when the symbol is drawn. A value of 36 gives a 0.5 inch (36 points) width which is nice in most cases.

/H={*legendSymbolWidth*, *minThickness*, *maxThickness*}

A newer form (Igor Pro 4.0) of the /H flag. The *legendSymbolWidth* parameter works the same as described above.

The *minThickness* and *maxThickness* parameters allow you to create a legend whose line and marker thicknesses are different from the thicknesses of the associated traces in the graph. This can be handy to make the legend more readable when you use very thin lines or markers for the traces.

minThickness and *maxThickness* are values from 0.0 to 10.0. Also, setting *minThickness* to 0.0 and *maxThickness* to 0.0 (default) uses the same thicknesses for the legend symbols as for the traces.

/J

Disables the default legend mechanism so that a default legend is not created even if *legendStr* is an empty string ("") or omitted.

Window recreation macros use /J in case *legendStr* is too long to fit on the same command line as the Legend operation itself. In this case, an AppendText command appears after the Legend command to append *legendStr* to the empty legend. For really long values of *legendStr*, there may be multiple AppendText commands.

/M[=*saMeSize*]

/M or /M=1 specifies that legend markers should be the same size as the marker in the graph.

/M=0 turns same-size mode off so that the size of the marker in the legend is based on text size.

For all other flags:

/A=*anchorCode*	/C	/E[=*exterior*]	/F=*frame*	/K
/N=*name*	/R=*newName*	/S=*style*	/T=*tabSpec*	/V=*vis*
/W=*winName*	/X=*xoffset*	/Y=*yoffset*		

see the **TextBox** and **AppendText** operations. Also see **ColorScale** and **Tag**.

Examples

The command `Legend` (with no parameters) creates a default legend. A default legend in a layout contains a line for each wave in each of the graphs in the layout, starting from the bottom graph and working toward the front.

The command:

```
Legend/C/N=name ""
```

changes the named existing legend to a default legend.

You can put a legend in a page layout with a command such as:

```
Legend "\s(Graph0.wave0) this is wave0"
```

This creates a legend in the layout that shows the symbol for wave0 in Graph0. The graph named in the command is usually in the layout but it doesn't have to be.

See Also
Legend Text on page III-52 and the **TextBox**, **Tag**, and **ColorScale** operations.

legendreA

legendreA(*n*, *m*, *x*)

The legendreA function returns the associated Legendre polynomial: $P_n^m(x)$ where n and m are integers such that $0 \le m \le n$ and $|x| \le 1$.

References
Arfken, G., *Mathematical Methods for Physicists*, Academic Press, New York, 1985.

limit

limit(*num*, *low*, *high*)
The limit function returns *num*, limited to the range from *low* to *high*:

num if *low* <= *num* <= *high*.

low if *num* < *low*.

high if *num* > *high*.

See Also
SelectNumber function.

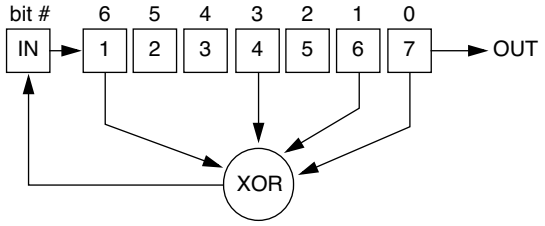
LinearFeedbackShiftRegister

LinearFeedbackShiftRegister [***flags***]
The LinearFeedbackShiftRegister operation implements a, well, linear feedback shift register, or LFSR. A LFSR is a way to produce a sequence of very bad pseudorandom numbers, or a random bit stream (that is, a random sequence of zeroes of ones that over time are nearly equal in number).

If it produces bad random numbers, why would I want to use a LFSR? A properly-configured LFSR will create a "maximal-length sequence": a LFSR of N bits will produce 2^N-1 numbers in a quasi-random sequence without repeating. That is, it will produce all the N-bit numbers except zero. This gives the sequence good spectral properties for certain applications, and, taking the least-significant bit as the output, it creates a pseudorandom bit stream with nearly equal numbers of zeroes and ones (*nearly* means one more one than zeroes).

The LinearFeedbackShiftRegister operation generates either a wave full of the sequential states of the shift register or a wave full of ones and zeroes representing the least significant bit of the shift register.

Linear Feedback Shift Registers
A LFSR is a shift register with taps. The tap bits are XOR'ed together and the result, after the register is shifted, becomes the new most significant bit. Here is a diagram of a 7-bit LFSR:



Each successive number is generated by shifting the contents of the register (boxes 1-7) to the right, while shifting in the output of the XOR node. The XOR node samples specified bits of the register contents, generating its output ready to be shifted in. Thus, the inputs of the XOR are bits sampled *before* a shift; the output of the XOR becomes the leading bit in the register *after* a shift.

In many applications the output of interest is the stream of bits that appear in the last position. This stream of bits is a pseudorandom sequence of ones and zeroes (or ones and minus ones, or whatever other binary sequence you need).

The bits fed into the XOR node are referred to as *taps*. The taps illustrated here would be specified with the tap list 7,6,4,1. As implemented in Igor Pro, the output tap (tap 7 in the illustration) is the least significant bit, so an alternate way to express the tap list is as the binary number 1001011_2 (77_{10}).

With the right taps, a LFSR produces a maximal-length sequence. The list of sequential states in a maximal-length sequence has length 2^N-1 without repeating a state. That means that every possible N-bit nonzero number appears exactly once in the maximal-length sequence.

Maximal-length tap lists always have an even number of taps.

If you have a tap list that gives a maximal-length sequence, you can generate another tap list from it. If your tap list is (n, A, B, C) the new tap list is (n, n-C, n-B, n-A). This new tap list will generate a bit stream that is the mirror image in time of the bit stream produced by the first tap list.

Flags

/DEST=*wavename* Specifies a wave to receive the generated sequence. With /MODE=0, the number type of the wave must have at least *nbits* bits for an integer wave, or at least an *nbit* mantissa if it is a floating-point wave. That is, /N=25 requires a double-precision wave or a 32-bit integer wave. /N=18 requires any floating-point wave or a 16- or 32-bit integer wave. If you use an integer wave, we recommend an unsigned integer wave for /N=8, 16, or 32.

If /MODE=1 is used, any number type is acceptable. See the Details for what happens if you don't use /DEST.

If *wavename* doesn't exist, a suitable integer wave will be made.

If *wavename* already exists, LinearFeedbackShiftRegister will use it as-is. The sequence length will be taken from the wave. If the number type of the wave is not suitable, an error is issued. If the sequence length is less than the number of points in your wave, it will be truncated to match.

/FREE In a user-defined function, makes a free wave. See **Free Waves** on page IV-71 for details.

/INIT=*initialValue* Sets the initial value of the shift register to *initialValue*. This will also be the first value in the output for /MODE=0, or the least-significant bit of *initialValue* will be the first output for /MODE=1. You can use this initial value to restart a very long sequence from the last state of a previous run.

Default is a single 1 bit in the first position (bit *nbits*-1 for /N=*nbits*).

/LEN=*length* Sets the length of sequence to generate. If the sequence repeats before *length* states are generated, the sequence is terminated early. If *length* is larger than the number of states in a maximal-length sequence, you will get a maximal-length sequence, or a shorter sequence if the initial value is seen again (that is, your sequence is not a maximal-length sequence).

You can specify *length* greater than the maximal-length sequence length, but it will be truncated to the maximal length.

/MAX=*index* An internal table of tap lists gives maximal-length sequences. This table has up to 32 tap lists for each value of *nbits*. You select a tap list by setting index to a number from 0 to 31. For values of *nbits* that do not have 32 maximal-length tap lists, the table repeats. Most *nbits* values have many more than 32 possible maximal-length sequences. For each tap list in the table, another tap list can be accessed using the /MROR flag.

/MODE=*doBitStream*

Sets the output stream format.

doBitStream=0: Succession of bit register states (default).

doBitStream=1: Stream of ones and zeroes.

/MROR [=*doMirror*] Transforms the tap list into its complementary tap list, creating a mirror-image bit stream, when you use /MROR or *doMirror*=1. Specify the tap list using /TAPS, /TAPB, or /MAX.

/N=*nbits* Determines the number of bits in the shift register. A maximal-length sequence will have 2^{nbits}-1 states. *nbits* must be in the range of 1-32. Note that *nbits* = 1 or 2 is not very interesting.

/STOP=*stopValue* Terminates the sequence when *stopValue* is the next shift register value. You can use this flag to generate long sequences using multiple calls to LinearFeedbackShiftRegister by storing the initial value of the first call, and setting *stopValue* to that initial value in subsequent calls.

/TAPB=*tapbits* An alternate way to express the tap list. *tapbits* is a number in which each bit represents a tap, with bit 0 representing the tap with tap number *nbits*.

/TAPS={*t1*, *t2*, …} Specifies the tap list. Tap numbers are in the range from 1 to *nbits*.

Details

If the /TAPS, /TAPB, or /MAX flags are absent, the maximal-length sequence corresponding to /MAX=0 is generated.

In you omit the /DEST flag, a wave named W_LFSR will be generated for you. W_LFSR is an unsigned integer wave with number type set to the minimum size for the shift register size and /MODE setting. Thus, if you set `/N=10/MODE=0`, W_LFSR will be an unsigned 16-bit integer wave.

Because W_LFSR is an unsigned integer wave, you will need to redimension the wave to a floating-point wave for many purposes. Use the **Redimension** operation or the Redimension Waves item in the Data menu.

Up to /N=18, W_LFSR will initially be created large enough to hold a maximal-length sequence, unless you request a shorter sequence using the /LEN flag. If the initial value is seen again before a maximal-length sequence is generated, it means that the tap list specified was not one that generates a maximal-length sequence, and generation is terminated. The wave is shortened to the generated sequence length.

If you set a register size greater than /N=18 and you do not use /LEN, the generated sequence will stop after 2^{18}-1 (262143) states. Note that beyond some N, it will be impossible to create a wave large enough to hold a maximal-length sequence.

Some tap lists do not generate maximal-length sequences but also do not repeat the initial value. In that case, the generated sequence will be of maximal length but will contain repeated subsequences. The V_flag variable will be set to 0 if the sequence was not a maximal-length sequence, or 1 if it was. If /LEN=*length* values were generated, V_flag is set to 2.

If you specify your own wave using /DEST, the sequence length will be the same as the length of your wave. Your wave will be resized if a shorter sequence is generated.

Generating Long Sequences in Smaller Segments

Very long maximal-length sequences will not fit in the largest wave you can make. It may also be more convenient to make multiple, small fragments of a longer sequence. You can do this using the /INIT, /STOP, and /LEN flags, along with the V_nextValue variable. Here is an example of making 1000-point subsequences from a 16-bit maximal-length sequence:

```
// Start with the first 1000 states, with initial value of 1
LinearFeedbackShiftRegister/N=16/LEN=1000/INIT=1
// Restart using the V_nextValue variable to continue the sequence
// /STOP=1 sets the stopping value to the first initial value
LinearFeedbackShiftRegister/N=16/LEN=1000/INIT=(V_nextValue)/STOP=1
// Continue…
LinearFeedbackShiftRegister/N=16/LEN=1000/INIT=(V_nextValue)/STOP=1
```

Variables

The LinearFeedbackShiftRegister operation returns information in the following variables:

`V_flag` Set to zero when a nonmaximal-length sequence was detected. This occurs if the initial value is seen again before a maximal-length sequence was generated, or if a maximal-length sequence was generated but the final state was not the same as the initial state.

Set to 1 when a maximal-length sequence was generated.

Set to 2 when the sequence was limited by /LEN=*length* or by the default limit of 2^{18}-1 (262143) states.

`V_tapValue` Set to the binary representation of the tap sequence used. That is, you can generate the same sequence using `/TAPB=V_tapValue`. If you use /MROR=1, V_tapValue reflects that setting. It will also give the actual tap value used when you specify the a maximal-length sequence using the /MAX flag.

`V_nextValue` Set to the next value beyond the last generated register state. This can be used to restart a truncated sequence.

Examples

Generate a 16-bit maximal-length sequence and reprocess the output values to be centered on zero and normalized to a maximum value of 1:

```
LinearFeedbackShiftRegister/N=16
Redimension/D W_LFSR
```

```
W_LFSR -= 2^15
W_LFSR /= 2^15-1
```

Another way to do the same thing that avoids the Redimension operation, which could lead to fragmentation of memory:

```
Make/D/N=(2^16-1) LFSR_output
LinearFeedbackShiftRegister/N=16/DEST=LFSR_output
LFSR_output -= 2^15
LFSR_output /= 2^15-1
```

Make a bit stream with random +1 and -1 instead of 0 and 1:

```
LinearFeedbackShiftRegister/N=16/MODE=1
Redimension/B W_LFSR
W_LFSR = W_LFSR*2-1
```

See Also

If you really need random numbers, we provide high-quality RNG's that return random deviates from a number of distributions. See **enoise**, **gnoise**, and others.

References

A discussion of LFSR's can be found in "Generation of Random Bits" (Section 7.4) in Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992. They refer to "primitive polynomials modulo 2" and do not use the name Linear Feedback Shift Register, but it is the same thing. We use an implementation equivalent to their Method I.

ListBox

ListBox [**/Z**] ***ctrlName*** [***keyword = value*** [**,** ***keyword = value*** **…**]]

The ListBox operation creates or modifies the named control that displays, in the target window, a list from which the user can select any number of items.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the ListBox control to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **Button** and **DefaultGUIControls** for more appearance details.

clickEventModifiers=*modifierSelector*

Selects modifier keys to ignore when processing clicks to start editing a cell or when toggling a checkbox. That is, use this keyword if you want to prevent a shift-click (for instance) from togging checkbox cells. Allows the action procedure to receive mousedown events with those modifiers without interfering actions on the part of the listbox control.

modifierSelector is a bit pattern with a bit for each modifier key; sum these values to get the desired combination of modifiers:

modifierSelector=1:	Control key (Ctrl)
modifierSelector=2:	Option (Macintosh) or Alt (Windows)
modifierSelector=4:	Context click (right click on Windows, control-click on Macintosh)
modifierSelector=8:	Shift key
modifierSelector=16:	Cmd (Macintosh) key
modifierSelector=32:	Caps lock key

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

col=*c*

Sets the left-most visible column (user scrolling will change this). The list is scrolled horizontally as far as possible. Sometimes this won't be far enough to

actually make column *c* the first column, but it will at least be visible. Use *c* =1 to put the left edge of column 1 (the *second* column) at the left edge of the list.

colorWave= *cw*
: Specifies a 3 column numeric wave containing red, green, and blue values as short unsigned integers. Used in conjunction with planes in selWave to define foreground and background colors for individual cells. Values range from 65535 (full on) to 0.

disable=*d*
: Sets user editability of the control.
  d=0: Normal.
  d=1: Hide.
  d=2: Draw in gray state; disable control action.

editStyle= *e*
: Sets the style for cells designated as editable (see selWave, bit 1).
  e=0: Uses a light blue background (default).
  e=1: Draws a frame around the cell with a white background.
  e=2: Combines the frame with the blue background. The background in all cases can be overridden using the colorWave parameter.

font="*fontName*"
: Sets the font used for the list box items, e.g., `font="Helvetica"`.

frame= *f*
: Specifies the list box frame style.
  f=0: No frame.
  f=1: Simple rectangle.
  f=2: 3D well.
  f=3: 3D raised.
  f=4: text well style.

fsize= s
: Sets list box font size.

fstyle=*fs*
: *fs* is a binary coded number with each bit controlling one aspect of the font style as follows:
  bit 0: Bold.
  bit 1: Italic.
  bit 2: Underline.
  bit 3: Outline (*Macintosh only*).
  bit 4: Shadow (*Macintosh only*).
  See **Setting Bit Parameters** on page IV-12 for details about bit settings.

hScroll=*h*
: Scrolls the list to the right by *h* pixels (user scrolling will change this). *h* is the total amount of horizontal scrolling, not an increment from the current scroll position: *h* will be the value returned in the V_horizScroll variable by **ControlInfo**.
  The hScroll value will not be automatically included in a ListBox recreation macro until Igor Pro 6 or later (to keep experiments compatible with Igor 5.0 and 5.01) unless you execute `SetIgorOption recreateListboxHScroll=1`. This means that normally the scroll position will be lost when saving a ListBox window recreation macro, and hScroll won't be included in the ControlInfo S_recreation string or in the WinRecreation result.

keySelectCol= *col*
: Sets scan column number *col* when doing keyboard selection. Default is to scan column zero.

listWave= *w*
: A 1D or 2D text wave containing the list contents.

mode=*m*
: List selection mode specifying how many list selections can be made at a time.
  m=0: No selection allowed.
  m=1: One or zero selection allowed.
  m=2: One and only one selection allowed.
  m=3: Multiple, but not disjoint, selections allowed.
  m=4: Multiple and disjoint selections allowed.
  When multiple columns are used, you can enable individual cells to be selected using modes 5, 6, 7, and 8 in analogy to *m*=1-4. When using *m*=3 or 4 with multiple

columns, only the first column of the selWave is used to indicate selections. Checkboxes and editing mode, however, use all cells even in modes 0-4.

Modes 9 and 10 are the same as modes 4 and 8 except they use different selection rules and require testing bit 3 as well as bit 0 in selWave. In modes 4 and 8, a shift click toggles individual cells or rows, but in modes 9 and 10, the Command (*Macintosh*) or Ctrl (*Windows*) key toggles individual cells or rows whereas Shift defines a rectangular selection. T o determine if a cell is selected, perform a bitwise AND with `0x09`.

proc= *p* — Set name of user function proc to be called upon certain events. See discussion below.

pos={*left*,*top*} — Sets the location of top left corner of the list box in pixels.

pos+={*dx*,*dy*} — Offsets the position of the list box in pixels.

row=*r* — *r* is desired top row (user scrolling will change this). Use a value of -1 to scroll to the first selected cell (if any). Combine with selRow to select a row and to ensure it is visible (modes 1 and 2).

selCol= *c* — Defines the selected column when mode is 5 or 6 and no selWave is used. To read this value, use **ControlInfo** and the V_selCol variable.

selRow= *s* — Defines the selected row when mode is 1 or 2; when no selWave is used, it is defined by modes 5 or 6. Use -1 for no selection.

To read this value, use **ControlInfo** and the V_value variable.

selWave= *sw* — *sw* is a numeric wave with the same dimensions as listWave. It is optional for modes 0-2, 5 and 6 and required in all other modes.

In modes greater than 2, *sw* indicates which cells are selected. In modes 1 and 2 use **ControlInfo** to find out which row is selected.

In all modes *sw* defines which cells are editable or function as checkboxes or disclosure controls.

Numeric values are treated as integers with individual bits defined as follows:

Bit 0 (0x01): Cell is selected.

Bit 1 (0x02): Cell is editable.

Bit 2 (0x04): Cell editing requires a double click.

Bit 3 (0x08): Current shift selection.

Bit 4 (0x10): Current state of a checkbox cell.

Bit 5 (0x20): Cell is a checkbox.

Bit 6 (0x40): Cell is a disclosure cell. Drawn as a disclosure triangle (*Macintosh*) or a treeview expansion node (*Windows*).

In modes 3 and 4 bit 0 is set only in column zero of a multicolumn listbox.

Other bits are reserved. Additional dimensions are used for color info. See the discussion for colorWave. As of Igor Pro 6, selWave is not required for modes 5 and 6.

`setEditCell={`*`row`*`,`*`col`*`,`*`selStart`*`,`*`selEnd`*`}`

Initiates edit mode for the cell at *row*, *col*. An error is reported if *row* or *col* is less than zero. Nothing happens and no error is reported if *row*, *col* is beyond the limits of the listbox, or if the cell has not been made editable by setting bit 1 of *selWave*.

selStart and *selEnd* set the range of characters that are selected when editing is initiated; 0 is the start of the text. If there are N characters in the listbox cell, setting *selStart* or *selEnd* to N or greater moves the start or end of the selection to the point after the last character. Setting *selStart* and *selEnd* to the same value selects no characters and the insertion point is set to *selStart*. Setting *selStart* to -1 always causes all characters to be selected.

size={*width*,*height*} — Sets list box size in pixels.

special={*kind*,*height*,*style*}

Specifies special cell formatting or contents.

kind=0: Normal text but with specified *height* (if nonzero). Use a *style* of 1 to autocalculate widths based on the entire list contents. In this case, user widths are taken to be minimums and the last is not repeated.

kind=1: Text taken to be the names of graphs or tables. Images of the graphs or tables are displayed in the cells. Use a *style* of 0 to display just the presentation portion of the graph or 1 to display it entirely. For tables, only the presentation portion is displayed.

kind=2: Text taken to be the names of pictures. Images are displayed in the cells.

kind=3: Displays a PNG, TIFF, or JPEG image. You can obtain binary picture data using SavePict.

For *kind*=1 or 2, *height* may be zero to auto-set cell height to same as width or a specific value.

titleWave=*w* Specifies a text wave containing titles for the listbox columns, instead of using the list wave dimension labels. Each row is the title for one column; if you have N columns you must have a wave with N rows. Allows more than 31 characters for a title, which is particularly important if you use styled text.

userColumnResize= *u* Enables resizing the list columns using the mouse.

u =0: Columns are not resizable (default). The widths parameter still works, though.

u =1: User can resize columns by dragging the column dividers.

When resizing a column without Option, Alt, or Shift modifiers (a "normal" resizing), any width added to the column is subtracted from the following column (if any).

When resizing while pressing Option (*Macintosh*) or Alt (*Windows*), only columns following the dragged divider will move (the same way table columns are resized).

When pressing Shift, all columns are set to the same width as the column being resized. If the total widths of all columns is less than the width of the listbox, then each column expands to fill the available width.

userdata=*UDStr* Sets the unnamed user data to *UDStr*.

userdata(*UDName*)=*UDStr*
Sets the named user data, *UDName*, to *UDStr*.

userdata+=*UDStr* Appends *UDStr* to the current unnamed user data.

userdata(*UDName*)+=*UDStr*
Appends *UDStr* to the current named user data, *UDName*.

widths={*w1*,*w2*,…} Optional list of minimum column widths in screen pixels. If more columns than widths, the last is repeated. If total of widths is greater than list box width then a horizontal scroll bar will appear. If total is less than available width then each expands proportionally.

widths+={*w1*,*w2*,…} Additional column widths. Because only 400 characters fit on a command line, lists with many columns may require multiple widths+= parameters to define all the column widths. However if all the widths are the same, widths+= is not needed; just use:

```
ListBox ctrlName widths={sameWidth}
```

win=*winName* Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

If the list wave has column dimension labels (see **SetDimLabel**), then those will be used as column titles. Note that a 1D wave is subtly different from a 1 column 2D wave. The former does not have any columns and therefore no column dimension labels.

Alternately, use a text wave with the titleWave keyword to specify column titles.

You can make a column title bold using TextBox-style escape codes. For instance, to make a title bold use the \f01 escape sequence:

```
SetDimLabel 1, columnNum, $"\\f01My Label", textWave
```

If you can't fit title text within the 31 character limit (styled text can be especially long), use the titleWave keyword with a text wave. The wave must have as many rows as the list wave has columns. When using a title wave, there are no restrictions on the number of or what characters you can use.

This example uses a title wave to add a red up-arrow graph marker to the end of a centered title:

```
Make/O/T/N=(numColumns) columnTitles
columnTitles[colNum]="\\JCThis is the title\\K(65535,0,0)\\k(65535,0,0)\\W523"
ListBox list0 titleWave=columnTitles
```

That's a 51-character title that results in 19 characters or symbols that you actually see. \JC requests centered text, \K sets the text color (which colors the inside of the graph marker), \k sets the marker stroke color, and \W523 inserts a down-pointing triangular graph marker.

When using modes that allow multiple selections, use Shift to extend or add to the selection.

You can specify individual cells as being editable by setting bit 1 (counting from zero on the right) in selWave. The user can start editing a cell by either clicking in it or, if the cell is selected, by pressing Enter (or Return). When finished, the user can press Enter to accept the changes or can press Escape to reject changes. The user may also press Up or Down Arrow to accept changes and begin editing the next editable cell in a column. Likewise, Tab and Shift-Tab moves to the next or previous column in a row. If bit 2 of selWave is set then a double click will be required rather than a single click. **Note**: in edit mode, Tab and Shift-Tab are used to move left and right because the Left and Right Arrow keys are used to move the text entry cursor left and right.

When the listbox has keyboard focus (either by tabbing to the list box or by clicking in the box), the keyboard arrow keys move a cell selection (or row depending on mode). When not in cell edit mode, Tab and Shift-Tab move the keyboard focus to other objects in the window. The Home, End, Page Up, and Page Down keys affect the vertical scroll bar.

When the listbox has focus, the user may type the first few chars of an entry in the list to select that entry. Only the first column is used. If a match is not found then nothing is done. The search is case insensitive.

You may define a user-function that will be called when certain events occur. Your action function must have the following syntax:

```
Function MyListboxProc(ctrlName,row,col,event) : ListboxControl
   String ctrlName     // name of this control
   Variable row        // row if click in interior, -1 if click in title
   Variable col        // column number
   Variable event      // event code
   …
   return 0            // other return values reserved
End
```

The "`: ListboxControl`" designation tells Igor to include this procedure in the Procedure pop-up menu in the List Box Control dialog.

The action procedure for a ListBox control can also use a predefined structure `WMListboxAction` as a parameter to the function. The control will use this more efficient method when the function properly matches the structure prototype for a ListBox control, otherwise it will use the old-style method.

A ListBox action procedure using a structure has the format:

```
Function newActionProcName(LB_Struct) : ListboxControl
   STRUCT WMListboxAction &LB_Struct
   …
End
```

For a ListBox control, the `WMListboxAction` structure has members as described in the following table:

`WMListboxAction` Structure Members

Member	Description
`char ctrlName[MAX_OBJ_NAME+1]`	Control name.
`char win[MAX_WIN_PATH+1]`	Host (sub)window.
`STRUCT Rect winRect`	Local coordinates of host window.

WMListboxAction Structure Members

Member	Description
STRUCT Rect ctrlRect	Enclosing rectangle of the control.
STRUCT Point mouseLoc	Mouse location.
Int32 eventCode	Event that caused the procedure to execute. See table following for eventCode values.
String userData	Primary (unnamed) user data. If this changes, it is written back automatically.
Int32 blockReentry	Prevents reentry of control action procedure. See **Control Structure blockReentry Field** on page III-386.
Int32 eventCode2	Used for the second event when two events happen at the same time. In this case, the various values in the structure refer to this second event. As of Igor 5.02, an event queue is used to deliver events to new-style action procedures and eventCode2 is not used. Instead your procedure simply gets called multiple times. See **Event Queue** on page V-343 for more details
Int32 eventMod	Bitfield of modifiers. See **Control Structure eventMod Field** on page III-385.
Int32 row, col	Row number of selection in interior or -1 if in title area, and column number of selection. The meanings of row and col are different for eventCodes 8-11:

Code	row	col
8	top visible row	horiz shift in pixels.
9	top visible row	horiz shift (user scroll).
9	-1	horiz shift (hScroll keyword).
10	top visible row	-1 (row keyword).
10	-1	first visible col (col keyword).
11	column shift	column resized by user.

	If eventCode is 11, row is the horizontal shift in pixels of the column col that was resized, not the total horizontal shift of the list as reported in V_horizScroll by ControlInfo. If row is negative, the divider was moved to the left. col=0 corresponds to adjusting the divider on the right side of the first column. Use **ControlInfo** to get a list of all column widths.
WAVE/T listWave	List wave specified by ListBox command.
WAVE selWave	Selection wave specified by ListBox command.
WAVE colorWave	Color wave specified by ListBox command.
WAVE/T titleWave	Title wave specified by ListBox command

The event code passed to your action procedure has the following meanings:

Action functions should respond only to documented `eventCode` values. Other event codes may be added along with more fields. Although the return value is not currently used, action functions should always return zero.

The constants used to specify the size of structure `char` arrays are internal to Igor Pro and may change.

eventCode	Meaning
-1	Control being killed.
1	Mouse down.
2	Mouse up.
3	Double click.
4	Cell selection (mouse or arrow keys).
5	Cell selection plus Shift key.
6	Begin edit.
7	Finish edit.
8	Vertical scroll. See **Scroll Event Warnings** on page V-343.
9	Horizontal scroll by user or by the hScroll=*h* keyword.
10	Top row set by row=*r* or first column set by col=*c* keywords.
11	Column divider resized.
12	Keystroke, character code is place in row field.
13	Checkbox was clicked. This event is sent after selWave has been updated by Igor Pro 6.20 or later.

Some events cause another event immediately. For example a cell selection usually follows a mouse down. Before Igor Pro 5.02, the second event is stored in eventCode 2. In Igor Pro 5.02 and later, each event causes a separate call to your event handler. See **Event Queue** for more details.

The background and foreground (text) color of individual cells may be defined by providing colorWave in conjunction with specific planes in selWave. The planes in selWave are taken to be integer indexes into colorWave. The planes are defined by specific dimension labels and not by specific plane numbers. To provide foreground colors, define a plane labeled "foreColors" that contains the desired index values. Likewise define and fill a plane labeled "backColors" for background colors. The value 0 is special and indicates that the default colors should be used. Note that if you have a one column list for which you want to supply colors, the selWave needs to be three dimensional but with just one column. Here is an example:

```
Make/T/N=(5,1) tw= "row "+num2str(p)        // 5 row, 1 col text wave (2D)
Make/B/U/N=(5,1,2) sw                       // 5 row, 1 col, 2 plane byte wave
Make/O/W/U myColors={{0,0,0},{65535,0,0},{0,65535,0},{0,0,65535},{0,65535,65535}}
MatrixTranspose myColors      // above was easier to enter as 3 rows, 5 cols

NewPanel
ListBox lb,mode=3,listWave= tw,selWave= sw,size={200,100},colorWave=myColors
sw[][][1]= p                          // arbitrary index values into plane 1
```

Now, execute the following commands one at a time and observe the results:

```
SetDimLabel 2,1,backColors,sw        // define plane 1 as background colors

SetDimLabel 2,1,foreColors,sw        // redefine plane 1 s foreground colors

sw[][][%foreColors]= 4-p             // change the color index values
```

In the above example, the selWave was defined as unsigned byte. If you need more than 254 colors, you will have to use a larger number size.

Checkboxes in Cells

You can cause a cell to contain a checkbox by setting bit 5 in selWave. The title (if any) is taken from listWave and the results (selected/deselected) is bit 4 of selWave. If a checkbox cell is selected then the space bar will toggle the checkbox. (Clicking a checkbox cell does not select it — use the arrow keys.)

Errors

Your listbox may be drawn with a red X and an error code. The error codes are:

Error	Meaning
E1	Too small.
E2	listWave is invalid (missing, not text or no rows).
E3	listWave and selWave do not match in dimensions.
E4	mode > 2 with no selWave.

Event Queue

It is possible for a single user action to produce more than one event. For instance, using the Up Arrow key while editing to select a cell that is not visible will generate events 4, 8, and 6. As of Igor 5.02, such a cascade of events will generate three separate calls to your new-style action procedure that uses a structure for input. Before Igor 5.02 eventCode2 was used to deliver a second event to your action procedure (scroll events 8, 9, and 10 were not available then).

If your code tests for a nonzero value in eventCode2, and uses it only if it is nonzero, then the event queue method will not break your code. If you have determined empirically when you need to use eventCode2 and you do not test, your code will break.

Scroll Event Warnings

Events 8, 9, and 10 report to you that the listbox has been scrolled vertically or horizontally. These events are envisioned as allowing you to keep two listboxes synchronized (you may find other uses for these events). You might use an action procedure like this one to keep two listboxes (named list0 and list1) in sync:

```
Function ListBoxProc2(LB_Struct) : ListBoxControl
   STRUCT WMListboxAction &LB_Struct

   if (LB_Struct.eventCode == 8)
      String listname
      if (CmpStr(LB_Struct.ctrlName, "list1") == 0)
         listname = "list0"
      else
         listname = "list1"
      endif
      ControlInfo $listname
      if (V_startRow != LB_Struct.row)
         listbox $listname,row=LB_Struct.row
         ControlUpdate $listname
      endif
   endif
End
```

It is very easy to create an infinite cascade of events feeding back between the two listboxes, especially if you use event 10. When this happens, you will see your listboxes jigging up and down endlessly. The test using ControlInfo is intended to make this unlikely.

The slow response of the old-style, nonstructure action procedure can defeat the ControlInfo test by delaying the action procedure execution. If you use events 8, 9, or 10, we recommended that you use the new-style action procedure.

Examples

Here is a simple Listbox example:

```
Make/O/T/N=30 tjack="this is row "+num2str(p)
Make/O/B/N=30 sjack=0
NewPanel /W=(19,61,319,261)
ListBox lb1,pos={42,9},size={137,94},listWave=tjack,selWave=sjack,mode= 3
Edit/W=(367,61,724,306) tjack,sjack
ModifyTable width(tjack)=148
```

Make selections in the list and note changes in the table and vice versa. Edit one of the list text values in the table and note update of the list.

Here is an example using a titleWave and styled text in the title cells. Note that the last title isn't very long when rendered, but requires a 63 character specification.

```
Make/O/T/N=(4,3) ListWave="row "+num2str(p)+" col "+num2str(q)
Make/O/T/N=3 titles                    // three rows to match 3-column ListWave
titles[0] = "\f01Bold Title"
titles[1] = "title with semicolon;"
titles[2] = "Marker in Gray: \K(40000,40000,40000)\k(40000,40000,40000)\W517"
```

```
NewPanel /W=(515,542,1011,794)
ListBox list0,pos={1,2},size={391,120},listWave=ListWave
ListBox list0,titleWave=titles
```

An example experiment that lets you easily experiment with ListBox settings is available in "Examples:Testing & Misc:ListBox Demo.pxp".

See Also

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Setting Bit Parameters on page IV-12 for further details about bit settings.

The **GetUserData** operation for retrieving named user data.

ListMatch

ListMatch(*listStr*, *matchStr* [, *listSepStr*])

The ListMatch function returns each list item in *listStr* that matches *matchStr*.

ListStr should contain items separated by the *listSepStr* character, such as "abc;def;".

You may include asterisks in *matchStr* as a wildcard character. Note that matching is case-insensitive. See **stringmatch** for wildcard details.

ListSepStr is optional. If missing, it is taken to be ";".

See Also

The **GrepList**, **stringmatch**, **StringFromList**, and **WhichListItem** functions.

ln

ln(*num*)

The ln function returns the natural logarithm of *num*, -INF if *num* is 0, or NaN if *num* is less than 0. In complex expressions, *num* is complex, and ln(*num*) returns a complex value.

To compute a logarithm base n use the formula:

$$\log_n(x) = \frac{\log(x)}{\log(n)}.$$

See Also

The **log** function.

LoadData

LoadData [*flags*] *fileOrFolderNameStr*

The LoadData operation loads data from the named file or folder. "Data" means Igor waves, numeric and string variables and data folders containing them. The specified file or folder must be an Igor packed experiment file or a folder containing Igor binary data, such as an Igor unpacked experiment folder or a folder in which you have stored Igor binary wave files.

LoadData loads data objects into memory and they become part of the current Igor experiment, disassociated from the file from which they were loaded.

If loading from a file-system folder, the data (waves, variables, strings) in the folder, including any subfolders if /R is specified, is loaded into the current Igor data folder.

If loading from a packed Igor experiment file, the data in the file, including any packed subdata folders if /R is specified, is loaded into the current Igor data folder.

Use LoadData to load experiment data using Igor procedures. To load experiment data interactively, use the Data Browser (Data menu).

Parameters

If you use a full or partial path for *fileOrFolderNameStr*, see **Path Separators** on page III-398 for details on forming the path.

If *fileOrFolderNameStr* is omitted you get to locate the file (if /D is omitted) or the folder (if /D is present) via a dialog.

Flags

/D — If present, loads from a file-system folder (a directory). If omitted, LoadData loads from an Igor packed experiment file.

/I — Interactive. Forces LoadData to present a dialog.

/J=*objectNamesStr* — Loads only the objects named in the semicolon-separated list of object names.

/L=*loadFlags* — Controls what kind of data objects are loaded with a bit for each data type:

loadFlags	Bit Number	Loads this Object Type
1	0	Waves
2	1	Numeric Variables
4	2	String Variables

To load multiple data types, sum the values shown in the table. For example, /L=1 loads waves only, /L=2 loads numeric variables only, and /L=3 loads both waves and numeric variables. See **Setting Bit Parameters** on page IV-12 for further details about bit settings.

If no /L is specified, all object types are loaded. This is equivalent to /L=7. All other bits are reserved and should be set to zero.

/O[=*overwriteMode*] — If /O alone is used, overwrites existing data objects in case of a name conflict.

overwriteMode is defined as follows:

0: No overwrite, as if there were no /O.

1: Normal overwrite. In the event of a name conflict, objects in the incoming file replace the conflicting objects in memory. Incoming data folders completely replace any conflicting data folders in memory.

2: Mix-in overwrite. In the event of a name conflict, objects in the incoming file replace the conflicting objects in memory but nonconflicting objects in memory are left untouched.

See **Details** for more about overwriting.

/P=*pathName* — Specifies folder to look in for the specified file or folder. *pathName* is the name of an existing symbolic path.

/Q — Suppresses the normal messages in the history area.

/R — Recursively loads subdata folders.

/S=*subDataFolderStr* — Specifies a subdata folder within a packed experiment file to be loaded. See **Details** for more.

/T[=*topLevelName*] — If /T=*topLevelName* is specified, it creates a new data folder in the current data folder with the specified name and places the loaded data in the new data folder.

If just /T is specified, it creates a new data folder in the current data folder with a name derived from the name of the unpacked experiment folder, packed experiment file or packed subdata folder being loaded.

Details

If /T is present, LoadData loads the top level data folder and its contents. If /T is omitted, it loads just the contents of the top level data folder and not the data folder itself. This distinction has an analogy in the desktop. You can drag the contents of disk folder A into folder B or you can drag folder A itself into folder B.

If present, /S=*subDataFolderStr* specifies the subdata folder within the packed experiment file from which the load is to start. For example:

```
LoadData/P=Path1/S="Folder A:Folder B" "aPackedExpFile"
```

This starts loading from data folder “Folder B” which is in “Folder A” in the packed experiment file. Note that the string specified by /S must specify each subdata folder until the desired data folder is reached. Since this parameter is specified as a string, you must not use single quotes.

/S has no effect if you are loading from a file system folder rather than from a packed experiment file.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-34 for details.

If /J=*objectNamesStr* is used, then only the objects named in *objectNamesStr* are loaded into the current experiment. For example, `/J="wave0;wave1"` will load only the two named waves, ignoring any other data in the file or folder being loaded.

Assume that you have an experiment that contains 5 runs where each run, stored in a separate data folder in a packed experiment file, consists of data acquired from four channels from an ADC card. Using the /J flag, you can load just one specific channel from each run. This way you can compare that channel's data from all runs without loading the other channels.

The list of object names used with /J must be semicolon-separated. A semicolon after the last object name in the list is optional. Because the object names exist in a string expression, they should not be quoted. The list is limited to 1000 characters.

Using `/J=""` acts like no /J at all.

If you load a hierarchy of data folders (using the /R flag) with /J in effect, LoadData will create each data folder in the hierarchy even if it contains none of the named objects. This behavior is necessary to avoid loading a subdata folder without loading its parent, as well as other such complications.

If you do a load of a data folder, overwriting an existing data folder of the same name, the behavior of LoadData depends on whether you use /J. If you do not use /J, the entire data folder and all of its contents are replaced. If you do use /J, just the specified objects in the data folder are replaced, leaving any other preexisting objects in the data folder unchanged.

If you do not use the /O (overwrite) flag or if you use /O=0 and there is a conflict between objects or data folders in the current data folder and objects or data folders in the file or folder being loaded, LoadData will present a dialog to ask you if you want to replace the existing data. However, LoadData can not replace an object with an object of a different type and will refuse to do so.

You can overwrite an object that is in use, such as a wave that is displayed in a graph or table. You can also overwrite a data folder that contains objects that are in use. This is a powerful feature. Imagine that you define a data structure consisting of waves, variables and possibly subdata folders. You can display the data in graphs and tables and you can display these in a layout. You can then overwrite the data with an analogous data structure from a packed experiment file and Igor will automatically update any graphs, tables, or layouts that need to be updated.

Because LoadData can load from a complex packed Igor experiment file or from a complex hierarchy of file-system folders, it does not set the variables normally set by a file loader: S_path, S_fileName, and S_waveNames. The variable V_flag is set to the total number of objects loaded, or to -1 if the user cancelled the open file dialog. To find what objects were created by LoadData, you can use the **CountObjects** and **GetIndexedObjName** functions.

See Also

The **SaveData** operation. Chapter II-9, **Importing and Exporting Data**; **Data Browser** on page II-130.

LoadPackagePreferences

LoadPackagePreferences [**/MIS=***mismatch* **/P=***pathName*] ***packageName, prefsFileName, recordID, prefsStruct***

The LoadPackagePreferences operation loads preference data previously stored on disk by the **SavePackagePreferences** operation. The data is loaded into the specified structure.

If the /P flag is present then the location on disk of the preference file is determined by *pathName* and *prefsFileName*. However in the usual case the /P flag will be omitted and the preference file is located in a file named *prefsFileName* in a directory named *packageName* in the Packages directory in Igor's preferences directory.

Note: You must choose a very distinctive name for *packageName* as this is the only thing preventing collisions between your package and someone else's package.

See **Saving Package Preferences** on page IV-226 for background information and examples.

Parameters

packageName is the name of your package of Igor procedures. It is limited to 31 characters and must be a legal name for a directory on disk. This name must be very distinctive as this is the only thing preventing collisions between your package and someone else's package.

prefsFileName is the name of a preference file to be loaded by LoadPackagePreferences. It should include an extension, typically ".bin".

prefsStruct is the structure into which data from disk, if it exists, will be loaded.

recordID is a unique positive integer that you assign to each record that you store in the preferences file. If you store more than one structure in the file, you would use distinct *recordID*s to identify which structure you want to load. In the simple case you will store just one structure in the preference file and you can use 0 (or any positive integer of your choice) as the *recordID*.

Flags

/MIS=*mismatch* Controls what happens if the number of bytes in the file does not match the size of the structure:

0: Returns an error. Default behavior if /MIS is omitted.

1: Returns the smaller of the size of the structure and the number of bytes in the file. Does not return an error. Use this if you want to read and update old versions of a preferences structure.

/P=*pathName* Specifies the directory to look in for the file specified by *prefsFileName*.

pathName is the name of an existing symbolic path. See **Symbolic Paths** on page II-34 for details.

`/P=$<empty string variable>` acts as if the /P flag were omitted.

Details

LoadPackagePreferences sets the following output variables:

`V_flag` Set to 0 if no error occurred or to a nonzero error code.

If the preference file does not exist, V_flag is set to zero so you must use V_bytesRead to detect that case.

`V_bytesRead` Set to the number of bytes read from the file. This will be zero if the preference file does not exist.

`V_structSize` Set to the size in bytes of *prefsStruct*. This may be useful in handling structure version changes.

After calling LoadPackagePreferences if V_flag is nonzero or V_bytesRead is zero then you need to create default preferences as illustrated by the example referenced below.

V_bytesRead, in conjunction with the /MIS flag, makes it possible to check for and deal with old versions of a preferences structure as it loads the version field (typically the first field) of an older or newer version structure. However in most cases it is sufficient to omit the /MIS flag and treat incompatible preference data the same as missing preference data.

Example

See the example under **Saving Package Preferences in a Special-Format Binary File** on page IV-226.

See Also

SavePackagePreferences.

LoadPICT

LoadPICT [***flags***] [***fileNameStr***][**, *pictName***]

The LoadPICT operation loads a picture from a file or from the Clipboard into Igor. Once you have loaded a picture, you can append it to graphs and page layouts.

Parameters

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

If you want to force a dialog to select the file, omit the *fileNameStr* parameter.

If *fileNameStr* is "Clipboard" and /P=*pathName* is omitted, LoadPICT loads its data from the Clipboard rather than from a file.

pictName is the name that you want to give to the newly loaded picture. You can refer to the picture by its name to append it to graphs and page layouts. LoadPICT generates an error if the name conflicts with some other type of object (e.g., wave or variable) or if the name conflicts with a built-in name (e.g., the name of an operation or function).

If you omit *pictName*, LoadPICT automatically names the picture as explained in **Details**.

Flags

/I=*resIndex*	Specifies the resource to load by resource index, starting from 1 (*Macintosh only*).
/M=*promptStr*	Specifies a prompt to use if LoadPICT needs to put up a dialog to find the file.
/N=*resNameStr*	A string that specifies the resource to load by resource name (*Macintosh only*).
/O	Overwrites an existing picture with the same name. If /O is omitted and there is an existing picture with the same name, LoadPICT displays a dialog in which you can resolve the name conflict.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/Q	Quiet: suppresses the insertion of picture info into the history area.
/R=*resourceID*	Specifies the resource to load by resource ID (*Macintosh only*).
/Z	Doesn't load the picture, just checks for its existence.

Details

If the picture file is not fully specified then LoadPICT presents a dialog from which you can select the file. "Fully specified" means that LoadPICT can determine the name of the file (from the *fileNameStr* parameter) and the folder containing the file (from the flag /P=*pathName* flag or from the *fileNameStr* parameter). If you want to force a dialog, omit the *fileNameStr* parameter.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-34 for details.

On Macintosh, LoadPICT can load picture data from the file's data or resource fork. Most graphics programs store a picture in the data fork. Some programs may store one or more pictures as resources. If you use /R, /I, or /N, it loads the picture from the specified resource in the file. Otherwise, it loads the picture from the file's data fork.

If you omit *pictName*, LoadPICT automatically names the picture as follows:

On Macintosh, if the picture was loaded from the resource fork of a file (you used /R, /I, or /N) and the resource had a nonempty name, it uses the resource name. If necessary, the name is made legal by replacing illegal characters or shortening it.

If the picture was loaded from a file, LoadPICT uses the file name. If necessary, it makes it into a legal name by replacing illegal characters or shortening it.

Otherwise, LoadPICT uses a name of the form "PICT_*n*".

If the resulting name is in conflict with an existing picture name, Igor puts up a name conflict resolution dialog.

LoadPICT sets the variable V_flag to 1 if the picture exists and fits in available memory or to 0 otherwise.

It also sets the string variable S_info to a semicolon-separated list of values:

Keyword	Information Following Keyword
NAME	Name of the loaded PICT, often "PICT_0", etc.
SOURCE	One of "data fork", "resource fork" or "Clipboard".
RESOURCENAME	Name of the resource the picture was loaded from, or `""` if the source was not the file's resource fork.
RESOURCEID	Resource ID the picture was loaded from, or 0 if the source was not the file's resource fork.
TYPE	One of: "Enhanced metafile", "Windows metafile", "DIB", "Windows bitmap", "PICT", "PNG", or "Unknown type".

Keyword	Information Following Keyword
BYTES	Amount of memory used by the picture.
WIDTH	Width of the picture in pixels.
HEIGHT	Height of the picture in pixels.
PHYSWIDTH	Physical width of the picture in points.
PHYSHEIGHT	Physical height of the picture in points.

See Also

See **Pictures** on page III-421 for general information on how Igor handles pictures.

The **ImageLoad** operation for loading PICT and other image file types into waves, and the **PICTInfo** function.

LoadWave

LoadWave [***flags***] [***fileNameStr***]

The LoadWave operation loads data from the named Igor binary, Igor text, delimited text, fixed field text, or general text file into waves. LoadWave can load 1D and 2D data from delimited text, fixed field text and general text files, or data of any dimensionality from Igor binary and Igor text files.

Parameters

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If LoadWave can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

If *fileNameStr* is "Clipboard" and /P=*pathName* is omitted, LoadWave takes its data from the Clipboard rather than from a file. This is not implemented for binary loads.

If *fileNameStr* is omitted you get to locate the file via a dialog.

Flags

/A "Auto-name and go" option (used with /G, /F or /J).

This skips the dialog in which you normally enter wave names. Instead it automatically assigns names of the form *wave*0, *wave*1, choosing names that are not already in use. When used with /W, it reads wave names from the file instead of automatically assigning names and /A just skips the wave name dialog. The /B flag can also override names specified by /A.

/A=*baseName* Same as /A but it automatically assigns wave names of the form *baseName*0, *baseName*1.

/B=*columnInfoStr* Specifies the name, format, numeric type, and field width for columns in the file. See **Specifying Characteristics of Individual Columns**.

/C This is used in experiment recreation commands generated by Igor to force experiment recreation to continue if an error occurs in loading a wave.

/D Creates double precision waves. (Used with /G, /F or /J.) The /B flag can override the numeric precision specified by the /D flag.

/E=*editCmd*

editCmd= 1: Makes a new table containing the loaded waves.

editCmd= 2: Appends the loaded waves to the top table. If no table exists, a new table is created.

editCmd= 0: Same as if /E had not been specified (loaded waves are not put in any table).

/F={*numColumns*, *defaultFieldWidth*, *flags*}

Indicates that the file uses the fixed field file format. Most FORTRAN programs generate files in this format.

numColumns is the number of columns of data in the file.

defaultFieldWidth is the default number of characters in each column. If the columns do not all have the same number of characters, you need to use the /B flag to provide more information to LoadWave.

flags is a bitwise parameter that controls the conversion of text to values. The bits are defined as follows:

bit 0: If set, any field that consists entirely of the digit "9" is taken to be blank. A field that consists entirely of the digit "9" except for a leading "+" or "-" is also taken to be blank.

All other bits are reserved and you should use 0 for them.

/G Indicates that the file uses the general text format.

/H Loads the wave into the current experiment and severs the connection between the wave and the file. When the experiment is saved, the wave copy will be saved as part of the experiment. For a packed experiment this means it is saved in the packed experiment file. For an unpacked experiment this means it is saved in the experiment's home folder.

See **Sharing Versus Copying Igor Binary Files** on page II-165.

/J Indicates that the file uses the delimited text format.

/K=*k* Controls how to determine whether a column in the file is numeric or text (only for delimited text and fixed field text files).

k=0: Deduces the nature of the column automatically.

k=1: Treats all columns as numeric.

k=2: Treats all columns as text.

This flag as well as the ability to load text data into text waves were added in Igor Pro 3.0. The default for the LoadWave operation is /K=1, meaning that it will treat all columns as numeric. We did this so that existing procedures would behave the same in Igor Pro 3.0 as before. Use /K=0 when you want to load text columns into text waves. /K=2 may have use in a text-processing application.

For finer control, the /B flag specifies the format of each column in the file individually.

/L={*nameLine*, *firstLine*, *numLines*, *firstColumn*, *numColumns*}

Affects loading delimited text, fixed field text, and general text files only (/J, /F or /G). /L is accepted no matter what the load type but is ignored for Igor binary and Igor text loads. Line and column numbers start from 0.

nameLine is the number of the line containing column names. For general text loads, 0 means auto. See **Loading General Text Files** on page II-153 for details.

firstLine is the number of the first line to load into a wave. For general text loads, 0 means auto. See **Loading General Text Files** on page II-153 for details.

numLines is the number of lines that should be treated as data. 0 means auto which loads until the end of the file or until the end of the block of data in general text files. The *numLines* parameter can also be used to make loading very large files more efficient. See **Loading Very Large Files**.

firstColumn is the number of the first column to load into a wave. This is useful for skipping columns.

numColumns is the number of columns to load into a wave. 0 means auto, which loads all columns.

/M Loads data as matrix wave. If /M is used then it ignores the /W flag (read wave names) and follows the /U flags instead.

The wave is autonamed unless you provide a specific wave name using the /B flag.

The type of the wave (numeric or text) is determined by an assessment of the type of the first loaded column unless you override this using the /K flag or the /B flag.

See **The Load Waves Dialog for Delimited Text — 2D** on page II-149 for further information.

/N Same as /A except that, instead of choosing names that are not in use, it overwrites existing waves.

/N=*baseName* Same as /N except that it automatically assigns wave names of the form *baseName0*, *baseName1*.

/O Overwrite existing waves in case of a name conflict.

/P=*pathName* Specifies the folder to look in for *fileNameStr*. *pathName* is the name of an existing symbolic path.

/Q Suppresses the normal messages in the history area.

/R={*languageName*, *yearFormat*, *monthFormat*, *dayOfMonthFormat*, *dayOfWeekFormat*, *layoutStr*, *pivotYear*}
Specifies a custom date format for dates in the file. If the /R flag is used, it overrides the date setting that is part of the /V flag.

languageName controls the language used for the alphabetic date components, namely the month and the day-of-week. *languageName* is one of the following:

Default			
Chinese	ChineseSimplified	Danish	Dutch
English	Finnish	French	German
Italian	Japanese	Korean	Norwegian
Portuguese	Russian	Spanish	Swedish

Default means the system language on Macintosh or the user default language on Windows.

yearFormat is one of the following codes:

1: Two digit year.

2: Four digit year.

monthFormat is one of the following codes:

1: Numeric, no leading zero.

2: Numeric with leading zero.

3: Abbreviated alphabetic (e.g., Jan).

4: Full alphabetic (e.g., January).

dayOfMonthFormat is one of the following codes:

1: Numeric, no leading zero.

2: Numeric with leading zero.

dayOfWeekFormat is one of the following codes:

1: Abbreviated alphabetic (e.g., Mon).

2: Full alphabetic (e.g., Monday).

layoutStr describes which components appear in the date in what order and what separators are used. *layoutStr* is constructed as follows (but with no line break):

```
"<component keyword><separator>
<component keyword><separator>
<component keyword><separator>
<component keyword>"
```

where <component keyword> is one of the following:

```
Year
Month
DayOfMonth
DayOfWeek
```

and <separator> is a string of zero to 15 characters.

Starting from the end, parts of the layout string must be omitted if they are not used.

Extraneous spaces are not allowed in *layoutStr*. Each separator must be no longer than 15 characters. No component can be used more than once. Some components may be used zero times.

pivotYear determines how LoadWave interprets two-digit years. If the year is specified using two digits, yy, and is less than *pivotYear* then the date is assumed to be 20yy. If the two digit year is greater than or equal to *pivotYear* then the year is assumed to be 19yy. *pivotYear* must be between 4 and 40.

See **Loading Custom Date Formats** for further discussion of date formats.

/T Indicates that the file uses the Igor text format.

Although LoadWave is generally thread-safe, it is not thread-safe to load an Igor text file containing an Igor command (e.g., "X <command>").

/U={*readRowLabels*, *rowPositionAction*, *readColLabels*, *colPositionAction*}

These parameters affect loading a matrix (/M) from a delimited text (/J) or a fixed field text (/F) file. They are accepted no matter what the load type is but are ignored when they don't apply.

If *readRowLabels* is nonzero, it reads the first column of data in the file as the row labels for the matrix wave.

rowPositionAction has one of the following values:

0: The file has no row position column.

1: Uses the row position column to set the row scaling of the matrix wave.

2: Creates a 1D wave containing the values in the row position column. The name of the 1D wave will be the same as the matrix wave but with the suffix "_RP".

The *readColumnLabels* and *columnPositionAction* parameters have analogous meanings. The suffix used for the column position wave is "_CP".

See Chapter II-9, **Importing and Exporting Data** for further details.

/V={*delimsStr*, *skipCharsStr*, *numConversionFlags*, *loadFlags*}

These parameters affect loading delimited text (/J) and fixed field text (/F) data and column names. They are accepted no matter what the load type is but are ignored when they don't apply. These parameters should rarely be needed.

delimsStr is a string expression containing the characters that should act as delimiters for delimited file loads. The default is "\t," for tab and comma. You can specify the space character as a delimiter, but it is always given the lowest priority behind any other delimiters contained in *delimsStr*. The low priority means that if a line of text contains any other delimiter besides the space character, then that delimiter is used rather than the space character.

skipCharsStr is a string expression containing characters that should always be treated as garbage and skipped when they appear before a number. The default is "$" for space and dollar sign. This parameter should rarely be needed.

numConversionFlags is a bitwise parameter that controls the conversion of text to numbers. The bits are defined as follows:

bit 0: If set: dates are dd/mm/yy.
If cleared: dates are mm/dd/yy.

bit 1: If set: decimal character is comma.
If cleared: it is period.

bit 2: If set: thousands separators in numbers are ignored when loading delimited text only (LoadWave/J).

The thousands separator is the comma in 1,234 or, if comma is the decimal character, the dot in 1.234.

Most numeric data files do not use thousands separators and searching for them slows loading down so this bit should usually be 0.

This bit has no effect if the thousands separator (e.g., comma) is also a delimiter character as specified by delimsStr.

All other bits are reserved and must be cleared.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

If the /R flag is used to specify the date format, this overrides the setting of bit 0.

loadFlags is a bitwise parameter that controls the overall load. The bits are defined as follows:

bit 0: If set, ignores blanks at the end of a column. This is appropriate if the file contains columns of unequal length. Set *loadFlags* = 1 to set bit 0.

bit 1: If set, when the /W flag is specified and the line containing column labels starts with one or more space characters, the spaces are taken to be a blank column name. The resulting column will be named Blank. Use this if both the line containing column labels and the lines containing data start with leading spaces in a space-delimited file. Set loadFlags = 2 to set bit 1.

bit 2: If set: Disables pre-counting of lines of data. See **Loading Very Large Files** below.

bit 3: If set: Disables unescaping of backslash characters in text columns. See **Escape Sequences** below.

All other bits are reserved and you should use 0 for them.

/W Looks for wave names in a file. (With /G, /F, and /J.)

Use /W/A to read wave names from the file and then continue the load without displaying the normal wave name dialog.

Details

Without the /G, /F, /J, or /T flags, LoadWave loads Igor Binary files.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-34 for details.

When loading a general text file, the delimiters are always tab, comma and space.

Loading Custom Date Formats

Here are some examples showing custom date formats and how you would specify them using the /R flag:

Date Format	Specification
October 11, 1999	`/R={English, 2, 4, 1, 1, "Month DayOfMonth, Year", 40}`
Oct 11, 1999	`/R={English, 2, 3, 1, 1, "Month DayOfMonth, Year", 40}`
11 October 1999	`/R={English, 2, 4, 1, 1, "DayOfMonth Month Year", 40}`
11 Oct 1999	`/R={English, 2, 3, 1, 1, "DayOfMonth Month Year", 40}`
10/11/99	`/R={English, 1, 2, 1, 1, "Month/DayOfMonth/Year", 40}`
11-10-99	`/R={English, 1, 2, 2, 1, "DayOfMonth-Month-Year", 40}`
11-Jun-99	`/R={English, 1, 3, 2, 1, "DayOfMonth-Month-Year", 40}`
991011	`/R={English,1,2,2,1,"YearMonthDayOfMonth", 40}`
19991011	`/R={English,2,2,2,1,"YearMonthDayOfMonth", 40}`

When loading data as delimited text, if you use a date format containing a comma, such as "October 11, 1999", you must use the /V flag to make sure that LoadWave will not treat the comma as a delimiter.

When loading a date format that consists entirely of digits, such as 991011, you must use the LoadWave/B to specify that the data is a date. Otherwise, LoadWave will treat it as a regular number.

Loading Very Large Files

The number of waves (columns) or points (rows) that LoadWave can handle when loading a text file is limited only by available memory.

You can improve the speed and efficiency of loading very large files (i.e., more than 50,000 lines of data) using the *numLines* parameter of the /L flag. Normally this parameter is used to load a section of the file instead of the whole file. However, in delimited, general text and fixed field text loads, the *numLines* parameter also specifies how many rows the waves should initially have. Thus all of the required memory is allocated at the start of the load, rather than increasing the number of wave rows over and over as more lines of data are loaded. When loading very large files, if you know the exact number of lines of data in the file, use the *numLines* parameter of the /L flag. If you don't know the exact number of lines, you can provide a number that is guaranteed to be larger.

As of Igor Pro 6.02, if you omit the /L flag or if the *numLines* parameter is zero, and if you are loading a file greater than 500,000 bytes, and if you are running on Windows, LoadWave will automatically count the lines of data in the file so that the entire wave can be allocated before data loading starts. This acts as if you used /L and set *numLines* to the exact correct value. For very large files on Windows, this can speed the loading process considerably. For small files on Windows and for files of any size on Macintosh, it actually makes the load slower. That's why this feature takes effect only on Windows and only for files of greater than 500,000 bytes. You can disable this feature by using the /V flag and setting bit 2 to 1.

Escape Sequences

An escape sequence is a two-character sequence used to represent special characters in plain text. Escape sequences are introduced by a backslash character.

By default, in a text column, LoadWave interprets the following escape sequences: \t (tab), \n (linefeed), \r (carriage-return), \\ (backslash), \" (double-quote) and \' (single-quote). This works well with Igor's **Save** operation which uses escape sequences to encode the first four of these characters.

If you are loading a file that does not use escape sequences but which does contain backslashes, you can disable interpretation of these escape sequences by setting bit 3 of the *loadFlags* parameter of the /V flag. This is mainly of use for loading a text file that contains unescaped Windows file system paths.

Generating Wave Names

The /N flag automatically names new waves "wave" (or *baseName* if =*baseName* is used) plus a digit. The digit starts from zero and increments by one for each wave loaded from the file. If the resulting name conflicts with an existing wave, the existing wave is overwritten.

The /A flag is like /N except that it skips names already in use.

Specifying Characteristics of Individual Columns

The /B=*columnInfoStr* flag provides information to LoadWave for each column in a delimited text (/J), fixed field text (/F) or general text (/G) file. The flag overrides LoadWave's normal behavior. In most cases, you will not need to use it.

columnInfoStr is constructed as follows:

```
"<column info>;<column info>; … ;<column info>;"
```

where <column info> consists of one or more of the following:

C=<number> The number of columns controlled by this column info specification. <number> is an integer greater than or equal to one.

F=<format> A code that specifies the data type of the column or columns. <format> is an integer from -2 to 10. The meaning of the <format> is:

- -2: Text. The column will be loaded into a text wave.
- -1: Format unknown. It will deduce the format.
- 0 to 5: Numeric.
- 6: Date.
- 7: Time.
- 8: Date/Time.
- 9: Octal number.
- 10: Hexadecimal number.

The F= flag is used for delimited text and fixed field text files only. It is ignored for general text files.

N=<name> A name to use for the column. <name> can be a standard name (e.g., wave0) or a quoted liberal name (e.g., 'Heart Rate'). If <name> is '_skip_' (including single quotation marks) then LoadWave will skip the column.

The N= flag works for delimited text, fixed field text and general text files.

T=<numtype> A number that specifies what the numeric type for the column should be. This flag overrides the LoadWave/D flag. It has no effect on columns whose format is text. <numtype> must be one of the following:

- 2: 32-bit float.
- 4: 64-bit float.
- 8: 8-bit signed integer.

16: 16-bit signed integer.

32: 32-bit signed integer.

72: 8-bit unsigned integer.

80: 16-bit unsigned integer.

96: 32-bit unsigned integer.

W=<width> The column field width for fixed field files. <width> is an integer greater than or equal to one. Fixed width files are FORTRAN-style files in which a fixed number of characters is allocated for each column and spaces are used as padding.

The W= flag is used for fixed field text only.

Here is an example of the /B=*columnInfoStr* flag:

```
/B="C=1,F=-2,T=2,W=20,N=Factory; C=1,F=6,W=16,T=4,N=MfgDate;
C=1,F=0,W=16,T=2,N=TotalUnits; C=1,F=0,W=16,T=2,N=DefectiveUnits;"
```

This example is shown on two lines but in a real command it would be on a single line. In a procedure, it could be written as:

```
String columnInfoStr = ""
columnInfoStr += "C=1,F=-2,T=2,W=20,N=Factory;"
columnInfoStr += "C=1,F=6,T=4,W=16,N=MfgDate;"
columnInfoStr += "C=1,F=0,T=2,W=16,N=TotalUnits;"
columnInfoStr += "C=1,F=0,T=2,W=16,N=DefectiveUnits;"
```

Note that each flag inside the quoted string ends with either a comma or a semicolon. The comma separates one flag from the next within a particular column info specification. The semicolon marks the end of a column info specification. The trailing semicolon is required. Spaces and tabs are permitted within the string.

This example provides information about a file containing four columns.

The first column info specification is "`C=1;F=-2,T=2,W=20,N=Factory;`". This indicates that the specification applies to one column, that the column format is text, that the numeric format is single-precision floating point (but this has no effect on text columns), that the column data is in a fixed field width of 20 characters, and that the wave created for this column is to be named Factory.

The second column info specification is "`C=1;F=6,T=4,W=16,N=MfgDate;`". This indicates that the specification applies to one column, that the column format is date, that the numeric format is double-precision floating point (double precision should always be used for dates), that the column data is in a fixed field width of 16 characters, and that the wave created for this column is to be named MfgDate.

The third column info specification is "`C=1;F=0,T=2,W=16,N=TotalUnits;`". This indicates that the specification applies to one column, that the column format is numeric, that the numeric format is single-precision floating point, that the column data is in a fixed field width of 16 characters, and that the wave created for this column is to be named TotalUnits.

The fourth column info specification is the same as the third except that the wave name is DefectiveUnits.

All of the items in a column specification are optional. The default value for each item in the column info specification is as follows:

C=<number> C=1. Specifies that the column info describes one column.

F=<format> F=-1. Determines the format as dictated by the /K flag. If /K=0 is used, LoadWave will automatically determine the column format.

N=<name> N=_auto_. Generates the wave name as it would if the /B flag were omitted.

T=<numtype> Defaults to T=4 (double precision) if the LoadWave/D flag is used or to T=2 (single precision) if the /D flag is omitted.

W=<width> W=0. For a fixed width file, LoadWave will use the default field width specified by the /F flag unless you provide an explicit field width greater than 0 using W=<width>.

Taking advantage of the default values, we could abbreviate the example as follows:

```
/B="F=-2,W=20,N=Factory; F=6,T=4,W=16,N=MfgDate;
W=16,N=TotalUnits; W=16,N=DefectiveUnits;"
```

If the file were not a fixed field text file, we would omit the W= flag and the example would become:

```
/B="F=-2,N=Factory;F=6,T=4,N=MfgDate;N=TotalUnits;N=DefectiveUnits;"
```

Here are some more examples and discussion that illustrate the use of the /B=*columnInfoStr* flag.

In this example, the /B flag is used solely to specify the name to use for the waves created from the columns in the file:

```
/B="N=WaveLength; N=Absorbance;"
```

The wave names in the previous example are standard names. If you want to use liberal names, such as names containing spaces or dots, you must use single quotes. For example:

```
/B="N='Wave Number'; N='Reflection Angle';"
```

The name that you specify via N= can not be used if overwrite is off and there is already a wave with this name or if the name conflicts with a macro, function or operation or variable. In these cases, LoadWave generates a unique name by adding one or more digits to the name specified by the N= flag for the column in question. You can avoid the problem of a conflict with another wave name by using the overwrite (/O) flag or by loading your data into a newly-created data folder. You can minimize the likelihood of a name conflict with a function, operation or variable by avoiding vague names.

If you specify the same name in two N= flags, LoadWave will generate an error, so make sure that the names are unique.

Except if the specified name is '_skip_', the N= flag generates a name for one column only, even if the C= flag is used to specify multiple columns. Consider this example:

```
/B="C=10,N=Test;"
```

This ostensibly uses the name Test for 10 columns. However, wave names must be unique, so LoadWave will not do this. It will use the name Test for just the first column and the other columns will receive default names.

Also, when loading data into a matrix wave, LoadWave uses only one name. If you specify more than one name, only the first is used.

In this example, the /B flag solely specifies the format of each column in the file. The file in question starts with a text column, followed by a date column, followed by 3 numeric columns.

```
/B="F=-2; F=6; C=3,F=0"
```

In most cases, it is not necessary to use the F= flag because LoadWave can automatically deduce the formats. The flag is useful for those cases where it deduces the column formats incorrectly. It is also useful to force LoadWave to interpret a column as octal or hexadecimal because LoadWave can not automatically deduce these formats.

The numeric codes (0…10) used by the F= flag are the same as the codes used by the ModifyTable operation. If you create a table using the /E flag, the F= flag controls the numeric format of table columns.

The code -1 is not a real column format code. If you use F=-1 for a particular column, LoadWave will deduce the format for that column from the column text.

In this example, the /B flag is used solely to specify the width of each column in a fixed field file. This file contains a 20 character column followed by ten 16 character columns followed by three 24 character columns.

```
/B="C=1,W=20; C=10,W=16; C=3,W=24"
```

The field widths specified via W= override the default field width specified by the /F flag. If all of the columns in the file have the same field width then you can use just the /F flag.

You can load a subset of the columns in the file using the /L flag. Even if you do this, the column info specifications that you provide via the /B flag start from the first column in the file, not from the first column to be loaded.

Output Variables

LoadWave sets the following variables:

`V_flag`	Number of waves loaded.
`S_fileName`	Name of the file being loaded.
`S_path`	File system path to the folder containing the file.
`S_waveNames`	Semicolon-separated list of the names of loaded waves.

The setting of S_path was added in Igor Pro 3.14. S_path uses Macintosh path syntax (e.g., "`hd:FolderA:FolderB:`"), even on Windows. It includes a trailing colon. If LoadWave is loading from the Clipboard, S_path is set to `""`.

When LoadWave presents an Open File dialog and the user cancels, V_flag is set to 0 and S_fileName is set to `""`.

See Also

The **ImageLoad** operation.

See Chapter II-9, **Importing and Exporting Data** for further information on loading waves, including loading multidimensional data from HDF, PICT, TIFF and other graphics files. Check the "More Extensions:File Loaders" folder other file-loader extensions.

Setting Bit Parameters on page IV-12 for further details about bit settings.

Loess

```
Loess [flags] srcWave = srcWaveName [, factors = factorWaveName1
    [, factorWaveName2 …]]
```

The Loess operation smooths *srcWaveName* using locally-weighted regression smoothing. This algorithm is sometimes classified as a "nonparametric regression" procedure. The regression can be constant, linear, or quadratic. A robust option that ignores outliers is available. See **Basic Algorithm**, **Robust Algorithm**, and **References** for additional details and terminology.

This implementation works with waveforms, XY pairs of waves, false-color images, matrix surfaces, and multivariate data (one dependent data wave with multiple independent variable data waves).

Unlike the **FilterFIR** operation, Loess discards any NaN input values and will not generate a result that is wholly NaN.

Parameters

srcWaveName is the input data to be smoothed. It may be a one-dimensional or a two-dimensional wave, and it may contain NaNs.

When no /DEST flag is specified, Loess will overwrite *srcWaveName* with the smoothed result.

If *srcWaveName* is one-dimensional and no factors are provided, X values are derived from the X scaling of *srcWaveName*.

If *srcWaveName* is two-dimensional, the factors keyword is not permitted and the X and Y values are derived from the X and Y scaling of *srcWaveName*.

Higher dimensions of *srcWaveName* are not supported.

The optional factors parameter(s) provide the independent variable value(s) that correspond to the observed value in *srcWaveName*.

Note: Cleveland et al. (1992) use the term "multiple factors" instead of "multivariate", hence the keyword name "factors" is used to denote these waves.

Use one factors wave when *srcWaveName* is the one-dimensional Y wave of an XY data pair:

```
srcWaveName[i] = someFunction(factorWaveName1[i])
```

Use multiple factors waves when *srcWaveName* contains the values of a multivariate function. "Multivariate" means that *srcWaveName* contains the observed results of a process that combines multiple independent input variables:

```
srcWaveName[i] = someFunction(factorWaveName1[i], factorWaveName2[i],…)
```

A maximum of 10 factors waves is supported.

All factors wave(s) must be numeric, noncomplex, one-dimensional and have the same number points as *srcWaveName*.

Any NaN values in *srcWaveName*[i], *factorWaveName1*[i], *factorWaveName2*[i], … cause all corresponding values to be ignored. Factors waves may contain NaN values only when /DFCT is specified.

Loess does not support NaNs in any of *destFactorWaveName1*, *destFactorWaveName2*,… and the results are undefined.

Flags

/CONF={*confInt*, *ciPlusWaveName* [,*ciMinusWaveName*]}
: *confInt* specifies the confidence interval (a probability value from 0 to 1). *ciPlusWaveName* and the optional *ciMinusWaveName* are the names of new or overwritten output waves to hold the fitted value ± the confidence interval.

  Note: /CONF uses large memory allocations, approximately N*N*8 bytes, where N is the number of points in *srcWaveName* (see **Memory Details**).

/DEST=*destWaveName*
: Specifies the name of the wave to hold the smoothed data. It creates *destWaveName* if it does not already exist or overwrites it if it does. The x (and possibly y) scaling

of *destWave* determines the independent (factor) coordinates unless /DFCT={*destFactorWaveName1* [,*destFactorWaveName2*...]} is also specified.

/DFCT
Specifies that the /DEST wave's x (and possibly y) scaling determines the independent (factor) coordinates at which to compute the smoothed data.

/DFCT={*destFactorWaveName1* [,*destFactorWaveName2*…]}
Specifies the names of one-dimensional waves providing the independent coordinates at which to compute the smoothed data.

If /DFCT={...} is used, the same number of waves must be specified for /DFCT and for factors = {*factorWaveName1* [, *factorWaveName2*…]}, though their lengths may (and usually will) be different. The length of *destFactorWaveName* waves must be the same as that of the *destWaveName* wave.

All destination factor waves must be numeric, noncomplex, and one-dimensional. The number of destination factor waves must match the number of source factor waves (if specified), or match the dimensionality of *srcWaveName* (one destination factor wave if *srcWaveName* is one-dimensional, two destination factors waves if *srcWaveName* is two-dimensional.)

The values in the destination factor waves may not be NaN.

/E=*extrapolate*
Set *extrapolate* to nonzero to use a slower fitting method that computes values beyond the domain defined by the source factors. This is the "surface" parameter named "direct" in Cleveland et al. (1992). The default is *extrapolate* = 0, which uses the "interpolate" surface parameter, instead.

/N=*neighbors*
Specifies the number of values in the smoothing window.

If *neighbors* is even, the next larger odd number is used. When *neighbors* is less than two, no smoothing is done.

The default is `0.5*numpnts(`*`srcWaveName`*`)` rounded up to the next odd integer or 3, whichever is larger.

Use either /N or /SMTH, but not both.

/NORM [=*norm*]
Set *norm* to 0 when specifying multiple factors and they all have the same scale and meaning, for example multiple factors all in units of meters.

The default is *norm* = 1, which normalizes each factor independently when computing the weighting function. This is appropriate when the factors are not dimensionally related, for example one factor measures wavelength and another measures temperature.

/ORD=*order*
Specifies the regression (fitting) order, the *d* parameter in Cleveland (1977):

order=0: Fits a constant to the locally-weighted neighbors around each point.

order=1: Fits a line to the locally-weighted neighbors around each point (Lowess smoothing).

order=2: Default; fits a quadratic (Loess smoothing).

/PASS=*passes*
The number of iterations of local weighting and regression fitting performed. The minimum is 1 and the default is 4. In Cleveland (1977), *passes* corresponds to *r*.

/R [=*robust*]
Set *robust* to nonzero to use a robust fitting method that uses a bisquare weight function instead of the normal tricube weight function. This corresponds to the "symmetric" family in Cleveland et al. (1992). The robust method is less affected by outliers. The default is *robust* = 0, which is the "gaussian" family in Cleveland et al. (1992).

/SMTH=*sf*
Another way to express the number of values in the smoothing window, $0 \le sf \le 1$. The default is 0.5.

To compute *neighbors* from *sf*, use:

`neighbors = 1+floor(`*`sf`*`*numpnts(`*`srcWaveName`*`))`.

Use either /N or /SMTH, but not both.

/TIME=*secs*
secs is the number of seconds allowed to complete the calculation before either warning (default) or stopping.

If the stop bit (4) of /V=*verbose* is set, the caculcation stops after the alloted time. If the diagnostic bit of /V=*verbose* is also set, warnings about the calculation exceeding the allotted time are printed to the history window.

As an example, use /TIME=30/V=6 to abort calculations longer than 30 seconds and print the warning to the history window.

/V [=*verbose*] Controls how much information to print to the history window. *verbose* is a binary coded number with each bit controlling one aspect:

verbose=0: Prints nothing to the history area (default).

verbose=1: Prints the number of observations, equivalent number of parameters, and residual standard error.

verbose=2: Prints diagnostic information and error messages.

verbose=4: Use with /TIME. If this bit is set, calculations that exceed *secs* seconds are aborted.

Set *verbose* to 6 to both limit the time and print diagnostic and error messages.

/V alone is the same as /V=3, which prints all information.

S_info contains all the informational messages regardless of the value of verbose.

/Z[=*z*] Set *z* to nonzero to prevent an error from stopping execution. Use the V_flag variable to see if the smoothing succeeded.

Basic Algorithm

The basic locally-weighted regression algorithm fits a constant, line, or quadratic to each point of the source data, using data that falls within the given span of neighbors over the factor data. Data outside of the span is ignored (given a weight of zero), and data inside the span is given a weight that depends on the distance of the data from the point being evaluated: data closer to the point being evaluated have higher weights and have a greater affect on the fit.

The basic algorithm uses the "tricube" weighting function to emphasize near values and deemphasize far values. For the one-factor case (simple XY data), the weighting function can be expressed as:

$$w_i = \left(1 - \left|\frac{(x - x_i)}{\max_i(x - x_i)}\right|^3\right)^3$$

where $\max_q(x - x_i)$ is the maximum Euclidean distance of the q factor values within the given span from the factor point (x) whose observation (y) value is being evaluated.

The weights are applied to the factor values in the span to compute the constant, linear, or quadratic regression at x.

When multiple factors are used, the Euclidean distance is computed using one dimension per factor. The default is to normalize each factor's range by the standard deviation of that factor's values before computing the Euclidean distances. When factors are dimensionally equal, use the /NORM=0 option to skip this normalization. (See /NORM, about "dimensionally equal".)

Robust Algorithm

The robust algorithm adds to the basic algorithm a method to identify and remove outliers by rejecting values that exceed a threshold related to the "median absolute deviation" of the basic regression's residuals. The remaining values are used to compute robust "bisquare" weighting values:

$$r_i = \begin{cases} \left(1 - \left|\dfrac{e_i}{6 \bullet \text{median}(|e_i|)}\right|^2\right)^2 & \text{for } 0 \le |e_i| < 6 \bullet median(|e_i|) \\ 0 & \text{for } (|e_i| \ge 6 \bullet median(|e_i|)) \end{cases}$$

where e_i is the difference between the observed value and the regression's fitted value, and $\text{median}(|e_i|)$ is evaluated for all the observed values.

These robust weighting values are multiplied with the original weighting values and a new regression (with new residuals) is computed. This process repeats 4 times by default. Use the /PASS flag to specify a different number of repetitions.

Details

Loess sets the variable V_flag to 0 if the smoothing was successful, or to an error code if not. Unlike other operations, the /Z flag allows execution to continue even if input parameters are in error.

Information printed to the history area when /V is set is always stored in the S_info string, even if /V=0 (the default). S_Info also contains the error message text if V_flag is an error code.

The error messages are described in Cleveland et al. (1992). They are often more dire than they seem.

The error message "Span too small. Fewer data values than degrees of freedom" usually means that the /SMTH or /N values are too small. The error code returned in V_Flag for this case is 1106.

The "Extrapolation not allowed with blending" (V_Flag = 1115) error usually means that the destination factors are trying to compute observations outside of the source factors domain without specifying /E=1. This happens if the /DEST *destWaveName* already exists and has X scaling that extends beyond the X scaling of *srcWaveName*. The solution is either kill the /DEST wave, limit the X scaling to the domain of the source wave, or use /E=1.

Memory Details

Loess requires a lot of memory, especially with the /CONF flag. Even without /CONF, the memory allocations exceed this approximation:

Number of bytes allocated = number of points in *srcWaveName* * 216

With /CONF, Loess can allocate large amounts of memory, approximately N*N*8 bytes, where N is the number of points in *srcWaveName*. The 2GB memory limit of 32-bit addressing limits *srcWaveName* to approximately 10,000 points when using /CONF.

More precisely, the memory allocation may be approximated by this function:

```
Function ComputeLoessMemory(srcPoints, numFactorsWaves, doConfidence)
   Variable srcPoints         // number of points in srcWave, aka N
   Variable numFactorsWaves// 1 or number of factors (independent variables)
   Variable doConfidence      // true if /CONF is specified
   Variable doubles= 9 * srcPoints            // 9 allocated double arrays
   doubles += 5 * numFactorsWaves * srcPoints // 5 more arrays
   doubles += (1+numFactorsWaves) * srcPoints // another array
   doubles += (1+numFactorsWaves) * srcPoints // another array
   doubles += (4+5) * srcPoints               // two more arrays
   if( doConfidence )
      doubles += srcPoints*srcPoints          // one HUGE array
   endif
   Variable bytes= doubles * 8
   return bytes
End

Macro DemoLoessMemory()
   Make/O wSrcPoints={10,100,1000,2000,3000,5000,7500,10000,12500,15000,20000}
   Duplicate/O wSrcPoints, loessMemory, loessMemory3, loessMemoryConf
   SetScale d, 0,0, "Points", wSrcPoints
   SetScale d, 0,0, "Bytes", loessMemory, loessMemory3, loessMemoryConf
   loessMemory= ComputeLoessMemory(wSrcPoints[p],1, 0)// 1 factor (X) no /CONF
   loessMemory3= ComputeLoessMemory(wSrcPoints[p],3, 0)// 3 factors (X,Y,Z) no /CONF
   loessMemoryConf= ComputeLoessMemory(wSrcPoints[p],1, 1)// 1 factor (X)with /CONF
   Display loessMemory vs wSrcPoints; Append loessMemory3 vs wSrcPoints
   ModifyGraph highTrip(bottom)=1e+08, rgb(loessMemory3)=(0,0,65535)
   ModifyGraph lstyle(loessMemory3)=2
   Legend
   Display loessMemoryConf vs wSrcPoints
   AutoPositionWindow
   ModifyGraph highTrip(bottom)=1e+08
End
```

Examples

1-D, factors are X scaling, output in new wave:

```
Make/O/N=200 wv=2*sin(x/8)+gnoise(1)
KillWaves/Z smoothed                    // ensure Loess creates a new wave
Loess/DEST=smoothed srcWave=wv          // 21-point loess.
Display wv; ModifyGraph mode=3,marker=19
AppendtoGraph smoothed; ModifyGraph rgb(smoothed)=(0,0,65535)
```

1-D, output in existing wave with more points than original data:

```
Make/O/N=100 short=2*cos(x/4)+gnoise(1)
Make/O/N=300 out; SetScale/I x, 0, 99, "" out  // same X range
```

```
Loess/DEST=out/DFCT/N=30 srcWave=short
Display short; ModifyGraph mode=3,marker=19
AppendtoGraph out
ModifyGraph rgb(out)=(0,0,65535),mode(out)=2,lsize(out)=2
```

1-D Y vs X wave data interpolated to waveform (Y vs X scaling) with 99% confidence interval outputs:

```
// NOx = f(EquivRatio)
// Y wave
// Note: The next 2 Make commands are wrapped to fit on the page.
Make/O/D NOx = {4.818, 2.849, 3.275, 4.691, 4.255, 5.064, 2.118, 4.602, 2.286, 0.97,
3.965, 5.344, 3.834, 1.99, 5.199, 5.283, 3.752, 0.537, 1.64, 5.055, 4.937, 1.561};

// X wave (Note that the X wave is not sorted)
Make/O/D EquivRatio = {0.831, 1.045, 1.021, 0.97, 0.825, 0.891, 0.71, 0.801,  1.074,
1.148, 1, 0.928, 0.767, 0.701, 0.807, 0.902,   0.997, 1.224, 1.089, 0.973, 0.98, 0.665};

// Interpolate to dense waveform over X range
Make/O/D/N=100 fittedNOx
WaveStats/Q EquivRatio
SetScale/I x, V_Min, V_max, "", fittedNOx
Loess/CONF={0.99,cp,cm}/DEST=fittedNOx/DFCT/SMTH=(2/3) srcWave=NOx, factors={EquivRatio}
Display NOx vs EquivRatio; ModifyGraph mode=3,marker=19
AppendtoGraph fittedNOx, cp,cm       // fit and confidence intervals
ModifyGraph rgb(fittedNOx)=(0,0,65535)
ModifyGraph mode(fittedNOx)=2,lsize(fittedNOx)=2
```

Interpolate X, Y, Z waves as a 3D surface.

```
// Note: The next 3 Make commands are wrapped to fit on the page.
Make/O/D vels= {1769, 1711, 1538, 1456, 1608, 1574, 1565, 1692, 1538, 1505, 1764, 1723,
1540, 1441, 1428, 1584, 1552, 1690, 1673, 1548, 1485, 1526, 1536, 1591, 1671, 1647, 1608,
1562, 1740, 1753, 1590, 1466, 1409, 1429}
Make/O/D ews={8.46279, 3.46303, -1.51508, -6.51483, 16.597, -5.95541, -28.5078, 9.68438,
-6.00159, -21.7557, 14.263, 6.02058, -2.25772, -10.536, -18.7785, 10.7509, -6.07024,
1.77531, 0.767701, -0.235545, -1.24315, 21.7298, 10.3964, 0.133859, -10.1733, -20.4359,
13.7658, -8.88429, 10.8869, 4.91318, -0.0649319, -5.06469, -10.0428, -11.0601}
Make/O/D nss={-38.1732, -15.6207, 6.83407, 29.3865, 3.67947, -1.32028, -6.32004, -
10.3852, 6.43591, 23.3302, -37.1565, -15.6842, 5.88156, 27.4473, 48.9196, 10.0254, -
5.66059, -40.6613, -17.5832, 5.39486, 28.4729, 43.5833, 20.852, 0.26848, -20.4045, -
40.988, 3.0518, -1.9696, -49.1077, -22.1619, 0.292889, 22.8453, 45.3001, 49.8887}

// Evaluate the smoothed function as interpolated image
Make/O/N=(50,50) velsImage
WaveStats/Q ews
SetScale/I x, V_Min, V_Max, "" velsImage   // destination factors
WaveStats/Q nss
SetScale/I y, V_Min, V_Max, "" velsImage   // are X and Y scaling

Loess/DEST=velsImage/DFCT/NORM=0/SMTH=0.75/E/Z srcWave=vels, factors={ews,nss}

// Display source data as a contour with x, y markers.
Display; AppendXYZContour vels vs {ews,nss}
ModifyContour vels xymarkers=1, labels=0
ColorScale

// Display interpolated surface as an image
AppendImage velsImage
ModifyImage velsImage ctab= {*,*,Grays256,0}
ModifyGraph mirror=2
```

References

Cleveland, W.S., Robust locally weighted regression and smoothing scatterplots, *J. Am. Stat. Assoc.*, *74*, 829-836, 1977.

Cleveland, W.S., E. Grosse, and M.-J. Shyu, A Package of C and Fortran Routines for Fitting Local Regression Models, Technical Report, Bell Labs, 54pp, 1992. <http://cm.bell-labs.com/cm/ms/departments/sia/wsc/webpapers.html>.

NIST/SEMATECH, LOESS (aka LOWESS), in *NIST/SEMATECH e-Handbook of Statistical Methods*, <http://www.itl.nist.gov/div898/handbook/pmd/section1/pmd144.htm>, 2005.

See Also

Smooth, Interpolate XOP, **interp**, **MatrixFilter**, **MatrixConvolve**, and **ImageInterpolate**.

log

log(*num*)

The log function returns the log base 10 of *num*.

It returns -INF if *num* is 0, and returns NaN if *num* is less than 0.

To compute a logarithm base n use the formula:

$$\log_n(x) = \frac{\log(x)}{\log(n)}.$$

See Also

The **ln** function.

logNormalNoise

logNormalNoise(*m*,*s*)

The logNormalNoise function returns a pseudo-random value from the lognormal distribution function whose probability distribution function is

$$f(x,m,s) = \frac{1}{xs\sqrt{2\pi}} \exp\left\{\frac{\left(\ln(x)-m\right)^2}{2s^2}\right\},$$

with a mean $\exp\left(m+\frac{1}{2}s^2\right)$,

and variance $\exp\left(2m^2+s^2\right)\left[\exp\left(s^2\right)-1\right]$. .

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-332.

Chapter III-12, **Statistics** for a function and operation overview.

LombPeriodogram

LombPeriodogram [*flags*] *srcTimeWave, srcAmpWave* [, *srcFreqWave*]

The LombPeriodogram is used in spectral analysis of signal amplitudes specified by *srcAmpWave* which are sampled at possibly random sampling times given by *srcTimeWave*. The only assumption about the sampling times is that they are ordered from small to large time values. The periodogram is calculated for either a set of frequencies specified by *srcFreqWave* (slow method) or by the flags /FR and /NF (fast method). Unless you specify otherwise, the results of the operation are stored by default in W_LombPeriodogram and W_LombProb in the current data folder.

Flags

/DESP=*datafolderAndName*

Saves the computed P-values in a wave specified by *datafolderAndName*. The destination wave will be created or overwritten if it already exists. *dataFolderAndName* can include a full or partial path with the wave name.

Creates by default a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-56 for details.

If this flag is not specified, the operation saves the P-values in the wave W_LombProb in the current data folder.

/DEST=*datafolderAndName*

Saves the computed periodogram in a wave specified by *datafolderAndName*. The destination wave will be created or overwritten if it already exists. *datafolderAndName* can include a full or partial path with the wave name (`/DEST=root:bar:destWave`).

Creates by default a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-56 for details.

If this wave is not specified the operation saves the resulting periodogram in the wave W_LombPeriodogram in the current data folder.

/FR=fRes — Use /FR to specify the frequency resolution of the output. This flag is used together with /NF to specify the range of frequencies for which the periodogram is computed. Note that *fRes* is also the lowest frequency in the output.

/NF=numFreq — Use /NF to specify the number of frequencies at which the periodogram is computed. The range of frequencies of the periodogram is then [*fRes*, (*numFreq*-1)**fRes*].

/Q — Quiet mode; suppresses printing results in the history area.

/Z — Do not report any errors.

Details

The LombPeriodogram (sometimes referred to as "Lomb-Scargle" periodogram) is useful in detection of periodicities in data. The main advantage of this approach over Fourier analysis is that the data are not required to be sampled at equal intervals. For an input consisting of N points this benefit comes at a cost of an O(N^2) computations which becomes prohibitive for large data sets. The operation provides the option of computing the periodogram at equally spaced (output) frequencies using /FR and /NF or at completely arbitrary set of frequencies specified by *srcFreqWave*. It turns out that when you use equally spaced output frequencies the calculation is more efficient because certain parts of the calculation can be factored.

The Lomb periodogram is given by

$$LP(\omega) = \frac{1}{2\sigma^2}\left\{\frac{\left[\sum_{i=0}^{N-1}\left(y_i-\bar{y}\right)\cos\left[\omega\left(t_i-\tau\right)\right]\right]^2}{\sum_{i=0}^{N-1}\cos^2\left[\omega\left(t_i-\tau\right)\right]}+\frac{\left[\sum_{i=0}^{N-1}\left(y_i-\bar{y}\right)\sin\left[\omega\left(t_i-\tau\right)\right]\right]^2}{\sum_{i=0}^{N-1}\sin^2\left[\omega\left(t_i-\tau\right)\right]}\right\}$$

Here yi is the ith point in *srcAmpWave*, ti is the corresponding point in *srcTimeWave*,

$$\bar{y} = \frac{1}{N}\sum_{i=0}^{N-1} y_i ,$$

$$\tan(2\omega\tau) = \frac{\sum_{i=0}^{N-1}\sin(2\omega t_i)}{\sum_{i=0}^{N-1}\cos(2\omega t_i)} .$$

and

$$p = 1-\left\{1-\exp\left[LP(w)\right]\right\}^{N_{ind}} .$$

In the absence of a Nyquist limit, the number of independent frequencies that you can compute can be estimated using:

$$N_{ind} = -6.362 + 1.193N + 0.00098N^2 .$$

This expression was given by Horne and Baliunas derived from least square fitting. Nind is used to compute the P-values as:

$$p = 1 - \left\{1 - \exp\left[LP(w)\right]\right\}^{N_{ind}} .$$

Note that you can invert the last expression to determine the value of LP(w) for any significance level.

See Also
The **FFT** and **DSPPeriodogram** operations.

References
1. J.H. Horne and S.L. Baliunas, *Astrophysical Journal*, 302, 757-763, 1986.

2. N.R. Lomb, *Astrophysics and Space Science*, 39, 447-462, 1976.

3. W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes*, 3rd ed., Section 13.8.

lorentzianNoise

`lorentzianNoise(a,b)`
The function returns a pseudo-random value from a Lorentzian distribution

$$f(x) = \frac{1}{\pi}\frac{(b/2)}{(x-a)^2 + (b/2)^2} .$$

Here *a* is the center and *b* is the full line width at half maximum (FWHM).

See Also
SetRandomSeed, **enoise**, **gnoise**.

Noise Functions on page III-332.

Chapter III-12, **Statistics** for a function and operation overview.

LowerStr

`LowerStr(str)`
The LowerStr function returns a string expression identical to *str* except that all upper-case characters are converted to lower-case.

See Also
The **UpperStr** function.

Macro

`Macro macroName([parameters]) [:macro type]`
The Macro keyword introduces a macro. The macro will appear in the Macros menu unless the procedure file has an explicit Macros menu definition. See Chapter IV-4, **Macros** and **Macro Syntax** on page IV-98 for further information.

MacroList

`MacroList(matchStr, separatorStr, optionsStr)`
The MacroList function returns a string containing a list of the names of user-defined procedures that start with the Proc, Macro, or Window keywords that also satisfy certain criteria. Note that if the procedures need to be compiled, then MacroList may not list all of the procedures.

Parameters

Only macros having names that match *matchStr* string are listed. See **WaveList** for examples.

The first character of *separatorStr* is appended to each macro name as the output string is generated. *separatorStr* is usually ";" for list processing (See **Processing Lists of Waves** on page IV-174 for details on list processing).

optionsStr is used to further qualify the macros. It is a string containing keyword-value pairs separated by commas. Available options are:

KIND:*nk*
- *nk*=1: List Proc procedures.
- *nk*=2: List Macro procedures.
- *nk*=4: List Window procedures.

nk can be the sum of these values to match multiple procedure kinds. For example, use 3 to list both Proc and Macro procedures.

NPARAMS:*np* Restricts the list to macros having exactly *np* parameters. Omitting this option lists macros having any number of parameters.

SUBTYPE:*typeStr* Lists macros that have the type *typeStr*. That is, you could use ButtonControl as *typeStr* to list only macros that are action procedures for buttons.

WIN:*windowNameStr*
Lists macros that are defined in the named procedure window. "Procedure" is the name of the built-in procedure window.

Note: Because *optionsStr* keyword-value pairs are comma separated and procedure window names can have commas in them, the WIN: keyword must be the last one specified.

Examples

To list all Macros with three parameters:

```
Print MacroList("*",";","KIND:2,NPARAMS:3")
```

To list all Macro, Proc, and Window procedures in the main procedure window whose names start with b:

```
Print MacroList("b*",";","WIN:Procedure")
```

See Also

The **DisplayProcedure** operation and the **FunctionList**, **OperationList**, **StringFromList**, and **WinList** functions.

For details on procedure subtypes, see **Procedure Subtypes** on page IV-179, as well as **Button**, **CheckBox**, **SetVariable**, and **PopupMenu**.

magsqr

magsqr(*z*)

The magsqr function returns the sum of the squares of the real and imaginary parts of the complex number *z*, that is, the magnitude squared.

Examples

Assume waveCmplx is complex and waveReal is real.

```
waveReal = sqrt(magsqr(waveCmplx))
```

sets each point of waveReal to the magnitude of the complex points in waveCmplx.

You may get unexpected results if the number of points in waveCmplx differs from the number of points in waveReal because of interpolation. See **Mismatched Waves** on page II-99 for details.

See Also

The **cabs** function.

WaveMetrics provides Igor Technical Note 006, "DSP Support Macros" which uses the magsqr function to compute the magnitude of FFT data, and Power Spectral Density with options such as windowing and segmenting. See the Technical Notes folder. Some of the techniques discussed there are available as Igor procedure files in the "WaveMetrics Procedures:Analysis:" folder.

Make

```
Make [flags] waveName [, waveName]…
Make [flags] waveName [= {n0,n1,…}]…
Make [flags] waveName [= {{n0,n1,…},{n0,n1,…},…}]…
```

The Make operation creates the named waves. Use braces to assign data values when creating the wave.

Flags

/B Makes 8-bit signed integer waves or unsigned waves if /U is present.

/C Makes complex waves.

/D Makes double precision waves.

/DF Wave holds data folder references.

Requires Igor Pro 6.1 or later. For advanced programmers only.

See **Data Folder References** on page IV-61 for more discussion.

/FREE Creates a free wave. Allowed only in functions and only if a simple name or structure field is specified.

Requires Igor Pro 6.1 or later. For advanced programmers only.

See **Free Waves** on page IV-71 for more discussion.

/I Makes 32-bit signed integer waves or unsigned waves if /U is present.

/N=*n* *n* is the number of points each wave will have. If *n* is an expression, it must be enclosed in parentheses: `Make/N=(myVar+1) aNewWave`

/N=(*n1*, *n2*, *n3*, *n4*)

n1, *n2*, *n3*, *n4* specify the number of rows, columns, layers and chunks each wave will have. Trailing zeros can be omitted (e.g., `/N=(n1, n2, 0, 0)` can be abbreviated as `/N=(n1, n2))`.

/O Overwrites existing waves in case of a name conflict. After an overwrite, you cannot rely on the contents of the waves and you will need to reinitialize them or to assign appropriate values.

/R Makes real value waves (default).

/T Makes text waves.

/T=*size* Makes text waves with pre-allocated storage.

size is the number of bytes preallocated by Igor for each element in each text wave. The waves are not initialized - it is up to you to initialize them.

Preallocation can dramatically speed up text wave assignment when the wave has a very large number of points but only when all strings assigned to the wave are exactly the same size as the preallocation size.

/U Makes unsigned Integer waves.

/W Makes 16-bit signed integer waves or unsigned waves if /U is present.

/WAVE Wave holds wave references.

Requires Igor Pro 6.1 or later. For advanced programmers only.

See **Wave References** on page IV-56 for more discussion.

/Y=*type* Specifies wave data type. See details below.

Wave Data Types

As a replacement for the above number type flags you can use /Y=*numType* to set the number type as an integer code. See the **WaveType** function for code values. The /Y flag overrides other type flags but you may still need to use the /C, /T, /DF or /WAVE flags to define the type of an automatic WAVE reference variable when used in user functions.

Details

Unless overridden by the flags, the created waves have the default length, type, precision, units and scaling. The factory defaults are:

Note: The *preferred* precision set by the Miscellaneous Settings dialog only presets the Make Waves dialog checkbox and determines the precision of imported waves. It does not affect the Make operation.

Property	Default
Number of points	128
Precision	Single precision floating point
Type	Real
dimensions	1
x, y, z, and t scaling	offset=0, delta=1 (“point scaling”)
x, y, z, and t units	"" (blank)
Data Full Scale	0, 0
Data units	"" (blank)

See Also
The **SetScale**, **Duplicate**, and **Redimension** operations.

MakeIndex

MakeIndex [**/A/C/R**] ***sortKeyWaves, indexWaveName***
The MakeIndex operation sets the data values of *indexWaveName* such that they give the ordering of *sortKeyWaves*.

For simple sorting problems, MakeIndex is not needed. Just use the **Sort** operation.

Parameters
sortKeyWaves is either the name of a single wave, to use a single sort key, or the name of multiple waves in braces, to use multiple sort keys.

indexWaveName must specify a numeric wave.

All waves must be of the same length and must not be complex.

Flags

/A	Alphanumeric. When *sortKeyWaves* includes text waves, the normal sorting places “wave1” and “wave10” before “wave9”. Use /A to sort the number portion numerically, so that “wave9” is sorted before “wave10”.
/C	Case-sensitive. When *sortKeyWaves* includes text waves, the ordering is case-insensitive unless you use the /C flag which makes it case-sensitive.
/R	Reverse the index so that ordering is from largest to smallest.

Details
MakeIndex is used in preparation for a subsequent **IndexSort** operation. If /R is used the ordering is from largest to smallest. Otherwise it is from smallest to largest.

See Also
MakeIndex and IndexSort Operations on page III-137.

MandelbrotPoint

MandelbrotPoint(*x, y, maxIterations, algorithm*)
The MandelbrotPoint function returns a value between 0 and *maxIterations* based on the Mandelbrot set complex quadratic recurrence relation z[n] = z[n-1]^2 + c where *x* is the real component of c, *y* is the imaginary component of c and z[0] = 0.

The returned value is the number of iterations the equation was evaluated before |z[n]| > 2 (the escape radius of the Mandelbrot set), or maxIterations, whichever is less.

Parameters

algorithm=0	The "Escape Time" algorithm returns the integer n which is the number of iterations until |z[n]| > 2.

algorithm=1 The "Renormalized Iteration Count Algorithm" algorithm returns a floating point value which is a refinement of the number of iterations n by adding the quantity:

5 - ln(ln(|z[n+4]|)) / ln(2)

(which requires four more iterations of the recurrence relation). The returned value is clipped to maxIterations.

See Also

The "MultiThread Mandelbrot Demo" experiment.

References

http://en.wikipedia.org/wiki/Mandelbrot_set

http://linas.org/art-gallery/escape/escape.html

MarcumQ

MarcumQ(*m, a, b*)

The MarcumQ function returns the generalized Q-function defined by the integral

$$Q_m(a,b) = \int_b^\infty u\left(\frac{u}{a}\right)^{m-1} \exp\left(-\frac{(a^2+u^2)}{2}\right) I_{m-1}(au)\,du$$

where I_k is the modified Bessel function of the first kind and order k.

Depending on the input arguments, the MarcumQ function may be computationally intensive but you can abort the calculation at any time.

References

Cantrell, P.E., and A.K. Ojha, Comparison of Generalized Q-Function Algorithms, *IEEE Transactions on Information Theory*, *IT-33*, 591-596, 1987.

MarkPerfTestTime

MarkPerfTestTime *idval*

Use the MarkPerfTestTime operation for performance testing of user-defined functions in conjunction with `SetIgorOption DebugTimer`. When used between `SetIgorOption DebugTimer, Start` and `SetIgorOption DebugTimer, Stop`, MarkPerfTestTime stores the ID value and the time of the call in a buffer. When `SetIgorOption DebugTimer, Stop` is called the contents of the buffer are dumped to a pair of waves: W_DebugTimerIDs will contain the ID values and W_DebugTimerVals will contain the corresponding times of the calls relative to the very first call. The timings use the same high precision mechanism as the startMSTimer and stopMSTimer calls.

By default, `SetIgorOption DebugTimer, Start` allocates a buffer for up to 10000 entries. You can allocate a different sized buffer using `SetIgorOption DebugTimer, Start=bufsize`.

See Also

SetIgorOption, **startMSTimer**, and **stopMSTimer**.

Additional documentation can be found in an example experiment, PerformanceTesting.pxp, and a WaveMetrics procedure file, PerformanceTestReport.ipf.

MatrixConvolve

MatrixConvolve [**/R=*roiWave***] ***coefMatrix, dataMatrix***

The MatrixConvolve operation convolves a small coefficient matrix *coefMatrix* into the destination *dataMatrix*.

Flags

/R=roiWave Modifies only data contained inside the region of interest. The ROI wave should be 8-bit unsigned with the same dimensions as *dataMatrix*. The interior of the ROI is defined by zeros and the exterior is any nonzero value.

Details

On input *coefMatrix* contains an *N*x*M* matrix of coefficients where *N* and *M* should be odd. Generally *N* and *M* will be equal. If *N* and *M* are greater than 13, it is more efficient to perform the using the Fourier transform (see **FFT**).

The convolution is performed in place on the data matrix and is acausal, i.e., the output data is not shifted.

Edges are handled by replication of edge data.

When *dataMatrix* is an integer type, the results are clipped to limits of the given number type. For example, unsigned byte is clipped to 0 to 255.

MatrixConvolve works also when both *coefMatrix* and *dataMatrix* are 3D waves. In this case the convolution result is placed in the wave M_Convolution in the current data folder, and the optional /R=*roiWave* is required to be an unsigned byte wave that has the same dimensions as *dataMatrix*.

This operation does not support complex waves.

See Also

MatrixFilter and **ImageFilter** for filter convolutions.

Matrix Math Operations on page III-141 for more about Igor's matrix routines.

The **Loess** operation.

MatrixCorr

MatrixCorr [**/COV**][**/DEGC**] ***waveA*** [**,** ***waveB***]

The MatrixCorr operation computes the correlation or covariance or degree of correlation matrix for the input 1D wave(s).

If we denote elements of *waveA* by $\{x_i\}$ and elements of *waveB* by $\{y_i\}$ then the correlation matrix for these waves is the vector product of the form:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix} \begin{bmatrix} y_1 & y_2 & y_3 & \cdots & y_n \end{bmatrix}^* = \begin{bmatrix} x_1y_1^* & x_1y_2^* & x_1y_3^* & \cdots & x_1y_n^* \\ x_2y_1^* & x_2y_2^* & x_2y_3^* & \cdots & x_2y_n^* \\ x_3y_1^* & x_3y_2^* & x_3y_3^* & \cdots & x_3y_n^* \\ & & . & & \\ & & . & & \\ & & . & & \\ x_ny_1^* & x_ny_2^* & x_ny_3^* & \cdots & x_ny_n^* \end{bmatrix}$$

where * denotes complex conjugation. If you use the optional *waveB* then the matrix is the cross correlation matrix. *waveB* must have the same length of *waveA* but it does not have to be the same number type.

Flags

The flags are mutually exclusive; only one matrix can be generated at a time.

/COV Calculates the covariance matrix.

The covariance matrix for the same input is formed in a similar way after subtracting from each vector its mean value and then dividing the resulting matrix elements by (*n*-1) where *n* is the number of elements of *waveA*.

Results are stored in the M_Corr or M_Covar waves in the current data folder.

/DEGC Calculates the complex degree of correlation. The degree of correlation is defined by:

$$degC = \frac{M\_Covar}{\sqrt{Var(waveA) \cdot Var(waveB)}}$$

here *M_Covar* is the covariance matrix and *Var*(*wave*) is the variance of the wave.

The complex degree of correlation should satisfy: $0 \le |degC| \le 1$.

Examples

The covariance matrix calculation is equivalent to:

```
Variable N=1/(DimSize(waveA,0)-1)
Variable ma=mean(waveA,-inf,inf)
```

```
Variable mb=mean(waveB,-inf,inf)
waveA-=ma
waveB-=mb
MatrixTranspose/H waveB
MatrixMultiply waveA,waveB
M_product*=N
```

See Also

Matrix Math Operations on page III-141 for more about Igor's matrix routines.

References

Hayes, M.H., *Statistical Digital Signal Processing And Modeling*, 85 pp., John Wiley, 1996.

MatrixDet

matrixDet(*dataMatrix*)

The matrixDet function returns the determinant of *dataMatrix*. The matrix wave must be a real, square matrix or else the returned value will be NaN.

Details

The function calculates the determinant using LU decomposition. If, following the decomposition, any one of the diagonal elements is either identically zero or equal to 10^{-100}, the return value of the function will be zero.

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-141 for more about Igor's matrix routines.

MatrixDot

MatrixDot(*waveA, waveB*)

The MatrixDot function calculates the inner (scalar) product for two 1D waves. A 1D wave **A** represents a vector in the sense:

$$\mathbf{A} = \sum(\alpha_i \hat{\boldsymbol{e}}_i), \qquad \hat{\boldsymbol{e}}_i \text{ is a unit vector .}$$

Given two such waves **A** and **B**, the inner product is defined as

$$ip = \sum \alpha_i \beta_i .$$

When both *waveA* and *waveB* are complex and the result is assigned to a complex-valued number MatrixDot returns:

$$ipc = \sum \alpha_i^* \beta_i .$$

If the result is assigned to a real number, MatrixDot returns:

$$ip = \left| \sum \alpha_i^* \beta_i \right| .$$

If either *waveA* or *waveB* is complex and the result is assigned to a real-valued number, MatrixDot returns:

$$ip = \left| \sum \alpha_i \beta_i \right| .$$

When the result is assigned to a complex-valued number MatrixDot returns:

$$ipc = \sum \alpha_i \beta_i .$$

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-141 for more about Igor's matrix routines.

MatrixEigenV

MatrixEigenV [***flags***] ***matrixWave***

The MatrixEigenV operation computes the eigenvalues and eigen vectors of a square matrix using LAPACK routines. There are two mutually exclusive branches for the operation. The first is designed for a general square matrix and uses the appropriate routines depending on the data type of *matrixWave*. The second branch is designed for symmetric matrices. The first group, Flags for General Matrices, applies to general matrices only. The second group, Flags for Symmetric Matrices, applies to symmetric matrices only while the /Z flag applies to both. Both branches support single and double precision in real or complex waves.

Flags for General Matrices

/B=*balance* Determines how the input matrix should be scaled and or permuted to improve the conditioning of the eigenvalues.

balance=0 (default), 1, 2, or 3, corresponding respectively to N, P, S, or B in the LAPACK routines. Applicable only with the /X flag.

0: Do not scale or permute.

1: Permute.

2: Do diagonal scaling.

3: Scale and permute.

/L Calculates for left eigen vectors.

/O Overwrites *matrixWave*, requiring less memory.

/R Calculates for right eigen vectors.

/S=*sense* Determines which reciprocal condition numbers are calculated.

sense=0 (default), 1, 2, or 3, corresponding respectively to N, E, V, or B in the LAPACK routines. Applicable only with the /X flag.

0: None.

1: Eigenvalues only.

2: Right eigen vectors.

3: Eigenvalues and right eigen vectors.

If sense is 1 or 3 you must compute both left and right eigenvectors.

/X Uses LAPACK expert routines, which require additional parameters (see /B and /S flags). The operation creates additional waves.

The W_MatrixOpInfo wave contains in element 0 the ILO, in element 1 the IHI, and in element 2 the ABNRM from the LAPACK routines.

The wave W_MatrixRCONDE contains the reciprocal condition numbers for the eigenvalues, and the wave W_MatrixRCONDV contains the reciprocal condition number for the eigen vectors.

Flags for Symmetric Matrices

/SYM Computes the eigenvalues of an NxN symmetric matrix and stores them in the wave W_eigenValues. You must specify this flag if you want to use the special routines for symmetric matrices. The number of eigenvalues is stored in the variable V_npnts. Because W_eigenValues has N points, only the first V_npnts will contain relevant eigenvalues.

/EVEC Computes eigen vectors in addition to eigenvalues. Eigenvalues will be stored in the wave M_eigenVectors, which is of dimension NxN. The first V_npnts columns of the wave will contain the V_npnts eigen vectors corresponding to the eigenvalues in W_eigenValues. /EVEC must be preceded by /SYM.

/RNG={*method*,*low*,*high*}

method=0: Computes all the eigenvalues or eigen vectors (default).

method=1: Computes eigenvalues for *low* and *high* double precision range.

method=2: Computes eigenvalues for *low* and *high* integer indices (1 based). For example, to compute the first 3 eigenvalues use: `/RNG={2,1,3}`.

/RNG must be preceded by /SYM.

General Flags

/Z No error reporting (except for setting V_flag).

Details for General Matrices

The eigenvalues are stored in the 1D complex wave W_eigenValues. The eigen vectors are stored in the 2D wave M_R_eigenVectors or M_L_eigenVectors.

The calculated eigen vectors are normalized to unit length.

Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having the positive imaginary part first.

If the *jth* eigenvalue is real, then the corresponding eigen vector *u*(*j*)=*M*[][*j*] is the *jth* column of M_L_eigenVectors or M_R_eigenVectors. If the *jth* and (*j*+1)*th* eigenvalues form a complex conjugate pair, then *u*(*j*) = *M*[][*j*] + *i***M*[][*j*+1] and *u*(*j*+1) = *M*[][*j*] - *i***M*[][*j*+1].

The variable V_flag is set to 0 when there is no error.

Details for Symmetric Matrices

The LAPACK routines that compute the eigenvalues and eigen vectors of symmetric matrices claim to use the Relatively Robust Representation whenever possible.

See Also

Matrix Math Operations on page III-141 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

Symmetric matrices can also be decomposed using the **MatrixSchur** operation.

MatrixFilter

**`MatrixFilter [`*`flags`*`] `*`Method dataMatrix`*

The MatrixFilter operation performs one of several standard image filter type operations on the destination *dataMatrix*.

Note: The parameters below are also available in ImageFilter. See **ImageFilter** for additional parameters.

Parameters

Method selects the filter type. *Method* is one of the following names:

avg	*n*x*n* average filter.
FindEdges	3x3 edge finding filter.
gauss	*n*x*n* gaussian filter.
gradN, gradNW, gradW, gradSW, gradS, gradSE, gradE, gradNE	3x3 North, NorthWest, West, … pointing gradient filter.
median	*n*x*n* median filter. You can assign values other than the median by specifying the desired rank using the /M flag.
min	*n*x*n* minimum rank filter.
max	*n*x*n* maximum rank filter.
NanZapMedian	*n*x*n* filter that only affects data points that are NaN. Replaces them with the median of the *n*x*n* surrounding points. Unless /P is used, automatically cycles through matrix until all NaNs are gone or until *cols***rows* iterations.
point	3x3 point finding filter `8*`*`center-outer`*.
sharpen	3x3 sharpening *`filter`*`=(12*`*`center-outer`*`)/4`.
sharpenmore	3x3 sharpening *`filter`*`=(9*`*`center-outer`*`)`.
thin	Calculates binary image thinning using neighborhood maps based on the algorithm in *Graphics Gems IV*, p. 465. **Note**: The thin keyword to MatrixFilter will be removed someday. The functionality will be available — just not as a part of MatrixFilter. The /R flag does not apply to the lame duck thin keyword.

Flags

/B=*b*	Specifies value that is considered background. Used with thin. If object is black on white background, use 255. If object is white on a black background, use 0.
/M=*rank*	Assigns a pixel value other than the median when used with the median filter. Valid *rank* values are between 0 and n^2-1 (for the default median *rank*= $n^{2/2}$).

/N=*n*	For any method described above as "*n*x*n*", you can specify that the filtering kernel will be a square matrix of size *n*. In the absence of the /N flag, the default size is 3.
/P=*p*	Filter passes over the data *p* times. The default is one pass.
/R=*roiWave*	Only the data outside the region of interest will be modified. *roiWave* should be an 8-bit unsigned wave with the same dimensions as the data matrix. The exterior of the ROI is defined by zeros and the interior is any nonzero value.
/T	Applies the thining algorithm of Zhang and Suen with the thin parameter. The wave M_MatrixFilter contains the results; the input wave is not overwritten.

Details

This operation does not support complex waves.

See Also

ImageFilter operation for additional options. **Matrix Math Operations** on page III-141 for more about Igor's matrix routines. The **Loess** operation.

References

Heckbert, Paul S., (Ed.), *Graphics Gems IV*, 575 pp., Morgan Kaufmann Publishers, 1994.

Zhang, T. Y., and C. Y. Suen, A fast thinning algorithm for thinning digital patterns, *Comm. of the ACM*, *27*, 236-239, 1984.

MatrixGaussJ

MatrixGaussJ *matrixA, vectorsB*

The MatrixGaussJ operation solves matrix expression A*x=b for column vector x given matrix A and column vector b. The operation can also be used to calculate the inverse of a matrix.

Parameters

matrixA is a NxN matrix of coefficients and *vectorsB* is a NxM set of right-hand side vectors.

Details

On output, the array of solution vectors *x* is placed in M_x and the inverse of *A* is placed in M_Inverse.

If the result is a singular matrix, V_flag is set to 1 to indicate the error. All other errors result in an alert, and abort any calling procedure.

All output objects are created in the current data folder.

An error is generated if the dimensioning of the input arrays is invalid.

This routine is provided for completeness only and is not recommended for general work (use LU decomposition — see **MatrixLUD**). MatrixGaussJ does calculate the inverse matrix but that is not generally needed either.

See Also

Matrix Math Operations on page III-141 for more about Igor's matrix routines. The **MatrixLUD** operation.

MatrixInverse

MatrixInverse [*flags*] *srcWave*

The MatrixInverse operation calculates the inverse or the pseudo-inverse of a square matrix. *srcWave* may be real or complex.

MatrixInverse saves the result in the wave M_Inverse in the current data folder.

Flags

/D	Creates the wave W_W that contains eigenvalues of the singular value decomposition (SVD) for the pseudo-inverse calculation. If one or more of the eigenvalues are small, the matrix may be close to singular.
/G	Calculates only the direct inverse; does not affect calculation of pseudo-inverse. By default, it calculates the inverse of the matrix using LU decomposition. The inverse is calculated using Gauss-Jordan method. The only advantage in using Gauss-Jordan is that it is more likely to flag singular matrices than the LU method.
/O	Overwrites the source with the result.

/P Calculates the pseudo-inverse of a square matrix using the SVD algorithm. The calculated pseudo-inverse is a unique minimal solution to the problem: $min\|AX - Im\|$.

X belongs to R^{mxn}.

Example

```
Make/N=(2,2) mat0={{2,3},{1,7}}
MatrixInverse mat0                    // Creates wave M_inverse
// Check the results
MatrixOP/O mat1=M_inverse x mat0
Print mat1                            // Verify that you got the identity matrix
```

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-141 for more about Igor's matrix routines.

References

See sec. 5.5.4 of:

Golub, G.H., and C.F. Van Loan, *Matrix Computations*, 2nd ed., Johns Hopkins University Press, 1986.

MatrixLinearSolve

MatrixLinearSolve [***flags***] ***matrixA matrixB***

The MatrixLinearSolve operation solves the linear system *matrixA* *X=*matrixB* where *matrixA* is an N-by-N matrix and *matrixB* is an N-by-NRHS matrix of the same data type.

Flags

/M=*method* Determines the solution method which best suites input *matrixA*.

method=1: Uses simple LU decomposition (default). See also LAPACK documentation for SGESV, CGESV, DGESV, and ZGESV.

Creates the wave W_IPIV that contains the pivot indices that define the permutation matrix P. Row (i) if the matrix was interchanged with row ipiv(i).

method=2: If *matrixA* is band diagonal, you also have to specify /D. See also LAPACK documentation for SGBSV, CGBSV, DGBSV, and ZGBSV.

Creates the wave W_IPIV, which contains the pivot indices that define the permutation matrix P. Row (i) if the matrix was interchanged with row ipiv(*i*). Also note that if you are using the /O flag, the overwritten waves may have a different dimensions.

method=4: For tridiagonal matrix; still expecting full matrix in *matrixA*, but it will ignore the data in the elements outside the 3 diagonals. See also LAPACK documentation for SGTSV, CGTSV, DGTSV, and ZGTSV.

method=8: Symmetric/hermitian. See also LAPACK documentation for SPOSV, CPOSV, DPOSV, and ZPOSV.

method=16: Complex symmetric (complex only). See also LAPACK documentation for CSYSV and ZSYSV.

/D={*sub*,*super*} Specifies a band diagonal matrix. The subdiagonal (*sub*) and superdiagonal (*super*) size must be positive integers.

/L Uses the lower triangle of *matrixA*. /L and /U are mutually exclusive flags.

/U Uses the upper triangle of *matrixA*. /U is the default.

/O Overwrites *matrixA* and *matrixB* with the results of the operation. This will save on the amount of memory needed.

/Z No error reporting.

Details

If /O is not specified, the operation also creates the n-by-n wave M_A and the n-by-nrhs solution wave M_B.

The variable V_flag is created by the operation. If the operation completes successfully, V_flag is set to zero, otherwise it is set to the LAPACK error code.

See Also

Matrix Math Operations on page III-141 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

MatrixLinearSolveTD

MatrixLinearSolveTD [**/Z**] ***upperW, mainW, lowerW, matrixB***

The MatrixLinearSolveTD operation solves the linear system *TDMatrix**X = *matrixB*. In the matrix product on the left hand side, *TDMatrix* is a tridiagonal matrix with upper diagonal *upperW*, main diagonal *mainW*, and lower diagonal *lowerW*. It solves for vector(s) X depending on the number of columns (NRHS) in *matrixB*.

Flags

/Z No error reporting.

Details

The input waves can be single or double precision (real or complex). Results are returned in the wave M_TDLinearSolution in the current data folder. The wave *mainW* determines the size of the main diagonal (N). All other waves must match it in size with *upperW* and *mainW* containing one less point and *matrixB* consisting of N-by-NRHS elements of the same data type.

MatrixLinearSolveTD should be more efficient than MatrixLinearSolve with respect to storage requirements.

MatrixLinearSolveTD creates the variable V_flag, which is zero when it finishes successfully.

See Also

Matrix Math Operations on page III-141; the **MatrixLinearSolve** and **MatrixOp** operations.

MatrixLLS

MatrixLLS [**/O/Z/M=*method***] ***matrixA matrixB***

The MatrixLLS operation solves overdetermined or underdetermined linear systems involving MxN *matrixA*, using either QR/LQ or SV decompositions. Both *matrixA* and *matrixB* must have the same number type. Supported types are real or complex single precision and double precision numbers.

Flags

/M=*method* Specifies the decomposition method.

method=0: Decomposition is to QR or LQ (default). Creates the 2D wave M_A, which contains details of the QR/LQ factorization.

method=1: Singular value decomposition. Creates the 2D wave MA, which contains the right singular vectors stored row-wise in the first `min(m,n)` rows. Creates the 1D wave M_SV, which contains the singular values of *matrixA* arranged in decreasing order.

/O Overwrites *matrixA* with its decomposition and *matrixB* with the solution vectors. This requires less memory.

/Z No error reporting.

Details

When the /O flag is not specified, the solution vectors are stored in the wave M_B, otherwise the solution vectors are stored in *matrixB*. Let *matrixA* be *m* rows by *n* columns and *matrixB* be an *m* by NRHS (if NRHS=1 it can be omitted). If $m \geq n$, MatrixLLS solves the least squares solution to an overdetermined system: $\text{Minimize} \parallel matrixB - matrixA \cdot X \parallel$.

Here the first *n* rows of M_B contain the least squares solution vectors while the remaining rows can be squared and summed to obtain the residual sum of the squares. If you are not interested in the residual you can resize the wave using, for example:

```
Redimension/N=(n,NRHS) M_B
```

If $m<n$, MatrixLLS finds the minimum norm solution of the underdetermined system: $matrixA \cdot X = matrixB$.

In this case, the first *m* rows of M_B contain the minimum norm solution vectors while the remaining rows can be squared and summed to obtain the residual sum of the squares for the solution. If you are not interested in the residual you can resize the wave using, for example:

```
Redimension/N=(m,NRHS) M_B
```

Note: Here *matrixB* consists of one or more column vectors B corresponding to one or more solution vectors X that are computed simultaneously. If *matrixB* consists of a single column, M_B is a 2D matrix wave that contains a single solution column.

The variable V_flag is set to 0 when there is no error; otherwise it contains the LAPACK error code.

See Also

Matrix Math Operations on page III-141 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

MatrixLUBkSub

MatrixLUBkSub *matrtixL, matrixU, index, vectorB*

The MatrixLUBkSub operation provides back substitution for LU decomposition.

Details

This operation is used to solve the matrix equation Ax=b after you have performed LU decomposition (see **MatrixLUD**). Feed this routine M_Lower, M_Upper and W_LUPermutation from MatrixLUD along with your right-hand-side vector b. The solution vector x is returned as M_x. The array b can be a matrix containing a number of b vectors and the M_x will contain a corresponding set of solution vectors.

Generates an error if the dimensions of the input matrices are not appropriate.

See Also

Matrix Math Operations on page III-141 for more about Igor's matrix routines.

MatrixLUD

MatrixLUD *matrixA*

The MatrixLUD operation decomposes NxN wave, *matrixA*, into a pair of upper and lower triangular matrices.

Details

The output matrices are placed into waves named M_Upper and M_Lower. A permutation wave is returned in W_LUPermutation. This is needed for the back substitution routine, **MatrixLUBkSub**.

The variable V_flag is set to zero if the operation succeeds and to 1 if singular. The polarity of the matrix is returned in the variable V_LUPolarity.

All output objects are created in the current data folder.

See Also

Matrix Math Operations on page III-141 for more about Igor's matrix routines.

MatrixMultiply

MatrixMultiply *matrixA* [/T], *matrixB* [/T] [, *additional matrices*]

The MatrixMultiply operation calculates matrix expression *matrixA*matrixB* and puts the result in a matrix wave named M_product generated in the current data folder. The /T flag can be included to indicate that the transpose of the specified matrix should be used.

If any of the source matrices are complex, then the result is complex.

Parameters

If *matrixA* is an NxP matrix then *matrixB* must be a PxM matrix and the product is an NxM matrix. Up to 10 matrices can be specified although it is unlikely you will need more than three. The inner dimensions must be the same. Multiplication is performed from right to left.

It is legal for M_product to be one of the input matrices. Thus `MatrixMultiply A,B,C` could also be done as:

```
MatrixMultiply B,C
MatrixMultiply A,M_product
```

Details

Supports multiplication of complex matrices.

An error is generated if the dimensioning of the input arrays is invalid.

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-141 for more about Igor's matrix routines.

MatrixOp

MatrixOp [/C /FREE /NTHR=*n* /O /S] *destwave = matrixExpression*

The MatrixOp operation evaluates *matrixExpression* and creates an output wave whose name is specified by *destWave*.

matrixExpression combines waves and scalar values using the set of MatrixOp-supported operators and functions described below.

MatrixOp is faster and more readable than standard Igor waveform assignments and matrix operations.

For example, the expression

```
MatrixOp matA = Inv((Identity(vecSize) - matB x matC) x matD)
```

is equivalent to:

```
Make/O/N=(vecSize,vecSize) identityMatrix = p==q ? 1 : 0
MatrixMultiply matB,matC
identityMatrix-=M_Product
MatrixMultiply identityMatrix,matD
MatrixInverse M_Product
Rename M_Inverse, matA
```

Parameters

destWave Specifies a destination wave for the assignment expression. *destWave* is created at runtime. If it already exists, you must use the /O flag to overwrite it or the operation will return an error.

When the operation is completed, *destWave* has the dimensions and data type implied by *matrixExpression*. In particular, it may be complex if *matrixExpression* evaluates to a complex quantity. If *matrixExpression* evaluates to a scalar, destWave is a 1x1 wave.

The data type of *destWave* depends on the data types of the operands and the nature of the operations on the right-hand side of the assignment. If *matrixExpression* references integer waves only, *destWave* will be integer but all operation with a scalars convert destWave into a double precision wave. Note: even if *destWave* exists before the operation, MatrixOp changes its data type and dimensionality as implied by *matrixExpression*.

matrixExpression *matrixExpression* is an algebraic expression referencing waves, local variables, global variables and numeric constants together with the following MatrixOp operators:

MatrixOp Operator	Precedence
^h ^t	**Highest**
x	
* /	
+ -	
&& ||	**Lowest**

Operators

This section describes the behavior of the MatrixOp operators.

+ Addition between scalars, matrix addition or addition of a scalar (real or complex) to each element of a matrix.

- Subtraction of one scalar from another, matrix subtraction, or subtracting a scalar from each element of a matrix. Subtraction of a matrix from a scalar is not defined.

* Multiplication between two scalars, multiplication of a matrix by a scalar, or element-by-element multiplication of two waves of the same dimensions.

/	Division of two scalars, division of a matrix by a scalar, or element-by-element division between two waves of the same dimensions.
x	Matrix multiplication (lower case x symbol only). Operator *must* be preceded and followed by a space. Matrix multiplication requires that the number of columns in the matrix on the left side be equal to the number of rows in the matrix on the right.
.	Generalized form of a dot product. In an expression *a*.*b* it is expected that *a* and *b* have the same number of points although they may be of arbitrary numeric type. The operator returns the sum of the products of the sequential elements as if both *a* and *b* were 1D arrays.
^t	Matrix transpose. This is a postfix operator meaning that ^t appears after the name of a matrix wave.
^h	Hermitian transpose. This is a postfix operator meaning that ^t appears after the name of a matrix wave.
&&	Logical AND operator supports all real data types and results in signed byte numerical type with values of either 0 or 1. The operation acts on an element by element basis and is performed for each element of the operand waves.
||	Logical OR operator supports all real data types and results in signed byte numerical type with values of either 0 or 1. The operation acts on an element by element basis and is performed for each element of the operand waves.

Functions

These functions are available for use with MatrixOp.

Abs()	Absolute value of a real number or the magnitude of a complex number.
acos(*z*)	Arc cosine of the generic argument *z*.
asin(*z*)	Arc sine of the generic argument *z*.
asyncCorrelation(*w*)	Asynchronous spectrum correlation matrix for a real valued input matrix wave *w*. See also syncCorrelation.
atan(*z*)	Arc tangent of the generic argument *z*.
atan2(*y*,*x*)	Arc arc tangent of real *y*/*x*.
beam(*w*,*row*,*col*)	When *w* is a 3D wave the beam function returns a 1D array corresponding to the data in the beam defined by w[*row*][*col*][]. In other words, it returns a 1D array consisting of all elements in the specified row and column from all layers. See also **ImageTransform** getBeam. When *w* is a 4D wave it returns a 2D array containing w[*row*][*col*][][]. In other words, it returns a matrix consisting of all elements in the specified row and column from all layers and all chunks. The beam function belongs to a special class in that it does not operate on a layer by layer basis. It therefore does not permit compound expressions in place of any of its parameters. The beam function has the highest precedence. beam is a non-layered function which requires that *w* be a proper multi-dimensional wave and not the result of another expression.
ceil(*z*)	Smallest integer larger than *z*.
clip(*w*,*low*,*high*)	Returns the values in the wave *w* clipped between the *low* and the *high* parameters. If *w* contains NaN or INF values, they are not modified. The result retains the same number type as the input wave *w* irrespective of the range of the *low* and *high* input parameters.
ChirpZ(*data*,*A*,*W*,*M*)	Chirp Z Transform of the 1D wave data calculated for the contour defined by $z_k = AW^{-k},$ $k = 0,1,...M-1.$ Here both *A* and *W* are complex and the standard z transform for a sequence {x(n)} is defined by

$$X(z) = \sum_{k=0}^{N-1} x(n) z^{-k}.$$

The phase of the output is inverted to match the result of the ChirpZ transform on the unit circle with that of the FFT.

`ChirpZf(data,f1,f2,df)`

Chirp Z Transform except that the transform parameters are specified by real-valued starting frequency *f1,* end frequency *f2* and frequency resolution *df*. The transform is confined to the unit circle because both *A* and *W* have unit magnitude.

`chol(w)` Returns the Cholesky decomposition U of a positive definite symmetric matrix w such that w=U^t x U. Note that only the upper triangle of w is actually used in the computation.

`col(w,c)` Returns column *c* from matrix wave *w*.

`Conj(matrixWave)` Ccomplex conjugate of the input expression.

`convolve(w1,w2,opt)`

Convolution of *w1* with *w2* subject to options opt. The dimensions of the result are determined by the largest dimensions of *w1* and *w2* with the number of rows padded (if necessary) so that they are even. Supported options include *opt*=0 for circular convolution and *opt*=4 for acausal convolution.

For fast 2D convolutions where where *w1* is an image and *w2* is a square kernel of the same numeric type, you can use *opt*=-1 or *opt*=-2. When *opt*=-1 the convolution at the boundaries is evaluated using zero padding. When *opt*=-2 the padding is a reflection of *w1* about the boundaries. When working with integer waves the kernel is internally normalized by the sum of its elements. The kernel for floating point waves remain unchanged.

To convolve an image (in *w1*) with a smaller point spread function (in *w2*) you can use opt=-1 if you want to pad the image boundaries with zeros or opt=-2 if you want to pad the boundaries by reflecting image values about each boundary. Note that the negative options are designed for a very optimized convolution calculation which requires that *w1* and *w2* have the same numeric type. If the size of the point spread function is larger than about 13x13 it may become more efficient to compute the convolution using the positive options.

`correlate(w1,w2,opt)`

Correlation of *w1* with *w2* subject to options *opt*. The dimensions of the result are determined by the largest dimensions of *w1* and *w2* with the number of rows padded (if necessary) so that they are even. Supported options include *opt*=0 for circular correlation and *opt*=4 for acausal correlation.

`cos(z)` Cosine of the generic argument *z*.

`crossCovar(w1,w2,opt)`

Returns the cross-covariance for 1D waves w1 and w2. The options parameter opt can be set to 0 for the raw cross-covariance or to 1 if you want the results to be normalized to 1 at zero offset. If w1 has N rows and w2 has M rows then the returned vector is of length N+M-1. The cross-covariance is computed by subtracting the mean of each input followed by correlation and optional normalization. See also **Correlate** with the /NODC flag.

`Det(matrixWave)` Returns a scalar corresponding to the determinant of *matrixWave,* which must be real.

`Diagonal(wave1D)` Creates a square matrix that has the same number of rows as *wave1D*. All entries are zero except for the diagonal elements for which the entries are taken from the first column in *wave1D*. Use DiagRC if the input is not an existing wave (such as a result from another function).

`DiagRC(waveA,rows,cols)`

2D matrix of dimensions *rows* by *cols*. All matrix elements are set to zero except those of the diagonal which are filled sequentially from elements of *waveA*. The dimensionality of *waveA* is unimportant. If the total number of elements in *waveA*

	is less than the number of elements on the diagonal then all elements will be used and the remaining diagonal elements will be set to zero.
`e`	Returns the base of the natural logarithm.
`equal(a,b)`	Returns unsigned byte result with 1 for equality and zero otherwise. The dimensionality of the result matches the dimensionality of the largest parameter. Either or both *a* or *b* can be constants (i.e., one row by one column). If *a* and *b* are not constants, they must have the same dimensions. Both parameters can be either real or complex. A comparison of a real with a complex parameter returns zero.
`exp(z)`	Exponential function for a generic argument *z*, which can be real or complex, scalar or a matrix.
`FFT(inWave,options)`	FFT of *inWave* subject to the *options* field. *inWave* must have an even number of rows. The options field contains a binary field flag. Set the second bit to 1 if you want to disable the zero centering (see /Z flag in the **FFT** operation). Other bits are reserved. MatrixOp does not support wave scaling and therefore it does not produce the same wave scaling changes as the **FFT** operation.
`floor(z)`	Largest integer smaller than *z*.
`Frobenius(matWave)`	Returns the Frobenius norm of a matrix defined as the square root of the sum of the squared absolute values of all elements.
`greater(a,b)`	Returns an unsigned byte for the truth of *a* > *b*. Both *a* and *b* must be real but one or both can be constants (see `equal()` above). The dimensionality of the result matches the dimensionality of the largest parameter.
`IFFT(inWave,options)`	IFFT of *inWave* subject to the *options* field. The *options* field corresponds to the /C and /Z flags of the **IFFT** operation. Set the first bit to 1 if you want to allow real-valued output (in the case of a complex input of 2n+1 rows. Set the second bit to 1 if you want to disable the zero centering. Other bits are reserved. Note that MatrixOp does not support wave scaling and therefore it does not produce the same wave scaling changes as the **IFFT** operation.
`Identity(n,m)` `Identity(n)`	Creates a computational object that is an identity matrix. If you use a single argument *n*, the identity created is an *n*x*n* square matrix with 1's for diagonal elements (the remaining elements are set to zero). If you use both arguments, the function creates an *n*x*m* zero matrix and fills its diagonal elements with 1's. Note that the identity is created at runtime and persists only for the purpose of the specific operation.
`imag(inWave)`	Imaginary part of *inWave*.
`Inv(matrixWave)`	Inverse of a square *matrixWave*.
`log(z)`	Log (i.e., log base 10) of a generic argument *z* which can be real or complex, scalar or a matrix.
`ln(z)`	Natural logarithm of a generic argument *z* which can be real or complex, scalar or a matrix.
`mag(inWave)`	Returns a real valued wave containing the magnitude of each element of *inWave*. This is equivalent to the `Abs()` function.
`magSqr(inWave)`	Returns a real value wave containing the square of a real *inWave* or the squared magnitude of complex *inWave*.
`maxVal(w)`	Returns the maximum value of the wave *w*. If *w* is complex it returns the maximum magnitude of *w*. Note that this function does not support NaN values.
`mean(w)`	Returns the mean value of the wave *w*.
`minVal(w)`	Returns the minimum value of the wave *w*. If *w* is complex the function returns the minimum magnitude of *w*. Note that this function does not support NaN values.
`Normalize(inWave)`	Normalized version of a vector or a matrix. Normalization is such that the returned wave should have a unity magnitude except if all entries are zero, in which case output is unchanged.
`NormalizeCols(w)`	Divides each column of the real wave *w* with the square root of the sum of the squares of all elements of the column.

`NormalizeRows(w)` Divides each row of the real wave *w* with the square root of the sum of the squares of all the elements in that row.

`numCols(w)` Returns the number of columns in the wave *w*.

`numPoints(w)` Returns the number of points in a layer of the wave *w*.

`numRows(w)` Returns the number of rows in the wave *w*.

`numType(z)` Number type for the element passed to this function.

`p2Rect(inWave)` Performs the equivalent of the **p2rect** function on each element of *inWave*, i.e., each complex number of *inWave* is converted from polar to rectangular representation.

`phase(inWave)` Returns a real valued wave containing the phase of each element of *inWave* calculated using `phase=atan2(y,x)`.

`Pi` Returns π.

`powC(z1,z2)` Complex valued $z1^{z2}$ where *z1* and *z2* can be real or complex.

`powR(x,y)` Returns *x*^*y* for real *x* and *y*.

`r2Polar(inWave)` Performs the equivalent of the **r2polar** function on each element of *inWave*, i.e., each complex number (x+iy) is converted into the polar represenation r,theta with x+iy=r*exp(i*theta)

`real(inWave)` Real part of *inWave*.

`rec(inWave)` Reciprocal of each element in *inWave*.

`Replace(w,findVal,replacementVal)`

Replace in wave *w* every occurance of *findVal* with *replacementVal*. The wave *w* retains its dimensionality and number type. *replacementVal* is converted to the same number type as *w* which may cause truncation.

`ReplaceNaNs(w,replacementVal)`

Replaces every occurance of NaN in the wave *w* with *replacementVal*. The wave *w* retains its dimensionality. *replacementVal* is converted to the same number type as *w* which may cause truncation.

`rotateRows(w,nr)` Rotates the rows of a 2D wave *w* so that the last *nr* rows are moved to rows [0,*nr* -1] of the data. If *nr* is negative the first abs(*nr*) rows are moved to rows [n-1-*nr* , n-1]. Here n is the total number of rows. It is an error to pass NaN for *nr*. If *nr* is greater than the number of rows then the effective rotation is mod(*nr* ,actualRows).

`rotateCols(w,nc)` Rotates the columns of a 2D wave *w* so that the last *nc* columns are moved to columns [0,*nc* -1] of the data. If *nc* is negative the first abs(*nc*) columns are moved to columns [n-1-*nc* ,n-1]. Here n is the total number of columns. It is an error to pass NaN for *nc* . If *nc* is greater than the number of columns then the effective rotation is mod(*nc* ,actualCols).

`round(z)` Rounds *z* to the nearest integer. The rounding method is “away from zero”.

`row(w,r)` Returns row *r* from matrix wave *w*. The returned row is a (1xC) wave where C is the number of columns in *w*. To convert it to a 1D wave use `Redimension/N=(C)`. See also ImageTransform getRow.

`scale(w,low,high)` Returns the values in the wave *w* scaled between the *low* and the *high* parameters. If *w* contains NaN or INF values, they are not modified. The result retains the same number type as that of *w* irrespective of the range of the *low* and *high* input parameters.

`sgn(w)` Returns the sign of each element in *w*. It returns -1 for negative numbers and 1 otherwise. It does not accept complex numbers.

`shiftVector(w, n, val)`

Shifts the element of a 1D row-vector *w* by *n* elements and fills the displaced elements with *val*, which must match the data type of *w* and should be expressed as `cmplx(a,b)` for complex *w*.

`sin(z)` Sine of the generic argument *z*.

`sqrt(z)` Square root of the generic argument *z*.

`SubtractMean(w,opt)`

Computes the mean of the real wave *w* and returns the values of the wave minus

the mean value (*opt*=0). Computes the mean of each column and subtracts it from that column (*opt*=1). Subtracts the mean of each row from row values (*opt*=2).

`sum(z)` Returns the sum of all the elements in expression *z*.

`sumBeams(w)` Returns an nxm matrix containing the sum over all layers of all the beams of the 3D wave *w*:

$$out_{ij} = \sum_{k=0}^{nLayers-1} w_{ijk}.$$

A beam is a 1D array in the Z-direction.

sumBeams is a non-layered function which requires that *w* be a proper 3D wave and not the result of another expression.

`sumCols(w)` Returns an 1xm matrix containing the sums of the m columns in the nxm input wave *w*:

$$out_{j} = \sum_{i=0}^{nRows-1} w_{ij}.$$

`sumRows(w)` Returns an nx1 matrix containing the sums of the n rows in the nxm input wave *w*:

$$out_{i} = \sum_{j=0}^{nCols-1} w_{ij}.$$

`sumSqr(inWave)` Sum of the squares of all elements in *inWave*.

`syncCorrelation(w)` Synchronous spectrum correlation matrix for a real valued input matrix wave *w*. See also `asyncCorrelation`.

The correlation matrix is computed by subtracting from each column of *w* its mean value, multiplying the resulting matrix by its transpose, and finally dividing all entries by (nrows-1) where nrows is the number of rows in *w*.

`tan(z)` Tangent of the generic argument *z*.

`Trace(matrixWave)` Returns a real or complex scalar which is the sum of the diagonal elements of *matrixWave*. If *matrixWave* is not a square matrix, the sum is over the elements for which the row and column indices are the same.

`transposeVol(w,mode)`

For 3D wave *w*, transposeVol returns a transposed 3D wave depending on the value of the *mode* parameter:

mode=1:	output=w[p][r][q]
mode=2:	output=w[r][p][q]
mode=3:	output=w[r][q][p]
mode=4:	output=w[q][r][p]
mode=5:	output=w[q][p][r]

transposeVol is a non-layered function which requires that *w* be a proper 3D wave and not the result of another expression.

`TriDiag(w1,w2,w3)` Returns a tri-diagonal matrix where *w1* is the upper diagonal, *w2* the main diagonal and *w3* the lower diagonal. If *w2* has *n* points than *w1* and *w3* are expected to have *n*-1 points. The waves can be of any numeric type and the returned wave has a numeric type that accommodates the input.

`varCols(w)` Returns a 1xcols wave where each element contains the variance of the corresponding column in *w*.

`waveIndexSet(w1,w2,w3)`

Returns a matrix of the same dimensions as *w1* with values taken either from *w1* or from *w3* depending on values in *w2* using:

$$out[i][j] = \begin{cases} w1[i][j] & if\ w2[i][j] < 0 \\ w3[w2[i][j]] & otherwise \end{cases}.$$

w1 and *w2* must have the same number of rows and columns. *w1* and *w3* must match in number type. *w2* cannot be unsigned.

Values from *w2* are used as point number indices into *w3* which is treated like a 1D wave regardless of its actual dimensionality.

An index value from *w2* is out-of-bounds if it is greater than or equal to the number of points in *w3*. In this case, the output value is taken from *w1* as if the index value were negative.

`waveMap(w1,w2)` Returns an array of the same dimensions as *w2* containing the values w1[*w2*[*i*][*j*]]. The data type of the output is the same as that of *w1*. Values of *w2* are taken as 1D integer indices into the *w1* array. See also **IndexSort**.

Wave Arguments

MatrixOp was designed to work with 2D matrices but it also works with a variety of other formats such as the following "nonmatrix" formats:

`wave1d[a]` Where *a* is a constant index or a local variable. This will be converted into a constant equal to the corresponding wave element. The index *a* will be clipped to the valid range for wave1d.

`wave2d[a][b]` Where *a* and *b* are constants or local variables. Both indices will be clipped to the valid range in the corresponding dimension.

`wave3d[a][b][c]` Where *a, b,* and *c* are constants or local variables. All three indices will be clipped to the valid range in the corresponding dimension.

`wave3d[][][a]` Layer *a* from the 3D wave will be treated as a 2D matrix. The first and second bracket pairs must be empty. *a* must be a constant that will be clipped to the valid range of layers.

`wave3d[][][a,b]` *matrixExpression* is evaluated for all layers between layer *a* and layer *b*. The result is a 3D wave.

`wave3d[][][a,b,c]` *matrixExpression* is evaluated for layers starting with layer *a* and increasing the layer number up to layer *b* using increments *c*. *a, b,* and *c* must be constants. Layers are clipped to the valid range and *c* must be a positive integer.

MatrixOp does not support expressions that include the same 3D wave on both sides of the expressions. For example, the expression

```
MatrixOp/O wave3D=wave3D+A
```

is not permitted. You can also use waves of any dimensions in the standard numerical functions. For example:

```
Make/O/N=128 wave1d=x
MatrixOp/O outWave=powR(wave1d,2)
```

Flags

/C Provides a complex wave reference for *destWave*. If omitted, MatrixOp creates a real wave reference for *destWave*. The wave reference allows you to refer to the output wave in a subsequent statement of a user-defined function.

/FREE Creates *destWave* as a free wave. Allowed only in functions and only if a simple name or structure field is specified.

Requires Igor Pro 6.1 or later. For advanced programmers only.

See **Free Waves** on page IV-71 for more discussion.

/NTHR=*n* Sets the number of threads used to compute the results for 3D waves. Each thread computes the results for a single layer of the input. By default *n*=0 and the calculations are preformed by the main thread. For *n*=1 the operation uses as many threads as the number of processors on your computer. Higher values of *n* may or may not improve performance.

/O Overwrites *destWave* if it already exists.

/S Preserves the dimension scaling, units and wave note of a pre-existing wave that appears on the lefthand side of the MatrixOp expression.

Details

MatrixOp does not support wave scaling, it does not support the standard p, q, r, s, or x, y, z, t symbols supported in a regular waveform assignment statement. It does not support operator combinations of the form +=.

The operation creates a 1D or 2D destination wave, depending on *matrixExpression*. *matrixExpression* can reference 1D, 2D and 3D waves.

You can use any combination of data types for operands. In particular, you can mix real and complex types in *matrixExpression* without having to worry about /C declarations for complex waves. MatrixOp determines the appropriate output data type at runtime.

Operator predence is indicated in the table above. When in doubt, use parentheses. When operators have the same precedence, execution associativity is from right to left. This means that a * b / c is equivalent to a * (b / c).

When using MatrixOp, simple matrix operations execute approximately 20-30 times faster than normal Igor syntax. In most situations MatrixOp is faster than **FastOp**. However, for small waves, the extra overhead in calling MatrixOp may make it slower than FastOp or a regular wave assignment statement. It is important to note that usually you will find the best performance for single or double precision data types, so unless there is a severe memory restriction, it is advantageous to convert integer waves to single precision floating point before performing any operations.

Examples

The following matrices are used in the examples:

```
Make/O/N=(3,3) r1=x, r2=y
```

Matrix addition and matrix multiplication by a scalar:

```
MatrixOp/O outWave = r1+r2-3*r1
```

Using the matrix `Identity` function:

```
MatrixOp/O outWave = Identity(3) x r1
```

Create a persisting identity matrix for another calculation:

```
MatrixOp/O id4 = Identity(4)
```

Using the `Trace` function:

```
MatrixOp/O outWave = (Trace(r1)*identity(3) x r1)-3*r1
```

Using matrix inverse function `Inv()` with matrix multiplication:

```
MatrixOp/O outWave = Inv(r2) x r2
```

Using the determinant function `Det()`:

```
MatrixOp/O outWave = Det(r1)+Det(r2)
```

Using the Transpose postfix operator:

```
MatrixOp/O outWave = r1^t+(r2-r1)^t-r2^t
```

Using a mix of real and complex data:

```
Variable/C complexVar = cmplx(1,2)
MatrixOp/O outWave = complexVar*r2 - Cmplx(2,4)*r1
```

Hermitian transpose operator:

```
MatrixOp/O outWave = Trace(complexVar*r2)^h -Trace(cmplx(2,4)*r1)^h
```

In-place operation and conversion to complex:

```
MatrixOp/O r1 = r1*cmplx(1,2)
```

Image filtering using 2D spatial filter filterWave:

```
MatrixOp/O filteredImage=IFFT(FFT(srcImage,2)*filterWave,3)
```

Positive shift:

```
Make/O w={0,1,2,3,4,5,6}
MatrixOp/O w=shiftVector(w,2,77)
Print w
// w[0]= {77,77,0,1,2,3,4}
```

Negative shift:

```
Make/O w={0,1,2,3,4,5,6}
MatrixOp/O w=shiftVector(w,(-2),77)
Print w
// w[0]= {2,3,4,5,6,77,77}
```

References

syncCorrelation and asyncCorrelation:

Noda, I., Determination of Two-Dimensional Correlation Spectra Using the Hilbert Transform, *Applied Spectroscopy 54*, 994-999, 2000.

ChirpZ:

Rabiner, L.R., and B. Gold, *The Theory and Application of Digital Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1975.

See Also

Matrix Math Operations on page III-141 for more about Igor's matrix routines.

The **FastOp** operation and **p2rect** and **r2polar** functions.

MatrixRank

matrixRank(*matrixWaveA* [, *conditionNumberA*])

The matrixRank function returns the rank of *matrixWaveA* subject to the specified condition number.

The matrix is not considered to have full rank if its condition number exceeds the specified *conditionNumberA*.

If the optional parameter *conditionNumberA* is not specified, Igor Pro uses the value 10^{20}.

matrixRank supports real and complex single precision and double precision numeric wave data types.

The value of *conditionNumberA* should be large enough but taking into account the accuracy of the numerical representation given the numeric data type.

If there are any errors the function returns NaN.

See Also

Matrix Math Operations on page III-141 for more about Igor's matrix routines.

MatrixSchur

MatrixSchur [**/Z**] ***srcMatrix***

The MatrixSchur operation computes for an N-by-N real nonsymmetric *srcMatrix*, the eigenvalues, the real Schur form T, and, the matrix of Schur vectors Z which gives the Schur factorization A = Z*T*(Z**T).

Flags

/Z	No error reporting.

Details

The operation creates:

M_A	Matrix containing the Schur form T.
M_V	Matrix containing the orthogonal matrix Z of the Schur vectors.
W_REigenValues W_IEigenValues	Waves containing the real and imaginary parts of the eigenvalues when *srcMatrix* is a real wave. If *srcMatrix* is complex, the eigenvalues are stored in W_eigenValues.

The variable V_flag is set to 0 when there is no error; otherwise it contains the LAPACK error code.

Examples

You can test this operation for an N-by-N source matrix srcWave:

```
MatrixSchur srcWave
Duplicate M_V M_VT            // for the transpose
MatrixTranspose M_VT
MatrixMultiply M_V,M_A,M_VT
Edit M_Product                // compare the result with srcWave
```

See Also

Matrix Math Operations on page III-141 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

MatrixSolve

MatrixSolve *method*, *matrixA*, *vectorB*

The MatrixSolve operation was superseded by MatrixLLS and is included for backward compatibility only.

Used to solve matrix equation Ax=b using the method of your choice. Choices for *method* are:

method	Solution Method
GJ	Gauss Jordan.
LU	LU decomposition.
SV	Singular Value decomposition.

Details

The array b can be a matrix containing a number of b vectors and the output matrix M_x will contain a corresponding set of solution vectors.

V_flag is set to zero if success, 1 if singular matrix using GJ or LU and 1 if SV fails to converge.

For normal problems you should use LU. GJ is provided only for completeness and has no practical use.

When using SV, singular values smaller than 10^{-6} times the largest singular value are set to zero before back substitution.

Generates an error if the dimensions of the input matrices are not appropriate.

See Also

The **MatrixLLS** operation. **Matrix Math Operations** on page III-141 for more about Igor's matrix routines.

MatrixSVBkSub

MatrixSVBkSub *matrixU*, *vectorW*, *matrixV*, *vectorB*

The MatrixSVBkSub operation does back substitution for SV decomposition.

Details

Used to solve matrix equation Ax=b after you have performed an SV decomposition.

Feed this routine the M_U, W_W and M_V waves from **MatrixSVD** along with your right-hand-side vector b. The solution vector x is returned as M_x.

The array b can be a matrix containing a number of b vectors and the M_x will contain a corresponding set of solution vectors.

Generates an error if the dimensions of the input matrices are not appropriate.

See Also

Matrix Math Operations on page III-141 for more about Igor's matrix routines.

MatrixSVD

MatrixSVD [***flags***] ***matrixWave***

The Matrix SVD operation uses the singular value decomposition algorithm to decompose an MxN matrixWave into a product of three matrices. The default decomposition is into MxM wave M_U, min(M,N) wave W_W and NxN wave M_VT.

Flags

/B Assures backwards compatibility with the old version (Igor Pro 3) of MatrixSVD. This option creates the variable V_SVConditionNumber and the M_V matrix. None of the other flags will have an effect when /B is specified. The decomposition is such that:

U*W*V^T = *matrixWave*

U: MxN column-orthonormal matrix.

W: NxN diagonal matrix of positive singular values.

V: NxN orthonormal matrix.

/O Overwrites *matrixWave* with the first columns of U. Use this flag to if you need to conserve memory. See also related settings of /U and /V.

/U =*UMatrixOptions*

UMatrixOptions can have the following values:

0: All cols of U are returned in the wave M_U (default).

1: The first min(m,n) columns of U are returned in the wave M_U.

2: The first min(m,n) columns of U overwrite matrixWave (/O must be specified).

3: No columns of U are computed.

/V=*VMatrixOptions*

VMatrixOptions can have the following values:

0: All rows of V^T are returned in the wave M_VT (default).

1: The first min(m,n) rows of V^T are returned in the wave M_VT.

2: The first min(m,n) rows of V^T are overwritten on *matrixWave* (/O must be specified).

3: No rows of V^T are computed.

/Z No error reporting.

Details

The singular value decomposition is computed using LAPACK routines. The diagonal elements of matrix W are returned as a 1D wave named W_W. If /B is used W_W will have N elements. Otherwise the number of elements in W_W is min(M,N).

The matrix V is returned in a matrix wave named M_V if /B is used otherwise the transpose V^T is returned in the wave M_VT.

If /B is used, the variable V_SVConditionNumber is set to the condition number of the input matrix A. The condition number is the ratio of the largest singular value to the smallest.

All output objects are created in the current data folder.

The variable V_flag is set to zero if the operation succeeds. It is set to 1 if the algorithm fails to converge.

Example

```
Make/O/D/N=(10,20) A=gnoise(10)
MatrixSVD A
MatrixOp/O diff=abs(A-(M_U x DiagRC(W_W,10,20) x M_VT))
Print sum(diff,-inf,inf)
```

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-141 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

MatrixTrace

matrixTrace(*dataMatrix*)

The matrixTrace function calculates the trace (sum of diagonal elements) of a square matrix. *dataMatrix* can be of any numeric data type.

If the matrix is complex, it returns the sum of the magnitudes of the diagonal elements.

See Also

Matrix Math Operations on page III-141 for more about Igor's matrix routines.

MatrixTranspose

MatrixTranspose [**/H**] ***matrix***

The MatrixTranspose operation Swaps rows and columns in *matrix*.

Does not take complex conjugate if data are complex. You can do that as a follow-on step.

Swaps row and column labels, units and scaling.

This works with text as well as numeric waves. If the matrix has zero data points, it just swaps the row and column scaling.

Flags

/H Computes the Hermitian conjugate of a complex wave.

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-141 for more about Igor's matrix routines.

max

max(*num1, num2*)

The max function returns the more positive of *num1* and *num2*.

See Also

The **min** function.

mean

mean(*waveName* [, *x1, x2*])

The mean function returns the arithmetic mean of the wave for points from x=*x1* to x=*x2*.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

The wave values from *x1* to *x2* are summed, and the result divided by the number of points in the range.

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x2*. To use point indexing, replace *x1* with `pnt2x(waveName,pointNumber1)`, and a similar expression for *x2*.

If the points nearest to *x1* or *x2* are not within the point range of 0 to numpnts(*waveName*)-1, mean limits them to the nearest of point 0 or point numpnts(*waveName*)-1.

If any values in the point range are NaN, mean returns NaN.

Unlike the area function, reversing the order of *x1* and *x2* does *not* change the sign of the returned value.

Examples

```
Make/O/N=100 data;SetScale/I x 0,Pi,data
data=sin(x)
Print mean(data,0,Pi)          // the entire point range, and no more
Print mean(data)               // same as -infinity to +infinity
Print mean(data,Inf,-Inf)      // +infinity to -infinity
```

The following is printed to the history area:

```
•Print mean(data,0,Pi)         // the entire point range, and no more
  0.630201
•Print mean(data)              // same as -infinity to +infinity
  0.630201
•Print mean(data,Inf,-Inf)     // +infinity to -infinity
  0.630201
```

See Also

Variance, **WaveStats**

The figure "Comparison of area, faverage and mean functions over interval (12.75,13.32)", in the **Details** section of the **faverage** function.

MeasureStyledText

MeasureStyledText [/W=*winName* /A=*axisName* /F=*fontName* /SIZE=*fontSize* /STYL=*fontStyle*] *styledTextStr*

The MeasureStyledText operation takes as input a string optionally containing style codes such as are used in graph annotations. It sets various variables with information about the dimensions of the string.

Flags

/W=*winName*	Takes default text information from the window *winName*.
/A=*axisName*	Takes default text information from the axis named *axisName*. If the /W flag is used, the axis should be in that window (the window should also be a graph). If the /W flag is not used, MeasureStyledText looks at the top graph window.
/F=*fontNameStr*	The name of the default font.
/SIZE=*size*	Sets default font size.
/STYL=*fontStyle*	Sets default font style: bit 0: Bold. bit 1: Italic. bit 2: Underline. bit 3: Outline (Macintosh only). bit 4: Shadow (Macintosh only).

Parameters

styledTextStr	The text to be measured. The text can contain formatting codes such as those used in a graph annotation (see **TextBox** on page V-695). Any such codes, just as in an annotation, will override defaults.

Details

In the absence of formatting codes within the text that set the font, font size and font style, some mechanism must be provided that sets them. The /W flag tells MeasureStyledText to look at a particular window and get defaults from that window.

The /A flag specifies that the defaults should come from a graph's axis of the given name. MeasureStyledText will look for the axis in the window named by /W, or in the top graph window in the absence of the /W flag.

The /F, /SIZE and /STYL flags set defaults that override any defaults from a window or axis. If you don't use any flags, the defaults are Igor's overall defaults.

Variables

The MeasureStyledText operation returns information in the following variables:

V_width	The width in points of the text.
V_height	The height in points of the text.

See Also

TextBox on page V-695 for a list of text formatting codes.

Menu

Menu *menuNameStr* [, hideable, dynamic, contextualmenu]

The Menu keyword introduces a menu definition. You can use this to create your own menu, or to add items to a built-in Igor menu.

Use the optional *hideable* keyword to make the menu hideable using **HideIgorMenus**.

Use the optional dynamic keyword to cause Igor to re-evaluate the menu definition when the menu is used. This is helpful when the menu item text is provided by a user-defined function. See **Dynamic Menu Items** on page IV-109.

Use the optional contextualmenu keyword for menus invoked by **PopupContextualMenu**/N.

See Chapter IV-5, **User-Defined Menus** for further information.

min

min(*num1, num2*)

The min function returns the more negative of *num1* and *num2*.

See Also

The **max** function.

mod

mod(*num, div*)

The mod function returns the remainder when *num* is divided by *div*.

The mod function will give unexpected results when *num* or *div* is fractional. For example:

```
Print/D mod(0.3,0.05)
```

prints

```
2.77555756156289e-17
```

This is because the numbers 0.3 and 0.05 can not be precisely represented by a finite precision binary number.

modDate

modDate(*waveName*)

The modDate function returns the modification date/time of the wave.

Details

The returned value is a double precision Igor date/time value, which is the number of seconds from 1/1/1904. It returns zero for waves created by versions of Igor prior to 1.2, for which no modification date/time is available.

See Also

The **Secs2Date** and **Secs2Time** functions.

Modify

Modify

We recommend that you use **ModifyGraph**, **ModifyTable**, **ModifyLayout**, or **ModifyPanel** rather than Modify. When interpreting a command, Igor treats the Modify operation as ModifyGraph, ModifyTable, ModifyLayout or ModifyPanel, depending on the target window. This does not work when executing a user-defined function.

ModifyContour

ModifyContour [**/W=*winName***] ***contourInstanceName, keyword=value***
[, ***keyword=value***…]

ModifyContour modifies the number, Z value and appearance of the contour level traces associated with *contourInstanceName*.

contourInstanceName is a name derived from the name of the wave that provides the Z data values. It is usually just the name of the wave, but may have #1, #2, etc. added to it in the unlikely event that the same Z wave is contoured more than once in the same graph.

contourInstanceName can also take the form of a null name and instance number to affect the instanceth contour plot. That is,

```
ModifyContour ''#1
```

modifies the appearance of the second contour plot in the top graph, no matter what the contour plot names are. Note: Two single quotes, not a double quote.

The number of contour level traces and their Z values are set by the *autoLevels*, *manLevels,* and *moreLevels* keywords, described in the **Parameters** section. Normally, you will use either *autoLevels* or *manLevels,* and then optionally generate additional levels using *moreLevels*.

Parameters

Each parameter has the syntax

```
keyword = value
```

and is applied to all of the contour level traces associated with *contourInstanceName*.

To modify an individual contour level trace, use **ModifyGraph**.

autoLevels= {*minLevel*, *maxLevel*, *numLevels*}

Controls automatic determination of contour levels.

If *numLevels* is zero, no automatic levels are generated. If it is nonzero, it specifies the desired number of automatic contour levels.

minLevel specifies the minimum contour level and *maxLevel* specifies the maximum contour level. The values that you specify are an approximate guide for Igor to use in determining the actual levels.

However, if *minLevel* or *maxLevel* is * (asterisk symbol), Igor uses the minimum or maximum value of the Z data for the corresponding contour level.

Using the autoLevels keyword cancels the effect of any previous autoLevels or manLevels keyword.

When you first append a contour plot to a graph, default contour levels are generated by the default setting `autoLevels={*,*,11}`.

boundary=*b*

Draws an outline around the XY domain of the contour data. For a matrix, this draws a rectangle showing the minimum and maximum X and Y values. For XYZ triples, the outline is a polygon enclosing the outside edges of the Delaunay Triangulation. Like the contour lines, the boundary is drawn using a graph trace, whose name is usually something like "`contourInstanceName = boundary`".

b=0: Hides the data boundary (default).

b=1: Shows the data boundary.

cIndexLines= *matrixWave*

Sets the Z value mapping mode such that contour line colors are determined by doing a lookup in the specified matrix wave.

matrixWave is a 3 column wave that contains red, green, and blue values from 0 to 65535. (The matrix can actually have more than 3 columns. Any extra columns are ignored.)

The color for a the contour line at Z=*z* is determined by finding the RGB values in the row of *matrixWave* whose scaled X index is *z*. In other words, the red value is *matrixWave*(*z*)[0], the green value is *matrixWave*(*z*)[1] and the blue value is *matrixWave*(*z*)[2].

If *matrixWave* has default X scaling, where the scaled X index equals the point number, then row 0 contains the color for Z=0, row 1 contains the color for Z=1, etc.

If you use cIndexLines, you must not use ctabLines or rgbLines in the same command.

ctabLines={*zMin*, *zMax*, *ctName*, *mode*}

Sets the Z value mapping mode such that contour line colors are determined by doing a lookup in the specified color table. *zMin* is mapped to the first color in the color table. *zMax* is mapped to the last color. Z values between the min and max are linearly mapped to the colors between the first and last in the color table.

You can enter * (an asterisk) for *zMin* and *zMax*, which uses the minimum and maximum Z values of the data. The default is {*,*,Rainbow}.

Set parameter *mode* to 1 to reverse the color table; zero or missing does not reverse the color table.

ctName can be any color table name returned by the **CTabList** function, such as Grays or Rainbow. Also see **Color Tables** on page II-349.

If you use ctabLines, you must not use cIndexLines or rgbLines in the same command.

equalVoronoiDistances=*e*

Normally the x range and y range of the data are each normalized to a 0-1 range separately to generate the Voronoi triangulation. Voronoi triangulation is a distance-based ("nearest neighbor") algorithm that may benefit from scaling the X and Y ranges

together to avoid numerical problems that occur when the triangles become very thin because of widely differing x and y ranges.

e=0: The x and y ranges are scaled individually to the 0-1 range (default).

e =1: The x and y ranges are scaled so that that maximum range of x or y is scaled to the 0-1 range, and the other is proportionally smaller. For example, if yMax-yMin = 1000 and xMax-xMin = 5, then the y range is scaled to 0-1 and the y range is scaled to 5/1000 = 0 - 0.005.

The equalVoronoiDistances keyword is allowed only for XYZ contour plots.

interpolate=*i*

XYZ contours can be interpolated to increase the apparent resolution, resulting in smoother contour lines.

This keyword is allowed only for XYZ contours, created by **AppendXYZContour**.

i=0: Linear interpolation (default). This means that only the original Delaunay triangulation generates contour lines.

i=1: Four times the resolution generates a smoother set of contour lines. As expected, this takes longer than Linear interpolation.

i=2: Sixteen times the resolution generates a much smoother set of contour lines. This is rather slow….

The interpolate parameter can be up to 8. Each time you increase *i* by one, you quadruple the apparent resolution and get smoother contour lines at the expense of computation time. Values of *i* greater than two are impractical because of the computation time required.

labelBkg=(*r*, *g*, *b*)

Sets the background color for all contour level labels to the specified color. *r*, *g*, and *b* are values from 0 to 65535.

labelBkg=*b*

b=0: Uses each label's individual background color, as set via the Modify Annotation dialog.

b=1: Makes all contour level labels transparent.

b=2: Uses the plot area background color as the label background color (default).

b=3: Uses the window background color as the label background color.

labelDigits=*d*

d is the number of digits after the decimal point when using labelFormat=3 or labelFormat=5.

labelFont=*fontName*

Default; specifies the font to use for contour level labels. If you pass "" for *fontName*, it will use the graph font (set via the Modify Graph dialog) for contour labels.

labelFormat=*l*

Controls the formatting of contour labels. See the **printf** operation for a discussion of formatting.

l=0: Uses general format that is suitable for most data. This is equivalent to "%<sigDigits>g".

l=1: Uses integer format, equivalent to "%<sigDigits>d".

l=3: Uses fixed point format, equivalent to "%<decimalDigits>f".

l=5: Uses exponential format, equivalent to "%<decimalDigits>e".

labelFSize=*s*

Specifies the font size of contour labels in points. For example, use labelSize=12 for 12 point type. The default value is 0, which chooses the size automatically based on the size of the graph.

labelFStyle=*n*

n is a binary coded number with each bit controlling one aspect of the font style for the contour level labels. The default is 0, plain text.

bit 0: Bold.

bit 1: Italic.

bit 2: Underline.

bit 3: Outline (*Macintosh only*).

bit 4: Shadow (*Macintosh only*).

bit 5: Condensed (*Macintosh only*).

bit 6: Extended (*Macintosh only*).

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

labelHV=*hv* Specifies the contour label orientation.

If *hv* is 3, 4, 5, or 6, the contour label's text rotates whenever it is redrawn, usually when the underlying contour data changes, the graph is resized, or the label is reattached to a new contour trace point.

hv=0: Horizontal contour level labels.

hv=1: Vertical contour level labels.

hv=2: Horizontal or vertical contour level labels, depending on the slope of the contour line.

hv=3: Tangent to the contour line.

hv=4: Tangent to the contour line, snaps to vertical or horizontal if within 2 degrees of vertical or horizontal (default).

hv=5: Perpendicular to the contour line.

hv=6: Perpendicular to the contour line, snaps to vertical or horizontal if within 2 degrees of vertical or horizontal.

labelRGB=(*r*, *g*, *b*) Sets the text color for all contour level labels. *r*, *g*, and *b* are values from 0 to 65535. The default is black, labelRGB=(0,0,0).

labels=*l* *l*=0: Hides contour level labels.

l=1: Leaves any contour level labels in place but stops updating them and stops generation of new labels.

l=2: Generates or updates labels for the existing contour levels and window size when the command executes, but disables further updating of labels when window size or contour plot changes. This is the recommended setting if updating the labels takes long enough to annoy you.

l=3: Default; generates labels for all contour levels whenever the contoured data changes but not when the window size changes. If you resize the graph, the labels may overlap or be too sparse.

l=4: Generates labels for all contour levels whenever the contoured data, contour levels, axis range, or the graph size changes. (Actually, there are too many causes to list here. If all this update annoys you, use labels=2 "update once, now".)

labelSigDigits=*d* *d* is the number of significant digits when labelFormat=0 is used.

logLines= 1 or 0 0 sets the default linearly-spaced contour line colors.

1 turns on logarithmically-spaced line colors. This requires that the contour levels values be greater than 0 to display correctly.

Affects line color only when the cIndexLines or ctabLines parameter is used.

logLines does not affect the contour levels. To assign logarithmically-spaced contour levels, use the moreLevels parameter and disable autoLevels, for example:

```
ModifyContour ''#0 autoLevels={*,*,0}       // No auto levels
ModifyContour ''#0 moreLevels=0,moreLevels={1e-07,1e-06,1e-05,1e-04}
```

The logLines keyword was added in Igor Pro 6.22.

manLevels= {*firstLevel*, *increment*, *numLevels*}

Explicitly specifies contour levels. ModifyContour will generate *numLevels* contour levels, evenly spaced starting from *firstLevel* and stepping by *increment*.

manLevels cancels the effect of any previous manLevels or autoLevels settings.

manLevels= *manLevelsWave*

Explicitly specifies contour levels. ModifyContour will generate contour levels at the values in *manLevelsWave*.

manLevels cancels the effect of any previous manLevels or autoLevels settings.

moreLevels= {*level*, *level* …}

Explicitly specifies contour levels. ModifyContour will generate a contour trace for each of the listed levels. The maximum number of levels that you can specify in a single command is the 50. However, you can concatenate any number of ModifyContour moreLevels commands. moreLevels adds levels in addition to any specified by manLevels or autoLevels. It does not override other parameters.

moreLevels=0: Removes all levels generated by previous moreLevels settings.

nullValue=*zValue* Treat blanks (NaNs) in the data as if they were the specified zValue.

The nullValue keyword is allowed only for XYZ contours, created by **AppendXYZContour**.

nullValueAuto Treats blanks (NaNs) in the data as if they were the minimum value in the Z wave minus 1.

The nullValueAuto keyword is allowed only for XYZ contours, created by **AppendXYZContour**.

perturbation=*p* Enable or disable perturbation (alteration) of the x and y values by a miniscule amount to improve the natural neighbor triangulation of XYZ contours.

p=0: Disables perturbation, preserving the original x and y values unchanged.

p=1: Enables x/y perturbation (default). The values are shifted by random values less than +/-0.000005 times the x and y domain extents.

You can observe the perturbed x/y coordinates in the triangulation trace added by ModifyContour triangulation=1.

The perturbation keyword is allowed only for XYZ contour plots.

rgbLines=(*r*, *g*, *b*) Specifies red, green, and blue values for all contour lines. *r*, *g*, and *b* are values from 0 to 65535.

If you use rgbLines, you must not use cIndexLines or ctabLines in the same command.

triangulation=*t* Draws the Delaunay Triangulation. As part of the XYZ contouring algorithm, the XY domain is subdivided into triangles in a process called Delaunay Triangulation. Like the contour lines, the triangulation is drawn using a graph trace, whose name is usually something like "contourInstanceName =triangulation".

The triangulation keyword is allowed only for XYZ contours, created by **AppendXYZContour**.

t=0: Hides the Delaunay triangulation (default).

t=1: Shows the Delaunay triangulation.

update=*u* Sets the type of updating of contour traces when the data or contour settings change.

u=0: Turns off dynamic updates, which might be advisable if updates take a long time.

u=1: Updates the contours only once, or until you next execute an update=1 command.

u=2: Updates are automatic (default).

u=3: Marks the contour plot as having been updated once (*u*=1) already. This option is used in recreation macros to prevent an extra redraw of a graph saved with *u*=1 update mode in effect.

If you use it in a command, the result is similar to *u*=0, but the Modify Contour Appearance dialog will automatically select "update once, now" from the Update Contours pop-up menu.

xymarkers=*x* *x*=0: Hides markers showing XY coordinates of the Z data (default).

x=1: Displays markers showing XY coordinates of Z data. Initially, this uses marker number zero. You can change this using the Modify Trace Appearance dialog.

Flags

/W=*winName* Applies to contours in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

See Also

AppendMatrixContour and **AppendXYZContour**.

References

Watson, David F., *nngridr - An Implementation of Natural Neighbor Interpolation*, Dave Watson Publisher, Claremont, Australia, 1994.

ModifyControl

ModifyControl [**/Z**] ***ctrlName*** [***keyword = value*** [**,** ***keyword = value*** **…**]]

The ModifyControl operation modifies the named control. ModifyControl works on any kind of existing control. To modify multiple controls, use **ModifyControlList**.

Parameters

ctrlName specifies the name of the control to be created or changed. The control must exist.

Keywords

The following keyword=value parameters are supported:

activate	appearance	bodywidth	disable	fColor	font
fSize	fStyle	help	labelBack	noproc	pos
proc	rename	size	title	userdata	valueBackColor
valueColor	win				

For details on these keywords, see the documentation for **SetVariable** on page V-565.

The following keywords are not supported:

mod	popmatch	popvalue	value	variable

Flags

/Z No error reporting.

Details

Use ModifyControl to move, hide, disable, or change the appearance of a control without regard to its kind

Example

Here is a **TabControl** procedure that shows and hides all controls in the tabs appropriately, without knowing what kind of controls they are.

The "trick" here is that all controls that are to be shown within particular tab *n* have been assigned names that end with "_tab*n*" such as "_tab0" and "_tab1":

```
Function TabProc(ctrlName,tabNum) : TabControl
   String ctrlName
   Variable tabNum

   String curTabMatch= "*_tab"+num2istr(tabNum)

   String controls= ControlNameList("")
   Variable i, n= ItemsInList(controls)
   for(i=0; i<n; i+=1)
      String control= StringFromList(i, controls)
      Variable isInATab= stringmatch(control,"*_tab*")
      if( isInATab )
         Variable show= stringmatch(control,curTabMatch)
         ControlInfo $control                    // gets V_disable
         if( show )
            V_disable= V_disable & ~0x1          // clear the hide bit
         else
            V_disable= V_disable | 0x1           // set the hide bit
         endif
         ModifyControl $control disable=V_disable
      endif
   endfor
   return 0
End

// Action procedures which enable or disable the buttons
Function Tab1CheckProc(ctrlName,enableButton) : CheckBoxControl
   String ctrlName
   Variable enableButton

   ModifyControl button_tab1, disable=(enableButton ? 0 : 2 )
End
```

```
Function Tab0CheckProc(ctrlName,enableButton) : CheckBoxControl
   String ctrlName
   Variable enableButton

   ModifyControl button_tab0, disable=(enableButton ? 0 : 2 )
End

// Panel macro that creates a TabControl using TabProc
Window TabbedPanel() : Panel
   PauseUpdate; Silent 1            // building window...
   NewPanel /W=(381,121,614,237) as "Tab Demo"
   TabControl tab, pos={12,9},size={205,91},proc=TabProc,tabLabel(0)="Tab 0"
   TabControl tab, tabLabel(1)="Tab 1",value= 0
   Button button_tab0, pos={54,39},size={110,20},disable=2
   Button button_tab0, title="Button in Tab0"
   Button button_tab1, pos={54,63},size={110,20},disable=1
   Button button_tab1, title="Button in Tab1"
   CheckBox check1_tab1, pos={51,41}, size={117,14}, disable=1, value= 1
   CheckBox check1_tab1, proc=Tab1CheckProc, title="Enable Button in Tab 1"
   CheckBox check0_tab0, pos={51,73}, size={117,14}, proc=Tab0CheckProc
   CheckBox check0_tab0, value= 0, title="Enable Button in Tab 0"
EndMacro
```

Run TabbedPanel to create the panel. Then click on “Tab 0” and “Tab 1” to run TabProc.

See Also

See Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Related functions **ModifyControlList** and **ControlNameList**.

The **Button**, **Chart**, **CheckBox**, **GroupBox**, **ListBox**, **PopupMenu**, **SetVariable**, **Slider**, **TabControl**, **TitleBox**, and **ValDisplay** controls.

ModifyControlList

ModifyControlList [**/Z**] ***listStr*** [**, keyword = *value***]…

The ModifyControlList operation modifies the controls named in the *listStr* string expression. ModifyControlList works on any kind of existing control.

Parameters

listStr is a semicolon-separated list of names in a string expression. The expression can be an explicit list of control names such as "button0;checkbox1;" or it can be any string expression such as a call to the ControlNameList string function:

```
ModifyControlList ControlNameList("",";","*_tab0") disable=1
```

The controls must exist.

Keywords

The following keyword=value parameters are supported:

activate	appearance	bodywidth	disable	fColor	font
fSize	fStyle	help	labelBack	noproc	pos
proc	rename	size	title	userdata	valueBackColor
valueColor	win				

For details on these keywords, see the documentation for **SetVariable** on page V-565.

The following keywords are not supported:

mod	popmatch	popvalue	value	variable

Flags

/Z No error reporting.

Details

Use ModifyControlList to move, hide, disable, or change the appearance of multiple controls without regard to their kind.

If *listStr* contains the name of a nonexistent control, an error is generated.

if *listStr* is "" (or any list element in *listStr* is ""), it is ignored and no error is generated.

Example

Here is the **TabControl** procedure example from **ModifyControl** rewritten to use ModifyControlList. It shows and hides all controls in the tabs appropriately, without knowing what kind of controls they are, but the code is simpler. This method does not, however, preserve the enable bit when a control is hidden.

The "trick" here is that all controls that are to be shown within particular tab *n* have been assigned names that end with "_tab*n*" such as "_tab0" and "_tab1":

```
// Action procedure
Function TabProc2(ctrlName,tabNum) : TabControl
    String ctrlName
    Variable tabNum

    String controlsInATab= ControlNameList("",";","*_tab*")

    String curTabMatch= "*_tab"+num2istr(tabNum)
    String controlsInCurTab= ListMatch(controlsInATab, curTabMatch)
    String controlsInOtherTabs=ListMatch(controlsInATab,"!"+curTabMatch)

    ModifyControlList controlsInOtherTabs disable=1          // hide
    ModifyControlList controlsInCurTab disable=0             // show

    return 0
End

// Panel macro that creates a TabControl using TabProc2():
Window TabbedPanel2() : Panel
    PauseUpdate; Silent 1                   // building window…
    NewPanel /W=(35,208,266,374) as "Tab Demo"
    TabControl tab,pos={12,9},size={205,140},proc=TabProc2
    TabControl tab,tabLabel(0)="Tab 0"
    TabControl tab,tabLabel(1)="Tab 1",value= 0
    Button button_tab0,pos={26,43},size={110,20},title="Button in Tab0"
    Button button2_tab0,pos={26,74},size={110,20},title="Button in Tab0"
    Button button3_tab0,pos={26,106},size={110,20},title="Button in Tab0"
    Button button_tab1,pos={85,43},size={110,20},title="Button in Tab1"
    Button button2_tab1,pos={85,75},size={110,20},title="Button in Tab1"
    Button button3_tab1,pos={84,108},size={110,20},title="Button in Tab1"
    ModifyControlList ControlNameList("",";","*_tab1") disable=1
EndMacro
```

Run TabbedPanel2 and then click on "Tab 0" and "Tab 1" to run TabProc2.

See Also

See Chapter III-14, **Controls and Control Panels** for details about control panels and controls.

Related functions **ModifyControl** and **ControlNameList**.

The **Button**, **Chart**, **CheckBox**, **GroupBox**, **ListBox**, **PopupMenu**, **SetVariable**, **Slider**, **TabControl**, **TitleBox**, and **ValDisplay** controls.

ModifyFreeAxis

ModifyFreeAxis [**/W=*winName***] ***axisName*, master=*mastName***
    [**, hook=*funcName***]

The ModifyFreeAxis operation designates the free axis (created with **NewFreeAxis**) to follow a controlling axis from which it gets axis range and units information. The free axis updates whenever the controlling axis changes. The axis limits and units can be modified by a user hook function.

Parameters

axisName is the name of the free axis (which must have been created by **NewFreeAxis**).

masterName is the name of the master axis controlling *axisName*.

funcName is the name of the user function that modifies the limits and units properties of the axis. If *funcName* is $"", the named hook function is removed.

Flags

/W=*winName* Modifies *axisName* in the named graph window or subwindow. If /W is omitted the command affects the top graph window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

The free axis can also be designated to call a user-defined hook function that can modify limits and units properties of the axis. The hook function must be of the following form:

```
Function MyAxisHook(info)
   STRUCT WMAxisHookStruct &info

   <code to modify graph units or limits>
   return 0
End
```

where `WMAxisHookStruct` is a built-in structure with the following members:

`WMAxisHookStruct` Structure Members

Member	Description
`char win[MAX_WIN_PATH+1]`	Host (sub)window.
`char axName[MAX_OBJ_NAME+1]`	Name of the axis.
`char mastName[MAX_OBJ_NAME+1]`	Name of controlling axis or nil.
`char units[MAX_UNITS+1]`	Axis units. User modifiable.
`double min, max`	Axis range minimum and maximum values. User modifiable.

The constants used to size the `char` arrays are internal to Igor and are subject to change in future versions.

The hook function is called when refreshing axis range information (generally early in the update of a graph). Your hook must never kill a graph or an axis.

See Also

The **SetAxis**, **KillFreeAxis**, and **NewFreeAxis** operations.

The **ModifyGraph (axes)** operation for changing other aspects of a free axis.

ModifyGraph *(general)*

`ModifyGraph` `[/W=`***`winName`***`/Z]` ***`key=value`*** `[,` ***`key=value`***`]…`

The ModifyGraph operation modifies the target or named graph. This section of ModifyGraph relates to general graph window settings.

Parameters

expand=*e*
: Specifies the onscreen expansion (or magnification) factor of a graph. *e* may be zero or 0.125 to 8 times expansion.

  Graph magnification affects only base graphs (not subwindowed graphs), and it affects only the onscreen display; it has **no** effect on graph exporting or printing.

  When magnification changes, the graph window will automatically resize except for negative values, which are used in recreation macros where the size is already correct.

frameInset= *i*
: Specifies the number of pixels by which to inset the frame of the graph subwindow.

frameStyle= *f*
: Specifies the frame style for a graph subwindow.

  f=0: None.
  f=1: Single.
  f=2: Double.
  f=3: Triple.
  f=4: Shadow.
  f=5: Indented.
  f=6: Raised.
  f=7: Text well.

The last three styles are fake 3D and will look good only if the background color of the enclosing space and the graph itself is a light shade of gray.

gfMult=*f* — Multiplies font and marker size by *f* percent. Clipped to between 25% and 400%; it is applied after all other font and marker size calculations.

gFont=*fontStr* — Specifies the name of the default font for the graph, overriding the normal default font. The normal default font for a subgraph is obtained from its parent while a base graph uses the value set by the **DefaultFont** operation.

gfSize=*gfs* — Sets the default size for text in the graph. Normally, the default size for text is proportional to the graph size; gfSize will override that calculation as will the gfRelSize method. Use a value of -1 to make a subgraph get its default font size from its parent.

gfRelSize=*pct* — Specifies the percentage of the graph size to use in calculating a default size for text in the graph. This overrides the normal method for setting default font size as a function of graph size. When used, the default marker size is set to one third the font size. Use a value of 0 to revert to the default method.

gmSize=*gms* — Sets the default size for markers in the graph. Use a value of -1 to make a subgraph get its default marker size from its parent.

height=*heightSpec* — Sets the height for the graph area. See the **Examples**.

swapXY=*s*
- *s*=0: Normal orientation of X and Y axes.
- *s*=1: Swap X and Y values to plot Y coordinates versus the horizontal axes and X coordinates versus the vertical axes. The effect is similar to mirroring the graph about the lower-left to upper-right diagonal.

useComma=*uc*
- *uc*=0: Use period as decimal separator and comma as thousands separator (default) when displaying numbers in graph labels and annotations.
- *uc*=1: Use comma as decimal separator and period as the thousands separator. This does not alter the presentation of numbers in \{*expression*} constructs in annotations.

useLongMinus=*m* — Uses a normal (*m*=0; default) or long dash (*m*=1) for the minus sign.

width=*widthSpec* — Sets the width of the graph area. See the examples.

Flags

/W=*winName* — Modifies the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/Z — Does not generate an error if the indexed trace, named wave, or named axis does not exist in a style macro.

Examples

The following code creates a graph where all the text expands and contracts directly in relation to the window size:

```
Make jack=sin(x/8);display jack
ModifyGraph mode=4,marker=8,gfRelSize= 5.0
TextBox/N=text0/A=MC "Some \\Zr200big\\]0 and \\Zr050small\\]0\rtext"
```

The *widthSpec* and *heightSpec*s set the width and height mode for the top graph. The following examples illustrate how to specify the various modes.

```
ModifyGraph width=0, height=0
```
Set to auto height, width mode. The width, height of horizontal and vertical axes are automatically determined based on the overall size of the graph and other factors such as axis offset setting and effect of exterior textboxes. This is the normal, default mode.

```
Variable n=72*5
ModifyGraph width=n
```
Five inches as points absolute width mode, horizontal axis width constrained to `n` points.

```
ModifyGraph height=n
```
Absolute height mode, `n` is in points. The height of the vertical axes is constrained to `n` points.

```
Variable n=2
ModifyGraph
width={perUnit,n,bottom}
```

Per unit width mode. The width of the horizontal axes is `n` points times the range of the bottom axis.

```
ModifyGraph height={Aspect,n}
```

Aspect height mode, `n = aspect ratio`. The height of the vertical axes is `n` times the width of the horizontal axes.

```
ModifyGraph
width={Plan,n,bottom,left}
```

Plan width mode. The width of the horizontal axes is `n` times the height of the vertical axes times range of the bottom axis divided by the range of the left axis.

ModifyGraph *(traces)*

ModifyGraph [**/W=*winName*/Z**] ***key*** [**(*traceName*)**] **= *value***
[**, *key*** [**(*traceName*)**] **= *value***]…

This section of ModifyGraph relates to modifying the appearance of wave "traces" in a graph. A trace is a representation of the data in a wave, usually connected line segments.

Parameters

Each *key* parameter may take an optional *traceName* enclosed in parentheses. Usually *traceName* is simply the name of a wave displayed in the graph, as in "`mode(myWave)=4`". If "(*traceName*)" is omitted, all traces in the graph are affected. For instance, "`ModifyGraph lSize=0.5`" sets the lines size of all traces to 0.5 points.

For multiple trace instances, *traceName* is followed by the "#" character and instance number. For example, "`mode(myWave#1)=4`". See **Instance Notation** on page IV-16.

A string containing a trace name can be used with the $ operator to specify *traceName*. For example, `String MyTrace="myWave#1"; mode($MyTrace)=4.`

Though not shown in the syntax, the optional "(*traceName*)" may be replaced with "[*traceIndex*]", where *traceIndex* is zero or a positive integer denoting the trace to be modified. "[0]" denotes the first trace appended to the graph, "[1]" denotes the second trace, etc. This syntax is used for style macros, in conjunction with the /Z flag.

For certain modes and certain properties, you can set the conditions at a specific point on a trace by appending the point number in square brackets after the trace name. For more information, see the **Customize at Point** on page V-407. This feature was added in Igor Pro 6.20.

The parameter descriptions below omit the optional "(*traceName*)". When using ModifyGraph from a user-defined function, be careful not to pass wave references to ModifyGraph. ModifyGraph expects trace names, not wave references. See **Trace Name Parameters** on page IV-71 for details.

arrowMarker=0

arrowMarker={*aWave*, *lineThick*, *headLen*, *headFat*, *posMode* [, *barbSharp=b*, *barbSide=s*, *frameThick=f*]}

Draws arrows instead of conventional markers at each data point in a wave. Arrows are not clipped to the plot area and will be drawn wherever a data point is within the plot area.

aWave contains arrow information for each data point. It is a two (or more) column wave containing arrow line lengths (in points) in column 0 and angles (in radians measured counterclockwise) in column 1. Zero angle is a horizontal arrow pointing to the right. If an arrow is below the minimum length of 4 points, a default marker is drawn.

You can change arrow markers into standard meteorological wind barbs by adding a column to *aWave* and giving it a column label of windBarb. Values are integers from 0 to 40 representing wind speeds up to 4 flags. Use positive integers for clockwise barbs and negative for the reverse. Use NaN to suppress the drawing. (See **Examples**.)

Additional columns may be supplied in *aWave* to control parameters on a point by point basis. These optional columns are specified by dimension label and not by specific column numbers. The labels are *lineThick*, *headLen*, and *headFat* that correspond to the same parameters listed above.

lineThick is the line thickness in points.

headLen is the arrow head length in points.

headFat controls the arrow fatness. It is the width of the arrow head divided by the length.

posMode specifies the arrow location relative to the data point.

posMode=0: Start at point.

posMode=1: Middle on point.

posMode=2: End at point.

In addition to the wave specification, *aWave* can also be the literal `_inline_` to draw lines and arrows between points on the trace (see **Examples**). If *aWave* is `_inline_`, *posMode* values are:

posMode=0: Arrow at end.

posMode=1: Arrow in middle.

posMode=2: Arrow at start.

posMode=3: Arrow in middle pointing backwards.

Note: Arrow behavior as specified by *posMode* is different when you use a multicolumn wave vs. `_inline_` for *aWave*.

Optional parameters must be specified using *keyword* = *value* syntax and can only be appended after *posMode* in any order.

barbSharp is the continuously variable barb sharpness between -1.0 and 1. 0:

barbSharp=1: No barb; lines only.

barbSharp=0: Blunt (default).

barbSharp=-1: Diamond.

barbSide specifies which side of the line has barbs relative to a right-facing arrow:

barbSide=0: None.

barbSide=1: Top.

barbSide=2: Bottom.

barbSide=3: Both (default).

frameThick specifies the stroke outline thickness of the arrow in points. The default is *frameThick* = 0 for solid fill.

aWave can contain columns with data for each optional parameter using matching column names.

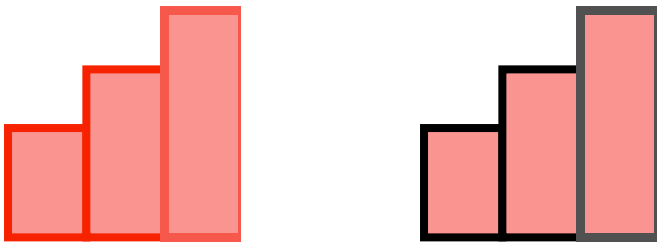
barStrokeRGB=(*r*,*g*,*b*)

Specifies a separate color for bar strokes (outlines) if useBarStrokeRGB is 1. *r*, *g* and *b* specify the amount of red, green and blue in the color of the stroked lines as an integer from 0 to 65535. The default is black (0,0,0).

Applies only to Histogram Bars drawing mode (mode=5).

The bar fill color continues to be set with the rgb=(*r*,*g*,*b*), zColor={...}, usePlusRGB, plusRGB=(*r*,*g*,*b*), useNegRGB, and negRGB=(*r*,*g*,*b*) parameters.

Use barStrokeRGB and useBarStrokeRGB to put a differently-colored outline around Histogram Bars:



cmplxMode=*c* Display method for complex waves.

c=0: Default mode displays both real and imaginary parts (imaginary part offset by dx/2).

c=1: Real part only.

c=2: Imaginary part only.

c=3: Magnitude.

c=4: Phase (radians).

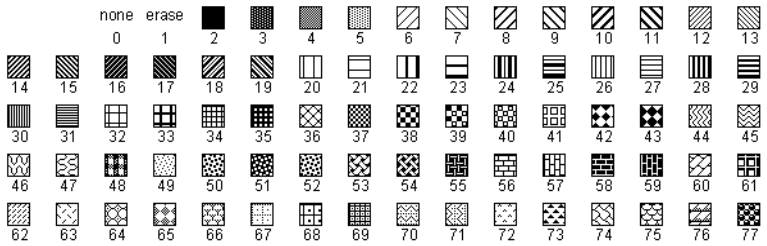
column=*n* Changes the displayed column from a matrix. Out of bounds values are clipped.

gaps=*g*
- *g*=0: No gaps (ignores NaNs).
- *g*=1: Gaps (shows NaNs as gaps).

hBarNegFill=*n* Fill kind for negative areas if useNegPat is true. *n* is the same as for the hbFill keyword.

hbFill=*n* Sets the fill pattern.
- *n*=0: No fill.
- *n*=1: Erase.
- *n*=2: Solid black.
- *n*=3: 75% gray.
- *n*=4: 50% gray.
- *n*=5: 25% gray.

n= 6 through 77 selects one of the 72 fill patterns.



hideTrace=*h* Removes a trace from the graph display.
- *h*=1: Hides the trace and removes it from autoscale calculations.
- *h*=2: Hides the trace.

When using *h*=1 to hide a graph trace, the hidden trace symbol and following text in annotations are also hidden. The amount of hidden text is the lesser of: 1) the remaining text on the same line or 2) the text up to but not including another trace symbol `"\s(traceName)"`.

lHair=*lh* Sets the hairline factor for traces printed on a PostScript® printer.

live=*lv* Turns Live Mode off (*lv*=0) or on (*lv*=1).

logZColor=*lzc*
- *lzc* =0: Sets the default linearly-spaced zColors (see the zColor parameter).
- *lzc* =1: Turns on logarithmically-spaced zColors. This requires that the zWave values be greater than 0 to display correctly.

Affects trace line color only when the zColor parameter is used with a color table or color index wave - it has no effect if rgb=(*r*,*g*,*b*) parameter or zColor={...,directRGB} are used.

logZColor was added in Igor Pro 6.22.

lOptions=*options* *options* is a bitwise parameter:

bit 0: If set, dashed lines use round end caps. If cleared they use square end caps.

All other bits are reserved and must be cleared.

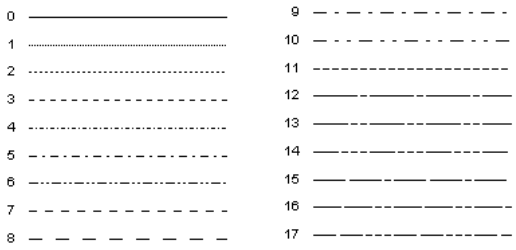
See **Setting Bit Parameters** on page IV-12 for details about bit settings.

lSize=*l* Sets the line thickness, which can be fractional or zero, which hides the line.

lSmooth=*ls* Sets the smoothing factor for traces printed on a PostScript® printer.

lStyle=*s* Sets trace line style or dash pattern.

s=0 for solid lines. *s*=1 to s=17 for various dashed line styles.



marker=*n*

n =0 to 62 designates various markers if mode=3 or 4.

Markers 51 through 62 require Igor Pro 6.1 and are available in new graphics only (see Graphics Technology). You can also create custom markers. See the **SetWindow** markerHook keyword.

See **Markers** on page II-250 for a table of marker values.

mask={*maskwave*,*mode*,*value*} or 0

Specifies individual points for display by comparing values in *maskWave* with *value* as specified by *mode*.

mode=0: Exclude if equal.

mode=1: Include if equal.

mode=2: Include if bitwise AND is true.

mode=3: Include if bitwise AND is false.

maskwave can be specified using subrange notation. The length of *maskwave* (or subrange) must match the size of specified trace's wave (or subrange.) Bitwise modes should be used with integer waves with the intent of using one mask wave with multiple traces. (See **Examples**.)

mode=*m*

Sets trace display mode.

m=0: Lines between points.

m=1: Sticks to zero.

m=2: Dots at points.

m=3: Markers.

m=4: Lines and markers.

m=5: Histogram bars.

m=6: Cityscape.

m=7: Fill to zero.

m=8: Sticks and markers.

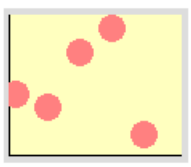
mrkStrokeRGB=(*r*,*g*,*b*)

Specifies the color for marker stroked lines if useMrkStrokeRGB = 1. *r*, *g*, and *b* values are the amount of red, green, and blue in the color of the lines as an integer from 0 to 65535. The default is black (0,0,0).

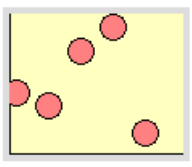
The marker fill color continues to be set with the rgb=(*r*,*g*,*b*) or zColor={…} parameters.

Applies only to the nontext and nonarrow marker modes.

Use mrkStrokeRGB and useMrkStrokeRGB to put a colored outline around filled markers, such as marker=19:



useMrkStrokeRGB=0



useMrkStrokeRGB=1

Note: The stroke color of unfilled markers such as marker 8 is also affected by mrkStrokeRGB, but their fill color is only affected by the opaque parameter (and the opaque fill color is always white, so if you want a color-filled marker, don't use unfilled markers).

mrkThick=*t* Sets the thickness of markers in points, which can be fractional.

msize=*m* Specifies the marker size in points.

m=0: Autosize markers.

m>0: Sets marker size.

m can be fractional, which will only make a difference when the graph is printed because fractional points can not be displayed on the screen.

mskip=*n* Puts a marker on only every *n*th data point in Lines and Markers mode (mode=4). Useful for displaying many data points when you want to identify the traces with markers. The maximum value for *n* is 32767.

muloffset={*mx*,*my*} Sets the display multiplier for X (*mx*) and Y (*my*). The effective value for a given X or Y data point then becomes *muloffset***data*+*offset*. A value of zero means "no multiplier" — not multiply by zero.

negRGB=(*r*, *g*, *b*) Specifies the color for negative areas if useNegRGB is 1. *r*, *g*, and *b* specify the amount of red, green, and blue in the color of the trace as an integer from 0 to 65535.

offset={*x*,*y*} Sets the display offset in horizontal (X) and vertical (Y) axis units.

opaque=*o* Displays transparent (*o*=0) or opaque (*o*=1) markers.

patBkgColor= 0, 1, 2 or (r,g,b)

Specifies the background color for fill patterns.

0, the default, is white, 1 is graph background, 2 is transparent (does not work when exporting in the Enhanced Metafile or Windows Metafile formats).

Use (r,g,b) for a specific RGB color.

plotClip=*p* *p* =1 clips the trace by the operating system (not by Igor) to the plot rectangle. This trims overhanging markers and thick lines. On Windows, this may not be supported for certain printers or by certain applications when importing.

plusRGB=(*r*, *g*, *b*) Specifies the color for positive areas if usePlusRGB is 1. *r*, *g*, and *b* specify the amount of red, green, and blue in the color of the trace as an integer from 0 to 65535.

quickdrag=*q* *q*=0: Normal traces.

q=1: Traces that can be instantly dragged without the normal one second delay. See the **Quickdrag** section below.

q=2: Causes the mouse cursor to change to 4 arrows when over the trace and a reduced search is used.

rgb=(*r*,*g*,*b*) Specifies the amount of red, green, and blue (*r*, *g*, and *b)* in the color of the trace as an integer from 0 to 65535.

textMarker={<*char* or *wave*>,*font*,*style*,*rot*,*just*,*xOffset*,*yOffset*} or 0

Uses the specified character or text from the specified wave in place of the marker for each point in the trace.

If the first parameter is a quoted string or a string expression of the form `""+strexpr` in a user function, ModifyGraph uses the first three bytes of the string as the marker for all points (three bytes are provided mainly for multi-byte Asian fonts but can be used for 3 separate one byte characters). Otherwise, it interprets the first parameter as the name of a wave. If the wave is a text wave, it uses the value of each point in the text wave as the marker for the corresponding point in the trace. If the wave is a numeric wave, the value for each point is converted into text and the result is used as the marker for the corresponding point in the trace.

xOffset and *yOffset* are offsets in fractional points. Each marker will be drawn offset from the location of the corresponding point in the trace by these amounts.

style is a font style code as used with the ModifyGraph fstyle keyword.

rot is a text rotation between -360 and 360 degrees.

just is a justification code as used in the **DrawText** operation except the X and Y codes are combined as y*4+x. Use 5 for centered.

The font size is 3*marker size. Note that marker size and color can be dynamically set via the zColor and zmrkSize keywords.

toMode=*t* Modifies the behavior of the display modes as determined by the mode parameter.

t=0: Fill to zero.

t=1: Fill to next trace. Applies to Sticks to zero (mode=1), histogram bars (mode=5), and fill to zero (mode=7).

t=2: Add the current trace's Y values to the next trace's Y values. Works with all display modes.

t=3: Stack on next and is the same as *t*=2 except that the added value is clipped to zero. Works with all display modes.

t=-1: This mode is used only with category plots and means "keep with next" (i.e., put in the same subcategory as the next trace). It is used for special effects only.

useBarStrokeRGB=*u*

If *u*=1 then bar stroked lines use the color specified by the barStrokeRGB keyword.

Applies only to Histogram Bars drawing mode (mode=5).

The bar fill color continues to be set with the rgb=(*r*,*g*,*b*), zColor={...}, usePlusRGB, plusRGB=(*r*,*g*,*b*), useNegRGB, and negRGB=(*r*,*g*,*b*) parameters.

if *u*=0 then the bar stroked line colors are set with the rgb=(*r*,*g*,*b*) or zColor={...} parameters, just like the bar fill color.

useMrkStrokeRGB=*u*

If *u* =1 then marker stroked lines use the color specified by the mrkStrokeRGB keyword. The marker fill color continues to be set with the rgb=(*r*,*g*,*b*) or zColor={…} parameters.

Applies only to the nontext and nonarrow marker modes.

If *u*=0 then the marker stroked line colors are set with the rgb=(*r*,*g*,*b*) or zColor={…} parameters, just like the marker fill color.

useNegPat=*u* If *u*=1, negative fills use the mode specified by the hBarNegFill keyword. Applies to the fill-to-zero, fill-to-next and histogram bar modes.

useNegRGB=*u* If *u* =1, negative fills use the color specified by the negRGB keyword. Applies to the fill-to-zero, fill-to-next and histogram bar modes.

usePlusRGB=*u* If *u* =1, positive fills use the color specified by the plusRGB keyword. Applies to the fill-to-zero, fill-to-next and histogram bar modes.

userData={*udName*, *doAppend*, *data*}

Attaches arbitrary data to a trace. You should specify a trace name (userData(<traceName>)={...}). Otherwise copies of the data will be attached to every trace, which is most likely not what you intend.

Use the GetUserData function to retrieve the data, with the trace name as the object ID.

udName: The name of your user data. Use $"" for unnamed user data.

doAppend=0: Do not append. Any pre-existing data is replaced.

doAppend=1: Append the data. Data is added to the end of any pre-existing data.

data: A string expression containing the data you wish to attach to the trace.

zColor={*zWave*,*zMin*,*zMax*,*ctName* [,*reverseMode* [,*ciWave*]]} or 0

Dynamically sets color based on the values in *zWave* and color table specified by *ctName*. Use * or a missing parameter for *zMin* and *zMax* to autoscale. See color index mode, below, for usage of *ciWave*.

zWave may be a subrange expression such as `myZWave[2,9]` when *zWave* has more points than the trace, in which case `myZWave[2]` provides the Z value for the first point of the trace, and autoscaled *zMin* or *zMax* is determined over only the *zWave* subrange.

If a value in the *zWave* is NaN then a gap or missing marker will be observed. If a value is out of range it will be replaced with the nearest valid value. See also the **zColorMax** and **zColorMin** parameters.

ctName can be any color table name returned by the **CTabList** function, such as Grays or Rainbow (for color table mode), or it can be cindexRGB for color index mode, or directRGB for direct color mode.

For color table mode, set the *reverseMode* parameter to 1 to reverse the color table; zero sets the color table to unreversed. A *reverseMode* value of -1 (or if *reverseMode* is missing) leaves the color table reverse state unchanged. Also see **Color Table Details** on page II-353.

Normally the colors from the color table are linearly distributed between *zMin* and *zMax*. Use logZColor=1 to distribute them logarithmically.

The *zMin* and *zMax* parameters are not used and should be set to *. Set *ctName* to cindexRGB and *reverseMode* to 0.

Specify a 3 column wave of RGB values for *ciWave* (not for *zWave*). The *zWave* values select the color from the row of *ciWave* whose X scaling is closest to the *zWave* value.

Example color table zColor command:

```
zColor(data)={myZWave[2,9],*,*,Rainbow,1}
```

You can directly specify the color of each point in a trace by using `directRGB` for *ctName*. In this case, *zWave* is a 3 column wave of RGB values (column 0 is red, 1 is green, and 2 is blue) corresponding to each point in the trace. If *zWave* is 8-bit unsigned integer, then color values range from 0 to 255; for other numeric types color values range from 0 to 65535. The *zMin* and *zMax* parameters are not used and should be set to `*`. Also see the **ColorTab2Wave** operation (which generates such a wave) and **Indexed Color Details** on page II-356.

Example directRGB zColor command:

```
zColor(data)={zWaveRGB,*,*,directRGB}
```

Color index mode uses both a *zWave* and a three-column RGB color index wave. See **Color Index Wave** on page II-330.

The *zMin* and *zMax* parameters are not used and should be set to `*`, set *ctName* to cindexRGB. Specify a value for *reverseMode*, 0 for normal, 1 for reversed color indexing. Specify a a 3 column wave of RGB values for *ciWave* (not for *zWave*). Normally, the *zWave* values select the color from the row of *ciWave* whose X scaling is closest to the *zWave* value. *ReverseMode*=1 reverses the colors.

Example cindexRGB zColor command:

```
zColor(data)={myZWave,*,*,cindexRGB,0,M_colors}
```

(M_colors is generated by the **ColorTab2Wave**.)

zColor = 0 turns the zColor modes off.

Normally the colors from the color index wave are linearly distributed between minimum and maximum X scaling of the color index wave. Use logZColor=1 to distribute them logarithmically.

zColorMax=(*red*, *green*, *blue*)

Sets the color of the trace for zColor={*zWave*, …} values greater than the zColor's *zMax*. Also turns on zColorMax mode.

The *red*, *green*, and *blue* color values are in the range of 0 to 65535.

zColorMax=1, 0, or NaN

Turns zColorMax mode off, on, or transparent. These modes affect the color of zColor={*zWave*, …} values greater than the zColor's *zMax*.

1: Turns on zColorMax mode. The color of the affected trace pixels is black or the last color set by zColorMax=(*red*, *green*, *blue*).

0: Turns off zColorMax mode (default). The color of the affected trace pixels is the last color in the zColor's *ctname* color table.

NaN: Transparent zColorMax mode. Affected trace pixels are not drawn.

zColorMin=(*red*, *green*, *blue*)

Sets the color of the trace for zColor={*zWave*, …} values less than the zColor's *zMin*. Also turns zColorMin mode on.

The *red*, *green*, and *blue* color values are in the range of 0 to 65535.

zColorMin=1, 0, or NaN

Turns zColorMin mode off, on, or transparent. These modes affect the color of zColor={*zWave*, …} values less than the zColor's *zMin*.

1: Turns on zColorMin mode. The color of the affected image pixels is black or the last color set by zColorMin=(*red*, *green*, *blue*).

0: Turns off zColorMin mode (default). The color of the affected trace pixels is the first color in the zColor's *ctname* color table.

NaN: Transparent zColorMin mode. Affected trace pixels are not drawn.

zmrkNum={*zWave*} or 0

Dynamically sets the marker number for each point to the corresponding value in *zWave*. The values in *zWave* are the marker numbers (as used with the marker keyword). If a value in the *zWave* is NaN then no marker will be drawn at the corresponding point. If a value is out of range it will be replaced with the nearest valid value.

zmrkNum=0 turns this mode off.

zmrkSize={*zWave*,*zMin*,*zMax*,*mrkmin*,*mrkmax*} or 0

Dynamically sets marker size based on values in *zWave*. Use * or a missing parameter for *zMin* and *zMax* to autoscale. *mrkmin* and *mrkmax* can be fractional. If a value in the *zWave* is NaN then a gap or missing mark will be observed. The marker size is clipped to 20 on the high end and 1 on the low end. If a value is out of range it will be replaced with the nearest valid value.

Warning: The LaserWriter printer driver will not print certain marker types if they are "too small" relative to their line thickness. For example, the circle or square markers will disappear if they fall below about 2 in size with a thickness of 0.5. This effect is not observed if exported using Igor PostScript.

zmrkSize = 0 turns this mode off.

zpatNum={*zWave*} or 0

Dynamically sets the positive fill type/pattern number for each point to the corresponding value in *zWave*. The values in *zWave* are the pattern numbers (as used with the hbFill keyword). If a value in the *zWave* is NaN then the corresponding point will not be drawn. If a value is out of range it will be replaced with the nearest valid value.

zpatNum=0 turns this mode off.

Flags

/W=*winName* Modifies the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the Command Line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/Z Does not generate an error if the indexed trace, named wave, or named axis does not exist in a style macro.

Details

Live Mode (live=1) improves graph update performance when one or more of the waves displayed in the graph is frequently modified, for example, if the waves are being acquired from a data acquisition system. Live Mode traces do not autoscale the axes.

Waves supplied with zmrkSize, zmrkNum, and zColor may use **Subrange Display Syntax** on page II-288.

Quickdrag

Quick drag mode (quickdrag=1) is a special purpose mode for creating cross hair cursors using a package of Igor procedures. (See the Cross Hair Demo example experiment.) Normally you would have to click and hold on a trace for one second before entering drag mode. When quickdrag is in effect, there is no delay. If a trace is in quickdrag mode it should also be set to live mode. With this combination you can click a trace and immediately drag it to a new XY offset. In addition to quick drag mode, the cross hair package relies on Igor to store information about the drag in a string variable if certain conditions are in effect. The string variable name (that you have to create) is S_TraceOffsetInfo, which must reside in a data folder that has the same name as the graph (not title!) which in turn must reside in root:WinGlobals:. If these conditions are met, then after a trace is dragged, information will be stored in the string using the following key-value format: `GRAPH:<name of graph>;XOFFSET:<x offset value>;YOFFSET:<y offset value>;TNAME:<trace name>;`

Customize at Point

You can customize the appearance of individual points on a trace in a graph for bar, marker, dot and lines to zero modes using `key(tracename[pnt])=value` syntax. The point number must be a literal number and the trace name is not optional. To turn off a customization, use `key(tracename[-pnt-1])=value`

where value is not important but must match the syntax for the keyword. The offset of -1 is needed because point numbers start from zero.

Although the syntax is allowed for all trace modifiers, it has meaning only for the following: rgb, marker, msize, mrkThick, opaque, mrkStrokeRGB, barStrokeRGB, hbFill, patBkgColor and lSize.

Note that useBarStrokeRGB and useMrkStrokeRGB are not needed. The act of using barStrokeRGB or mrkStrokeRGB is enough to customize the point. But as a convenience, since these are generated by the modify graph dialog, they are ignored if used with [pnt] syntax.

Also note that legend symbols can use [pnt] syntax like so:

```
\s(<tracename>[pnt])
```

Automatically generated legends automatically include symbols for customized points.

For example:

```
Make/O/N=10 jack=sin(x); Display jack
ModifyGraph mode=5,hbFill=6,rgb=(0,0,0)
ModifyGraph hbFill(jack[2])=7,rgb(jack[2])=(0,65535,0)
ModifyGraph rgb(jack[3])=(65535,0,0)
Legend/C/N=text1/F=0/A=MC
```

Examples

Following is a simple example of arrow markers.

```
Make/N=10 wave1= x; Display wave1
Make/N=(10,2) awave
awave[][0]= p*5               // length
awave[][1]= pi*p/9            // angle
ModifyGraph mode=3,arrowMarker(wave1)={awave,1,10,0.3,0}

// Now add an optional column to control headLen
Redimension/N=(-1,3) awave
awave[][2]= 7+p               // will be head length

// Note: nothing changes until the following is executed
SetDimLabel 1,2,headLen,awave
```

Create meteorological wind barb symbols.

```
Make/O/N=50 jack= floor(x/10),jackx= mod(x,10)
Display jack vs jackx
Make/O/N=(50,3) jackbarb
jackbarb[][0]= 40             // length of stem
jackbarb[][1]= 45*pi/180      // angle (45deg)
jackbarb[][2]= p              // wind speed code
SetDimLabel 1,2,windBarb,jackbarb
ModifyGraph mode=3,arrowMarker(jack)={jackbarb,1,10,0.5,0}
ModifyGraph margin(top)=62,margin(right)=84
```

Inline arrows and barb sharpness.

```
Make/O/N=20 wavex=cos(x/3),wavey=sin(x)
Display wavey vs wavex
ModifyGraph mode=3,arrowMarker={_inline_,1,20,.5,0,barbSharp= 0.2}
```

Use direct color mode to individually color each point in a trace:

```
Make jack=sin(x/8)
Make/N=(128,3)/B/U jackrgb
Display jack
ModifyGraph mode=3,marker=19
jackrgb= enoise(128)+128
ModifyGraph zColor(jack)={jackrgb,*,*,directRGB}
```

Use masking.

```
Make/N=100 jack= (p&1) ? sin(x/8) : cos(x/8)
Display jack

Make/N=100 mjack= (p&1) ? 0 : NaN       // just to show NaN can be used
ModifyGraph mask(jack)={mjack,0,NaN}

// now switch which points are shown
mjack= (p&1) ? NaN : 0
```

ModifyGraph *(axes)*

ModifyGraph [**/W=*winName*/Z**] ***key*** [**(*axisName*)**] **= *value***
[**, *key*** [**(*axisName*)**] **= *value***]…

This section of ModifyGraph relates to modifying the appearance of axes in a graph.

Parameters

Each *key* parameter may take an optional *axisName* enclosed in parentheses.

axisName is "left", "right", "top", "bottom" or the name of a free axis such as "vertCrossing". For instance, "`ModifyGraph axThick(left)=0.5`" sets the axis thickness for only the left axis.

If "(*axisName*)" is omitted, all axes in the graph are affected. For instance, "`ModifyGraph standoff=0`" disables axis standoff for all axes in the graph.

The parameter descriptions below omit the optional "(*axisName*)".

axisClip= *c* — Specifies one of three clipping modes for traces.

- *c*=0: Clips traces to a plot rectangle as defined by the pair of axes used by a given trace (default).
- *c*=1: Plots traces on an axis with a restricted range (as set by axisEnab) to extend to the full range of the normal plot rectangle.
- *c*=2: Traces extend outside the normal plot rectangle to the full extent of the graph area.

axisEnab={*lowFrac*,*highFrac*} — Restricts the length of an axis to a subrange of normal. The axis is drawn from *lowFrac* to *highFrac* of graph area height (vertical axis) or width (horizontal axis). For instance, {0.1,0.75} specifies that the axis is drawn from 10% to 75% of the graph area height/width, instead of the normal 0% to 100%. AxisEnab is discussed in **Creating Split Axes** on page II-297 and **Creating Stacked Plots** on page II-293.

axisOnTop=*t* — Specifies drawing level of axis and associated grid lines.

- *t*=0: Draws axis before traces and images (default).
- *t*=1: Draws the axis after all traces and images.

axOffset=*a* — Specifies the distance from default axis position to actual axis position in units of the width of a zero character (0) in a tick mark label. Unlike margin, axOffset adjusts to changes in the size of the graph.

axThick=*t* — Specifies the axis thickness in points.

barGap=*fraction* — Sets the fraction of the width available for bars to be used as gap between bars.

barGap sets the gap between bars within a single category while catGap sets the gap between categories.

btLen=*p* — Sets the length of major ("big") tick marks to *p* points. If *p* is zero, it uses the default length. *p* may be fractional.

btThick=*p* — Sets the thickness of major ("big") tick marks to *p* points. If *p* is zero, it uses the default thickness. *p* may be fractional.

catGap=*fraction* — The value for catGap is the fraction of the category width to be used as gap. The gap is divided equally between the start and end of the category width. A value of 0.2 would use 20% of the available space for the gap and leave 80% of the available space for the bars.

catGap sets the gap between categories while barGap sets the gap between bars within a single category.

dateFormat={*languageName*, *yearFormat*, *monthFormat*, *dayOfMonthFormat*, *dayOfWeekFormat*, *layoutStr*, *commonFormat*} — Sets the custom date format used in the active graph.

Note: Use a custom date format only if you turn it on via a ModifyGraph dateInfo command. The last parameter to the ModifyGraph dateInfo command must be -1 to turn on the custom date format.

Parameters are the same as for the LoadWave/R flag except for the last one.

commonFormat selects the common date format to use in the Modify Axis dialog. The legal values correspond to the choices in the Common Format pop-up menu of the Modify Axis dialog. They are:

Value	Date Format	Value	Date Format
1	mm/dd/yy	16	mm/yy
2	mm-dd-yy	17	mm.yy
3	mm.dd.yy	18	Abbreviated month and year
4	mmddyy	19	Full month and year
6	dd/mm/yy	21	mm/dd
7	dd-mm-yy	22	dd.mm
8	dd.mm.yy	23	Abbreviated month and day
9	ddmmyy	24	Full month and day
11	yy/mm/dd	26	Abbreviated date without day of week
12	yy-mm-dd	27	Abbreviated date with day of week
13	yy.mm.dd	28	Full date without day of week
14	yymmdd	29	Full date with day of week

If the *commonFormat* parameter is negative, then it will select the Use Custom Format radio button in the Modify Axis dialog rather than Use Common Format and will then use the absolute value of *commonFormat* to determine which item to select in the Common Format pop-up menu.

dateInfo={*sd*,*tm*,*dt*} Controls formatting of date/time axes.

sd=0: Show date in the date&time format.

sd=1: Suppress date.

tm=0: 12 hour (AM/PM) time.

tm=1: 24 hour (military) time.

tm=2: Elapsed time.

dt=-1: Custom date as specified via the dateFormat keyword.

dt=0: Short dates (2/22/90).

dt=1: Long dates (Thursday, February 22, 1990).

dt=2: Abbreviated dates (Thurs, Feb 22, 1990).

font="*fontName*" Sets the axis label font, e.g., `font(left)="Helvetica"`.

freePos(*freeAxName*)=*p*

Sets the position of the *free* axis relative to the edge of the plot area to which the axis is anchored. *p* is in points. i.e., if the axis was made via /R=*axName* then the axis is placed *p* points from the right edge of the plot area. Positive is away from the central plot area. *freeAxName* may not be any of the standard axes: “left”, “bottom”, “right” or “top”.

freePos(*freeAxName*)={*crossAxVal*,*crossAxName*}

Positions the *free* axis so it will cross the perpendicular axis *crossAxName* where it has a value of *crossAxVal*. *freeAxName* may not be any of the standard axis names “left”, “bottom”, “right”, or “top”, though *crossAxName* may.

You can position a free axis as a fraction of the distance across the plot area by using `kwFraction` for *crossAxName*. *crossAxVal* must then be between 0 and 1; any values outside this range are clipped to valid values.

fsize=*s* Autosizes (*s*=0) tick mark labels and axis labels.

If *s* is between 3 and 99 then the labels are fixed at *s* points.

fstyle=*f* *f* is a binary coded number with each bit controlling one aspect of the font style for the axis and tick mark labels as follows:

bit 0: Bold.

bit 1: Italic.

bit 2: Underline.

bit 3: Outline (*Macintosh only*).
bit 4: Shadow (*Macintosh only*).
bit 5: Condensed (*Macintosh only*).
bit 6: Extended (*Macintosh only*).
See **Setting Bit Parameters** on page IV-12 for details about bit settings.

ftLen=*p* Sets the length of 5th (or emphasized minor) tick marks to *p* points. If *p* is zero, it uses the default length. *p* may be fractional.

ftThick=*p* Sets the thickness of 5th (or emphasized minor) tick marks to *p* points (fractional). If *p* is zero, it uses the default thickness.

grid=*g* *g*=0: Grid off.
g=1: Grid on.
g=2: Grid on major ticks only.

gridEnab={*lowFrac*,*highFrac*}
Restricts the length of axis grid lines to a subrange of normal. The grid is drawn from *lowFrac* to *highFrac* of graph area height (if axis is horizontal) or width (if axis is vertical).

gridHair=*h* Sets the grid hairline thickness (*h* =0 to 3; 0 for thicker lines, 3 for thinner; default is 2). If *h*=0, the thickness of grid lines on major tick marks is the same as the axis thickness, half for a minor tick and one tenth for a subminor tick (log axis only). As *h* increases these thicknesses decrease by a factor of 2^*h*. If you want to see the effect of different values of gridHair, you will need to print a sample graph because you generally can't see the effect of thin lines on the screen. Also see the example experiment "Examples:Testing & Misc:Graph Grid Demo".

gridStyle=*g* Sets the grid style to various combinations of solid and dashed lines. In the following discussion, major, minor and subminor refer to grid lines the corresponding tick marks. Subminor ticks are used only on log axes when there is a small range and sufficient room (they correspond to hundredths of a decade). The different grid styes are solid, dotted, dashed, and blank. The possible grids are as follows:

g=0: Same as mode 1 if graph background is white else uses mode 5.
g=1: Major dotted, minor and subminor dashed.
g=2: All dotted.
g=3: Major solid, minor dotted, subminor blank.
g=4: Major and minor solid, subminor dotted.
g=5: All solid.

Also see the example experiment "Examples:Testing & Misc:Graph Grid Demo".

highTrip=*h* If the extrema of an axis are between its lowTrip and its highTrip then tick mark labels use fixed point notation. Otherwise they use exponential (scientific or engineering) notation.

lblLatPos=*p* Sets a lateral offset for the axis label. This is an offset parallel to the corresponding axis. *p* is in points. Positive is down for vertical axes and to the right for horizontal axes.

lblMargin=*l* Specifies the distance from the edge of graph to a label in points.

lblPos=*p* Sets the distance from an axis to the corresponding axis label in points. If *p*=0, it automatically picks an appropriate distance.

This setting is used only if the given graph edge has at least one free axis. Otherwise, the lblMargin setting is used to position the axis label.

lblPosMode= *m* Affects the meaning and usage of lblPos, lblLatPos, and lblMargin parameters. Mainly for use when you have multiple axes on a side and you need axis labels to be properly positioned even as you make graph windows dramatically larger or smaller.

m=0: Default compatibility mode (Margin or Axis absolute depending on presence of free axis).
m=1: Margin absolute.
m=2: Margin scaled.
m=3: Axis absolute.
m=4: Axis scaled.

The absolute modes are measured in points whereas scaled modes have similar values but automatically expand or contract as the axis font height changes. Mode 0 is the default and results in no change relative to previous versions of Igor Pro that used lblMargin unless a given side used a free axis in which case it used lblPos in absolute mode. The margin modes measure relative to an edge of the graph while the axis modes measure relative to the position of the axis. When using stacked axes, use either margin modes. With multiple nonstacked axes, use Axis scaled if the graph edge is not using a fixed margin or use axis absolute if it is.

lblRot=*r* Rotates the axis label by *r* degrees. *r* is a value from -360 to 360. Rotation is counterclockwise and starts from the label's normal orientation.

linTkLabel=*tl* *tl*=1 attaches the data units with any exponent or prefix to each tick label on a normal axis. *tl*=0 removes them.

log=*l*
- *l*=0: Normal axis.
- *l*=1: Log base 10.
- *l*=2: Log base 2.

logHTrip=*h* Same as highTrip but for log axes.

logLabel=*l* Sets the maximum number of decades in a log axis before minor tick labels are suppressed.

logLTrip=*l* Same as lowTrip but for log axes.

loglinear=*l* Switches to a linear tick method (*l*=1) on a log axis if the number of decades of ranges is less than 2. It switches to a linear tick exponent method if the number of decades is greater than five.

logTicks=*t* Sets the maximum number of decades in log axis before minor ticks are suppressed.

lowTrip=*l* If the extrema of an axis are between its lowTrip and its highTrip then tick mark labels use fixed point notation. Otherwise they use exponential (scientific or engineering) notation.

manminor={*number*, *emphasizeEvery*}
Specifies how to draw minor ticks in manual tick mode. There will be *number* ticks between each major (labeled) tick. You will usually want to set this to 4 to make 5 divisions, or 9 to make 10 divisions. A medium-sized tick (an emphasized minor tick) will be drawn every *emphasizeEvery* minor tick.

manTick={*cantick*, *tickinc*, *exp*, *digitsrt* [, *timeUnit*]}
Turns on manual tick mode. The tick from which all other ticks are calculated is the cononic tick (*cantick*). The numerical spacing between ticks is set by *tickinc*. *cantick* and *tickinc* are multiplied by 10*exp*. The number of digits to the right of the decimal point displayed in the tick labels is set by *digitsrt*.

The optional parameter *timeUnit* is used with Date/Time axes to specify the units of tickinc. In this case, tickinc must be an integer. The value of *timeUnit* is one of the following keywords:

```
second, minute, hour, day, week, month, year
```

On a date/time axis, the *exp* and *digitsrt* keywords are ignored, but must be present. You can set them to zero.

manTick=0 Turns off manual tick mode.

margin=*m* Sets a fixed margin from the edge of the window to the axis in points. Used principally to make axes of multiple graphs on a page line up when "stacked". You can use the left, right, bottom, and top axis names (even if an axis with that name doesn't exist) to adjust the graph plot area. See **Types of Axes** on page II-239.
- *m*=0: Sets "automatic" margin size (dependent on the length and height of tick marks and labels).
- *m*=-1: Sets the margin to "none", or 0. The axis is drawn at the graph window's edge.

minor=*m* Disables (*m*=0) or enables (*m*=1) minor ticks.

mirror=*m*
- *m*=1: Right axis mirroring left or top mirroring bottom.
- *m*=2: Mirror axis without tick marks.
- *m*=3: Mirror axis with tick marks and tick labels.
- *m*=0: No mirroring.

mirrorPos=*pos*
: Specifies the position of the mirror axis relative to the normal position. *pos* is a value between 0 and 1.

noLabel=*n*
: *n*=0: Normal labels.
: *n*=1: Suppresses tick mark labels.
: *n*=2: Suppresses tick mark labels and axis labels.

notation=*n*
: Uses engineering (*n*=0) or scientific (*n*=1) notation for tick mark labels.
: Affects tick mark labels displayed exponentially. See highTrip and lowTrip. Does not affect log axes.

nticks=*n*
: Specifies the approximate number of ticks marks (*n*) on axis.

prescaleExp=*exp*
: Multiplies axis range by 10^*exp* for tick labeling and *exp* is subtracted from the axis label exponent. In other words, the exponent is moved from the tick labels to the axis label. (This affects the display only, not the source data.)

sep=*s*
: Specifies the minimum number of screen points (s) between minor ticks.

standoff=*s*
: Suppresses (*s*=0) or enables (*s*=1) axis standoff.
: Axis standoff prevents waves or markers from covering the axis.

stLen=*p*
: Sets the length of minor ("small") tick marks to *p* points. If *p* is zero, it uses the default length. *p* may be fractional.

stThick=*p*
: Sets the thickness of minor ("small") tick marks to *p* points. If *p* is zero, it uses the default thickness. *p* may be fractional.

tick=*t*
: Sets tick position.
: *t*=0: Outside axis.
: *t*=1: Crossing axis.
: *t*=2: Inside axis.
: *t*=3: None.
: In a category plot, adding 4 to the usual values for the tick keyword will place the tick marks in the center of each category rather than at the edges.

tickEnab={*lowTick*,*highTick*}
: Restricts axis ticking to a subrange of normal. Ticks are drawn and labelled only if they fall within this inclusive numerical range.

tickExp=*te*
: *te*=1 forces tick labels to exponential notation when labels have units with a prefix. *te*=0 turns this off.

tickUnit=*tu*
: Suppresses (*tu* =1) or turns on (*tu* =0) units labels attached to tick marks.

tickZap={[*v1* [,*v2* [,*v3*]]]}
: Suppresses drawing of the tick mark label for values given in the list. This is useful when you have crossing axes to prevent tick mark labels from overlapping. The list may contain zero, one, two or three values. The values must be exact to suppress the label.

tkLblRot=*r*
: Rotates the tick mark labels by *r* degrees. *r* is a value from -360 to 360. Rotation is counterclockwise and starts from the label's normal orientation.

tlOffset=*o*
: Offsets the tick mark labels by *o* fractional points relative to the default tick mark label position. Positive is away from the axis.

ttLen=*p*
: Sets the length of subminor ("tiny") tick marks to *p* points. If *p* is zero, it uses the default length. *p* may be fractional. Subminor ticks are used only in log axes.

ttThick=*p*
: Sets the thickness of subminor ("tiny") tick marks to *p* points. If *p* is zero, it uses the default thickness. *p* may be fractional.

userticks={*tickPosWave*, *tickLabelWave*}
: Draws axes with purely user-defined tick mark positions and labels. *tickPosWave* is a numeric wave containing the desired positions of the tick marks, and *tickLabelWave* is a text wave containing the labels. See **User Ticks from Waves** on page II-274 for an example.
: The tick mark labels can be multiline and use styled text. For more details, see **Fancy Tick Mark Labels** on page II-312.
: *tickPosWave* need not be monotonic. Igor will plot a tick if a value is in the range of the axis. Both linear and log axes are supported.

Graph margins will adjust to accommodate tick labels. This will not prevent overlap between labels, which you will need fix yourself.

useTSep=*t* — *t*=1 displays a thousand's separator character between every group of three digits in the tick mark label (e.g., "1,000" instead of "1000"). The default is *t*=0.

zapLZ=*t* — Removes (*t*=1) leading zeros from tick mark labels. For example 0.5 becomes .5 and -0.5 becomes -.5. Default is *t*=0.

zapTZ=*t* — Removes (*t*=1) trailing zeros from tick mark labels. The the radix point will also be removed if all digits are zero. Default is *t*=0.

zero=*z* — *z*=1: Zero line at x=0 or y=0.

z=0: No zero line.

zeroThick=*zt* — Sets the thickness of the zero line in points, from 0.0 to 5.0 points. *zt*=0.0 means the zero line thickness automatically follows the thickness of the axis; this is the default. You can use 0.1 for a thin zero line thickness.

ZisZ=*t* — *t*=1 uses the single digit 0 as the zero tick mark label (if any) regardless of the number of digits used for other labels. Default is *t*=0.

Flags

/W=*winName* — Modifies the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/Z — Does not generate an error if the named axis does not exist in a style macro.

Details

With the prescaleExp parameter, you can force tick and axis label scaling to values different from the defaults. For example, if you have data whose X scaling ranges from 9pA to 120pA and you display this on a log axis, the tick marks will be labelled 10pA and 100pA. But if you really want the tick marks labeled 10 and 100 with pA in the axis label, you can set the prescaleExp to 12. To see this, execute the following commands:

```
Make/O jack=x
Display jack
SetScale x,9e-12,120e-12,"A",jack
ModifyGraph log(bottom)=1
```

then execute:

```
ModifyGraph prescaleExp(bottom)=12
```

The tickExp parameter applies to units that do not traditionally use SI prefix characters. For example, one usually speaks of 10^{-3} Torr and not mTorr. To see how this feature works, execute the following example commands:

```
Make/O jack=x
Display jack
SetScale x,1E-7,1E-5,"Torr",jack
ModifyGraph log(bottom)=1
```

then execute:

```
ModifyGraph tickExp(bottom)=1
```

at this point, the tick mark labels have Torr in them. If you want to eliminate the units from the tick marks, execute:

```
ModifyGraph tickUnit(bottom)=1
```

and if you now want Torr in the label string, use the \U escape in the label string:

```
Label bottom "\\U"
```

To see the effect of linTkLabel, execute these commands:

```
Make/O jack=x
Display jack
SetScale x,1E-7,1E-5,"Torr",jack
```

then execute:

```
ModifyGraph linTkLabel(bottom)=1
```

and then try:

```
ModifyGraph tickExp(bottom)=1
```

and finally:

```
ModifyGraph tickUnit(bottom)=1
```

ModifyGraph *(colors)*

ModifyGraph [**/W=winName/Z**] ***key*** [**(*axisName*)**] **= (*r*,*g*,*b*)**
    [**, *key*** [**(*axisName*)**] **= (*r*,*g*,*b*)**]…

This section of ModifyGraph relates to modifying the use of colors in a graph.

Parameters

Most (but not all) of the *key* parameters may take an optional *axisName* enclosed in parentheses. *axisName* is "left", "right", "top", "bottom" or the name of an free axis such as "vertCrossing".

Where the parameter descriptions indicate an "(*axisName*)", it may be omitted to change all axes in the graph.

r, *g*, and *b* are each an integer from 0 to 65535 where (0, 0, 0) is black and (65535, 65535, 65535) is white.

Parameter Specification	Object Colored
`alblRGB(`*`axisName`*`)=(`*`r`*`,`*`g`*`,`*`b`*`)`	Axis labels
`axRGB(`*`axisName`*`)=(`*`r`*`,`*`g`*`,`*`b`*`)`	Axis
`cbRGB=(`*`r`*`,`*`g`*`,`*`b`*`)`	Control bar background
`gbRGB=(`*`r`*`,`*`g`*`,`*`b`*`)`	Graph background
`gridRGB(`*`axisName`*`)=(`*`r`*`,`*`g`*`,`*`b`*`)`	Axis grid lines
`tickRGB(axisName )=(`*`r`*`,`*`g`*`,`*`b`*`)`	Axis Tick marks
`tlblRGB(`*`axisName`*`)=(`*`r`*`,`*`g`*`,`*`b`*`)`	Axis Tick labels
`wbRGB=(`*`r`*`,`*`g`*`,`*`b`*`)`	Window background

Flags

/W=*winName* — Modifies the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/Z — Does not generate an error if the named axis does not exist in a style macro.

Details

On Windows, use maximum white to set the control bar background color to track the 3D Objects color in the Appearance Tab of the Display Properties control panel:

```
ModifyGraph cbRGB=(65535,65535,65535)
```

See Also

See **Instance Notation** on page IV-16.

ModifyImage

ModifyImage [**/W=*winName***] ***imageInstance, keyword = value***
    [**, *keyword* = *value***]…

The ModifyImage operation changes properties of the given image in the top graph (or the specified graph if /W is used). *imageInstance* is the name of the image to be altered. This name is usually simply the name of the matrix wave containing the image data. If the same matrix wave is displayed more than once, you must append #0, #1 etc. to the name to distinguish which is which.

imageInstance can also take the form of a null name with an instance number to affect the instanceth image. That is,

```
ModifyImage ''#1
```

modifies the appearance of the second image that was appended to the top graph, no matter what the image names are. Note: two single quotes are used, not a double quote.

Parameters

Here are the keyword-value pairs. These apply to false color images in which the data in the matrix is used as an index into a color table. They do not apply to direct color images in which the data in the matrix specifies the color directly.

cindex=*matrixWave* Sets the Z value mapping mode such that image colors are determined by doing a lookup in the specified matrix wave.

matrixWave is a 3 column wave that contains red, green, and blue values from 0 to 65535. (The matrix can actually have more than 3 columns. It ignores any extra columns.)

The color at Z=z is determined by finding the RGB values in the row of *matrixWave* whose scaled X index is z. In other words, the red value is *matrixWave*(z)[0], the green value is *matrixWave*(z)[1] and the blue value is *matrixWave*(z)[2].

If *matrixWave* has default X scaling, where the scaled X index equals the point number, then row 0 contains the color for Z=0, row 1 contains the color for Z=1, etc.

If you use cindex, you should not use ctab in the same command.

ctab={*zMin*, *zMax*, *ctName*, *reverse*}

Sets z mapping mode by which values in the matrix are mapped linearly into the color table specified by *ctName*. The color table name can be missing if you want to leave it as is. *zMin* and *zMax* set the range of z values to map. Omit *zMin* or *zMax* to leave as is or use * to autoscale. *ctName* can be any color table name returned by the **CTabList** function, such as Grays or Rainbow. Also see **Color Tables** on page II-349. Set parameter *reverse* to 1 to reverse the color table; zero or missing does not reverse the color table.

ctabAutoscale=*autoBits*

Sets the range of data used for autoscaling ctab * values.

Bit 0: Autoscales only the XY subset being displayed.

Bit 1: Autoscales only the current plane being displayed.

If neither bit is set (if *autoBits* = 0, the default), then all of the data in the image wave is used to autoscale the *'d *zMin*, *zMax* values for ctab.

eval={*value*, *red*, *green*, *blue*}

If the red, green, and blue values are in the valid range for a color value (0 to 65535) the explicit value-color pair is added (or updated if value already exists). If the color values are out of range (-1 is suggested) then the value is removed from the list if it is present (no error if it is not).

explicit=1 *or* 0 Turns explicit (monochrome) mode on (1) or off (0). Meant to be used with unsigned byte data but will do the best it can for other types. If value of data is equal to one of the defined explicit values then its defined color is used otherwise the pixel will be blank. The default predefined values are:

255	black
0	white

You can add, change, or delete explicit values with the eval keyword.

imCmplxMode=*m* Sets complex data display mode.

m=0: Magnitude (default).

m=1: Real only.

m =2: Imaginary only.

m =3: Phase in radians.

interpolate= 1 Turns on smoothing of the boundaries between pixels. Since this is implemented via system graphics calls and not by Igor actually doing the interpolation, it will not affect EPS or EMF export on Windows and will not affect EPS export on Mac. Although this may create a more esthetically pleasing display, it is not clear that it is appropriate for scientific data.

log= 1 or 0 0 sets the default linearly-spaced false-image colors.

1 turns on logarithmically-spaced false-image colors. This requires that the image values be greater than 0 to display correctly.

Affects the image colors for color table and color index images only (see **Color Table Details** on page II-353 and **Indexed Color Details** on page II-356).

The log keyword was added in Igor Pro 6.22.

lookup= *waveName* Specifies an optional 1D wave that can be used to modify the mapping of scaled z values into the color table specified with the ctab parameter. Values should range from 0.0 to 1.0. A linear ramp from 0 to 1 would have no effect while a ramp from 1 to 0 would reverse the image. Used to apply gamma correction to grayscale images or for special effects. Use a NULL wave ($"") to remove the option.

maxRGB=(*red*, *green*, *blue*)

Sets the color of image values greater than the ctab *zMax* or greater than the cindex of the *matrixWave* maximum X scaling value. Also turns max color mode on.

The *red*, *green*, and *blue* color values are in the range of 0 to 65535.

maxRGB=1 *or* 0 *or* NaN

Turns max color mode off, on, or transparent. These modes affect the display of image values greater than the ctab *zMax* or greater than the cindex of the *matrixWave* maximum X scaling value.

1: Turns on max color mode. The color of the affected image pixels is black or the last color set by maxRGB=(*red*, *green*, *blue*).

0: Turns off max color mode (default). The color of the affected image pixels is the last color table or color index color.

NaN: Transparent max color mode. The affected image pixels are not drawn.

minRGB=(*red*, *green*, *blue*)

Sets the color of image values less than the ctab *zMin* or less than the cindex of the *matrixWave* minimum X scaling value. Also turns min color mode on.

The *red*, *green*, and *blue* color values are in the range of 0 to 65535.

minRGB=1 *or* 0 *or* NaN

Turns min color mode off, on, or transparent. These modes affect the display of image values less than the ctab *zMin* or less than the cindex of the *matrixWave* minimum X scaling value.

1: Turns on min color mode. The color of the affected image pixels is black or the last color set by minRGB=(*red*, *green*, *blue*).

0: Turns off min color mode (default). The color of the affected image pixels is the first color table or color index color.

NaN: Transparent min color mode. The affected image pixels are not drawn.

plane=*p* Determines which part of a 3D or 4D image wave to display.

The meaning of *p* depends on the nature of the image wave. If the size of the layer dimension of the image wave is exactly three then the wave is treated as RGB data with R, G, and B data in the three layers. Otherwise each layer of the wave is treated as a separate grayscale image.

Plane=p With RGB Data

If the wave is 3D, plane=*p* has no effect.

If the wave is 4D, each chunk contains a different set of R, G and B layers and *p* selects which chunk to display.

Plane=p With Grayscale Data

If the wave is 3D, *p* selects which layer to display.

If the wave is 4D, plane=*p* acts as if all of the chunks were combined into a virtual 3D wave and *p* selects which layer of this virtual 3D wave to display.

rgbMult=*m* If *m* is non-zero, direct color values (3 plane RGB) are multiplied by *m*. This would typically be used for 10, 12 or 14 bit integers in a 16 bit word. For example, if your image data is 14 bits, use rgbMult=4.

Flags

/W=*winName* Directs action to a specific window or subwindow rather than the top graph window. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

See Also
AppendImage and **RemoveImage**.

ModifyLayout

ModifyLayout [***flags***] ***key*** [**(*objectName*)**] **=*value*** [**, *key*** [**(*objectName*)**] **=*value***]…

The ModifyLayout operation modifies objects in the top layout or in the layout specified by the /W flag.

Parameters

Each *key* parameter may take an optional *objectName* enclosed in parentheses. If "(*objectName*)" is omitted, all objects in the layout are affected.

Though not shown in the syntax, the optional "(*objectName*)" may be replaced with "[*objectIndex*]", where *objectIndex* is zero or a positive integer denoting the object to be modified. "[0]" denotes the first object appended to the layout, "[1]" denotes the second object, etc. This syntax is used for style macros, in conjunction with the /Z flag.

The parameter descriptions below omit the optional "(*objectName*)".

The "units", "mag" and "bgRGB" keywords apply to the layout as a whole, not to a specific object and do not accept an *objectName*.

bgRGB=(*r*,*g*,*b*)	Specifies the background color for the layout. *r*, *g*, and *b* are integers from 0 to 65535.	
columns=*c*	Specifies the number of columns for a table object.	
fidelity=*f*	*f*=0:	Low fidelity.
	f=1:	High fidelity.
frame=*f*	Specifies the type of frame enclosing the object.	
	f=0:	No frame.
	f=1:	Single frame.
	f=2:	Double frame.
	f=3:	Triple frame.
	f=4:	Shadow frame.
height=*h*	Sets the height of the object.	
left=*l*	*l* is the horizontal coordinate of the left edge of the object relative to the left edge of the paper.	
mag=*m*	Sets the layout magnification where *m*=0.25, 0.5, 1, or 2.	
rows=*r*	Specifies the number of rows for table object.	
top=*t*	*t* is the vertical coordinate of the top edge of the object relative to the top edge of the paper.	
trans=*t*	*t*=0:	Opaque (default).
	t=1:	Transparent. For this to be effective, the object itself must also be transparent. Annotations have their own transparent/opaque settings. Graphs are transparent only if their backgrounds are white. PICTs may have been created transparent or opaque, and Igor cannot make an opaque PICT transparent.
units=*u*	Sets dimension units in the layout info panel and in the Modify Objects dialog.	
	u=0:	Points.
	u=1:	Inches.
	u=2:	Centimeters.
width=*w*	Sets the object width.	

Flags

/I	Dimensions in inches.
/M	Dimensions in centimeters.
/W=*winName*	*winName* is the name of the page layout window to be modified. If /W is omitted or if *winName* is `$""`, the top page layout is modified.
/Z	Does not generate an error if the indexed or named object does not exist in a style macro.

The /I and /M flags affect the units of the parameters for the left, top, width and height keywords only. If neither /I nor /M is present then the parameters for the left, top, width and height keywords are points.

Details

Note that the units keyword affects only the units used in the layout info panel and in the Modify Objects dialog. It has nothing to do with the units used for the left, top, width and height keywords. Those units are points unless the /I or /M flags is present.

See Also

NewLayout, **AppendLayoutObject** and **RemoveLayoutObjects**.

ModifyPanel

`ModifyPanel [/W=winName] keyword = value [, keyword = value …]`

The ModifyPanel operation modifies properties of the top or named control panel window or subwindow.

Parameters

keyword is one of the following:

cbRGB=(*r*,*g*,*b*) Specifies the background color of the entire control panel or the graph's control bar area. *r*, *g*, and *b* are values from 0 to 65535.

fixedSize=*f*

- *f*=0: Panel can be resized (default).
- *f*=1: Panel cannot be resized by adjusting the size box or frame (nor maximized on Windows), but the window can be minimized (on Windows) and the MoveWindow operation can still change the size.

frameInset= *i* Specifies the number of pixels by which to inset the frame of the panel subwindow. Mostly useful for overlaying panels in graphs to give a fake 3D frame a better appearance.

frameStyle= *f* Specifies the frame style for a panel subwindow. frameStyle values are:

- 0: None.
- 1: Single.
- 2: Indented.
- 3: Raised.
- 4: Text well.

The last three styles are fake 3D and will look good only if the background color of the enclosing space and the panel itself is a light shade of gray.

noEdit= *e* Sets the editability of the panel.

- *e*=0: Editable (default).
- *e*=1: Not editable. For a panel window, the Panel menu item is not present and the ShowTools command is ignored. For a panel subwindow, it can not be activated by clicking.

Flags

/W= *winName* Modifies the control panel in the named graph or control panel window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

On Windows, set *r*, *g*, and *b* = 65535 (maximum white) to set the background color of the control panel to track the 3D Objects color in the Appearance Tab of the Display Properties control panel.

See Also

The **NewPanel** operation.

Controls in Graphs on page III-388.

ModifyTable

ModifyTable [**/W=*winName*/Z**] ***key*** [**(*columnSpec*)**] **=*value*** [**, *key*** [**(*columnSpec*)**] **=*value***]…

The ModifyTable operation modifies the appearance the top or named table window or subwindow.

Parameters

Many of the parameter keywords take an optional *columnSpec* enclosed in parentheses. Usually *columnSpec* is simply the name of a wave displayed in the table. All table columns are affected when you omit `(columnSpec)`.

More precisely, column specifications are wave names for waves in the current data folder or data folder paths leading to waves in any data folder optionally followed by the suffixes .i, .l, .d, .id or .ld to specify dimension indices, dimension labels, data values, dimension indices and data values, or dimension labels and data values of the wave. For example, `ModifyTable font(myWave.i)="Helvetica"`. If the wave is complex, the column specification may be followed by `.real` or `.imag` suffixes.

One additional *columnSpec* is Point, which refers to the first column containing the dimension index numbers. If multidimensional waves are displayed in the table, this column may have the title "Row", "Column", "Layer", "Chunk" or "Element", but the *columnSpec* for this column is always Point. See **Column Names** on page II-198 for details.

Though not shown in the syntax, the optional `(columnSpec)` may be replaced with `[columnIndex]`, where *columnIndex* is zero or a positive integer denoting the column to be modified. `[0]` denotes the Point column, `[1]` denotes the first column appended to the table, `[2]` denotes the second appended column, etc. This syntax is used for style macros, in conjunction with the /Z flag.

As of Igor Pro 6 you can use a range of column numbers instead of just a single column number, for example `[0,3]`.

The parameter descriptions below omit the optional (*columnSpec*).

alignment=*a*

a=0: Left aligned.

a=1: Center aligned.

a=2: Right aligned.

autosize={*mode*, *options*, *padding*, *perColumnMaxSeconds*, *totalMaxSeconds*}

Autosizes the specified column or columns.

mode=0: Sets width of each data column from a given multidimensional wave individually.

mode=1: Sets width of all data columns from a given multidimensional wave the same.

options is a bitwise parameter. Usually 0 is the best choice.

bit 0: Ignores column names.

bit 1: Ignores horizontal indices.

bit 2: Ignores data cells.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

padding specifies extra padding for each column in points. Use -1 to get the default amount of padding (16 points).

perColumnMaxSeconds specifies the maximum amount of time to spend autosizing a single column. Use 0 to get the default amount of time (one second).

totalmaxSeconds specifies the maximum amount of time for autosizing the entire table. Use 0 to get the default amount of time (ten seconds).

digits=*d*

Specifies the number of digits after decimal point or, for hexadecimal and octal columns, the number of total digits.

elements=(*row*, *col*, *layer*, *chunk*)

Selects the view of a multidimensional wave in the table. The values given to *row*, *col*, *layer*, and *chunk* specify how to change the view.

-1: No change from current view.

-2: Display this dimension vertically.

-3: Display this dimension horizontally.

≥0: For waves with 3 or 4 dimensions, display this element of the other dimensions.

See **ModifyTable Elements Command** on page II-221 for a detailed discussion of this keyword.

entryMode=*m* Queries or sets the table's entry line mode.

m=0: Just queries.

m=1: Accepts any entry that was started if possible.

m=2: Cancels any entry that was started if possible.

If *m* is 0 then the entry line state is not changed but is returned via V_flag as follows:

0: No entry is in progress.

-1: An entry is in progress and is valid.

Other: An entry is in progress and is invalid.

If *m* is 1 then the entry is accepted if it is valid and its state is returned via V_flag as follows:

0: No entry is in progress.

-1: The entry was accepted.

Other: The entry is invalid and was not accepted.

If *m* is 2 then the entry is cancelled if possible and its state is returned via V_flag as follows:

0: No entry is in progress.

-1: The entry was cancelled.

font="*fontName*" Sets font used in the table, e.g., `font="Helvetica"`.

format=*f* Sets the data format for the table.

f=0: General.

f=1: Integer.

f=2: Integer with thousands (e.g., "1,234").

f=3: Fixed point (e.g., "1234.56").

f=4: Fixed point with thousands (e.g., "1,234.56").

f=5: Exponential (scientific only).

f=6: Date format.

f=7: Time format (always 24 hour time).

f=8: Date&time format (date followed by time).

f=9: Octal.

f=10: Hexadecimal.

As of Igor Pro 6, you cannot apply date or date&time formats to a wave that is not double-precision (see **Date, Time, and Date&Time Units** on page II-85). To avoid this error, use **Redimension** to change the wave to double-precision.

frameInset= *i* Specifies the number of pixels by which to inset the frame of the table subwindow.

frameStyle= *f* Specifies the frame style for a table subwindow. frameStyle values are:

0: None

1: Single

2: Double

3: Triple

4: Shadow

5: Indented

6: Raised

7: Text well

The last three styles are fake 3D and will look good only if the background color of the enclosing space and the table itself is a light shade of gray.

horizontalIndex=*h* Controls what is displayed in the horizontal index row when multidimensional waves are displayed.

h=0: Displays dimension labels if the multidimensional wave's label column is displayed, otherwise displays numeric indices (default).

h=1: Always displays numeric indices for multidimensional waves.

h=2: Always displays dimension labels for multidimensional waves.

The horizontal index row appears below the row of column names if the table contains a multidimensional wave. Use horizontalIndex to override the default behavior in order to display labels for the horizontal dimension while displaying numeric indices for the vertical dimension or vice versa.

horizontalIndex controls the horizontal index row only. To control what is displayed vertically, use **AppendToTable** to append a numeric index or dimension label column.

rgb=(*r*, *g*, *b*) Sets color of text. *r*, *g*, and *b* are red, green, and blue components of the color and range from 0 to 65,535. Default is black: (0,0,0).

selection=(*firstRow*, *firstCol*, *lastRow*, *lastCol*, *targetRow*, *targetCol*)
Sets the selected cells in the table.

If any of the parameters have the value -1 then the corresponding part of the selection is not changed.

Otherwise they set the first and last selected cell and the target cell. Row and column values are 0 or greater. The Point column can not be selected.

The proposed parameters are clipped to avoid invalid combinations, such as the last selected row being before the first selected row.

With one exception, it does not support selecting unused cells. Therefore the proposed selection is clipped to prevent this. The exception is that, if the parameters call for selecting the first cell in the first unused column, then this is permitted.

showFracSeconds=*s* Shows (*s*=1) or hides (*s*=0; default) fractional seconds.

showParts=*parts* Specifies what elements of the table should be visible. Other elements are hidden.

parts is a bitwise parameter specifying what to show.

bit 0: Entry line and other top line controls.

bit 1: Name row.

bit 2: Horizontal index row.

bit 3: Point column.

bit 4: Horizontal scroll bar.

bit 5: Vertical scroll bar.

bit 6: Selection highlighting.

bit 7: Insertion cells.

All other bits are reserved and must be set to zero except that you can pass -1 to indicate that you want to show all parts of the table.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Presentation tables in subwindows in graphs and page layouts do not have an entry line or scroll bars and therefore never show these items.

See **Parts of a Table** on page II-192 and **Showing and Hiding Parts of a Table** on page II-193 for further information.

sigdigits=*d* *d* is the number of significant digits when the numeric format is general.

size=*s* Font size, e.g., `size=14`.

style=*n* *n* is a binary coded number with each bit controlling one aspect of the column's font style as follows:

bit 0: Bold.

bit 1: Italic.

bit 2: Underline.

bit 3: Outline (*Macintosh only*).

bit 4: Shadow (*Macintosh only*).

bit 5: Condensed (*Macintosh only*).

bit 6: Extended (*Macintosh only*).

For example, bold underlined is $2^0 + 2^2 = 1 + 4 = 5$. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

title="*title*" Sets the title of a column to *title*.

topLeftCell=(*row*, *column*)
Scrolls the table contents so that the cell identified by (*row*, *column*) is the top left visible data cell, or as close as possible.

If *row* is -1 then the table's vertical scrolling is not changed. If *column* is -1 then the table's horizontal scrolling is not changed.

If they are positive, *row* and *column* are zero-based numbers which are clipped to valid values before being used. *row*=0 refers to the first row of data in the table, *column*=0 refers to the first column of data.

The Point column can not be scrolled horizontally.

trailingZeros=*t* Shows trailing zeros (*t*=1). This affects the general numeric format only.

width=*w* Sets column width to *w* points.

You will not always get the exact number of points that you request. This is because a column must have an even number of screen pixels, so that grid lines look good. Igor will modify your requested number of points to meet this requirement.

Flags

/W= *winName* Modifies the named table window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/Z No errors generated if the indexed or specified column does not exist in a style macro.

Examples

```
ModifyTable size(myWave)=14      // change font size of myWave column
ModifyTable width(Point)=0                  // hide Point column
ModifyTable style(cmplxWave.imag)=32        // condensed= bit 5 = 2^5 = 32
```

See Also

See **Column Names** on page II-198 and **ModifyTable Elements Command** on page II-221.

ModifyWaterfall

ModifyWaterfall [**/W=*winName***] ***keyword* = *value*** [**, *keyword* = *value*** …]

The ModifyWaterfall operation modifies the properties of the waterfall plot in the top or named graph.

Parameters

keyword is one of the following:

angle= *a* Angle in degrees from horizontal of the angled Y axis (*a* =10 to 90).

axlen= *len* Relative length of angled Y axis. *len* is a fraction between 0.1 and 0.9.

hidden= *h*
- *h* =0: Turns hidden lines off.
- *h* =1: Uses painter's algorithm.
- *h* =2: True hidden.
- *h* =3: Hides lines with bottom removed.
- *h* =4: Hides lines using a different color for the bottom. When specified, the top color is the normal color for lines and the bottom color is set using `ModifyGraph negRGB=(r,g,b)`.

Hidden lines are active only when the mode is lines between points.

Flags

/W= *winName* Modifies waterfall plot in the named graph window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

Painter's algorithm draws the traces from back to front and erases hidden lines while modes 2, 3 and 4 detect which line segments are hidden and suppresses the drawing of these segments.

See Also

Waterfall Plots on page II-296.

The **NewWaterfall** and **ModifyGraph** operations.

ModuleName

#pragma ModuleName = *modName*

The ModuleName pragma assigns a name, which must be unique, to a procedure file so that you can use static functions and Proc Pictures in a global context, such as in the action procedure of a control or on the Command Line.

Using the ModuleName pragma involves at least two steps. First, within the procedure file assign it a name using `#pragma ModuleName=`*`modName`*, and then access objects in the named file by preceding the object name with the name of the module and the # character, such as or example: `ModName#StatFuncName()`.

See Also

The **Regular Modules** on page IV-212, **Static**, **Picture**, and **#pragma**.

MoveDataFolder

MoveDataFolder *sourceDataFolderSpec*, *destDataFolderPath*

The MoveDataFolder operation removes the source data folder (and everything it contains) and places it at the specified location with the original name.

Parameters

sourceDataFolderSpec can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

destDataFolderPath can be a partial path (relative to the current data folder) or an absolute path (starting from root).

Details

MoveDataFolder generates an error if a data folder of the same name already exists at the destination.

Examples

Move data folder `foo` into data folder `bar`:

```
MoveDataFolder foo,root:bar:
```

Move data folder `foo` into data folder `bar`:

```
MoveDataFolder foo,:bar:
```

See Also

See the **DuplicateDataFolder** operation. Chapter II-8, **Data Folders**.

MoveFile

MoveFile [***flags***] [***srcFileStr***] [**as** ***destFileOrFolderStr***]

The MoveFile operation moves or renames a file on disk. A file is renamed by "moving" it to the same folder it is already in using a different name.

Parameters

srcFileStr can be a full path to the file to be moved or renamed (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*.

If Igor can not determine the location of the file from *srcFileStr* and *pathName*, it displays an Open File dialog allowing you to specify the source file.

destFileOrFolderStr is interpreted as the name of (or path to) an existing folder when /D is specified, otherwise it is interpreted as the name of (or path to) a possibly existing file.

If *destFileOrFolderStr* is a partial path, it is relative to the folder associated with *pathName*.

If /D is specified, the source file is moved inside the folder using the source file's name.

If Igor can not determine the location of the destination file from *pathName*, *srcFileStr*, and *destFileOrFolderStr*, it displays a Save File dialog allowing you to specify the destination file (and folder).

If you use a full or partial path for either *srcFileStr* or *destFileOrFolderStr*, see **Path Separators** on page III-398 for details on forming the path.

Folder paths should not end with single Path Separators. See the **Details** section for **MoveFolder**.

Flags

/D — Interprets *destFileOrFolderStr* as the name of (or path to) an existing folder (or directory). Without /D, *destFileOrFolderStr* is the name of (or path to) a file.

If *destFileOrFolderStr* is not a full path to a folder, it is relative to the folder associated with *pathName*.

/I [=*i*] — Specifies the level of interactivity with the user.

- /I=0: Interactive only if one or *srcFileStr* or *destFileOrFolderStr* is not specified or if the source file is missing. (Same as if /I was not specified.)
- /I=1: Interactive even if *srcFileStr* is specified and the source file exists.
- /I=2: Interactive even if *destFileOrFolderStr* is specified.
- /I=3: Interactive even if *srcFileStr* is specified and the source file exists. Same as /I only.

/M=*messageStr* — Specifies the prompt message in the Open File dialog. If /S is not specified, then *messageStr* will be used for both Open File and for Save File dialogs.

/O — Overwrite existing destination file, if any. Without /O, the user is asked if replacing the existing file is to be allowed.

/P=*pathName* — Specifies the folder to look in for the source file, and the folder into which the file is copied. *pathName* is the name of an existing symbolic path.

Using /P means that both *srcFileStr* and *destFileOrFolderStr* must be either simple file or folder names, or paths relative to the folder specified by *pathName*.

/S=*saveMessageStr* — Specifies the prompt message in the Save File dialog.

/Z[=*z*] — Prevents procedure execution from aborting if it attempts to move a file that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort.

- /Z=0: Same as no /Z at all.
- /Z=1: Moves a file only if it exists. /Z alone has the same effect as /Z=1.
- /Z=2: Moves a file if it exists or displays a dialog if it does not exist.

Variables

The MoveFile operation returns information in the following variables:

`V_flag` — Set to zero if the file was moved, to -1 if the user cancelled either the Open File or Save File dialogs, and to some nonzero value if an error occurred, such as the specified file does not exist.

`S_fileName` — Stores the full path to where the file was moved from. If an error occurred or if the user cancelled, it is set to an empty string.

`S_path` — Stores the full path where the file was moved to. If an error occurred or if the user cancelled, it is set to an empty string.

Examples

Rename a file, using full paths:

```
MoveFile "HD:folder:aFile.txt" as "HD:folder:bFile.txt"
```

Rename a file, using a symbolic path:

```
MoveFile/P=myPath "aFile.txt" as "bFile.txt"
```

Move a file into a subfolder (the subfolder must exist):

```
MoveFile/D "Macintosh HD:folder:aFile.txt" as ":subfolder"
```

Move a file into an unrelated folder (the subfolder must exist):

```
MoveFile/D "Macintosh HD:folder:afile.txt" as "Server:archive"
```

Move a file from one folder to another and rename it:

```
MoveFile "Macintosh HD:folder:afile.txt" as "Server:archive:destFile.txt"
```

Move user-selected file into a particular folder:

```
MoveFile/D as "C:My Data:Selected Files Folder"
```

Move user-selected file in any folder as `bFile.txt` in same folder:

```
MoveFile as "bFile.txt"
```

Move user-selected file in any folder as `bFile.txt` in any folder:

```
MoveFile/I=2 as "bFile.txt"
```

See Also

The **Open**, **MoveFolder**, **CopyFolder**, **NewPath**, and **CreateAliasShortcut** operations. The **IndexedFile** function. **Symbolic Paths** on page II-34.

MoveFolder

MoveFolder [***flags***] [***srcFolderStr***] [**as** ***destFolderStr***]

The MoveFolder operation moves or renames a folder on disk. A folder is renamed by "moving" it into the same folder it is already in, but with a different name.

Warning: *The `MoveFolder` command can destroy data* by overwriting another folder and its contents!

If you overwrite an existing folder on disk, MoveFolder will do so only if permission is granted by the user. The default behavior is to display a dialog asking for permission. The user can alter this behavior via the Miscellaneous Settings dialog's Misc category. For further details see **Misc Settings** on page III-414.

If permission is denied, the folder will not be moved and V_Flag will return 1088 (Command is disabled) or 1275 (You denied permission to overwrite a folder). Command execution will cease unless the /Z flag is specified.

Parameters

srcFolderStr can be a full path to the folder to be moved or renamed (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a folder within the folder associated with *pathName*.

If the location of the source folder cannot be determined from *srcFolderStr* and *pathName*, it displays a Select Folder dialog allowing you to specify the source.

If /P=*pathName* is given, but *srcFolderStr* is not, then the folder associated with *pathName* is moved or renamed.

destFolderStr specifies the final location of the folder or, if /D is used, the parent of the final location of the folder.

destFolderStr can be a full path to the output (destination) folder (in which case /P is not needed), or a partial path relative to the folder associated with *pathName*.

If the location of the destination folder cannot be determined from *destFolderStr* and *pathName*, it displays a Save Folder dialog allowing you to specify the destination.

If you use a full or partial path for either file, see **Path Separators** on page III-398 for details on forming the path.

Flags

/D — Interprets *destFolderStr* as the name of (or path to) an existing folder (or "directory") to move the source folder into. Without /D, it interprets *destFolderStr* as the name of (or path to) the moved folder.

If *destFolderStr* is not a full path to a folder, it is relative to the source folder.

/I [=*i*] — Specifies the level of interactivity with the user.

- /I=0: Interactive only if one or *srcFolderStr* or *destFolderStr* is not specified or if the source folder is missing. (Same as if /I was not specified.)
- /I=1: Interactive even if *srcFolderStr* is specified and the source file exists.
- /I=2: Interactive even if *destFolderStr* is specified.
- /I=3: Interactive even if *srcFolderStr* is specified and the source file exists. Same as /I only.

/M=*messageStr* — Specifies the prompt message in the Open File dialog. If /S is not used, then *messageStr* will be used for both Open File and for Save File dialogs.

/O — Overwrite existing destination folder, if any. This deletes the existing destination folder. When /O is specified, the source folder can't be moved into an existing folder without specifying the name of the moved folder in *destFolderStr*.

/P=*pathName* — Specifies the folder for relative paths in *srcFolderStr* and *destFolderStr*. *pathName* is the name of an existing symbolic path.

If *srcFolderStr* is omitted, the folder associated with *pathName* is moved. If *destFolderStr* is omitted, the source folder is moved into the folder associated with *pathName*.

Using /P means that *srcFolderStr* (if specified) and *destFolderStr* must be either simple folder names or paths relative to the folder specified by *pathName*.

/S=*saveMessageStr* — Specifies the prompt message in the Save File dialog.

/Z[=*z*] — Prevents procedure execution from aborting if it attempts to move a file that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort.

/Z=0: Same as no /Z at all.

/Z=1: Moves a file only if it exists. /Z alone has the same effect as /Z=1.

/Z=2: Moves a file if it exists or displays a dialog if it does not exist.

Variables

The MoveFolder operation returns information in the following variables:

`V_flag` — Set to zero if the file was moved, to -1 if the user cancelled either the Open File or Save File dialogs, and to some nonzero value if an error occurred, such as the specified file does not exist.

`S_fileName` — Stores the full path to the folder that was moved, with a trailing semicolon. If an error occurred or if the user cancelled, it is set to an empty string.

`S_path` — Stores the full path of the moved folder, with a trailing semicolon. If an error occurred or if the user cancelled, it is set to an empty string.

Details

You can use only /P=*pathName* (omitting *srcFolderStr*) to specify the source folder to be moved.

A folder path should not end with single Path Separators. For example:

```
MoveFolder "Macintosh HD:folder" as "Macintosh HD:Renamed Folder:"
MoveFolder "Macintosh HD:folder:" as "Macintosh HD:Renamed Folder"
MoveFolder "Macintosh HD:folder:" as "Macintosh HD:Renamed Folder:"
```

will do weird, unexpected things (and probably damaging things when /O is also used). Instead, use:

```
MoveFolder "Macintosh HD:folder" as "Macintosh HD:Renamed Folder"
```

Beware of PathInfo and other command which return paths with an ending path separator. (They can be removed with the **RemoveEnding** function.)

A folder may not be moved into one of its own subfolders.

Conversely, the command:

```
MoveFolder/O/P=myPath "afolder"
```

which attempts to overwrite the folder associated with myPath with a folder that is inside it (namely "afolder") is not allowed. Instead, use:

```
MoveFolder/O/P=myPath "::afolder"
```

On Windows, renaming or moving a folder never updates the value of any Igor Symbolic Paths that point to a moved folder:

```
// Create a folder
NewPath/O/C myPath "C:\\My Data\\My Work"

// Move the folder
MoveFolder/P=myPath as "C:\\My Data\\Moved"

// Display the path's value
PathInfo myPath               // (or use the Path Status dialog)
Print S_Path
• C:My Data:My Work
```

You can use PathInfo to determine if a folder referred to by an Igor symbolic path exists and where it is on the disk. Use NewPath/O to reset the path's value.

On the Macintosh, however, renaming or moving a folder on the same volume does alter the value of symbolic path. This is because MoveFolder uses a Mac OS alias to keep track of the folder. A folder renamed or moved on the same volume retains the original "volume refnum" and "directory ID" stored in the alias mechanism, so that the alias (and hence Igor's symbolic path) remains pointing to the moved folder. After moving the folder, using the unchanged volume refnum and directory ID (in PathInfo or when you use /P=pathName) returns the updated path.

Moving the folder to a different volume actually creates a new folder with new volume refnum and directory IDs, and symbolic paths pointing to or into the moved folder aren't updated. They will be pointing at a deleted folder (they're probably invalid).

Examples

Rename a folder ("move" it to the same folder):

```
MoveFolder "Macintosh HD:folder" as "Macintosh HD:Renamed Folder"
```

Rename a folder referred to by only a path:

```
NewPath/O myPath "Macintosh HD:folder"
MoveFolder/P=myPath as "::Renamed Folder"
```

Move a folder from one volume to another. This moves "Macintosh HD:My Folder" inside "Server:My Folder" if "Server:My Folder" already exists:

```
MoveFolder "Macintosh HD:My Folder" as "Server:My Folder"
```

Move a folder from one volume to another. This overwrites "Server:My Folder" (if it existed) with the moved "Macintosh HD:My Folder":

```
MoveFolder/O "Macintosh HD:My Folder" as "Server:My Folder"
```

Move user-selected folder in any folder as "Renamed Folder" into a user-selected folder (possibly the same one):

```
MoveFolder as "Renamed Folder"
```

Move user-selected file in any folder as "Moved Folder" in any folder:

```
MoveFolder/I=3 as "Moved Folder"
```

See Also

MoveFile, **CopyFolder**, **IndexedDir**, **PathInfo**, and **RemoveEnding**. **Symbolic Paths** on page II-34.

MoveString

MoveString ***sourceString, destDataFolderPath*** [***newname***]

The MoveString operation removes the source string variable and places it in the specified location optionally with a new name.

Parameters

sourceString can be just the name of a string variable in the current data folder, a partial path (relative to the current data folder) and variable name or an absolute path (starting from root) and variable name.

destDataFolderPath can be a partial path (relative to the current data folder) or an absolute path (starting from root).

Details

An error is issued if a variable or wave of the same name already exists at the destination.

Examples

```
MoveString :foo:s1,:bar:      // Move string s1 into data folder bar
MoveString :foo:s1,:bar:ss1   // Move string s1 into bar with new name ss1
```

See Also

The **MoveVariable**, **MoveWave**, and **Rename** operations; andChapter II-8, **Data Folders**.

MoveSubwindow

MoveSubwindow [**/W=*winName***] ***key* = (*values*)**[**, *key* = (*values*)**]…

The MoveSubwindow operation moves the active or named subwindow to a new location within the host window. This command is primarily for use by recreation macros; users should use layout mode for repositioning subwindows.

Parameters

fguide=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the frame guide name(s) to which the outer frame of the subwindow is attached inside the host window.

The frame guides are identified by the standard names or user-defined names as defined by the host. Use * to specify a default guide name.

When the host is a graph, additional standard guides are available for the outer graph rectangle and the inner plot rectangle (where traces are plotted).

See **Details** for standard guide names.

fnum=(*left*, *top*, *right*, *bottom*)

Specifies the new location of the subwindow. The location coordinates of the subwindow sides can have one of two possible meanings:

1) When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.
2) When any value is greater than 1, coordinates are taken to be fixed locations measured in points, or pixels for control panels, relative to the top left corner of the host frame.

pguide=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the guide name(s) to which the plot rectangle of the graph subwindow is attached inside the host window.

Guides are identified by the standard names or user-defined names as defined by the host. Use * to specify a default guide name.

See **Details** for standard guide names.

Flags

/W= *winName*

Moves the subwindow in the named window or subwindow. When omitted, action will affect the active subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

When moving an exterior subwindow, only the fnum keyword may be used. The values are the same as the **NewPanel** /W flag for exterior subwindows.

The names for the built-in guides are as defined in the following table:

	Left	Right	Top	Bottom
Subwindow Frame	FL	FR	FT	FB
Outer Graph Rectangle	GL	GR	GT	GB
Inner Plot Rectangle	PL	PR	PT	PB

The frame guides apply to all window and subwindow types. The graph rectangle and plot rectangle guide types apply only to graph windows and subwindows.

See Also

The **MoveWindow** operation. Chapter III-4, **Embedding and Subwindows** for further details and discussion.

MoveVariable

MoveVariable ***sourceVar, destDataFolderPath*** [***newname***]

The MoveVariable operation removes the source numeric variable and places it in the specified location optionally with a new name.

Parameters

sourceVar can be just the name of a numeric variable in the current data folder, a partial path (relative to the current data folder) and variable name or an absolute path (starting from root) and variable name.

destDataFolderPath can be a partial path (relative to the current data folder) or an absolute path (starting from root).

Details

An error is issued if a variable or wave of the same name already exists at the destination.

Examples

```
MoveVariable :foo:v1,:bar:          // Move v1 into data folder bar
MoveVariable :foo:v1,:bar:vv1       // Move v1 into bar with new name vv1
```

See Also

The **MoveString**, **MoveWave**, and **Rename** operations; and Chapter II-8, **Data Folders**.

MoveWave

MoveWave ***sourceWave, destDataFolderPath*** [***newname***]

The MoveWave operation removes the source wave and places it in the specified location optionally with a new name.

Parameters

sourceWave can be just the name of a wave in the current data folder, a partial path (relative to the current data folder) and wave name or an absolute path (starting from root) and wave name.

destDataFolderPath can be a partial path (relative to the current data folder) or an absolute path (starting from root).

Details

An error is issued if a variable or wave of the same name already exists at the destination.

Examples

```
MoveWave :foo:w1,:bar:        // Move wave w1 into data folder bar
MoveWave :foo:w1,:bar:ww1     // Move w1 into bar with new name ww1
```

See Also

The **MoveString**, **MoveVariable**, and **Rename** operations; and Chapter II-8, **Data Folders**.

MoveWindow

MoveWindow [***flags***] ***left, top, right, bottom***

The MoveWindow operation moves the target or specified window to the given coordinates.

Flags

/C — Moves Command window instead of the target window.

/F — *Windows*: Moves the Igor Pro application "frame" and the frame is then adjusted so that no part is offscreen.
Macintosh: Moves nothing.

/I — Coordinates are in inches.

/M — Coordinates are in centimeters.

/P=*procedureTitleAsName* — Moves the specified procedure window instead of the target window.

/W=*winName* — Moves the named window.

Details

Note that neither *winName* nor *procedureTitleAsName* is a string but is the actual window name or procedure window title. If the procedure window's title (procedure windows don't have names) has a space in it, use $ and quotes:

```
MoveWindow/P=$"Log Histogram" 0,0,600,400
```

If /W, /F, /C, and /P are omitted, MoveWindow moves the target window.

The coordinates are in points if neither /I nor /M is used.

On Windows, you can use the MoveWindow operation to minimize, restore, or maximize a window by specifying 0, 1, or 2 for all of the coordinates, respectively, as follows:

```
MoveWindow 0, 0, 0, 0      // Minimize target window.
MoveWindow 1, 1, 1, 1      // Restore target window.
MoveWindow 2, 2, 2, 2      // Maximize target window.
```

These special commands are of use on Windows only. Macintosh Igor Pro 3.1 or later ignores these special commands. In earlier versions, the commands set the target window to its smallest allowed size.

See Also

The **MoveSubwindow** and **DoWindow** operations.

MultiThread

MultiThread *wave* **=** *expression*

In user-defined functions, the MultiThread keyword can be inserted in front of wave assignment statements to speed up execution on multiprocessor computer systems.

Warning: Misuse of this keyword can result in a performance penalty or even a crash. Be sure to read **Automatic Parallel Processing with MultiThread** on page IV-283 before using MultiThread.

The expression must be thread-safe. This means that if it calls a function, the function must be thread-safe. This goes for both built-in and user-defined functions.

Not all built-in functions are thread-safe. Use the Command Help tab in the Igor Help Browser to see which functions are thread-safe.

User-defined functions are thread-safe if they are defined using the **ThreadSafe** keyword. See **ThreadSafe Functions** on page IV-83 for details.

See Also

Automatic Parallel Processing with MultiThread on page IV-283.

Waveform Arithmetic and Assignments on page II-93.

The "MultiThread Mandelbrot Demo" experiment.

NameOfWave

NameOfWave(*wave*)

The NameOfWave function returns a string containing the name of the specified wave.

In a user-defined function that has a parameter or local variable of type WAVE, NameOfWave returns the actual name of the wave identified by the WAVE reference. It can also be used with wave reference functions such as **WaveRefIndexed**.

NameOfWave does not return the full data folder path to the wave. Use **GetWavesDataFolder** for this information.

A null wave reference returns a zero-length string. This might be encountered, for instance, when using WaveRefIndexed in a loop to act on all waves accessed by WaveRefIndexed, and the loop has incremented beyond the highest valid index.

Examples

```
Function ZeroWave(w)
    Wave w
    w = 0
    Print "Zeroed the contents of", NameOfWave(w)
End
```

See Also

See **WAVE**; the **GetWavesDataFolder** and **WaveRefIndexed** functions; and **Wave Reference Functions** on page IV-173.

NaN

NaN

The NaN function returns the "Not a Number" value according to the IEEE standards.

NeuralNetworkRun

NeuralNetworkRun [**/Q/Z**] **Input=***testWave*, **WeightsWave1=***w1*, **WeightsWave2=***w2*

The NeuralNetworkRun operation uses the interconnection weights generated by NeuralNetworkTrain, and saved in the waves M_Weights1 and M_Weights2, to execute the network for a given input. The input

can contain a single run represented by a 1D wave or M runs represented by M columns of a 2D wave. The output of the calculation is saved in the wave W_NNResults or M_NNResults depending on the dimensionality of the input wave. The structure of the network is completely specified by the two weights waves and must match the number of rows in the input wave.

Flags

/Q Suppresses printing information in the History area.

/Z No error reporting.

Key words

Input=*testWave* Specifies the input to the neural network. *testWave* must be a single or double precision wave containing entries in the range [0,1] and have the correct number of rows to match the weights. Execute the network for multiple runs by using a 2D input wave where each column corresponds to a single run. For a 2D input, the result will be stored in M_NNResults with a corresponding column structure.

WeightsWave1=*w1* Specifies the interconnection weights between the input and the hidden layer.

WeightsWave2=*w2* Specifies the interconnection weights between the hidden layer and the output.

See Also

The **NeuralNetworkTrain** operation.

NeuralNetworkTrain

`NeuralNetworkTrain [/Q/Z] [keyword = value]…`

The NeuralNetworkTrain operation trains a three-layer neural network. The training produces two 2D waves that store the interconnection weights between the network neurodes. Once you obtain the weights, you can use them with NeuralNetworkRun.

Flags

/Q Suppresses printing information in the History area.

/Z No error reporting.

Parameters

keyword is one of the following:

Input=*inWave* Specifies the input patterns for training. *inWave* is a 2D wave where each row corresponds to a single training event and each column corresponds to the input values. The number of rows in *inWave* (the number of training sets) and in the output wave must be equal. *inWave* must be single or double precision and all entries must be in the range [0,1].

Iterations=*num* Specifies the number of iterations. Default is 10000.

MinError=*val* Terminates training when the total error drops below *val* (default is 1e-8). The total error is normalized, and is defined as the sum of the squared errors divided by the number of training sets times outputs.

Momentum=*val* Specifies a coefficient for the back-propagation algorithm. This coefficient adds to the change in a particular weight a contribution proportional to the error in a previous iteration. Default momentum is 0.075.

NHidden=*num* Specifies the number of hidden neurodes. You do not need to use the Structure keyword with NHidden because the network is completely specified by the training waves and NHidden.

NReport=*num* Specifies over how many iterations (default is 1000) to print the global RMS error to the history window. Ignored with /Q.

Output=*outWave* Specifies the expected outputs corresponding to the entries in the input wave. The number of rows in *outWave* (the number of training sets) and in the input wave must be equal. *outWave* must be single or double precision and all entries must be in the range [0,1].

LearningRate=*val* Sets the network learning rate, which is used in the backpropagation calculation. Default is 0.15.

Restart Allows specification of your own set of weights as the starting values. Use this to run the training and feed the output weights of one training session as the input for the next.

Structure={*Ni*, *Nh*, *No*}
Specifies the structure of the network. *Ni* is the number of neurodes at the input, *Nh* is the number of hidden neurodes, and *No* is the number of output neurodes. Structure is unnecessary when using NHidden is because the remaining numbers are determined by the sizes of the input and output waves.

WeightsWave1=*w1* Specifies the weights for propagation from the first layer to the second. The 2D wave must be double precision and the dimensions must match the specified neurodes with the same numbers of rows and inputs and with matching numbers of columns and hidden neurodes.

WeightsWave2=*w2* Specifies the weights for propagation from the second to the third layer. The 2D wave must be double precision and the dimensions must match the specified neurodes with the same numbers of rows and hidden neurodes and with matching numbers of columns and outputs.

Details

NeuralNetworkTrain is the first half of the implementation of a three-layer neural network in which both in inputs and outputs are taken as normalized quantities in the range [0,1]. Network training is based on back-propagation to iteratively minimize the error between the output and the expected output for any given training set. Training creates in two 2D waves that contain the interconnection weights between the neurodes. M_Weights1 contains the weights between the input layer and the hidden layer and M_Weights2 contains the weights between the hidden layer and the output layer. During the iteration stage, global error information can be printed in the history window.

The algorithm computes the output of the *k*th neurode by

$$V_k = \left[1 + \exp\left(-\sum_{i=1}^{n} w_i s_i\right)\right]^{-1}$$

where w_i is the weight corresponding to input i, s_i is the signal corresponding to that input, and n is the number of inputs connected to the neurode.

The total error is defined as the sum (over all training sets and all outputs) of the squared differences between the network outputs and the expected values. The sum is normalized by the product of the number of training sets and the number of outputs. The history reports (see NReport parameter) the square root of the total error (RMS error). The square root of the error computed at the end of the last iteration is stored in the variable V_rms.

See Also

The **NeuralNetworkRun** operation.

NewDataFolder

NewDataFolder [/O/S] ***dataFolderSpec***

The NewDataFolder operation creates a new data folder of the given name.

Parameters

dataFolderSpec can be just a data folder name, a partial path (relative to the current data folder) with name or a full path (starting from root) with name. If just a data folder name is used then the new data folder is created in the current data folder. If a full or partial path is used, all data folders except for the last in the path must already exist.

Flags

/O No error if a data folder of the same name already exists.

/S Sets the current data folder to dataFolderSpec after creating the data folder.

Examples

```
NewDataFolder foo          // Creates foo in the current data folder
NewDataFolder :bar:foo     // Creates foo in bar in current data folder
NewDataFolder root:foo     // Creates foo in the root data folder
```

See Also

Chapter II-8, **Data Folders**.

NewFIFO

NewFIFO *FIFOName*

The NewFIFO operation creates a new FIFO.

Details

Useless until channel info is added with **NewFIFOChan**.

An error is generated if a FIFO of same name already exists. *FIFOName* needs to be unique only among FIFOs. You can not overwrite a FIFO.

See Also

FIFOs are used for data acquisition. See **FIFOs and Charts** on page IV-276 and the **NewFIFOChan** operation for more information.

NewFIFOChan

NewFIFOChan [***flags***] ***FIFOName, channelName, offset, gain, minusFS, plusFS, unitsStr*** [, ***vectPnts***]

The NewFIFOChan operation creates a new channel for the named FIFO.

Parameters

channelName must be unique for the specified FIFO.

The *offset*, *gain*, *plusFS*, *minusFS* and *unitsStr* parameters are used when the channel's data is displayed in a chart or transferred to a wave. If given, *vectPnts* must be between 1 and 65535.

Flags

The flags define the type of data to be stored in the FIFO channel:

/B	8-bit signed integer. Unsigned if /U is present.
/C	Complex.
/D	Double precision IEEE floating point.
/I	32-bit signed integer. Unsigned if /U is present.
/S	Single precision IEEE floating point (default).
/U	Unsigned integer data.
/W	16-bit signed integer. Unsigned if /U is present.
/Y=*type*	Specifies wave data type. See details below.

Wave Data Types

As a replacement for the above number type flags you can use /Y=*numType* to set the number type as an integer code. See the **WaveType** function for code values. Do not use /Y in combination with other type flags.

Details

You can not invoke NewFIFOChan while the named FIFO is running.

If you provide a value for *vectPnts*, you will create a channel capable of holding a vector of data rather than just a single data value. When such a channel is used in a Chart, it is displayed as an image using one of the built-in color tables.

Igor scales values in the FIFO channel before displaying them in a chart or transferring them to a wave as follows:

```
scaled_value = (FIFO_value - offset) * gain
```

Igor uses the *plusFS* and *minusFS* parameters (plus and minus full scale) to set the default display scaling for charts.

The *unitsStr* parameter is limited to a maximum of three characters.

When you transfer a channel's data to a wave, using the **FIFO2Wave** operation, Igor stores the *plusFS* and *minusFS* values and the *unitsStr* in the wave's Y scaling.

See Also

FIFOs are used for data acquisition. See **FIFOs and Charts** on page IV-276 and the **NewFIFO** and **FIFO2Wave** operations for more information.

The **Chart** operation for displaying FIFO data.

NewFreeAxis

NewFreeAxis[***flags***] ***axisName***

The NewFreeAxis operation creates a new free axis that has no controlling wave.

Parameters

axisName is the name for the new free axis.

Flags

/L/R/B/T	Specifies whether to attach the free axis to the Left, Right, Bottom, or Top plot edge, respectively. The Left edge is used by default.
/O	Replaces *axisName* if it already exists, which means any existing axis is marked as truly free.
/W=*winName*	Draws in the named graph window. *winName* may also be the name of a subwindow. *winName* must not conflict with other axis names except when using the /O flag. If /W is omitted, it creates a new axis in the active graph window or subwindow.

Details

A truly free axis does not use any scaling or units information from any associated waves (which need not exist.) You can set the properties of a free axis using **SetAxis** or **ModifyFreeAxis**.

Example

Copy this function to your Procedure window and compile:

```
Function axhook(s)
    STRUCT WMAxisHookStruct &s

    Variable t= s.max
    s.max= s.min
    s.min= t
    return 0
End
```

Now execute this code on the Command line:

```
Make jack=x
Display jack
NewFreeAxis fred
ModifyFreeAxis fred, master=left, hook=axhook
```

See Also

The **SetAxis**, **KillFreeAxis**, and **ModifyFreeAxis** operations.

NewFreeDataFolder

NewFreeDataFolder()

The NewFreeDataFolder function creates a free data folder and then returns its data folder reference.

Requires Igor Pro 6.1 or later.

Recommended for advanced programmers only.

Details

Free data folders are those that are not a part of the normal data folder hierarchy and can not be located by name.

See Also

Chapter II-8, **Data Folders**, **Free Data Folders** on page IV-75 and **Data Folder References** on page IV-61.

NewFreeWave

NewFreeWave(*type, numPoints*)

The NewFreeWave function creates a free 1D wave of the given type and number of points and then returns its wave reference.

Requires Igor Pro 6.1 or later.

Recommended for advanced programmers only.

Details

NewFreeWave creates a free wave named 'f'.

You can also create free waves using **Make**/FREE and **Duplicate**/FREE. These are preferable for creating multidimensional free waves and also fine for general use.

The type parameter can be either a code as documented for **WaveType** or can be 0x100 to create a data folder reference wave or 0x200 to create a wave reference wave.

You can redimension free waves as desired but, for maximum efficiency, you should create the wave with the desired type and total number of points and then use the /E=1 flag with **Redimension** to simply reshape without moving data.

A free wave is automatically discarded when the last reference to it disappears.

See Also

Free Waves on page IV-71, **Make**, **Duplicate**.

NewImage

NewImage [***flags***] ***matrix***

The NewImage operation creates a new image graph much like "`Display;AppendImage matrix`" except the graph is prepared using a style more appropriate for images. Rather than using preferences, NewImage provides several discrete styles to choose from.

Parameters

matrix is usually an MxN matrix containing image data. See **AppendImage** for details.

Flags

/F By default, the image is flipped vertically to correspond to normal image orientation. if /F is present then the image is not flipped.

/G=*g* *g*=1: Suppresses the autodetection of three plane images as direct (RGB) color.

g= 0: Same as no /G flag (default).

/HIDE=*h* Hides (h = 1) or shows (h = 0, default) the window.

/HOST=*hcSpec*

Embeds the new image plot in the specified host window or subwindow *hcSpec*.

When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/K=*k* Specifies window behavior when closed.

k =0: Normal with dialog (default).

k =1: Kills with no dialog.

k =2: Disables killing.

k =3: Hides the window.

If you use /K=2 or /K=3, the only way to kill the window is via the DoWindow/K operation.

/N=*name* Requests that the created graph have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen graph name. Use `DoWindow/K name` to ensure that *name* is available.

/S=*s* Specifies one of several window styles.

s=0: Fills entire window with image. No axes. However, this can result in the lower-right corner not being visible due to the target icon or grow icon (*Macintosh*).

s=1: Like *s*=0 but insets image to avoid corner icon.

s=2: Provides minimalist axes (default).

Details

The graph is sized to make the image pixels a multiple of the screen pixels with the graph size constrained to be not too small and not too large.

If *matrix* appears to fit Igor's standard monochrome category, then explicit mode is set (See ModifyImage explicit). To be considered monochrome the wave must be unsigned byte and contain only values of 0, 64 or 255.

Once the graph is created it is a normal graph and has no special properties other than the settings it was created with. Specifically, it will not autosize itself if the dimensions of *matrix* are changed. NewImage is just a shortcut for creating a graph window with a style appropriate for images.

This operation is limited in scope by design. If you need to specify the position, size or title, then use the operations Display and AppendImage.

If the styles provided are not what you desire, touch up an image graph to meet your needs and then use Capture Graph Prefs from the Graphs menu. Then use "`Display;AppendImage`" rather than NewImage.

See Also

The **Display**, **DoWindow**, **AppendImage**, and **ModifyImage** operations.

NewLayout

`NewLayout [`*`flags`*`] [as `*`titleStr`*`]`

The NewLayout operation creates a page layout.

Unlike the Layout operation, NewLayout can be used in user-defined functions. Therefore, NewLayout should be used in new programming instead of Layout.

NewLayout just creates the layout window. Use **AppendLayoutObject** to add objects to the window.

Parameters

The optional *titleStr* parameter is a string expression containing the layout's title. If not specified, Igor will provide one which identifies the objects displayed in the graph.

Flags

/B=(*r*,*g*,b) Specifies the background color for the layout. *r*, *g*, and *b* are integers from 0 to 65535. Defaults to white (65535, 65535, 65535).

/C=*colorOnScreen* Obsolete; still accepted but has no effect. Prior to Igor Pro 5, this switched the layout display mode between black and white and color, but now layouts are always in color.

/HIDE=*h* Hides (h = 1) or shows (h = 0, default) the window.

/K=*k* Specifies window behavior when the user attempts to close it.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

k=3: Hides the window.

If you use /K=2 or /K=3, the only way to kill the window is via the DoWindow/K operation.

/N=*name* Requests that the layout have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen layout name. Use `DoWindow/K name` to ensure that *name* is available.

If /N is not used, a name of the form "Layout*n*", where *n* is some integer, is assigned. In a function or macro, the assigned name is stored in the S_name string. This is the name you can use to refer to the page layout window from a procedure. Use the **RenameWindow** operation to rename the window.

/P=*orientation* Sets the orientation of the page in the layout to either Portrait or Landscape (e.g., `Layout/P=Landscape`). See **Details**.

/W=(*left*,*top*,*right*,*bottom*)

Gives the layout window a specific location and size on the screen. Coordinates for /W are in points.

Details

The orientation of the page is controlled by the page setup record associated with the layout. When you create a new layout window, the page setup record comes from your preferred page setup (which you specify via the Capture Layout Prefs dialog) or from a default page setup that Igor creates by calling the current printer driver. When you recreate a layout window using a Window macro, Igor reuses the page setup originally used for the layout window.

See Also

AppendLayoutObject, **DoWindow**, **RemoveLayoutObjects**, and **ModifyLayout**.

NewMovie

NewMovie [***flags***] [**as** ***fileNameStr***]

The NewMovie operation opens a movie file in preparation for adding frames. QuickTime is used by default but on Windows if QuickTime is not installed or if the /A flag is present, then Windows AVI files will be created. Prior to Igor Pro 6.12, QuickTime was required on Windows.

Parameters

The file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If NewMovie can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

On Macintosh, the name of the movie file is limited to 31 characters because Igor calls Apple routines that have this limit.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

Flags

/A	On Windows, creates an AVI file even if QuickTime is present. On Macintosh /A is ignored. Requires Igor Pro 6.12 or later.
/F=*frameRate*	Frames per second between 1 and 60. Defaults to 10.
/I	Presents a system-provided dialog in which you can change the compression settings. The selections you make become the new default settings but only until you quit Igor Pro.
/L[=*flatten*]	Flattens a QuickTime movie when done. Use this option when transferring the movie to a different platform. Creates the movie as a single file rather than two files (*Windows*) or a file with both a data fork and resource fork (*Macintosh*). This flag has no effect on AVI movies. As of Igor Pro 6.21, flattened movies are created by default. You can use /L=0 to force the old non-flattened method.
/O	Overwrite existing file, if any.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/PICT=*pictName*	Uses the specified picture (see **Pictures** on page III-421) rather than the top graph. Requires Igor Pro 6.12 or later.
/S=*soundWave*	Creates and defines sound track. The specified wave can be either a full-range 16 bit or 8 bit integer type. Floating point waves can also be used and are assumed to contain values from -128 to +127. The wave's time/point, as determined by its X scaling, must be between 1.5625e-5 to 2e-4 which correspond to sampling rates of 5000 to 64000 hertz. The duration should match the duration of a video frame.
/Z	No error reporting; an error is indicated by nonzero value of the output variable V_flag. If the user clicks the cancel button in the Save File dialog, V_flag is set to -1.

Details

If either the path or the file name is omitted then NewMovie displays a Save File dialog to let you create a movie file. If both are present, NewMovie creates the file automatically.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-34 for details.

There can be only one open movie at a time.

The target window at the time you invoke NewMovie must be a graph (unless the /PICT flag is present) and the graph size should remain constant while adding frames to the movie. The graph and optional sound wave are used to determine the size and sound properties only; they do not specify the first frame.

The /PICT flag allows you to create a movie from a page layout in conjunction with the SavePICT/P=_PictGallery_ method. See **SavePICT** on page V-543.

See Also

The **AddMovieFrame**, **AddMovieAudio**, **CloseMovie**, **PlayMovie**, **PlayMovieAction** and **SavePICT** operations.

NewNotebook

NewNotebook [***flags***] [**as** ***titleStr***]

The NewNotebook operation creates a new notebook document.

Parameters

The optional *titleStr* is a string containing the title of the notebook window.

Flags

/HOST=*hcSpec*
: Embeds the new notebook in the host window or subwindow specified by *hcSpec*. The host window or subwindow must be a control panel. Graphs and page layouts are not supported as hosts for notebook subwindows.

When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

See **Notebooks as Subwindows in Control Panels** on page III-94 for more information.

/F=*format*
: *format*=0: Plain text.

format=1: Formatted text.

format=-1: Either plain or formatted (default).

/K=*k*
: Specifies window behavior when closed.

k =0: Normal with dialog (default).

k =1: Kills with no dialog.

k =2: Disables killing.

k =3: Hides the window.

If you use /K=2 or /K=3, the only way to kill the window is via the DoWindow/K operation.

/N=*winName*
: Sets the notebook's window name to *winName*.

/OPTS=*options*
: Sets special options. *options* is a bitwise parameter interpreted as follows:

Bit 0: Hide the vertical scroll bar.

Bit 1: Hide the horizontal scroll bar.

Bit 2: Set the write-protect icon initially to on.

Bit 3: Sets the changeableByCommandOnly bit. When set, the user can not make any modifications.

All other bits are reserved for future use and are currently ignored. Pass zero for all reserved bits.

If /OPTS is omitted, all bits default to zero.

See **Setting Bit Parameters** on page IV-12 for details about setting bits.

/V=*visible*
: Specifies whether the notebook window is visible (*visible*=1; default) or invisible (*visible*=0).

/W=(*left*,*top*,*right*,*bottom*)
: Sets window location. Coordinates are in points for normal notebook windows.

When used with the /HOST flag, the specified location coordinates can have one of two possible meanings:

1. When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.

2. When any value is greater than 1, coordinates are taken to be fixed locations measured in pixels relative to the top left corner of the host frame.

Details

A notebook has a file name, a window name, *and* a window title. In the simplest case these will all be the same.

The file name is the name by which the operating system identifies the notebook once it is saved to disk. When you initially create a notebook, it is not associated with any file. However it still has a file name. This is the name that will be used when the file is saved to disk.

The window name is the name by which Igor identifies the window and therefore the name you specify in operations that act on the notebook.

The window title is what appears in the window's title bar. If you omit the title, NewNotebook uses a default title that is the same as the window name.

If you specify the window name and the notebook format and omit the window title, this is the simplest case. NewNotebook creates the document with no user interaction. The file name, window name and window title will all be the same. For example:

```
NewNotebook/N=Notebook1/F=0
```

If you omit the window name, NewNotebook chooses a default name (e.g., "Notebook0") and presents the standard New Notebook dialog.

If you omit the format or specify a format of -1 (either plain or formatted text), NewNotebook presents the standard New Notebook dialog. For example:

```
NewNotebook/N=Notebook1       // no format specified
```

See Also

The **Notebook** and **OpenNotebook** operations, and Chapter III-1, **Notebooks**.

Notebooks as Subwindows in Control Panels on page III-94.

NewPanel

NewPanel [***flags***] [**as** ***titleStr***]

The NewPanel operation creates a control panel window or subwindow, which may contain Igor controls and drawing objects.

Flags

/EXT=*e*
: Creates an exterior subwindow in combination with /HOST. *e* specifies the host window side location:
  e =0: Right.
  e =1: Left.
  e =2: Bottom.
  e =3: Top.

/FG=(*gLeft*, *gTop*, *gRight*, *gBottom*)
: Specifies the frame guide to which the outer frame of the subwindow is attached inside the host window.
  The standard frame guide names are FL, FR, FT, and FB, for the left, right, top, and bottom frame guides, respectively, or user-defined guide names as defined by the host. Use * to specify a default guide name.
  Guides may override the numeric positioning set by /W.

/FLT[=*f*]
: /FLT or /FLT=1 makes the panel a floating panel.
  /FLT=2 makes it a floating panel with no close box.
  /FLT=0 is the same as omitting /FLT and creates a regular (non-floating) control panel.
  You must execute the following after the NewPanel command:
  `SetActiveSubwindow _endfloat_`
  See **Floating Panels** below for further information.

/HIDE=*h*
: Hides (h = 1) or shows (h = 0, default) the window.

/HOST=*hcSpec*
: Embeds the new control panel in the specified host window or subwindow *hcSpec*.
  When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/I
: Sets coordinates to inches.

/K=*k*
: Specifies window behavior when the user attempts to close it.

k=0: Normal with dialog (default).

k=1: Kills without dialog.

k=2: Disables killing.

k=2: Hides the window.

If you use /K=2 or /K=3, the only way to kill the window is via the DoWindow/K operation.

Exterior subwindows never display a dialog when killed.

/M — Sets coordinates to centimeters.

/N=*name* — Requests that the created panel have this name, if it is not in use. If it is in use, then *name*0, `name`1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen panel name. Use `DoWindow/K` *`name`* to ensure that *name* is available.

Note that a function or macro with the same name will cause a name conflict.

/NA= *n* — Sets panel no-activate mode.

n=0: Normal (default).

n=1: Button click doesn't activate window but click outside of any control does.

n=2: No activation even if click is outside controls. Title bar clicks still activate.

/W=(*left*,*top*,*right*,*bottom*) — Sets the initial coordinates of the panel window (in pixels unless /I or /M are used before /W).

When used with the /HOST flag, the specified location coordinates of the sides can have one of two possible meanings:

1) When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.
2) When any value is greater than 1, coordinates are taken to be fixed locations measured in pixels relative to the top left corner of the host frame.

When the subwindow position is fully specified using guides (using the /HOST or /FG flags), the /W flag may still be used although it is not needed.

Details

If /N is not used, NewPanel automatically assigns to the panel a window name of the form "Panel*n*", where *n* is some integer. In a function or macro, the assigned name is stored in the S_name string. This is the name you can use to refer to the panel from a procedure. Use the **RenameWindow** operation to rename the panel.

Floating Panels

Floating control panels float above all other windows except dialogs. Because floating panels cover up other windows, you should use them sparingly and you should take care to make them small and unobtrusive.

Floating panels are not resizable by default. To allow panel resizing use

```
ModifyPanel fixedSize=0
```

Because floating panels always act as if they are on top, the standard rules for target windows and keyboard focus do not apply.

Normally, a floating panel is never the target window and control procedures will need to explicitly designate the target. But a newly-created floating panel is the default target and will remain so until you execute

```
SetActiveSubwindow _endfloat_
```

It also becomes the default target when the tools are showing and in any non-Operate mode. Similarly, a floating panel with tools not in Operate mode has keyboard focus. To avoid confusion, do not attempt to work on other windows when a floating panel is the default target.

When working with a floating panel, you can show or hide tools or create a recreation macro by Control-clicking (*Macintosh*) or right-clicking (*Windows*) in the panel.

A floating panel does not have keyboard focus. However, a floating panel gains keyboard focus when a control that needs focus is clicked. Focus remains until you press Enter or Escape for a text entry in a setvariable, press Tab until no control has the focus, or until you click outside a focusable control.

On Macintosh, if a floating panel has focus and you activate another window, focus will leave the panel. However on Windows, if a floating panel has focus and you activate another window, the activate sequence will be fouled up leaving the windows in an indeterminate state. Consequently, it is important that you

always finish any keyboard interaction started in a floating panel before moving on to other windows. If this can cause confusion, you should not use controls such as SetVariable and ListBox in a floating panel.

On Macintosh, floating panels are hidden when dialogs are up or when Igor Pro is not the front application.

Exterior Subwindows

Exterior subwindows are automatically positioned along the designated side of a host graph window. You can designate fixed sizes or automatic size with minima. Subwindows are stacked beside the designated side in their creation order with the first one closest.

Subwindow dimensions have various meanings depending on their location. Interior values are taken to be additional grout, exterior values are taken to be sizes. For left or right panels, top is taken to be the minimum height and bottom, if not zero, is height. For top and bottom, left is taken to be the minimum width and right, if not zero, is width. Zero values default to 50 for width and height or size of host.

Exterior subwindows are nonresizable by default. Use `ModifyPanel fixedSize=0` to allow manual resizing. If you resize a panel, the original window dimensions are lost. You can also use **MoveSubwindow** to resize the subwindow.

Unlike normal subwindows, exterior subwindows have a tools palette. Click in the window and then choose the Show Tools or Hide Tools menu item.

Exterior subwindows have hook functions independent of the host window.

Examples

In a new experiment, execute these commands on the command line to create two exterior subwindows:

```
Display
// Create panel on right with min height of 200 pixels, width of 100.
NewPanel/HOST=Graph0/EXT=0/W=(0,200,100,0)
// Create another panel on right with grout of 10 and height= width= 100.
NewPanel/HOST=Graph0/EXT=0/W=(10,0,100,100)
```

Now try resizing and moving the graph.

For a demonstration of how the various exterior panels work, copy the following code to the procedure window in a new experiment:

```
Function bpNewExSw(ba) : ButtonControl
    STRUCT WMButtonAction &ba

    switch( ba.eventCode )
        case 2:                                    // mouse up
            ControlInfo/W=$ba.win ckUseRect
            Variable useR= V_Value
            ControlInfo/W=$ba.win popSide
            Variable side= V_Value-1
            ControlInfo/W=$ba.win ckResizeable
            Variable resizeable= V_Value
            WAVE w=root:epsizes
            if( useR )
                NewPanel/HOST=$ba.win/EXT=(side)/W=(w[0],w[1],w[2],w[3])
            else
                NewPanel/HOST=$ba.win/EXT=(side)
            endif
            if( resizeable )
                ModifyPanel fixedSize=0 // default is 1 for floating and exterior sw
            endif
            break
    endswitch

    return 0
End

Window ExSwTest() : Graph
    PauseUpdate; Silent 1                  // building window...
    Display /W=(803,377,1158,591)
    Button bNewSW,pos={35,21},size={181,30},proc=bpNewExSw,title="Exterior Subwindow"
    SetVariable svLeft,pos={118,82},size={96,15},title="left"
    SetVariable svLeft,limits={0,100,1},value= epsizes[0],bodyWidth= 76
    SetVariable svTop,pos={120,97},size={94,15},title="top"
    SetVariable svTop,limits={0,100,1},value= epsizes[1],bodyWidth= 76
    SetVariable svRight,pos={112,113},size={102,15},title="right"
    SetVariable svRight,limits={0,100,1},value= epsizes[2],bodyWidth= 76
    SetVariable svBottom,pos={103,129},size={111,15},title="bottom"
    SetVariable svBottom,limits={0,100,1},value= epsizes[3],bodyWidth= 76
```

```
    CheckBox ckUseRect,pos={70,62},size={61,14},title="Use Rect:",value= 0
    PopupMenu popSide,pos={73,149},size={78,20},title="Side"
    PopupMenu popSide,mode=1,popvalue="Right",value= #"\"Right;Left;Bottom;Top\""
    CheckBox ckResizeable,pos={76,176},size={65,14},title="Resizeable",value= 0
EndMacro

Function test()
    Make/O/N=4 epsizes=0
    Execute "ExSwTest()"
End
```

After compiling the procedures, execute `test()` on the command line. You can now experiment with different sides and size values.

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

The **ModifyPanel** operation.

NewPath

NewPath [***flags***] ***pathName*** [**,** ***pathToFolderStr***]

The NewPath operation creates a new symbolic path name that can be used as a shortcut to refer to a folder on disk.

Parameters

pathToFolderStr is a string containing the path to the folder for which you want to make a symbolic path. *pathToFolderStr* can also point to an alias (*Macintosh*) or shortcut (*Windows*) for a folder.

If you use a full path for *pathToFolderStr*, see **Path Separators** on page III-398 for details on forming the path. If you use a partial path or just a simple name for *pathToFolderStr*, and you use the /C flag, a new folder is created relative to the Igor Pro folder. No dialog is presented.

If you omit *pathToFolderStr*, you get a chance to select a folder or create a new folder from a dialog.

Flags

/C	Create the folder specified by *pathToFolderStr* if it does not already exist.
/M=*messageStr*	Specifies the prompt message in the dialog.
/O	Overwrites the symbolic path if it exists.
/Q	Suppresses printing path information in the history.
/Z	Doesn't generate an error if the folder does not exist.

Details

Symbolic paths help to isolate your experiments from specific file system paths that contain files created or used by Igor. By using a symbolic path, if the actual location or name of the folder changes, you won't need to change all of your commands. Instead, you need only to change the symbolic path so that it points to the changed folder location.

NewPath sets the variable V_flag to zero if the operation succeeded or to nonzero if it failed. The main use for this is to determine if the user clicked Cancel when you use NewPath to display a choose-folder dialog.

On the Macintosh, pressing Command-Option as the Choose Folder dialog comes up will allow you to choose package folders and folders inside packages.

Examples

```
NewPath Path1, "hd:IgorStuff:Test 1"                // Macintosh
NewPath Path1, "C:IgorStuff:Test 1"                 // Windows
```

creates the symbolic path named Path1 which refers to the specified folder (the path's "value"). You can then refer to this folder in many Igor operations and dialogs by using the symbolic path name Path1.

Windows Note:

You can use either the colon or the backslash character to separate folders. However, the backslash character is Igor's escape character in strings. This means that you have to double each backslash to get one backslash like so:

```
NewPath stuff, "C:\\IgorStuff\\Test 1"
```

Because of this complication, it is recommended that you use Macintosh path syntax even on Windows. See **Path Separators** on page III-398 for details.

See Also

The **PathInfo** operation; especially if you need to preset a starting path for the dialog.

KillPath

NewWaterfall

NewWaterfall [***flags***] ***mwave*** [**vs {*wavex*,*wavez*}**]

The NewWaterfall operation creates a new waterfall plot window or subwindow using each column in the 2D matrix wave, *mwave*, as a waterfall trace.

You can manually set x and z scaling by specifying *wavex* and *wavez* to override the default scalings. Either *wavex* or *wavez* may be omitted by using a "*".

Flags

/FG=(*gLeft*, *gTop*, *gRight*, *gBottom*)
: Specifies the frame guide to which the outer frame of the subwindow is attached inside the host window.

  The standard frame guide names are FL, FR, FT, and FB, for the left, right, top, and bottom frame guides, respectively, or user-defined guide names as defined by the host. Use * to specify a default guide name.

  Guides may override the numeric positioning set by /W.

/HIDE=*h*
: Hides (h = 1) or shows (h = 0, default) the window.

/HOST=*hcSpec*
: Embeds the new waterfall plot in the specified host window or subwindow *hcSpec*.

  When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/I
: Sets window coordinates to inches.

/K=*k*
: Specifies window behavior when closed.

  k =0: Normal with dialog (default).

  k =1: Kills with no dialog.

  k =2: Disables killing.

  k =3: Hides the window.

  If you use /K=2 or /K=3, the only way to kill the window is via the DoWindow/K operation.

/M
: Sets window coordinates to centimeters.

/N=*name*
: Requests that the created waterfall plot window have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen name. Use `DoWindow/K name` to ensure that *name* is available.

/PG=(*gLeft*, *gTop*, *gRight*, *gBottom*)
: Specifies the inner plot rectangle of the waterfall plot subwindow inside its host window.

  The standard plot rectangle guide names are PL, PR, PT, and PB, for the left, right, top, and bottom plot rectangle guides, respectively, or user-defined guide names as defined by the host. Use * to specify a default guide name.

  Guides may override the numeric positioning set by /W.

/W=(*left*,*top*,*right*,*bottom*)
: Specifies window size. Coordinates are in points unless /I or /M is specified before /W.

  When used with the /HOST flag, the specified location coordinates of the sides can have one of two possible meanings:

  1) When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.
  2) When any value is greater than 1, coordinates are taken to be fixed locations measured in points relative to the top left corner of the host frame.

When the subwindow position is fully specified using guides (using the /HOST, /FG, or /PG flags), the /W flag may still be used although it is not needed.

Details

The X and Z axes are always at the bottom and left, whereas the Y axis runs at a default 45 degrees along the right-hand side. The angle and length of the Y axis can be changed using the ModifyWaterfall operation. Other features of the graph can be changed using normal graph operations.

Each column from *mwave* is plotted in (and clipped by) a rectangle defined by the X and Z axes with the rectangle displaced along the angled Y axis as a function of the y value.

Except when hidden lines are active, the traces are drawn from back to front.

To modify certain properties of a waterfall plot, you need to use the ModifyWaterfall operation. For other properties, use the usual axis and trace dialogs.

See Also

Waterfall Plots on page II-296.

The **ModifyWaterfall** and **ModifyGraph** operations.

norm

norm(*srcWave*)

The norm function evaluate the norm of *srcWave*. It returns: $\sqrt{\sum \text{abs}(w[i])^2}$.

This function does not support TEXT waves.

note

note(*waveName*)

The note **function** returns a string containing the note associated with the specified wave.

See Also

To create a wave note, use the **Note** *operation*.

Note

Note [**/K/NOCR**] ***waveName*** [**, *str***]

The Note operation appends *str* to the wave note for the named wave.

Parameters

str is a string expression.

Flags

/K Kills existing note for specified wave.

/NOCR Appends note without a preceding carriage return (\r character). No effect when used with /K.

Examples

```
Note/K wave0        // remove existing note
Note wave0, "This is the first line of the note"
Note wave0, "This is the second line of the note"
Note/K wave0, "This is now the only line of the note"
```

See Also

To get the contents of a wave note, use the **note** *function*.

Notebook

Notebook *winName*, *keyword=value* [, *keyword=value*]…

The Notebook operation sets various properties of the named notebook window. Notebook also inserts text and graphics. See Chapter III-1, **Notebooks**, for general information on notebooks.

Notebook returns an error if the notebook is open for read-only. Keywords that don't materially change the notebook, including findText, findPicture, selection, visible, magnification, statusWidth, userKillMode,

showRuler and rulerUnits, are still permitted. See **Notebook Read/Write Properties** on page III-12 for further information.

Parameters

winName is either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-95 for details on host-child specifications.

If *winName* is an hcSpec, the host window or subwindow must be a control panel. Graphs and page layouts are not supported as hosts for notebook subwindows.

The parameters to the Notebook operation are of the form *keyword=value* where *keyword* says what to do and *value* is a parameter or list of parameters. Igor limits the parameters that you specify to legal values before applying them to the notebook.

The parameters are classified into related groups of *keyword*s.

See Also

To create or modify a notebook action special character, see **NotebookAction**.

To create a notebook subwindow in a control panel, see **Notebooks as Subwindows in Control Panels** on page III-94.

Notebook (*Document Properties*)

Notebook document property parameters

This section of Notebook relates to setting the document properties of the notebook.

adopt=*a*
: Adopts a notebook if it is a file saved to disk. Adopting a notebook makes it part of the packed experiment file, which becomes more self-contained; if you send the experiment to a colleague you will not need to send a notebook file.
  - *a*=0: Checks only whether the notebook is adoptable. Sets V_flag to 0 if the notebook is already adopted or to 1 if it is adoptable.
  - *a*=1: Adopts the notebook (breaks the link to the file on disk). Sets V_flag to 1 if the adoption succeeded or to 0 if it did not (was already adopted, or some error).

backRGB=(*r*,*g*,*b*)
: Sets background color. *r*, *g*, and *b* are integers from 0 to 65535.

changeableByCommandOnly=*c*
: This changeableByCommandOnly property is used to prevent manual modifications to the notebook but allow modifications using commands.
  - *c*=0: Turn changeableByCommandOnly off.
  - *c*=1: Turn changeableByCommandOnly on.

  See **Notebook Read/Write Properties** on page III-12 for details.

defaultTab=*dtw*
: *dtw* is the default tab width in points.

magnification=*m*
: Specifies the desired magnification in percent (between 25 and 500). Otherwise, *m* can be one of these special values:
  - *m*=1: Default magnification.
  - *m*=2: Fit Width.
  - *m*=3: Fit Page.

pageMargins={*left*, *top*, *right*, *bottom*}
: Sets page margins in points. *left*, *top*, *right*, and *bottom* are distances from the respective edges of the physical page.

rulerUnits=*r*
  - *r*=0: Points.
  - *r*=1: Inches.
  - *r*=2: Centimeters.

showRuler=*s*
: Hides (*s*=0) or shows (*s*=1) the ruler.

startPage=*sp*
: Sets the starting page number for printing.

statusWidth=*sw*
: Sets the width in points of the status area on the left of the horizontal scroll bar.

userKillMode=*k*
: Specifies window behavior when the user attempts to close it.

k=0: Normal with dialog (default).

k=1: Clicking the close button kills the notebook with no dialog.

k=2: Clicking the close button does nothing.

k=3: Clicking the close button hides the notebook with no dialog.

writeProtect=*wp* The write-protect property is used to prevent inadvertent manual changes to the notebook.

wp=0: Turn write-protect off.

wp=1: Turn write-protect on.

See **Notebook Read/Write Properties** on page III-12 for details.

Notebook (*Headers and Footers*)

Notebook headers and footers

You can turn headers and footers on and off and position headers and footers using the keywords in this section.

There is currently no way to set the content of headers and footers except manually through the Document Settings dialog. You may be able to use stationery files to create files with specific headers and footers.

footerControl={*defaultFooter*, *firstFooter*, *evenOddFooter*}

defaultFooter is 1 to turn the default footer on, 0 to turn it off.

firstFooter is 1 to turn the first page footer on, 0 to turn it off.

evenOddFooter is 1 to turn different footers for even and odd pages on, 0 to use the same footer for even and odd pages.

footerPos=*pos* *pos* is the position of the footer relative to the bottom of the page in points.

headerControl={*defaultHeader* , *firstHeader* , *evenOddHeader*}

defaultHeader is 1 to turn the default header on, 0 to turn it off.

firstHeader is 1 to turn the first page header on, 0 to turn it off.

evenOddHeader is 1 to turn different headers for even and odd pages on, 0 to use the same header for even and odd pages.

headerPos=*pos* *pos* is the position of the header relative to the top of the page in points.

Notebook (*Miscellaneous*)

Notebook miscellaneous parameters

This section of Notebook relates to setting miscellaneous properties of the notebook.

autoSave=*v*

v=0: Notebook subwindow contents will not be saved in recreation macros.

v=1: Notebook subwindow contents will be saved in recreation macros (default).

This affects notebook subwindows in control panels only. Use autoSave=0 if you do not want the notebook's contents to be saved and restored when the control panel is recreated. Otherwise the notebook subwindow's contents will be restored when recreated.

status={*messageStr*, *flags*}

Sets the message in the status area at the bottom left of the notebook window.

flags is interpreted bitwise. Message is erased when:

bit 0: Selection changes.

bit 1: Window is activated.

bit 2: Selection is deactivated.

bit 3: Document is modified.

If all bits are zero, the message stays until a new message comes along. All other bits are reserved for future use and should be zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

updating={*flags*, *r*} Sets parameters related to the updating of special characters.

flags is interpreted bitwise:

bit 0: Automatically update time or date special characters periodically.

bit 1: Allow updating of special characters via the specialUpdate keyword or via the Special menu.

All other bits are reserved for future use and should be zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

r is the update rate in seconds for updating time or date special characters.

These settings have no effect on the updating of special characters in headers or footers. These characters are always automatically updated when the document is printed.

We recommend that you leave automatic updating off (bit 0 of *flags* parameter) so that updating occurs only via the specialUpdate keyword or via the Special menu.

visible=*v*

v=0: Hides notebook.

v=1: Shows notebook but does not make it top window.

v=2: Shows notebook and makes it top window.

Notebook (*Paragraph Properties*)

Notebook paragraph property parameters

This section of Notebook relates to setting the paragraph properties of the current selection in the notebook.

The margins, spacing, justification, tabs and rulerDefaults keywords provide control over paragraph properties which are governed by rulers. These keywords, in conjunction with the ruler and newRuler keywords, allow you to set paragraph properties. They are allowed for formatted text notebooks only, not for plain text notebooks.

The ruler keywords are described in detail below. Before we get to the detail, you should understand the different things you can do with rulers.

There are four things you can do with a ruler:

modify it	(analogous to manually adjusting a ruler).
redefine it	(analogous to the Redefine Ruler dialog).
create it	(analogous to the Define New Ruler dialog).
apply it	(analogous to selecting a ruler name from Ruler pop-up menu).

Igor's behavior in response to ruler keywords depends on the order in which the keywords appear.

To modify the ruler(s) for the selected paragraph(s), use the margins, spacing, justification, tabs and rulerDefaults keywords *without* using the newRuler or ruler keywords. For example:

```
Notebook Notebook0 tabs={36,144,288},justification=1
```

To redefine an existing ruler, invoke the ruler=*rulerName* keyword *before* any other keywords. For example:

```
Notebook Notebook0 ruler=Ruler1,tabs={36,144,288},justification=1
```

Unlike redefining the ruler manually, when you redefine an existing ruler using ruler=*rulerName*, it does not apply the ruler to the selected text. However, it does update any text governed by the redefined ruler.

To create a new ruler, invoke the newRuler=*rulerName* keyword *before* any other keywords. For example:

```
Notebook Notebook0 newRuler=Ruler1,tabs={36,144,288},justification=1
```

Unlike creating it manually, when you create a new ruler using newRuler=*rulerName*, it does not apply the new ruler to the selected text. If you do not set a particular ruler property when creating a new ruler, the property will be the same as for the Normal ruler. If the specified ruler already exists, newRuler=*rulerName* overwrites the existing ruler.

To apply an existing ruler to the selected text, invoke the ruler=*rulerName* keyword without any other keywords. For example:

```
Notebook Notebook0 ruler=ruler1
```

You and Igor will get confused if you mix ruler keywords with other types of keywords in the same command. It is alright, however to put a selection keyword at the start of the command. Mixing will not cause a crash or any drastic problem but it will likely produce results that you don't understand.

To keep things clear, follow these rules:

- If you use ruler=*rulerName* or newRuler=*rulerName*, put them before any other ruler keywords.
- Do not mix ruler keywords with other kinds, except that it is alright to use the selection keyword at the start of the command.

justification=*j*

j=0: Left aligned.

j=1: Center aligned.

j=2: Right aligned.

j=3: Fully justified.

margins={*indent*,*left*,*right*}

indent sets the indentation of first line from left page margin.

left sets the paragraph's left margin in points measured from the left page margin.

right sets the paragraph's right margin in points measured from the left page margin.

newRuler=*rulerName* Creates a new ruler with the specified name. If a ruler with this name already exists, it is overwritten.

ruler=*rulerName* Applies the named ruler to the selected text or to redefine the named ruler, as explained above.

rulerDefaults={*"fontName"*, *fSize*, *fStyle*, (*r*,*g*,*b*)}

"fontName" sets the ruler's text font, e.g., `"Helvetica"`.

fSize sets the ruler's text size.

fStyle sets the ruler's text style.

(*r*,*g*,*b*) sets the ruler's text color. *r*, *g*, and *b* are integers from 0 to 65535.

You can only use rulerDefaults if you are redefining an existing ruler, using ruler=*rulerName*, or you are creating a new ruler using newRuler=*rulerName*.

spacing={*spaceBefore*,*spaceAfter*,*lineSpace*}

spaceBefore sets the extra space before paragraph in points.

spaceAfter sets the extra space after paragraph in points.

lineSpace sets the extra space between lines of a paragraph in points.

tabs={*tabSpec*} *tabSpec* is list of tab stops in points added to special values that change the tab stop type.

Tab stops have two parts: the tab stop position and the tab type. Each integer in the list of tabs encodes both of these parts as follows:

The low 11 bits contains the tab stop position in points.

The next two bits are reserved for future use and must be zero.

The high three bits are used to contain the tab type as follows:

left tab	0	
center tab	1	add 1*8192 to tab stop position.
right tab	2	add 2*8192 to tab stop position.
decimal tab	3	add 3*8192 to tab stop position.
comma tab	4	add 4*8192 to tab stop position.

Tabs Example

The following puts a left tab at 1 inch, a center tab at 3 inches and a decimal tab at 5 inches:

```
Notebook Notebook1 tabs={1*72, 3*72 + 8192, 5*72 + 3*8192}
```

Notebook (*Selection*)

Notebook selection parameters

This section of Notebook relates to selecting a range of the content of the notebook.

findPicture={*graphicNameStr*, *flags*}

Searches for the picture containing the named graphic (*Macintosh only*) or the next picture if you pass "". Sets V_flag to 1 if the picture was found or to 0 if not found.

flags is a bitwise parameter interpreted as follows:

bit 0: Show selection after the find.

All other bits are reserved for future use. Set bit 0 by setting *flags* = 1.

The search is always forward from the end of the current selection to the end of the document.

findSpecialCharacter={*specialCharacterNameStr*, *flags*}

Searches for the special character with the specified name or the next special character if you pass "". Selects the special character if it is found.

Sets V_flag to 1 if the special character was found or to 0 if not. Sets S_name to the name of the found special character or to "" if it was not found.

flags is a bitwise parameter interpreted as follows:

bit 0: show selection after the find.

All other bits are reserved for future use. Set bit 0 by setting *flags* = 1.

If *specialCharacterNameStr* is empty (""), the search proceeds from the end of the current selection to the end of the document. Otherwise the search always covers the entire document.

findText={*textToFindStr*, *flags*}

Searches for the specified text. Sets V_flag to 1 if the text was found or to 0 if not found.

textToFindStr is a string expression for the text you want to find. If the text contains a carriage return, Igor considers only the part of the text before the carriage return.

flags is a bitwise integer parameter interpreted as follows:

bit 0: Show selection after the find.

bit 1: Do case-sensitive search.

bit 2: Search for whole words.

bit 3: Wrap around.

bit 4: Search backward.

To set bit 0 and bit 3, use $2^0+2^3 = 9$ for *flags*. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

If you are searching forward, the search starts from the end of the current selection. If you are searching backward, the search starts from the start of the current selection.

If you specify "" as the text to search for, it "finds" the current selection. This displays the current selection using `findText={"", 1}`.

selection={*selStart*, *selEnd*}

selStart and *selEnd* are locations within the document. You can specify these document locations by using the following expressions:

(*paragraph*, *pos*)	*paragraph* and *pos* are numeric expressions. *paragraph* is a paragraph number from 0 to *n*-1 where *n* is the number of paragraphs in the document. *pos* is a character position from 0 to *n* where *n* is the number of characters in the paragraph. Position 0 is to the left of the first character in the paragraph. Position *n* is to the right of the last character in the paragraph.
startOfFile	Start of the document.
endOfFile	End of the document.

startOfParagraph	Start of current *selStart* paragraph.
endOfParagraph	End of current *selStart* paragraph.
startOfNextParagraph	Start of paragraph after current *selStart* paragraph.
endOfNextParagraph	End of paragraph after current *selStart* paragraph.
startOfPrevParagraph	Start of paragraph before current *selStart* paragraph.
endOfPrevParagraph	End of paragraph before current *selStart* paragraph.
endOfChars	Just before the carriage return of current *selStart* paragraph.

Igor clips the specified locations to legal values. It also sets the V_flag variable to 0 if the *selStart* location that you specified was valid, to 1 if the start paragraph was out of bounds and to 2 if the start position was out of bounds. You can use the startOfNextParagraph keyword to step through the document one paragraph at a time. When V_flag is nonzero, you are at the end of the document.

The terms next and prev are relative to the paragraph containing the start of the selected text before the selection keyword was invoked.

Selection Examples

Following are some examples of setting the selection:

```
// select all text in notebook
Notebook Notebook1 selection={startOfFile, endOfFile}

// move selection to the end of the notebook
Notebook Notebook1 selection={endOfFile, endOfFile}

// select all of paragraph 3
Notebook Notebook1 selection={(3,0), (4,0)}

// select all of current paragraph except for trailing CR, if any
Notebook Notebook1 selection={startOfParagraph, endOfChars}

// find the first occurrence of "Hello" in the document
Notebook Notebook1 selection={startOfFile,startOfFile}, findText={"Hello",1}

// find the first picture in the document
Notebook Notebook1 selection={startOfFile,startOfFile}, findPicture={"",1}
```

See Also

The **GetSelection** operation to "copy" the selection.

Notebook (*Text Properties*)

Notebook text property parameters

This section of Notebook relates to setting the text properties of the current selection in the notebook.

font="*fontName*" *"fontName"* is the name of the font. Use `"default"` to specify the paragraph's ruler font.

If you specify an unavailable font, it does nothing. This is so that, when you share procedures with a colleague, using a font that the colleague does not have will not cause your procedures to fail. The downside of this behavior is that if you misspell a font name you will get no error message.

fSize=*fontSize* Text size from 3 to 32000 points.

Use -1 to specify the paragraph's ruler size.

fStyle=*fontStyle* A binary coded integer with each bit controlling one aspect of the text style as follows:

bit 0: Bold.

bit 1: Italic.

bit 2: Underline.

bit 3: Outline (*Macintosh only*).

bit 4: Shadow (*Macintosh only*).

Use -1 to specify the paragraph's ruler style. To set bit 0 and bit 1 (bold italic), use $2^0+2^1 = 3$ for *fontStyle*. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

textRGB=(*r*,*g*,*b*) Specifies text color. *r*, *g*, and *b* are integers from 0 to 65535. (0, 0, 0) specifies black. (65535, 65535, 65535) specifies white.

vOffset=*v* Sets the vertical offset in points (positive offset is down, negative is up). Use this to create subscripts and superscripts. vOffset is allowed for formatted text files only, not for plain text files.

Notebook (*Writing Graphics*)

Writing notebook graphics parameters

This section of Notebook relates to inserting graphics at the current selection in the notebook.

These graphics keywords are allowed for formatted text files only, not for plain text files.

convertToPNG=*x* Converts all pictures in the current selection to cross-platform PNG format. If the picture is already PNG, it does nothing.

x is the resolution expansion factor, an integer from 1 to 16 times screen resolution. *x* is clipped to legal limits.

frame=*f* Sets the frame used for the picture and insertPicture keywords.

f=0: No frame (default).

f=1: Single frame.

f=2: Double frame.

f=3: Triple border.

f=4: Shadow frame.

insertPicture={*pictureName*, *pathName*, *filePath*, *options*}

Inserts a picture from a file specified by *pathName* and *filePath*. The supported graphics file formats are listed under **Inserting Pictures** on page III-16.

pictureName is the special character name (see **Special Character Names** on page III-17) to use for the inserted notebook picture or $"" to automatically assign a name.

pathName is the name of an Igor symbolic path created via **NewPath** or $"" to use no path.

filePath is a full path to the file to be loaded or a partial path or simple file name relative to the specified symbolic path.

If *pathName* and *filePath* do not fully specify a file, an Open File dialog is displayed from which the user can choose the file to be inserted.

options is a bitwise parameter defined as follows:

Bit 0: If set, an Open File dialog is displayed even if the file is fully specified by *pathName* and *filePath*.

Bit 1: Determines what to do in the event of a name conflict. If set, the existing special character with the conflicting name is overwritten. If cleared, a unique name is created and used as the special character name for the inserted picture.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The variable V_flag is set to 1 if the picture was inserted or to 0 otherwise, for example, if the user canceled from the Open File dialog.

The string variable S_name is set to the special character name of the picture that was inserted or to "" if no picture was inserted.

The string variable S_fileName is set to the full path of the file that was inserted or to "" if no picture was inserted.

picture={*objectSpec*, *mode*, *flags*}

Inserts a picture based on the specified object.

objectSpec is usually just an object name, which is the name of a graph, table, page layout or picture from Igor's picture gallery (Misc→Pictures). See further discussion below.

mode controls what happens when you insert a picture of a graph, table or page layout window. It does not affect insertions of pictures from the picture gallery.

mode specifies the format of the graph, table, or page layout picture as follows:

mode	Macintosh	Windows
-5	PNG	PNG
-4	HiRes bitmap	Device-independent bitmap
-2	HiRes PICT	Enhanced metafile
-1	Quartz PDF	Enhanced metafile
0	Quartz PDF	Enhanced metafile
1	1X PICT	Enhanced metafile
2	2X PICT	Enhanced metafile
4	4X PICT	Enhanced metafile
8	8X PICT	Enhanced metafile

Mode 0 is recommended unless you are concerned about cross-platform compatibility in which case you must use mode -5 (PNG).

See Chapter III-5, **Exporting Graphics (Macintosh)**, or Chapter III-6, **Exporting Graphics (Windows)**, for further discussion of these formats. The PNG format is discussed further in **Picture Compatibility** on page III-395.

flags is interpreted bitwise:

bit 0: 0 for black and white, 1 for color.

For color, set *flags* = 2^0 = 1. All other bits are reserved for future use and should be zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

scaling={*h*, *v*} Sets the horizontal(*h*) and vertical (*v*) scaling for the selected picture or the picture and insertPicture keywords. *h* and *v* are in percent.

When using the picture keyword, you may include a coordinate specification after the object name in *objectSpec*. For example:

```
Notebook Notebook1 picture={Layout0(100, 50, 500, 700), 1, 1}
```

The coordinates are in points. A coordinate specification of (0, 0, 0, 0) behaves the same as no coordinate specification at all.

If the object is a graph, the coordinate specification determines the width and height of the graph. If you omit the coordinate specification, Igor takes the width and height from the graph window.

If the object is a layout, the coordinate specification identifies a section of the layout. If you omit the coordinate specification, Igor selects a section of the layout that includes all objects in the layout plus a small margin.

For any other kind of object, Igor ignores the coordinate specification if it is present.

The scaling and frame keywords affect the selected picture, if any. If no picture is selected, they affect the insertion of a picture using the picture or insertPicture keywords. For example, this command inserts a picture of Graph0 with 50% scaling and a double frame:

```
Notebook Test1 scaling={50, 50}, frame=2, picture={Graph0, 1, 1}
```

If no picture is selected and no picture is inserted, scaling and frame have no effect.

InsertPicture Example

```
Function InsertPictureFromFile(nb)
   String nb           // Notebook name or "" for top notebook

   if (strlen(nb) == 0)
      nb = WinName(0, 16, 1)
   endif

   if (strlen(nb) == 0)
      Abort "There are no notebooks"
   endif

   // Display Open File dialog to get the file to be inserted
```

```
    Variable refNum     // Required for Open but not really used
    String fileFilter = "Graphics Files:.eps,.jpg,.png;All Files:.*;"
    Open /D /R /F=fileFilter refNum
    String filePath = S_fileName
    if (strlen(filePath) == 0)
       Print "You cancelled"
       return -1
    endif

    Notebook $nb, insertPicture={$"", $"", filePath, 0}
    if (V_flag)
       Print "Picture inserted"
    else
       Print "No picture inserted"
    endif

    return 0
End
```

Save notebook pictures to files

The savePicture keyword is allowed for formatted text files only, not for plain text files.

savePicture={*pictureName*, *pathName*, *filePath*, *options*}

Saves a picture from a formatted text notebook to a file specified by *pathName* and *filePath*.

pictureName is the special character name (see **Special Character Names** on page III-17) of the picture to be saved or $"" to save the selected picture in which case one picture and one picture only must be selected in the notebook.

pathName is the name of an Igor symbolic path created via **NewPath** or $"" to use no path.

filePath is a full path to the file to be written or a partial path or simple file name relative to the specified symbolic path.

If *pathName* and *filePath* do not fully specify a file, a Save File dialog is displayed in which the user can specify the file to be written.

options is a bitwise parameter defined as follows:

Bit 0: If set, a Save File dialog is displayed even if the file is fully specified by *pathName* and filePath.

Bit 1: If set, a file with the same name is overwritten if it exists. If cleared, a Save File dialog is displayed if the specified file already exists.

Bit 2: If set then the leaf name specified by *filePath* is ignored and a name is automatically generated based on the picture name.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The variable V_flag is set to 1 if the picture was written or to 0 otherwise, for example, if the user canceled from the Save File dialog.

The string variable S_name is set to the special character name of the picture that was saved or to "" if no picture was saved.

The string variable S_fileName is set to the full path of the file that was written or to "" if no picture was written.

Notebook (*Writing Special Characters*)

Writing special character parameters

This section of Notebook relates to inserting special characters at the current selection in the notebook. To insert a notebook action, see **NotebookAction**.

The special characters are page break, short date, long date, abbreviated date and time. They act in some respects like a single character but have special properties. You can insert the special characters using the specialChar keyword.

The specialChar keyword is allowed for formatted text files only, not for plain text files.

Other special characters are allowed in headers and footers only and you can not insert them in a document using the specialChar keyword. These are window title, page number and total pages.

The special characters other than page break character are dynamic and update periodically.

specialChar={*type*, *flags*, *optionsStr*}

type is the special character type as follows:

1: Page break.

2: Short date.

3: Long date.

4: Abbreviated date.

5: Time.

flags is reserved for future use. You should pass 0 for *flags*.

optionsStr is reserved for future use. You should pass "" for *optionsStr*.

specialUpdate=*flags* Updates special characters in the notebook (*Macintosh only* feature).

flags is interpreted bitwise:

bit 0: 0 to update all special characters.

1 to update special characters in the selected text.

bit 1: If 0 and updating for the document is disabled, it generates an error.

If 1, updates regardless of whether updating is enabled or not. Use this at your own risk!

All other bits are reserved for future use and should be zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

On Macintosh, the specialUpdate keyword can update pictures of graphs, tables, and page layouts that were created from windows in the current experiment.

See Also

Chapter III-1, **Notebooks**. The **NewNotebook**, **NotebookAction**, and **OpenNotebook** operations; the **SpecialCharacterInfo** and **SpecialCharacterList** functions.

Notebook (*Accessing Contents*)

Accessing Notebook Contents

getData=*mode* Causes Igor to return the contents of the notebook in the S_value variable. The contents are binary data in a private Igor format encoded as text. The only use for this keyword is to transfer data from one notebook to another by calling getData followed by setData.

mode=1: Stores in S_value plain text or formatted text data, depending on the type of the notebook, from the entire notebook.

mode=2: Stores in S_value plain text data, regardless of the type of the notebook, from the entire notebook.

mode=3: Stores in S_value plain text or formatted text data, depending on the type of the notebook, from the notebook selection only.

mode=4: Stores in S_value plain text data, regardless of the type of the notebook, from the notebook selection only.

See the Notebook In Panel example experiment for examples using getData and setData.

Notebook (*Writing Text*)

Writing notebook text parameters

This section of Notebook relates to inserting text at the current selection in the notebook.

text=*textStr* Inserts the text at the current selection.

Before the text is inserted, Igor converts escape sequences in *textStr* as described in **Escape Characters in Strings** on page IV-13.

Then, it checks for illegal characters. The only character code that is illegal is zero (ASCII NUL character). If it finds an illegal character, Igor generates an error and does not insert the text.

setData=*dataStr* Inserts the data at the current selection.

dataStr is either a regular string expression or the result returned by Notebook getData.

zData=*dataStr* This keyword is used by Igor during the recreation of a notebook subwindow in a control panel. *dataStr* is encoded binary data created by Igor when the recreation macro was generated. It represents the contents of the notebook subwindow in a format private to Igor.

zDataEnd=1 This keyword is used by Igor during the recreation of a notebook subwindow in a control panel. It marks the end of encoded binary data created by Igor when the recreation macro was generated.

NotebookAction

NotebookAction [**/W=*winName***] ***keyword* = *value*** [**, *keyword* = *value*** …]

The NotebookAction operation creates or modifies an "action" in a notebook. A notebook action is an object that executes commands when clicked.

See Chapter III-1, **Notebooks**, for general information about notebooks.

NotebookAction returns an error if the notebook is open for read-only. See **Notebook Read/Write Properties** on page III-12 for further information.

Parameters

The parameters are in *keyword =value* format. Parameters are automatically limited to legal values before being applied to the notebook.

bgRGB=(*r*, *g*, *b*) Specifies the action background color. *r*, *g*, and *b* are values from 0 to 65535.

commands=*str* Specifies the command string to be executed when clicking the action. For multiline commands, add a carriage return (`\r`) between lines.

enableBGRGB=*enable* Uses the background color specified by bgRGB (*enable*=1). Background color is ignored for *enable*=0.

frame=*f* Specifies the frame enclosing the action.

- *f*=0: No frame.
- *f*=1: Single frame.
- *f*=2: Double frame.
- *f*=3: Triple border.
- *f*=4: Shadow frame.

helpText=*helpTextStr* Specifies the help string for the action. The text is limited to 255 characters. On Macintosh, help appears when the cursor is over the action after choosing Help→Show Igor Tips. On Windows, help appears in the status line when the cursor is over the action.

ignoreErrors=*ignore* Controls whether an error dialog will appear (*ignore*=0) or not (*ignore* is nonzero) if an error occurs while executing the action commands.

linkStyle=*linkStyle* Controls the action title text style. If *linkStyle*=1, the style is the same as a help link (blue underlined). If *linkStyle*=0, the style properties are the same as the preceding text.

name=*name* Specifies the name of the new or modified notebook action. This is a standard Igor name. See **Standard Object Names** on page III-415 for details.

padding={*leftPadding*, *rightPadding*, *topPadding*, *bottomPadding*, *internalPadding*} Sets the padding in points. *internalPadding* sets the padding between the title and the picture when both elements are present.

picture=*name* Specifies a picture for the action icon. *name* is the name of a picture in the picture collection (see **Pictures** on page III-421).

If *name* is null (`$""`), it clears the picture parameter.

procPICTName=*name* Specifies a Proc Picture for the action icon (see **Proc Pictures** on page IV-43). *name* is the name of a Proc Picture or null (`$""`) to clear it. This will be a name like

ProcGlobal#myPictName or MyModuleName#myPictName. If you use a module name, the Proc Picture must be declared static.

If you specify both picture and procPICTName, picture will be used.

quiet=*quiet* Displays action commands in the history area (*quiet*=0), otherwise (*quiet*=1) no commands will be recorded.

scaling={*h*, *v*} Scales the picture in percent horizontally, *h*, and vertically, *v*.

showMode=*mode* Determines if the title or picture are displayed.

mode=1: Title only.

mode=2: Picture only.

mode=3: Picture below title.

mode=4: Picture above title.

mode=5: Picture to left of title.

mode=6: Picture to right of title.

Without a picture specification, the action will use title mode regardless of what you specify.

title=*titleStr* Sets the action title to *titleStr*, which is limited to 255 characters.

Flags

/W= *winName* Specifies the notebook window of interest.

winName is either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-95 for details on host-child specifications.

If /W is omitted, NotebookAction acts on the top notebook window.

Examples

```
String nb = WinName(0, 16, 1)           // Top visible notebook
NotebookAction name=Action0, title="Beep", commands="Beep"// Create action
NotebookAction name=Action0, enableBGRGB=1, padding={4,4,4,4,4}
Notebook $nb, findSpecialCharacter={"Action0",1}         // Select action
Notebook $nb, frame=1                                    // Set frame
```

See Also

Chapter III-1, **Notebooks**.

The **Notebook**, **NewNotebook**, and **OpenNotebook** operations; the **SpecialCharacterInfo** and **SpecialCharacterList** functions.

num2char

num2char(*num*)

The num2char function returns a string containing a single byte whose value is the low 8 bits of *num*.

Examples

```
Print num2char(65)      // prints A
Print num2char(97)      // prints a
```

See Also

The **char2num**, **str2num** and **num2str** functions.

num2istr

num2istr(*num*)

The num2istr function returns a string representing *num* after rounding to the nearest integer.

num2str

num2str(*num*)

The num2str function returns a string representing the number *num*.

Precision is limited to only five decimal places. This can cause unexpected and confusing results. For this reason, we recommend that you use **num2istr** or **sprintf** for better control of the format and precision of the number conversion.

See Also

The **sprintf** operation.

The **str2num**, **char2num** and **num2char** functions.

NumberByKey

NumberByKey(*keyStr, kwListStr* [, *keySepStr* [, *listSepStr* [, *matchCase*]]])

The NumberByKey function returns a numeric value extracted from *kwListStr* based on the specified key contained in *keyStr*. *kwListStr* should contain keyword-value pairs such as "`KEY=value1,KEY2=value2`" or "`Key:value1;KEY2:value2`", depending on the values for *keySepStr* and *listSepStr*.

Use NumberByKey to extract a numeric value from a strings containing "`key1=value1;key2=value2;`" style lists such as those returned by functions like **AxisInfo** or **TraceInfo**.

If the key is not found or if any of the arguments is `""` or if the conversion to a number fails then it returns NaN.

keySepStr, *listSepStr*, and *matchCase* are optional; their defaults are ":", ";", and 0 respectively.

Details

keyStr is limited to 255 characters.

kwListStr is searched for an instance of the key string bound by *listSepStr* on the left and a *keySepStr* on the right. The text up to the next *listSepStr* is converted to the returned number.

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are usually case-insensitive. Setting the optional *matchCase* parameter to 1 makes the comparisons case sensitive.

Only the first characters of *keySepStr* and *listSepStr* are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *matchCase* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print NumberByKey("AKEY", "AKEY:123;")                    // prints 123
Print NumberByKey("BKEY", "AKEY=123;Bkey=456;", "=")      // prints 456
Print NumberByKey("KEY2", "KEY1=123,KEY2=999,", "=", ",")// prints 999
Print NumberByKey("ckey", "CKEY=123;ckey=456;", "=")      // prints 123
Print NumberByKey("ckey", "CKEY=123;ckey=456;", "=", ";", 1)// prints 456
```

See Also

The **StringByKey**, **RemoveByKey**, **ReplaceNumberByKey**, **ReplaceStringByKey**, **ItemsInList**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

numpnts

numpnts(*waveName*)

The numpnts function returns the total number of data points in the named wave. To find the number of elements in a dimension of a multidimensional wave, use the **DimSize** function.

numtype

numtype(*num*)

The numtype function returns a number which indicates what kind of value *num* contains.

Details

If *num* is a real number, numtype returns a real number whose value is:

0: If *num* contains a normal number.

1: If *num* contains +/-INF.

2: If *num* contains NaN.

If *num* is a complex number, numtype returns a complex number in which the real part is the number type of the real part of *num* and the imaginary part is the number type of the imaginary part of *num*.

NumVarOrDefault

NumVarOrDefault(*pathStr, defVal*)

The NumVarOrDefault function checks to see if the *pathStr* points to a numeric variable. If the numeric variable exists, NumVarOrDefault returns its value. If the numeric variable does not exist, it returns *defVal* instead.

Details

NumVarOrDefault initializes input values of macros so they can remember their state without needing global variables to be defined first. String variables use the corresponding numeric function, **StrVarOrDefault**.

Examples

```
Macro foo(nval,sval)
   Variable nval=NumVarOrDefault("root:Packages:mypack:nvalSav",2)
   String sval=StrVarOrDefault("root:Packages:mypack:svalSav","Hi")
   String dfSav= GetDataFolder(1)
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S mypack
   Variable/G nvalSav= nval
   String/G svalSav= sval
   SetDataFolder dfSav
End
```

NVAR

NVAR [/C][/Z] *localName* [= *pathToVar*][, *localName1* [= *pathToVar1*]]…

NVAR is a declaration that creates a local reference to a global numeric variable accessed in a user-defined function.

The NVAR reference is required when you access a global numeric variable in a function. At compile time, the NVAR statement specifies the local name referencing a global numeric variable. At runtime, it makes the connection between the local name and the actual global variable. For this connection to be made, the global numeric variable must exist when the NVAR statement is executed.

When *localName* is the same as the global numeric variable name and you want to reference a global variable in the current data folder, you can omit *pathToVar*. Prior to Igor Pro 4.0, *pathToVar* was always required.

pathToVar can be a full literal path (e.g., root:FolderA:var0), a partial literal path (e.g., :FolderA:var0) or $ followed by string variable containing a computed path (see **Converting a String into a Reference Using $** on page IV-47).

You can also use a data folder reference or the /SDFR flag to specify the location of the numeric variable if it is not in the current data folder. See **Data Folder References** on page IV-61 and **The /SDFR Flag** on page IV-63 for details.

If the global variable may not exist at runtime, use the /Z flag and call **NVAR_Exists** before accessing the variable. The /Z flag prevents Igor from flagging a missing global variable as an error and dropping into the Igor debugger. For example:

```
NVAR/Z nv=<pathToPossiblyMissingNumericVariable>
if( NVAR_Exists(nv) )
   <do something with nv>
endif
```

Note that to create a global numeric variable, you use the **Variable**/G operation.

Flags

/C	Variable is complex.
/Z	Ignores variable reference checking failures.

See Also

NVAR_Exists function.

Accessing Global Variables and Waves on page IV-50.

Converting a String into a Reference Using $ on page IV-47.

NVAR_Exists

NVAR_Exists(*name*)

The NVAR_Exists function returns one if specified NVAR reference is valid or zero if not. It can be used only in user-defined functions.

For example, in a user function you can test if a global numeric variable exists like this:

```
NVAR /Z var1 = gVar1          // /Z prevents debugger from flagging bad NVAR
if (!NVAR_Exists(var1))       // No such global numeric variable?
   Variable/G gVar1 = 0       // Create and initialize it
endif
```

See Also

WaveExists, **SVAR_Exists**, and **Accessing Global Variables and Waves** on page IV-50.

Open

Open [***flags***] ***refNum*** [**as** ***fileNameStr***]

The Open operation can, depending on the flags passed to it:

1. Open an existing file to read data from (/R flag without /D).
2. Open a to append results to (/A flag without /D).
3. Create a new file or overwrite an existing file to write results to (no /D, /R or /A flags).
4. Display an Open File dialog (/D/R or /D/A flags with or without /MULT).
5. Display a Save File dialog (/D flag without /R or /A).

Parameters

refNum is the name of a numeric variable to receive the file reference number. *refNum* is set by Open if Open actually opens a file for reading or writing (cases 1, 2 and 3). You use *refNum* with the **FReadLine**, **FStatus**, **FSetPos**, **FBinWrite**, **FBinRead**, **fprintf**, and **wfprintf** operations to read from or write to the file. When you're finished, use pass *refNum* to the **Close** operation to close the file.

Open does not set the file reference number when the /D flag is used (cases 4 and 5) but you must still supply a *refNum* parameter.

The following discussion of the *pathName* and *fileNameStr* parameters applies when you are attempting to open a file for reading or writing (cases 2, 3, and 5 above).

The targeted file is specified by a combination of the *pathName* parameter and the *fileNameStr* parameter. There are three ways to specify the targeted file:

Method	How To Use It
Symbolic path and simple file name	Use /P=*pathName* and *fileNameStr*, where *pathName* is the name of an Igor symbolic path (see **Symbolic Paths** on page II-34) that points to the folder containing the file and *fileNameStr* is the name of the file.
Symbolic path and partial path	Use /P=*pathName* and *fileNameStrs*, where *pathName* is the name of an Igor symbolic path that points to the folder containing the file and *fileNameStr* is a partial path starting from the folder and leading to the file.
Full path	Use just *fileNameStr*, where *fileNameStr* is a full path to the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

The targeted file is fully specified if *fileNameStr* is a full path or if both *pathName* and *fileNameStr* are present and not empty strings.

The targeted file is not fully specified in any of these cases:

as fileNameStr is omitted
fileNameStr is an empty string
fileNameStr is not a full path and no symbolic path is specified

Opening an Existing File For Reading Only

This covers cases 1 (/R without /D).

If the file is fully-specified but does not exist, an error is generated. If you want to detect and handle the error yourself, use the /Z flag.

If the file is not fully-specified, Open displays an Open File dialog.

If a file is opened, *refNum* is set to the file reference number.

Opening an Existing File For Appending

This covers cases 1 (/R without /D) and 2 (/A without /D).

If the file is fully-specified and exists, it is opened for read/write and the current file position is moved to the end of the file.

If the file is fully-specified but does not exist, the file is created and opened for read/write.

If the file is not fully-specified, Open displays an Open File dialog.

If a file is opened, *refNum* is set to the file reference number.

Opening a File For Write

This covers case 3 (no /R, /A or /D).

If the targeted file exists, it is overwritten.

If the targeted file does not exist and it is fully-specified and targets a valid path, a new file is created.

If the file is fully-specified and targets an invalid path, an error is generated. If you want to detect and handle the error yourself, use the /Z flag.

If the file is not fully-specified, Open displays a Save File dialog.

If a file is opened, *refNum* is set to the file reference number.

Displaying an Open File Dialog To Select a Single File

This covers cases 4 (/D with /R or /A).

Open does not actually open the file but just displays the Open File dialog.

If the user chooses a file in the Open File dialog, the S_fileName output string variable is set to a full path to the file. You can use this in subsequent commands. If the user cancels, S_fileName is set to "".

See the documentation for the /D, /F and /M flags and then read **Displaying an Open File Dialog** on page IV-126 for details.

refNum is left unchanged.

Displaying an Open File Dialog To Select Multiple Files

This covers cases 4 (/D with /R or /A) with the /MULT=1 flag.

Open does not actually open the file but just displays the Open File dialog.

If the user chooses one or more files in the Open File dialog, the S_fileName output string variable is set to a carriage-return-delimited list of full paths to one or more files. You can use this in subsequent commands. If the user cancels, S_fileName is set to "".

See the documentation for the /D, /F, /M and /MULT flags and then read **Displaying a Multi-Selection Open File Dialog** on page IV-126 for details.

refNum is left unchanged.

Displaying a Save File Dialog

This covers cases 5 (/D without /R or /A).

Open does not actually open the file but just displays the Save File dialog.

If the user chooses a file in the Save File dialog, the S_fileName output string variable is set to a full path to the file. You can use this in subsequent commands. If the user cancels, S_fileName is set to "".

See the documentation for the /D, /F and /M flags and then read **Displaying a Save File Dialog** on page IV-128 for details.

refNum is left unchanged.

Flags

/A — Opens an existing file for appending or, if the file does not exist, creates a new file and opens it for appending.

/C=*creatorStr* — Specifies the file creator code. This is meaningful on Macintosh only and is ignored on Windows. For opening an existing file, creator defaults to "????" which means "any creator". For creating a new file, *creatorStr* defaults to "IGR0" which is Igor's creator code.

/D[=*mode*] — Specifies dialog-only mode.

/D: A dialog is always displayed.

/D=1: Same as /D.

/D=2: A dialog is displayed only if *pathName* and *fileNameStr* do not specify a valid file.

The /D=1 and /D=2 forms of this command were added in Igor Pro 6.1.

Use this mode to allow the user to choose a file to be opened by a subsequent operation, such as **LoadWave**.

With /D or /D=1, open presents a dialog from which the user can select a file but does not actually open the file. Instead, Open puts the full path to the file into the string variable S_fileName.

/D=2 does the same thing except that it skips the dialog if *pathName* and *fileNameStr* specify a valid file. In this case, if pathName and fileNameStr refer to an alias (Macintosh) or shortcut (Windows), the target of the alias or shortcut is returned.

If the user clicks the Cancel button, S_fileName is set to an empty string.

Use Open/D/R to bring up an Open File dialog. See **Displaying an Open File Dialog** on page IV-126 for details.

Use Open/D/R/MULT=1 to bring up an Open File dialog to select multiple files. See **Displaying a Multi-Selection Open File Dialog** on page IV-126 for details.

Use Open/D to bring up a Save File dialog. See **Displaying a Save File Dialog** on page IV-128 for details.

See **Using Open in a Utility Routine** on page IV-129 for an example using /D=2.

Do not use /Z with /D.

/F=*fileFilterStr* — /F provides control over the file filter menu in the Open File dialog. /F was added in Igor Pro 6.10. See **Open File Dialog File Filters** on page IV-127 and **Save File Dialog File Filters** on page IV-128 for details.

/M=*messageStr* — Prompt message text in the dialog used to select the file, if any.

/MULT=m — Use /D/R/MULT=1 to display a multi-selection Open File dialog.

/D/R/MULT=0 or just /D/R displays a single-selection Open File dialog.

/MULT=1 is allowed only if /D or /D=1 and /R are specified.

See **Displaying a Multi-Selection Open File Dialog** on page IV-126 for details.

/P=*pathName* — Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

/R — The file is opened read only.

/T=typeStr — When creating a new file on Macintosh (/A and /R flag omitted), /T sets the Macintosh file type property for the file if it does not already exist. For example, /T="BINA" sets the Macintosh file type to 'BINA'. If /T is omitted the Macintosh file type will be 'TEXT'. Apple has deemphasized Macintosh file types in favor of file name extensions.

For new code, /F is recommended instead of /T.

When opening an existing file (/A or /R flag used), /T provides control over the file filter menu in the Open File dialog. See **Open File Dialog File Filters** on page IV-127 for details.

When creating a new file (/A and /R flag omitted), /T provides control over the file filter menu in the Save File dialog. See **Save File Dialog File Filters** on page IV-128 for details.

/Z[=*z*] — Prevents aborting of procedure execution if an error occurs, for example if the procedure tries to open a file that does not exist for reading. Use /Z if you want to handle this case in your procedures rather than having execution abort.

When using /Z, /Z=1, or /Z=2, V_flag is set to 0 if no error occurred or to a nonzero value if an error did occur.

Do not use /Z with /D.

/Z=0: Same as no /Z.

/Z=1: Suppresses normal error reporting. When used with /R, it opens the file if it exists. /Z alone has the same effect as /Z=1.

/Z=2: Suppresses normal error reporting. When used with /R, it opens the file if it exists or displays a dialog if it does not exist.

Details

When Open returns, if a file was actually opened, the *refNum* parameter will contain a file reference number that you can pass to other operations to read or write data. If the file was not opened because of an error or because the user canceled or because /D was used, *refNum* will be unchanged.

If you use /R (open for read), Open opens an existing file for reading only.

If you use /A, Open opens an existing file for appending. If the file does not exist, it is created and then opened for appending.

If both /R and /A are omitted then Open creates and opens a file. If the specified file does not already exist, Open creates it and opens it for writing. If the file does already exist then Open opens it and sets the current file position to the start of the file. The current file position determines where in the file data will be written. Thus, you will be overwriting existing data in the file.

Warning: If you open an existing file for writing (you do not use /R) then you will overwrite or truncate existing data in the file. To avoid this, open for read (use /R) or open for append (use /A).

Output Variables

The Open operation returns information in the following variables:

`V_flag` Set only when the /Z flag is used.

V_flag is set to zero if the file was opened, to -1 if Open displayed a dialog (because the file was not fully-specified) and the user canceled, and to some nonzero value if an error occurred.

`S_fileName` Stores the full path to the file that was opened.

If /MULT=1 is used, S_fileName is a carriage-return-separated list of full paths to one or more files.

If an error occurred or if the user canceled, S_fileName is set to an empty string.

When using /D, the value of V_flag is undefined. Do not use /Z with /D. Use S_fileName to determine if the user selected a file or canceled.

Examples

This example function illustrates using Open to open a text file from which data will be read. The function takes two parameters: an Igor symbolic path name and a file name. If either of these parameters is an empty string, the Open operation will display a dialog allowing the user to choose the file. Otherwise, the Open operation will open the file without displaying a dialog.

```
Function DemoOpen(pathName, fileName)
    String pathName     // Name of symbolic path or "" for dialog.
    String fileName     // File name, partial path, full path or "" for dialog.
    Variable refNum
    String str

    // Open file for read.
    Open/R/Z=2/P=$pathName refNum as fileName

    // Store results from Open in a safe place.
    Variable err = V_flag
    String fullPath = S_fileName

    if (err == -1)
        Print "DemoOpen canceled by user."
        return -1
    endif

    if (err != 0)
        DoAlert 0, "Error in DemoOpen"
        return err
    endif
```

```
    Printf "Reading from file \"%s\". First line is:\r", fullPath
    FReadLine refNum, str      // Read first line into string variable
    Print str
    Close refNum
    return 0
End
```

See Also

Symbolic Paths on page II-34.

Close, **FReadLine**, **FStatus**, **FSetPos**, **FBinWrite**, **FBinRead**, **fprintf**, and **wfprintf**.

Displaying an Open File Dialog on page IV-126, **Displaying a Multi-Selection Open File Dialog** on page IV-126, **Open File Dialog File Filters** on page IV-127

Displaying a Save File Dialog on page IV-128, **Save File Dialog File Filters** on page IV-128

Using Open in a Utility Routine on page IV-129

The Load File Demo example in "Igor Pro Folder:Examples:Programming".

OpenNotebook

OpenNotebook [***flags***] [***fileNameStr***]

The OpenNotebook operation opens a file for reading or writing as an Igor notebook.

Unlike the Open operation, OpenNotebook will not create a file if the specified file does not exist. To create a new notebook, use the **NewNotebook** operation.

Parameters

The file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

Flags

/A — Moves the notebook's selection to the end of the notebook.

/K=*k* — Specifies window behavior when closed.
- *k* =0: Normal with dialog (default).
- *k* =1: Kills with no dialog.
- *k* =2: Disables killing.
- *k* =3: Hides the window.

If you use /K=2 or /K=3, the only way to kill the window is via the DoWindow/K operation.

/M=*messageStr* — Prompt message text in the dialog used to find the file, if any.

/N=*winName* — Specifies the window name to be assigned to the new notebook. If omitted, it assigns a name like "Notebook0".

/P=*pathName* — Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

/R — Opens the file as read only.

/T=*typeStr* — Specifies the type or types of files that can be opened.

/V=*visible* — Hides (*visible*= 0) or shows (*visible*= 1; default) the notebook.

/W=(*left*,*top*,*right*,*bottom*) — Specifies window size and position. Coordinates are in points.

/Z — Suppresses error generation. Use this to check if a file exists. If you use /Z, OpenNotebook sets the variable V_flag to 0 if the notebook was opened or to nonzero if there was an error, usually because the specified file does not exist.

Details

The /A (append) flag has no effect other than to move the selection to the end of the notebook after it is opened.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-34 for details.

The /T=*typeStr* flag affects only the dialog that OpenNotebook presents if you do not specify a path and filename. The dialog presents only those files whose type is specified by /T=*typeStr*. There are two file types that are allowed for notebooks: `'TEXT'` which is a plain text file and `'WMT0'` which is a WaveMetrics formatted text file. Therefore, the file type, if you use it, should be either "`TEXT`" or "`WMT0`". If /T=*typeStr* is missing, it defaults to "`TEXTWMT0`". This opens either type of notebook file. On Windows, Igor considers files with ".txt" extensions to be of type TEXT and considers files with ".ifn" to be of type WMT0. See **File Types and Extensions** on page III-404 for details.

See Also

The **Notebook** and **NewNotebook** operations, and Chapter III-1, **Notebooks**.

OpenProc

OpenProc [***flags***] [***fileNameStr***]

The OpenProc operation opens a file as an Igor procedure file.

Note: This operation is used automatically to open procedure files when you open an Igor experiment. You can invoke OpenProc only from the command line. Do not invoke it from a procedure. To open procedure files from a procedure or from a menu definition, use the Execute/P operation.

Parameters

The file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

Flags

/A	Moves the procedure window's selection to the end of the window.
/M=*messageStr*	Prompt message text in the dialog used to find the file, if any.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/R	The file is opened read only.
/T=*typeStr*	Specifies the type or types of files that can be opened.
/V=*visible*	Hides (*visible*= 0) or shows (*visible*= 1; default) the procedure window.
/Z	Suppresses error generation if the specified file does not exist.

Details

The /A (append) flag has no effect other than to move the selection to the end of the procedure file after it is opened.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-34 for details.

OpenProc automatically opens procedure files when you open an Igor experiment. Normally, you will have no use for it. You can not open a procedure file while procedures are executing. Thus, you can't invoke OpenProc from within a procedure. You can only invoke it from the command line or from a user menu definition (actually, you may get away with it in a macro, but it's not recommend).

See Also

Chapter III-13, **Procedure Windows**.

The **Execute** operation.

OperationList

OperationList(*matchStr*, *separatorStr*, *optionsStr*)

The OperationList function returns a string containing a list of internal (built-in) or external operation names corresponding to *matchstr*.

Parameters

Only operation names that match *matchStr* string are listed. Use `"*"` to match all names. See **WaveList** for examples.

The first character of *separatorStr* is appended to each operation name as the output string is generated. *separatorStr* is usually ";" for list processing (See **Processing Lists of Waves** on page IV-174 for details).

Use *optionsStr* to further qualify the list of operations. *optionsStr* is a (case-insensitive) string containing one of these values:

`"internal"`	Restricts the list to built-in operations.
`"external"`	Restricts the list to external operations (see **Igor Extensions** on page III-423).

Any other value for *optionsStr* (`"all"` is recommended) will return both internal and external operations.

See Also

The **DisplayProcedure** operation and the **FunctionList**, **MacroList**, **StringFromList**, and **WinList** functions.

Optimize

`Optimize [`*`flags`*`] `*`funcspec, pWave`

The Optimize operation determines extrema (minima or maxima) of a specified nonlinear function. The function must be defined in the form of an Igor user function.

Use the first form for univariate functions (one dimensional functions; functions taking just one variable). Use the second form with multivariate functions (functions in more than one dimension; functions of more than one variable).

Optimize uses Brent's method for univariate functions. For multivariate functions you can choose several variations of quasi-Newton methods or simulated annealing.

Flags

/A [= *findMax*] Finds a maximum (/A=1 or /A) or minimum (/A=0 or no flag).

/D=*nDigits* Specifies the number of good digits returned (default is 15) by the function being optimized. If you use /X=*xWave* with a single-precision wave, the default is seven.

Ignored with simulated annealing (/M={3,0}).

/DSA=*destWave* Sets a wave to track the current best model only found with simulated annealing (/M={3,0}). *destWave* must have the same number of points as the X vector.

/F=*trustRegion* Sets the initial trust region when /M={1 or 2, …} with multivariate functions. The value is a scaled step size (see **Multivariate Details**). After the first iteration the trust region is adjusted according to conditions.

Ignored with simulated annealing (/M={3,0}).

/H= *highBracket*
/L= *lowBracket* Find the extrema of a univariate function. *lowBracket* and *highBracket* are X values on either side of the extreme point. An extreme point is found between the bracketing values.

If *lowBracket* and *highBracket* are equal, Optimize adds 1.0 to *highBracket* before looking for an extreme point.

Default values for *lowBracket* and *highBracket* are zero. Thus, if neither *lowBracket* nor *highBracket* is present, this is the same as /L=0/H=1.

Ignored with simulated annealing (/M={3,0}).

/I=*maxIters* Sets the maximum number of iterations in searching for an extreme point to *maxIters*. Default is 100 for *stepMethod* (/M flag) 0-2, 10000 for *stepMethod* = 3 (simulated annealing).

If you use this form of the /I flag with simulated annealing, *maxItersAtT* is set to *maxIters*/2 and *maxAcceptances* is set to *maxIters*/10.

/I={*maxIters*, *maxItersAtT*, *maxAcceptances*}
Specifies the number of iterations for simulated annealing. The maximum number of iterations is set by *maxIters*, *maxItersAtT* sets the maximum number of iterations at a given temperature in the cooling schedule, and *maxAcceptances* sets the total number of accepted changes in the X vector (whether they increase or decrease the function) at a given temperature.

If you use this form of the flag with any *stepMethod* (/M flag) other than 3, *maxItersAtT* and *maxAcceptances* are ignored.

Defaults for *stepMethod* = 3 are {10000, 5000, 500}.

/M={*stepMethod*, *hessMethod*}

Specifies the method used for selecting the next step (*stepMethod*) and the method for calculating the Hessian (matrix of second derivatives) with multivariate functions.

stepMethod	Method
0	Line Search
1	Dogleg
2	More-Hebdon
3	Simulated Annealing

hessMethod	Method
0	secant (BFGS)
1	finite differences

Default values are {0,0}. The *hessMethod* variable is ignored if you select *stepMethod* = 3.

/Q

Suppresses printout of results in the history area. Ordinarily, the results of root searches are printed in the history.

/R={*typX1*, *typX2*, …}

/R=*typXWave*

Specifies the expected size of X values with multivariate functions. These values are used to scale X values. If the X values you expect are very different from one, you will get more accurate results if you can give a reasonable estimate. Optimize will use *typXi* to scale X_i to reduce floating-point truncation error.

You must provide the same number of values in either a wave or a list of values as you provide to the /X flag.

Ignored with simulated annealing (/M={3,0}).

/S=*stepMax*

Limits the largest scaled step size allowed with multivariate functions. Optimize will stop if five consecutive steps exceed *stepMax*.

Ignored with simulated annealing (/M={3,0}).

/SSA=*stepWave*

Name of a 3-column wave having number of rows equal to the length of the X vector only when used with simulated annealing (/M={3,0}). *stepWave* sets information about the step size used to generate new X vectors. The step sizes are in terms of normalized X values. The normalization is such that the X_i ranges from -1 to 1 based on the ranges set by the /XSA flag.

Column zero sets the step size used when creating new trial X vectors. Default is 1.0.

Column one sets the minimum step size. Default is 0.001.

Column two sets the maximum step size. Default is 1.0.

/T=*tol*

Sets the stopping criterion with univariate functions. Optimize will attempt to find a minimum within ± *tol*.

When this form is used with a multivariate function, *gradTol* is set to *tol* and *stepTol* is set to $gradTol^2$.

Ignored with simulated annealing (/M={3,0}).

/T={*gradtol*, *stepTol*}

Sets the stopping criteria for multivariate functions. Iterations stop if a point is found with estimated scaled gradient less than *gradTol*, or if an iteration takes a scaled step shorter than *stepTol*. Default values are $\{8.53618 \times 10^{-6}, 7.28664 \times 10^{-11}\}$. These values are $(6.022 \times 10^{-16})^{1/3}$ and $(6.022 \times 10^{-16})^{2/3}$ as suggested by Dennis and Schnabel. 6.022×10^{-16} is the smallest double precision floating point number that, when added to 1, is different from 1.

Ignored with simulated annealing (/M={3,0}).

/TSA={*InitialTemp*, *CoolingRate*}

Used only with simulated annealing (/M={3,0}).

InitialTemp sets the initial temperature. If *InitialTemp* is set to zero, Optimize calls your function 100 times to estimate the best initial temperature. This is the recommended setting unless your function is very expensive to evaluate (in which case, you may not want to use simulated annealing at all).

CoolingRate sets the factor by which the temperature is decreased.

/X=*xWave*

/X={*x1*, *x2*, …} Sets the starting point for searching for an extreme point with multivariate functions or with simulated annealing (/M={3,0}). The starting point can be specified with a wave having as many points as the number of independent variables, or you can write out a list of X values in braces. If you are finding extreme points of a univariate function, use /L and /H instead unless you are using the simulated annealing method. If you specify a wave, this wave is also used to receive the result of the extreme point search.

/XSA=*XLimitWave* Name of a 2-column wave having number of rows equal to the length of the X vector only when used with simulated annealing (/M={3,0}).

Column zero sets the minimum value allowed for each element of the X vector.

Column one sets the maximum value allowed for each element of the X vector.

Default is $\pm X_i$*10 if X_i is nonzero, or ±1 if X_i is zero. While a default is provided, it is highly recommended that you provide an *XLimitWave*.

/Y=*funcSize* Specifies expected sizes of function values with multivariate functions. If you expect your function will return values very different from one, you should set *funcSize* to the expected size. Optimize will use this value to scale the function results to reduce floating-point truncation error.

Parameters

func specifies the name of your user-defined function that will be optimized.

pwave gives the name of a parameter wave that will be passed to your function as the first parameter. It is not modified by Igor. It is intended for your private use to pass adjustable constants to your function.

Function Format

Finding extreme points of a nonlinear function requires that you realize the function as a Igor user function of a certain form. See **Finding Minima and Maxima of Functions** on page III-289 for detailed examples.

Your function must look like this:

```
Function myFunc(w,x1, x2, …)
    Wave w
    Variable x1, x2
    return f(x1, x2, …)            // an expression …
End
```

A univariate function would have only one X variable.

A multivariate function can use a wave to pass in the X values:

```
Function myFunc(w,xw)
    Wave w
    Wave xw
    return f(xw)                   // an expression …
End
```

Replace "f(…)" with an appropriate numerical expression.

Univariate Details

The method used by Optimize to find extreme points of univariate functions requires that the point be bracketed before starting. If you don't use /L and /H to specify the bracketing X values, the defaults are zero and one. Optimize first attempts to find the requested extreme point using the bracketing values (or the default). If that is unsuccessful, it attempts to bracket an extreme point by expanding the bracketing interval. If a suitable interval is found (the search is by no means perfectly reliable), then the search for an extreme point is made again.

Optimize uses Brent's method for univariate functions, which requires no derivatives. This combines a quadratic extrapolation with checking for wild results. In the case of wild results (points beyond the best current bracketing values) the method reverts to a golden section bisection algorithm. For well-behaved functions, the quadratic extrapolation converges superlinearly. The golden section bisection algorithm converges more slowly but features global convergence, that is, if an extremum is there, it will be found.

The stopping criterion is $\left|\frac{x-(a+b)}{2}\right| + \frac{a-b}{2} \le \frac{2}{3} \cdot tol$.

In this expression, a and b are the current bracketing values, and x is the best estimate of the extreme point within the bracketing interval.

The left side of this expression works out to being simply the distance from the current solution to the boundary of the bracketing interval.

Note: Optimizing a univariate function with the simulated annealing method (/M={3,0}) works like a multivariate function, and this section does not apply. See the sections devoted to simulated annealing.

Multivariate Details

With multivariate functions, Optimize scales certain quantities to reduce floating point truncation error. You enter scaling factors using the /R and /Y flags. The /R flag specifies the expected magnitude of X values; Optimize then uses $X_i/typX_i$ in all calculations. Likewise, /Y specifies the expected magnitude of function values.

This scaling can be important for maintaining accuracy if your X's or Y's are very different from one, and especially if your X's have values spanning orders of magnitude.

The Optimize operation uses a quasi-Newton method with derivatives estimated numerically. The function gradient is calculated using finite differences. For estimation of the Hessian (second derivative matrix) you can use either a secant method (*hessMethod* = 0) or finite differences (*hessMethod* = 1). The finite difference method gives a more accurate estimate and may succeed with difficult functions but requires more function evaluations per iteration. The finite difference method's greater accuracy may reduce the total number of iterations required, so the overall number of function evaluations depends on details of the problem being solved. Usually the secant method requires fewer function evaluations and is preferred for functions that are expensive to evaluate.

Once a Newton step is calculated, there are three choices for the method used to find the best next value-line search along the Newton direction (*stepMethod* = 0), double dogleg (*stepMethod* = 1), or More-Hebdon (*stepMethod* = 2). The best method can be found only by experimentation. See Dennis and Schnabel (cited in **References**) for details.

The /F=*trustRegion*, /S=*stepMax* and /T={…, *stepTol*} all refer to scaled step sizes. That is,

$$stepX_i = \left\{ \frac{|\Delta x_i|}{max(x_i, typX_i)} \right\}.$$

The Optimize operation presumes that an extreme point has been found when either the gradient at the latest point is less than *gradTol* or when the last step taken was smaller than *stepTol*. These criteria both refer to scaled quantities:

$$\max_{1 \le i \le n} \left\{ |g_i| \cdot \frac{max(|x_i|, typX_i)}{max(|f|, funcSize)} \right\} \le gradTol \text{ or } \max_{1 \le i \le n} \left\{ \frac{|\Delta x_i|}{max(x_i, typX_i)} \right\} \le stepTol .$$

Simulated Annealing Introduction

The simulated annealing or Metropolis algorithm optimizes a function using a random search of the X vector space. It does not use derivatives to guide the search, making it a good choice if the function to be optimized is in some way poorly behaved. For instance, it is a good method for functions with discontinuities in the function value or in the derivatives.

Simulated annealing also has a good chance of finding a global minimum or maximum of a function having multiple local minima or maxima.

Because simulated annealing uses a random search method, it may require a large number of function evaluations to find a minimum, and it is not guaranteed that it will stop at an actual minimum. For these reasons, it is best to use one of the other methods unless those methods have failed.

The simulated annealing method generates new trial solutions by adding a random vector to the current X vector. The elements of the random vector are set to $stepsize_i * R_i$, where R_i is a random number in the interval (-1, 1). As the solution progresses, the *stepsize* is gradually decreased.

Bad trials, that is, those that change the function value in the wrong direction are accepted with a probably that depends on the simulated temperature. It is this aspect that allows simulated annealing to find a global minimum.

Function values are generated and accepted or rejected for some number of iterations at a given temperature, then the temperature is reduced. The probability of a bad iteration being accepted decreases with decreasing temperature. A too-fast cooling rate can freeze in a bad solution.

Simulated Annealing Details

It is highly recommended that you use the XSA flag to specify *XLimitWave*. This wave sets bounds on the values of the elements of the X vector during the random search. The defaults may be adequate but are totally *ad hoc*. You are better off to specify bounds that make sense to the problem you are solving.

The values of *XLimitWave* in addition to bounding the search space also scale the X vector during computations of probabilities, temperatures, etc. Consequently, the X limits can affect the performance of the algorithm.

A large number of iterations is required to have a good probability of finding a reasonable solution.

It is recommended that you set the initial temperature to zero so that Optimize can estimate a good initial temperature. If you can't afford the 100 function evaluations required, you probably shouldn't be using simulated annealing.

Optimize uses an exponential cooling schedule in which $T_{i+1} = CoolingRate * T_i$ (see the /TSA flag). *CoolingRate* must be in the range 0 to 1. A fast cooling rate (small value of *CoolingRate*) can cause simulated quenching; that is, a bad solution can be frozen in. Very slow cooling will result in slow convergence.

When simulated annealing is selected, the optimization is treated as multivariate even if your function has only a single X input. That is, the output variables and waves are the ones listed under multivariate functions.

Variables and Waves for Output

The Optimize operation reports success or failure via the V_flag variable. A nonzero value is an error code.

Variables for a univariate function:

`V_flag` 0: Search for an extreme point was successful.
57: User abort.
785: Function returned NaN.
786: Unable to find bracketing values for an extreme point.

If you searched for a minimum:

`V_minloc` X value at the minimum.
`V_min` Function value (Y) at the minimum.

If you searched for a maximum:

`V_maxloc` X value at the maximum.
`V_max` Function value (Y) at the maximum.

For simulated annealing only:

`V_SANumIncreases` Number of "bad" iterations accepted.
`V_SANumReductions` Number of iterations resulting in a better solution.

Variables for a multivariate function:

`V_flag` 0: Search for an extreme point was successful.
57: User abort.
788: Iteration limit was exceeded.
789: Maximum step size was exceeded in five consecutive iterations.
790: The number of points in the typical X size wave specified by /R does not match the number of X values specified by the /X flag
791: Gradient nearly zero and no iterations taken. This means the starting point is very nearly a critical point. It could be a solution, or it could be so close to a saddle point or a maximum (when searching for a minimum) that the gradient has no useful information. Try a slightly different starting point.

`V_OptTermCode` Indicates why Optimize stopped. This may be useful information even if V_flag is zero. Values are:
1: Gradient tolerance was satisfied.
2: Step size tolerance was satisfied.
3: No step was found that was better than the last iteration. This could be because the current step is a solution, or your function may be too nonlinear for Optimize to

solve, or your tolerances may be too large (or too small), or finite difference gradients are not sufficiently accurate for this problem.

4: Iteration limit was exceeded.

5: Maximum step size was exceeded in five consecutive iterations. This may mean that the maximum step size is too small, or that the function is unbounded in the search direction (that is, goes to -inf if you are searching for a minimum), or that the function approaches the solution asymptotically (function is bounded but doesn't have a well-defined extreme point).

6: Same as V_flag = 791.

If you searched for a minimum:

`V_min` Function value (Y) at the minimum.

If you searched for a maximum:

`V_max` Function value (Y) at the maximum.

Variables for all functions:

`V_OptNumIters` Number of iterations taken before Optimize terminated.

`V_OptNumFunctionCalls`
Number of times your function was called before Optimize terminated.

Waves for a multivariate function:

`W_extremum` Solution if you didn't use /X=<*xWave*>. Otherwise the solution is returned in your X wave.

`W_OptGradient` Estimated gradient of your function at the solution.

See Also

Finding Minima and Maxima of Functions on page III-289 for further details and examples.

References

The Optimize operation uses Brent's method for univariate functions. *Numerical Recipes* has an excellent discussion (see section 10.2) of this method (but we didn't use their code):

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

For multivariate functions Optimize uses code based on Dennis and Schnabel. To truly understand what Optimize does, read their book:

Dennis, J. E., Jr., and Robert B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Methods*, 378 pp., Society for Industrial and Applied Mathematics, Philadelphia, 1996.

Override

`Override constant objectName = newVal`

`Override strconstant objectName = newVal`

`Override Function funcName()`

The Override keyword redefines a constant, strconstant, or user function. The *objectName* or *funcName* must be the same as the name of the original object or function that is being redefined. The override must be defined before the target object appears in the compile sequence.

See Also

Function Overrides on page IV-84 and **Constants** on page IV-40 for further details.

p

`p`

The p function returns the row number of the current row of the destination wave when used in a wave assignment statement. The row number is the same as the point number for a 1D wave.

Details

Outside of a wave assignment statement p acts like a normal variable. That is, you can assign a value to it and use it in an expression.

See Also
Waveform Arithmetic and Assignments on page II-93.

For other dimensions, the **q**, **r**, and **s** functions.

For scaled dimension indices, the **x**, **y**, **z**, and **t** functions.

p2rect

p2rect(*z*)

The p2rect function returns a complex value in rectangular coordinates derived from the complex value *z* which is assumed to be in polar coordinates (magnitude is stored in the real part and the angle, in radians, in the imaginary part of *z*).

Examples

Assume waveIn and waveOut are complex, then:

```
waveOut = p2rect(waveIn)
```

sets each point of waveOut to the rectangular coordinates based on the magnitude in the real part and the angle (in radians) in the imaginary part of the points in waveIn.

You may get unexpected results if the number of points in waveIn differs from the number of points in waveOut.

See Also

The functions **cmplx**, **conj**, **imag**, **r2polar**, and **real**.

PadString

PadString(*str*, *finalLength*, *padValue*)

The PadString function returns a string identical to *str* except that it has been extended to a total length of *finalLength* using bytes of *padValue*. Use zero to create a C-language style string or use 0x20 to pad with spaces (FORTRAN style). This is useful when reading or writing binary files using **FBinRead** and **FBinWrite**.

See Also

The **UnPadString** function.

Panel

Panel

Panel is a procedure subtype keyword that identifies a macro as being a control panel recreation macro. It is automatically used when Igor creates a window recreation macro for a control panel. See **Procedure Subtypes** on page IV-179 and **Saving a Window as a Recreation Macro** on page II-61 for details.

ParamIsDefault

ParamIsDefault(*pName*)

The ParamIsDefault function determines if an optional user function parameter *pName* was specified during the function call. It returns 1 when *pName* is default (not specified) or it returns 0 when it was specified.

Details

ParamIsDefault works only in the body of a user function and only with optional parameters. The variable *pName* must be valid at compile time; you can not defer lookup to runtime with $.

See Also

Optional Parameters on page IV-30 and **Using Optional Parameters** on page IV-46.

ParseFilePath

ParseFilePath(*mode*, *pathInStr*, *separatorStr*, *whichEnd*, *whichElement*)

The ParseFilePath function provides the ability to manipulate file paths and to extract sections of file paths.

Parameters

Several of the modes refer to elements of the file path. In these modes, *whichEnd* is 0 to select an element relative to the beginning of *pathInStr*, 1 to select an element relative to the end. *whichElement* is zero-based.

mode	**Information Returned**
0	Element specified by *whichEnd* and *whichElement*.
1	Entire *pathInStr*, up to but not including the element specified by *whichEnd* and *whichElement*.
2	Entire *pathInStr* with a trailing separator added if it is not already there. This is useful when you have a path to a folder and want to tack on a file name.
3	Last element of *pathInStr* with the extension, if any, removed. The extension is anything after the last dot in *pathInStr*.
4	Extension in *pathInStr* or "" if there is no extension. The extension is anything after the last dot in *pathInStr*.
5	Entire *pathInStr* but converts it to a format determined by *separatorStr*. *separatorStr* = ":" Converts the path to Macintosh style if it is Windows style. Does nothing to a Macintosh path. *separatorStr* = "\\" Converts the path to Windows style if it is Macintosh style. Does nothing to a Windows path. *separatorStr* = "*" Converts the path to the native style of the operating system Igor is running on. Does nothing to a native path. *separatorStr* = "/" Macintosh-only: Converts the Macintosh-style *pathInStr* input to a Posix (UNIX) path. Unlike the other conversions, the directory or file to which *pathInStr* refers must exist, otherwise "" is returned. To generate a Posix path for a non-existent file, generate the path for the existing folder and append the file name. This always returns "" on Windows.
6	UNC volume name ("\\Server\Share") if *pathIn* is a UNC path or "" if it is not.
7	UNC server name ("Server" from "\\Server\Share") if *pathIn* is a UNC path or "" if it is not.
8	UNC share name ("Share" from "\\Server\Share") if *pathIn* is a UNC path or "" if it is not.

pathInStr is the input path, assumed to either a full path, a partial path (starts with the separator specified by *separatorStr*) or a simple file or folder name (contains no separators).

separatorStr is ":" if *pathInStr* is a Macintosh path or "\\" if *pathInStr* is a Windows path. See **Path Separators** on page III-398 for details about Macintosh versus Windows paths.

whichEnd is 0 to extract an element from the beginning of *pathInStr* or 1 to extract if from the end. *whichEnd* is used only if *mode* is 0 or 1. For other modes, pass 0.

whichElement is a zero-based index to the element that you want to extract. *whichElement* is used only if *mode* is 0 or 1. For other modes, pass 0.

Details

When dealing with Windows paths, you need to be aware that Igor treats the backslash character as an escape character. When you want to put a backslash in a literal string, you need to use two backslashes. See see **Escape Characters in Strings** on page IV-13 and **Path Separators** on page III-398 for details.

On Windows two types of file paths are used: drive-letter paths and UNC ("Universal Naming Convention") paths. For example:

```
// This is a drive-letter path.
C:\Program Files\WaveMetrics\Igor Pro Folder\Igor.exe

// This is a UNC path.
\\BigServer\SharedApps\Igor Pro Folder\Igor.exe
```

In this example, ParseFilePath considers the volume name to be `C:` in the first case and `\\BigServer\SharedApps` in the second. The volume name is treated as one element by ParseFilePath, except for modes 7 and 8 which permit you to extract the components of the UNC volume name.

Except for the leading backslashes in a UNC path, ParseFilePath modes 0 and 1 internally strip any leading or trailing separator (as defined by the *separatorStr* parameter) from *pathInStr* before it starts parsing. So if you pass `":Igor Pro Folder:WaveMetrics Procedures:"`, it is the same as if you had passed `"Igor Pro Folder:WaveMetrics Procedures"`.

If there is no element corresponding to *whichElement* and *mode* is 0, ParseFilePath returns `""`.

If there is no element corresponding to *whichElement* and *mode* is 1, ParseFilePath returns the entire *pathInStr*.

Examples

```
String pathIn, pathOut

// Full path
pathIn= "hd:Igor Pro Folder:WaveMetrics Procedures:Waves:Wave Lists.ipf"

// Extract first element.
Print ParseFilePath(0, pathIn, ":", 0, 0)      // Prints "hd"

// Extract second element.
Print ParseFilePath(0, pathIn, ":", 0, 1)  // Prints "Igor Pro Folder"

// Extract last element.
Print ParseFilePath(0, pathIn, ":", 1, 0)  // Prints "Wave Lists.ipf"

// Extract next to last element.
Print ParseFilePath(0, pathIn, ":", 1, 1)      // Prints "Waves"

// Get path to folder containing the file.
// Prints "hd:Igor Pro Folder:WaveMetrics Procedures:Waves:"
Print ParseFilePath(1, pathIn, ":", 1, 0)

// Extract the file name without extension.
Print ParseFilePath(3, pathIn, ":", 0, 0)      // Prints "Wave Lists"

// Extract the extension.
Print ParseFilePath(4, pathIn, ":", 0, 0)      // Prints "ipf"

// Make sure the given path ends with a colon and concatenate file name.
String path = <routine that returns a Macintosh-style path to a folder>
path = ParseFilePath(2, path, ":", 0, 0)
path += "AFile.txt"
```

See Also

Escape Characters in Strings on page IV-13, **UNC Paths** on page III-399, and **Path Separators** on page III-398 for details. The **RemoveEnding** function.

ParseOperationTemplate

ParseOperationTemplate [***flags***] ***cmdTemplate***

The ParseOperationTemplate operation helps XOP programmers and WaveMetrics programmers write code to implement Igor operations. If you are not an XOP programmer nor a WaveMetrics programmer, it will be of no interest.

ParseOperationTemplate is used to generate starter code for programmers who are creating Igor operations. The starter code is copied to the Clipboard, overwriting any previous Clipboard contents.

Flags

/C=*c* If *c* is nonzero, ParseOperationTemplate stores code for your ExecuteOperation and RegisterOperation functions in the Clipboard.

- c =0: Do not generate code
- c =1: Generate simplified C code - not recommended
- c =2: Generate C code
- c =6: Generate C++ code - added in Igor Pro 6.22

The only difference between /C=6 and /C=2 is that the function is declared as extern "C" instead of static. C++ files that use static work fine although extern "C" is correct.

/S=*s* Stores a definition of your runtime parameter structure in the Clipboard if *s* is nonzero.

This parameter affects code generation if you use mnemonic names in your template. With /S=1, your structure field names will be the same as the specified mnemonic names. This is fine if your mnemonic names are unique. However, if two or more of your parameters use the same mnemonic name, use /S=2. In this case, ParseOperationTemplate will create unique field names by concatenating flag or keyword text and the specified mnemonic names.

/T Stores a comment listing your command template in the Clipboard.

/TS Identifies a ThreadSafe operation by adding an extra field to the runtime parameter structure. This is only of use to WaveMetrics programmers.

Parameters

cmdTemplate is the template that describes the syntax for your operation. See the *Igor XOP Toolkit Reference Manual* for details.

Details

ParseOperationTemplate parses your command template, generating structures that embody the syntax of your operation. If the /C, /S or /T flags is used, it then uses these structures to generate code that can serve as a starting point for implementing your operation. The starter code is stored in the Clipboard.

For most uses, the recommended flags are /T/S=1/C=2.

ParseOperationTemplate sets the following output variable, but only when called from a function or macro:

V_flag 0: *cmdTemplate* was successfully parsed.

-1: *cmdTemplate* was not successfully parsed.

If V_flag is nonzero, this indicates that your *cmdTemplate* syntax is incorrect. See the *Igor XOP Toolkit Reference Manual* for details.

Examples

```
Function Test()
   String cmdTemplate
   cmdTemplate = "MyTest"
   cmdTemplate += " /A={number:aNum1,string:aStrH}"
   cmdTemplate += " /B=wave:bWaveH"
   cmdTemplate += " key1={name:k1N1[,wave:k1WaveH,name:k1N2,string[2]:k1StrHArray]}"

   // If your XOP is C instead of C++, use /C=2 instead of /C=6
   TestOperationParser/T/S=1/C=6 cmdTemplate
   Print V_flag, S_value
End
```

See Also

Igor Extensions on page III-423.

PathInfo

PathInfo [**/S /SHOW**] ***pathName***

The PathInfo operation stores information about the named symbolic path in the following variables:

V_flag: 0 if path does not exist, 1 if it does exist.

S_path: The full path (e.g., "hd:This:That:").

The path returned is a colon-separated path which can be used on Macintosh or Windows. See **Path Separators** on page III-398 for details.

Flags

/S Presets the next otherwise undirected open or save file dialog to the given disk folder.

/SHOW Shows the folder, if it exists, in the Finder (Mac OS X) or Windows Explorer (Windows).

Examples

```
// The following lines perform equivalent actions:
PathInfo/S myPath;Open refNum
Open/P=myPath refNum

// Show Igor's Preferences folder in the Finder/Windows Explorer.
String fullpath= SpecialDirPath("Preferences",0,0,0)
NewPath/O/Q tempPathName, fullpath
PathInfo/SHOW tempPathName
```

See Also

The **NewPath**, **GetFileFolderInfo**, **ParseFilePath** and **SpecialDirPath** operations.

PathList

PathList(*matchStr, separatorStr, optionsStr*)

The PathList function returns a string containing a list of symbolic paths selected based on the *matchStr* parameter.

Details

For a path name to appear in the output string, it must match *matchStr*. The first character of *separatorStr* is appended to each path name as the output string is generated.

PathList works like the WaveList function, except that the *optionsStr* parameter is reserved for future use. Pass "" for it.

Examples

When a new experiment is created there is only one path:

```
Print PathList("*",";","")
```

Prints the following in the history area:

```
  Igor;
```

See Also

The **WaveList** function for an explanation of the *matchStr* and *separatorStr* parameters and for examples. See also **Symbolic Paths** on page II-34 for an explanation of symbolic paths.

PauseForUser

PauseForUser [**/C**] ***mainWindowName*** [**,** ***targetWindowName***]

The PauseForUser operation pauses procedure execution to allow the user to manually interact with a window. For example, you can call PauseForUser from a loop to allow the user to move the cursors on a graph. In this scenario, *targetWindowName* would be the name of the graph and *mainWindowName* would be the name of a control panel containing a message telling the user to adjust the cursors and then click, for example, the Continue button.

If *targetWindowName* is omitted then *mainWindowName* plays the role of target window.

PauseForUser works with graph, table, and panel windows only.

Flags

/C Tells PauseForUser to return immediately after handling any pending events. See Details.

Details

During execution of PauseForUser, only mouse and keyboard activity directed toward either *mainWindowName* or *targetWindowName* is allowed.

While waiting for user action, PauseForUser disables double-clicks and any contextual menus that can lead to dialogs in order to prevent changes on the command line. It also disables killing windows by clicking the close icon in the title bar unless the window was originally created with the /K=1 flag (kill with no dialog).

If /C is omitted, PauseForUser returns only when the main window has been killed.

If /C is present, PauseForUser handles any pending events, sets V_Flag to the truth the target window still exists, and then returns control to the calling user-defined function. Use PauseForUser/C in a loop if you need to do something while waiting for the user to finish interacting with the target window. The /C flag requires Igor Pro 6.1 or later.

See Also

Pause For User on page IV-130 for examples and further discussion.

PauseUpdate

PauseUpdate

The PauseUpdate operation delays the updating of graphs and tables until you invoke a corresponding ResumeUpdate command.

Details

PauseUpdate is useful in a macro that changes a number of things relating to the appearance of a graph. It prevents the graph from being updated after each change. Its effect ends when the macro in which it occurs ends. It also affects updating of tables.

This operation is not allowed from the command line. It is allowed but has no effect in user-defined functions. During execution of a user-defined function, windows update only when you explicitly call the DoUpdate operation.

See Also

The **DelayUpdate**, **DoUpdate**, **ResumeUpdate,** and **Silent** operations.

PCA

PCA [***flags***][***wave0, wave1,… wave99***]

The PCA operation performs principal component analysis. Input data can be in the form of a list of 1D waves, a single 2D wave, or a string containing a list of 1D waves. The operation can produce multiple output waves depending on the specified flags.

Flags

/ALL	Shortcut for the combination of commonly used flags: /CVAR, /SL, /NF, /IND, /IE, and /RMS.
/COV	Calculates the input wave(s) covariance matrix, which as the input for the remainder of the analysis. The covariance matrix is computed by first creating a matrix copying each input 1D wave into sequential columns and then multiplying that matrix by its transpose.
/CVAR	Computes the cumulative percent variance defined as 100 * sum of first *m* eigenvalues divided by the sum of all eigenvalues. The results are stored in the wave W_CumulativeVAR in the current data folder. See also /VAR.
/IE	Computes the imbedded error. Returns errors in the wave W_IE in the current data folder. The wave is scaled using `SetScale/P x 1,1,"", W_IE`. The imbedded error is a function of the number of factors, the number of rows and columns and the sum of the eigen vectors not included in the significant factors. The behavior of IE determines the number of significant factors.
/IND	Computes the factor indicator function. Note that if you specify /IND the residual standard deviation will also be calculated. Returns results in the wave W_IND in the current data folder. The wave is scaled using `SetScale/P x 1,1,"", W_IND`.
/LEIV	Limits eigenvalues so that the SVD calculation does not require too much memory. The limit is set to the minimum of the number of rows or columns of the input.
/NF	Finds the number of significant factors and stores it in the variable V_npnts. You must use /IND in order to compute the significant factors.
/O	Overwrites input waves.
/Q	Suppresses printing of factors in the history area.
/RSD[=*rsdMode*]	Computes the Residual Standard Deviation (RSD) and returns the RSD in the wave W_RSD in the current data folder. The first element in W_RSD is NaN and all remaining wave elements correspond to the number of significant factors. *rsdMode* =0: Covariance about the origin. *rsdMode* =1: Correlation about the origin.
/RMS	Computes the RMS error. Returns results in the wave W_RMS in the current data folder. The wave is scaled using `SetScale/P x 1,1,"", W_RMS`.
/SCMT	Saves C matrix after the singular value decomposition (SVD) in the wave M_C in the current data folder.
/SCR	Converts the individual wave input into standard scores. Does not work when the input is a single 2D wave. It is an error to convert to standard scores when one or more

	entries in the waves are NaN or INF. If you use this feature make sure to use the appropriate form of the RSD calculation.
/SDM	Saves a copy of the data matrix at the end of the calculation. This is useful if your input consists of individual waves or if you want to save the computed standard scores. If the input is a 2D matrix, you will get a copy of the input matrix in the wave M_D.
/SEVC	Saves the eigenvalue vector in the wave W_Eigen, which are the raw eigenvalues generated by the SVD, in the current data folder. Normally, if the SVD was applied to a raw data matrix, i.e., not covariance or correlation matrix, you must square each element of the wave to obtain the PCA eigenvalues. Note that this wave has default wave scaling.
/SL	Computes percent significance level and stores it in the wave W_PSL in the current data folder.
/SQEV	Does not square SVD eigenvalues. If you specify /COV there is no need to use this flag. Use only if your input is already a covariance matrix. In this case the results of the SVD are the eigenvalues not their square roots.
/SRMT	Saves R matrix after the SVD in the wave M_R in the current data folder.
/U	Leaves the input waves unchanged only when the input is a 2D wave. Note that covariance calculations will not be made even if the appropriate flag is used.
/VAR	Computes the variance associated with each eigenvalue. The variance is defined as the ratio of the eigenvalue to the sum of all eigenvalues. The results are stored in the wave W_VAR in the current data folder. See also /CVAR above.
/WSTR=*waveListStr*	String containing a list of names for all input waves.
/Z	No error reporting.

Details

The input is either via /WSTR=*waveListStr* or a list of up to 100 1D waves or a single 2D wave following the last flag.

waveListStr is string containing a semicolon-separated list of 1D waves to be used for the data matrix. *waveListStr* can include any legal path to a wave. Liberal names can be quoted or not quoted. It is assumed that all waves are of the same numerical type (either single or double precision) and that all waves have the same number of points.

Regardless of the inputs, the operation expects that the number of rows in the resulting matrix is greater than or equal to the number of columns.

The operation starts by creating the data matrix from the input wave(s). If you provide a list of 1D waves they become the columns of the data matrix. You can choose to use the covariance matrix (/COV) as the data matrix and you can also choose to normalize each column of the data matrix to convert it into standard scores. This involves computing the average and standard deviation of each column and then setting the new values to be:

$$newValue = \frac{oldValue - colAverage}{colStdv}.$$

You can pre-process the input data using **MatrixOp** with the SubtractMean, NormalizeRows, and NormalizeCols functions.

After creating the data matrix the operation computes the singular value decomposition (SVD) of the data matrix. Results of the SVD can be saved or processed further. Save the C and R matrices using /SCMT and /SRMT. These are related to the input data matrix through: $D = R \cdot C$.

The remainder of the operation lets you compute various statistical quantities defined by Malinowski (see References). Use the flags to determine which ones are computed.

The operation generates a number of output waves. All waves are stored in the current data folder.

You can save the input matrix D in the wave M_D, the optional SVD results are stored in the waves M_C that contains the column matrix C, M_R that contains the row matrix R, and W_Eigen that contains the eigenvalues of the data matrix. Note that these can be the eigenvalues or the square of the eigenvalues depending on the input matrix being a covariance matrix or not (see /SQEV).

The optional 1D output waves (W_RSD, W_RMS, W_IE, W_IND, W_PSL) are saved with wave scaling to make it easier to display the wave as a function of the number of factors.

References

Kaiser, H., Computer Program for Varimax Rotation in Factor Analysis, *Educational and Psychological Measurement*, *XIX*, 413-420, 1959.

Malinowski, E.R., *Factor Analysis in Chemistry*, 3rd ed., John Wiley, 2002.

pcsr

pcsr(*cursorName* [, *graphNameStr*])

The pcsr function returns the point number of the point which the specified cursor (A through J) is on in the top (or named) graph. When used with cursors on images or waterfall plots, pcsr returns the row number, and when used with a free cursor, it returns the relative X coordinate.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

The pcsr result is not affected by any X axis.

See Also

The **Cursor** operation and the **CsrInfo**, **hcsr**, **qcsr**, **vcsr**, **xcsr**, and **zcsr** functions.

Pi

Pi

The Pi function returns π (3.141592…).

PICTInfo

PICTInfo(*pictNameStr*)

The PICTInfo function returns a string containing a semicolon-separated list of information about the named picture. If the named picture does not exist, then "" is returned. Valid picture names can be found in the Pictures dialog.

Details

The string contains six pieces of information, each prefaced by a keyword and colon and terminated with a semicolon.

Keyword	Information Following Keyword
TYPE	One of: “PICT”, “PNG”, “JPEG”, “Enhanced metafile”, “Windows metafile”, “DIB”, “Windows bitmap”, or “Unknown type”.
BYTES	Amount of memory used by the picture.
WIDTH	Width of the picture in pixels.
HEIGHT	Height of the picture in pixels.
PHYSWIDTH	Physical width of the picture in points.
PHYSHEIGHT	Physical height of the picture in points.

Examples

```
print PictInfo("PICT_0")
```

will print the following in the history area:

```
TYPE:PICT;BYTES:55734;WIDTH:468;HEIGHT:340;PHYSWIDTH:468;PHYSHEIGHT:340;
```

See Also

The **ImageLoad** operation for loading PICT and other image file types into waves, and the **PICTList** function. The **StringFromList** operation for parsing the information string.

See **Pictures** on page III-421 and **Pictures Dialog** on page III-423 for general information on picture handling.

PICTList

PICTList(*matchStr*, *separatorStr*, *optionsStr*)

The PICTList function returns a string containing a list of pictures based on *matchStr* and *optionsStr* parameters. See **Details** for information on listing pictures in graphs, panels, layouts, and the picture gallery.

Details

For a picture name to appear in the output string, it must match *matchStr* and also must fit the requirements of *optionsStr*. The first character of *separatorStr* is appended to each picture name as the output string is generated.

The name of each picture is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches all picture names.
"xyz"	Matches picture name xyz only.
"*xyz"	Matches picture names which end with xyz.
"xyz*"	Matches picture names which begin with xyz.
"*xyz*"	Matches picture names which contain xyz.
"abc*xyz"	Matches picture names which begin with abc and end with xyz.

matchStr may begin with the ! character to return windows that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches picture names which do not end with xyz.

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

optionsStr is used to further qualify the picture.

Use `""` accept all pictures in the Pictures Dialog that are permitted by *matchStr*.

Use the WIN: keyword to limit the pictures to the named or target window:

"`WIN:`"	Match all pictures displayed in the top graph, panel, or layout.
"`WIN:`*`windowName`*"	Match all pictures displayed in the named graph, panel, or layout window.

Examples

`PICTList("*",";","")`	Returns a list of all pictures in the Pictures Dialog.
`PICTList("*", ";","WIN:")`	Returns a list of all pictures displayed in the top panel, graph, or layout.
`PICTList("*_bkg", ";", "WIN:Layout0")`	Returns a list of pictures whose names end in "_bkg" and which are displayed in Layout0.

See Also

The **ImageLoad** operation for loading PICT and other image file types into waves, and the **PICTInfo** function. Also the **StringFromList** function for retrieving items from lists.

See **Pictures** on page III-421 and **Pictures Dialog** on page III-423 for general information on picture handling.

Picture

Picture *pictureName*

The Picture keyword introduces an ASCII code picture definition of binary image data.

See Also

Proc Pictures on page IV-43 for further information.

PixelFromAxisVal

PixelFromAxisVal(*graphNameStr, axNameStr, val*)

The PixelFromAxisVal function returns the local graph pixel coordinate corresponding to the axis value in the graph window or subwindow.

Parameters

graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

If the specified axis is not found and if the name is "left" or "bottom" then the first vertical or horizontal axis will be used.

If *graphNameStr* references a subwindow, the returned pixel value is relative to top left corner of base window, not the subwindow.

See Also

The **AxisValFromPixel** and **TraceFromPixel** functions.

PlayMovie

PlayMovie [*flags*] [**as** *fileNameStr*]

The PlayMovie operation opens a QuickTime movie file into a window and plays it.

Parameters

The file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

On Windows, if QuickTime is not installed or cannot open the file, the file is passed to the operating system to be opened with the default program for the given filename extension. This allows PlayMovie to display movies without QuickTime. If this happens, the /W parameters are ignored.

Flags

/I	Coordinates are in inches.
/M	Coordinates are in centimeters.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/W=(*left*,*top*,*right*,*bottom*)	Sets the initial coordinates of the movie window (in points unless /I or /M are used before /W).
/Z	No error reporting; an error is indicated by nonzero value of the variable V_flag. If the user clicks the cancel button in the Open File dialog, V_flag is set to -1.

Details

Coordinates are the initial coordinates of the movie window in points unless /I or /M are used before /W. Only the *top* and *left* coordinates are used. The window has the standard width and height for movies.

If either the path or *fileNameStr* is omitted then PlayMovie will bring up a dialog to let you find a movie file. If both are present, PlayMovie opens the file automatically.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "hd:Folder1:" or "C:\\Folder1\\". See **Symbolic Paths** on page II-34 for details.

Any movie file can be played, not just movies made by Igor. There is no limit on the number of movie windows opened for playing.

Movie windows are considered transient and are not restored when an experiment is reopened.

See Also

The **PlayMovieAction** operation.

PlayMovieAction

PlayMovieAction [**/Z**] ***keyword*** [**=*value***][**, *keyword*** [**=*value***]]

PlayMovieAction operates on the top movie window, opened via **PlayMovie**, or on a movie file opened via the open keyword (requires Igor Pro 6.12 or later).

If the /Z flag is present, errors are not fatal. V_flag is set to error return regardless.

Parameters

extract	Extracts current frame into an 8-bit RGB image wave named M_MovieFrame. (Can be combined with frame=*f*.)
extract=*e*	Extracts *e* frames into a single multiframe wave, M_MovieChunk. This wave will have 3 planes for RGB and will have *e* chunks.
frame=*f*	Moves to specified movie frame (and stops movie).
getID	Returns top movie ID number in V_Value. Do not use in same call with getTime.
getTime	Reads current movie time into variable V_value (in seconds).
gotoBeginning	Goes to beginning of movie.
gotoEnd	Goes to end of movie.
kill	Kills movie window. Windows only: when used with the ref keyword, closes the open movie file.
open=*fullPath*	This keyword is supported on Windows only and fullPath must point to an AVI file with a ".avi" extension. Opens the specifed movie file to enable frame extraction. No movie window is involved. V_Flag is set to zero if no error occurred and V_Value is set to the file reference number. Requires Igor Pro 6.12 or later.
ref=*refNum*	This keyword is supported on Windows only and for AVI files only. The ref keyword is required with all PlayMovieAction commands after using the open keyword to access a movie file. *refNum* must be the file reference number returned in V_Value in the open step. Requires Igor Pro 6.12 or later.
setFrontMovie= *id*	Sets movie with given *id* as top window. Error if no such window (use /Z to suppress errors). Do not use in same call with getID.
start	Starts movie playing.
step=*s*	Moves by *s* frames into movie (0 is same as 1, negative values move backwards).
stop	Stops movie.

Flags

/Z	No error reporting; an error is indicated by nonzero value of the variable V_flag.

Details

Operations are performed in the following order: kill, stop, gotoBeginning, gotoEnd, frame, step, getTime, extract, start. kill overrides all other parameters.

If you want to extract a grayscale image, you can convert the RGB image into grayscale using the ImageTransform command as follows:

```
PlayMovieAction extract
ImageTransform rgb2gray M_MovieFrame
NewImage M_RGB2Gray
```

Windows Only Features

The open and ref keywords support extracting frames from AVI files on Windows only. They require Igor Pro 6.12 or later.

When accessing a file using the open keyword, none of the keywords related to movie windows or playing a movie are supported. Each command that targets the file must include the ref keyword using the file reference number returned in the open step.

When you are finished extracting frames, use the kill keyword to close the file.

To get a full path for use with the open keyword, use the **PathInfo** or **Open** /D/R commands.

Examples

How to determine the number of frames in a simple movie:

```
PlayMovie
PlayMovieAction stop,gotoEnd,getTime
Variable tend= V_value
PlayMovieAction step=-1,getTime
Print "frames= ",tend/(tend-V_value)
PlayMovieAction kill
```

See Also

The **PlayMovie** operation.

PlaySnd

PlaySnd [***flags***] ***fileNameStr***

Note: PlaySnd is obsolete. Use **PlaySound** instead.

Available only on the Macintosh.

The PlaySnd operation plays a sound from the file's data fork, or from an `'snd '` resource.

Parameters

The file containing the sound is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

Flags

/I=*resourceIndex*	Specifies the `'snd '` resource to load by resource index, starting from 1.
/M=*promptStr*	Specifies a prompt if PlaySnd needs to put up a dialog to find the file.
/N=*resNameStr*	Specifies the resource to load by resource name.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/Q	Quiet: suppresses the insertion of `'snd '` info into the history area.
/R=*resourceID*	Specifies the `'snd '` resource to load by resource ID.
/Z	Does not play the sound, just checks for its existence.

Details

If none of /I, /N or /R are specified, PlaySnd tries to play a sound stored in the data fork of the file. If the file dialog is used, only files of type `'sfil'` are shown.

If any of /I, /N or /R are specified, PlaySnd tries to play a sound from an `'snd '` resource. Most programs store sounds in `'snd '` resources. If the file dialog is used, files of all types are shown.

If /P=*pathName* is omitted, then *fileNameStr* can take on three special values:

"Clipboard"	Loads data from Clipboard.
"System"	Loads data from System file.
"Igor" (*not* "Igor Pro")	Loads data from Igor Pro application.

If you specify /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-34 for details.

There are no sounds in the Igor Pro application file.

If the file is not fully specified and *fileNameStr* is not one of these special values, then PlaySnd presents a dialog from which you can select a file. "Fully specified" means that Igor can determine the name of the file (from the *fileNameStr* parameter) and the folder containing the file (from the /P=*pathName* flag or from the *fileNameStr* parameter).

PlaySnd sets the variable V_flag to 1 if the sound exists and fits in available memory or to 0 otherwise.

If the sound exists, PlaySnd also sets the string variable S_Info to:

"SOURCE:*sourceName*;RESOURCENAME:*resourceName*;RESOURCEID:*resourceID*"

If the sound is not a resource then *resourceName* is "" and *resourceID* is 0. *sourceName* will be the name of the file that was loaded or "Clipboard", "System" or "Igor".

Examples

```
PlaySnd/I=1/P=mySnds/Z "Wild Eep"
If (V_flag)            // Any 'snd ' in the "Wild Eep" file?
   Print S_info        // Yes, print resource number, etc.
Endif
```

This prints the following into the history area:

```
SOURCE:resource fork;RESOURCENAME:Wild Eep;RESOURCEID:8;
```

PlaySound

PlaySound [**/A**[**=***a*] **/C**] ***soundWave***

PlaySound /A[**=***a*] {***soundWave1, soundWave2*** [**, *soundWaveN…***]}

The PlaySound operation plays the audio samples in the named wave. The various sound output parameters — number of samples, sample rate, number of channels, and number of bits of resolution — are determined by the corresponding parameters of the wave.

Flags

/A[=*a*] Plays sounds asynchronously so that sounds will continue to play after the command itself has executed.

- /A=0: Same as no /A flag.
- /A=1: Plays sounds asynchronously; same as /A.
- /A=2: Stop playing any current sound before starting this one
- /A=3: Return with user abort error if output buffers are full (rather than waiting.) Use `GetRTError(1)` to detect and clear the error condition.

/C Obsolete - do not use.

On Windows /C causes sound wave data greater than 16-bits to be converted to 16-bit integer. Such data should range from -32768 to +32767.

On Macintosh /C is ignored.

Details

The wave's time per point, as determined by its X scaling, must be a valid sampling rate. A value of 1/44100 (CD standard) is typical.

Sound waves should be 16 bit integers with a range of -32768 to +32767. On Macintosh as of Igor version 6.11, 32-bit floating point data with a range of -1 to +1 can also be used. For backward compatibility, 8-bit integer data with a range of -128 to +127 is also supported.

With the /A flag, the sound plays asynchronously (i.e., the command returns before the sound is finished). If another command is issued before the sound is finished then the new command will wait until the last sound finishes. A PlaySound without the /A flag can play on top of the current sound. The transition between sounds should be seamless on Macintosh but may be slightly delayed on Windows.

It is OK to kill a sound wave immediately after PlaySound returns even if the /A flag is used.

To play a stereo sound, provide a 2 column wave with the left channel in column 0. Actually, the software will attempt to play as many channels as there are columns in the wave. You can also use multiple1D waves with the /A flag. To use this method, enclose the list of 1D waves in braces

Note: The SoundInput operations provide matching sound recording capabilities. See the **SoundInStatus** operation.

Examples

Under Windows, support for sound is somewhat idiosyncratic so these sound examples may not work correctly with your particular hardware configuration.

```
Make/B/O/N=1000 sineSound                  // 8 bit samples
SetScale/P x,0,1e-4,sineSound              // Set sample rate to 10Khz
sineSound= 100*sin(2*Pi*1000*x)            // Create 1Khz sinewave tone
PlaySound sineSound
```

The following example will create a rising pitch in the left channel and a falling pitch in the right channel:

```
Make/W/O/N=(20000,2) stereoSineSound        // 16 bit data
SetScale/P x,0,1e-4,stereoSineSound         // Set sample rate to 10Khz
stereoSineSound= 20000*sin(2*Pi*(1000 + (1-2*q)*150*x)*x)
PlaySound/A stereoSineSound                 // 16 bit, asynchronous
```

Multichannel sounds as in the previous example but from multiple 1D waves:

```
Make/W/O/N=20000 stereoSineSoundL,stereoSineSoundR   // 16 bit data
SetScale/P x,0,1e-4,stereoSineSoundL,stereoSineSoundR// Set sample rate to 10Khz
stereoSineSoundL= 20000*sin(2*Pi*(1000 + 150*x)*x)// rising pitch in left
stereoSineSoundR= 20000*sin(2*Pi*(1000 - 150*x)*x)// falling in right
PlaySound/A {stereoSineSoundL,stereoSineSoundR}   // two 1D waves
```

pnt2x

pnt2x(*waveName, pointNum*)

The pnt2x function returns the X value of the named wave at the point *pointNum*. The point number is truncated to an integer before use.

Details

The result is derived from the wave's X scaling, not any X axis of a graph it may be displayed in. If you would like to convert a fractional point number to an X value you can use:
`leftx(`*waveName*`)+deltax(`*waveName*`)*`*pointNum*.

There is no equivalent function for multidimensional waves. To calculate this information for other dimensions, use this expression:

`DimOffset(`*waveName, dim*`) + `*ScaledDimPos*`*DimDelta(`*waveName,dim*`)`

This expression calculates the scaled position of an element in the dimension *dim* (x, y, z, or t). *ScaledDimPos* is the element number in that dimension (p, q, r, or s). *dim* is 0 for rows, 1 for columns, 2 for layers or 3 for chunks. Setting *dim* =0 is equivalent to using pnt2x. You may want to use **round**, **ceil**, or **floor** to truncate the result to an integer.

See Also

The functions **DimDelta** and **DimOffset**.

Waveform Model of Data on page II-77 and **Changing Dimension and Data Scaling** on page II-83 for an explanation of waves and dimension scaling.

Point

The point structure is used as a substructure usually to store the location of the mouse on the screen.

```
Structure Point
   Int16 v
   Int16 h
EndStructure
```

poissonNoise

poissonNoise(*num*)

The poissonNoise function returns a pseudo-random value from the Poisson distribution whose probability distribution function is

$$f(x;\lambda) = \frac{e^{-\lambda}\lambda^{x}}{x!}, \qquad \begin{array}{c}\lambda > 0 \\ x = 0,1,2...\end{array}$$

with mean and variance equal to *numI* (= λ).

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-332.

Chapter III-12, **Statistics** for a function and operation overview.

poly

poly(*coefsWaveName, x1*)

The poly function returns the value of a polynomial function at x = *x1*.

coefsWaveName is a wave that contains the polynomial coefficients. The number of points in the wave determines the number of terms in the polynomial.

Examples

To fill `wave0` with 100 points containing the polynomial $1 + 2*x + 3*x^2 + 4*x^3$ evaluated over the range from x = -1 to x= 1 (and graph it):

```
Make coefs = {1, 2, 3, 4}         // f(x) = 1 + 2*x + 3*x^2 + 4*x^3
Make/N=100/O wave0; SetScale/I x, -1, 1, wave0; Display wave0
wave0 = poly(coefs, x)
```

poly2D

poly2D(*coefsWaveName, x1, y1*)

The poly2D function returns the value of a 2D polynomial function at x = *x1*, y = *y1*.

coefsWaveName is a wave that contains the polynomial coefficients. The number of points in the wave determines the number of terms in the polynomial and therefore the polynomial degree.

Details

The coefficients wave contains polynomial coefficients for low degree terms first. All coefficients for terms of a given degree must be present, even if they are zero. Among coefficients for a given degree, those for terms having higher powers of X are first. Thus, poly2D returns, for a coefficient wave cw:

f(x,y) = cw[0] + cw[1]*x + cw[2]*y + cw[3]*x^2 + cw[4]*x*y + cw[5]*y^2 + …

A 2D polynomial of degree N has (N+1)(N+2)/2 terms.

Examples

To fill `wave0` with 400 points (20 by 20) containing the polynomial $1 + 2*x + 2.5*y + 3*x^2 + 3.5*xy + 4*y^2$ evaluated over the range x = (-1, 1) and y = (-1, 1) and make a contour plot of it:

```
Make/O coefs = {1, 2, 2.5, 3, 3.5, 4}
Make/N=(20,20)/O wave0
SetScale/I x, -1, 1, wave0
SetScale/I y, -1, 1, wave0
wave0 = poly2D(coefs, x, y)
Display; AppendMatrixContour wave0
```

The polynomial is second degree, so the first command above made the wave coefs with six elements because (2+1)(2+2)/2 = 6.

To fill `wave0` with 100 points containing the polynomial $1 + 2*x + 3*y + 4*x^2 + 4*y^2 + 5*x^3 + 6*y^3$ (note the lack of cross terms) evaluated over the range x = (-1, 1) and y = (-1, 1) (the contour plot already made should update with the new data). The first zero eliminates the second-order cross term x*y and the second and third zeros eliminate the third-order cross terms x^2*y and $x*y^2$:

```
Make/O coefs = {1, 2, 3, 4, 0, 4, 5, 0, 0, 6}
wave0 = poly2D(coefs, x, y)
```

PolygonArea

PolygonArea(*xWave, yWave*)

The PolygonArea function returns the area of a simple, closed, convex or nonconvex planar polygon described by consecutive vertices in *xWave* and *yWave*.

A simple polygon has no internal "holes" and its boundary curve does not intersect itself. Both *xWave* and `yWave` must be 1D, real, numerical waves of the same dimensions. The minimum number of vertices is 3. The function uses the shoelace algorithm to compute the area (see theorem 1.3.3 in the reference below). If there is any error in the input, the function returns NaN.

Example

```
Function estimatePi(num)
   Variable num
```

```
    Make/O/N=(num+1) xxx,yyy
    xxx=sin(2*pi*x/num)
    yyy=cos(2*pi*x/num)
    printf "Relative Error=%g\r",(pi-PolygonArea(xxx,yyy))/pi
End
```

See also

The **areaXY** and **faverageXY** functions.

References

O'Rourke, Joseph, *Computational Geometry in C*, 2nd ed., Cambridge University Press, New York, 1998.

popup

popup *menuList*

The popup keyword is used with Prompt statements in Functions and Macros. It indicates that you want a pop-up menu instead of the normal text entry item in a DoPrompt simple input dialog (or a Macro's missing parameter dialog (archaic)). *menuList* is a string expression containing a list of items, separated by semicolons, that are to appear in the pop-up menu.

Pop-up menus accept both numeric and string parameters. For numeric parameters, the number of the item selected is placed in the variable. Numbering starts from one. For string parameters, the selected item's text is placed in the string variable.

Pop-up items support all of the special characters available for user-defined menu definitions (see **Special Characters in Menu Item Strings** on page IV-114) with the exception that items in pop-up menus are limited to 50 characters, keyboard shortcuts are not supported, and special characters must be enabled.

See Also

Prompt, **DoPrompt**, and **Pop-Up Menus in Simple Dialogs** on page IV-123.

See **WaveList**, **TraceNameList**, **ContourNameList**, **ImageNameList**, **FontList**, **MacroList**, **FunctionList**, **StringList**, and **VariableList** for functions useful in generating lists of Igor objects.

Chapter III-14, **Controls and Control Panels** for details about control panels and controls.

PopupContextualMenu

PopupContextualMenu [/C=(*xpix*, *ypix*) /N] *popupStr*

The PopupContextualMenu operation displays a popup menu until the user makes a selection or cancels the menu by clicking outside of its window.

The menu appears at the current mouse position or at the location specified by the /C flag.

The content of the menu is contained in *popupStr* as a semicolon-separated list of items or in a user-defined menu definition referred to by the name contained in *popupStr*.

Parameters

If *popupStr* specifies the pop-up menu's items (/N is not specified), then *popupStr* is a semicolon-separated list of items such as "yes;no;maybe;", or a string expression that returns such a list, such as **TraceNameList**.

The menu items can be formatted and checkmarked, like user-defined menus can. See **Special Characters in Menu Item Strings** on page IV-114.

If /N is specified, *popupStr* must be the name of a user-defined menu that also has the popupcontextualmenu keyword. See Example 3.

Flags

/C=(*xpix*, *ypix*) Sets the coordinates of the menu's top left corner. Units are in pixels relative to the top-most window, such as the MOUSEX and MOUSEY values passed to a window hook. See the window hook example, below and **SetWindow**.

If /C is not specified, the menu's top left corner appears at the current mouse position.

/N Indicates that *popupStr* contains the name of a menu definition instead of containing a list of menu items.

Details

If the /N flag is not set, PopupContextualMenu sets the following variables:

V_flag	0:	User cancelled the menu without selecting an item, or there was an error such as an empty *popupStr*.
	>= 1:	1 if the first menu item was selected, 2 for the second, etc.
S_selection	"" if the user cancelled or error, else the text of the selected menu item.	

If the /N flag is set, PopupContextualMenu sets the following variables in a manner similar to (though different from) **PopupContextualMenu**:

V_kind — The kind of menu that was selected:

V_kind	Menu Kind
0	Normal text menu item, including **Optional Menu Items** (see page IV-110) and **Multiple Menu Items** (see page IV-111).
3	"*FONT*"
6	"*LINESTYLEPOP*"
7	"*PATTERNPOP*"
8	"*MARKERPOP*"
9	"*CHARACTER*"
10	"*COLORPOP*"
13	"*COLORTABLEPOP*"

See **Specialized Menu Item Definitions** on page IV-112 for details about these special user-defined menus.

V_flag — -1 if the user didn't select any item, otherwise V_flag returns a value which depends on the kind of menu the item was selected from:

V_kind	V_flag Meaning
0	Text menu item number (the first menu item is number 1).
3	Font menu item number (use S_selection, instead).
6	Line style number (0 is solid line)
7	Pattern number (1 is the first selection, a SW-NE light diagonal).
8	Marker number (1 is the first selection, the X marker).
9	Character as an integer, = char2num(S_selection). Use S_selection instead.
10	Color menu item (use V_red, V_green, and V_blue instead).
13	Color table list menu item (use S_selection instead).

S_selection — The menu item text, depending on the kind of menu it was selected from:

V_kind	S_selection Meaning
0	Text menu item text.
3	Font name or "default".
6	Name of the line style menu or submenu.
7	Name of the pattern menu or submenu.
8	Name of the marker menu or submenu.

V_kind	S_selection Meaning
9	Character as string.
10	Name of the color menu or submenu.
13	Color table name.

In the case of **Specialized Menu Item Definitions** (see page IV-112), S_selection will be the title of the menu or submenu, etc.

V_Red, V_Green, V_Blue

If a user-defined color menu ("*COLORPOP*" menu item) was selected then these values hold the red, green, and blue values of the chosen color. The values range from 0 to 65535.

Will be 0 if the last user-defined menu selection was not a color menu selection.

Example 1 - *popupStr* contains a list of menu items

```
// Menu formatting example
String checked= "\\M0:!" + num2char(18) + ":"  // checkmark code
String items= "first;\M1-;"+checked+"third;"   // 2nd is divider, 3rd is checked
PopupContextualMenu items
switch( V_Flag )
   case 1:
      // do something because first item was chosen
      break;
   case 3:
      // do something because first item was chosen
      break;
endswitch
```

Example 2 - *popupStr* contains a list of menu items

```
// Window hook example
SetWindow kwTopWin hook=TableHook, hookevents=1      // mouse down events
Function TableHook(infoStr)
   String infoStr

   Variable handledEvent=0
   String event= StringByKey("EVENT",infoStr)
   strswitch(event)
      case "mousedown":
         Variable isContextualMenu= NumberByKey("MODIFIERS",infoStr) & 0x10
         if( isContextualMenu )
            Variable xpix= NumberByKey("MOUSEX",infoStr)
            Variable ypix= NumberByKey("MOUSEY",infoStr)
            PopupContextualMenu/C=(xpix,ypix) "yes;no;maybe;"
            strswitch(S_selection)
               case "yes":
                  // do something because "yes" was chosen
                  break
               case "no":
                  break
               case "maybe":
                  // do something because "maybe" was chosen
                  break
            endswitch
            handledEvent=1
         endif
   endswitch
   return handledEvent
End
```

Example 3 - popupStr contains the name of a user-defined menu

```
// User-defined contextual menu example
// dynamic menu (to keep WaveList items updated), otherwise not required.
// contextualmenu keyword is required, and implies /Q for all menu items.
//
// NOTE: Actions here are accomplished by the menu definition's
// execution text, such as DoSomethingWithColor.
// See Example 4 for another approach.
```

```
//
Menu "ForContext", contextualmenu, dynamic
    "Hello", Beep
    Submenu "Color"
        "*COLORPOP*", DoSomethingWithColor()
    End
    Submenu "Waves"
        WaveList("*",";",""), /Q, DoSomethingWithWave()
    End
End

Function DoSomethingWithColor()
    GetLastUserMenuInfo
    Print V_Red, V_Green, V_Blue
End

Function DoSomethingWithWave()
    GetLastUserMenuInfo
    WAVE w = $S_value
    Print "User selected "+GetWavesDataFolder(w,2)
End

// Use this code in a function or macro:
PopupContextualMenu/N "ForContext"
if( V_flag < 0 )
    Print "User did not select anything"
endif
```

Example 4 - popupStr contains the name of a user-defined menu

```
// User-defined contextual menu example
Menu "JustColorPop", contextualmenu
    "*COLORPOP*(65535,0,0)", ;// initially red, empty execution text
End

// Use this code in a function or macro
PopupContextualMenu/C=(xpix, ypix)/N "JustColorPop"
if( V_flag < 0 )
    Print "User did not select anything"
else
    Print V_Red, V_Green, V_Blue
endif
```

See Also

Creating a Contextual Menu on page IV-139.

Special Characters in Menu Item Strings on page IV-114 and Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

The **SetWindow** and **PopupMenu** operations.

PopupMenu

PopupMenu [**/Z**] ***ctrlName*** [***keyword = value*** [**,** ***keyword = value*** **…**]]

The PopupMenu operation creates or modifies a pop-up menu control in the target or named window.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the PopupMenu control to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **Button** and **DefaultGUIControls** for more appearance details.

bodyWidth=*width*

Specifies an explicit size for the body (nontitle) portion of a PopupMenu control. By default (bodyWidth=0), the body portion autosizes depending on the current text. If you supply a bodyWidth>0, then the body is fixed at the size you specify

regardless of the body text. This makes it easier to keep a set of controls right aligned when experiments are transferred between Macintosh and Windows, or when the default font is changed.

disable=*d*
Sets user editability of the control.
d=0: Normal.
d=1: Hide.
d=2: Draw in gray state; disable control action.

fColor=(*r*,*g*,*b*)
Sets the initial color of the title. *r*, *g*, and *b* range from 0 to 65535. fColor defaults to black (0,0,0). To further change the color of the title text, use escape sequences as described for title=*titleStr*.

font="*fontName*"
Sets the font used for the pop-up title, e.g., `font="Helvetica"`.

fsize=*s*
Sets the font size for the pop-up title.

fstyle=*fs*
fs is a binary coded number with each bit controlling one aspect of the font style as follows:
bit 0: Bold.
bit 1: Italic.
bit 2: Underline.
bit 3: Outline (*Macintosh only*).
bit 4: Shadow (*Macintosh only*).
See **Setting Bit Parameters** on page IV-12 for details about bit settings.

help={*helpStr*}
Sets the help for the control. The help text is limited to a total of 255 characters. On Macintosh, the help appears if you turn Igor Tips on. On Windows, the help for the first 127 characters or up to the first line break appears in the status line. If you press F1 while the cursor is over the control, you will see the entire help text. You can insert a line break by putting "\r" in a quoted string.

mode=*m*
Specifies the Pop-up title location.
m=0: Title is in pop-up menu.
m>0: Title is to the left of pop-up menu, the chosen menu item appears in the pop-up menu, and menu item number *m* is initially selected.

noproc
Specifies that no procedure is to execute when choosing in the pop-up menu.

popColor=(*r*,*g*,*b*)
Specifies the color initially chosen in the color pop-up palette. *r*, *g*, and *b* are integers from 0 to 65535. See the **Colors, Color Tables, Line Styles, Markers, and Patterns** section.

popmatch=matchStr
Sets mode to the enabled menu item that matches matchStr. matchStr may be a "wildcard" expression. See **stringmatch**. If no item is matched, mode is unchanged.

popvalue=*valueStr*
Sets the string displayed by the menu when first created, if mode is not zero. See **Popvalue Keyword** section.

pos={*left*,*top*}
Sets the position of the pop-up menu in pixels.

pos+={*dx*,*dy*}
Offsets the position of the pop-up in pixels.

proc=*procName*
Specifies the procedure to execute when the pop-up menu is clicked. See ***Action Procedure*** below.

rename=*newName*
Gives pop-up menu a new name.

size={*width*,*height*}
Sets pop-up menu size in pixels.

title=*titleStr*
Sets title of pop-up menu to the specified string expression. Defaults to "" (no title).
titleStr can contain formatting escape codes in order to create fancy, styled results. The escape codes are the same as used by the **TextBox** operation. The easiest way to generate fancy text is to make selections from the Insert popup in the PopupMenu Control dialog.

userdata(*UDName*)=*UDStr*
Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create. You can retrieve the data using **GetUserData**.

userdata(*UDName*)+=*UDStr*
: Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

value=*itemListSpec*
: Specifies the pop-up menu's items. *itemListSpec* can take several forms as described below under ***Setting The Popup Menu Items***.

win=*winName*
: Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

: When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

The target window, or the window named with the win=*winName* keyword, must be a graph or panel.

Action Procedure

The action procedure is called when the user chooses a menu item. It can take one of two forms.

The original form of the action procedure takes there parameters:

```
Function PopupMenuAction (ctrlName,popNum,popStr) : PopupMenuControl
   String ctrlName
   Variable popNum     // which item is currently selected (1-based)
   String popStr       // contents of current popup item as string
   ...
End
```

The ": PopupMenuControl" designation tells Igor to include this procedure in the list of available popup menu action procedures in the PopupMenu Control dialog used to create a popup menu.

The second form of the action procedure takes one parameter, a reference to the built-in WMPopupAction structure:

```
Function PopupMenuAction(PU_Struct) : PopupMenuControl
   STRUCT WMPopupAction &PU_Struct
   ...
End
```

The WMPopupAction structure has members as described in the following table:

`WMPopupAction` Structure Members

Member	Description
`char ctrlName[MAX_OBJ_NAME+1]`	Control name.
`char win[MAX_WIN_PATH+1]`	Host (sub)window.
`STRUCT Rect winRect`	Local coordinates of host window.
`STRUCT Rect ctrlRect`	Enclosing rectangle of the control.
`STRUCT Point mouseLoc`	Mouse location.
`Int32 eventCode`	Event that executed the procedure. `eventCode` / Event -1 Control being killed 2 Mouse up
`Int32 eventMod`	Bitfield of modifiers. See **Control Structure eventMod Field** on page III-385.
`String userData`	Primary (unnamed) user data. If this changes, it is written back to the control automatically.

WMPopupAction Structure Members

Member	Description
`Int32 blockReentry`	Prevents reentry of control action procedure. See **Control Structure blockReentry Field** on page III-386.
`Int32 popNum`	Item number currently chosen (1-based).
`char popStr[MAXCMDLEN]`	Contents of current popup item.

Action procedures should respond only to documented `eventCode` values. Other event codes may be added along with more fields. Although the return value is not currently used, action procedures should always return zero.

The constants used to specify the size of structure `char` arrays are internal to Igor Pro and may change.

Setting The Popup Menu Items

This section discusses popup menus containing lists of text items. The next section discusses popup menus for choosing colors, line styles, markers and patterns.

The items in the popup menu are determined by the *itemListSpec* parameter used with the value keyword. *itemListSpec* can take several different forms from simple to complex.

No matter what the form, Igor winds up storing an expression that returns a string in the popup menu's internal structure. This expression may be a literal string ("Red;Green;Blue;"), a call to a built-in or user-defined function that returns a string, or the path to a global string variable. Igor evaluates this expression when the popup menu is first created and again each time the user clicks on the menu. You can see the string expression for a given popup menu using the PopupMenu Control dialog.

The right form for *itemListSpec* depends on your application. Here is a guide to choosing the right form with the simpler forms first.

A literal string expression

Use this if you know the items you want in your popup menu when you write the PopupMenu call. For example:

```
Function PopupDemo1()  // Literal string
   NewPanel
   PopupMenu popup0, value="Red;Green;Blue;"
End
```

This method is limited to 400 characters of menu item text.

A function call

Use this if you need to compute the popup menu item list when the user clicks the popup menu. The function must return a string containing a semicolon-separated list of menu items. This example creates a popup menu which displays the name of each wave in the current data folder at the time the menu is clicked:

```
Function PopupDemo2()  // Built-in function
   NewPanel
   PopupMenu popup0, value=WaveList("*", ";", "")
End
```

You can also use a user-defined function. This example shows how to list waves from other than the current data folder:

```
Function/S MyPopupWaveList()
   String saveDF

   // Create some waves for demo purposes
   saveDF = GetDataFolder(1)
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S PopupMenuDemo
   Make/O demo0, demo1, demo2
   SetDataFolder saveDF

   saveDF = GetDataFolder(1)
   SetDataFolder root:Packages:PopupMenuDemo
   String list = WaveList("*", ";", "")
   SetDataFolder saveDF
```

```
    return list
End

Function PopupDemo3()   // User-defined function
    NewPanel
    PopupMenu popup0, value=MyPopupWaveList()
End
```

 # followed by a local string variable specifying items

Use this when the popup menu item list is not known when you write the code but you can compute it at runtime. For example:

```
Function PopupDemo4()   // Local string variable specifying items
    NewPanel
    String quote = "\""
    String list
    if (CmpStr(IgorInfo(2),"Windows") == 0)
        list = quote + "Windows XP; Windows VISTA;" + quote
    else
        list = quote + "Mac OS X 10.4;Mac OS X 10.5;" + quote
    endif
    PopupMenu popup0, value=#list
End
```

The strange-looking use of the quote string variable is necessary because the parameter passed to the value=# keyword is evaluated once when the PopupMenu command executes and the result of that evaluation is evaluated again when the PopupMenu is created or clicked. The result of the first evaluation must be a legal string expression.

This method is limited to 400 characters of menu item text.

 # followed by a local string variable specifying a function

Use this when you need to compute the popup menu item list at click time and you need to select the function which computes the list when the popup menu is created. For example:

```
Function/S WindowsItemList()
    String list
    list = "Windows XP; Windows VISTA;"
    return list
End

Function/S MacItemList()
    String list
    list = "Mac OS X 10.4;Mac OS X 10.5;"
    return list
End

Function PopupDemo5()   // Local string variable specifying function
    String listFunc
    if (CmpStr(IgorInfo(2),"Windows") == 0)
        listFunc = "WindowsItemList()"
    else
        listFunc = "MacItemList()"
    endif
    NewPanel
    PopupMenu popup0, value=#listFunc
End
```

This form is useful when you create a control panel in an independent module. Since the control panel runs in the global name space, you must specify the independent module name in the invocation of the function that provides the popup menu items. For example:

```
// Calling a non-static function in an independent module from #included code
#pragma IndependentModuleName=IM
. . .
String listFunc= GetIndependentModuleName()+"#PublicFunctionInIndepMod()"
PopupMenu popup0, value=#listFunc

// Calling a static function in an independent module from #included code
#pragma IndependentModuleName=IM
#pragma ModuleName=ModName
. . .
String listFunc= GetIndependentModuleName()+"#ModName#StaticFunctionInIndepMod()"
PopupMenu popup0, value=#listFunc
```

We use GetIndependentModuleName rather than hard-coding the name of the independent module so that the code will continue to work if the name of the independent module is changed. Also, because this code does not depend on the specific name of the independent module, it can be added to an independent module via a #included procedure file.

Also see **GetIndependentModuleName** and **Independent Modules and Popup Menus** on page IV-218.

 # followed by a quoted literal path to a global string variable

Use this if you want to compute the popup menu item list before it is clicked, not each time it is clicked. This would be advantageous if it takes a long time to compute the item list, and the list changes only at well-defined times when you can set the global string variable.

The global string variable must exist when the PopupMenu command executes and when the menu is clicked. In this example, the gPopupMenuItems global string variable is created and initialized when the popup menu is created but can be changed to a different value later before the menu is clicked:

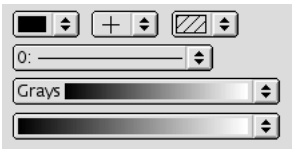
```
Function PopupDemo6()  // Global string variable containing list
   NewDataFolder/O root:Packages
   NewDataFolder/O root:Packages:PopupMenuDemo
   String/G root:Packages:PopupMenuDemo:gPopupMenuItems = "Red;Green;Blue;"

   NewPanel
   PopupMenu popup0 ,value=#"root:Packages:PopupMenuDemo:gPopupMenuItems"
End
```

 # followed by a local string variable containing a path to a global string variable

Use this when the popup menu item list contents will be stored in a global string variable whose location is not known until the popup menu is created. For example:

```
Function PopupDemo7()  // Local string containing path to global string
   String graphName = WinName(0, 1, 1)// Name of top graph
   if (strlen(graphName) == 0)
      Print "There are no graphs."
      return -1
   endif

   NewDataFolder/O root:Packages
   NewDataFolder/O root:Packages:PopupMenuDemo

   // Create data folder for graph
   NewDataFolder/O root:Packages:PopupMenuDemo:$(graphName)

   String list = "Red;Green;Blue;"
   String/G root:Packages:PopupMenuDemo:$(graphName):gPopupMenuItems = list

   NewPanel

   String path         // Local string containing path to global string
   path = "root:Packages:PopupMenuDemo:" + graphName + ":gPopupMenuItems"
   PopupMenu popup0, value=#path

   return 0
End
```

Colors, Color Tables, Line Styles, Markers, and Patterns



You can create PopupMenu controls for color, color tables, line style (dash modes), markers, and patterns. To do so, simply specify the *itemListSpec* parameter to the value keyword as one of `"*COLORPOP*"`, `"*COLORTABLEPOP*"`, `"*COLORTABLEPOPNONAMES*"`, `"*LINESTYLEPOP*"`, `"*MARKERPOP*"`, or `"*PATTERNPOP*"`. In these modes the body of the control will contain a color box, a color table (gradient), a line style sample, a marker, or a pattern sample.

For these special pop-up menus, `mode=0` ("Title in Box" checked) is not used.

For a line style pop-up menu, the mode value is the line style number plus one. Thus line style 0 (a solid line) is `mode=1`.

For a marker pop-up, the mode value is the marker number plus one, and marker 0 (the + marker) is `mode=1`.

For a pattern pop-up, the mode value is the **SetDrawEnv** fillPat number minus 4, so mode=1 corresponds to `fillpat=5`, the SW-NE lines fill pattern shown above.

For a color table pop-up, the mode value is the CTabList() index plus 1, so mode=1 corresponds to the first item in the list returned by **CTabList**, which is "Grays":

```
ControlInfo $ctrlName                              // Sets V_Value
Print StringFromList(V_Value-1,CTabList())         // Prints "Grays"
```

ControlInfo also returns the color table name in S_Value.

To set the pop-up to a given color table name, you can use code like this:

```
Variable m = 1 + WhichListItem(ctabName, CTabList())
PopupMenu $ctrlName mode=m
```

For color pop-up menus, you set the current value using the `popColor=(r,g,b)` keyword. On output (via the popStr parameter of your action procedure or via the S_value output from **ControlInfo**) the color is encoded as "(*r*,*g*,*b*)" where *r*, *g*, and *b* are numbers. To get these numerical values, you can extract them from the string using the MyRGBstrToRGB function below or use ControlInfo which sets V_Red, V_Green, V_Blue. The following example demonstrates the line style and color pop-up menus. To run the example, copy the following code to the procedure window of a new experiment and then run the panel macro.

```
Window Panel0() : Panel
    PauseUpdate; Silent 1      // building window …
    NewPanel /W=(150,50,400,182)
    PopupMenu popup0,pos={74,31},size={96,20},proc=ColorPopMenuProc,title="colors"
    PopupMenu popup0,mode=1,popColor= (0,65535,65535),value= "*COLORPOP*"
    PopupMenu popup1,pos={9,68},size={221,20},proc=LSytlePopMenuProc
    PopupMenu popup1,title="line styles",mode=1,value= "*LINESTYLEPOP*"
EndMacro

Function ColorPopMenuProc(ctrlName,popNum,popStr) : PopupMenuControl
    String ctrlName
    Variable popNum
    String popStr

    Variable r,g,b
    MyRGBstrToRGB(popStr,r,g,b)       // One way to get r, g, b
    print popStr," gives: ",r,g,b

    ControlInfo $ctrlName         // Another way: sets V_Red, V_Green, V_Blue
    Printf "ControlInfo returned (%d,%d,%d)\r", V_Red, V_Green, V_Blue
End

// Take (r,g,b) string and extract out numeric r,g,b values
Function MyRGBstrToRGB(rgbStr,r,g,b)
    String rgbStr
    Variable &r, &g, &b

    r= str2num(rgbStr[1,inf])
    variable spos= strsearch(rgbStr,",",0)
    g= str2num(rgbStr[spos+1,inf])
    spos= strsearch(rgbStr,",",spos+1)
    b= str2num(rgbStr[spos+1,inf])
    return 1
End

Function LSytlePopMenuProc(ctrlName,popNum,popStr) : PopupMenuControl
    String ctrlName
    Variable popNum
    String popStr

    print "style:",popNum-1
End
```

Popvalue Keyword

There are times when the displayed value cannot be determined and saved such that it can be displayed when the pop-up menu is recreated. For instance, because window recreation macros are evaluated in the root folder, a pop-up menu of waves may not contain the correct list when a panel is recreated. That is, the intention may be to have the menu show a particular wave from a data folder other than root. When the panel recreation macro runs, the function that lists waves will list waves in the root data folder. The desired selection may be wrong or nonexistent.

Similarly, a pop-up menu of fonts may need to display a particular font upon recreation on a different computer having a different list of fonts. The mode=*m* keyword probably won't pick the correct font from the new list.

The solution to these problems is to save the correct selection with the popvalue=*valueStr* keyword. The list function will not be executed when the menu is first created. If the menu is popped, the list function will be evaluated, and the correct list will be displayed then.

It is a good idea to set the mode=*m* keyword to the correct number, if it is known. That way, when the menu is popped the correct item is chosen.

Normally you can let Igor redraw the pop-up menu when it redraws the graph or control panel containing it. However, there are situations in which you may want to force the pop-up menu to be redrawn. This can be done using the **ControlUpdate** operation.

See Also

The **ControlInfo** operation for information about the control. The **ControlUpdate**, **WaveList**, and **TraceNameList** operations. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data. **Special Characters in Menu Item Strings** on page IV-114.

PopupMenuControl

PopupMenuControl

PopupMenuControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined pop-up menu control. See **Procedure Subtypes** on page IV-179 for details. See **PopupMenu** for details on creating a popup menu control.

PossiblyQuoteName

PossiblyQuoteName(*nameStr*)

The PossiblyQuoteName function returns the input name string if it conforms to the rules of standard wave or Data Folder names. If it does not, then the name is returned in single quotes. This is used when generating a command string that you will pass to the Execute command. You might get the input name string from a function such as **NameOfWave** or **CsrXWave**.

Examples

```
Print PossiblyQuoteName("wave0")        // prints wave0
Print PossiblyQuoteName("wave 0")       // prints 'wave 0'
```

Details

See **Programming with Liberal Names** on page IV-147 for an example.

Preferences

Preferences [**/Q**] [***newPrefsState***]

The Preferences operation sets or displays the state of user preferences.

User preferences affect the creation of *new* graphs, panels, tables, layouts, notebooks, procedure windows, and the command window. They also affect the appearance of waves appended to graphs and tables, and objects appended to layouts.

Parameters

If *newPrefsState* is present, it sets the state of user preferences as follows:

newPrefsState=0: Preferences off (use factory defaults).

newPrefsState=1: Preferences on.

If *newPrefsState* is omitted, the state of user preferences is printed in the history area.

Flags

/Q Disables printing to the history.

Details

The Preferences operation sets the variable V_flag to the state of user preferences that were in effect *before* the Preferences command executed: 1 for on, 0 for off.

You can also set the state of Preferences with the Misc menu.

Under most circumstances we want procedures to be independent of preferences so that a particular procedure will do the same thing regardless of the state of preferences. To achieve this, preferences are

automatically off when you initiate procedure execution. When execution is complete, the state of preferences is restored to what it was before.

If you want preferences to be in effect during procedure execution, you must turn it on with the Preferences operation.

If the preferences setting is changed by a procedure, the effect of the call is propagated down the calling chain. If a macro changes the preferences setting, that change is undone when the macro returns. If a function changes the preferences setting, the change persists after the function returns. However, even with a function, the changed preferences state does not persist when Igor regains control.

Examples

```
Function Test()
   Variable oldPrefState
   Preferences 1; oldPrefState=V_flag      // remember prefs setting
   Make wave0=x
   Display wave0                 // Display uses preferences
   Preferences oldPrefState      // put prefs back, like a macro would
End
```

See Also

Chapter III-17, **Preferences**.

PrimeFactors

PrimeFactors [/Q] ***inNumber***

PrimeFactors calculates the prime factors of *inNumber*. By default factors are printed in the history and are also stored in the wave W_PrimeFactors in the current data folder.

Flags

/Q Suppresses printing of factors in the history area.

Details

The largest number that this operation can handle is 2^{32}-1.

Print

Print [***flags***] ***expression*** [**,** ***expression***]…

The Print operation prints the evaluated expressions in the history area.

Parameters

An expression can be a wave, a numeric expression (e.g., `3*π/4`), a string expression (e.g., `"Today is " + date()`), or a individual structure element or an entire structure variable.

Flags

/C Evaluates all numeric expressions as complex.

/D Prints a greater number of digits.

/F Prints numeric wave data (1D and 2D waves only) using "nice," easily readable formatting.

/LEN=*len* Sets the string break length to *len* number of characters (default is 200).

/S Obsolete. Numeric results are printed with a moderate number of digits whether you use /S or not. To print more digits, use /D.

/SR Prints a wave subrange for expressions that start as "*waveName*[". Without /SR, such an expression is taken as the start of a numeric expression such as `wave[3]-wave[2]`. (You can still use *wave*[*pnt*] but only if it does not start the numeric expression.)

Wave subrange printing is not done with /F.

You can specify a single row or column using [*r*] syntax. For example, to print column 4 of a matrix, use:

```
Print mymat[][4]
```

Details

Numeric expressions are always evaluated in double precision. The /D flag just controls the number of digits displayed.

Print determines if an expression is real, complex, or string from the first symbol in the expression. Usually this works fine, but occasionally Print guesses wrong and you may have to rearrange your expression. For example:

```
Print 1+cmplx(1,2)
```

will give an error because the first symbol, "1", is real but the expression should be complex. Changing this to

```
Print cmplx(1,2)+1
```

will work.

Print will break long string expressions into multiline pieces. If there are no natural breaks (carriage returns or semicolons) within a default length, then it will break the string arbitrarily.

The default line length is 200 chars or it can be set using the /LEN flag. The maximum number of characters that can be printed on a line in the History area is 400.

When printing waves, you can use either formatted (specified by /F) or unformatted (default) methods. Unformatted output is in an executable syntax for each printed line: `wave={}`.

Note: Executing lines printed from floating point waves will not exactly reproduce the source data due to round-off or insufficient digits in the printed output.

Printing formatted wave data gives easily (human) readable output, and works best for small 1D and 2D waves. If the data are too large or in an unsupported format (3D or greater, or the wave is text), then the output will be unformatted. Formatting is done using spaces, so the output will look best in a fixed-width font.

Printed wave data, both formatted and unformatted, are limited to no more than 100 lines of output. When the line limit is exceeded a warning message will be printed at the end of the truncated output. For text waves, output is limited to 50 characters of each string element, and there is no warning when a string is truncated.

See Also

The **printf** operation.

printf

printf ***formatStr*** [, ***parameter*** [, ***parameter***]…]

The printf operation prints formatted output to the history area.

Parameters

formatStr is a string which specifies the formatting of the output.

The type of the parameter, string or numeric, must agree with the corresponding conversion specification in *formatStr*, or else the results will be indeterminate.

The printf parameters can be numeric or string expressions. Numeric and string structure fields are allowed except that complex structure fields and non-numeric (e.g., WAVE, FUNCREF) structure fields are not allowed.

Details

The *formatStr* contains literal text and conversion specifications.

A conversion specification starts with the % character and ends with a conversion character (for example, g, e, f, d, or s as illustrated below). In between the % and the conversion character you may include one or more flag characters, a field width specifier, and a precision specifier. The first % corresponds to the first parameter, the second % corresponds to the second parameter, etc. If *formatStr* contains no % characters, no parameters are expected.

Here are some simple examples. `numVar` is a numeric variable and `strVar` is a string variable.

```
printf "The answer is: %g\r", numVar
printf "Created wave %s\r", strVar
printf "Created wave %s, %d points\r", strVar, numVar
```

%g is a general-purpose format (floating point or scientific notation) that represents the value of numVar. %d is an integer format that represents the value of `numVar`. %s specifies that the corresponding parameter (strVar) is a string.

The "\r" in these examples appends a carriage return to the end of the printed text.

Here is a complex example using all of these elements of a conversion specification:

```
printf "%+015.4f\r", 1e6*PI
```

This prints:

```
The answer is: +003141592.6536
```

"+" is a flag character that tells printf to put a + or - sign in front of the number.

"015" is a field width specifier that tells printf to print the number in a field of at least 15 characters, padded with leading zeros. Using "15" instead of "015" would cause printf to pad with spaces before the + sign instead of zeros after it.

".4" is a precision specifier that tells printf to print four digits after the decimal point.

"f" tells printf to use a floating point format.

The most common conversions characters are “f” for floating point, “g” for general, “d” for decimal, and “s” for string. They are interpreted as for the printf() function in the C programming language.

The escape codes \t and \r represent the tab and return characters respectively. See **Escape Characters in Strings** on page IV-13 for more information.

The supported flag characters and their meanings are as follows:

-	Left align the result in the field.
+	Put a plus or minus sign before the number.
<space>	Put a space before a positive number.
#	Specifies alternate form for e, f, g, and x formats.

The meaning of the precision specifier depends on the numeric format (%g, %e, %f, %d, etc.) being used:

e, E, f	Precision specifies number of digits after decimal point.
g, G	Precision specifies maximum number of significant digits.
d, o, u, x, X	Precision specifies minimum number of digits.

You can replace both the field width and precision specifiers with an asterisk. This gets the field width or precision specifier from a parameter. For example:

```
printf "%*.*f\r" 4, 3, 1e6*PI
```

means that the field width is 4 and the precision is 3. You could use numeric expressions instead of the literal numbers to control the field width and precision algorithmically.

Here is a complete list of the conversion characters supported by printf:

f	Converts a numeric parameter as [-]ddd.ddd, where the number of digits after the decimal point is determined by the precision specifier and defaults to 6. If the # flag is present, a decimal point will be used even if there are no digits to the right of it.
e, E	Converts a numeric parameter as [-]d.ddde+/-dd, where the number of digits after the decimal point is determined by the precision specifier and defaults to 6. If you use “E” instead of “e” then printf uses a capital “E” in the number. If the # flag is present, a decimal point will be used even if there are no digits to the right of it.
g, G	Converts a numeric parameter using “f” or “e” style conversion depending on the magnitude of the number. “e” is used if the exponent is less than -4 or greater than the precision. “G” uses “f” or “E” style conversion. If the # flag is present, a decimal point will be used even if there are no digits to the right of it and trailing zeros will not be removed.
d, o, u	Converts a numeric parameter as a signed decimal integer, unsigned octal integer or unsigned decimal integer, truncating any fractional part. The precision defaults to one and specifies the minimum number of digits to print.
x, X	Converts a numeric parameter as an unsigned hexadecimal integer, truncating any fractional part. The “x” style uses lower case for the hexadecimal numerals “abcdef” where the “X” style uses upper case. The precision defaults to one and specifies the minimum number of digits to print. If the # flag is present, the string “0x” or “0X” is prepended to the number if it is not zero.

s	Converts a numeric parameter as an unsigned hexadecimal integer, truncating any fractional part. The "x" style uses lower case for the hexadecimal numerals "abcdef" where the "X" style uses upper case. The precision defaults to one and specifies the minimum number of digits to print. If the # flag is present, the string "0x" or "0X" is prepended to the number if it is not zero.
b	WaveMetrics extension. Converts a numeric parameter to binary.
c	Converts a numeric parameter to a single character.
%	Prints a % sign. No parameter is used.
%W	WaveMetrics extension. See description below.

Igor also supports a non-C, WaveMetrics extension to the conversion characters recognized by printf. This conversion specification starts with "%W". It is followed by a flag digit and a format character. For example,

```
printf "%W0Ps", 12.345E-6
```

prints 12.345000μs. In this example, the "%W0" introduces the WaveMetrics conversion specification. The "0" (zero) following the "W" is the flag digit. The "P" that follows is the format specifier character, which prints the number using a prefix, in this case, "μ".

There is only one WaveMetrics format specifier character, "P", which prints using a prefix such as μ, m, k, or M. It recognizes two flag-digits, "0" or "1". Option "0" prints with no space between the numeric part and the prefix character while flag "1" prints with 1 space. Numbers greater than tera or less than femto print using a power of ten notation. Here are a few examples:

```
printf "%.2W0PHz", 12.342E6      // prints 12.34MHz
printf "%.2W1PHz", 12.342E6      // prints 12.34 MHz
printf "%.0W0Ps", 12.342E-6      // prints 12μs
printf "%.0W1Ps", 12.342E-9      // prints 12 ns
```

See Also

The **sprintf**, **fprintf,** and **wfprintf** operations; **Creating Formatted Text** on page IV-230 and **Escape Characters in Strings** on page IV-13.

PrintGraphs

PrintGraphs [***flags***] ***graphSpec*** [**,** ***graphSpec***]…

The PrintGraphs operation prints one or more graphs.

PrintGraphs prints one or more graphs on a single page from the command line or from a procedure. The graphs can be overlaid or positioned any way you want.

Parameters

The *graphSpec* specifies the name of a graph to print, the position of the graph on the page and some other options.

Flags

/C=*num*	Renders graphs in black and white (*num*=0) or in color (*num*=1; default).
/D	Disables high resolution printing. This flag is of use only on Macintosh. It has no effect on Windows.
/G=*grout*	Specifies grout, the spacing between objects, for tiling in prevailing units.
/I	Coordinates are in inches.
/M	Coordinates are in centimeters.
/R	Coordinates are in percent of page size (see **Examples**).
/S	Stacks graphs.
/T	Tiles graphs.

Details

Graph coordinates are in inches (/I) or centimeters (/M) relative to the top left corner of the physical page. If none of these options is present, coordinates are assumed to be in points.

The form of a *graphSpec* is:

graphName [**(*left*, *top*, *right*, *bottom*)**] [**/F=*f***] [**/T**]

Here are some examples:

```
// Take size and position from window size and position.
PrintGraphs Graph0, Graph1

// Specify size and position explicitly.
PrintGraphs/I Graph0(1, 1, 6, 5)/F=1, Graph1(1, 6, 6, 10)/F=1
```

If the coordinates are missing and the /T or /S flags are present before *graphSpec* then the graphs are tiled or stacked. If the coordinates are missing but no /T or /S flags are present then the graph is sized and positioned based on its position on the desktop.

Finally there are these *graphSpec* options, which appear after the graph name:

/F=*f* Specifies a frame around the graph.

- *f*=1: Single frame.
- *f*=2: Double frame.
- *f*=3: Triple frame.
- *f*=4: Shadow box frame.

/T Graph is transparent. This allows special effects when graphs are overlaid.

For this to be effective, the graph and its contents must also be transparent. Graphs are transparent only if their backgrounds are white. Annotations have their own transparent/opaque settings. PICTs may have been created transparent or opaque; an opaque PICT cannot be made transparent.

Examples

You can put an entire *graphSpec* into a string variable and use the string variable in its place. In this case the name of the string variable must be preceded by the $ character. This is handy for printing from a procedure and also keeps the PrintGraphs command down to a reasonable number of characters. For example:

```
String spec0, spec1, spec2
spec0 = "Graph0(1, 1, 6, 5)/F=1"
spec1 = "Graph1(1, 6, 6, 10)/F=1"
spec2 = ""                          // PrintGraphs will ignore spec2.
PrintGraphs/I $spec0, $spec1, $spec2
```

If you use a string for a *graphSpec* and that string contains no characters then PrintGraphs will ignore that *graphSpec*.

PrintLayout

PrintLayout [**/C=*num* /D**] ***winName***

The PrintLayout operation prints the named page layout window.

Parameters

winName is the window name of the page layout to print.

Flags

/C=*num* Renders graphs, tables, and annotations in black-and-white (*num*=0) or in color (*num*=1; default). It has no effect on pictures, which are colored independently.

/D Prints the layout at the default resolution of the output device. Otherwise it is printed at the highest resolution. This flag is of use only on Macintosh. It has no effect on Windows.

Details

Normally page layouts are printed at the highest available resolution of the output device (printer, plotter, or whatever). On Macintosh, it may not work properly at high resolution with some unusual output devices. If this happens, you can try using the /D flag to see if it works properly at the default resolution.

PrintNotebook

PrintNotebook [***flags***] ***notebookName***

The PrintNotebook operation prints the named notebook window.

Parameters

notebookName is either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-95 for details on host-child specifications.

Flags

/B=*hiResMethod*	*Macintosh only*; this flag has no effect on Windows.	
	hiResMethod=1:	Print HiRes PICTs using high resolution bitmaps.
	hiResMethod=0:	Don't print HiRes PICTs using high resolution bitmaps.
	hiResMethod=-1:	Print using the default method. Prints HiRes PICTs using high resolution bitmaps and is the same as method 1.
/P=(*startPage*,*endPage*)	Specifies a page range to print. 1 is the first page.	
/S=*selection*	*selection*=0:	Print entire notebook (default).
	selection=1:	Print selection only.

Details

If no /B flag is given, the default method of handling HiRes PICTs is used (/B=1). Printing of HiRes PICTs is not well supported on the Macintosh, so by default it prints them using temporary high resolution bitmaps. If a future version of the Mac OS improves in this respect, we will change the default method to print directly.

See Also

Chapter III-1, **Notebooks**.

PrintSettings

```
PrintSettings [/I /M /W=winName] [copySource=source, orientation=o,
   margins={left,top,right,bottom}, scale=s, colorMode=m, getPrinterList,
   getPrinter, setPrinter=printerNameStr, getPageSettings, getPageDimensions]
```

The PrintSettings operation gets or sets parameters associated with printing, such as a list of available printers or page setup information for a particular window.

When getting or setting page setup information, PrintSettings acts on a particular window called the destination window. The destination window is the top graph, table, page layout, or notebook window or the window specified by the /W flag.

PrintSettings can not act on page setup records associated with the command window, procedure windows, help windows, control panel, XOP windows, or any type of window other than graphs, tables, page layouts, and notebooks.

The PrintSettings operation services the keywords in the order shown above, not in the order in which they appear in the command. Thus, for example, the getPageSettings and getPageDimensions keywords report the settings after all other keywords are executed.

Flags

/I	Measurements are in inches. If both /I and /M are omitted, measurements are in points.
/M	Measurements are in centimeters. If both /I and /M are omitted, measurements are in points.
/W=*winName*	Acts on the page setup record of the graph, table, page layout, or notebook window identified by *winName*. If *winName* is omitted or if *winName* is `""`, then it used the page setup for the top window.

Keywords

colorMode=*m*	Sets the color mode for the page setup to monochrome (*m*=0) or to color (*m*=1).	
	This keyword does nothing on Macintosh because it is not supported by Mac OS X.	
copySource=*source*	Copies page setup settings from the specified source to the destination window. *source* can be the name of a graph, table, page layout, or notebook window or it can be one of the following special keywords:	
	Default_Settings:	Sets the page setup record to the default for the associated printer as specified by the printer driver.
	Factory_Settings:	Sets the page setup record to the WaveMetrics factory default. This is the page setup you get when creating a new window with user preferences turned off.
	Preferred_Settings:	Sets the page setup record to the user preferred page setup. This is the page setup you get when creating a new window with user preferences turned on. Because there is only one page setup for all

graphs and one page setup for all tables, this has no effect when the destination window is a graph or table. It does work for layouts and notebooks.

getPageDimensions Returns page dimensions via the string variable S_value, which contains keyword-value pairs that can be extracted using **NumberByKey** and **StringByKey**. See **Details** for keyword-value pair descriptions.

getPageSettings Returns page setup settings in the string variable S_value, which contains keyword-value pairs that can be extracted using **NumberByKey** and **StringByKey**. See **Details** for keyword-value pair descriptions.

getPrinter Returns the name of the selected printer for the destination window in the string variable S_value. On Macintosh the returned value will be "" if the setPrinter keyword was never used on the destination window. This means that the window will use the operating system's "current printer".

getPrinterList Returns a semicolon-separated list of printer names in the string variable S_value.

Mac OS X: Returns a list of printers added through Print Center.

Windows: Returns the names of any local printers and names of network printers to which the user has made previous connections.

margins={*left*, *top*, *right*, *bottom*}

Sets the page margins. Dimensions are in points unless /I or /M is used.

The margins are clipped so that they are no smaller than the minimum allowed by the printer driver and no larger than one-half the size of the paper.

The terms *left*, *top*, *right*, and *bottom* refer to the sides of the page after possible rotation for landscape orientation.

Passing zero for all four margins sets the margins to the minimum margin allowed by the printer.

On Macintosh only, passing -1 for all four margins sets the margins to whatever minimum margin is allowed by the printer, even if the printer is changed later. This is how Igor Pro behaved on Macintosh prior to the creation of the PrintSettings operation, when the minimum printer margins were always used.

orientation=*o* Sets the paper orientation to portrait (*o*=0) or to landscape (*o*=nonzero).

scale=*s* Sets the page scaling in percent. *s* is clipped to the range 5 to 5000. A value of 50 prints graphics at one-half the normal size. Some printer drivers do not support scaling in which case the scale keyword does nothing. Many Windows printer drivers that do support scaling still do not work with the scale keyword because their scaling support does not use the standard Windows technique.

setPrinter=*printerNameStr*

Sets the selected printer for the destination window.

SetPrinter attempts to preserve orientation, margins, scale, and color mode but other settings may revert to the default state.

printerNameStr is a name as returned by the getPrinterList keyword and may not be identical to the name displayed in various dialogs. For example, on Mac OS X, the printer name "DESKJET 840C" is returned by getPrinterList as "DESKJET_840C". The latter is the "Queue Name" displayed by the Mac OS X Print Center or Printer Setup Utility programs.

If *printerNameStr* is "", the printer for the destination window is set to the default state. This means different things depending on the operating system:

Mac OS X: The destination window will use the operating system's "current printer", as if the setPrinter keyword had never been used.

Windows: The destination window will use the system default printer.

If you receive an error when using setPrinter, use the getPrinterList keyword to verify that the printer name you are using is correct. Verify that the printer is connected and turned on.

Windows printer names are sometimes UNC names of the form "\\Server\Printer". You must double-up backslashes when using a UNC name in a literal string. See **UNC Paths** on page III-399 for details.

Details

All graphs in the current experiment share a single page setup record so if you change the page setup for one graph, you change it for all graphs.

All tables in the current experiment share a single page setup record.

Each page layout window has its own page setup record.

Each notebook window has its own page setup record.

The keyword-value pairs for the getPageSettings keyword are as follows:

Keyword	Information Following Keyword
ORIENTATION:	0 if the page is in portrait orientation, 1 if it is in landscape orientation.
MARGINS:	The left, top, right, and bottom margins in points, separated by commas.
SCALE:	The page scaling expressed in percent. 50 means that the graphics are drawn at 50% of their normal size.
COLORMODE:	0 for black&white, 1 for color. This is not supported on Macintosh and always returns 1.

The keyword-value pairs for the getPageDimensions keyword are as follows:

Keyword	Information Following Keyword
PAPER:	The left, top, right, and bottom coordinates of the paper in points, separated by commas. The top and left are negative numbers so that the page can start at (0,0).
PAGE:	The left, top, right, and bottom coordinates of the page in points, separated by commas. The term page refers to the part of the paper inside the margins. The top/left corner of the page is always at (0, 0).
PRINTAREA:	The left, top, right, and bottom coordinates of the page in points, separated by commas. The print area is the part of the paper on which printing can occur, as determined by the printer. This is equal to the paper inset by the minimum supported margins. The top and left are negative numbers so that the page can start at (0,0).

Examples

For an example using the PrintSettings operation, see the PrintSettings Tests example experiment file in the "Igor Pro Folder:Examples:Testing & Misc" folder.

Here are some simple examples showing how you can use the PrintSettings operation.

```
Function GetOrientation(name)     // Returns 0 (portrait) or 1 (landscape)
   String name             // Name of graph, table, layout or notebook

   PrintSettings/W=$name getPageSettings
   Variable orientation = NumberByKey("ORIENTATION", S_value)
   return orientation
End

Function SetOrientationToLandscape(name)
   String name             // Name of graph, table, layout or notebook

   PrintSettings/W=$name orientation=1
End

Function/S GetPrinterList()
   PrintSettings getPrinterList
   return S_value
End

Function SetPrinter(destWinName, printerName)
   String destWinName, printerName

   PrintSettings/W=$destWinName setPrinter=printerName
   return 0
End
```

PrintTable

PrintTable [**/P=(*startPage*,*endPage*) /S=*selection*] ***winName***

The PrintTable operation prints the named table window.

Parameters

winName is the window name of the table to print.

Flags

/P=(*startPage*,*endPage*)	Specifies a page range to print. 1 is the first page.	
	If /P is omitted all pages are printed unless /S is used.	
/S=*selection*	*selection*=0:	Print entire table (default).
	selection=1:	Print selection only.

See Also

Chapter II-11, **Tables**.

Proc

Proc *macroName*([*parameters*]) [:*macro type*]

The Proc keyword introduces a macro that does not appear in any menu. Otherwise, it works the same as **Macro**. See **Macro Syntax** on page IV-98 for further information.

ProcedureText

ProcedureText(*macroOrFunctionNameStr* [, *linesOfContext* [, *procedureWinTitleStr*]])

The ProcedureText function returns a string containing the text of the named macro or function as it exists in some procedure file, optionally with additional lines that are before and after to provide context or to collect documenting comments.

Alternatively, all of the text in the specified procedure window can be returned.

Parameters

macroOrFunctionNameStr identifies the macro or function. It may be just the name of a global (nonstatic) procedure, or it may include a module name, such as `"myModule#myFunction"` to specify the static function myFunction in a procedure window that contains a `#pragma ModuleName=myModule` statement.

If *macroOrFunctionNameStr* is set to "", and *procedureWinTitleStr* specifies the title of a single procedure window, then all of the text in the procedure window is returned.

linesOfContext optionally specifies the number of lines around the function to include in the returned string. The default is 0 (no additional contextual lines of text are returned). This parameter is ignored if *macroOrFunctionNameStr* is "" and *procedureWinTitleStr* specifies the title of a single procedure window.

Setting *linesOfContext* to -1 returns lines before the procedure that are not part of the preceding macro or function. Usually these lines are comment lines describing the named procedure. Blank lines are omitted.

Setting *linesOfContext* to a positive number returns that many lines before the procedure and after the procedure. Blank lines are not omitted.

The optional *procedureWinTitleStr* can be the title of a procedure window (such as `"Procedure"` or `"File Name Utilities.ipf"`). The text of the named macro or function in the specified procedure window is returned.

You can use *procedureWinTitleStr* to select one of several static functions with identical names among different procedure windows, even if they do not use a `#pragma moduleName=myModule` statement.

Advanced Parameters

If `SetIgorOption IndependentModuleDev=1`, *procedureWinTitleStr* can also be a title followed by a space and, in brackets, an independent module name. In such cases ProcedureText retrieves function text from the specified procedure window and independent module. (See **Independent Modules** on page IV-214 for independent module details.)

For example, in a procedure file containing:

```
#pragma IndependentModule=myIM
#include <Axis Utilities>
```

A call to ProcedureText like this:

```
String text=ProcedureText("HVAxisList",0,"Axis Utilities.ipf [myIM]")
```

will return the text of the `HVAxisList` function located in the Axis Utilities.ipf procedure window, which is normally a hidden part of the `myIM` independent module.

You can see procedure window titles in this format in the Windows→Procedure Windows menu when `SetIgorOption IndependentModuleDev=1` and when an experiment contains procedure windows that comprise an independent module, as does `#include <New Polar Graphs>`.

procedureWinTitleStr can also be just an independent module name in brackets to retrieve function text from *any* procedure window that belongs to the named independent module:

```
String text=ProcedureText("HVAxisList",0,"[myIM]")
```

See Also

Regular Modules on page IV-212 and **Independent Modules** on page IV-214.

The **WinRecreation** and **FunctionList** functions.

ProcGlobal

ProcGlobal#*procPictureName*

The ProcGlobal keyword is used with Proc Pictures to avoid possible naming conflicts with any other global pictures in the experiment. When you add a picture to an experiment using the Pictures dialog, such a picture is global in scope and may potentially have the same name as a Proc Picture. When a Proc Picture is global (and only then), you should use the ProcGlobal keyword to make sure that the Proc Picture is used with your code and to avoid confusion with pictures in the Pictures dialog.

See Also

See **Proc Pictures** on page IV-43 for details. **Pictures Dialog** on page III-423.

Project

Project [**/C={*long*,*lat*}/M=*method* /P={*p1*,*p2*,…}**] *longitudeWave*, *latitudeWave***

The Project operation calculates projections of XY data, which most often are longitude and latitude waves of geographic coordinates. The output waves are W_XProjection and W_YProjection. Longitude and Latitude are in degrees.

Parameters

longitudeWave is the name of the wave supplying the longitude or equivalent coordinates. *latitudeWave* is the name of the wave supplying the latitude or equivalent coordinates.

Flags

/C={*long*,*lat*} Specifies longitude and latitude center of projection. By default *long*=0 and *lat*=90.

/M=*method* Indicates the type of projection. *method* can be one of the following:

0: Orthographic (default).
1: Stereographic.
2: Gnomonic.
3: General perspective.
4: Lambert equal area.
5: Equidistant.
6: Mercator.
7: Transverse Mercator.
8: Albers Equal Area conic.

/P={*p1*,*p2*,…} One or more parameters required by a particular projection. See the following sections for parameters required by the various projections.

Gnomonic

Here there is one extra parameter that defines the boundaries based on the angle. The specific expression for the limit is that cos(c) in Eq. (5-3) of Snyder is greater than the specified parameter:

```
/P={cos(c)}
```

The actual transformation uses Eqs. (22-4) and (22-5) of Snyder with k' given by (22-3).

General Perspective

Here there is one extra parameter that defines the boundaries based on the angle. The specific expression for the limit is that cos(c) in Eq. (5-3) of Snyder is greater than the specified parameter.

The actual transformation uses Eqs. (22-4) and (22-5) with k' given by (22-3). Here we specify the height H is units of sphere radius. The tilt of the plane is specified by omega and gamma following the notation of Snyder page 175.

The parameters actually specified by the command are:

/P={*H*, *omega*, *gamma*, *deltax*, *deltay* }

H is the height (in radii) above the surface of the earth, *gamma* is the azimuth east of north of the Y axis, and *omega* is the tilt angle or the angle between the projection plane and the tangent plane. The x output will be limited to ± *deltax* and the y output will be limited to the range ± *deltay*.

Mercator

This projection requires the following parameters:

/P={*minLong*, *maxLong*, *minLat*, *maxLat*}

If /P is not specified, the default is {0,360,-90,90}

Note that this projection flips the sign of y when cos(longitude-long_0) changes sign. If you are plotting a continuous path in which consecutive points exhibit the sign change, you should add a NaN entry in the wave so that the path does not wrap.

Albers Equal Area Conic

This projection requires:

/P={*minLong*, *maxLong*, *minLat*, *maxLat*, *Phi1*, *Phi2*}

Phi1 and *Phi2* are the specification of the two standard parallels, the other four parameters determine the boundary of the map area for display.

References

Snyder, John P., *Map Projections—A Working Manual*, U.S.G.S. Professional Paper 1395, U.S. Government Printing Office, Washington D.C., 1987, reprinted 1989, 1994, 1997 with corrections.

Prompt

Prompt *variableName*, *titleStr* [, popup, *menuListStr*]

The Prompt command is used in functions for the simple input dialog and in macros for the missing parameter dialog. Prompt supplies text to describe *variableName* to the user, and optionally provides a pop-up menu of choices for the value of *variableName*.

Parameters

variableName is the name of a macro input parameter or function variable.

titleStr is a string or string expression containing the text to present in the dialog to describe what *variableName* is. This string should be short if the number of items exceeds 5 (when the dialog uses two columns).

The optional keyword popup is used to provide a pop-up list of choices for the values of *variableName*. If popup is used, then *menuListStr* is required.

menuListStr is a string or string expression that contains a semicolon-separated list of choices for the value of *variableName*. If *variableName* is a string, choosing from this list will set the string to the selection. If it is a numeric variable, then it is set to the item number of the selection (if the first item is selected, the numeric variable is set to 1, etc.).

Details

In macros, there must be a blank line after the set of input parameter declarations and prompt statements and there must not be any blank lines within the set.

In user-defined functions, Prompt may be used anywhere within the body of the function, but must precede any DoPrompt that uses the Prompt variable.

menuListStr may be continued on succeeding lines only in macros, as long as no comment is appended to the Prompt line. The additional lines should start with a semicolon, and are appended to the *menuListStr*s on preceding lines.

See Also

For use in user-defined functions, see **The Simple Input Dialog** on page IV-122.

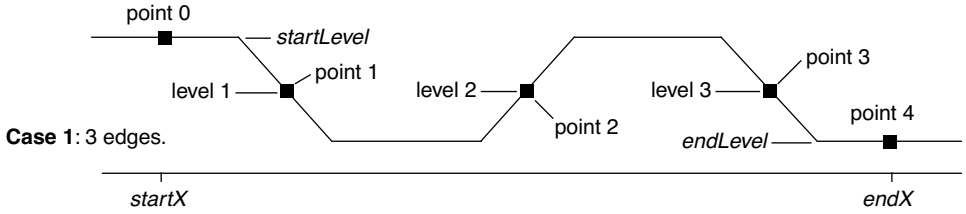
For use in macros, see **The Missing Parameter Dialog** on page IV-101.

For use in functions and macros, see the **DoPrompt** and **popup** keywords.

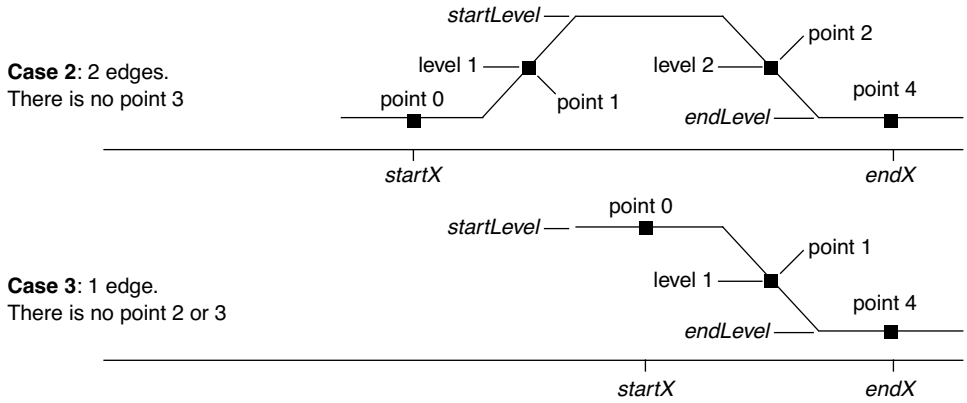
PulseStats

PulseStats [***flags***] ***waveName***

The PulseStats operation produces simple statistics on a region of the named wave that is expected to contain three edges as shown below. If more than three edges exist, PulseStats works on the first three edges it finds.



PulseStats handles other cases in which there are only one or two edges.



Flags

/A=*n* Determines *startLevel* and *endLevel* automatically by averaging *n* points centered at *startX* and *endX*. This does not work in case 2, which requires that you use the /L flag. Default is /A=1.

/B=*box* Sets box size for sliding average. This should be an odd number. If /B=*box* is omitted or *box* equals 1, no averaging is done.

/F=*f* Specifies levels 1, 2, and 3 as a fraction of (*endLevel-startLevel*):

level1 = level2 = level3 = *f* * (*endLevel-startLevel*) + *startLevel*

f must be between 0 and 1. The default value is 0.5 which sets the levels to midway between the base levels.

/L=(*startLevel*, *endLevel*)

Sets *startLevel* and *endLevel* explicitly.

/M=*dx* Sets minimum edge width. Once an edge is found, the search for the next edge starts *dx* units beyond the found edge. Default *dx* is 0.

/P Output edge locations (see **Details**) are set in terms of point number. If /P is omitted, edge locations are set in terms of X values.

/Q Prevents results from being printed in history and prevents error if edge is not found.

/R=(*startX*,*endX*) Specifies an X range of the wave to search. You may exchange *startX* and *endX* to reverse the search direction.

/R=[*startP*,*endP*] Specifies a point range of the wave to search. You may exchange *startP* and *endP* to reverse the search direction.

If you specify the range as /R=[*startP*] then the end of the range is taken as the end of the wave. If /R is omitted, the entire wave is searched.

/T=*dx* Forces search in two directions for a possibly more accurate result. *dx* controls where the second search starts.

Details

The /B=box, /T=dx, /P and /Q flags behave the same as for the **FindLevel** operation.

PulseStats considers a region of the input wave between two X locations, called *startX* and *endX*. *startX* and *endX* are set by the /R=(*startX*,*endX*) flag. If this flag is missing, *startX* and *endX* default to the start and end of the entire wave.

The *startLevel* and *endLevel* values define the base levels of the pulse. You can explicitly set these levels with the /L=(*startLevel*, *endLevel*) flag or you can let PulseStats find the base levels for you by using the /A=*n* flag. With this flag, PulseStats determines *startLevel* and *endLevel* by averaging *n* points centered at *startX* and at *endX*. In case 2, you must use /L=(*startLevel*, *endLevel*) since *startLevel* is not at point 0.

Given *startLevel* and *endLevel* and an *f* value (which you can set with the /F=*f* flag), PulseStats computes level1, level2 and level3 which are always equal. With the default *f* value of 0.5, level1 is midway between *startLevel* and *endLevel*.

With these levels defined, PulseStats searches the wave from *startX* to *endX* looking for one, two or three level crossings. PulseStats sets the following variables:

`V_flag`	0:	All three level crossings were found.
	1:	One or two level crossings were found.
	2:	No level crossings were found.
`V_PulseLoc1`	X location where level1 was found.	
`V_PulseLoc2`	X location where level2 was found.	
`V_PulseLoc3`	X location where level3 was found.	
`V_PulseLvl0`	*startLevel* value.	
`V_PulseLvl123`	Level1 value that is the same as level2 and level3.	
`V_PulseLvl4`	*endLevel* value.	
`V_PulseAmp4_0`	Pulse amplitude (*endLevel - startLevel*).	
`V_PulseWidth2_1`	Left pulse width (x distance between point 2 and point 1).	
`V_PulseWidth3_2`	Right pulse width (x distance between point 3 and point 2).	
`V_PulseWidth3_1`	Pulse period (x distance between point 3 and point 1).	
`V_PulsePolarity`	Trend of the edge at point 1 (-1 if decreasing, +1 if increasing).	

X locations and distances are in terms of the X scaling of the source wave, unless you use the /P flag in which case they are in terms of point number.

If any level crossings are missing then PulseStats sets the associated variables to NaN (Not a Number). If one crossing is missing, variables depending on point 3 are set to NaN. If two crossings are missing, variables depending on points 2 and 3 are set to NaN. If all crossings are missing, variables depending on points 1, 2, and 3 are set to NaN. You can use the numtype function to test a variable to see if it is NaN.

The PulseStats operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

See Also

The **FindLevel** operation about the /B=*box*, /T=*dx*, /P and /Q flags, **EdgeStats** and the **numtype** function.

PutScrapText

`PutScrapText` *`textStr`*

The PutScrapText operation places *textStr* on the Clipboard (aka "scrap"). This text will be used when the user subsequently chooses Paste from the Edit menu.

Details
All contents of the Clipboard (including pictures) are cleared before the text is placed there.

Examples
Put two lines of text into the Clipboard:

```
String text = "This is the first line.\rAnd this is the second."
PutScrapText text
```

Empty the Clipboard:

```
PutScrapText ""
```

See Also
The **GetScrapText** function and the **SavePICT** operation.

pwd

pwd
The pwd operation prints the full path of the current data folder to the history area. It is equivalent to Print GetDataFolder(1).

pwd is named after the UNIX "print working directory" command.

See Also
GetDataFolder, **cd**, **Dir**, **Data Folders** on page II-121

q

q
The q function returns the current column index of the destination wave when used in a multidimensional wave assignment statement. The corresponding scaled column index is available as the **y** function.

Details
Unlike **p**, outside of a wave assignment statement, q does not act like a normal variable.

See Also
Waveform Arithmetic and Assignments on page II-93.

For other dimensions, the **p**, **r**, and **s** functions.

For scaled dimension indices, the **x**, **y**, **z** and **t** functions.

qcsr

qcsr(*cursorName* [, *graphNameStr*])
The qcsr function can be used with cursors on images or waterfall plots to return the column number. It can also be used with free cursors to return the relative Y coordinate.

Parameters
cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

See Also
The **Cursor** operation and the **CsrInfo**, **hcsr**, **pcsr**, **vcsr**, **xcsr**, and **zcsr** functions.

Quit

Quit [/N/Y]
The Quit operation quits Igor Pro.

Flags

/N	Quits without saving changes and without dialog.
/Y	Saves current experiment before quitting without putting up dialog unless current experiment is “Untitled”.

r

r

The r function returns the current layer index of the destination wave when used in a multidimensional wave assignment statement. The corresponding scaled layer index is available as the **z** function.

Details

Unlike **p**, outside of a wave assignment statement, r does not act like a normal variable.

See Also

Waveform Arithmetic and Assignments on page II-93. For other dimensions, the **p**, **q**, **s**, and **t** functions. For scaled dimension indices, the **x**, **y**, **z**, and **t** functions.

r2polar

r2polar(*z*)

The r2polar function returns a complex value in polar coordinates derived from the complex value *z*, which is assumed to be in rectangular coordinates. The magnitude is stored in the real part and the angle (in radians) is stored in the imaginary part of the returned complex value.

Examples

Assume waveIn and waveOut are complex.

```
waveOut= r2polar(waveIn)
```

sets each point of waveOut to the polar coordinates derived from the real and imaginary parts of waveIn.

You may get unexpected results if the number of points in waveIn differs from the number of points in waveOut.

See Also

The functions **cmplx**, **conj**, **imag**, **p2rect**, and **real**.

RatioFromNumber

RatioFromNumber [***flags***] ***num***

The RatioFromNumber operation computes two integers whose ratio is equal to *num* ± *maxError* (/MERR flag). The ratio is returned in V_numerator and V_denominator.

Parameters

num is the number to approximate by V_numerator/V_denominator.

Flags

/MERR=*maxError*	Specifies the maximum tolerable error. The computed ratio differs from *num* by no more than *maxError* (default value is *num**1e-6). *maxError* must be a value between 0 and *num*. See Details about setting *maxError* to 0.
/MITS = *maxIterations*	Keeps returned values small by specifying a small number for *maxIterations*. *maxIterations* must be a value between 1 and 32767 (default is 100).
/V[=*v*]	Prints output variables to history. *v*=1: Prints variables (same as /V). *v*=0: Nothing printed (same as no /V).

Details

The ratio is computed by continued fraction expansion and recurrence relations for the convergents and checking num - (V_numerator/V_denominator) against *maxError*.

Setting *maxError* = 0 computes a maximally accurate ratio. The returned values can be surprisingly large:

```
RatioFromNumber/V/MERR=0 (1/1666)
  V_numerator= 4398046511104; V_denominator= 7.3271454874993e+15;
  ratio= 0.00060024009603842; V_difference= 0;
```

Using the default /MERR returns the expected 1 and 1666. The difference is attributable to floating-point roundoff errors.

The ratio is computed by continued fraction expansion and recurrence relations for the convergents and checking *num* - (V_numerator/V_denominator) against /MERR.

Prior to Igor Pro 6.22, RatioFromNumber iterated one less time than specified by *maxIterations*. This was corrected in Igor Pro 6.22.

Output Variables

RatioFromNumber sets the following output variables:

`V_difference` V_numerator/V_denominator - *num* (positive if the approximation is too big).

`V_flag` 0: V_difference less than or equal to /MERR.
1: V_difference greater than /MERR.

`V_numerator, V_denominator`
Values for the numerator and denominator. The ratio of V_numerator/V_denominator approximates *num*.

`V_iterations` The number of iterations actually used.

Examples

```
RatioFromNumber/V pi
  V_numerator= 355; V_denominator= 113; ratio= 3.141592920354;
  V_difference= 2.667641894049 7e-07; V_iterations= 3;
RatioFromNumber/V/MITS=2 pi
  V_numerator= 22; V_denominator= 7; ratio= 3.1428571428571;
  V_difference= 0.0012644892673497; V_iterations= 1;
```

See Also

The **gcd** and **trunc** functions.

Rect

The Rect structure is used as a substructure usually to store the coordinates of a window or control.

```
Structure Rect
   Int16 top
   Int16 left
   Int16 bottom
   Int16 right
EndStructure
```

ReadVariables

ReadVariables

The ReadVariables operation reads variables into an experiment.

ReadVariables is used automatically when you open an experiment. You need not invoke it.

real

real(*z*)

The real function returns the real component of the complex value *z*.

See Also

The functions **cmplx**, **conj**, **imag**, **p2rect**, and **r2polar**.

Redimension

Redimension [***flags***] ***waveName*** [**,** ***waveName***]…

The Redimension operation remakes the named waves, preserving their contents as much as possible.

Flags

/B Converts waves to 8-bit signed integer or unsigned integer if /U is present.

/C Converts real waves to complex.

/D Converts single precision waves to double precision.

/E=*e* *e*=0: No special action (default).
e=1: Force reshape without converting or moving data.
e=2: Perform endian swap. (See **FBinRead** for a discussion of endian byte ordering.)

/I Converts waves to 32-bit signed integer or unsigned integer if /U is present.

/N=*n* *n* is the new number of points each wave will have. Multidimensional waves are converted to 1 dimension.

/N=(*n1*, *n2*, *n3*, *n4*)
n1, *n2*, *n3*, *n4* specify the number of rows, columns, layers, and chunks each wave will have. Trailing zeros can be omitted (e.g., /N=(*n1*, *n2*, 0, 0) can be abbreviated as /N=(*n1*, *n2*)). If any dimension size is to remain unchanged, pass -1 for that dimension.

/R Converts complex waves to real by discarding the imaginary part.

/S Converts double precision waves to single precision.

/U Converts integer waves to unsigned.

/W Converts waves to 16-bit integer (unsigned integer if /U is present).

/Y=*type* Specifies wave data type. See details below.

Wave Data Types

As a replacement for the above number type flags you can use /Y=*numType* to set the number type as an integer code. See the **WaveType** function for code values. Do not use /Y in combination with other type flags. This technique cannot be used to change the number type without changing the real/complex setting.

Details

The waves must already exist. New points in waves that are extended are zeroed.

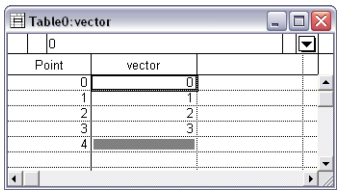
In general, Redimension does not move data from one dimension to another. For instance, if you have a 6x6 matrix wave, and you would like it to be 3x12, the rows have been shortened and the data for the last three rows is lost.

As a special case, if converting to or from a 1D wave, Redimension will leave the data in place while changing the dimensionality of the wave. For example, you can use Redimension to convert a 36-element 1D wave into a 6x6 matrix in which the elements in the first column (column 0) are the first 6 elements of the 1D wave, the elements of the second column are the next 6, etc. When redimensioning from a 1D wave, columns are filled first, then layers, followed by chunks.

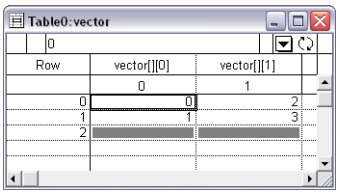
Examples

Reshaping a 1D wave having 4 elements to make a 2x2 matrix:

```
Make/N=4 vector=x
```




```
Redimension/N=(2,2) vector
```



See Also

Lists of Values on page II-96; the **DeletePoints** and **Make** operations.

Remove

Remove

When interpreting a command, Igor treats the Remove operation as **RemoveFromGraph**, **RemoveFromTable**, or **See Also**, depending on the target window. This does not work when executing a user-defined function. Therefore, we recommend that you use **RemoveFromGraph**, **RemoveFromTable**, or **RemoveLayoutObjects** rather than Remove.

RemoveByKey

RemoveByKey(*keyStr*, *kwListStr* [, *keySepStr* [, *listSepStr* [, *matchCase*]]])

The RemoveByKey function returns *kwListStr* after removing the keyword-value pair specified by *keyStr*. *kwListStr* should contain keyword-value pairs such as `"KEY=value1,KEY2=value2"` or `"Key:value1;KEY2:value2"`, depending on the values for *keySepStr* and *listSepStr*.

Use RemoveByKey to remove information from a string containing a `"key1:value1;key2:value2;"` or `"key1=value1,key2=value2,"` style list such as those returned by functions like **AxisInfo** or **TraceInfo**.

If *keyStr* is not found then *kwListStr* is returned unchanged.

keySepStr, *listSepStr*, and *matchCase* are optional; their defaults are ":", ";", and 0 respectively.

Details

keyStr is limited to 255 characters.

kwListStr is searched for an instance of the key string bound by *listSepStr* on the left and a *keySepStr* on the right. The key, the *keySepStr*, and the text up to and including the next *listSepStr* (if any) are removed from the returned string.

If the resulting string contains only *listSepStr* characters, then an empty string ("") is returned.

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are usually case-insensitive. Setting the optional *matchCase* parameter to 1 makes the comparisons case sensitive.

Only the first characters of *keySepStr* and *listSepStr* are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *matchCase* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print RemoveByKey("AKEY", "AKEY:123;BKEY:val")    // prints "BKEY:val"
Print RemoveByKey("AKEY", "akey=1;BK=b;", "=")    // prints "BK:b;"
Print RemoveByKey("AKEY", "AKEY=1,BK=b,", "=", ",")// prints "BK:b,"
Print RemoveByKey("ckey","CKEY:1;BKEY:2")         // prints "BKEY:2"
Print RemoveByKey("ckey","CKEY:1;BKEY:2",":",";",1)// prints "AKEY:1;BKEY:2"
```

See Also

The **NumberByKey**, **StringByKey**, **ReplaceNumberByKey**, **ReplaceStringByKey**, **ItemsInList**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

RemoveContour

RemoveContour [**/W=*winName***] ***contourInstanceName*** [, ***contourInstanceName***]…

The RemoveContour operation removes the traces, and releases memory associated with the contour plot of *contourInstanceName* in the target or named graph.

Parameters

contourInstanceName is usually simply the name of a wave. More precisely, *contourInstanceName* is a wave name, optionally followed by the # character and an instance number to identify which contour plot of a given wave is to be removed.

Flags

/W=winName Removes contours from the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

If the axes used by the contour plot are no longer in use, they will also be removed.

An contour instance name in a string can be used with the $ operator to specify *imageInstance*.

Examples

```
Display;AppendMatrixContour zw       //new graph, contour of zw matrix
AppendMatrixContour zw               //two contours of zw
RemoveContour zw#1                   //remove the second contour
```

See Also

The **AppendMatrixContour** and **AppendXYZContour** operations.

RemoveEnding

RemoveEnding(*str* [, *endingStr*])

The RemoveEnding function removes one character from the end of *str*, or it removes the *endingStr* from the end of *str*.

endingStr is optional. If missing, one character is removed from the end of *str*.

Details

endingStr is compared to the end of *str* using a case insensitive comparison (such as cmpstr uses). If the end of *str* does not match *endingStr*, the unaltered *str* is returned.

Examples

```
Print RemoveEnding("123")                    // prints "12"
Print RemoveEnding("no semi" , ";")          // prints "no semi"
Print RemoveEnding("trailing semi;" , ";")   // prints "trailing semi"
Print RemoveEnding("file.txt" , ".TXT")      // prints "file"
```

See Also

The **cmpstr** and **ParseFilePath** functions.

RemoveFromGraph

RemoveFromGraph [**/W=*winName*/Z**] ***traceName*** [**, *traceName***]…

The RemoveFromGraph operation removes the specified wave traces from the target or named graph. A trace is a representation of the data in a wave, usually connected line segments.

Parameters

traceName is usually just the name of a wave.

More generally, *traceName* is a wave name, optionally followed by the # character and an instance number - for example, wave0#1. See **Instance Notation** on page IV-16 for details.

Flags

/W=*winName* Removes traces from the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/Z Suppresses errors if specified trace or image is not on the graph.

Details

Up to 100 *traceName*s may be specified, subject to the 400 character command limit.

If the axes used by the given trace are not in use after removing the trace, they will also be removed.

A string containing a trace name can be used with the $ operator to specify *traceName*.

Specifying $"#0" for *traceName* removes the first trace in the graph. $"#1" removes the second trace in the graph, and so on. $"" is equivalent to $"#0".

Note that removing all the contour traces from a contour plot is not the same as removing the contour plot itself. Use the **RemoveContour** operation.

Examples

The command:

```
Display myWave,myWave;Modify mode(myWave#1)=6
```

appends two instances of myWave to the graph.The first/backmost instance of myWave is instance 0, and its trace name is just myWave as a synonym for myWave#0. The second or frontmost instance of myWave is myWave#1 and it is displayed with the cityscape mode.

To remove the second instance from the graph requires the command:

```
RemoveFromGraph myWave#1
```

or

```
String MyTraceName="myWave#1"
RemoveFromGraph $MyTraceName
```

See Also

Instance Notation on page IV-16.

RemoveFromLayout

RemoveFromLayout ***objectSpec*** [**,** ***objectSpec***]…

Deprecated — use **RemoveLayoutObjects**.

The RemoveFromLayout operation removes the specified objects from the top layout.

Parameters

objectSpec is either an object name (e.g., Graph0) or an *objectName* with an instance (e.g., Graph0#1). An instance is needed only if the same object appears in the layout more than one time. Graph0 is equivalent to Graph0#0 and Graph0#1 refers to the second instance of Graph0 in the layout.

See Also

The **RemoveLayoutObjects** operation.

RemoveFromList

RemoveFromList(*itemOrListStr, listStr* [, *listSepStr* [, *matchCase*]])

The RemoveFromList function returns *listStr* after removing the item or items specified by *itemOrListStr*. *listStr* should contain items separated by the *listSepStr* character, such as "abc;def;".

If *itemOrListStr* contains multiple items, they should be separated by the *listSepStr* character, too.

Use RemoveFromList to remove item(s) from a string containing a list of items separated by a single character, such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

If all items in *itemOrListStr* are not found or if any of the arguments is `""` then *listStr* is returned unchanged (unless *listStr* contains only list separators, in which case an empty string is returned).

listSepStr and *matchCase* are optional; their defaults are ";" and 1 respectively.

Details

itemStr may have any length.

listStr is searched for an instance of the item string(s) bound by *listSepStr* on the left and right. The item and any trailing *listSepStr* (if any) are removed from the returned string.

If the resulting string contains only *listSepStr* characters, then an empty string (`""`) is returned.

listStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *listSepStr* are case-sensitive. Searches for *items* in *itemOrListStr* are usually case-sensitive. Setting the optional *matchCase* parameter to 0 makes the comparisons case insensitive.

Only the first character of *listSepStr* is used.

If *matchCase* is specified, then *listSepStr* must also be specified.

Examples

```
Print RemoveFromList("wave1", "wave0;wave1;")         // prints "wave0;"
Print RemoveFromList("wave1", ";wave1;;;;")           // prints ""
Print RemoveFromList("KEY=joy", "AX=3,KEY=joy", ",")  // prints "AX=3,"
Print RemoveFromList("fred", "fred\twilma", "\t")     // prints "wilma"
Print RemoveFromList("fred;barney","fred;wilma;barney")// prints "wilma;"
Print "X"+RemoveFromList("",";;;;")+"Y"              // prints "XY"
Print RemoveFromList("FRED", "fred;wilma")         // prints "fred;wilma"
Print RemoveFromList("FRED", "fred;wilma", ";", 0)    // prints "wilma"
```

See Also

The **FindListItem**, **FunctionList**, **ItemsInList**, **RemoveByKey**, **RemoveListItem**, **StringFromList**, **StringList**, **TraceNameList**, **UpperStr**, **VariableList**, and **WaveList** functions.

RemoveFromTable

RemoveFromTable [**/W=*winName***] ***columnSpec*** [**,** ***columnSpec***]…

The RemoveFromTable operation removes the specified columns from the top table.

Parameters

*columnSpec*s are the same as for the **Edit** operation; usually they are just the names of waves.

Flags

/W=*winName* Removes columns from the named table window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

See Also

Edit about *columnSpec*s, and **AppendToTable**.

RemoveImage

RemoveImage [**/W=*winName*/Z**] ***imageInstance*** [, ***imageInstance***]…

The RemoveImage operation removes the given image from the target or named graph.

Parameters

imageInstance is usually simply the name of a wave. More precisely, *imageInstance* is a wave name, optionally followed by the # character and an instance number to identify which image of a given wave is to be removed.

Flags

/W=*winName* Removes an image from the named graph window or subwindow. When omitted, action will affect the active window or subwindow. Must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/Z Suppresses errors if specified image is not on the graph.

Details

If the axes used by the given image are not in use after removing the image, they will also be removed.

An image name in a string can be used with the $ operator to specify *imageInstance*.

See Also

The **AppendImage** operation.

RemoveLayoutObjects

RemoveLayoutObjects [**/W=*winName*/Z**] ***objectSpec*** [**, *objectSpec***]

The RemoveLayoutObjects operation removes the specified object or objects from the top page layout, or from the layout specified by the /W flag.

Unlike the RemoveFromLayout operation, RemoveLayoutObjects can be used in user-defined functions. Therefore, RemoveLayoutObjects should be used in new programming.

Parameters

objectSpec is either an object name (e.g., Graph0) or an *objectName* with an instance (e.g., Graph0#1). An instance is needed only if the same object appears in the layout more than one time. Graph0 is equivalent to Graph0#0 and Graph0#1 refers to the second instance of Graph0 in the layout.

Flags

/W=*winName* *winName* is the name of the page layout window from which the object is to be removed. If /W is omitted or if *winName* is `$""`, the top page layout is used.

/Z Does not report errors if the specified layout object does not exist.

See Also

NewLayout, **AppendLayoutObject** and **ModifyLayout**.

RemoveListItem

RemoveListItem(*itemNum, listStr* [*, listSepStr*])

The RemoveListItem function returns *listStr* after removing the item specified by the list index *itemNum*. *listStr* should contain items separated by the *listSepStr* character, such as "abc;def;". *itemNum* should be a number between 0 (the index of the first item in a list) and ItemsInList(*listStr*) - 1.

RemoveListItem removes an item from a string containing a list of items separated by a single character, such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

listSepStr is optional. If missing, *listSepStr* is presumed to be ";".

Details

RemoveListItem differs from **RemoveFromList** in that RemoveListItem specifies the item to be removed by index and removes only that item, while RemoveFromList specifies the item to be removed by value, and removes all matching items.

If *itemNum* less than 0 or greater than ItemsInList(*listStr*) - 1, or if *listSepStr* is `""` then *listStr* is returned unchanged (unless *listStr* contains only list separators, in which case an empty string is returned).

If the resulting string contains only *listSepStr* characters, then an empty string (`""`) is returned.

listStr is treated as if it ends with a *listSepStr* even if it doesn't. Searches for *itemStr* and *listSepStr* are case-sensitive.

Only the first character of *listSepStr* is used.

Examples

```
Print RemoveListItem(1, "wave0;wave1;w2;")        // prints "wave0;w2;"
Print RemoveListItem(0, "wave1;;;;")              // prints ""
Print RemoveListItem(1, "AX=3,KEY=joy", ",")      // prints "AX=3,"
Print RemoveListItem(1, "fred\twilma", "\t")      // prints "fred\t"
```

See Also

The **AddListItem**, **FindListItem**, **FunctionList**, **ItemsInList**, **RemoveByKey**, **RemoveFromList**, **StringFromList**, **StringList**, **TraceNameList**, **VariableList**, **WaveList**, and **WhichListItem** functions.

RemovePath

RemovePath [**/A/Z**] ***pathName***

The RemovePath operation removes a path from the list of symbolic paths. RemovePath is an old name for the new **KillPath** operation, which we recommend you use instead.

Rename

Rename *oldName, newName*

The Rename operation renames waves, strings, or numeric variables from *oldName* to *newName*.

Parameters

oldName may be a simple object name or a data folder path and name. *newName* must be a simple object name.

Details

You can not rename an object using a name that already exists. The following will result in an error:

```
Make wave0, wave1
// Rename wave0 and overwrite wave1.
Rename wave0, wave1           // This will not work.
```

However, you can achieve the desired effect as follows:

```
Make wave0, wave1
Duplicate/O wave0, wave1; KillWaves wave0
```

See Also

The **Duplicate** operation.

RenameDataFolder

RenameDataFolder *sourceDataFolderSpec, newName*

The RenameDataFolder operation changes the name of the source data folder to the new name.

sourceDataFolderSpec can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

newName is just the new name for the data folder, without any path.

Details

RenameDataFolder generates an error if the new name is already in use as a data folder contained within the source data folder.

Examples

```
RenameDataFolder root:foo,foo2       // Change name of foo to foo2
```

See Also

Chapter II-8, **Data Folders**.

RenamePath

RenamePath *oldName, newName*

The RenamePath operation renames an existing symbolic path from *oldName* to *newName*.

See Also

Symbolic Paths on page II-34

RenamePICT

RenamePICT *oldName, newName*

The RenamePICT operation renames an existing picture to from *oldName* to *newName*.

See Also

Pictures on page III-421.

RenameWindow

RenameWindow *oldName, newName*

The RenameWindow operation renames an existing window or subwindow from *oldName* to *newName*.

Parameters

oldName is the name of an existing window or subwindow.

When identifying a subwindow with *oldName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

See Also

The **DoWindow** operation.

ReorderImages

ReorderImages [**/W=*winName***] ***anchorImage*, {*imageA*, *imageB*, …}**

The ReorderImages operation changes the ordering of graph images to that specified in the braces.

Flags

/W=*winName* Reorders images in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

Igor keeps a list of images in a graph and draws the images in the listed order. The first image drawn is consequently at the bottom. All other images are drawn on top of it. The last image is the top one; no other image obscures it.

ReorderImages works by removing the images in the braces from the list and then reinserting them at the location specified by *anchorImage*. If *anchorImage* is not in the braces, the images in braces are placed before *anchorImage*.

If the list of images is A, B, C, D, E, F, G and you execute the command

```
ReorderImages F, {B,C}
```

images B and C are placed just before F: A, D, E, **B**, **C**, **F**, G.

The result of

```
ReorderImages E, {D,E,C}
```

is to reorder C, D and E and put them where E was. Starting from the initial ordering this gives A, B, **D**, **E**, **C**, F, G.

ReorderImages generates an error if the same trace is in the list twice.

See Also

The **ReorderTraces** operation.

ReorderTraces

ReorderTraces [**/W=*winName***] ***anchorTrace*, {*traceA*, *traceB*, …}**

The ReorderTraces operation changes the ordering of graph traces to that specified in the braces.

Flags

/W=winName Reorders traces in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

Igor keeps a list of traces in a graph and draws the traces in the listed order. The first trace drawn is consequently at the bottom. All other traces are drawn on top of it. The last trace is the top one; no other trace obscures it.

ReorderTraces works by removing the traces in the braces from the list and then reinserting them at the location specified by *anchorTrace*. If *anchorTrace* is not in the braces, the traces in braces are placed before *anchorTrace*.

If the list of traces is A, B, C, D, E, F, G and you execute the command

```
ReorderTraces F, {B,C}
```

traces B and C are placed just before F: A, D, E, **B**, **C**, **F**, G.

The result of

```
ReorderTraces E, {D,E,C}
```

is to reorder C, D and E and put them where E was. Starting from the initial ordering results in A, B, **D**, **E**, **C**, F, G.

ReorderTraces generates an error if the same trace is in the list twice.

See Also

The **ReorderImages** operation.

ReplaceNumberByKey

**ReplaceNumberByKey(*keyStr*, *kwListStr*, *newNum* [, *keySepStr*
[, *listSepStr* [, *case*]]])**

The ReplaceNumberByKey function returns *kwListStr* after replacing the numeric value of the keyword-value pair specified by *keyStr*. *kwListStr* should contain keyword-value pairs such as `"KEY=value1,KEY2=value2"` or `"Key:value1;KEY2:value2"`, depending on the values for *keySepStr* and *listSepStr*.

Use ReplaceNumberByKey to add or modify numeric information in a string containing a `"key1:value1;key2:value2;"` style list such as those returned by functions like **AxisInfo** or **TraceInfo**.

If *keyStr* is not found in *kwListStr*, then the key and the value are appended to the end of the returned string.

keySepStr, *listSepStr*, and *case* are optional; their defaults are ":", ";", and 0 respectively.

Details

The actual string appended is:

[*listSepStr*] *keyStr keySepStr newNum listSepStr*

The optional leading list separator *listSepStr* is added only if *kwListStr* does not already end with a list separator.

keyStr is limited to 255 characters.

kwListStr is searched for an instance of the key string bound by *listSepStr* on the left and a *keySepStr* on the right. The text up to the next ";" is replaced by *newNum* after conversion to text using the %.15g format (see **printf** for format conversion specifications).

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are usually case-insensitive. Setting the optional *case* parameter to 0 makes the comparisons case sensitive.

Only the first characters of *keySepStr* and *listSepStr* are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *case* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print ReplaceNumberByKey("K1", "K1:7;", 4)            // prints "K1:4;"
Print ReplaceNumberByKey("k2", "K2=8;", 5, "=")       // prints "K2=5;"
Print ReplaceNumberByKey("K3", "K3:9,", 6, ":", ",")  // prints "K3:6,"
Print ReplaceNumberByKey("k3", "K0:9", 6, ":", ",")   // prints "K0:9,k3:6,"
Print ReplaceNumberByKey("k3", "K3:9,", 6, ":", ",")  // prints "K3:9,"
Print ReplaceNumberByKey("k3", "K3:9,", 6, ":", ",", 1)// prints "k3:9,k3:6,"
```

See Also

The **ReplaceStringByKey**, **NumberByKey**, **StringByKey**, **RemoveByKey**, **ItemsInList**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

ReplaceString

ReplaceString(*replaceThisStr, inStr, withThisStr* [, *caseSense* [, *maxReplace*]])

The ReplaceString function returns *inStr* after replacing any instance of *replaceThisStr* with *withThisStr*.

The comparison of *replaceThisStr* to the contents of *inStr* is case-insensitive. Setting the optional *caseSense* parameter to nonzero makes the comparison case-sensitive.

Usually all instances of *replaceThisStr* are replaced. Setting the optional *maxReplace* parameter limits the replacements to that number.

Details

If *replaceThisStr* is not found, *inStr* is returned unchanged.

If *maxReplace* is less than 1, then no replacements are made. Setting *maxReplace* = Inf is the same as omitting it.

Examples

```
Print ReplaceString("hello", "say hello", "goodbye")// prints "say goodbye"
Print ReplaceString("\r\n", "line1\r\nline2", "") // prints "line1line2"
Print ReplaceString("A", "an Ack-Ack", "a", 1)    // prints "an ack-ack"
Print ReplaceString("A", "an Ack-Ack", "a", 1, 1) // prints "an ack-Ack"
Print ReplaceString("", "input", "whatever")   // prints "input" (no change)
```

See Also

The **ReplaceStringByKey**, **cmpstr**, **stringmatch**, and **strsearch** functions.

ReplaceStringByKey

ReplaceStringByKey(*keyStr, kwListStr, newTextStr* [, *keySepStr* [, *listSepStr* [, *matchCase*]]])

The ReplaceStringByKey function returns *kwListStr* after replacing the text value of the keyword-value pair specified by *keyStr*. *kwListStr* should contain keyword-value pairs such as `"KEY=value1,KEY2=value2"` or `"Key:value1;KEY2:value2"`, depending on the values for *keySepStr* and *listSepStr*.

Use ReplaceStringByKey to add or modify text information in a string containing a `"key1:value1;key2:value2;"` style list such as those returned by functions like **AxisInfo** or **TraceInfo**.

If *keyStr* is not found in *kwListStr*, then the key and the value are appended to the end of the returned string.

keySepStr, *listSepStr*, and *matchCase* are optional; their defaults are ":", ";", and 0 respectively.

Details

The actual string appended is:

[*listSepStr*] *keyStr keySepStr newTextStr listSepStr*

The optional leading list separator *listSepStr* is added only if *kwListStr* does not already end with a list separator.

keyStr is limited to 255 characters.

kwListStr is searched for an instance of the key string bound by a ";" on the left and a ":" on the right. The text up to the next ";" is replaced by *newTextStr*.

If *newTextStr* is `""`, any existing value is deleted, but the key, the key separator, and the list separator are retained. To remove a keyword-value pair, use the **RemoveByKey** function.

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are case-insensitive. Setting the optional *matchCase* parameter to 1 makes the comparisons case-sensitive.

Only the first characters of *keySepStr* and *listSepStr* are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *matchCase* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print ReplaceStringByKey("KY", "KY:a;KZ:c", "b")      // prints "KY:b;KZ:c"
Print ReplaceStringByKey("KY", "ky=a;", "b", "=")     // prints "ky=b;"
Print ReplaceStringByKey("KY", "KY:a,", "b", ":", ",")// prints "KY:b,"
Print ReplaceStringByKey("ky", "ZZ:a,", "b", ":", ",")// prints "ZZ:a,ky:b,"
Print ReplaceStringByKey("kz", "KZ:a,", "b", ":", ",")// prints "KZ:b,"
Print ReplaceStringByKey("kz", "KZ:a,", "b", ":", ",", 1)// prints "KZ:a,kz:b,"
```

See Also

The **ReplaceString**, **ReplaceNumberByKey**, **NumberByKey**, **StringByKey**, **ItemsInList**, **RemoveByKey**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

ReplaceText

ReplaceText [**/W=*winName*/N=*name***] ***textStr***

The ReplaceText operation replaces the text in the most recently created or changed annotation or in the annotation specified by /W=*winName* and/N=*name*.

Parameters

textStr can contain escape codes to control font, font size and other stylistic variations.

If the annotation is a color scale, this command replaces the text of the color scale's main axis label.

Flags

/N=*name*	Replaces the text of the named tag or textbox.
/W=*winName*	Replaces text in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

See Also

See the **Tag** and **TextBox** operations for details about the *textStr* parameter.

The **ColorScale** operation.

ReplaceWave

```
ReplaceWave [/W=winName] allinCDF
ReplaceWave [/X/W=winName] trace=traceName, waveName
ReplaceWave [/X/Y/W=winName] image=imageName, waveName
ReplaceWave [/X/Y/W=winName] contour=contourName, waveName
```

The ReplaceWave operation replaces waves displayed in a graph with other waves. The waves to be replaced, and the replacement waves are chosen by the flags, the keyword and the wave names on the command line.

Flags

/W=*winName* Replaces the wave in the named graph window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/X Replaces the wave defining the X data spacing.

/Y Replaces the wave defining the Y data spacing.

Keywords

allinCDF Searches the current data folder for waves with the same names as waves used in the graph. If found and if the waves are of the correct type, they replace the existing waves. Thus, if you have several data folders with identically-named waves containing data from different experimental runs, you can browse through the runs by moving from one data folder to another, using `ReplaceWave allinCDF` to update the graph.

contour=*contourName* Replaces the wave supplying the Z data for *contourName*. If /X or /Y is used, replaces the wave used to set the X or Y data spacing (if the Z data are in a matrix) or the wave used to supply the X or Y positions if XYZ triplets were specified with three separate waves.

image=*imageName* Replaces the wave supplying the Z data for *imageName*. If /X or /Y is used, replaces the wave used to set the X or Y data spacing.

trace=*traceName* Replaces the wave associated with *traceName*. With the /X flag, *waveName* will replace the X wave associated with *traceName*, otherwise it will replace the Y wave. Note that *traceName* is derived from the Y wave name; if you created a graph using `Display jack vs sam`, you would use `ReplaceWave/X trace=jack,newsam` to replace the X wave.

Details

Waves are replaced in the graph specified by /W=*winName* otherwise waves are replaced in the top graph.

Updating a contour plot in response to replacing a wave can be time-consuming. If you must replace more than one wave, put all the commands separated by semicolons on a single line. In a macro, use **DelayUpdate** to prevent updates between command lines.

When using the allinCDF keyword, ReplaceWave cannot find waves buried in dynamic annotation text (for instance, using the \{} syntax in an annotation). ReplaceWave will not replace waves used for error bars, either.

Subsets of data, including individual rows or columns from a matrix, may be specified using **Subrange Display Syntax** on page II-288.

Examples

Make XY plot, then replace the waves:

```
Make fred=x, sam=log(x)
Display fred vs sam
Make fred2=2*x, sam2=ln(x)
ReplaceWave/X trace=fred, sam2
ReplaceWave trace=fred, fred2           // trace is now named fred2
```

Make contour plot with XYZ triplet waves, then replace the waves. Note the DelayUpdate commands after the first two ReplaceWave commands:

```
Make/N=100 junkx, junky, junkz          // Waves for XYZ triplets
junkx=trunc(x/10)                       // X wave for XYZ triplets
junky=mod(x,10)                         // Y wave for XYZ triplets
junkz=sin(junkx[p])*cos(junky[p])       // Z wave for XYZ triplets
Display; AppendXYZContour junkz vs {junkx, junky}  // Make contour plot
Make/O/N=150 junkx2, junky2, junkz2     // Make replacement waves
junkx2=trunc(x/15)
junky2=mod(x,15)
junkz2=sin(junkx2[p])*cos(junky2[p])
ReplaceWave/X contour=junkz,junkx2; DelayUpdate
ReplaceWave/Y contour=junkz,junky2; DelayUpdate
ReplaceWave contour=junkz,junkz2
```

This example is suitable for copying all the lines and pasting into the command line, or for use in a macro. If you are typing on the command line, you would want to put the ReplaceWave commands all on one line:

```
ReplaceWave/X contour=junkz,junkx2; ReplaceWave/Y contour=…
```

Resample

Resample [***flags***] ***waveName*** [, ***waveName***]…

The Resample operation resamples *waveName* by interpolating or up-sampling (set by /UP=*upSample*), lowpass filtering, and decimating or down-sampling (set by /DOWN=*downSample*).

Lowpass filtering is specified with /N and /WINF or with /COEF=*coefsWaveName*.

The sampling frequency (1/DimDelta) of a resampled output wave *waveName* is changed by the ratio of *downSample/upSample*. For example, if *upSample*=4 and *downSample*=3, then the final sampling rate is 3/4 of the original value.

Straight interpolation can be accomplished by setting *upSample* to the interpolation factor and *downSample*=1, in which case the sample rate is multiplied by *upSample*. Deltax(*waveName*) will be proportionally smaller.

For decimation only, set *upSample*=1 and *downSample* to the decimation factor. The sample rate is divided by *downSample*, and deltax(*waveName*) will be proportionally larger.

Use **RatioFromNumber** to choose appropriate values for *upSample* and *downSample*, or use /SAME=*sWaveName* or /RATE=*sampRate*. See **Resampling Rates Example** for details.

When using /COEF=*coefsWaveName*, the filter coefficients should implement a low-pass filter appropriate for the *upSample* and *downSample* values or aliasing (filtering errors) will result. See **Advanced Externally-Supplied Low Pass Filter Example** for details.

Resampling Rates Flags

The *upSample* and *downSample* values define how much interpolation and decimation to perform. They can be set directly with /UP and /DOWN or indirectly with /SAME or /RATE

/DOWN=*downSample*

Down-samples or decimates the filtered result by this integer factor after up-sampling and lowpass filtering. The default is 1 (no down-sampling).

For example, /DOWN=3 places only every third value in the output wave.

Down-sampling divides the sampling rate of the filtered data by a factor of *downSample*. The DimDelta(*waveName*, *dim*) value is multiplied by the same factor.

/RATE=*sampRate* Converts the output *waveName* to the specified sampling rate frequency (normally Hz).

The necessary *upSample* and *downSample* values for each *waveName* are computed internally as if you had executed:

```
RatioFromNumber (deltax(waveName)*sampRate)
upSample = V_numerator
downSample = V_denominator
```

/RATE returns V_numerator and V_denominator set to these automatically-determined values for the last *waveName*.

/SAME=*sWaveName*

Converts the output *waveName* to the same sampling rate as *sWaveName*, 1/DimDelta(*sWaveName*, *dim*). The necessary *upSample* and *downSample* values are computed internally as if you had executed:

```
Variable dd = DimDelta(waveName,dim)
RatioFromNumber dd/DimDelta(sWaveName,dim)
upSample = V_numerator
downSample = V_denominator
```

/SAME returns V_numerator and V_denominator set to these automatically determined values for the last *waveName*.

/UP=*upSample* Up-samples or interpolates the input by this integer factor. The default is 1 (no up-sampling).

For example, /UP=4 inserts three extra points between each input point (producing 4 times as many values) before the lowpass filtering and down-sampling occurs.

Up-sampling multiplies the sampling rate of the input data by a factor of *upSample*, though no additional signal information is created. The DimDelta(*waveName*, *dim*) value is divided by the same factor.

Internal Sinc Reconstruction Filter Flags

If /COEF=*coefsWaveName* is not specified, Resample computes a windowed sinc filter from /N, /DOWN, /UP, and /WINF flag values.

If /COEF=*coefsWaveName* is specified, then *coefsWaveName* supplies the filter, and /N and /WINF are ignored. See **Externally-Supplied Low Pass Filter Flags**.

/COEF Replaces the first *waveName* with coefficients generated by *downSample*, *upSample*, *numReconstructionSamples*, and *windowKindName*, a windowed sinc impulse response.

When resampling multiple *waveName*s with different filters (because /RATE or /SAME were specified and the multiple *waveName*s had different sampling rates), the filter used to resample the last *waveName* is returned.

/N=*numReconstructionSamples*

Specifies the number of input values used to created the up-sampled values (default is 21).

The value of *numReconstructionSamples* must be odd.

The size of the computed filter is (*numReconstructionSamples*-1) * *upSample* + 1.

Bigger is better: 15 is usually on the low side for yielding reasonably accurate results, and although 101 will nearly always give very good results, it will be slow.

Use /COEF to output the impulse response, and the FFT to display the frequency response of the interpolator:

```
Make/O coefs
Variable numReconstructionSamples= 51, upSample= 5
Resample/COEF/N=(numReconstructionSamples)/UP=(upSample) coefs
Variable evenNum= 2*floor((numpnts(coefs)+1)/2)
FFT/OUT=3/PAD={evenNum}/DEST=coefs_FFT coefs
Display coefs_FFT
```

Bigger is also slower: the filtering is computed in the time-domain, and execution time is linearly related to

upSample/downSample * *numReconstructionSamples*.

/WINF=*windowKindName*

Applies the window, *windowKindName*, to the computed filter coefficients. If /WINF is omitted, the Hanning window is used. For no coefficient windowing, use /WINF=None, though this is discouraged.

Windows alter the frequency response of the filter in obvious and subtle ways, enhancing the stop-band rejection or steepening the transition region between passed and rejected frequencies. They matter less when *numReconstructionSamples* is large.

Choices for *windowKindName* are (see the **FFT** /WINF flag for details):

Bartlett, Blackman367, Blackman361, Blackman492, Blackman474, Cos1, Cos2, Cos3, Cos4, Hamming, Hanning, KaiserBessel20, KaiserBessel25, KaiserBessel30, Parzen, Poisson2, Poisson3, Poisson4, and Riemann.

Externally-Supplied Low Pass Filter Flags

/COEF =*coefsWaveName*

Identifies the wave, *coefsWaveName*, containing filter coefficients that implement a low-

pass filter with a cutoff frequency of the lesser of 0.5/*upSample* and 0.5/*downSample*, where 0.5 corresponds to the Nyquist frequency of the up-sampled data.

For example, if *upSample*=2, then the filter must contain the classic "half-band" filter, which stops the higher half of the frequencies and passes the lower half. If *upSample*=10, then the filter must pass only the lowest 1/10th of the frequencies.

For *downSample* > *upSample*, the low-pass filter's cutoff frequency must be 0.5/*downSample*. This prevents the decimation from introducing aliasing to the resampled data.

To avoid shifting the output with respect to the input, *coefsWaveName* must have an odd length with the "center" coefficient in the middle of the wave.

The length of *coefsWaveName* must be 1+*upSample***n*, where *n* is any even integer.

Note: Instead of using /N=*numReconstructionSamples* with /COEF=*coefsWaveName*, *numReconstructionSamples* is computed from *upSample* and the number of points in *coefsWaveName*:

numReconstructionSamples=1+(numpnts(*coefsWaveName*)-1)/*upSample*.

Coefficients are usually symmetrical about the middle point, but this is not enforced.

coefsWaveName must not be a destination *waveName*.

coefsWaveName must be single- or double-precision numeric and one-dimensional.

Data Range Flags

/DIM=*d* Specifies the wave dimension to resample.

For *d* =0, 1, …, resampling is along rows, columns, etc.

The default is /DIM=0, which resamples each individual column (each one a channel, say left and right) in a multidimensional *waveName* where each row comprises all sound samples at a particular time.

To resample in multiple dimensions, execute the command once for each dimension. For example, use /DIM=0 followed by another command with /DIM=1 to resample a two-dimensional wave in each direction.

/E=*endEffect* Determines how to handle the ends of the resampled wave(s) (w) when fabricating missing neighbor values.

endEffect=0: Bounce method. Uses w[*i*] in place of the missing w[-*i*] and w[*n*-*i*] in place of the missing w[*n*+*i*].

endEffect=1: Wrap method. Uses w[*n*-*i*] in place of the missing w[-*i*] and vice versa.

endEffect=2: Zero method (default). Uses 0 for any missing value.

endEffect=3: Repeat method. Uses w[0] in place of the missing w[-*i*] and w[*n*] in place of the missing w[*n*+*i*].

Parameters

waveName can be a wave with any number of dimensions. Only one dimension is resampled. Use multiple Resample calls to resample across multiple dimensions.

Without /DIM, resampling is done along the row (first) dimension. Each column is resampled as if it were a separate one-dimensional row. This allows multichannel audio to be resampled to another frequency.

If /DIM=1, then resampling proceeds across all the columns of each row.

If /COEF is specified without *coefsWaveName*, then the first *waveName* is overwritten by the filter coefficients instead of being resampled.

Details

The filtering convolution is performed in the time-domain. That is, the FFT is not employed to filter the data. For this reason the coefficients length (/N or the length of *coefsWaveName*) should be small in comparison to the resampled waves.

Resample assumes that the middle point of *coefsWaveName* corresponds to the delay=0 point. The "middle" point number = trunc(numpnts(*coefsWaveName* -1)/2). *coefsWaveName* usually contains the two-sided impulse response of a filter, an odd number of points, and implements a low-pass filter whose cutoff frequency is the lesser of 0.5/*upSample* and 0.5/*downSample* (0.5 corresponds to the Nyquist frequency = 1/2 sampling frequency).

When /COEF creates a coefficients wave it sets the X scale deltax to 1 and alters the leftx value so that the zero-phase (center) coefficient is located at x=0.

Simple Examples

Interpolation by factor of 4, default filter:

```
Resample/UP=4 data
```

Decimation by factor of 3, default filter:

```
Resample/DOWN=3 data
```

Match sampling rates, default filter:

```
Resample/SAME=dataAtDesiredRate dataAtWrongRate1, dataAtWrongRate2,...
```

Resample waves to 10 KHz sampling rate:

```
Resample/RATE=10e3 dataAtWrongRate1, dataAtWrongRate2,...
```

Interpolate an image by a factor of 2:

```
Resample/UP=2 image              // default is /DIM=0, resample rows
Resample/UP=2/DIM=1 image        // resample across columns
```

Note: Interpolating by a factor of two does not produce an image with twice as many rows and columns. The new number of rows = (original rows-1)*_upSample_ +1, and a similar computation applies to columns.

Resampling Rates Example

Suppose we have an audio wave sampled at 44,100 Hz and we wish to resample it to a higher 192,000 Hz frequency.

We can use /RATE= 192000 and let Resample determine the correct values (provided *waveName* has its X scaling set properly to reflect sampling at 44100 Hz), but let's compute *upSample* and *downSample* ourselves.

Because the sampling rate = 1/deltax(*wave*), we can recast the /SAME formula to RatioFromNumber (*desiredSamplingRate*/*currentSamplingRate*):

```
•RatioFromNumber/V (192000 / 44100)
  V_numerator= 640; V_denominator= 147;
  ratio= 4.3537414965986;
  V_difference= 0;
```

Then *upSample*=640 and *downSample*=147.

The 44100 Hz input data will be interpolated by 640 to 28,224,000 Hz.

The result is low-pass filtered with a "cutoff frequency" of 1/640th of the interpolated Nyquist frequency = (28224000/2)/640 = 22,050 Hz, the same as the input signal's original Nyquist frequency.

The result will be decimated by 147 to 192,000 Hz, which is the desired output sampling frequency.

Note: If *downSample* had been greater than *upSample*, then the low-pass filter's cutoff frequency would have been 1/*downSample*th of the interpolated Nyquist frequency = (28224000/2)/*downSample*. This prevents the decimation from introducing aliasing to the resampled data.

```
Resample/UP=640/DOWN=147 sound          // convert 44.1 KHz to 192 KHz
```

Advanced Externally-Supplied Low Pass Filter Example

You can generate an appropriate filter by executing commands like these:

```
// Compute a filter for after the input is upsampled
// to restore the frequency content to the original range.
Variable fc = min(0.5 / upSample, 0.5 * upSample / downSample)
// Transition width, small widths need big n
Variable tw= fc/10
// Set end of pass band
Variable f1= fc-tw/2
// Set start of stop band
Variable f2= fc+tw/2
// Use bigger values of n to make the filter smoother
Variable nReconstruct= 31
Variable n= (nReconstruct-1)*upSample+1       // odd = no phase shift
// Create a wave to hold the coefficients; it gets resized to n
Make/O/N=0 coefsWaveName
FilterFIR/COEF/LO={f1,f2,n} coefsWaveName
```

However, FilterFIR does not create windowed sinc lowpass filters that have the endearing property that the original input values are unaltered in the filtered output, though only if *upSample* > *downSample*. This is called a "Nyquist filter" or "Kth-band filter" in the literature.

If *upSample* > *downSample*, you can enforce the Nyquist criterion by "zeroizing" the designed filter by setting every *upSample*th value to 0 except the center one.

```
// coefsWaveName length must be 1+upSample*n, where n is any even integer
Function Zeroize(w, upSample)
    Wave w                          // coefsWaveName
    Variable upSample               // upSample value

    Variable n= DimSize(w,0)
    Variable centerP= floor((n-1)/2)        // if n=101, centerP= 50
    Variable i
    for (i=0; i<n; i+=upSample)
        if( i != centerP )
            w[i] = 0
        endif
    endfor
End
```

Resample zeroizes the internally-generated low pass filter when *upSample* > *downSample*.

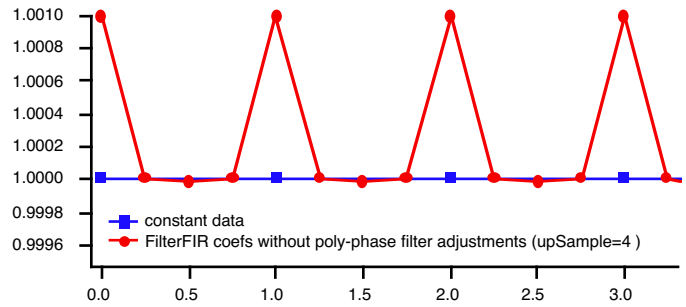
Additionally, the FilterFIR command generates a low-pass filter whose gain needs to be multiplied by *upsample*:

coefsWaveName *= *upSample*

When designing an externally supplied filter, you should also consider the filter's "polyphase" nature; *coefsWaveName* is actually a set of *upSample* interleaved filters, each with its own response. It makes sense to adjust these filters to produce consistent responses. If you don't, the results will contain ringing with a period of *upSample*/*downSample*. This is most apparent when *downSample* is 1.

Using the filter we've designed so far with *upSample*=4, here's the output of a constant-input wave:

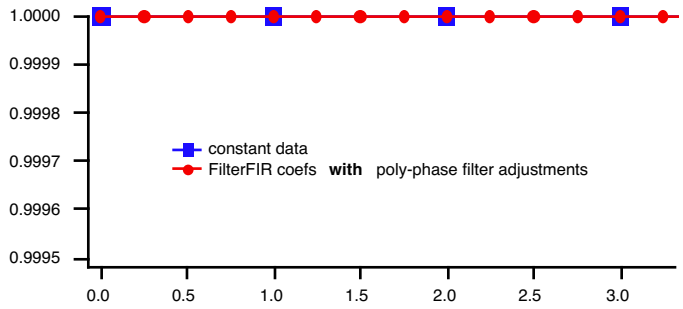
```
Make/O constantData= 1
Resample/COEF=coefsWaveName/UP=4 constantData
```



The graph shows that the filter response at 0 Hz for the first of 4 filters is 1.0010, the second and fourth filter's responses are very close to 1.0, and the third filter's response at 0 Hz is a little less than 1.0.

These variations can be eliminated by normalizing the sum of each polyphase filter to 1.0:

```
Function PolyphaseNormalize(w, upSample)
    Wave w                              // coefsWaveName
    Variable upSample                   // upSample value

    Variable n= DimSize(w,0)
    Variable filt
    // for each filter (0..upSample-1)
    for (filt=0; filt<upSample; filt+=1)
        Variable total=0
        Variable pt
        // compute total for this filter
        for (pt=filt; pt<n; pt+=upSample)
            total += w[pt]
        endfor
        // divide by total to normalize total to 1
        for (pt=filt; pt<n; pt+=upSample)
            w[pt] /= total
        endfor
    endfor
End
```



Now the filter is ready to be used to filter data:

```
Resample/COEF=coefsWaveName/UP=(upSample)/DOWN=(downSample) dataWave
```

You can see that designing an externally-supplied lowpass filter is much more complicated than using the internal sinc reconstruction filter, which does all this zeroizing, scaling, and polyphase normalization for you.

References

Mintzer, F., On half-band, third-band, and Nth band FIR filters and their design, *IEEE Trans. on Acoust., Speech, Signal Process., ASSP-30*, 734-738, 1982.

See Also

The **RatioFromNumber**, **FilterFIR**, **interp**, **Interp2D**, **ImageInterpolate**, and **Loess** operations; and the Interpolate XOP.

ResumeUpdate

ResumeUpdate

The ResumeUpdate operation cancels the corresponding **PauseUpdate**.

This operation is of use in macros. It is not allowed from the command line. It is allowed but has no effect in user-defined functions. During execution of a user-defined function, windows update only when you explicitly call the DoUpdate operation.

See Also

The **DelayUpdate**, **DoUpdate**, and **PauseUpdate** operations.

return

return [*expression*]

The return flow control keyword immediately stops execution of the current procedure. If called by another procedure, it returns *expression* and control to the calling procedure.

Functions can return only a single value directly to the calling procedure with a return statement. The return value must be compatible with the function type. A function may contain any number of return statements; only the first one encountered during procedure execution is evaluated.

A macro has no return value, so return simply quits the macro.

See Also

The Return Statement on page IV-31.

Reverse

Reverse [*type flags*] [**/DIM=***d* **/P**] *waveA* [**/D = *destWaveA***] [, *waveB* [**/D = *destWaveB***] [, …]]

The Reverse operation reverses data in a wave in a specified dimension. Reverse does not accept text waves.

Flags

/DIM = *d*	Specifies the wave dimension to reverse. *d*=-1: Treats entire wave as 1D (default). For *d*=0, 1, …, operates along rows, columns, etc.
/P	Suppresses adjustment of dimension scaling. Without /P the scaled dimension value of reversed points remains the same.

Type Flags *(used only in functions)*

Reverse also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when it is needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-58 and **WAVE Reference Type Flags** on page IV-58 for a complete list of type flags and further details.

Wave Parameters

Note: ***All*** wave parameters must follow *wave* in the command. All wave parameter flags and type flags must appear immediately after the operation name (Reverse).

/D=*destWave* Specifies the name of the wave to hold the reversed data. It creates *destWave* if it does not already exist or overwrites it if it exists.

Details

If the optional /D = *destWave* flag is omitted, then the wave is reversed in place.

See Also

The **Sort** operation and **Sorting** on page III-134.

RGBColor

The RGBColor structure is used as a substructure usually to store various color settings.

```
Structure RGBColor
   UInt16 red
   UInt16 green
   UInt16 blue
EndStructure
```

rightx

rightx(*waveName*)

The rightx function returns the X value corresponding to point N of the named 1D wave of length N.

Details

Note that the point numbers in a wave run from 0 to N-1 so there is no point with this X value. To get the X value of the last point in a wave (point N-1), use the following:

```
pnt2x(waveName,numpnts(waveName)-1)          // N = numpnts(waveName)
```

which is more accurate than:

```
rightx(waveName) - deltax(waveName)
```

The rightx function is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details. The equivalent information for any dimension can be calculated this way:

IndexN = DimSize(*wave*, *dim*)*DimDelta(*wave*, *dim*) + DimOffset(*wave*, *dim*)

Here *IndexN* is the value of the scaled dimension index corresponding to element N of the dimension *dim* in a wave named *wave* that has N elements in that dimension.

See Also

The **deltax** and **leftx** functions, also the **pnt2x** and **numpnts** functions.

For an explanation of waves and dimension scaling, see **Changing Dimension and Data Scaling** on page II-83.

For multidimensional waves, see **DimDelta**, **DimOffset**, and **DimSize**.

root

root[:*dataFolderName*[:*dataFolderName*[:…]]][:*objectName*]

Igor's data folder hierarchy starts with the root folder as its basis. The root data folder always exists and it contains all other objects (waves, variables, strings, and data folders). By default, the root data folder is the current data folder in a new experiment. In commands, root is used as part of a path specifying the location of a data object in the folder hierarchy.

See Also

Chapter II-8, **Data Folders**.

Rotate

Rotate *rotPoints*, *waveName* [, *waveName*]…

The Rotate operation rotates the Y values of waves in wavelist by *rotPoints* points.

Parameters

If *rotPoints* is positive then values are rotated from the start of the wave toward the end and *rotPoints* values from the end of a wave wrap around to the start of the wave.

If *rotPoints* is negative then values are rotated from the end of the wave toward the start and *rotPoints* values from the start of a wave wrap around to the end of the wave.

Details

The X scaling of the named waves is changed so that the X values for the Y values remains the same except for the points that wrap around.

The Rotate operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

The Rotate operation is not multidimensional aware. To rotate rows or columns of 2D waves, see the rotateRows and rotateCols keywords for the **MatrixOp** and **ImageTransform** operations.

For general information about multidimensional analysis, see **Analysis on Multidimensional Waves** on page II-110.

See Also

The shift parameter of the **WaveTransform** operation.

round

round(*num*)

The round function returns the integer value closest to *num*.

The rounding method is “away from zero”.

See Also

The **ceil**, **floor**, and **trunc** functions.

rtGlobals

#pragma rtGlobals = 0, 1, 2, or **3**

`#pragma rtglobals=<n>` is a compiler directive that controls compiler and runtime behaviors for the procedure file in which it appears.

This statement must be flush against the left edge of the procedure file with no indentation. It is usually placed at the top of the file.

`#pragma rtglobals=0` turns off runtime creation of globals. This is obsolete.

`#pragma rtglobals=1` is a directive that turns on runtime lookup of globals. This is the default behavior if #pragma rtGlobals is omitted from a given procedure file.

`#pragma rtGlobals=2` turns off compatibility mode. This is mostly obsolete. See **Legacy Code Issues** on page IV-90 for details.

`#pragma rtglobals=3` turns on runtime lookup of globals, strict wave reference mode and wave index bounds checking.

If your procedures will run only with Igor Pro 6.20 or later, rtGlobals=3 is recommended. Otherwise rtGlobals=1 is recommended.

See **The rtGlobals Pragma** on page IV-41 for a detailed explanation of rtGlobals.

s

s

The s function returns the current chunk index of the destination wave when used in a multidimensional wave assignment statement. The corresponding scaled chunk index is available as the **t** function.

Details

Unlike **p**, outside of a wave assignment statement, s does not act like a normal variable.

See Also

Waveform Arithmetic and Assignments on page II-93.

For other dimensions, the **p**, **r**, and **q** functions.

For scaled dimension indices, the **x**, **y**, **z** and **t** functions.

Save

Save [***flags***] ***waveList*** [**as** ***fileNameStr***]

The Save operation saves the named waves to disk as text (/F, /G or /J) or as Igor binary.

Parameters

waveList is either a list of wave names or, if the /B flag is present, a string list of references to waves. For example, the following commands are equivalent, assuming that the waves in question are in the root data folder and root is the current data folder:

```
Save/J wave0,wave1 as "Test.dat"
Save/J root:wave0,root:wave1 as "Test.dat"
Save/J/B "wave0;wave1;" as "Test.dat"
Save/J/B "root:wave0;root:wave1;" as "Test.dat"
String list="root:wave0;root:wave1;"; Save/J/B list as "Test.dat"
```

The form using the /B flag and a string containing a list of references to waves saves a very large number of waves using one command. This is not possible using a list of wave names because of the 400 character limit in a command. When using this form, the string must contain semicolon-separated wave names or data folder paths leading to waves. Liberal names in the string may be quoted or unquoted.

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If it cannot determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

Flags

/A[=*a*] Appends to the file rather than overwriting it (with /T, /G or /J).
- *a*=0: Does not append.
- *a*=1: Appends to the file with a blank line before the appended data (same as /A only).
- *a*=2: Appends to the file with no blank line before the appended data.

/B The *waveList* parameter is a string containing a list of references to waves instead of a literal list of waves.

/C Saves a copy of the wave when saving as Igor binary.

/DSYM=*dsStr* Specifies a string containing the character to use as the decimal symbol for all numbers (default is a period). If *dsStr* is empty (""), then the decimal symbol is as defined in system preferences.

/E=*useEscapeCodes* Determines whether to use escape sequences for special characters.
- /E=1: Converts carriage-return, linefeed, tab, and backslash characters to escape sequences when writing general or delimited text files (default; same as no /E).
- /E=0: No escape sequences used in general or delimited text files. When saving text waves containing backslashes (such as Windows paths) in a file intended for another program, you probably should use /E=0.

/F — Writes delimited and general text files with numeric formatting as it appears in the top table. Has no effect if there is no top table or if the wave being saved does not appear in the top table.

Note: The text written to the file is exactly as displayed in the table. Set the table to display as many digits of precision as you want in the file.

Note: When saving a multi-column wave (1D complex wave or multi-dimensional wave), all columns of the wave are saved using the table format for the first table column from the wave.

/G — Saves waves in general text format.

/H — "Adopts" the waves specified by *waveList*.

"Adopt" means that any connection between the waves and external files is severed. The waves become part of the current experiment. When the experiment is next saved, the waves are saved in the experiment file (for an packed experiment) or in the experiment folder (for an unpacked experiment).

When you use the /H flag, all other flags and the *fileNameStr* parameter are ignored. The wave is not actually saved but rather is marked for saving as part of the current experiment.

You would normally do this to make an experiment more self-contained which makes it easier to send to other people. See **Sharing Versus Copying Igor Binary Files** on page II-165 and the **LoadWave** /H flag.

/I — Presents a dialog from which you can specify file name and folder.

/J — Saves waves in tab-delimited text format.

/M=*termStr* — Specifies the terminator character or characters to use at the end of each line of text. The default is `/M="\r"`, which uses a carriage return character. This is the Macintosh convention. To use the Windows convention, carriage return plus linefeed, specify `/M="\r\n"`. To use the Unix convention, just a linefeed, specify `/M="\n"`.

/O — Overwrites file if it exists already.

/P=*pathName* — Specifies the folder to store the file in. *pathName* is the name of an existing symbolic path.

/T — Saves waves in Igor Text format.

/U={*writeRowLabels*, *rowPositionAction*, *writeColLabels*, *colPositionAction*}

These parameters affect the saving of a matrix (2D wave) to a delimited text (/J) or general text (/G) file. They are accepted no matter what the save type is but are ignored when they don't apply.

If *writeRowLabels* is nonzero, Save writes the row labels of the matrix as the first column of data in the file.

rowPositionAction has one of the following values:

0: Don't write a row position column.

1: Writes a row position column based on the row scaling of the matrix wave.

2: Writes a row position column based on the contents of the row position wave for the matrix. The row position wave is an optional 1D wave whose name is the same as the matrix wave but with the suffix "_RP".

writeColumnLabels and *columnPositionAction* have analogous meanings. The suffix used for the column position wave is "_CP".

See Chapter II-9, **Importing and Exporting Data**, for further details.

/W — Saves wave names (with /G or /J).

Details

The Save operation saves only the named waves; it does not save the entire experiment.

Waves saved in Igor binary format are saved one wave per file. If you are saving more than one wave, you must not specify a *fileNameStr*. Save will give each file a name which consists of the wave name concatenated with ".ibw".

When you save a wave as Igor binary, unless you use the /C flag to save a copy, the current experiment subsequently references the file to which the wave was saved. See **References to Files and Folders** on page II-37 for details.

In a general text file (/G), waves with different numbers of points are saved in different groups. Waves with different precisions and number types are saved in same group if they have the same number of points.

In a tab-delimited text file (/J), all waves are saved in one group whether or not they have the same number of points.

If you save multiple 2D waves, the blocks of data are written one after the other.

If you save 3D waves, the data for each wave is written as a contiguous block having as many columns as there are columns in the wave, and R*L rows, where R is the number of rows in the multidimensional wave and L is the number of layers. All rows for layer 0 are saved followed by all rows for layer 1, and so on.

If you save 4D waves, the data for each wave is written as a contiguous block having R*L*C rows, where R is the number of rows, L is the number of layers and C is the number of chunks. Igor writes all data for chunk 0 followed by all data for chunk 1, and so on.

The Save operation will always present a save dialog if you try to save to an existing file without using the overwrite flag.

Here are some details about saving an Igor binary file.

If you omit the path or the file name, the Save operation will normally present a save dialog. However, if the wave has already been saved to a stand-alone file and if you use the overwrite flag, it will save the wave to the same file without a dialog. Also, if the wave has never been saved and the current experiment is an unpacked experiment, it will save to the home folder without a dialog.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-34 for details.

Examples

This function uses the string list of references to waves to save some or all of the waves in the current data folder:

```
Function SaveWavesFromCurrentDF(matchStr)
    String matchStr                  // As for the WaveList function.

    String list
    list = WaveList(matchStr, ";", "")
    Save/O/J/W/I/B list
End
```

For example, to save all of the waves in the current data folder, execute:

```
SaveWavesFromCurrentDF("*")
```

To save those waves in the current data folder whose name starts with "wave", execute:

```
SaveWavesFromCurrentDF("wave*")
```

This function saves all of the waves used in a particular graph:

```
Function SaveWavesFromGraph(graphName)      // Saves all waves in graph.
    String graphName                        // "" for top graph.

    String list, traceList, traceName
    Variable index = 0
    list = ""
    traceList = TraceNameList(graphName, ";", 1)
    do
        traceName = StringFromList(index, traceList, ";")
        if (strlen(traceName) == 0)
            break
        endif
        Wave w = TraceNameToWaveRef(graphName, traceName)
        list += GetWavesDataFolder(w,2) + ";"
        index += 1
    while(1)

    if (strlen(list) > 0)
        Save/O/J/W/I/B list
    endif
End
```

See Also

Saving Waves on page II-175.

SaveData

SaveData [*flags*] *fileOrFolderNameStr*

The SaveData operation writes data from the current data folder of the current experiment to a packed experiment file on disk or to a file system folder. "Data" means Igor waves, numeric and string variables, and data folders containing them. The data is written as a packed experiment file or as unpacked Igor binary files in a file-system folder.

Warning: If you make a mistake using SaveData, it is possible to overwrite critical data, even entire folders containing critical data. It is your responsibility to make sure that any file or folder that you can not afford to lose is backed up. If you provide procedures for use by other people, you should warn them as well.

SaveData provides a way to save data for archival storage or unload data from memory during a lengthy process like data acquisition. The file or files that SaveData writes are disassociated from the current experiment.

Use SaveData to save experiment data using Igor procedures. To save experiment data interactively, use the Save Copy button in the Data Browser (Data menu).

Parameters

fileOrFolderNameStr specifies the packed experiment file (if /D is omitted) or the file system folder (if /D is present) in which the data is to be saved. The documentation below refers to this file or folder as the "target".

If you use a full or partial path for *fileOrFolderNameStr*, see **Path Separators** on page III-398 for details on forming the path.

If *fileOrFolderNameStr* is omitted or is empty (""), SaveData displays a dialog from which you can select the target. You also get a dialog if the target is not fully specified by *fileOrFolderNameStr* or the /P=*pathName* flag.

Flags

/D [=*d*] Writes to a file-system folder (a directory). If omitted, SaveData writes to an Igor packed experiment file.

d=1 If the target folder already exists, the new data is "mixed-in" with the data already there (same as /D).

d=2 If the target folder already exists, **it is completely deleted before the writing of data starts**.

If in doubt, use /D=1. See **Details** below.

/I Presents a dialog in which you can interactively choose the target.

/J=*objectNamesStr* Saves only the objects named in the semicolon-separated list of object names. See **Details** below.

/L=*saveFlags* Controls what kind of data objects are saved with a bit for each data type:

saveFlags	Bit Number	Saves this Type of Object
1	0	Waves
2	1	Numeric variables
4	2	String variables

To save multiple data types, sum the values shown in the *saveFlags* column. For example, /L=1 saves waves only, /L=2 saves numeric variables only and /L=3 saves both waves and numeric variables.

If /L is not specified, all of these object types are saved. This is equivalent to /L=7. All other bits are reserved and must be set to zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/M=*modDateTime* Saves waves modified on or after the specified modification date/time. Waves modified before *modDateTime* will not be saved. Applies to waves only (not variables or strings).

modDateTime is in standard Igor time format — seconds since 1/1/1904. If *modDateTime* is zero, all waves will be saved, as if there were no /M flag at all.

/O Overwrites existing files or folders on disk.

Warning: If you use the /O flag and if the target already exists, it will be overwritten without any warning. If you use /O with /D=2, **you will completely overwrite the target folder and all of its contents, including subfolders**. Do not use /O with /D unless you are absolutely sure you know what your doing.

/P=*pathName* Specifies the folder in which to save the specified file or folder.

pathName is the name of an Igor symbolic path, created via NewPath. It is not a file system path like "hd:Folder1:" or "C:\\Folder1\". See **Symbolic Paths** on page II-34 for details.

When used with the /D flag, if /P=*pathName* is present and *fileOrFolderNameStr* is ":", the target is the directory specified by /P=*pathName*.

/Q Suppresses normal messages in the history area.

/R Recursively saves subdata folders.

/T [=*topLevelName*] Creates an enclosing data folder in the target with the specified name, *topLevelName*, and writes the data to the new data folder.

If just /T is specified, it creates an enclosing data folder in the target using the name of the data folder being saved. However, if the data folder being saved is the root data folder, the name Data is used instead of root. In packed experiment files and unpacked experiment folders, the root data folder is implicit.

If /T is omitted, the contents of the current data folder are saved with no enclosing data folder.

Details

If /J=*objectNamesStr* is used, then only the objects named in *objectNamesStr* are saved. For example, specifying `/J="wave0;wave1;"` will save only the two named waves, ignoring any other data in all data folders.

The list of object names used with /J must be semicolon-separated. A semicolon after the last object name in the list is optional. The object names must not be quoted even if they are liberal. The list is limited to 1000 characters.

Using `/J=""` acts like no /J at all.

The /M=*modDateTime* flag can be used in data acquisition projects to save only those waves modified since the previous save. For example, assume that we have a global variable in the root data folder named gLastWaveSaveDateTime. Then this function will write out only those waves modified since the previous save:

```
Function SaveModifiedWaves(savePath)
    String savePath     // Symbolic path pointing to output directory
    NVAR lastSave = root:gLastWaveSaveDateTime
    SaveData/O/P=$savePath/D=1/L=1/M=(lastSave) ":"
    lastSave = datetime
End
```

Because the datetime function and the wave modification date have a coarse resolution (one second), this function may sometimes save the same wave twice.

The /M flag makes sense only in conjunction with the /D=1 flag because /D=1 is the only way to mix-in new data with existing data.

Writing to a Packed Experiment File

When writing to a packed file, SaveData creates a standard packed Igor experiment file which you can open as an experiment, browse using the Data Browser, or access using the **LoadData** operation.

If you do not use the /O (overwrite) flag and the packed file already exists on disk, SaveData will present a dialog to confirm which file you want to write to. If you use the /O flag, SaveData will overwrite without presenting a dialog. When writing a packed file, SaveData always completely overwrites the preexisting packed file.

Appending to a packed experiment file is not supported because dealing with the possibility of name conflicts (e.g., two waves with the same name in the same data folder in the packed experiment file) would be technically difficult, very slow and errors would result in corrupted files.

Writing to a File-System Folder

When saving to a folder on disk, SaveData writes wave files, variables files, and subfolders. This resembles the experiment folder of an unpacked experiment, but it does not contain other unpacked experiment files, such as history or procedures. You can browse the folder using the Data Browser or access it using the **LoadData** operation.

If the target directory does not exist, SaveData creates it.

If you do not use the /O (overwrite) flag and the target folder already exists on disk, SaveData will present a dialog to confirm that you want to write to it. SaveData checks for the existence of the top file system folder only. For example, if you write data to hd:Data:Run1, SaveData will display a dialog if hd:Data:Run1 exists. But SaveData will not display a dialog for any folders inside hd:Data:Run1.

If you use the /O flag, SaveData will write without presenting a dialog.

When writing to a directory, SaveData can operate in one of two modes. If you use /D=1 or just /D, SaveData operates in "mix-in" mode. If you use /D=2, SaveData operates in "delete" mode.

Warning: If the target directory exists and delete mode is used, SaveData deletes the target directory and all of its contents. Then SaveData creates the target directory and writes the data to it. This is a complete overwrite operation.

If the target directory exists and mix-in mode is used, SaveData does not do any explicit deletion. It writes data to the target directory and any subdirectories. Conflicting files in any directory are overwritten but other files are left intact.

To prevent you from inadvertently deleting an entire volume, SaveData will not permit you to target the root directory of any volume. You must target a subdirectory.

The /J flag will not work as expected when writing numeric and string variables in mix-in mode. Instead of mixing-in the specified variables, SaveData will overwrite all variables already in the target. This is because all numeric and string variables in a particular data folder are stored in a single file-system folder (named "variables"), so it is not possible to mix-in. Since waves are written one-to-a-file, /J will work as expected for waves.

When SaveData writes a wave to a file-system folder, the file name for the wave is the same as the wave name, with the extension ".ibw" added. This is true even if the wave in the experiment was loaded from a file with a different name.

Outputs

SaveData sets the variable V_flag to zero if the operation succeeded or to nonzero if it failed. The main use for this is to determine if the user clicked Cancel during an interactive save. This would occur if you use the /I flag or if you omit /O and the target already exists. V_flag will also be nonzero if an error occurs during the save.

SaveData sets the string variable S_path to the full file system path to the file or folder that was written. S_path uses Macintosh path syntax (e.g., "hd:FolderA:FolderB:"), even on Windows. When saving unpacked, S_path includes a trailing colon.

Examples

Write the contents of the current data folder and all subdata folders to a packed experiment file:

```
Function SaveDataInPackedFile(pathName, fileName)
    String pathName                // Name of symbolic path
    String fileName                // Name of packed file to be written

    SaveData/R/P=$pathName fileName
End
```

Write the contents of the current data folder and all subdata folders to an unpacked file-system folder:

```
Function SaveDataInUnpackedFolder(pathName, folderName)
    String pathName                // Name of symbolic path
    String folderName              // Name of file-system folder

    SaveData/D=1/R/P=$pathName folderName
End
```

Copy the contents of an unpacked file-system folder to a packed experiment file:

```
Function TransferUnpackedToPacked(path1, folderName, path2, fileName)
    String path1            // Points to parent of unpacked folder
    String folderName       // Name of folder containing unpacked data
    String path2            // Points to folder where file is to be written
    String fileName         // Name of packed file to be written

    String savedDF = GetDataFolder(1)

    NewDataFolder/O/S :TempTransfer

    // Load all data from the unpacked folder.
    LoadData/D/Q/R/P=$path1 folderName

    // Save all data to the packed file.
    SaveData/R/P=$path2 fileName
```

```
    KillDataFolder :              // Kill TempTransfer
    SetDataFolder savedDF
End
```

See Also

The **LoadData** and **SaveGraphCopy** operations; the **SpecialDirPath** function. **Saving Package Preferences** on page IV-226; Chapter II-9, **Importing and Exporting Data**; **Data Browser** on page II-130.

SaveExperiment

SaveExperiment [**/C/P=*pathName***] [**as *fileName***]

The SaveExperiment operation saves the current experiment.

Parameters

The optional *fileName* string contains the name of the experiment to be saved. *fileName* can be the currently open experiment, in which case it overwrites the experiment file.

If *fileName* and *pathName* are omitted and the experiment is Untitled, you will need to locate where the experiment file will be saved interactively via a dialog.

If you use a full or partial path for *pathName*, see **Path Separators** on page III-398 for details on forming the path.

Flags

/C	Saves an experiment copy (valid only when *fileName* or *pathName* is provided or both if experiment is Untitled).
/P=*pathName*	Specifies folder in which to save the experiment. *pathName* is the name of an existing symbolic path.

Details

SaveExperiment acts like the Save menu command in the File menu. If the experiment is associated with an already saved file, then SaveExperiment with no parameters will simply save the current experiment. If the experiment resides only in memory and has not yet been saved, then a dialog will be presented unless the path and file name are specified.

If you use a full path in the name you will not need the /P flag. If instead you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-34 for details.

SaveGraphCopy

SaveGraphCopy [***flags***][**as *fileNameStr***]

The SaveGraphCopy operation saves a graph and its waves in an Igor packed experiment file.

Parameters

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

Flags

/I	Presents a dialog from which you can specify file name and folder.
/O	Overwrites file if it exists already.
/P=*pathName*	Specifies the folder to store the file in. *pathName* is the name of an existing symbolic path.
/W= *winName*	*winName* is the name of the graph to be saved. If /W is omitted or if *winName* is `""`, the top graph is saved.
/Z	Errors are not fatal and error dialogs are suppressed. See Details.

Details

The main uses for saving as a packed experiment are to save an archival copy of data or to prepare to merge data from multiple experiments (see **Merging Experiments** on page II-32). The resulting experiment file preserves

the data folder hierarchy of the waves displayed in the graph starting from the "top" data folder, which is the data folder that encloses all waves displayed in the graph. The top data folder becomes the root data folder of the resulting experiment file. Only the graph, its waves, dashed line settings, and any pictures used in the graph are saved in the packed experiment file, not procedures, variables, strings or any other objects in the experiment.

SaveGraphCopy does not work well with graphs containing controls. First, the controls may depend on waves, variables or FIFOs (for chart controls) that SaveGraphCopy will not save. Second, controls typically rely on procedures which are not saved by SaveGraphCopy.

SaveGraphCopy does not know about dependencies. If a graph contains a wave, wave0, that is dependent on another wave, wave1 which is not in the graph, SaveGraphCopy will save wave0 but not wave1. When the saved experiment is open, there will be a broken dependency.

SaveGraphCopy sets the variable V_flag to 0 if the operation completes normally, to -1 if the user cancels, or to another nonzero value that indicates that an error occurred. If you want to detect the user canceling an interactive save, use the /Z flag and check V_flag after calling SaveGraphCopy.

The **SaveData** operation also has the ability to save data from a graph to a packed experiment file. SaveData is more complex but a bit more flexible than SaveGraphCopy.

Examples

This function saves all graphs in the experiment to individual packed experiment files.

```
Function SaveAllGraphsToPackedFiles(pathName)
    String pathName         // Name of an Igor symbolic path.

    String graphName
    Variable index

    index = 0
    do
        graphName = WinName(index, 1)
        if (strlen(graphName) == 0)
            break
        endif

        String fileName
        sprintf fileName, "%s.pxp", graphName

        SaveGraphCopy/P=$pathName/W=$graphName as fileName

        index += 1
    while(1)
End
```

See Also

SaveTableCopy and **SaveData** operations; **Merging Experiments** on page II-32.

SaveNotebook

SaveNotebook [***flags***] ***notebookName*** [**as** ***fileNameStr***]

The SaveNotebook operation saves the named notebook.

Parameters

notebookName is either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-95 for details on host-child specifications.

If *notebookName* is an host-child specification, /S must be used and *saveType* must be 3 or higher.

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

Flags

/H={*encodingName*, *writeParagraphProperties*, *writeCharacterProperties*, *PNGOrJPEG*, *quality*, *bitDepth*}
Controls the creation of an HTML file.

encodingName
Specifies the HTML file encoding. The recommended value for most purposes is "UTF-8". See **Details** for more information.

writeParagraphProperties
Determines what paragraph properties SaveNotebook will write to the HTML file. This is a bitwise parameter with the bits defined as follows:

Bit 0: Write paragraph alignment.

Bit 1: Write first indent.

Bit 2: Write minimum line spacing.

Bit 3: Write space-before and space-after paragraph.

All other bits are reserved for future use and should be set to zero.

writeCharacterProperties
Determines what character properties SaveNotebook will write to the HTML file. This is a bitwise parameter with the bits defined as follows:

Bit 0: Write font families.

Bit 1: Write font sizes.

Bit 2: Write font styles.

Bit 3: Write text colors.

Bit 4: Write text vertical offsets.

All other bits are reserved for future use and should be set to zero.

If you set bit 2, SaveNotebook exports only the bold, underline, and italic styles because other character styles are not supported by HTML.

PNGOrJPEG
Determines whether SaveNotebook will write picture files as PNG or JPEG:

0: PNG (default).

1: JPEG.

2: JPEG.

Prior to Igor Pro 5, *PNGOrJPEG*=2 wrote as JPEG if QuickTime JPEG support was available or as PNG otherwise. As of Igor Pro 5, which has built-in JPEG support, both *PNGOrJPEG*=2 and *PNGOrJPEG*=1 write as JPEG. For both codes, QuickTime is used if it is available because it supports the quality and depth parameters although built-in JPEG support currently always uses *quality*=0.9.

See **Details** for more on HTML picture files.

The next two parameters, *quality*, and *bitDepth*, are used only when writing JPEG files using QuickTime.

quality Specifies the degree of compression or image quality when writing pictures as JPEG files. It is a value from 1.0 (default) to 0.0, with 1.0 giving the greatest quality but largest file size and 0.0 giving the least quality but smallest file size. Use the highest quality that gives an acceptable file size. Often this will be 1.0.

bitDepth Specifies the color depth when writing pictures as JPEG files. It specifies the desired bit depth of the picture. Only the following values are legal: 1, 8 (default), 16, 24, 32. For most Web use, 8 is fine. As of QuickTime 4, QuickTime ignores this parameter.

/I Saves interactively. A dialog is displayed.

/M=*messageStr* Specifies prompt message used in save dialog.

/O Overwrites existing file without asking permission.

/P=*pathName* Specifies the folder to store the file in. *pathName* is the name of an existing symbolic path.

/S=*saveType*
1: Normal save (default).

2: Save-as.

3: Save-a-copy.

4: Export as RTF (Rich Text Format).

5: Export as HTML (Hypertext Markup Language).

6: Export as plain text.
7: Export as formatted notebook.

Details

Interactive (/I) means that Igor displays the Save, Save As, or Save a Copy dialog.

The save will be interactive under the following conditions:

- You include the /I flag and the *saveType* is 2, 3, 4, 5, 6, or 7.
- *saveType* is 2, 3, 4, 5, 6, or 7 and you do not specify the path or filename.

If the *saveType* is normal and the notebook has previously been saved to a file then the /I flag, the path and file name that you specify, if any, are ignored and the notebook is saved to its associated file without user intervention.

The full path to the saved file is stored in the string S_path. If the save was unsuccessful, S_path will be `""`.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-34 for details.

For background information on writing RTF files, see **Import and Export Via Rich Text Format Files** on page III-24.

For background information on writing HTML files, see **Exporting a Notebook as HTML** on page III-26.

For background information regarding the HTML *encodingName* parameter, see **HTML Character Encoding** on page III-29. This table lists how Igor responds to various values for this parameter:

"UTF-8"	Encodes the document using the UTF-8 encoding which is a kind of packed Unicode. It is good for documents that contain non-Roman text, such as symbols and Asian text. This is the best choice for most documents.
"UTF-2"	Encodes the document using the UTF-2 encoding which is 16-bit Unicode. It is generally not recommended because some browsers don't support it. However, as time goes by, browser support will improve.
"Native"	Writes the document using the native character set of the operating system. On Macintosh, this equates to "mac" (the Mac OS Roman character set). On Windows, it equates to "Windows-1252" (the Windows Western character set). This may give better results than UTF-8 for some documents and with some browsers but you should use it only if you determine that UTF-8 doesn't work for your application.
`""`	Writes the document using the native character set of the operating system but omits the Content-Type meta tag.

In addition, you can provide any other string of 63 or fewer bytes. The main use for this is for documents that are primarily in an Asian language. For example, if your document is primarily in Japanese, then the characters in the document are already in the Shift-JIS encoding. You can specify "Shift-JIS" as the *encodingName*. This uses "Shift-JIS" in the Content-Type meta tag and to write the text out without any special encoding. The main advantage of this technique over the UTF-8 encoding is that a file written using Shift-JIS can be edited in Igor or other commonly-available text editors while UTF-8 can not.

When creating an HTML file, SaveNotebook can write pictures using the PNG or JPEG graphics formats. PNG support is built into Igor Pro. Built-in JPEG support was added in Igor Pro 5. Previously Igor used QuickTime to write JPEG files. It still does if QuickTime is available but if not, it uses built-in routines. For most applications, the only reason to use JPEG rather than PNG is that some old web browsers do not support PNG.

See Also

Chapter III-1, **Notebooks**.

Setting Bit Parameters on page IV-12 for further details about bit settings.

SavePackagePreferences

```
SavePackagePreferences [/FLSH=flush /KILL /P=pathName] packageName,
   prefsFileName, recordID, prefsStruct
```

The SavePackagePreferences operation saves preference data in the specified structure so that it can be accessed later via by the **LoadPackagePreferences** operation.

The data is stored in memory and by default flushed to disk when the current experiment is saved or closed and when Igor quits.

If the /P flag is present then the location on disk of the preference file is determined by *pathName* and *prefsFileName*. However in the usual case the /P flag will be omitted and the preference file is located in a file named *prefsFileName* in a directory named *packageName* in the Packages directory in Igor's preferences directory.

Note: You must choose a very distinctive name for *packageName* as this is the only thing preventing collisions between your package and someone else's package.

See **Saving Package Preferences** on page IV-226 for background information and examples.

Parameters

packageName is the name of your package of Igor procedures. It is limited to 31 characters and must be a legal name for a directory on disk. This name must be very distinctive as this is the only thing preventing collisions between your package and someone else's package.

prefsFileName is the name of a preference file to be saved by SavePackagePreferences. It should include an extension, typically ".bin".

prefsStruct is the structure containing the data to be saved in the preference file on disk.

recordID is a unique positive integer that you assign to each record that you store in the preferences file. If you store more than one structure in the file, you would use distinct *recordID*s to identify which structure you want to save. In the simple case you will store just one structure in the preference file and you can use 0 (or any positive integer of your choice) as the *recordID*.

Flags

/FLSH=*flush* Controls when the data is actually written to the preference file:

- 0: The data will be flushed to disk when the current experiment is saved, reverted or closed or when Igor quits. This is the default behavior used when /FLSH is omitted and is recommended for most purposes.
- 1: The data is flushed to disk immediately.

/KILL Instead of saving prefsStruct under the specified record ID, that record is deleted from the package's preference if it exists. If it does not exist, nothing is done and no error is returned.

/P=*pathName* Specifies the directory in which to save the file specified by *prefsFileName*.

pathName is the name of an existing symbolic path. See **Symbolic Paths** on page II-34 for details.

`/P=$<empty string variable>` acts as if the /P flag were omitted.

Details

SavePackagePreferences sets the following output variables:

`V_flag` Set to 0 if preferences were successfully saved or to a nonzero error code if they were not saved. The latter case is unlikely and would indicate some kind of corruption such as if Igor's preferences directory were deleted.

`V_structSize` Set to the size in bytes of prefsStruct. This may be useful in handling structure version changes.

Example

See the example under **Saving Package Preferences in a Special-Format Binary File** on page IV-226.

See Also

LoadPackagePreferences.

SavePICT

SavePICT [***flags***] [**as** ***fileNameStr***]

The SavePICT operation creates a picture file representing the top graph, table or layout. The picture file can be opened by many word processing, drawing, and page layout programs.

To use any of the QuickTime graphic export formats (/T flag), you will need to have QuickTime 4 or later installed (full install) in your system.

Parameters

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

If you omit *fileNameStr* but include /P=*pathName*, SavePICT writes the file using a default file name. The default file name is the window name followed by an extension, such as ".png", ".emf" or ".PICT", that depends on the graphic format being exported.

If you specify the file name as "`Clipboard`", and do *not* specify a /P=*pathName*, Igor copies the picture to the Clipboard, rather than to a file. EPS is a file-only format and can not be stored in the clipboard. Also QuickTime formats (/T) can not be stored in the clipboard.

If you specify the file name as "`_string_`" the output will be saved into a string variable named S_Value, which is used with the **ListBox** binary bitmap display mode.

If you use the special name _PictGallery_ with the /P flag, then the picture will be stored in Igor's picture collection (see **Pictures** on page III-421) with the name you provide via *fileNameStr*. QuickTime formats (/T) can not be put in the picture collection. This feature was added in support of making movies using the /PICT flag with **NewMovie**. It requires Igor Pro 6.12 or later.

Flags

/B=*dpi* Controls image resolution in dots-per-inch (*dpi*). The legal values for *dpi* are *n**72 where *n* can be from 1 to 8. The actual image *dpi* is not used. Igor calculates *n* from your value of *dpi* and then multiplies *n* by your computer's screen resolution. This is because bitmap images that are not an integer multiple of the screen resolution look quite bad. The /B flag can be used with QuickTime export types (/T).

Also see the /RES flag.

/C=*c* Specifies color mode.

c=0: Black and white.

c=1: RGB color (default).

c=2: CMYK color (EPS and native TIFF only).

/D=*d* Specifies bit depth for QuickTime image export. *d* may be 1, 8, 16, 24 (default), or 32 bits. Not all formats use depth. An intermediate picture at the current screen depth is used and may limit the quality.

/E=*e* Sets graphics format used when exporting a graphic. See **Details** for formats. See also Chapter III-5, **Exporting Graphics (Macintosh)**, or Chapter III-6, **Exporting Graphics (Windows)**, for a description of these modes and when to use them.

/EF= *e* Sets font embedding.

e=0: No font embedding.

e=1: Embed nonstandard fonts.

e=2: Embed all fonts.

/I Specifies that /W coordinates are inches.

/M Specifies that /W coordinates are centimeters.

/N=*winSpec* Deprecated: Igor Pro 4.09-compatibility version of /WIN.

/O Overwrites file if it exists.

/P=*pathName* Saves file into a folder specified by *pathName*, which is the name of an existing symbolic path.

/PICT=*pict* Saves specified named picture rather than the target window. Native format of the picture is used and all format flags are ignored.

/PLL=*p* Specifies Postscript language level when used in conjunction with EPS export.

p=1: For very old Postscript printers.

p=2: For all other uses (default).

/Q=*q* Sets quality factor (0.0 is lowest, 1.0 is highest). Default is dependent on individual format. Used only by lossy formats such as QuickTime JPEG.

/R=*resID* Saves the picture in a resource fork with resource ID=*resID*, where *resID* is between 1 and 32767.

This flag is meaningful on Macintosh only. It is ignored on Windows. Also, it does not work with EPS or PNG graphics.

/RES=*dpi* Controls the resolution of image formats in dots-per-inch. Unlike the similar /B flag, the value for /RES is the actual output resolution and is useful when your publisher demands a specific resolution. The /RES flag can be used with QuickTime export types (/T).

/S Suppresses the preview that is normally included with an EPS file.

/SNAP=*s* Saves a snapshot (screen dump) of a graph or panel window.

s=1: Include all controls in capture.

s=2: Capture only window data content.

Snapshot mode is available only for graphs and panels and only for bitmap export formats PNG, JPEG, and TIFF at screen resolution. When using /W to specify the size of a graph, the capture is sized to fit within the specified rectangle while maintaining the window aspect ratio. Coordinates used with /W are in pixels.

/T=*t* Specifies a QuickTime export type. An error will be generated if QuickTime is not present or if the desired export type is not available. *t* is a string containing a 4 character code for the export type. Valid types include "PNGf" for PNG, "TIFF" for TIFF, "JPEG" for JPEG. GIF is "GIFf" but is not currently supported by QuickTime presumably due to the patent situation. This might change in the future if Apple or a third party provides a component for GIF writing.

/TRAN[=1 *or* 0] Makes white background areas transparent using an RGBA type PNG when used with native PNG export of graphs or page layouts. In layouts, only embedded graphs and textboxes marked as transparent will be transparent. Nonembedded graphs, pictures, and tables will be opaque. PNG transparency is not honored when placed in Igor.

On Macintosh in new graphics mode (see **Graphics Technology** on page III-421) under some versions of OS X, you may find text is not properly transparent. In this instance, you can use /TRAN=2 to turn off font smoothing which is the source of the problem.

/W=(*left*,*top*,*right*,*bottom*)

Specifies the size of the picture when exporting a graph. If /W is omitted, it uses the graph window size. Specify /W=(0,0,0,0) to use a full page size.

When exporting a page layout, specifies the part of the page to export. Only objects that fall completely within the specified area are exported. If /W is omitted, the area of the layout containing objects is exported.

Coordinates for /W are in points unless /I or /M are specified before /W.

/WIN=*winSpec* Saves the named window or subwindow. *winSpec* can be just a window name, or a window name following by a "#" character and the name of the subwindow, as in `/WIN=Panel0#G0`.

/Z Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SavePICT sets the variable V_flag to 0 if the operation succeeds or to a nonzero error code if it fails.

If you specify a path using the /P=*pathName* flag, then Igor saves the file in the folder identified by the path. Note that *pathName* is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-34 for details. Otherwise, with no path specified, Igor presents a standard save dialog to let you specify where the file is to be saved.

When writing an EPS file, it normally includes a screen preview. On Macintosh, this is a PICT in the resource fork of the file. On Windows, it is a TIFF embedded in the EPS file. Using the /S flag, you can suppress the preview to save disk space, to work around a program that is confused by the preview, or to create a completely platform-neutral file.

On Macintosh, the picture is normally written to the data fork of the file. Most applications read it from the data fork. Some, however, read PICTs from the resource fork. If you use /R=*resID*, it writes the PICT to the resource fork of the file, using *resID* as the resource ID.

Graphics formats, specified via /E, are as follows:

/E Value	Macintosh File Format	Windows File Format
-8	PDF file.	PDF file.
-7	TIFF file. Lossless but larger file than PNG; best for text, graph traces, and simple images with sharp edges. The default resolution is 72 dpi. You can specify the resolution with the /B or /RES flag. Cross-platform compatible.	
-6	JPEG file. Lossy compression; best used for grayscale and color images with smooth tones. The /Q flag specifies compression quality and the /B or /RES flag sets the resolution. Cross-platform compatible.	
-5	PNG (Portable Network Graphics) file. Lossless compression; best for text, graph traces, and simple images with sharp edges. The default resolution is 72 dpi. Specify the resolution with /B or /RES. Cross-platform compatible.	
-4	High resolution bitmap PICT file. Default resolution is 288 dpi. Specify the resolution with /B or /RES.	Device-independent bitmap file (DIB). Default resolution is 4x screen resolution. Specify the resolution with /B or /RES.
-3	Encapsulated PostScript (EPS) file.	Encapsulated PostScript (EPS) file.
-2	Quartz PDF.	High-resolution Enhanced Metafile (EMF).
-1	Quartz PDF (was PostScript PICT).	Obsolete (was PostScript-enhanced metafile).
0	Quartz PDF (was PostScript PICT with QuickDraw text).	Obsolete (was PostScript-enhanced metafile).
1	Low resolution Quartz PDF at 1x normal size.	High-resolution Enhanced Metafile (EMF).
2	Low resolution Quartz PDF at 2x normal size.	High-resolution Enhanced Metafile (EMF).
4	Low resolution Quartz PDF at 4x normal size.	High-resolution Enhanced Metafile (EMF).
8	Low resolution Quartz PDF at 8x normal size.	High-resolution Enhanced Metafile (EMF).

Prior to Igor Pro 6.1, e=-2 on Macintosh created a high-resolution Macintosh PICT.

The low resolution PDF formats on Macintosh are probably not useful and are just placeholders for compatibility with old procedures. Prior to Igor Pro 6.1, e=1, e=2, e=4 and e=8 created Macintosh PICTs on Macintosh.

On Windows, for *e*>=1, Enhanced Metafile (EMF) is used except for graph and table windows when old graphics mode is in effect (see **Graphics Technology** on page III-421) in which case Windows Metafile (WMF) is used at *e* times screen resolution.

See Also

The **ImageSave** operation for saving waves as PICTs and other image file formats. The **LoadPICT** operation.

See Chapter III-5, **Exporting Graphics (Macintosh)**, or Chapter III-6, **Exporting Graphics (Windows)**, for a description of the /E modes.

SaveTableCopy

SaveTableCopy [*flags*] [**as** *fileNameStr*]

The SaveTableCopy operation saves a copy of the data displayed in a table on disk. The saved file can be an Igor packed experiment file, a tab-delimited text file, or a comma-separated values text file.

When saving as text, the data format matches the format shown in the table. Keep in mind that this will cause truncation if the underlying data has more precision than shown in the table. The point column is never saved.

To save data as text with full precision, use the **Save** operation.

When saving 3D and 4D waves as text, only the visible layer is saved. To save the entirety of a 3D or 4D wave, use the **Save** operation.

Parameters

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-398 for details on forming the path.

Flags

/A=*a* Appends data to the file rather than overwriting.

a=0: Does not append.

a=1: Appends to the file with a blank line before the appended data.

a=2: Appends to the file with no blank line before the appended data.

/A applies when saving text files and is ignored when saving packed experiment files.

If the file does not exist, a new file is created and /A has no effect.

/I Presents a dialog from which you can specify file name and folder.

/M=*termStr* Specifies the terminator character or characters to use at the end of each line of text. The default is `/M="\r"` on Macintosh and `/M="\r\n"` on Windows; it is used when /M is omitted. To use the Unix convention, just a linefeed, specify `/M="\n"`.

/N=*n* Specifies whether to use column names, titles, or dimension labels.

n is a bitwise parameter with the bits defined as follows:

Bit 0: Include column names or titles. The column title is included if it is not empty. If it is empty, the column name is included.

Bit 1: Include horizontal dimension labels if they are showing in the table.

The default setting for *n* is 1. All other bits are reserved and must be zero.

/O Overwrites file if it exists already.

/P=*pathName* Specifies the folder to store the file in. *pathName* is the name of an existing symbolic path.

/S=*s* Saves all of the data in the table (*s*=0; default) or the selection only (*s*=1).

/S applies when saving text files and is ignored when saving packed experiment files.

/T=*saveType* Specifies the file format of the saved table.

saveType=0: Packed experiment file.

saveType=1: Tab-delimited text file.

saveType=2: Comma-separated values text file.

saveType=3: Space-delimited values text file.

/W= *winName* *winName* is the name of the table to be saved. If /W is omitted or if *winName* is `""`, the top table is saved.

/Z Errors are not fatal and error dialogs are suppressed. See Details.

Details

The main uses for saving a table as a packed experiment are to save an archival copy of data or to prepare to merge data from multiple experiments (see **Merging Experiments** on page II-32). The resulting experiment file preserves the data folder hierarchy of the waves displayed in the table starting from the "top" data folder, which is the data folder that encloses all waves displayed in the table. The top data folder becomes the root data folder of the resulting experiment file. Only the table and its waves are saved in the packed experiment file, not variables or strings or any other objects in the experiment.

SaveTableCopy does not know about dependencies. If a table contains a wave, wave0, that is dependent on another wave, wave1 which is not in the table, SaveTableCopy will save wave0 but not wave1. When the saved experiment is open, there will be a broken dependency.

The main use for saving as a tab or comma-delimited text file is for exporting data to another program.

When calling SaveTableCopy from a procedure, you should call DoUpdate before calling SaveTable copy. This insures that the table is up-to-date if your procedure has redimensioned or otherwise changed the number of points in the waves in the table.

SaveTableCopy sets the variable V_flag to 0 if the operation completes normally, to -1 if the user cancels, or to another nonzero value that indicates that an error occurred. If you want to detect the user canceling an interactive save, use the /Z flag and check V_flag after calling SaveTableCopy.

The **SaveData** operation also has the ability to save a table to a packed experiment file. SaveData is more complex but a bit more flexible than SaveTableCopy.

Examples

This function saves all tables to a single tab-delimited text file.

```
Function SaveAllTablesToTextFile(pathName, fileName)
   String pathName        // Name of an Igor symbolic path.
   String fileName

   String tableName
   Variable index

   index = 0
   do
      tableName = WinName(index, 2)
      if (strlen(tableName) == 0)
         break
      endif

      SaveTableCopy/P=$pathName/W=$tableName/T=1/A=1 as fileName

      index += 1
   while(1)
End
```

See Also

SaveGraphCopy and **SaveData** operations; **Merging Experiments** on page II-32.

sawtooth

sawtooth(*num*)

The sawtooth function returns ((*num* +n2π) mod 2π)/2π where n is used to correct if *num* is negative. Sawtooth is used to create arbitrary periodic waveforms like sine and cosine.

Examples

```
wave1 = sawtooth(x)
```

creates a sawtooth in wave1 whose Y values range from 0 to 1 as its X values go through 2π units.

```
wave1 = exp(sawtooth(x))
```

creates a series of exponentials in wave1 of amplitude exp(1) and period 2π.

You can also use sawtooth to create periodic repetitions of a given part of a wave:

```
wave1 = wave2(sawtooth(x))
```

creates a periodic repetition of wave2 in wave1 given the correct X scaling for the waves.

ScreenResolution

ScreenResolution

The ScreenResolution function returns the logical resolution of your video display screen in dots per inch (dpi). On Macintosh this is always 72. On Windows it is usually 96 (small fonts) or 120 (large fonts).

Examples

```
// 72 is the number of points in an inch which is constant.
Variable pixels = numPoints * (ScreenResolution/72)   // Convert points to pixels
Variable points = numPixels * (72/ScreenResolution)   // Convert pixels to points
```

sec

sec(*angle*)

The sec function returns the secant of *angle* which is in radians:

$$\sec(x) = \frac{1}{\cos(x)}.$$

In complex expressions, *angle* is complex, and sec(*angle*) returns a complex value.

See Also
sin, **cos**, **tan**, **csc**, **cot**

Secs2Date

Secs2Date(*seconds*, *format* [, *sep*])

The Secs2Date function returns a string containing a date.

With *format* values 0, 1, and 2, the formatting of dates depends on your operating system and on your preferences entered in the Date & Time control panel (*Macintosh*) or the Regional Settings control panel (*Windows*).

If *format* is -1, the format is independent of the operating system. The fixed-length format is "*day /month /year* (*dayOfWeekNum*)", where *dayOfWeekNum* is 1 for Sunday, 2 for Monday… and 7 for Saturday.

If format is -2, the format is YYYY-MM-DD.

The optional sep parameter affects format -2 only. If sep is omitted, the separator character is "-". Otherwise, sep specifies the separator character.

Parameters
seconds is the number of seconds from 1/1/1904 to the date to be returned.

format is a number between -2 and 2 which specifies how the date is to be constructed.

Examples
```
Print Secs2Date(DateTime,-2)      // 1993-03-14
Print Secs2Date(DateTime,-2,"/")  // 1993/03/14
Print Secs2Date(DateTime,-1)      // 15/03/1993 (2)
Print Secs2Date(DateTime,0)       // 3/15/93
Print Secs2Date(DateTime,1)       // Monday, March 15, 1993
Print Secs2Date(DateTime,2)       // Mon, Mar 15, 1993
```

See Also
For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-102.

The **date**, **date2secs** and **DateTime** functions.

Secs2Time

Secs2Time(*seconds*, *format*, [*fracDigits*])

The Secs2Time function returns a string containing a time.

Parameters
seconds is the number of seconds from 1/1/1904 to the time to be returned.

format is a number between 0 and 5 that specifies how the time is to be constructed. It is interpreted as follows:

0: Normal time, no seconds.
1: Normal time, with seconds.
2: Military time, no seconds.
3: Military time, with seconds and optional fractional seconds.
4: Elapsed time, no seconds.
5: Elapsed time, with seconds and optional fractional seconds.

"Normal" formats (0 and 1) follow the preferred formatting of the short time format as set in the International control panel (*Macintosh*) or in the Regional and Language Options control panel (*Windows*).

"Military" means that the hour is a number from 0 to 23. Hours greater than 23 are wrapped.

"Elapsed" means that the hour is a number from -9999 to 9999. The result for hours outside that range is undefined.

The *fracDigits* parameter is optional and specifies the number of digits of fractional seconds. The default value is 0. The *fracDigits* parameter is ignored for format=0, 1, 2,and 4.

Examples

```
Print Secs2Time(DateTime,0)                 // prints 1:07 PM
Print Secs2Time(DateTime,1)                 // prints 1:07:28 PM
Print Secs2Time(DateTime,2)                 // prints 13:07
Print Secs2Time(DateTime,3)                 // prints 13:07:29
Print Secs2Time(30*60*60+45*60+55,4)        // Prints 30:45
Print Secs2Time(30*60*60+45*60+55,5)        // Prints 30:45:55
```

See Also

For a discussion of how Igor represents dates, see **Date/Time Waves** on page II-102.

The **Secs2Date**, **date**, **date2secs** and **DateTime** functions. Also, **Operators** on page IV-5 for ?: details.

SelectNumber

SelectNumber(*whichOne, val1, val2* [, *val3*])

The SelectNumber function returns one of *val1*, *val2*, or (optionally) *val3* based on the value of *whichOne*.

SelectNumber(*whichOne*, *val1*, *val2*) returns *val1* if *whichOne* is zero, else it returns *val2*.

SelectNumber(*whichOne*, *val1*, *val2*, *val3*) returns *val1* if *whichOne* is negative, *val2* if *whichOne* is zero, or *val3* if *whichOne* is positive.

Details

SelectNumber works with complex (or real)*val1*, *val2*, and *val3* when the result is assigned to a complex wave or variable. (Print expects a real result, see the "causes error" example, below).

If *whichOne* is NaN, then NaN is returned.

whichOne must always be a real value.

Unlike the ? : conditional operator, SelectNumber always evaluates all of the numeric expression parameters *val1*, *val2*, …

SelectNumber works in a macro, whereas the conditional operator does not.

Examples

```
Print SelectNumber(0,1,2)                       // prints 1
Print SelectNumber(0,1,2,3)                     // prints 2
wv=SelectNumber(numtype(wv[p])==2,wv[p],0)      // replace NaNs with zeros

// chooses among complex values
Variable/C cx= SelectNumber(negZeroPos,cmplx(-1,-1),0,cmplx(1,1))

// causes error because Print expects a real value (not complex)
Print SelectNumber(negZeroPos,cmplx(-1,-1),0,cmplx(1,1))

// The real function expects a complex result
Print real(SelectNumber(negZeroPos,cmplx(-1,-1),0,cmplx(1,1)))
```

See Also

The **SelectString** and **limit** functions, and **Waveform Arithmetic and Assignments** on page II-93. Also, **Operators** on page IV-5 for details about the ?: operator.

SelectString

SelectString(*whichOne, str1, str2* [, *str3*])

The SelectString function returns one of *str1*, *str2*, or (optionally) *str3* based on the value of *whichOne*.

SelectString(*whichOne*, *str1*, *str2*) returns *str1* if *whichOne* is zero, else it returns *str2*.

SelectString(*whichOne*, *str1*, *str2*, *str3*) returns *str1* if *whichOne* is negative, *str2* if *whichOne* is zero, or *str3* if *whichOne* is positive.

Details

If *whichOne* is NaN, then `""` is returned.

whichOne must always be a real value.

Unlike the ? : conditional operator, SelectString always evaluates all of the string expression parameters *str1*, *str2*, …

SelectString works in a macro, whereas the conditional operator does not.

Examples

```
Print SelectString(0,"hello","there")                 // prints "hello"
Print SelectString(1,"hello","there")                 // prints "there"
Print SelectString(-3,"hello","there","jack")         // prints "hello"
Print SelectString(0,"hello","there","jack")          // prints "there"
Print SelectString(100,"hello","there","jack")        // prints "jack"
```

See Also

The **SelectNumber** function and **String Expressions** on page IV-12. Also, **Operators** on page IV-5 for details about the ?: operator.

SetActiveSubwindow

SetActiveSubwindow *subWinName*

The SetActiveSubwindow operation specifies the subwindow that is to be activated. This command is mainly for use by recreation macros.

Parameters

subWinName is the name of an existing subwindow. Use _endfloat_ for *subWinName* to make a newly-created floating panel not be the default target.

When identifying a subwindow with *subWinName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

SetAxis

SetAxis [***flags***] ***axisName*** [**, *num1*, *num2***]

The SetAxis operation sets the extent (or "range") of the named axis.

Parameters

axisName is usually "left", "right", "top" or "bottom", but it can also be the name of a free axis, such as "vertCrossing".

If *axisName* is a vertical axis such as "left" or "right" then *num1* sets the bottom end of the axis and *num2* sets the top end of the axis.

If *axisName* is a horizontal axis such as "top" or "bottom" then *num1* sets the left end of the axis and *num2* sets the right end of the axis.

You can flip the graph by reversing *num1* and *num2* (or by using /A/R). This is particularly useful for images, because Igor plots an image inverted.

Flags

/A[=*a*] Autoscale axis (when used, *num1*, *num2* should be omitted)

- *a*=0: No autoscale. Same as no /A flag.
- *a*=1: Normal autoscale. Same as /A.
- *a*=2: Autoscale Y axis to a subset of the data defined by the current X axis range.

/E=*z* Sets the treatment of zero when the axis is in autoscale mode.

- *z*=0: Normal mode where zero is not treated special.
- *z*=1: Forces the smaller end of the axis to be set to zero (autoscale from zero).
- *z*=2: Axis is symmetric about zero.
- *z*=3: If the data is unipolar (all positive or all negative), this behaves like /E=1 (autoscale from zero). If the data is bipolar, it behaves like /E=0 (normal autoscaling).

/N=*n* Sets the algorithm for axis autoscaling.

- *n*=0: Normal mode; sets the axis limits equal to the data limits.
- *n*=1: Picks nice values for the axis limits.
- *n*=2: Picks nice values; also ensures that the data is inset from the axis ends.

/R Reverses the autoscaled axis (smaller values at the left for horizontal axes, at the top for vertical axes) when used with /A. Although it only has an effect for autoscale, it can be used with nonautoscale version of SetAxis so that the next time the Axis Range tab is used the "reverse axis" checkbox will already be set.

/W=*winName*	Sets axes in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.
/Z	No error reporting if named axis doesn't exist in a style macro.

SetBackground

SetBackground *numericExpression*

The SetBackground operation sets *numericExpression* as the current unnamed background task.

SetBackground works only with the unnamed background task. New code should used named background tasks instead. See **Background Tasks** on page IV-279 for details.

The background task runs while Igor is not busy with other things. Normally, there won't be a background task. The most common use for the background task is to monitor or drive a continuous data acquisition process.

Parameters

numericExpression is a single precision numeric expression that Igor executes when it isn't doing anything else.

Details

numericExpression is expected to return one of three numeric values:

0: Background task executed normally.

1: Background task wants to stop.

2: Background task encountered error and wants to stop.

Usually the expression will be a call to a user-defined numeric function or external function to drive or monitor data acquisition. The expression should be designed to execute very quickly and it should not present a dialog to the user nor should it create or destroy windows. Generally, it should do nothing more than store data into waves or variables. You can use Igor's dependency mechanism to perform more extensive tasks.

SetBackground designates the background task but you must use CtrlBackground to start it. You can also use KillBackground to stop it. You can not call SetBackground from the background function itself.

See Also

The **BackgroundInfo**, **CtrlBackground**, **CtrlNamedBackground**, **KillBackground**, and **SetProcessSleep** operations, and **Background Tasks** on page IV-279.

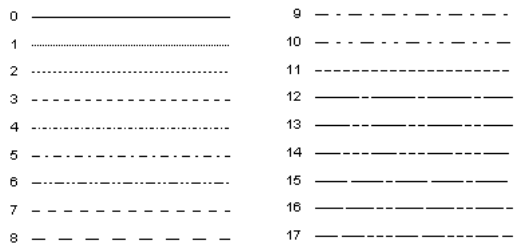
SetDashPattern

SetDashPattern *dashNumber*, {*d1, s1* [*, d2, s2*]…}

The SetDashPattern operation defines a dashed-line pattern for a user-defined dashed line. These dashed lines are used by the drawing tools and the Modify Waves Appearance dialog, and are elsewhere referred to as "line styles".

Parameters

dashNumber specifies which dash pattern is to be set. It must be between 1 and 17. Dash pattern 0 is reserved for a solid line.



{*d1,s1* [*,d2,s2*]…} defines the dash pattern. The dash pattern consists of 1 to 8 "dash,skip" pairs. Each pair consists of the number of drawn points followed by the number of skipped points.

d1 specifies the number of drawn points and *s1* specifies the number of skipped points in the first "dash,skip" pair. *d2* and *s2* specify the number of drawn and skipped points in the second pair and so on. Each draw or skip value must be between 1 and 127.

Details

SetDashPattern updates *all* graphs, panels and layouts so that any dashed lines will be updated with the new pattern. If you repeatedly call SetDashPattern from within a macro, you should precede the commands with the PauseUpdate operation to prevent multiple updates (which would be slow).

Dashed lines may also be redefined by the Dashed Lines dialog which you can choose from the Misc menu.

The dashed line patterns are saved as part of the experiment. When a new experiment is opened, the preferred dash patterns are restored.

Some programs and printer drivers do not properly render dashed lines with many "dash,skip" pairs.

Examples

```
Make test; Display test
SetDashPattern 17, {20,3,15,8}       // sets last dashed line pattern
ModifyGraph lstyle(test)=17          // apply pattern to trace
```

See Also

PauseUpdate and **ResumeUpdate** operations, and **Dashed Lines** on page III-410.

SetDataFolder

SetDataFolder *dataFolderSpec*

The SetDataFolder operation sets the current data folder to the specified data folder.

Parameters

dataFolderSpec can be a simple name (MyDataFolder), a path (root:MyDataFolder) or a string expression containing a name or path. It can also be a data folder reference created by the **DFREF** keyword or returned by **GetDataFolderDFR**.

If *dataFolderSpec* is a path it can be a partial path relative to the current data folder (:MyDataFolder) or an absolute path starting from root (root:MyDataFolder).

Examples

```
SetDataFolder foo             // Sets CDF to foo in the current data folder.
SetDataFolder :bar:foo        // Sets CDF to foo in bar current data folder.
SetDataFolder root:foo        // Sets CDF to foo in the root data folder.
String savedDF= GetDataFolder(1)     // Remember CDF in a string.
NewDataFolder/O/S root:MyDataFolder  // Set CDF to a new data folder.
Variable/G newVariable=1             // Do work in the new data folder.
SetDataFolder savedDF                // Restore CDF from the string value.
```

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-61.

SetDimLabel

SetDimLabel *dimNumber, dimIndex, label, wavelist*

The SetDimLabel operation sets the dimension label or dimension element label to the specified label.

Parameters

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers and 3 for chunks.

If *dimIndex* is -1, it sets the label for the entire dimension. For dimIndex ≥ 0, it sets the dimension label for that element of the dimension.

label is a name (e.g., time), not a string (e.g., "time").

label is limited to 31 characters.

See Also

The **GetDimLabel** function. **Dimension Labels** on page II-109 and **Example: Wave Assignment and Indexing Using Labels** on page II-99 for further usage details and examples.

SetDrawEnv

SetDrawEnv [**/W=*winName***] ***keyword*** [**=*value***][**, *keyword*** [**=*value***]]…

The SetDrawEnv operation sets properties of the drawing environment.

If one or more draw objects are selected in the top window then the SetDrawEnv command will apply only to those objects.

If no objects are selected *and* if the keyword save *is not* used *then* the command applies only to the next object drawn.

If no objects are selected *and* if the keyword "save" *is* used *then* the command sets the environment for all following objects.

Each draw layer has its own draw environment settings.

Parameters

SetDrawEnv can accept multiple *keyword=value* parameters on one line.

In the following descriptions, (*r*, *g*, *b*) specifies a color. *r*, *g*, and *b* are each a number from 0 to 65535. (0, 0, 0) specifies black. (65535, 65535, 65535) specifies white.

Also note that the abs and rel values for the coordinate keywords "xcoord" and "ycoord" are the literal characters "abs" and "rel"; they are not substitute names for numbers, names, or strings.

arrow=*arr* — Specifies the arrow head position on lines.
- *arr*=0: No arrowhead (default)
- *arr*=1: Arrowhead at end.
- *arr*=2: Arrowhead at start.
- *arr*=3: Arrowhead at start and end.

arrowfat=*afat* — Sets ratio of arrowhead width to length (default is 0.5).

arrowlen=*alen* — Sets length of arrowhead in points (default is 10).

arrowSharp=*s* — Specifies the continuously variable barb sharpness between -1.0 and 1. 0.
- *s*=1: No barb; lines only.
- *s*=0: Blunt (default).
- *s*=-1: Diamond.

arrowframe=*f* — Specifies the stroke outline thickness of the arrow in points (default is *f*=0 for solid fill).

astyle=s — Specifies which side of the line has barbs relative to a right-facing arrow.
- *s*=0: None.
- *s*=1: Top.
- *s*=2: Bottom.
- *s*=3: Both (default).

dash=*dsh* — *dsh* is a dash pattern number between 0 and 17 (see **SetDashPattern** for patterns). 0 (solid line) is the default.

fillbgc=(*r*, *g*, *b*) — Specifies fill background color. Default is the window's background color.

fillfgc=(*r*, *g*, *b*) — Specifies fill foreground color. The default is white.

fillpat=*fpatt* — Specifies fill pattern density.
- *fpatt*=-1: Erase to background color.
- *fpatt*=0: No fill.
- *fpatt*=1: 100% (solid pattern, default).
- *fpatt*=2: 75% gray.
- *fpatt*=3: 50% gray.
- *fpatt*=4: 25% gray.

fname="*fontName*" — Sets font name, default is the default font or the graph font.

fsize=*size* — Sets text size, default is 12 points.

fstyle=*style* — Sets text style (same as for the **TextBox** operation), default is plain.

gedit= *flag*	Supplies optional edit flag for a group of objects. Use with gstart. *flag*=0: Select entire group, moveable (default). *flag*=1: Individual components editable as if not grouped. Allows objects to be grouped by name but still be editable
gname= *name*	Supplies optional *name* for an object group. Use with gstart.
gstart	Marks the start of a group of objects.
gstop	Marks the end of a group of objects.
linebgc=(*r*, *g*, *b*)	Sets the line background color. Default is window's background color.
linefgc=(*r*, *g*, *b*)	Sets the line foreground color, default is black.
linepat=*patt*	Specifies the line pattern/density. *patt*=1: 100% (solid pattern, default). *patt*=2: 75% gray. *patt*=3: 50% gray. *patt*=4: 25% gray.
linethick=*thick*	*thick* is a line thickness ≥ 0, default is 1 point.
origin= *x0*,*d0*	Moves coordinate system origin to *x0*,*d0*. Unlike translate, rotate, and scale, this survives a change in coordinate system and is most useful that way. See **Coordinate Transformation**.
pop	Pops a draw environment from the stack. Pops should always match pushes.
push	Pushes the current draw environment onto a stack (limited to 10).
rotate= *deg*	Rotates coordinate system by *deg* degrees. Only makes sense if X and Y coordinate systems are the same. See **Coordinate Transformation**.
rounding=*rnd*	Radius for rounded rectangles in points, default is 10.
rsabout	Redefines coordinate system rotation or scaling to occur at the translation point instead of the current origin. To use, combine rotate or scale with translate and rsabout parameters.
save	Stores the current drawing environment as the default environment.
scale= *sx*,*sy*	Scales coordinate system by *sx* and *sy*. Affects only coordinates — not line thickness or arrow head sizes. See **Coordinate Transformation**.
textrgb=(*r*, *g*, *b*)	Sets text color, default is black.
textrot=*rot*	Text rotation in degrees. *rot* is a value from -360 to 360. 0 is normal (default) horizontal left-to-right text, 90 is vertical bottom-to-top text, etc.
textxjust=*xj*	Sets horizontal text alignment. *xj*=0: Left aligned text (default). *xj*=1: Center aligned text. *xj*=2: Right aligned text.
textyjust=*yj*	Sets vertical text alignment. *yj*=0: Bottom aligned text (default). *yj*=1: Middle aligned text. *yj*=2: Top aligned text.
translate= *dx*,*dy*	Shifts coordinate system by *dx* and *dy*. Units are in the current coordinate system. See **Coordinate Transformation**.
xcoord=abs	X coordinates are absolute window coordinates (default for all windows *except graphs* where the default is xcoord=prel). The unit of measurement is pixels if the window is a panel, otherwise they are points. The left edge of the window (or of the printable area in a layout) is at x=0.
xcoord=rel	X coordinates are relative window coordinates. x=0 is at the left edge of the window; x=1 is at the right edge.

xcoord=prel	X coordinates are relative plot rectangle coordinates (graphs only). x=0 is at the left edge of the rectangle; y=1 is at the right edge of the rectangle. This coordinate system ideal for objects that should maintain their size and location relative to the axes, and *is the default for graphs*.
xcoord=*axisName*	X coordinates are in terms of the named axis (graphs only).
ycoord=abs	Y coordinates are absolute window coordinates (default for all windows *except graphs* where the default is ycoord=prel). The unit of measurement is pixels if the window is a panel, otherwise they are points. The top edge of the window (or the of the printable area in a layout) is at y=0.
ycoord=rel	Y coordinates are relative window coordinates. y=0 is at the top edge of the window; y=1 is at the bottom edge.
ycoord=prel	Y coordinates are relative plot rectangle coordinates (graphs only). y=0 is at the top edge of the rectangle; y=1 is at the bottom edge of the rectangle. This coordinate system ideal for objects that should maintain their size and location relative to the axes, and *is the default for graphs*.
ycoord=*axisName*	Y coordinates are in terms of the named axis (graphs only).

Flags

/W=*winName*	Sets the named window or subwindow for drawing. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Coordinate Transformation

The execution order for the translate, rotate, scale, and origin parameters is important. Translation followed by rotation is different than rotation followed by translation. When using multiple keywords in one SetDrawEnv operation, the order in which they are applied is origin, translate, rotate followed by scale regardless of the command order (with the exception of the rsabout parameter). Before using origin with the save keyword, you should use push to save the current draw environment and then use pop after drawing objects using the new origin.

Examples

Following is a simple example of arrow markers:

```
NewPanel
SetDrawEnv arrow= 1,arrowlen= 30,save
SetDrawEnv arrowsharp= 0.3
DrawLine 61,67,177,31
SetDrawEnv arrowsharp= 1
DrawLine 65,95,181,59
SetDrawEnv astyle= 1
DrawLine 69,123,185,87
SetDrawEnv arrowframe= 1
DrawLine 73,151,189,115
```

You can position objects in one coordinate system and then draw them in another with the origin keyword. In the following coordinate transformation example, we position arrows in axis units but size them in absolute units.

```
Make/O jack=sin(x/8)
Display jack
SetDrawEnv xcoord=bottom,ycoord=left,save
SetDrawEnv push
SetDrawEnv origin=50,0
SetDrawEnv xcoord=abs,ycoord=abs,arrow=1,arrowlen=20,arrowsharp=0.2,save
DrawLine 0,0,50,0          // arrow 50 points long pointing to the right
DrawLine 0,0,0,50          // arrow 50 points long pointing down
// now let's move over, rotate a bit and draw the same arrows:
SetDrawEnv translate=100,0
SetDrawEnv rotate=30,save
DrawLine 0,0,50,0
DrawLine 0,0,0,50
SetDrawEnv pop
```

Now try zooming in on the graph. You will see that the first pair of arrows always starts at 50 on the bottom axis and 0 on the left axis whereas the second pair is 100 points to the right of the first.

See Also
Chapter III-3, **Drawing**, and the **TextBox** and **DrawAction** operations.

SetDrawLayer

SetDrawLayer [**/K/W=*winName***] ***layerName***

The SetDrawLayer operation makes all future drawing operations use the named layer.

Parameters
Valid *layerName*s for graphs:

ProgBack	UserBack	ProgAxes	UserAxes	ProgFront	UserFront

Valid *layerName*s for panels and page layouts:

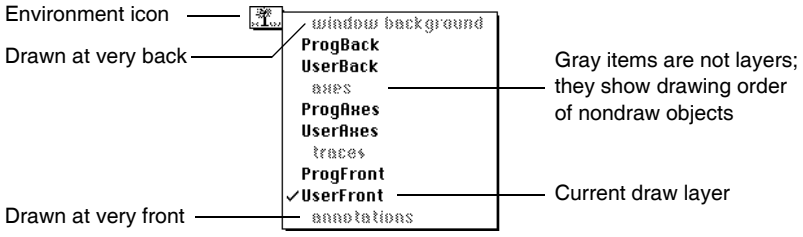
ProgBack	UserBack	ProgFront	UserFront

Flags

/K	Kills (erases) the given layer.
/W=*winName*	Sets the named window or subwindow for drawing. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details
There are really only two layers for Panels but ProgFront is allowable as an alias for ProgBack. Likewise UserFront is an alias for UserBack.

The back-to-front order of the layers is shown by the Environment pop-up menu if you hold down Option (*Macintosh*) or Alt (*Windows*). This is the Environment pop-up menu for graphs:



See Also
Layers on page III-78 and the **DrawAction** operation.

SetFileFolderInfo

SetFileFolderInfo [***flags***][***fileOrFolderNameStr***]

The SetFileFolderInfo operation changes the properties of a file or folder.

Parameters
fileOrFolderNameStr specifies the file or folder to be changed.

If you use a full or partial path for *fileOrFolderNameStr*, see **Path Separators** on page III-398 for details on forming the path.

Folder paths should not end with single Path Separators. See the **MoveFolder Details** section.

If Igor can not determine the location of the file or folder from *fileOrFolderNameStr* and /P=*pathName*, it displays a dialog allowing you to specify the file to be deleted. Use /D to select a folder in this event, otherwise Igor prompts your for a file.

Flags
At least one of the seven following flags is required, or nothing is actually accomplished:

/CDAT=*cdate* Specifies the number of seconds since midnight January 1, 1904 when the file or folder was first created.

/INV[=*inv*] Sets the visibility of a file.

inv=0: File is visible.

inv=1: Default; file is invisible (*Macintosh*) or Hidden (*Windows*).

/MDAT=*mDate* Specifies the number of seconds since midnight January 1, 1904 when the file or folder was modified most recently.

/RO[=*ro*] Sets the read/write state of a file or folder.

ro=0: File or folder is writable.

ro=1: File or folder is locked (default).

On Macintosh, locking the file or folder is equivalent to setting the locked property manually using the Get Info window in the Finder.

On Windows, locking the file or folder is equivalent to setting the read-only property manually using the Properties window in Windows Explorer.

If *fileOrFolderNameStr* refers to a file (not a folder), SetFileFolderInfo updates the file properties to reflect values given with the following keywords:

/CRE8=*creatorStr* Sets the four-character creator code string, such as 'IGR0' (Igor Pro creator code).

Ignored on Windows, where files have no “creator code”; instead file extensions are “registered” or “owned” by one, and only one, application. You cannot change that ownership from Igor Pro.

/FTYP=*fTypeStr* Sets the four-character file type code, such as 'TEXT' or 'IGsU' (packed experiment).

Ignored on Windows. Use **MoveFile** to change the file extension.

/STA[=*st*] Specifies whether the file is a stationery file or not.

st=1: Stationery file (default).

st=0: Normal file.

Ignored on Windows. Use **MoveFile** to change the file extension.

Optional Flags

/D Uses the Select Folder dialog rather than Open File dialog when *pathName* and *fileOrFolderNameStr* do not specify an existing file or folder.

/P=*pathName* Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

/R[=*r*] Recursively applies change(s) to all files or folders in the folder specified by /P=*pathName* or *fileOrFolderNameStr*, and the folder itself:

r=0: No recursion. Same as no /R.

r=1: Recursively apply changes to files.

r=2: Recursively apply changes to folders, including the folder specified by *pathName* or *fileOrFolderNameStr*.

r=3: Recursively apply changes to both files and folders (default).

/R requires /D and a folder specification.

/Z[=*z*] Prevents procedure execution from aborting if SetFileFolderInfo tries to set information about a file or folder that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort.

/Z=0: Same as no /Z at all.

/Z=1: Used for setting information for a file or folder only if it exists. /Z alone has the same effect as /Z=1.

/Z=2: Used for setting information for a file or folder if it exists and displaying a dialog if it does not exist.

Variables

SetFileFolderInfo returns information about the file or folder in the following variables:

`V_flag` 0: File or folder was found.

-1: User cancelled the Open File dialog.

>0: An error occurred, such as the specified file or folder does not exist.

S_path File system path of the selected file or folder.

Examples

Change the file creator code; no complaint if it doesn't exist:

```
SetFileFolderInfo/Z /CRE8="CWIE", "Macintosh HD:folder:afile.txt"
```

Set the file modification date:

```
Variable mDate= Date2Secs(2000,12,25) + hrs*3600+mins*60+secs
SetFileFolderInfo/P=myPath/MDAT=(mDate), "afile.txt"
```

Remove read-only property from a folder and everything within it:

```
SetFileFolderInfo/P=myPath/D/R/RO=0
```

See Also

The **GetFileFolderInfo**, **ImageFileInfo**, **MoveFile**, and **FStatus** operations. The **IndexedFile**, **date2secs**, and **ParseFilePath** functions.

SetFormula

SetFormula *waveName, expressionStr*

SetFormula *variableName, expressionStr*

The SetFormula operation binds the named wave, numeric or string variable to the expression or, if the expression is "", unbinds it from any previous expression. In user functions, SetFormula must be used to create dependencies.

Parameters

expressionStr is a string containing a numeric or string expression, depending on the type of the bound object.

Pass an empty string ("") for *expressionStr* to clear any previous dependency expression associated with the wave or variable.

Details

The dependent object (the wave or variable) will depend on the objects referenced in the string expression. The expression will be reevaluated any time an object referred to in the expression is modified.

Besides being set from a string expression this differs from just typing:

```
name := expression
```

in that syntax errors in *expressionStr* are not reported and are not fatal. You end up with a dependency assignment that is marked as needing to be recompiled. The recompilation will be attempted every time an object is created or when the procedure window is recompiled.

Use the Object Status dialog in the Misc menu to check up on dependent objects.

Examples

This command makes the variable v_sally dependent on the user-defined function anotherFunction, waves wave_fred and wave_sue, and the system variable K2:

```
SetFormula v_sally, "anotherFunction(wave_fred[1]) + wave_sue[0] + K2"
```

This is equivalent to:

```
v_sally := anotherFunction(wave_fred[1]) + wave_sue[0] + K2
```

except that no error will be generated for the SetFormula if, for instance, wave_fred does not exist.

A string variable dependency can be created by a command such as:

```
SetFormula myStringVar, "note(wave_joe)"
```

observe that *expressionStr* is a string *containing* a string expression, and that:

```
SetFormula myStringVar,note(wave_joe)
```

is not the same thing. In this case the note of wave_joe would contain the expression that myStringVar would depend on! Also, wave_joe would have to exist for Igor to understand the statement.

See Also

Chapter IV-9, **Dependencies**, and the **GetFormula** function.

SetIgorHook

SetIgorHook [/K/L] [*hookType* = [*procName*]]

The SetIgorHook operation tells Igor to call a user-defined "hook" function at the following times:

- Before a file is opened (**BeforeFileOpenHook**)
- After a file is opened (**AfterFileOpenHook**)
- After a window is created (**AfterWindowCreatedHook**)
- When a menu item is selected (**IgorMenuHook**)
- Before an experiment is saved (**BeforeExperimentSaveHook**)
- Before a new experiment is opened (**IgorBeforeNewHook**)
- Before the debugger is opened (**BeforeDebuggerOpensHook**)
- Before Igor quits (**IgorBeforeQuitHook**)
- During Igor's quit processing (**IgorQuitHook**)

The term “hook” is used as in the phrase “to hook into”, meaning to intercept or to attach.

Hook functions are typically used by a sophisticated procedure package to make sure that the package's private data is consistent.

In addition to using SetIgorHook, you can designate hook functions using fixed function names (see **User-Defined Hook Functions** on page IV-251). The advantage of using SetIgorHook over fixed hook names is that you don't have to worry about name conflicts.

You can designate hook functions for specific windows using window hooks (see **SetWindow** on page V-569).

Flags

/K Removes *procName* from the list of functions called for the *hookType* events.

If *procName* is not specified all *hookType* functions are removed.

If *hookType* is not specified all functions are removed for all *hookType* events, returning Igor to the pre-SetIgorHook state.

/L Executes *procName* last. Without /L, a newly added hook function runs before previously registered hook functions.

A function that has been previously registered with SetIgorHook can be moved from being called first to being called last by calling SetIgorHook again with /L.

To move a function from being called last to being called first requires removing the hook function with /K and then calling SetIgorHook without /L.

Parameters

hookType Specifies one of the fixed-name hook function names:

AfterFileOpenHook
AfterWindowCreatedHook
BeforeDebuggerOpensHook
BeforeExperimentSaveHook
BeforeFileOpenHook
IgorBeforeNewHook
IgorBeforeQuitHook
IgorMenuHook
IgorQuitHook

See the note below about these *hookType* names.

hookType is required except with /K.

procName Names the user-defined hook function that is called for the *hookType* event.

Details

The parameters and return type of the user-defined function *procName* varies depending on the *hookType* it is registered for.

For example, a function registered for the AfterFileOpenHook type must have the same parameters and return type as the shown for the **AfterFileOpenHook** on page IV-253.

The *procName* function is called *after* any window-specific hook for these *hookType*s, and the *procName* function is called *before* any other hook functions previously registered by calling SetIgorHook *unless the /L flag is given*, in which case it still runs after window-specific hook functions, but also *after* all other previously registered hook functions.

The *procName* function should return a nonzero value (1 is typical) to prevent later functions from being called. Returning 0 allows successive functions to be called.

SetIgorHook does not work at Igor start or new experiment time, so SetIgorHook IgorStartOrNewHook is disallowed. Define a global or static fixed-name **IgorStartOrNewHook** function (see page IV-263).

The saved Igor experiment file remembers the SetIgorHooks that are in effect when the experiment is saved:

Hook Function Interactions

After all the SetIgorHook functions registered for *hookType* have run (and all have returned 0), any static fixed-name hook functions are called and then the (only) fixed-name user-defined hook function, if any, is called. As an example, when a menu event occurs, Igor handles the event by calling routines in this order:

1. The top window's hook function as set by **SetWindow**
2. Any SetIgorHook-registered hook functions
3. All of the fixed-named **IgorMenuHook** functions:

3a. First any static IgorMenuHook functions (in any independent module)

3b. Then the one-and-only non-static IgorMenuHook function (in only the ProcGlobal independent module)

1. SetWindow event (called first)	2. SetIgorHook *hookType* (called second)	3. User-defined Hook Function(s) (called last)
enableMenu	IgorMenuHook	IgorMenuHook
menu	IgorMenuHook	IgorMenuHook

Note: Although you can technically use one of the fixed-name functions, as described in **User-Defined Hook Functions** on page IV-251, for *procName*, the result would be that the function will be called twice: once as a registered named hook function and once as the fixed-named hook function. That is, don't use SetIgorHook this way:

```
SetIgorHook AfterFileOpenHook=AfterFileOpenHook       // NO NO
```

Variables

SetIgorHook returns information in the following variables:

`S_info` Semicolon-separated list of all current hook functions associated with *hookType*, listed in the order in which they are called. As of Igor 6.13 the S_info includes the full independent module paths (e.g.,"ProcGlobal#MyMenuHook;MyIM#MyModule#MyMenuHook;").

Examples

This hook function invokes the Export Graphics menu item when Command-C (*Macintosh*) or Ctrl+C (*Windows*) is selected for a graph, preventing the usual Copy.

```
SetIgorHook IgorMenuHook=CopyIsExportHook

Function CopyIsExportHook(isSelection,menuName,itemName,itemNo,win,wType)
   Variable isSelection
   String menuName,itemName
   Variable itemNo
   String win
   Variable wType

   Variable handledIt= 0
   if( isSelection && wType==1 ) // menu was selected, window is graph
      if( Cmpstr(menuName,"Edit")==0 && CmpStr(itemName,"Copy")==0 )
         DoIgorMenu "Edit", "Export Graphics"       // dialog instead
         handledIt= 1           // don't call other IgorMenuHook functions.
      endif
   endif
   return handledIt
End
```

To unregister CopyIsExportHook as a hook procedure:

```
SetIgorHook/K IgorMenuHook=CopyIsExportHook // unregister CopyIsExportHook
```

To discover which functions are associated with a *hookType*, use a command such as:

```
SetIgorHook IgorMenuHook   // inquire about names registered for IgorMenuHook
Print S_info               // list of functions
```

To remove (or "unregister") named hooks:

```
SetIgorHook/K                 // removes all hook functions for all hookTypes
SetIgorHook/K IgorMenuHook    // removes all IgorMenuHook functions
SetIgorHook/K IgorMenuHook=CopyIsExportHook// removes only this hook function
```

See Also

The **SetWindow** operation and **User-Defined Hook Functions** on page IV-251.

Independent Modules on page IV-214.

SetIgorMenuMode

SetIgorMenuMode *MenuNameStr, MenuItemStr, Action*

The SetIgorMenuMode operation allows an Igor programmer to disable or enable Igor's built-in menus and menu items. This is useful for building applications that will be used by end-users who shouldn't have access to all Igor's extensive and confusing functionality.

Parameters

MenuNameStr — The name of an Igor menu, like "File", "Graph", or "Load Waves".

MenuItemStr — The text of an Igor menu item, like "Copy" (in the Edit menu) or "New Graph" (in the Windows menu).

Action — One of DisableItem, EnableItem, DisableAllItems, or EnableAllItems.

DisableItem and EnableItem disable or enable just the single item named by *MenuNameStr* and *MenuItemStr*. If *MenuItemStr* is `""`, then the menu itself is disabled.

DisableAllItems and EnableAllItems disable and enable all the items in the menu named by *MenuNameStr*.

Details

All menu names and menu item text are in English. This ensures that code developed for a localized version of Igor will run on all versions. Note that no trailing "..." is used in *MenuItemStr*.

The SetIgorMenuModeProc.ipf procedure file includes procedures and commands that disable or enable every menu and item possible. It is in your Igor Pro folder, in WaveMetrics Procedures:Utilities. It is not intended to be used as-is. You should make a copy and edit the copy to include just the parts you need.

The text of some items in the File menu changes depending on the type of the active window. In these cases you must pass generic text as the *MenuItemStr* parameter. Use "Save Window", "Save Window As", "Save Window Copy", "Adopt Window" and "Revert Window" instead of "Save Notebook" or "Save Procedure", etc. Use "Page Setup" instead of "Page Setup For All Graphs", etc. Use "Print" instead of "Print Graph", etc.

The Edit→Insert File menu item was previously named Insert Text. For compatibility reasons, you can specify either "Insert File" or "Insert Text" as MenuItemStr to modify this item.

See Also

The **DoIgorMenu** operation.

SetIgorOption

SetIgorOption [*mainKeyword*,] ***keyword= value***

SetIgorOption [*mainKeyword*,] ***keyword= ?***

The SetIgorOption operation makes unusual and temporary changes to Igor Pro's behavior. This operation is not compilable and you will need to use the **Execute** operation to use it in a user function. The details of the syntax depend on the application and are documented where the alternate behaviors are described. In most cases the current value of a setting can be read using the *keyword*=? syntax. Simple numeric options are stored in V_flag and color options are stored in V_Red, V_Green, and V_Blue. The settings last for the life of the Igor session.

See Also

Syntax Coloring on page III-352 for some usage examples; **Macintosh and LAPACK Library** on page III-142; **SetIgorOption IndependentModuleDev=1** on page IV-215; **Conditional Compilation** on page IV-86; **Pre-Carbon Page Setup Records** on page III-397. **MarkPerfTestTime** operation.

SetMarquee

SetMarquee [**/W=*winName***] ***left, top, right, bottom***

The SetMarquee operation creates a marquee on the target graph or layout window or the specified window or subwindow. The *left*, *top*, *right*, *bottom* coordinates are the same as those returned by the GetMarquee operation (screen units measured in points).

The optional axis modes supported by GetMarquee are not supported by SetMarquee.

Flags

/W=*winName* Specifies the named window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

See Also

The **GetMarquee** operation.

SetProcessSleep

SetProcessSleep *sleepTicks*

The SetProcessSleep operation determines how much time Igor will give to background tasks or other Macintosh applications executing in the background. This operation does nothing on Windows.

Parameters

sleepTicks is the amount of time given to background tasks in sixtieths of a second. *sleepTicks* values between 0 and 60 are valid.

Details

Igor starts up with sleepTicks = 1. Use 0 to give Igor maximum time, use a larger number to give other applications more time.

Background tasks are used mainly by data acquisition programs.

See Also

Background Tasks on page IV-279 and the **SetBackground** operation.

SetRandomSeed

SetRandomSeed *seed*

The SetRandomSeed operation seeds the random number generator used for the **enoise** and **gnoise** functions. Use SetRandomSeed if you need "random" numbers that are reproducible. If you don't use SetRandomSeed, the random number generator is initialized using the system clock when Igor starts. This almost guarantees that you will never get the same sequence twice unless you use SetRandomSeed.

Flags

/BETR[=better] If better is absent or non-zero, a better method is used for seeding the Mersenne Twister random number generator.

Parameters

seed should be a number in the interval (0, 1]. For any given *seed*, enoise or gnoise or any of the other random-number generator functions generates a particular sequence of pseudorandom numbers. Calling SetRandomSeed with the same seed restarts and repeats the sequence.

Details

Internally *seed* is scaled to a 32-bit unsigned integer. Consequently, the number of different values for the internally-scaled seed is less than the resolution of the double-precision numbers in the (0, 1] range.

You should use /BETR unless you need consistency with older versions of Igor. /BETR was introduced in Igor Pro 6.20.

The Mersenne Twister random number generator is used for most of Igor's noise functions (and optionally for enoise and gnoise), and internally for operations that need random sequences, such as **StatsSample** or **StatsResample**.

Without the /BETR flag, SetRandomSeed maps *seed* to an internal 16-bit integer for seeding the Mersenne Twister random number generator. With /BETR it maps to an internal 32-bit integer seed. So using /BETR reduces the chance that two values of *seed* will map to the same internal integer seed.

See Also

The **enoise** and **gnoise** functions. **Noise Functions** on page III-332.

SetScale

SetScale [**/I/P**] ***dim, num1, num2*** [, ***unitsStr***] ,***waveName*** [, ***waveName***]…
SetScale ***d, num1, num2*** [, ***unitsStr***], ***waveName*** [, ***waveName***]…

The SetScale operation sets the dimension scaling or the data full scale for the named waves.

Parameters

The first parameter *dim* must be one of the following:

Character	Signifies
d	Data full scale.
t	Scaling of the chunks dimension (t scaling).
x	Scaling of the rows dimension (x scaling).
y	Scaling of the columns dimension (y scaling).
z	Scaling of the layers dimension (z scaling).

If setting the scaling of any dimension (*x, y, z,* or *t*), *num1* is the starting index value — the scaled index for the first point in the dimension. The meaning of *num2* changes depending on the /I and /P flags. If you use /P, then *num2* is the delta value — the difference in the scaled index from one point to the next. If you use /I, *num2* is the "ending value" — the index value for the last element in the dimension. If you use neither flag, *num2* is the "right value" — the index value that the element *after the last element in the dimension* would have.

These three methods are just three different ways to specify the two scaling values, the starting value and the delta value, that are stored for each dimension of each wave.

If setting the data full scale (*d*), then *num1* is the nominal minimum and *num2* is the nominal maximum data value for the waves. The data full scale values are not used. They serve only to document the minimum and maximum values the waves are expected to attain. No flags are used when setting the data full scale.

The *unitsStr* parameter is a string that identifies the natural units for the x, y, z, t, or data values of the named waves. Igor will use this to automatically label graph axes. This string must be one to 49 characters such as "m" for meters, "g" for grams or "s" for seconds. If the waves have no natural units you can pass "" for this parameter.

Setting *unitsStr* to "dat" (case-sensitive) tells Igor that the wave is a date/time wave containing data in Igor date/time format (seconds since midnight on January 1, 1904). Date/time waves must be double-precision.

Flags

At most one flag is allowed, and then only if dimension scaling (not data full scale) is being set:

/I Inclusive scaling. *num2* is the ending index — the index value for the very last element in the dimension.

/P Per-point scaling. *num2* is the delta index value — the difference in scaled index value from one element to the next.

Details

SetScale will not allow the delta scaling value to be zero. If you execute a SetScale command with a delta value of zero, it will set the delta value to 1.0.

If you do not use the /P flag, SetScale converts *num1* and *num2* into a starting index value and a delta index value. If you call SetScale on a dimension with fewer than two elements, it does this conversion as if the dimension had two elements.

Prior to Igor Pro 3.0, Igor supported only 1D waves. "SetScale x" was used to set the scaling for the rows dimensions and "SetScale y" was used to set the data full scale. With the addition of multidimensional waves, "SetScale y" is now used to set the scaling of the columns dimension and "SetScale d" is used to set the data full scale. For backward compatibility, "SetScale y" on a 1D wave sets the data full scale.

When setting the dimension scaling of a numeric wave, you can omit the *unitsStr* parameter. Igor will set the wave's scaling but not change its units. However, when setting the dimension scaling of a text wave, you must supply a *unitsStr* parameter (use "" if the wave has no units). If you don't, Igor will think that the text wave is the start of a string expression and will attempt to treat it as the *unitsStr*.

Note: Prior to Igor Pro 3.0, SetScale with no wave names would set the default characteristics for waves made subsequently with the Make command. In Igor Pro 2.0, this behavior was considered obsolete and the manual warned that it was not recommended. As of Igor Pro 3.0 SetScale will still set the default x scaling, but will beep and put up a warning dialog. In future versions, SetScale without a wave name will not work. At present, you cannot set the default y, z, or t scaling with SetScale.

See Also

For an explanation of waves and dimension scaling, see **Changing Dimension and Data Scaling** on page II-83.

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-102.

SetVariable

SetVariable [**/Z**] ***ctrlName*** [***keyword* = *value*** [**,** ***keyword* = *value*** **…**]]

The SetVariable operation creates or modifies a SetVariable control in the target window.

A SetVariable control sets the value of a global numeric or string variable or a point in a wave when you type or click in the control. As of Igor Pro 6.1, a SetVariable can also hold its own value without the need for a global or wave.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the SetVariable control to be created or changed.

The following keyword=value parameters are supported:

activate
: Activates the control and selects the text that sets the value. Use **ControlUpdate** to deactivate the control and deselect the text.

appearance={*kind* [, *platform*]}
: Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **Button** and **DefaultGUIControls** for more appearance details.

bodyWidth=*width*
: Specifies an explicit size for the body (nontitle) portion of a SetVariable control. By default (bodyWidth=0), the body portion is the amount left over from the specified control width after providing space for the current text of the title portion. If the font, font size or text of the title changes, then the body portion may grow or shrink. If you supply a bodyWidth>0, then the body is fixed at the size you specify regardless of the body text. This makes it easier to keep a set of controls right aligned when experiments are transferred between Macintosh and Windows, or when the default font is changed.

disable=*d*
: Sets user editability of the control.

d=0: Normal.

d=1: Hide.

d=2: No user input.

fColor=(*r*,*g*,*b*)
: Sets the initial color of the title. *r*, *g*, and *b* range from 0 to 65535. fColor defaults to black (0,0,0). To further change the color of the title text, use escape sequences as described for title=*titleStr*.

font="*fontName*"
: Sets the font used to display the value of the variable, e.g., `font="Helvetica"`.

format=*formatStr*
: Sets the numeric format of the displayed value, e.g., format="%g". Not used with string variables. Never use leading text or the "%W" formats, because Igor reads the value back without interpreting the units. For a description of *formatStr*, see the **printf** operation.

frame=*f*
: *f*=0: Value unframed.
: *f*=1: Value framed (default).

fsize=*s*
: Sets the size of the type used to display the variable's value.

fstyle=*fs*
: *fs* is a binary coded number with each bit controlling one aspect of the font style as follows:
: bit 0: Bold.
: bit 1: Italic.
: bit 2: Underline.
: bit 3: Outline (*Macintosh only*).
: bit 4: Shadow (*Macintosh only*).
: See **Setting Bit Parameters** on page IV-12 for details about bit settings.

help={*helpStr*}
: Sets the help for the control. The help text is limited to a total of 255 characters. On Macintosh, the help appears if you turn Igor Tips on. On Windows, the help for the first 127 characters or up to the first line break appears in the status line. If you press F1 while the cursor is over the control, you will see the entire help text. You can insert a line break by putting "\r" in a quoted string.

labelBack=(*r*,*g*,*b*) or 0
: Specifies the background fill color for labels. *r*, *g*, and *b* are integers from 0 to 65535. The default is 0, which uses the window's background color.

limits={*low*,*high*,*inc*}
: Sets the limits of the allowable values (*low* and *high*) for the variable. *inc* sets the amount by which the variable is incremented if you click the control's up/down arrows. This applies to numeric variables, not to string variables. If *inc* is zero then the up/down arrows will not be drawn.

live=*l*
: *l* =0: Update only after variable changes (default).
: *l* =1: Update as variable changes.

noedit=*val*
: noedit=1 prevents the user from clicking (or tabbing into) a SetVariable control to directly edit its value. This is useful when you want to make a string read-only or when you want to restrict a numeric setting to those available only via the control's up or down arrow buttons.
: noedit=0 reactivates user editing.
: noedit=2 allows the use of fancy text using the escape codes defined for the **TextBox** operation. See **Annotation Escape Codes** on page V-691.

noproc
: No procedure is to execute when the control's value is changed.

pos={*left*,*top*}
: Sets the position of the control in pixels.

pos+={*dx*,*dy*}
: Offsets the position of the control in pixels.

proc=*procName*
: Sets the procedure to execute when the control's value is changed.

rename=*newName*
: Gives control a new name.

size={*width*,*height*}
: Sets width of control in pixels. *height* is ignored.

title=*titleStr*
: Sets title of control to the specified string expression. The title is displayed to the left of the control. If *titleStr* is empty (""), the name of the controlled variable is displayed as the title. Use `title=" "` (put a space within the quotation marks) to create a "blank" title.
: *titleStr* can contain formatting escape codes in order to create fancy, styled results. The escape codes are the same as used by the **TextBox** operation. The easiest way to generate fancy text is to make selections from the Insert popup in the SetVariable Control dialog.

userdata(*UDName*)=*UDStr*
: Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*
: Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

value=*varOrWaveName*
: Sets the numeric or string variable or wave element to be controlled.

  If *varOrWaveName* references a wave, the point is specified using standard bracket notation with either a numeric point number or a row label, for example: `value=awave[4]` or `value=awave[%alabel]`.

  You may also use a matrix wave and specify a column index in addition to the row index.

  As of Igor Pro 6.1, you can have the control store the value internally rather than in a global variable. In place of varName, use _STR:str or _NUM:num. For example:

```
NewPanel;SetVariable sv1,value=_NUM:123
```

valueColor=(*r*,*g*,*b*)
: Sets the color of the value text. *r*, *g*, and *b* range from 0 to 65535. valueColor defaults to black (0,0,0).

valueBackColor=(*r*,*g*,*b*)
: Sets the background color under the value text. *r*, *g*, and *b* range from 0 to 65535.

valueBackColor=0
: Sets the background color under the value text to the default color, the standard document background color used on the current operating system, which is usually white.

win=*winName*
: Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

  When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

The target window must be a graph or panel.

The procedure, which may be a function *or* a macro, has the format:

```
Function procName(ctrlName,varNum,varStr,varName) : SetVariableControl
   String ctrlName
   Variable varNum     // value of variable as number
   String varStr       // value of variable as string
   String varName      // name of variable
   …
End
```

The "`: SetVariableControl`" designation tells Igor to include this procedure in the Procedure pop-up menu in the SetVariable Control dialog.

As of Igor Pro 6.1, you can use ctrl-return to enter a carriage return in a string SetVariable. A carriage return in a string SetVariable appears as a symbol representing a carriage return.

As of Igor Pro 6.1, if you set noedit to 2, you can use fancy text using the escape codes defined for the **TextBox** operation. See **Annotation Escape Codes** on page V-691.

The action procedure for a SetVariable control can also use a predefined structure `WMSetVariableAction` as a parameter to the function. The control will use this more efficient method when the function properly matches the structure prototype for a SetVariable control, otherwise it will use the old-style method.

A SetVariable action procedure using a structure has the format:

```
Function newActionProcName(SV_Struct) : SetVariableControl
   STRUCT WMSetVariableAction &SV_Struct
   …
End
```

For a SetVariable control, the WMSetVariableAction structure has members as described in the following table:

WMSetVariableAction Structure Members

Member	Description
char ctrlName[MAX_OBJ_NAME+1]	Control name.
char win[MAX_WIN_PATH+1]	Host (sub)window.
STRUCT Rect winRect	Local coordinates of host window.
STRUCT Rect ctrlRect	Enclosing rectangle of the control.
STRUCT Point mouseLoc	Mouse location.
Int32 eventCode	Event that executed the procedure. -1: Control being killed 1: Mouse up 2: Entery key 3: Live update 4: Mouse scroll wheel up 5: Mouse scroll wheel down 6: Value changed by dependency update Code -1 is never sent to an old-style (non-structure parameter) action procedure. Codes 4 and 5 (Igor Pro 6.1 or later) are sent only for string SetVariables or numeric SetVariables whose increment setting is zero. For numeric SetVariables whose increment is non-zero, the mouse scroll wheel acts like a mouse click on the increment or decrement arrows. Code 6 (Igor Pro 6.1 or later) is by default sent to only structure-based action procedures. Use SetIgorOption EnableSVE6=0 to disable sending this event at all, =2 to send the event to both structure-based and old-style SetVariable action procedures. (The default is =1).
Int32 eventMod	Bitfield of modifiers. See **Control Structure eventMod Field** on page III-385.
String userData	Primary (unnamed) user data. If this changes, it is written back automatically.
Int32 blockReentry	Prevents reentry of control action procedure. See **Control Structure blockReentry Field** on page III-386.
Int32 isStr	TRUE for a string variable.
double dval	Numeric value of variable.
char sval[MAXCMDLEN]	Value of variable as a string.
char vName[MAX_OBJ_NAME+2+MAX_OBJ_NAME+4+1]	Name of variable or wave. Dimension labels can be used for waves.
WAVE svWave	Valid if using wave.

WMSetVariableAction Structure Members

Member	Description
`Int32 rowIndex`	Row index for a wave, if `rowLabel` is empty.
`char rowLabel[MAX_OBJ_NAME+1]`	Wave row label.
`nt32 colIndex`	Column index for a wave, if `colLabel` is empty.
`char colLabel[MAX_OBJ_NAME+1]`	Wave column label.

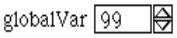
Action functions should respond only to documented `eventCode` values. Other event codes may be added along with more fields. Although the return value is not currently used, action functions should always return zero.

The constants used to specify the size of structure `char` arrays are internal to Igor Pro and may change.

Examples

Executing the commands:

```
Variable/G globalVar=99
SetVariable setvar0 size={120,20}
SetVariable setvar0 font="Helvetica", value=globalVar
```

creates this SetVariable control: 

See Also

The **printf** operation for an explanation of *formatStr*, and **Set Variable** on page III-362.

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

SetVariableControl

`SetVariableControl`

SetVariableControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined SetVariable control. See **Procedure Subtypes** on page IV-179 for details. See **SetVariable** for details on creating a SetVariable control.

SetWaveLock

`SetWaveLock` *`lockVal, waveList`*

The SetWaveLock operation locks a wave or waves and protects them from modification. Such protection is not absolute, but it should prevent most common attempts to change or kill a wave.

Parameters

lockVal can be 0, to unlock, or 1, to lock the wave(s).

waveList is a list of waves or it can be `allinCDF` to act on all waves in the current data folder.

See Also

WaveInfo to check if a wave is locked.

SetWindow

`SetWindow` *`winName`* `[,` *`keyword = value`*`]…`

The SetWindow operation sets the window note and user data for the named window or subwindow. SetWindow can also set hook functions for a base window or exterior subwindow (interior subwindows not supported).

Parameters

winName can be a window or subwindow name. It can also be the keyword `kwTopWin` to specify the topmost graph, panel, layout, table, or notebook window.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

hide =*h* Hides or unhides widows or subwindows.

h=0: Unhides a subwindow or base window

h=1: Hides a subwindow or base window.

h=2: Unhides without restoring minimized windows (*Windows* only).

When unhiding subwindows, you should combine with needUpdate=1 if conditions require the subwindow to be redrawn since the window was hidden.

hook=*procName* Sets the window hook function that Igor will call when certain events happen. Use `SetWindow hook=$""` to specify no hook function.

See **Unnamed Window Hook Functions** on page IV-271 for further details.

hook(*hName*)=*procName* Defines a named window hook *hName* and sets the function that Igor will call when certain events happen. *hName* can be any legal name. Named hooks are called before any unnamed hooks.

Use `$""` for *procName* to specify no hook.

See **Named Window Hook Functions** on page IV-265 for further details.

To hook a subwindow, see **Window Hooks and Subwindows** on page IV-265.

hookcursor=*number* Uses the cursor specified by hookcursor in the interior of the window if hook is specified, if hookevents bit 1 is set, and if hookcursor is nonzero. Currently, there are 27 cursors.

hookevents=*flags* Bitfield of flags to enable certain events for the unnamed hook function:

bit 0: Mouse button clicks.

bit 1: Mouse moved events.

bit 2: Cursor moved events.

To set bit 0 and bit 1 (mouse clicks and mouse moved), use $2^0+2^1 = 1+2 = 3$ for *flags*. Use 7 to also enable cursor moved events. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

This keyword applies to the unnamed hook function only. It does not affect named hook functions which always receive all events.

markerHook= {*hookFuncName*, *start*, *end*}

Specifies a user function and marker number range for custom markers. The marker range can be any positive integers less than 1000 and can overlap built-in marker numbers. See **Custom Marker Hook Functions** on page IV-274 for details.

Use $"" for hookFuncName to specify no hook.

needUpdate= *n* Marks a window as needing an update (*n*=1) or takes no action (*n*=0).

note=*noteStr* Sets the window note to *noteStr*, replacing any existing note.

note+=*noteStr* Appends *noteStr* to current contents of the window note.

userdata=*UDStr*

userdata(*UDName*)=*UDStr*

Sets the window or subwindow user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata+=*UDStr*

userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current window or subwindow user data. Use the optional (*UDName*) to append to the named user data.

Details

For details on named window hooks, see **Window Hook Functions** on page IV-264.

Unnamed window hook functions are supported for backward compatibility only. New code should use named window hook functions. For details on unnamed window hooks, see **Unnamed Window Hook Functions** on page IV-271.

For details on marker hooks, see **Custom Marker Hook Functions** on page IV-274.

See Also

The **GetWindow, SetIgorHook**, and **SetIgorMenuMode** operations and **AxisValFromPixel**, **NumberByKey**, **PopupContextualMenu**, and **TraceFromPixel** functions. The **GetUserData** operation for retrieving named user data.

ShowIgorMenus

ShowIgorMenus [*MenuNameStr* [, *MenuNameStr*] …

The ShowIgorMenus operation shows the named built-in menus or, if none are explicitly named, shows all built-in menus in the menu bar.

User-defined menus attached to built-in menus are also affected by this operation.

Parameters

MenuNameStr The name of an Igor menu, like "File", "Data", or "Graph".

Details

See **HideIgorMenus** for details.

See Also

Chapter IV-5, **User-Defined Menus**.

The **HideIgorMenus**, **DoIgorMenu**, and **SetIgorMenuMode** operations.

ShowInfo

ShowInfo [**/CP=***num* **/W=***winName*]

The ShowInfo operation puts an information box on the target or named graph. The information box contains cursors and readouts of values associated with waves in the graph.

Flags

/CP=*num* Selects a cursor pair to display in the info panel.
num=0: Selects cursor A and cursor B.
num=1: Selects cursor C and cursor D.
num=2: Selects cursor E and cursor F.
num=3: Selects cursor G and cursor H.
num=4: Selects cursor I and cursor J.

/W=*winName* Displays info box in the named window.

See Also

Info Box and Cursors on page II-286.

The **HideInfo** operation.

ShowTools

ShowTools [**/A/W=***winName*][*toolName*]

The ShowTools operation puts a tool palette for drawing along the left hand side of the target or named graph or control panel, and optionally activates the named tool.

Flags

/A Sizes window automatically to make extra room for the tool palette. This preserves the proportion and size of the actual graph area.

/W=*winName* Shows tool palette in the named window. This must be the first flag specified when used in a Proc or Macro or on the command line.

Parameters

If you specify a *toolName* (which can be one of: normal, arrow, text, line, rect, rrect, oval, or poly) the named tool is activated. Specifying the "normal" tool has the same effect as issuing the GraphNormal command for a graph that has the drawing tools selected.

Details

The activated tool is not highlighted until the top graph or control panel becomes the topmost (activated) window. Use `DoWindow/F` to bring a window to the top (or "front").

See Also

The **DoWindow**, **GraphNormal**, **GraphWaveDraw**, **GraphWaveEdit**, and **HideTools** operations.

sign

sign(*num*)

The sign function returns -1 if *num* is negative or 1 if it is not negative.

Silent

Silent *num*

The Silent operation enables or disables the display of macro commands in the command line as they are executed.

Parameters

If *num* is one, the display of macro commands is disabled. If *num* is zero, it is enabled.

When executed from the command line, "Silent 100" activates a compatibility mode that can run legacy macro code containing old-style comments, which use | instead of //, without causing errors. To exit this mode, execute "Silent 101" on the command line. Also see **Comments** on page IV-2.

If you create procedure files for use by others and you want to use the new logic operations such as || that require that Silent 101 be in effect, then you can specify

```
#pragma rtGlobals=2
```

in place of the normal `rtGlobals=1`.

If your procedure file is included in an experiment running in compatibility mode (Silent 100) then an alert dialog will allow the user to switch the experiment to the new syntax. However, keep in mind that when the procedures are recompiled in the new mode, the user's other procedures may not compile due to obsolete use of the vertical bar (|) as a comment character.

Details

Macros run faster if this display is disabled. If Silent is used in a macro its effect ends when the macro ends.

This has no effect in user-defined functions. During execution of a user-defined function, Igor does not display anything in the command line.

Commands sent to Igor Pro from another program via PPC or Apple events are normally logged into the history area. You can turn off the logging of these commands with the Silent operation. See also **The Silent Option** on page IV-102.

See Also

The **PauseUpdate** operation and **rtGlobals**.

sin

sin(*angle*)

The sin function returns the sine of *angle* which is in radians.

In complex expressions, *angle* is complex, and sin(*angle*) returns a complex value:

$$\sin(x+iy) = \sin(x)\cosh(y) + i\cos(x)\sinh(y).$$

See Also

cos, **tan**, **sec**, **csc**, **cot**

sinc

sinc(*num*)

The sinc function returns sin(*num*)/*num*. The sinc function returns 1.0 when *num* is zero. *num* must be real.

sinh

sinh(*num*)

The sinh function returns the hyperbolic sine of *num*:

$$\sinh(x) = \frac{e^{x} - e^{-x}}{2}.$$

In complex expressions, *num* is complex, and sinh(*num*) returns a complex value.

See Also
cosh, **tanh**, **coth**

Sleep

Sleep [***flags***] ***timeSpec***

The Sleep operation puts Igor to sleep for a while. After the while is up, Igor continues execution. You could use this, for example, to print a graph late at night, when the load on your printer is light.

Parameters

The format of *timeSpec* depends on which flags, if any, are present.

If no flags are present, then *timeSpec* is in *hh*:*mm*:*ss* format and specifies the number of elapsed hours, minutes and seconds to sleep.

Flags

/A	*timeSpec* is an absolute time in 24 hour format (e.g., 16:00:00).
/A/W	Wait until tomorrow if absolute time has passed.
/B	Stop sleeping if the user clicks the mouse button.
/C=*cursor*	Controls what kind of cursor to display during sleep. *cursor*=-1: No cursor change. *cursor*=0: Hour glass (default). *cursor*=1: Arrow. *cursor*=2: "Click". *cursor*=<any other value>: watch.
/Q	Continue executing the procedure containing the Sleep operation even if Command-period (*Macintosh*) or Ctrl+Break (*Windows*) was pressed.
/S	*timeSpec* is a numeric expression in seconds.
/T	*timeSpec* is a numeric expression in ticks (about 1/60 of a second).

Details

The Sleep operation does *not* let the user choose menus, move cursors, run procedures, draw in graphs, or do any other interactive task.

Normally *timeSpec* specifies an amount of elapsed time. If the /A flag is present, then *timeSpec* is an absolute time when sleep is to end. If the specified absolute time has already passed, no sleep occurs unless you also use /W, which makes it wait until tomorrow.

If you specify time in hh:mm:ss format, you can also specify the time indirectly through a string variable. See the examples.

You can end sleep by pressing Command-period (*Macintosh*) or Ctrl+Break (*Windows*). Normally when you do this, it aborts any procedure that is running. However, if you use the /Q flag, the procedure continues running normally.

Examples

These examples assume the current time is 4 pm:

```
Sleep 00:01:30              // sleeps for 1 minute, 30 seconds
Sleep/A 23:30:00            // sleeps until 11:30 PM
Sleep/A 03:00:00            // doesn't sleep at all because time is past
Sleep/A/W 03:00:00          // sleeps until 3 AM tomorrow
String str1= "03:00:00"     // put wakeup call time in string
Sleep/A/W $str1             // sleeps until 3 AM tomorrow
Sleep/B/C=2/S/Q 60          // sleep 60 seconds, or until user clicks,
                            //  and keep going (don't abort)
```

Slider

Slider [**/Z**] ***controlName*** [***key*** [**= *value***]][**, *key*** [**= *value***]]…

The Slider operation creates or modifies a Slider control in the target window.

A Slider control sets or displays a single numeric value. The user can adjust the value by dragging a thumb along the length of the Slider.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the Slider control to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}
: Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

  kind can be one of `default`, `native`, or `os9`.

  platform can be one of `Mac`, `Win`, or `All`.

  Note: The Slider control reverts to `os9` appearance on Macintosh if thumbColor isn't the default blue (0,0,65535).

  See **Button** and **DefaultGUIControls** for more appearance details.

disable=*d*
: Sets user editability of the control.

  d=0: Normal.

  d=1: Hide.

  d=2: Draw in gray state; disable control action.

fColor=(*r*,*g*,*b*)
: Sets the color of the tick marks. *r*, *g*, and *b* range from 0 to 65535. fColor defaults to black (0,0,0).

font="*fontName* "
: Sets the font used to display the tick labels, e.g., `font="Helvetica"`.

fsize=*s*
: Sets the size of the type for tick mark labels.

help={*helpStr*}
: Sets the help for the control. The help text is limited to a total of 255 characters. On Macintosh, the help appears if you turn Igor Tips on. On Windows, the help for the first 127 characters or up to the first line break appears in the status line. If you press F1 while the cursor is over the control, you will see the entire help text. You can insert a line break by putting "\r" in a quoted string.

limits= {*low*,*high*,*inc*}
: *low* sets left or bottom value, *high* sets right or top value. Use *inc*=0 for continuous or use desired increment between stops.

live=*l*
: *l* =0: Update only after mouse is released.

  l =1: Update as slider moves (default).

noproc
: Specifies that no procedure is to execute when the control's value is changed.

pos={*left*,*top*}
: Sets the position of the slider in pixels.

pos+={*dx*,*dy*}
: Offsets the position of the slider in pixels.

proc=*procName*
: Specifies the procedure to execute when the control's thumb is moved by the user.

rename=*newName*
: Gives control a new name.

side=*s*
: *s* =0: Thumb is blunt.

  s =1: Thumb points right or down (default).

  s =2: Thumb points up or left.

size={*width*,*height*}
: Sets width or height of control in pixels. *height* is ignored if vert=0 and *width* is ignored if vert=1.

thumbColor=(*r*,*g*,*b*)
: Sets dominant foreground color of thumb. *r*, *g*, and *b* are integers from 0 to 65535. Only the hue and saturation are used. Therefore (0,1000,0) is the same tint of green as (0,10000,0).

ticks=*t*
: *t* =0: No ticks.

  t =1: Number of ticks is calculated from limits (no ticks drawn if calculated value is less than 2 or greater than 100). Default value.

t >1: *t* is the number of ticks distributed between the start and stop position. Ticks are labeled using the same automatic algorithm used for graph axes. Use negative tick values to force ticks to not be labeled. Ticks are shown on the side specified by the side keyword and are not drawn if side=0.

tkLblRot= *deg* — Rotates tick labels. *deg* is a value between -360 and 360. Prior to Igor Pro 6.1, rotation was supported in multiples of 90 degrees only.

userdata(*UDName*)=*UDStr*
Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*
Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

userTicks={*tvWave*,*tlblWave*}
User-defined tick positions and labels. *tvWave* contains the tick positions, and text wave *tlblWave* contains the labels. See **ModifyGraph (axes) userticks** for more info. Overrides normal ticking specified by ticks keyword.

value=*v* — *v* is the new value for the Slider.

valueColor=(*r*,*g*,*b*) — Sets the color of the tick labels. *r*, *g*, and *b* range from 0 to 65535. valueColor defaults to black (0,0,0).

variable= *var* — Sets the variable (*var*) that the slider will update. It is not necessary to connect a Slider to a variable — you can get a Slider's value using the **ControlInfo** operation.

vert=*v* — Set vertical (*v* =1; default) or horizontal (*v* =0) orientation of the slider.

win=*winName* — Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

The target window must be a graph or panel.

The format of the action procedure (set by the proc keyword) that will be called as the user drags the Slider's thumb is:

```
Function MySliderProc(name, value, event) : SliderControl
   String name             // name of this slider control
   Variable value          // value of slider
   Variable event          // bit field:bit 0:value set; 1:mouse down,
                           //  2:mouse up, 3:mouse moved
   return 0                // other return values reserved
End
```

The "`: SliderControl`" designation tells Igor to include this procedure in the Procedure pop-up menu in the Slider Control dialog.

If you use negative ticks to suppress automatic labeling, you can label tick marks using drawing tools (panels only).

The action procedure for a Slider control can also use a predefined structure `WMSliderAction` as a parameter to the function. The control will use this more efficient method when the function properly matches the structure prototype for a Slider control, otherwise it will use the old-style method.

A Slider action procedure using a structure has the format:

```
Function newActionProcName(S_Struct) : SliderControl
   STRUCT WMSliderAction &S_Struct
   …
End
```

For a Slider control, the WMSliderAction structure has members as described in the following table:

WMSliderAction Structure Members

Member	Description
char ctrlName[MAX_OBJ_NAME+1]	Control name.
char win[MAX_WIN_PATH+1]	Host (sub)window.
STRUCT Rect winRect	Local coordinates of host window.
STRUCT Rect ctrlRect	Enclosing rectangle of the control.
STRUCT Point mouseLoc	Mouse location.
Int32 eventCode	Event that caused the procedure to execute. Sets bit fields: bit 0: value set. bit 1: mouse down. bit 2: mouse up. bit 3: mouse moved. When the control is about to be killed, eventCode is set to -1.
Int32 eventMod	Bitfield of modifiers. See **Control Structure eventMod Field** on page III-385.
String userData	Primary (unnamed) user data. If this changes, it is written back automatically.
Int32 blockReentry	Prevents reentry of control action procedure. See **Control Structure blockReentry Field** on page III-386.
double curval	Value of slider.

Action functions should respond only to documented eventCode values. Other event codes may be added along with more fields. Although the return value is not currently used, action functions should always return zero.

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

Examples

```
Function SliderExample()
   NewPanel /W=(150,50,501,285)
   Variable/G var1
   Execute "ModifyPanel cbRGB=(56797,56797,56797)"
   SetVariable setvar0,pos={141,18},size={122,17},limits={-Inf,Inf,1},value=var1
   Slider foo,pos={26,31},size={62,143},limits={-5,10,1},variable=var1
   Slider foo2,pos={173,161},size={150,53}
   Slider foo2,limits={-5,10,1},variable=var1,vert=0,thumbColor=(0,1000,0)
   Slider foo3,pos={80,31},size={62,143}
   Slider foo3,limits={-5,10,1},variable=var1,side=2,thumbColor=(1000,1000,0)
   Slider foo4,pos={173,59},size={150,13}
   Slider foo4,limits={-5,10,1},variable=var1,side=0,vert=0
   Slider foo4,thumbColor=(1000,1000,1000)
   Slider foo5,pos={173,90},size={150,53}
   Slider foo5,limits={-5,10,1},variable= var1,side=2,vert=0
   Slider foo5,ticks=5,thumbColor=(500,1000,1000)
End
```

See Also

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

Slow

Slow *ticks*

The Slow operation slows down execution of macros for debugging purposes.

Parameters

ticks is in units of 1/60 second. A value of 30 is about right to allow you to read the macro command lines as they are executed. `Slow 0` resumes normal macro execution.

Details

This has no effect in user-defined functions.

Smooth

Smooth [***flags***] ***num, waveName*** [**, *waveName*…**]**

The Smooth operation smooths the named waves using binomial (Gaussian) smoothing, boxcar (sliding average) smoothing, Savitzky-Golay (polynomial) smoothing, or running-median filtering.

Parameters

num is the number of smoothing operations to be applied for binomial smoothing or the number of points in the smoothing window for boxcar, Savitzky-Golay, and running-median smoothing.

Each *waveName* is smoothed in-place, overwriting the values with the smoothed result. *waveName* may be a floating point or integer wave.

If *waveName* contains NaNs, the results are undefined. (The **Loess** operation and the Interpolate XOP can fill in NaNs).

Flags

/B [=*b*] Invokes boxcar smoothing algorithm. If given, *b* specifies the number of passes to use when smoothing the data with smoothing factor *num* (box width). The number of passes can be any value between 1 and 32767.

/DIM=*d* Specifies the wave dimension to smooth.

d=-1: Treats entire wave as 1D (default).

For *d*=0, 1,…, operates along rows, columns, etc.

/E=*endEffect* Determines how to handle the ends of the wave (w) when fabricating missing neighbor values.

endEffect=0: Bounce method (default). Uses w[*i*] in place of the missing w[-*i*] and w[*n*-*i*] in place of the missing w[*n*+*i*].

endEffect=1: Wrap method. Uses w[*n*-*i*] in place of the missing w[-*i*] and vice versa.

endEffect=2: Zero method. Uses 0 for any missing value.

endEffect=3: Repeat method. Uses w[0] in place of the missing w[-*i*] and w[*n*] in place of the missing w[*n*+*i*].

/EVEN [=*evenAllowed*]

Specifies the smoothing increment for boxcar smoothing (/B). Values are:

0: Increments even values of *num* to the next odd value. Default when /EVEN omitted.

1: Uses even values of *num* for boxcar smoothing despite the half-sample shifting this introduces in the smoothed output (prior to version 6, this shift was prevented). Same as/EVEN alone.

/F [=*f*] Selects the boxcar or multipass binomial smoothing method:

f=0: Slow, but accurate, method (default).

f=1: Fast method. Same as /F alone.

/M=*threshold* Invokes running-median smoothing and specifies an absolute numeric threshold used to optionally replace “outliers”. Points that differ from the central median by an amount exceeding *threshold* are replaced, either with the *replacement* value specified by /R, or otherwise with the median value. Special *threshold* values are:

0: Replace all values with running-median values or the *replacement* value.

(NaN): Replace only NaN input values with running-median values or the *replacement* value.

The smoothing factor *num* is the number of points in the smoothing window used to compute each median.

/MPCT=*percentile* Used with /M to compute a smoothed value that is a different percentile than the median. /M must be present if /MPCT is used.

percentile is a value from 0 to 100.

Roughly speaking, the smoothed value returned is the smallest value in the smoothing window that is greater than the smallest *percentile* % of the values. See "Median and Percentile Smoothing Details", below.

percentile=0: The smoothed value is the minimum value in the smoothing window.

percentile=50: The smoothed value is the median of the values in the smoothing window. This is the default if /MPCT is omitted.

percentile=100: The smoothed value is the maximum value in the smoothing window.

/R=*replacement* Specifies the value that replaces input values that exceed the central median by *threshold* (requires /M). *threshold* can be any value (including NaN or ±Inf if *waveName* is floating point).

/S=*sgOrder* Invokes Savitzky-Golay smoothing algorithm and specifies the smoothing order. *sgOrder* must be either 2 or 4.

Details

When num is 1, no smoothing is done.

Binomial Smoothing Details

For binomial smoothing, use no flags (other than /DIM, /E, and /F) and a *num* value from 1 to 32767.

The binomial smooth algorithm automatically switches to a nearly equivalent (but *much* faster) multipass box smooth at smooth factor of 50. The original algorithm can be used when you set this global variable:

```
Variable/G root:V_doOrigBinomSmooth=1
```

To get the pre-Igor Pro 6 behavior you also need to add the /F flag.

The /F (fast boxcar smoothing) algorithm creates small errors when the data has a large offset. For some data sets you may want to subtract the mean of the data before smoothing and add it back in afterwards.

The binomial smoothing algorithm does not detect and ignore NaNs in the input data.

Boxcar Smoothing Details

For boxcar smoothing, use the /B flag and a *num* value from 1 to 32767.

For *num* < 2, no smoothing is done.

If *num* is even and /EVEN is not specified, *num* is incremented to the next (odd) integer.

If *num* is even and /EVEN is specified, each smoothed output is formed from one more previous value than future values.

The boxcar smoothing algorithm detects and ignores NaNs in the input data. If *num* is less than the number of NaNs near the output point, then the result is NaN. Otherwise the average of the non-NaN neighboring points is used to compute the smoothed result.

Savitzky-Golay Smoothing Details

For Savitzky-Golay smoothing, use the /S flag and an odd *num* value from 5 to 25. An even value for *num* returns an error. If *sgOrder*=4, then *num*= 5 gives no smoothing at all so *num* should be at least 7.

The Savitzky-Golay smoothing algorithm does not detect and ignore NaNs in the input data.

Median and Percentile Smoothing Details

For running-median smoothing, use the /M flag and a *num* value from 1 to 32767.

If *num* is even, the median is the average of the two middle values.

For example, the median of 6 values around data[i] is the median of data[i-3], data[i-2], data[i-1], data[i], data[i+1], and data[i+2], and if these values were already sorted, the median would be the average of data[i-1] and data[i].

Use /M=0 to replace all values with the median over the smoothing window or use /M=*threshold*/R=(NaN) to replace outliers with NaNs.

Use /M=(NaN) to replace only NaN input values with the running-median values or the *replacement* value.

The running-median smoothing algorithm detects and ignores NaNs in the input data. If *num* is less than the number of NaNs near the output point, then the result is NaN. Otherwise the median of the non-NaN neighboring points is used to compute the smoothed result.

The running-median is a special case of running-percentile, with *percentile*=50.

The /M and /MPCT algorithm uses an interpolated rank to compute the value of percentiles other than 0 and 100.

Using Example 1 from <http://cnx.org/content/m10805/latest/> ("A Third Definition"), the 25th percentile (/MPCT=25) of the 8 values:

```
Make/O sortedData={3,5,7,8,9,11,13,15}// Already sorted, rank 1 to 8
```

The first step is to compute the rank (R) of the 25th percentile. This is done using the following formula: R=(*percentile*/100)*(*num*+1), where *percentile* is 25 and *num* is 8, so here R = 2.25.

If R were an integer, the Pth percentile would be the number with rank R; if R were 2 the result would be the 2nd value = 5.

Since R is not an integer, we compute the Pth percentile by interpolation as follows:

1. Define IR as the integer portion of R (the number to the left of the decimal point). For this example, IR=2.
2. Define FR as the fractional portion of R. For this example, FR=0.25
3. Find the values with Rank IR and with Rank IR+1. For this example, this means the values with Rank 2 and the score with Rank 3. The values are 5 and 7.
4. Interpolate by multiplying the difference between the values by FR and add the result to the lower values. For these data, this is 0.25(7-5)+5=5.5

Therefore, the 25th percentile is 5.5:

```
Smooth/M=0/MPCT=(percentile) 8, sortedData  // 8-point smoothing window
Print sortedData[3] // prints 5.5, the 25th percentile of all 8 values
```

Smoothing Window and End Effects Details

These smoothing algorithms compute the output value for a given point using each point's neighbors. Except for running-median smoothing, each algorithm combines neighboring points before and after the point being smoothed. At the start or end of a wave some points will not have enough neighbors so some method for fabricating neighbor values must be implemented. The /E flag specifies the method.

The running-median filter, however, ignores /E. At each end of the data fewer values are included in the median calculation, so that values "beyond" the end of data are not needed.

The first output value is the median of wave[0, floor((*num*-1)/2)]. For example, if *num* = 7, then the first output value is the median of wave[0], wave[1], wave[2], and wave[3]. Because that is an even number of points, the median is the average of the two middle values. Continuing the example, if the values were 3, 1, 7, and 5, the two middle values are 3 and 5. The computed median would be (3+5)/2=4.

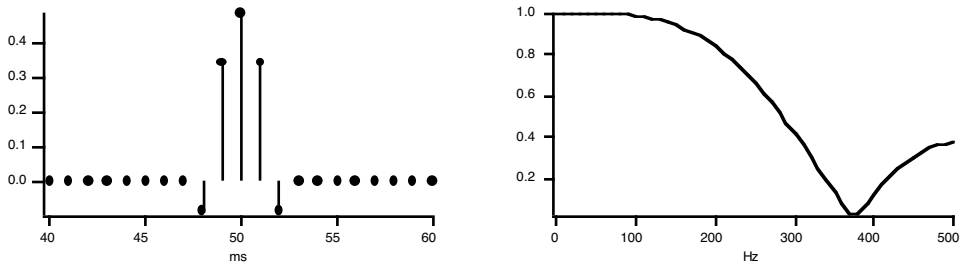
Examples

Box smoothing example:

```
Make/N=100 wv; Display wv
wv=gnoise(1)
Smooth/B/E=3 3,wv   // output[p] = average of wv[p-1], wv[p] and wv[p+1]
                    // /E=3 causes wv[0] = (w[0]+w[0]+w[1])/3
                    // and wv[n-1] = (w[n-2]+w[n-1]+w[n-1])/3
```

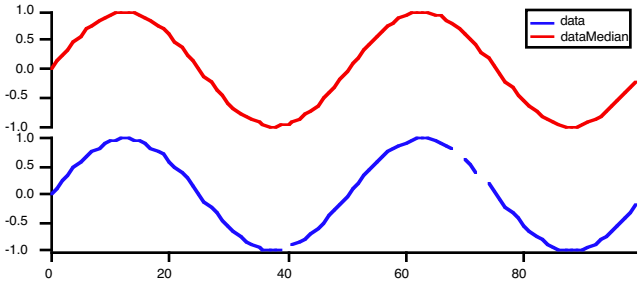
Demonstrate the impulse response of Savitzky-Golay Smoothing:

```
Make/O/N=100 wv
wv= p==50    // 1 at center of wave, 0 elsewhere; an impulse
SetScale/P x, 0, 1/1000, "s", wv           // 1000 Hz sampling rate
Smooth/S=2 5,wv
Display wv
ModifyGraph mode=8,marker=19
FFT/MAG/DEST=fftMag wv
Display fftMag
```



Replace NaN with median:

```
Make/O/N=100 data= enoise(1)>.9 ? NaN : sin(x/8)      // signal with NaNs
Duplicate/O data, dataMedian
Smooth/M=(NaN) 5, dataMedian      // replace (only) NaNs with 5-point median
```



Binomial Smoothing References

Marchand, P., and L. Marmet, *Revues of Scientific Instrumentation 54*, 1034, 1983.

Savitzky-Golay Smoothing References

Savitzky, A., and M.J.E. Golay, *Analytical Chemistry*, *36*, 1627-1639, 1964.

Steiner, J., Y. Termonia, and J. Deltour, *Analytical Chemistry*, *44*, 1906-1909, 1972.

Madden, H., *Analytical Chemistry*, *50*, 1386-1386, 1978.

Percentile References

<http://en.wikipedia.org/wiki/Percentile>

<http://cnx.org/content/m10805/latest/>

See Also

See the **Loess**, **MatrixConvolve**, and **MatrixFilter** operations for true 2D smoothing.

The **FilterFIR**, **FilterIIR**, and **Loess** operations; the Interpolate XOP.

Also see the Smooth Operation Responses example experiment.

SmoothCustom

SmoothCustom [**/E=*endEffect***] ***coefsWaveName*, *waveName*** [**, *waveName***]…

Note: SmoothCustom is obsolete. Use the **FilterFIR** operation instead. For multidimensional data use the **MatrixConvolve** or **MatrixFilter** operations.

The SmoothCustom operation smooths waves by convolving them with *coefsWaveName*.

Parameters

coefsWaveName must be single or double floating point, must not be one of the destination *waveName*s, must not be complex.

waveName is a numeric destination wave that is overwritten by the convolution of itself and *coefsWaveName*.

Flags

/E=*endEffect* End effect method, a value between 0 and 3. See the **Smooth** operation for a description of the /E flag.

Details

The convolution is in the time domain. That is, the FFT is not employed. For this reason the length of *coefsWaveName* should be small or small in comparison to the destination waves.

SmoothCustom presumes that the middle point of *coefsWaveName* corresponds to the delay = 0 point. The "middle" point number = trunc(numpnts(*coefsWaveName*-1)/2). *coefsWaveName* usually contains the two-sided impulse response of a filter, and contains an odd number of points. This is the type of wave created by FilterFIR.

SmoothCustom ignores the X scaling of all the waves.

The SmoothCustom operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

Sort

Sort [**/A /DIML /C /R**] ***sortKeyWaves, sortedWaveName*** [**, *sortedWaveName***]…

The Sort operation sorts the *sortedWaveName*s by rearranging their Y values to put the data values of *sortKeyWaves* in order.

Parameters

sortKeyWaves is either the name of a single wave, to use a single sort key, or the name of multiple waves in braces, to use multiple sort keys.

All waves must be of the same length.

The *sortKeyWaves* must not be complex.

Flags

/A Alphanumeric sort. When *sortKeyWaves* includes text waves, the normal sorting places "wave1" and "wave10" before "wave9". Use /A to sort the number portion numerically, so that "wave9" is sorted before "wave10".

/C Case-sensitive sort. When *sortKeyWaves* includes text waves, the sort is case-insensitive unless you use the /C flag to make it case-sensitive.

/DIML Moves the dimension labels with the values (keeps any row dimension label with the row's value).

/R Reversed sort; sort from largest to smallest.

Details

sortKeyWaves are not actually sorted unless they also appear in the list of destination waves.

The sort algorithm does not maintain the relative position of items with the same key value.

Examples

```
Sort/R myWave,myWave       // sorts myWave in decreasing order
Sort xWave,xWave,yWave     // sorts x wave in increasing order,
                           // corresponding yWave values follow.
Make/O/T myWave={"1st","2nd","3rd","4th"}
Make/O key1={2,1,1,1}      // places 2nd, 3rd, 4th before 1st.
Make/O key2={0,1,3,2}      // arranges 2nd, 3rd, 4th as 2nd, 4th, 3rd.
Sort {key1,key2},myWave    // sorts myWave in increasing order by key1.
                           // For equal key1 values, sorted by key2.
                           // Result is myWave={"2nd","4th","3rd","1st"}
Make/O/T tw={"w1","w10","w9","w-2.1"}
Sort/A tw,tw     // sorts tw in increasing number-aware order:
                 // Result is tw={"w-2.1","w1","w9","w10"}
```

See Also

The **MakeIndex**, **IndexSort**, **Reverse** and **SortList** operations. See **Sorting** on page III-134.

SortList

SortList(*listStr* [, *listSepStr* [, *options*])

The SortList function returns *listStr* after sorting it according to the default or *listSepStr* and *options* parameters. *listStr* should contain items separated by the *listSepStr* character, such as "the first item;second item;".

Use SortList to sort the items in a string containing a list of items separated by a single character, such as those returned by functions like **TraceNameList** or **WaveList**, or a line of text from a delimited text file.

listSepStr and *options* are optional; their defaults are ";" and 0 (alphabetic sort), respectively.

Details

listStr is treated as if it ends with a *listSepStr* even if it doesn't. The returned list will always have an ending *listSepStr* character.

Only the first character of *listSepStr* is used. If *listSepStr* is `""` then the default of ";" is used.

options is a literal number which controls the sorting method. *options* is one of:

0: Default sort (ascending case-sensitive alphabetic ASCII sort).
1: Descending sort.
2: Numeric sort.
4: Case-insensitive sort.
8: Case-sensitive alphanumeric sort using system script.
16: Case-insensitive alphanumeric sort that sorts wave0 and wave9 before wave10.

or a bitwise combination of the above with the following restriction: only one of 2, 4, 8, or 16 may be specified. The legal values are thus 0, 1, 2, 3, 4, 5, 8, 9, 16, and 17. Other values will produce undefined sorting criteria.

Examples

```
// alphabetic sorts
Print SortList("b;c;a;")                    // prints "a;b;c;"
Print SortList("you,me,i", ",", 4)          // prints "i,me,you,"
Print SortList("9,93,91,33,15,3", ",")      // prints "15,3,33,9,91,93,"
Print SortList("Zx;abc;All;", ";", 0)       // prints "All;Zx;abc;"
Print SortList("Zx;abc;All;", ";", 8)       // prints "abc;All;Zx;"
Print SortList("w9;w10;w02;", ";", 16)      // prints "w02;w9;w10;"

// numeric sorts
Print SortList("9,93,91,33,15,3",",",2)     // prints "3,9,15,33,91,93,"

// (reversed)
Print SortList("9,93,91,33,15,3",",",3)     // prints "93,91,33,15,9,3,"
```

See Also

The **Sort**, **StringFromList** and **WaveList** functions.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

SoundInRecord

SoundInRecord [**/Z**] ***wave***

The SoundInRecord operation records audio input at the sample rate obtained from the wave's X scaling and for the number of points determined by the length of the wave. The recording is done synchronously.

The number type of the wave must be one of the types reported by the **SoundInStatus** operation in the V_SoundInSampSize variable. On Windows this will typically be 8- or 16-bit integer while on Macintosh 16-bit integer and 32-bit floating point (the OS X native type) will be supported.

To record in stereo, provide a 2 column wave. (The software is designed to handle any number of channels but has not been tested on more than 2.)

Flags

/Z Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SoundInRecord requires a computer with sound inputs. Several sample experiments using sound input can be found in your Igor Pro Folder in the Examples folder.

See Also

The **SoundInSet**, **SoundInStartChart**, and **SoundInStatus** operations.

SoundInSet

SoundInSet [**/Z**][**gain=*g*, agc=*a***]

The SoundInSet operation is used to setup the input device for recording.

Parameters

SoundInSet can accept multiple *keyword =value* parameters on one line.

agc=*a* Turns automatic gain control mode on (*a*=1) or off (*a*=0). Will generate an error if device does not support setting agc. Use SoundInStatus to check or use /Z flag to make errors nonfatal.

Windows: This is not supported and V_SoundInAGC from the SoundInStatus command always returns -1.

gain=*g* Sets input gain, 0 is lowest gain and 1 is highest. Will generate an error if device does not support setting gain. Use SoundInStatus to check or use /Z flag to make errors nonfatal.

Windows: SoundInSet attempts to adjust the master gain of the sound input device but not all sound cards have a master gain. If V_SoundInGain from the SoundInStatus command returns -1, you will have to use your sound card software to adjust the input gain for the particular input source your are using. On some cards there are separate line-in and microphone-in sources.

Flags

/Z Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SoundInSet requires a computer with sound inputs. Several sample experiments using sound inputs are in your Igor Pro Folder in the Examples folder.

See Also

The **SoundInRecord**, **SoundInStartChart**, and **SoundInStatus** operations.

SoundInStartChart

SoundInStartChart [**/Z**] ***buffersize , destFIFOname***

The SoundInStartChart operation starts audio data acquisition into the given FIFO.

Parameters

buffersize is the number of bytes to allocate for the interrupt time buffer which then feeds into the given Igor named FIFO *destFIFOname*. The FIFO must be set up with the correct number of channels and number type - use **SoundInStatus** to find legal values. The sample rate is read from the FIFO also, so that also needs to be correct.

Flags

/Z Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SoundInStartChart requires a computer with sound inputs. Several sample experiments using sound inputs are in your Igor Pro Folder in the Examples folder.

On systems where 32-bit floating point data is supported, you can use NewFIFOChan with no flags and a range of -1 to 1.

See Also

The **SoundInRecord**, **SoundInSet**, **SoundInStatus** and **SoundInStopChart** operations, and **FIFOs and Charts** on page IV-276.

SoundInStatus

SoundInStatus

The SoundInStatus operation creates and sets a set of variables and strings with information about the current sound input device. The variable V_flag is set to an error code and will be zero if the device is available. If not then none of the following are valid.

Variables	Contents
S_SoundInName	String with name of device.
V_SoundInAGC	Automatic gain control on or off (1 or 0). This is an optional item and if the current device does not support AGC then V_SoundInAGC will be set to -1.
V_SoundInChansAv	Available number of channels (e.g., 1 for mono, 2 for stereo).
V_SoundInGain	Current input gain. Ranges from 0 (lowest) to 1. This is an optional item and if the current device does not support gain then V_SoundInGain will be set to -1.
V_SoundInSampSize	Bits set depending on number of bits available in a sample. Bit 0: Set if can do 8 bits. Bit 1: Set if can do 16 bits. Bit 3: Set if 32-bit floating point is supported (range is -1 to 1).
W_SoundInRates	Wave containing sample rate info: if point zero contains zero then points 1 and 2 contain the lower and upper limits of a continuous range else point zero contains the number of discrete rates which follow in the wave. The usual rates are 44100 Hz and 4800 Hz.

See Also
The **SoundInRecord**, **SoundInSet**, and **SoundInStartChart** operations.

SoundInStopChart

SoundInStopChart [/Z]
The SoundInStopChart operation stops audio data acquisition started by SoundInStartChart.

Flags
/Z Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details
SoundInStopChart requires a computer equipped with sound input hardware.

Audio data acquisition also stops automatically when an experiment is closed.

See Also
The **SoundInStartChart** and **SoundInStatus** operations.

SpecialCharacterInfo

SpecialCharacterInfo(*notebookNameStr, specialCharacterNameStr, whichStr*)
The SpecialCharacterInfo function returns a string containing information about the named special character in the named notebook window.

Parameters
If *notebookNameStr* is "", the top visible notebook is used. Otherwise *notebookNameStr* contains either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-95 for details on host-child specifications.

specialCharacterNameStr is the name of a special character in the notebook.

If specialCharacterNameStr is "" and if exactly one special character is selected, the selected special character is used. If other than exactly one special character is selected, an error is returned.

whichStr identifies the information item you want. Because SpecialCharacterInfo can return several items that may contain semicolons, it does not return a semicolon-separated keyword-value list like other info functions. Instead it returns just one item as specified by *whichStr*.

Details
Here are the supported values for *whichStr*.

Keyword	Returned Information
NAME	The name of the special character.
FRAME	0: None 1: Single 2: Double 3: Triple 4: Shadow
LOC	Paragraph and character position (e.g., 1,3).
SCALING	Horizontal and vertical scaling in units of one tenth of a percent (e.g., 1000,1000).
TYPE	Special character type is: Picture, Graph, Table, Layout, Action, ShortDate, LongDate, AbbreviatedDate, Time, Page, TotalPages, or WindowTitle.

These keywords apply to Igor-object pictures only. If the specified character is not an Igor-object picture, "" is returned.

Keyword	Returned Information
WINTYPE	1 for graphs, 2 for tables, 3 for layouts.
OBJECTNAME	The name of the window with which the special character is associated.

The remaining keywords apply to notebook action characters only. If the specified special character is not a notebook action character, "" is returned.

Keyword	Returned Information
BGRGB	Background color in RGB format (e.g., 65535,65534,49151).
COMMANDS	Command string.
ENABLEBGRGB	1 if the action's background color is enabled, 0 if not.
HELPTEXT	Help text string.
IGNOREERRORS	0 or 1.
LINKSTYLE	0 or 1.
PADDING	The value of the left, right, top, bottom and internal padding properties, in that order (.e.g, 4,4,4,4,8).
PICTURE	1 if the action has a picture, 0 if not.
PROCPICTNAME	The name of the action Proc Picture or "" if none.
QUIET	0 or 1.
SHOWMODE	1: Title only 2: Picture only 3: Picture below title 4: Picture above title 5: Picture to the left of title 6: Picture to the right of title
TITLE	Title string.

If *whichStr* is an unknown keyword, SpecialCharacterInfo returns "" but does not generate an error.

Examples

```
Function PrintSpecialCharacterInfo(notebookName, specialCharacterName)
   String notebookName, specialCharacterName

   String typeStr=SpecialCharacterInfo(notebookName, specialCharacterName, "TYPE")
   String locStr=SpecialCharacterInfo(notebookName, specialCharacterName, "LOC")
```

```
    Printf "TYPE: %s\r", typeStr
    Printf "LOC: %s\r", locStr
End
```

See Also

The **Notebook** and **NotebookAction** operations; the **SpecialCharacterList** function; **Using Igor-Object Pictures** on page III-21.

SpecialCharacterList

SpecialCharacterList(*notebookNameStr, separatorStr, mask, flags*)

The SpecialCharacterList function returns a string containing a list of names of special characters in a formatted text notebook.

Parameters

If *notebookNameStr* is "", the top visible notebook is used. Otherwise *notebookNameStr* contains either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-95 for details on host-child specifications.

separatorStr should contain a single character, usually semicolon, to separate the names.

mask determines which types of special characters are included. *mask* is a bitwise parameter with values:

1: Pictures including graphs, tables and layouts.

2: Notebook actions.

4: All other special characters such as dates and times.

or a bitwise combination of the above for more than one type. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

flags is a bitwise parameter. Pass 0 to include all special characters or 1 to include only selected special characters. All other bits are reserved and should be passed as zero.

Details

Only formatted text notebooks have special characters. When called for a plain text notebook, SpecialCharacterList always returns "".

Examples

Print a list of all special characters in the top notebook:

```
Print SpecialCharacterList("", ";", -1, 0)
```

Prints a list of notebook action characters in Notebook0:

```
Print SpecialCharacterList("Notebook0", ";", 2, 0)
```

Print a list of selected notebook action characters in Notebook0:

```
Print SpecialCharacterList("Notebook0", ";", 2, 1)
```

See Also

The **Notebook** and **NotebookAction** operations; the **SpecialCharacterInfo** function.

SpecialDirPath

SpecialDirPath(*dirIDStr, domain, flags, createDir*)

The SpecialDirPath function returns a full path to a file system directory specified by *dirIDStr* and *domain*. It provides a programmer with a way to access directories of special interest, such as the preferences directory and the desktop directory.

The path returned always ends with a separator character which may be a colon, backslash, or forward slash depending on the operating system and the *flags* parameter.

SpecialDirPath depends on operating system behavior. The exact path returned depends on the locale, the operating system, the specific installation, the current user, and possibly other factors.

Parameters

dirIDStr is one of the following strings:

"Packages"	Place for advanced programmers to put preferences for their procedure packages.
"Documents"	The OS-defined place for users to put documents.

"Preferences"	The OS-defined place for applications to put preferences.
"Desktop"	The desktop.
"Temporary"	The OS-defined place for applications to put temporary files.
"Igor Pro User Files"	A guaranteed-writable folder for the user to store their own Igor files, and to activate extensions, help, and procedure files by creating shortcuts or aliases in the appropriate subfolders. Use only with domain = 0 (the current user). This is the folder opened using the Show Igor Pro User Files menu item in the Help menu.

domain permits discriminating between, for example, the preferences folder for all users versus the preferences folder for the current user. It is supported only for certain *dirIDStrs*. It is one of the following:

0:	The current user	(recommended value for most purposes).
1:	All users	(may generate an error or return the same path as 0).
2:	System	(may generate an error or return the same path as 1).

flags a bitwise parameter:

Bit 0: If set, the returned path is a native path (Macintosh-style on Mac OS 9, Unix-style on Mac OS X, Windows-style on Windows). If cleared, the returned path is a Macintosh-style path regardless of the current platform. In most cases you should set this bit to zero since Igor accepts Macintosh-style paths on all operating systems. You must set this bit to one if you are going to pass the path to an external script.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

createDir is 1 if you want the directory to be created if it does not exist or 0 if you do not want it to be created. This flag will not work if the current user does not have sufficient privileges to create the specified directory. In almost all cases it is not needed, you can't count on it, and you should pass 0.

Details

The *domain* parameter has no effect in most cases. In almost all cases you should pass 0 (current user) for this parameter. For values other than 0, SpecialDirPath might return an error which you must be prepared to handle.

In the event of an error, SpecialDirPath returns a NULL string and sets a runtime error code. You can check for an error like this:

```
String fullPath = SpecialDirPath("Packages", 0, 0, 0)
Variable len = strlen(fullPath)         // strlen(NULL) returns NaN
if (numtype(len) == 2)                  // fullPath is NULL?
   Print "SpecialDirPath returned error."
endif
```

Here is sample output from `SpecialDirPath("Packages",0,0,0)`:

Mac OS X	hd:Users:<*user*>:Library:Preferences:WaveMetrics:Igor Pro 6 Intel:Packages:
Windows	C:Documents and Settings:<*user*>:Application Data:WaveMetrics:Igor Pro 6:Packages:

where <*user*> is the name of the current user. The preferences directory may be hidden by some operating systems.

Example

For an example using SpecialDirPath, see **Saving Package Preferences** on page IV-226.

sphericalBessJ

sphericalBessJ(*n*, *x* [, *accuracy*])

The sphericalBessJ function returns the spherical Bessel function of the first kind and order *n*.

$$j_n(x) = \sqrt{\frac{\pi}{2x}} J_{n+1/2}(x).$$

For example:

$$j_0(x) = \frac{\sin(x)}{x} \quad j_1(x) = \frac{\sin(x)}{x^2} - \frac{\cos(x)}{x} \quad j_2(x) = \left(\frac{3}{x^3} - \frac{1}{x}\right)\sin(x) - \frac{3}{x^2}\cos(x).$$

Details
See the **bessI** function for details on accuracy and speed of execution.

See Also
The **sphericalBessJD** and **sphericalBessY** functions.

References
Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

sphericalBessJD

sphericalBessJD(*n*, *x* [, *accuracy*])
The sphericalBessJD function returns the derivative of the spherical Bessel function of the first kind and order *n*.

Details
See the **bessI** function for details on accuracy and speed of execution.

See Also
The **sphericalBessJ** and **sphericalBessY** functions.

sphericalBessY

sphericalBessY(*n*, *x* [, *accuracy*])
The sphericalBessY function returns the spherical Bessel function of the second kind and order *n*.

$$y_n(x) = \sqrt{\frac{\pi}{2x}}\,Y_{n+1/2}(x) \qquad y_0(x) = -\frac{\cos(x)}{x}$$

$$y_1(x) = -\frac{\cos(x)}{x^2} - \frac{\sin(x)}{x} \qquad y_2(x) = \left(\frac{1}{x} - \frac{3}{x^3}\right)\cos(x) - \frac{3}{x^2}\sin(x).$$

Details
See the **bessI** function for details on accuracy and speed of execution.

See Also
The **sphericalBessYD** and **sphericalBessJ** functions.

References
Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

sphericalBessYD

sphericalBessYD(*n*, *x* [, *accuracy*])
The sphericalBessYD function returns the derivative of the spherical Bessel function of the second kind and order *n*.

Details
See the **bessI** function for details on accuracy and speed of execution.

See Also
The **sphericalBessJ** and **sphericalBessY** functions.

sphericalHarmonics

sphericalHarmonics(*L*, *M*, *q*, *f*)
The sphericalHarmonics function returns the complex-valued spherical harmonics

$$Y_L{}^M(\theta,\phi) = (-1)^M \sqrt{\frac{2L+1}{4\pi}\frac{(L-M)!}{(L+M)!}}\,P_L{}^M(\cos\theta)e^{iM\phi}$$

where $P_L^M(\cos\theta)$ is the associated Legendre function.

See Also

The **legendreA** function. The NumericalIntegrationDemo.pxp experiment.

References

Arfken, G., *Mathematical Methods for Physicists*, Academic Press, New York, 1985.

SphericalInterpolate

SphericalInterpolate *triangulationDataWave, dataPointsWave, newLocationsWave*

The SphericalInterpolate operation works in conjunction with the SphericalTriangulate operation to calculate interpolated values on a surface of a sphere. Given a set of $\{x_i, y_i, z_i\}$ points on the surface of a sphere with their associated values $\{v_i\}$, the SphericalTriangulate operation performs the Delaunay triangulation and creates an output that is used by the SphericalInterpolate operation to calculate values at any other point on the surface of a sphere. The interpolation calculation uses Voronoi polygons to weigh the contribution of the nearest neighbors to any given location on the sphere.

Parameters

triangulationDataWave is a 13 column wave that was created by the SphericalTriangulate operation.

dataPoints is a 4 column wave. The first 3 columns are the $\{x_i, y_i, z_i\}$ locations that were used to create the triangulation, and the last column corresponds to the $\{v_i\}$ values at the triangulation locations.

newLocationsWave is a 3 column wave that specifies the x, y, z locations on the sphere at which the interpolated values are calculated. Note that internally, each triplet is normalized to a point on the unit sphere before it is used in the interpolation.

Details

You will always need to use the SphericalTriangulate operation first to generate the *triangulationDataWave* input for this operation.

The result of the operation are put in the wave W_SphericalInterpolation.

See Also

The **SphericalTriangulate** operation.

SphericalTriangulate

SphericalTriangulate [**/Z**] ***tripletWaveName***

The SphericalTriangulate operation triangulates an arbitrary XYZ triplet wave on a surface of a sphere.

It starts by normalizing the data to make sure that $\text{sqrt}(x^2+y^2+z^2)=1$, and then proceeds to calculate the Delaunay triangulation.

Flags

/Z No error reporting.

Details

The result of the triangulation is the wave M_SphericalTriangulation. This 13 column wave is used in SphericalInterpolate to obtain the interpolated values.

See Also

The **SphericalInterpolate** operation.

SplitString

SplitString /E=*regExprStr str* [, *substring1* [, *substring2*,… *substringN*]]

The SplitString operation uses the regular expression *regExprStr* to split *str* into subpatterns. See **Subpatterns** on page IV-162 for details. Each matched subpattern is returned sequentially in the corresponding substring parameter.

Parameters

str is the input string to be split into subpatterns.

The *substring1*…*substringN* output parameters must be the names of existing string variables if you need to use the matched subpatterns. The first matched subpattern is returned in *substring1*, the second in *substring2*, etc.

Flags

/E=*regExprStr* Specifies the Perl-compatible regular expression string containing subpattern definition(s).

Details

regExprStr is a regular expression with successive subpattern definitions, such as shown in the examples. (Subpatterns are regular expressions within parentheses.)

For unmatched subpatterns, the corresponding substring is set to "". If you specify more substring parameters than subpatterns, the extra parameters are also set to "".

The number of matched subpatterns is returned in V_flag.

The part of *str* that matches *regExprStr* (often all of *str*) is stored in S_value.

Examples

```
// Split the output of the date() function:
Print date()
  Mon, May 2, 2005
String expr="([[:alpha:]]+), ([[:alpha:]]+) ([[:digit:]]+), ([[:digit:]]+)"
String dayOfWeek, monthName, dayNumStr, yearStr
SplitString/E=(expr) date(), dayOfWeek, monthName, dayNumStr, yearStr
Print V_flag
  4
Print dayOfWeek
  Mon
Print monthName
  May
Print dayNumStr
  2
Print yearStr
  2005
Print S_value
  Mon, May 2, 2005
// Get the part of str that matches regExprStr
SplitString/E=",.*," "stuff in front,second value,stuff at end"
Print S_value
  ,second value,
```

See Also

Regular Expressions on page IV-152 and **Subpatterns** on page IV-162.

The **sscanf** and **Grep** operations. The **strsearch** and **str2num** functions.

sprintf

sprintf *stringName, formatStr* [*, parameter*]…

The sprintf operation is the same as printf except it prints the formatted output to the string variable *stringName* rather than to the history area.

Parameters

formatStr See printf.

parameter See printf.

stringName Results are "printed" into the named string variable.

See Also

The **printf** operation for complete format and parameter descriptions and **Creating Formatted Text** on page IV-230.

sqrt

sqrt(*num*)

The sqrt function returns the square root of *num* or NaN if *num* is negative.

In complex expressions, *num* is complex, and sqrt(*num*) returns the complex value x + *i*y.

sscanf

sscanf *scanStr, formatStr, var* [**, *var***]

The sscanf operation is useful for parsing text that contains numeric or string data. It is based on the C sscanf function and provides a subset of the features available in C.

Here is a trivial example:

```
Variable v1
sscanf "Value= 1.234", "Value= %f", v1
```

This skips the text "Value=" and the following space and then converts the text "1.234" (or whatever number appeared there) into a number and stores it in the local variable v1.

The sscanf operation sets the variable V_flag to the number of values read. You can use this as an initial check to see if the *scanStr* is consistent with your expectations.

Note: The sscanf operation is supported in user functions only. It is not available using the command line, using a macro, or using the Execute operation.

Parameters

scanStr contains the text to be parsed.

formatStr is a format string which describes how the parsing is to be done.

formatStr is followed by the names of one or more local numeric or string variables or NVARs (references to global numeric variables) or SVARs (references to global string variables), which are represented by *var* above.

sscanf can handle a maximum of 100 *var* parameters.

Details

The format string consists of the following:

Normal text, which is anything other than a percent sign ("%") or white space.

White space (spaces, tabs, linefeeds, carriage returns).

A percent ("%") character, which is the start of a conversion specification.

The trivial example illustrates all three of these components.

```
Variable v1
sscanf "Value= 1.234", "Value= %f", v1
```

sscanf attempts to match normal text in the format string to the identical normal text in the scan string. In the example, the text "Value=" in the format string skips the identical text in the scan string.

sscanf matches a single white space character in the format string to 0 or more white space characters in the scan string. In the example, the single space skips the single space in the scan string.

When sscanf encounters a percent character in the format string, it attempts to convert the corresponding text in the scan string into a number or string, depending on the conversion character following the percent, and stores the resulting number or string in the corresponding variable in the parameter list. In the example, "%f" converts the text "1.234" into a number which it stores in the local variable v1.

A conversion specification consists of:

A percent character ("%").

An optional "*", which is a conversion suppression character.

An optional number, which is a maximum field width.

A conversion character, which specifies how to interpret text in the scan string.

Don't worry about the suppression character and the maximum width specification for now. They will be explained later.

The sscanf operation supports a subset of the conversion characters supported by the C sscanf operation. The supported conversion characters, which are case-sensitive, are:

d	Converts text representing a decimal number into an integer numeric value.
i	Converts text representing a decimal, octal or hexadecimal number into an integer value. If the text starts with "0x" (zero-x), it is interpreted as hexadecimal. Otherwise, if it starts with "0" (zero), it is interpreted as octal. Otherwise it is interpreted as decimal.
o	Converts text representing an octal number into an integer numeric value.

u	Converts text representing an unsigned decimal number into an integer numeric value.
x	Converts text representing a hexadecimal number into an integer numeric value.
c	Converts a single character into an integer value which is the ASCII code representing that character.
e	Converts text representing a decimal number into a floating point numeric value.
f	Same as e.
g	Same as e.
s	Stores text up to the next white space into a string.
[	Stores text that matches a list of specific characters into a string. The list consists of the characters inside the brackets ("%[abc]"). If the first character is "^", this means to match any character that is **not** in the list. You can specify a range of characters to match. For example `"%[A-Z]"` matches all of the upper case letters and `"%[A-Za-z]"` matches all of the upper and lower case letters.

Here are some simplified examples to illustrate each of these conversions.

```
Variable v1
String s1
```

Convert text representing a decimal number to an integer value:

```
sscanf "1234", "%d", v1
```

Convert text representing a decimal, octal, or hexadecimal number:

```
sscanf "1.234", "%i", v1          // Convert from decimal.
sscanf "01234", "%i", v1          // Convert from octal.
sscanf "0x123", "%i", v1          // Convert from hex.
```

Convert text representing an octal number:

```
sscanf "1234", "%o", v1
```

Convert text representing an unsigned decimal number:

```
sscanf "1234", "%u", v1
```

Convert text representing a hexadecimal number:

```
sscanf "1FB9", "%x", v1
```

Convert a single character:

```
sscanf "A", "%c", v1
```

Convert text representing a decimal number to an floating point value:

```
sscanf "1.234", "%e", v1
sscanf "1.234", "%f", v1
sscanf "1.234", "%g", v1
```

Copy a string of text up to the first white space:

```
sscanf "Hello There", "%s", s1
```

Copy a string of text matching the specified characters:

```
sscanf "+4.27", "%[+-]", s1
```

In a C program, you will sometimes see the letters "l" (ell) or "h" between the percent and the conversion character. For example, you may see "%lf" or "%hd". These extra letters are not needed or tolerated by Igor's sscanf operation.

When sscanf matches the format string to the scan string, it reads from the scan string until a character that would be inappropriate for the section of the format string that sscanf is trying to match. In the following example, sscanf stops reading characters to be converted into a number when it hits the first character that is not appropriate for a number.

```
Variable v1
String s1, s2
sscanf "1234Volts DC", "%d%s %s", v1, s1, s2
```

sscanf stops matching text for "%d" when it hits "V" and stores the converted number in v1. It stops matching text for the first "%s" when it hits white space and stores the matched text in s1. It then skips the

space in the scan string because of the corresponding space in the format string. Finally, it matches the remaining text to the second "%s" and stores the text in s2.

The maximum field width must appear just before the conversion character ("d" in this case).

```
Variable v1, v2
sscanf "12349876", "%4d%4d", v1, v2
```

The suppression character ("*") is used in a conversion specification to skip values in the scan string. It parses the value, but sscanf does not store the value in any variable. In the following example, we read one number into local variable v1, skip a colon, and read another number into local variable v2, skip a colon, and read another number into local variable v3.

```
Variable v1, v2, v3
sscanf "12:30:45", "%d%*[:]%d%*[:]%d", v1, v2, v3
```

Here "%*[:]" means "read a colon character but don't store it anywhere". The "*" character must appear immediately after the percent. Note that there is nothing in the parameter list corresponding to the suppressed strings.

If the text in the scan string is not consistent with the text in the format string, sscanf may not read all of the values that you expected. You can check for this using the V_flag variable, which is set to the number of values read. This kind of inconsistency does not cause sscanf to return an error to Igor, which would cause procedure execution to abort. It is a situation that you can deal with in your procedure code.

The sscanf operation returns the following kinds of errors:

- Out-of-memory.
- The number of parameters implied by *formatStr* does not match the number of parameters in the *var* list.
- *formatStr* calls for a numeric variable but the parameter list expects a string variable.
- *formatStr* calls for a string variable but the parameter list expects a numeric variable.
- *formatStr* includes an unsupported, unknown or incorrectly constructed conversion specification.
- The *var* list references a global variable that does not exist.

Examples

Here is a simple example to give you the general idea:

```
Function SimpleExample()
   Variable v1, valuesRead
   sscanf "Value=1.234", "Value=%g", v1
   valuesRead = V_flag
   if (valuesRead != 1)
      Printf "Error: Expected 1 value, got %d values\r", valuesRead
   else
      Printf "Value read = %g\r", v1
   endif
End
```

For an example that uses sscanf to load data from a text file, see the Load File Demo example in "Igor Pro Folder:Examples:Programming".

See Also

str2num, **strsearch**, and **stringmatch**.

Stack

Stack [***flags***] [***objectName***][**,** ***objectName***]…

The Stack operation stacks the named objects in the top page layout.

Parameters

objectName is the name of a graph, table, picture or annotation object in the top page layout.

Flags

/A=(*rows*,*cols*)	/I	/O=*objTypes*	/S
/G=*grout*	/M	/R	/W=(*left*,*top*,*right*,*bottom*)

See Also

The **Tile** operation for details on the flags and parameters.

StackWindows

StackWindows [***flags***] [***windowName*** [**,** ***windowName***]…]

The StackWindows operation stacks the named windows on the desktop.

Flags

/A=(*rows*,*cols*)	/G=*grout*	/O=*objTypes*	/P	/W=(*left*,*top*,*right*,*bottom*)
/C	/I	/M	/R	

See Also

See the **TileWindows** operation for details on the flags and parameters.

startMSTimer

startMSTimer

The startMSTimer function creates a new microsecond timer and returns a timer reference number.

Details

You can create up to ten different microsecond timers using startMSTimer. A valid timer reference number is a number between 0 and 9. If startMSTimer returns -1, there are no free timers available. startMSTimer works in conjunction with stopMSTimer.

See Also

The **stopMSTimer** and **ticks** functions.

Static

Static constant ***objectName*** **=** ***value***

Static strconstant ***objectName*** **=** ***value***

Static Function ***funcName*****()**

Static Picture ***pictName***

The Static keyword declaration specifies that a constant, user function, or Proc Picture is local to the procedure file in which it appears. Static objects can only be used by other functions; they cannot be accessed from macros; they cannot be accessed from other procedure files or from the command line.

See Also

Static Functions on page IV-83, **Proc Pictures** on page IV-43, and **Constants** on page IV-40.

StatsAngularDistanceTest

StatsAngularDistanceTest [***flags***][***srcWave1***, ***srcWave2***, ***srcWave3***…]

The StatsAngularDistanceTest operation performs nonparametric tests on the angular distance between sample data and reference directions for two or more samples in individual waves. The angular distance is the shortest distance between two points on a circle (in radians). Specify the sample waves using /WSTR or by listing them following the flags. Set reference directions with /ANG, /ANGW, or the sample mean direction.

Flags

/ALPH=*val*	Sets the significance level (default 0.05).
/ANG={*d1*, *d2*}	Sets reference directions (in radians) for two samples; for more than two samples use /ANGW.
/ANGM	Computes the mean direction of each sample and uses it as the reference direction.
/ANGW=*dWave*	Sets reference directions (in radians) for more than two samples using directions in *dWave*, which must be single or double precision.
/APRX=*m*	Controls the approximation method for computing the P-value in the case of two samples (Mann-Whitney Wilcoxon). See **StatsWilcoxonRankTest** for more details. The default value is 0, which may require long computation times if your sample size is large. Use /APRX=1 if you have a large sample and you expect ties in the data.
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/TAIL=*tail* *tail*=1: Lower tail.

tail=2: Upper tail (default).

tail=4: Two tail.

Use any test combination by adding the values. The P value corresponding to the last tail calculated will be entered in the table.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The inputs for StatsAngularDistanceTest are two or more waves each corresponding to individual sample. The waves must be single or double precision expressing the angles in radians. There is no restriction on the number of points or dimensionality of the waves but the data should not contain NaNs or INFs. We recommend that you use double precision waves, especially if there are ties in the data. The reference directions should also be in radians. For two samples, StatsAngularDistanceTest computes the angular distances between the input data and the reference directions and then uses the Mann-Whitney-Wilcoxon test (**StatsWilcoxonRankTest**). Results are stored in the W_WilcoxonTest wave and in the corresponding table. For more than two samples, StatsAngularDistanceTest uses the Kruskal-Wallis test, storing results in the wave W_KWTestResults wave in the current data folder.

V_flag will be set to -1 for any error and to zero otherwise.

References

See, in particular, Chapter 27 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWilcoxonRankTest** and **StatsKWTest**.

Examples:Statistics:Circular Statistics:AngularDistanceTest.pxp.

StatsANOVA1Test

StatsANOVA1Test [***flags***] [***wave1, wave2,… wave100***]

The StatsANOVA1Test operation performs a one-way ANOVA test (fixed-effect model). The standard ANOVA test results are stored in the M_ANOVA1 wave in the current data folder.

Flags

/ALPH=*val* Sets the significance level (default 0.05).

/BF Performs the Brown and Forsythe test computing F'' and degrees of freedom. The W_ANOVA1BnF wave in the current data folder contains the output.

/Q No results printed in the history area.

/T=*k* Displays results in a table; additional tables are created with /BF and /W. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/W Performs the Welch test F' and computes degrees of freedom. The W_ANOVA1Welch wave in the current data folder contains the output.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Inputs to StatsANOVA1Test are two or more 1D numerical waves containing (one wave for each group of samples). Use NaN for missing entries or use waves with different numbers of points. The standard ANOVA results are in the M_ANOVA1 wave with corresponding row and column labels. Use /T to display the results in a table. In each case you will get the two degrees of freedom values, the F value, the critical value Fc for the choice of alpha and the degrees of freedom, and the P-value for the result. V_flag will be set to -1 for any error and to zero otherwise.

In some cases the ANOVA test may not be appropriate. For example, if groups do not exhibit sufficient homogeneity of variances. Although this may not be fatal for the ANOVA test, you may get more insight by performing the variances test in **StatsVariancesTest**.

If there are only two groups this test should be equivalent to **StatsTTest**.

You can evaluate the power of an ANOVA test for a given set of degrees of freedom and noncentrality parameter using:

```
power=1-StatsNCFCDF(StatsInvFCDF((1-alpha),n1,n2),n1,n2,delta)
```

Here n1 is the Groups' degrees of freedom, n2 is the Error degrees of freedom, and delta is the noncentrality parameter. For more information see ANOVA Power Calculations Panel and the associated example experiment.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsVariancesTest**, **StatsTTest**, **StatsNCFCDF**, and **StatsInvFCDF**.

StatsANOVA2NRTest

StatsANOVA2NRTest [***flags***] ***srcWave***

The StatsANOVA2NRTest operation performs a two-factor analysis of variance (ANOVA) on the data that has no replication where there is only a single datum for every factor level. *srcWave* is a 2D wave of any numeric type. Output is to the M_ANOVA2NRResults wave in the current data folder or optionally to a table.

Flags

/ALPH=*val* Sets the significance level (default 0.05).

/FOMD Estimates one missing value. You will also have to use a single or double precision wave for *srcWave* and designate the single missing value as NaN. The estimated value is printed to the history as well as the bias used to correct the sum of the squares of factor A.

/INT=*val* Sets the degree of interactivity.

val=0: No interaction between the factors (default).

val=1: Significant interaction effect between factors.

Combination with /MODL determines which factors to test:

val	Model 1	Model2	Model3
1		A&B	A
0	A&B	A&B	A&B

/MODL=*m* Sets the model number.

m=1: Factor A and factor B are fixed.

m=2: Both factors are random.

m=3: Factor A is fixed and factor B is random (default).

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Input to StatsANOVA2NRTest is a 2D wave in which the Factor A corresponds to rows and Factor B corresponds to columns. H0 provides that there is no difference in the means of the respective populations, i.e., if H0 is rejected for Factor A but accepted for Factor B that means that there is no difference in the means of the columns but the means of the rows are different.

NaN and INF entries are not supported although you may use a single NaN value in combination with the /FOMD flag. If srcWave contains dimension labels they will be used to designate the two factors in the output.

The contents of the M_ANOVA2NRResults output wave columns are as follows:

Column 0	Sum of the squares (SS) values
Column 1	Degrees of freedom (DF)
Column 2	Mean square (MS) values
Column 3	Computed F value for this test
Column 4	Critical F value (Fc) for the specified alpha
Column 5	Conclusion with 0 to reject H0 or 1 to accept it

The variable V_flag is set to zero if the operation succeeds or to -1 otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test** and **StatsANOVA2Test**.

StatsANOVA2RMTest

StatsANOVA2RMTest [***flags***] ***srcWave***

The StatsANOVA2RMTest operation performs analysis of variance (ANOVA) on *srcWave* where replicates consist of multiple measurements on the same subject (repeated measures). *srcWave* is a 2D wave of any numeric type. Output is to the M_ANOVA2RMResults wave in the current data folder or optionally to a table.

Flags

/ALPH=*val* Sets the significance level (default 0.05).

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Input to StatsANOVA2RMTest is the 2D *srcWave* in which the factor A (Groups) are columns and the different subjects are rows. It does not support NaNs or INFs.

The contents of the M_ANOVA2RMResults output wave columns are: the first contains the sum of the squares (SS) values, the second contains the degrees of freedom (DF), the third contains the mean square (MS) values, the fourth contains the single F value for this test, the fifth contains the critical F value for the specified alpha and degrees of freedom, and the last column contains the conclusion with 0 to reject H_0 or 1 to accept it. In each case H_0 corresponds to the mean level, which is the same for all subjects.

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA2NRTest** and **StatsANOVA2Test**.

StatsANOVA2Test

StatsANOVA2Test [*flags*] *srcWave*

The StatsANOVA2Test operation performs a two-factor analysis of variance (ANOVA) on *srcWave*. Output is to the M_ANOVA2Results wave in the current data folder or optionally to a table.

Flags

/ALPH=*val* Sets the significance level (default 0.05).

/FAKE=*num* Specifies the number of points in *srcWave* obtained by "estimation". *num* is subtracted from the total and error degrees of freedom.

/MODL=*m* Sets the model number.

- *m*=1: Factor A and factor B are fixed (default).
- *m*=2: Both factors are random.
- *m*=3: Factor A is fixed and factor B is random.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

- *k*=0: Normal with dialog (default).
- *k*=1: Kills with no dialog.
- *k*=2: Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Input to StatsANOVA2Test is the single or double precision 3D *srcWave* in which the factor A levels are columns, the factor B levels are rows, and the replicates are layers. If *srcWave* contains dimension labels they will be used to designate the factors in the output.

Ideally, the number of replicates must be equal for each factor and each level. StatsANOVA2Test supports both equal replication and proportional replication. Proportional replication allows for different number of data in each cell with missing data represented as NaN and the number of points in each cell is given by

```
Nij=(sum of data in row i)*(sum of data in column j)/number of samples.
```

If you have no replicates (a single datum per cell) use **StatsANOVA2NRTest** instead. If the number of replicates in your data does not satisfy these conditions you may be able to "estimate" additional replicates using various methods. In that case use the /FAKE flag so that the operation can account for the estimated data by reducing the total and error degrees of freedom. /FAKE only accounts for the number of estimates being used. You must provide an appropriate number of estimated values.

The contents of the M_ANOVA2Results output wave columns are: the first contains the sum of the squares (SS) values, the second the degrees of freedom (DF), the third contains the mean square (MS) values, the fourth contains the computed F value for this test, the fifth contains the critical Fc value for the specified alpha and degrees of freedom, and the last contains the conclusion with 0 to reject H_0 or 1 to accept it. In each case H_0 corresponds to the mean level, which is the same for all populations.

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test** and **StatsANOVA2NRTest**.

StatsBetaCDF

StatsBetaCDF(*x*, *p*, *q* [, *a*, *b*])

The StatsBetaCDF function returns the beta cumulative distribution function

$$F(x,p,q,a,b)=\frac{1}{B(p,q)}\int_0^{\frac{x-a}{b-a}} t^{p-1}(1-t)^{q-1}\,dt, \qquad \begin{matrix} p,q>0 \\ a\le x\le b \end{matrix}$$

where $B(p,q)$ is the beta function

$$B(p,q)=\int_0^1 t^{p-1}(1-t)^{q-1}\,dt.$$

The defaults (a=0 and b=1) correspond to the standard beta distribution were a is the location parameter, (b-a) is the scale parameter, and p and q are shape parameters.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBetaPDF** and **StatsInvBetaCDF**.

StatsBetaPDF

StatsBetaPDF(x, p, q [, a, b])

The StatsBetaPDF function returns the beta probability distribution function

$$f(x;p,q,a,b)=\frac{(x-a)^{p-1}(b-x)^{q-1}}{B(p,q)(b-a)^{p+q-1}}, \qquad \begin{matrix} a\le x\le b \\ p,q>0 \end{matrix}$$

where $B(p,q)$ is the beta function

$$B(p,q)=\int_0^1 t^{p-1}(1-t)^{q-1}\,dt.$$

The defaults (a=0 and b=1) correspond to the standard beta distribution were a is the location parameter, (b-a) is the scale parameter, and p and q are shape parameters. When p<1, $f(x=a)$ returns Inf.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBetaCDF** and **StatsInvBetaCDF**.

StatsBinomialCDF

StatsBinomialCDF(x, p, N)

The StatsBinomialCDF function returns the binomial cumulative distribution function

$$F(x;p,N)=\sum_{i=1}^{x}\left(\begin{matrix} N \\ i \end{matrix}\right)p^i(1-p)^{N-i}, \qquad x=1,2,...$$

where

$$\left(\begin{matrix} N \\ i \end{matrix}\right)=\frac{N!}{i!(N-i)!}.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBinomialCDF** and **StatsBinomialPDF**.

StatsBinomialPDF

StatsBinomialPDF(*x*, *p*, *N*)

The StatsBinomialPDF function returns the binomial probability distribution function

$$f(x;p,N)=\left(\begin{array}{c} N \\ x \end{array}\right)p^x(1-p)^{N-x}, \qquad x=0,1,2,...$$

where

$$\left(\begin{array}{c} N \\ x \end{array}\right)=\frac{N!}{x!(N-x)!}.$$

is the probability of obtaining *x* good outcomes in *N* trials where the probability of a single successful outcome is *p*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBinomialCDF** and **StatsInvBinomialCDF**.

StatsCauchyCDF

StatsCauchyCDF(*x*, μ, σ)

The StatsCauchyCDF function returns the Cauchy-Lorentz cumulative distribution function

$$F(x;\mu,\sigma)=\frac{1}{2}+\frac{1}{\pi}\tan^{-1}\left(\frac{x-\mu}{\sigma}\right).$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCauchyCDF** and **StatsCauchyPDF**.

StatsCauchyPDF

StatsCauchyPDF(*x*, μ, σ)

The StatsCauchyPDF function returns the Cauchy-Lorentz probability distribution function

$$f(x;\mu,\sigma)=\frac{1}{\sigma\pi}\frac{1}{1+\left(\frac{x-\mu}{\sigma}\right)^2},$$

where μ is the location parameter and σ is the scale parameter. Use μ=0 and σ=1 for the standard form of the Cauchy-Lorentz distribution.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCauchyCDF** and **StatsInvCauchyCDF**.

StatsChiCDF

StatsChiCDF(*x*, *n*)

The StatsChiCDF function returns the chi-squared cumulative distribution function for the specified value and degrees of freedom *n*.

$$F(x;n)=\frac{\gamma\left(\frac{n}{2},\frac{x}{2}\right)}{\Gamma\left(\frac{n}{2}\right)}.$$

where is γ(*a*,*b*) the incomplete gamma function. The distribution can also be expressed as

$$F(x;n) = 1 - gammq\left(\frac{n}{2}, \frac{x}{2}\right).$$

See Also
Chapter III-12, **Statistics** for a function and operation overview; **StatsChiPDF**, **StatsInvChiCDF**, and **gammq**.

StatsChiPDF

StatsChiPDF(*x, n*)

The StatsChiPDF function returns the chi-squared probability distribution function for the specified value and degrees of freedom as

$$f(x;n) = \frac{\exp\left(-\frac{x}{2}\right)x^{\frac{n}{2}-1}}{2^{\frac{n}{2}}\Gamma\left(\frac{n}{2}\right)}.$$

See Also
Chapter III-12, **Statistics** for a function and operation overview; **StatsChiCDF** and **StatsChiPDF**.

StatsChiTest

StatsChiTest [***flags***] ***srcWave1, srcWave2***

The StatsChiTest operation computes a χ^2 statistic for comparing two distributions or a χ^2 statistic for comparing a sample distribution with its expected values. In both cases the comparison is made on a bin-by-bin basis. Output is to the W_StatsChiTest wave in the current data folder or optionally to a table.

Flags

/NCON=*nCon* Specifies the number of constraints (0 by default), which reduces the number degrees of freedom and the critical value by *nCon*.

/S Sets the calculation mode to a single distribution where *srcWave1* represents an array of binned measurements and *srcWave2* represents the corresponding expected values.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.
- *k*=0: Normal with dialog (default).
- *k*=1: Kills with no dialog.
- *k*=2: Disables killing.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The source waves, *srcWave1* and *srcWave2*, must have the same number of points and can be any real numeric data type. Any nonpositive values (including NaN) in either wave removes the entry in both waves from consideration and reduces the degrees of freedom by one. The number degrees of freedom is initially the number of points in *srcWave1*-1-*nCon*. By default it is assumed that *srcWave1* and *srcWave2* represent two distributions of binned data.

When you specify /S, *srcWave1* must consist of binned values of measured data and *srcWave2* must contain the corresponding expected values. The calculation is:

$$\chi^2 = \sum_{i=0}^{n-1} \frac{\left(Y_i - V_i\right)^2}{V_i}.$$

Here Y_i is the sample point from *srcWave1*, V_i is the expected value of Y_i based on an assumed distribution (*srcWave2*), and n is the number of points in the each wave. If you do not use /S, it calculates:

$$\chi^2 = \sum_{i=0}^{n-1} \frac{\left(Y_{1i} - Y_{2i}\right)^2}{Y_{1i} + Y_{2i}},$$

where Y_{1i} and Y_{2i} are taken from *srcWave1* and *srcWave2* respectively.

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsContingencyTable**.

StatsCircularCorrelationTest

StatsCircularCorrelationTest [*flags*] ***waveA, waveB***

The StatsCircularTwoSampleTest operation peforms a number of tests for two samples of circular data. Using the appropriate flags you can choose between parametric or nonparametric, unordered or paired tests. The input consists of two waves that contain one or two columns. The first column contains angle data expressed in radians and an optional second column contains associated vector lengths. The waves must be either single or double precision floating point. Results are stored in the W_StatsCircularCorrelationTest wave in the current data folder and optionally displayed in a table. Some flags generate additional outputs, described below.

Flags

/ALPH=*val*	Sets the significance level (default 0.05).
/NAA	Performs a nonparametric angular-angular correlation test.
/PAA	Performs a parametric angular-angular correlation test.
/PAL	Performs a parametric angular-linear correlation test. In this case the angle wave is *waveA* and the linear data corresponds to *waveB*.
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors.

Details

The nonparametric test (/NAA) follows Fisher and Lee's modification of Mardia's statistic, which is an analogue of Spearman's rank correlation. The test ranks the angles of each sample and computes the quantities *r'* and *r''* as follows:

$$r' = \frac{\left\{\sum_{i=0}^{n-1} \cos\left[\frac{2\pi}{n}\left(r_{ai} - r_{bi}\right)\right]\right\}^2 + \left\{\sum_{i=0}^{n-1} \sin\left[\frac{2\pi}{n}\left(r_{ai} - r_{bi}\right)\right]\right\}^2}{n^2},$$

$$r'' = \frac{\left\{\sum_{i=0}^{n-1} \cos\left[\frac{2\pi}{n}\left(r_{ai} + r_{bi}\right)\right]\right\}^2 + \left\{\sum_{i=0}^{n-1} \sin\left[\frac{2\pi}{n}\left(r_{ai} + r_{bi}\right)\right]\right\}^2}{n^2}.$$

Here n is the number of data pairs and r_{ai} and r_{bi} are the ranks of the ith member in the first and second samples respectively.

The test statistic is $(n\text{-}1)(r'\text{-}r'')$, which is compared with the critical value (for one and two tails). The CDF of the statistic is a highly irregular function. The critical value is computed by a different methods according to n. For $3 \le n \le 8$, a built-in table of CDF transitions gives a "conservative" estimate of the critical value. For $9 \le n \le 30$, the CDF is approximated by a 7th order polynomial in the region $x > 0$. For $n \ge 30$, the CDF is from the asymptotic expression. For $3 \le n \le 30$, CDF values are obtained by Monte-Carlo simulations using 1e6 random samples for each n.

The parametric test for angular-angular correlation (/PAA) involves computation of a correlation coefficient r_{aa} and then evaluating the mean $\overline{r_{aa}}$ and variance $s^2_{r_{aa}}$ of equivalent correlation coefficients computed from the same data but by deleting a different pair of angles each time. The mean and variance are then used to compute confidence limits L1 and L2:

$$L1 = nr_{aa} - (n-1)\overline{r_{aa}} - Z_{\alpha(2)}\sqrt{\frac{s^2_{r_{aa}}}{n}},$$

$$L2 = nr_{aa} - (n-1)\overline{r_{aa}} + Z_{\alpha(2)}\sqrt{\frac{s^2_{r_{aa}}}{n}}$$

where $Z_{\alpha(2)}$ is the normal distribution two-tail critical value at the a level of significance. H_0 (corresponding to no correlation) is rejected if zero is not contained in the interval [L1,L2].

The parametric test for angular-linear correlation (/PAL) involves computation of the correlation coefficient r_{al} which is then compared with a critical value from χ^2 for alpha significance and two degrees of freedom.

$$r_{al} = \sqrt{\frac{r_{xc}^2 + r_{xs}^2 - 2r_{xc}r_{xs}r_{cs}}{1 - r_{cs}^2}},$$

where:

$$r_{xc} = \frac{\sum_{i=0}^{n-1} X_i \cos(a_i) - \frac{1}{n}\sum_{i=0}^{n-1} X_i \sum_{i=0}^{n-1} \cos(a_i)}{\sqrt{\left(\sum_{i=0}^{n-1} X_i^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1} X_i\right)^2\right)\left(\sum_{i=0}^{n-1} \cos^2(a_i) - \frac{1}{n}\left(\sum_{i=0}^{n-1} \cos(a_i)\right)^2\right)}},$$

$$r_{xs} = \frac{\sum_{i=0}^{n-1} X_i \sin(a_i) - \frac{1}{n}\sum_{i=0}^{n-1} X_i \sum_{i=0}^{n-1} \sin(a_i)}{\sqrt{\left(\sum_{i=0}^{n-1} X_i^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1} X_i\right)^2\right)\left(\sum_{i=0}^{n-1} \sin^2(a_i) - \frac{1}{n}\left(\sum_{i=0}^{n-1} \sin(a_i)\right)^2\right)}},$$

$$r_{cs} = \frac{\sum_{i=0}^{n-1} \cos(a_i)\sin(a_i) - \frac{1}{n}\sum_{i=0}^{n-1} \sin(a_i) \sum_{i=0}^{n-1} \cos(a_i)}{\sqrt{\left(\sum_{i=0}^{n-1} \sin^2(a_i) - \frac{1}{n}\left(\sum_{i=0}^{n-1} \sin(a_i)\right)^2\right)\left(\sum_{i=0}^{n-1} \cos^2(a_i) - \frac{1}{n}\left(\sum_{i=0}^{n-1} \cos(a_i)\right)^2\right)}}.$$

References

Fisher, N.I., and A.J. Lee, Nonparametric measures of angular-angular association, *Biometrica*, *69*, 315-321, 1982.

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvChiCDF**, **StatsInvNormalCDF**, and **StatsKendallTauTest**.

StatsCircularMeans

StatsCircularMeans [***flags***] ***srcWave***

The StatsCircularMeans operation calculates the mean of a number of circular means, returning the mean angle (grand mean), the length of the mean vector, and optionally confidence interval around the mean angle. Output is to the history area and to the W_CircularMeans wave in the current data folder.

Flags

/ALPH=*val* Sets the significance level (default 0.05).

/CI Calculates the confidence interval (labeled CI_t1 and CI_t2) around the mean angle.

/NSOA Performs nonparametric second order analysis according to Moore's version of Rayleigh's test where H_0 corresponds to uniform distribution around the circle. Moore's test ranks entries by the lengths of the mean radii (second column of the input) from smallest (rank 1) to largest (rank *n*) and then computes the statistic:

$$R' = \sqrt{\frac{\left(\frac{1}{n}\sum_{i=0}^{n-1}(i+1)\cos(a_i)\right)^2 + \left(\frac{1}{n}\sum_{i=0}^{n-1}(i+1)\sin(a_i)\right)^2}{n}},$$

where a_i are the mean angle entries (from column 1) corresponding to vector length rank (i+1). The critical value is obtained from Moore's distribution **StatsInvMooreCDF**.

/PSOA Perform parametric second order analysis where H_0 corresponds to no mean population direction. It assumes that the second order quantities are from a bivariate normal distribution. If this is not the case, use /NSOA above. The test statistic is:

$$F = \frac{k(k-2)}{2}\left[\frac{\bar{X}^2 S_{y^2} - 2\bar{X}\bar{Y}S_{xy} + \bar{Y}^2 S_{x^2}}{S_{x^2}S_{y^2} - S_{xy}^2}\right]$$

where

$$\bar{X} = \frac{1}{n}\sum_{i=0}^{n-1} X_i = \frac{1}{n}\sum_{i=0}^{n-1} r_i\cos(a_i),$$

$$\bar{Y} = \frac{1}{n}\sum_{i=0}^{n-1} Y_i = \frac{1}{n}\sum_{i=0}^{n-1} r_i\sin(a_i),$$

$$S_{x^2} = \sum_{i=0}^{n-1} X_i^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1} X_i\right)^2,$$

$$S_{y^2} = \sum_{i=0}^{n-1} Y_i^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1} Y_i\right)^2,$$

$$S_{xy} = \sum_{i=0}^{n-1} X_i Y_i - \frac{1}{n}\sum_{i=0}^{n-1} X_i \sum_{i=0}^{n-1} Y_i.$$

Here *n* is the number of means in *srcWave* and the critical value is computed from the F distribution, equivalent to executing:

```
Print StatsInvFCDF(1-alpha,2,n-2)
```

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The *srcWave* input to StatsCircularMeans must be a single or double precision two column wave containing in each row a mean angle (radians) and the length of a mean radius (the first column contains mean angles and the second column contains mean vector lengths). *srcWave* must not contain any NaNs or INFs. The confidence interval calculation follows the procedure outlined by Batschelet.

V_flag will be set to -1 for any error and to zero otherwise.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCircularMoments**, **StatsInvMooreCDF**, and **StatsInvFCDF**.

StatsCircularMoments

StatsCircularMoments [***flags***] ***srcWave***

The StatsCircularMoments operation computes circular statistical moments and optionally performs angular uniformity tests for the data in *srcWave*. The extent of the calculation is determined by the requested moment. The default results are stored in the W_CircularStats wave in the current data folder and are optionally displayed in a table. Additional results are listed under the corresponding flags.

Flags

/ALPH=*alpha* Sets an *alpha* value for computing confidence intervals (default is 0.05).

/AXD=*p* Designates the input as p-axial data. For example, if the input represents undirected lines then *p* =2 and the operation multiplies the angles by a factor *p* (after shifting /ORGN and accounting for /CYCL). It does not back-transform the mean or median axis.

/CYCL=*cycle* Specifies the length of the data cycle. You do not need to do so if you are using one of the built-in modes, but this is still a useful option, as for setting the length of a particular month when using /MODE=5.

/GRPD={*start*, *delta*} Computes circular statistics for grouped data. In this case *srcWave* contains frequencies or the number of events that belong to a particular angle group. There are as many groups as there are elements in *srcWave*. The first group is centered at *start* radians and each consecutive group is centered *delta* radians away. You must set both the *start* and *delta* to sensible values. *srcWave* may contain NaNs but it is an error if all values are NaN. The only other flags that work in combination with this flag are /Q, /T, and /Z.

/KUPR Tests the uniformity of the distribution for ungrouped data using Kuiper statistic. The data are converted into a set $\{x_i\}$ by normalizing the input angles to the range [0,1], ranking the results then using the two quantities D_+ and D_- to compute the Kuiper statistic

$$V=\left(D_+ + D_-\right)\left(\sqrt{n}+0.155+0.24/\sqrt{n}\right),$$

where

$$D_+ = \text{Max of: } \frac{1}{n}\text{-}x_0\,,\ \frac{2}{n}\text{-}x_1,\ldots,1\text{-}x_{n-1},$$

$$D_- = \text{Max of: } x_0\,,\ x_{1-\frac{1}{n}},\ldots,\ x_{n-1-\frac{n-1}{n}},$$

and *n* is the number of valid points in *srcWave*. You can find the results in the wave W_CircularStats under row label “Kuiper V” and “Kuiper CDF(V)”. See Fisher and Press *et al*. for more information.

/LOS Computes Linear Order Statistics by sorting the angle values from small to large, dividing each angle by 2π and shifting the origin so that the output range is [0,1]. The results are stored in the wave W_LinearOrderStats in the current data folder. The X scaling of the wave is set so that the offset and the delta are 1/(*n*+1) where *n* is the number of non-NaN points in the input.

/M=*moment* Computes specified moments. By default, it computes the second order moments as well as skewness, kurtosis, median, and mean deviation. Use /M=1 for the first moment. For higher moments, both the specified moment and all the default quantities are computed.

/MODE=*mode* Handles special types of data.

mode	**Data in *srcWave***
0	Angles in radians [0,2π]
1	Angles in radians [-π, π]
2	Angles in degrees [0,360]
3	Angles in degrees [-180,180]
4	Igor date format for one year cycles.
5	Igor date format for one month cycles.
6	Igor date format for one week cycles.
7	Igor date format for one day cycles.
8	Igor date format for one hour cycles.

/ORGN=*origin* Specifies the origin of the data (the value corresponding to an angle of zero degrees). For example, if you are using Igor date format and you want the origin to be the first second in year YYYY, use `/ORGN=(date2secs(YYYY,1,1))`.

/Q No results printed in the history area.

/RAYL[=*meanDirection*]

Performs the Rayleigh test for uniformity. If the "alternative" mean direction is specified (in radians), the test computes

r0Bar=rBar cos(tBar-*meanDirection*)

and then computes the significance probability of r0Bar. The null hypothesis H_0 corresponds to uniformity. It is rejected when r0Bar is too large. If the mean direction is not specified then r0Bar is rBar which is always calculated as part of the first moments so the operation only computes the relevant significance probability (P-Value). The critical values for both cases are computed according to Durand and Greenwood.

/SAW Saves the translated angle data in the wave W_AngleWave in the current data folder.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results unless you moved the output wave to a different data folder. If the named table exists, but does not display the output wave from the current data folder, the table is renamed and a new table is created.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsCircularMoments is equivalent to **WaveStats** but it applies to circular data, which are distributed on the perimeter of a circle representing some period or cycle. If your data are not described by one of the built-in modes, you can specify the value of the origin (/ORGN), which is mapped to zero degrees and the size of a cycle or period.

When you use Igor date formats with the built-in modes for dates, the default origin is set to zero. The default cycle in the case of Mode 4 is 366. This is done in order to handle both leap and nonleap years. Similarly, Mode 5 uses a cycle of 31 days. Note that the internal conversion from Igor date to (year, month, day) is independent of the cycle specification and is therefore not affected by this choice. You should use the /CYCL flag if you use one of these modes with a fixed size of year or month.

The parameters listed below are computed and displayed (see row labels) in the table. Here *N* is the number of valid (non-NaN) angles $\{\theta_i\}$

$$C=\sum_{i=1}^{n}\cos\theta_i$$

$$S=\sum_{i=1}^{n}\sin\theta_i$$

$$R=\sqrt{C^2+S^2}$$

$$cBar=\overline{C}=C/n$$

$$sBar=\overline{S}=S/n$$

$$rBar=\overline{R}=R/n$$

$$tBar=\overline{\theta}=\begin{cases}\text{atan}(S/C) & S>0,C>0\\ \text{atan}(S/C)+\pi & C<0\\ \text{atan}(S/C)+2\pi & S<0,C>0\end{cases}$$

$$V=1-\overline{R}$$

$$v=\sqrt{-2\log(1-V)}$$

median is the value which minimizes

$$d(\theta)=\pi-\frac{1}{n}\sum_{i=1}^{n}\left|\pi-\left|\theta_i-\overline{\theta}\right|\right|$$

mean deviation = The minimum of the last equation when $\theta \rightarrow$ median.

Higher order moments are denoted with the moment number such that t3Bar is the uncentered third moment of the angle while primed quantities are relative to mean direction tBar. Using this notation

$$\widehat{\rho_2}=\frac{1}{n}\sum_{i=1}^{n}\cos 2\left(\theta_i-\overline{\theta}\right)$$

$$circular\ \ dispersion=\frac{1-\widehat{\rho_2}}{2\overline{R}^2}$$

$$skewness=\frac{\widehat{\rho_2}\sin\left(\hat{\mu}_2'-2\overline{\theta}\right)}{\left(1-\overline{R}\right)^{3/2}}$$

$$kurtosis=\frac{\widehat{\rho_2}\cos\left(\hat{\mu}_2'-2\overline{\theta}\right)-\overline{R}^4}{\left(1-\overline{R}\right)^2}$$

where

$$\hat{\mu}'_p = \begin{cases} \text{atan}\left(S_p/C_p\right) & S_p > 0, C_p > 0 \\ \text{atan}\left(S_p/C_p\right) + \pi & C_p < 0 \\ \text{atan}\left(S_p/C_p\right) + 2\pi & S_p < 0, C_p > 0 \end{cases}$$

and

$$C_p = \frac{1}{n}\sum_{i=1}^{n} \cos p\theta_i, \qquad S_p = \frac{1}{n}\sum_{i=1}^{n} \sin p\theta_i.$$

References

Fisher, N.I., *Statistical Analysis of Circular Data*, 295pp., Cambridge University Press, New York, 1995.

Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

Durand, D., and J.A. Greenwood, Modifications of the Rayleigh test for uniformity in analysis of two-dimensional orientation data, *J. Geol*., *66*, 229-238, 1958.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

WaveStats, **StatsAngularDistanceTest**, **StatsCircularCorrelationTest**, **StatsCircularMeans**, **StatsHodgesAjneTest**, **StatsWatsonUSquaredTest**, **StatsWatsonWilliamsTest**, and **StatsWheelerWatsonTest**.

StatsCircularTwoSampleTest

StatsCircularTwoSampleTest [***flags***] ***waveA, waveB***

The StatsCircularTwoSampleTest operation performs second order analysis of angles. Using the appropriate flags you can choose between parametric or nonparametric, unordered or paired tests. The input consists of two waves that contain one or two columns. The first column contains angle data (mean angles) expressed in radians and an optional second column that contains associated vector lengths. The waves must be either single or double precision. Results are stored in the W_StatsCircularTwoSamples wave in the current data folder and optionally displayed in a table. Some of the tests may have additional outputs.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/NPR	Performs nonparametric paired-sample test (Moore). The input waves must contain paired angular data so both must have single column and the same number of points.
/NSOA	Perform nonparametric second order two-sample test. Input waves must each contain two columns.
/PPR	Performs parametric paired-sample test. Input waves must contain paired data and must have the same number of points.
/PSOA	Performs parametric second order analysis of two samples. The input waves must each contain two columns.
/Q	No information printed in the history area.
/T= *k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores any errors.

Details

The nonparametric paired-sample test (/NPR) is Moore's test for paired angles applied in second order analysis. The input can consist of one or two column waves. When both waves contain a single column the

operation proceeds as if all the vector length were identically 1. The Moore statistic ($H_0 \rightarrow$ pair equality) is computed and compared to the critical value from the Moore distribution (see **StatsInvMooreCDF**).

The nonparametric second-order two-sample test (/NSOA) consists of pre-processing where the grand mean is subtracted from the two inputs followed by application of Watson's U^2 test (**StatsWatsonUSquaredTest**) with H_0 implying that the two samples came from the same population. The results of this test are stored in the wave W_WatsonUtest.

The parametric paired-sample test (/PPR) is due to Hotelling. In this test the input should consist of both angular and vector length data. The test statistic is compared with a critical value from the F distribution (**StatsInvFCDF**).

The parametric second order two-sample test (/PSOA) is an extension of Hotelling one-sample test to second order analysis where an F-like statistic is computed corresponding to H_0 of equal mean angles.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvMooreCDF**, **StatsWatsonUSquaredTest**, and **StatsInvFCDF**.

StatsCMSSDCDF

StatsCMSSDCDF(*C, n*)

The StatsCMSSDCDF function returns the cumulative distribution function of the C distribution (mean square successive difference), which is

$$f(C,n) = \frac{\Gamma(2m+2)}{a2^{2m+1}\left[\Gamma(m+1)\right]^2}\left(1-\frac{C^2}{a^2}\right)^m,$$

where

$$a^2 = \frac{\left(n^2+2n-12\right)\left(n-2\right)}{\left(n^3-13n+24\right)},$$

$$m = \frac{\left(n^4-n^3-13n^2+37n-60\right)}{2\left(n^3-13n+24\right)}.$$

The distribution (C>0) can then be expressed as

$$F(C,n) = \frac{\Gamma(2m+2)}{a2^{2m+1}\left[\Gamma(m+1)\right]^2} C\,{}_2F_1\left(\frac{1}{2},-m,\frac{3}{2},\frac{C^2}{a^2}\right),$$

where ${}_2F_1$ is the hypergeometric function **hyperG2F1**.

References

Young, L.C., On randomness in ordered sequences, *Annals of Mathematical Statistics*, *12*, 153-162, 1941.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCMSSDCDF** and **StatsSRTest**.

StatsCochranTest

StatsCochranTest [***flags***] [***wave1, wave2,… wave100***]

The StatsCochranTest operation performs Cochran's (Q) test on a randomized block or repeated measures dichotomous data. Output is to the M_CochranTestResults wave in the current data folder or optionally to a table.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results unless you moved the output wave to a different data folder. If the named table exists but it does not display the output wave from the current data folder, the table is renamed and a new table is created.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsCochranTest computes Cochran's statistic and compares it to a critical value from a Chi-squared distribution, which depends only of the significance level and the number of groups (columns). The null hypothesis for the test is that all columns represent the same proportion of the effect represented by a non-zero data.

The Chi-square distribution is appropriate when there are at least 4 columns and at least 24 total data points.

Dichotomous data are presumed to consist of two values 0 and 1, thus StatsCochranTest distinguishes only between zero and any nonzero value, which is considered to be 1; it does not allow NaNs or INFs. Input waves can be a single 2D wave or a list of 1D numeric waves, which can also be specified in a string list with /WSTR. In the standard terminology, data rows represent blocks and data columns represent groups. H_0 corresponds to the assumption that all groups have the same proportion of 1's.

With the /T flag, it displays the results in a table that contains the number of rows, the number of columns, the Cochran statistic, the critical value, and the conclusion (1 to accept H_0 and 0 to reject it).

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFriedmanTest**.

StatsContingencyTable

`StatsContingencyTable [`*`flags`*`] srcWave`

The StatsContingencyTable operation performs contingency table analysis on 2D and 3D tables. Output is to the W_ContingencyTableResults wave in the current data folder or optionally to a table or the history area.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/COR=*mode* Sets the correction type for 2x2 tables. By default there is no correction. Use *mode*=1 for Yates and *mode*=2 for Haber correction.

/FEXT={*row*, *col*} Computes Fisher's Exact P-value with 2x2 contingency tables. *row* and *col* are zero-based indices of the table entry where it computes the probability of getting the results in the table or more extreme values. Without the /Q flag, it prints the probabilities of each individual table in the history.

Given the contingency table:

	Succeeded	**Failed**
Group1	11	8
Group2	4	9

Example 1: When you use /FEXT={0,0} the P-value represents the sum of the probabilities of the first group having in the Succeeded column 11 or more extreme values, i.e., 12, 13, 14, and 15. In each case the remaining table elements are adjusted so that row and column sums remain constant.

Example 2: When you needed to evaluate the sum of the probabilities of Group2 having 4 counts or less in the Succeeded column, then the appropriate flag is /FEXT={1,1}, which effectively computes the equivalent of having 9, 10, 11, 12, and 13 Failed counts. In each case it computes the upper, the lower, and the two-tail probabilities.

/HTRG — Tests for heterogeneity between tables stored as layers of 3D wave.

/LLIK — Computes log likelihood statistic.

/Q — No results printed in the history area.

/T=*k* — Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results unless you moved the output wave to a different data folder. If the named table exists but it does not display the output wave from the current data folder, the table is renamed and a new table is created.

/Z — Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsContingencyTable supports 2D waves representing single contingency tables or 3D waves representing multiple 2D tables (where each table is a layer) or a single 3D table. Each entry in the wave must contain a frequency value and must be a positive number; it does not support 0's, NaNs, or INFs. In the special case of 2x2 tables, use the /COR flag to compute the statistic using either the Yates or Haber corrections. Except for the heterogeneity option you can also compute the log likelihood statistic. In all the tests, H_0 corresponds to independence between the tested variables.

For 3D tables StatsContingencyTable provides Chi-squared, degrees of freedom, the critical value, and optionally the log likelihood G statistic (/LLIK flag) for each of the following cases:

1. Mutual independence by testing if all three variables are independent of each other.
2. Partial dependence (rows) by testing if rows independent of columns and layers.
3. Partial dependence (columns) by testing if columns independent of rows and layers.
4. Partial dependence (layers) by testing if layers independent of rows and columns.

In each case you should compare the statistic with the critical value and reject H_0 if the statistic exceeds or equals the critical value.

You should examine the table entries to determine if the Chi-square statistic is appropriate (if the frequency is smaller than 6 for /ALPH=0.05 you should consider computing the Fisher exact test).

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvChiCDF**.

StatsCorrelation

StatsCorrelation(*waveA* [, *waveB*])

The StatsCorrelation function computes Pearson's correlation coefficient between two real valued arrays of data of the same length. Pearson r is give by:

$$r = \frac{\sum_{i=0}^{n-1}\left(waveA[i]-A\right)\left(waveB[i]-B\right)}{\sqrt{\sum_{i=0}^{n-1}\left(waveA[i]-A\right)^2\sum_{i=0}^{n-1}\left(waveB[i]-B\right)^2}}$$

Here *A* is the average of the elements in *waveA*, *B* is the average of the elements of *waveB* and the sum is over all wave elements.

Details

If you use both *waveA* and *waveB* then the two waves must have the same number of points but they could be of different number type. If you use only the *waveA* parameter then *waveA* must be a 2D wave. In this case StatsCorrelation will return 0 and create a 2D wave M_Pearson where the (*i*,*j*) element is Pearson's r corresponding to columns *i* and *j*.

Fisher's z transformation converts Person's r above to a normally distributed variable *z*:

$$z = \frac{1}{2}\ln\left(\frac{1+r}{1-r}\right),$$

with a standard error

$$\sigma_z = \frac{1}{\sqrt{n-3}}.$$

You can convert between the two representations using the following functions:

```
Function pearsonToFisher(inr)
   Variable inr
   return 0.5*(ln(1+inr)-ln(1-inr))
End
Function fisherToPearson(inz)
   Variable inz
   return tanh(inz)
End
```

See Also

Correlate, **StatsLinearCorrelationTest**, and **StatsCircularCorrelationTest**.

StatsDExpCDF

StatsDExpCDF(x, m, s)

The StatsDExpCDF function returns the double-exponential cumulative distribution function

$$F(x;\mu,\sigma) = \begin{cases} \exp\left(\frac{x-\mu}{\sigma}\right) & when\ x < \mu \\ 1-\frac{1}{2}\exp\left(-\left|\frac{x-\mu}{\sigma}\right|\right) & when\ x \ge \mu \end{cases}$$

for σ>0. It returns NaN when σ=0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsDExpPDF** and **StatsInvDExpCDF**.

StatsDExpPDF

StatsDExpPdf(*x, m, s*)

The StatsDExpPdf function returns the double-exponential probability distribution function

$$f(x;\mu,\sigma)=\frac{1}{2\sigma}\exp\left[-\left|\frac{x-\mu}{\sigma}\right|\right],$$

where μ is the location parameter and σ>0 is the scale parameter. Use μ=0 and σ=1 for the standard form of the double exponential distribution. It returns NaN when σ=0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsDExpCDF** and **StatsInvDExpCDF**.

StatsDIPTest

StatsDIPTest [/Z] *srcWave*

The StatsDIPTest operation performs Hartigan test for unimodality.

Flags

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The input to the operation *srcWave* is any real numeric wave. Outputs are: V_Value contains the dip statistic; V_min is the lower end of the modal interval; and V_max is the higher end of the modal interval. Percentage points or critical values for the dip statistic can be obtained from simulations using an identical sample size as in this example:

```
Function getCriticalValue(sampleSize,alpha)
Variable sampleSize,alpha

   Make/O/N=(sampleSize) dataWave
   Make/O/N=100000  dipResults
   Variable i
   for(i=0;i<100000;i+=1)
      dataWave=enoise(100)
      StatsDipTest dataWave
      dipResults[i]=V_Value
   endfor
   Histogram/P/B=4 dipResults              // Compute the PDF.
   Wave W_Histogram
   Integrate/METH=1 W_Histogram/D=W_INT    // Compute the CDF.
   Findlevel/Q  W_int,(1-alpha)            // Find the critical value.
   return V_LevelX
End
```

References

Hartigan, P. M., Computation of the Dip Statistic to Test for Unimodality, *Applied Statistics, 34*, 320-325, 1985.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsDunnettTest

StatsDunnettTest [***flags***] [***wave1, wave2,… wave100***]

The StatsDunnettTest operation performs the Dunnett test by comparing multiple groups to a control group. Output is to the M_DunnettTestResults wave in the current data folder or optionally to a table. StatsDunnettTest usually follows StatsANOVA1Test.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/CIDX=*cIndex*	Specifies the (zero based) index of the input wave corresponding to the control group. The default is zero (the first wave corresponds to the control group).
/Q	No results printed in the history area.

/SWN — Creates a text wave, T_DunnettDescriptors, containing wave names corresponding to each row of the comparison table (Save Wave Names). Use /T to append the text wave to the last column.

/T=*k* — Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/TAIL=*tc* — Specifies H_0.

tc=1: Default; one tailed test ($\mu_c \le \mu_a$).

tc=2: One tailed test ($\mu_c \ge \mu_a$).

tc=4: Two tailed test ($\mu_c = \mu_a$).

Code combinations are not allowed.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z — Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsDunnettTest inputs are two or more 1D numeric waves (one wave for each group of samples). The input waves may contain different number of points, but they must contain two or more valid entries per wave.

For output to a table (using /T), each labelled row represents the results of the test for comparing the means of one group to the control group, and rows are ordered so that all comparisons are computed sequentially starting with the group having the smallest mean. The contents of the labeled columns are:

First	The difference between the group means
Second	SE (which is computed for possibly unequal number of points)
Third	The q statistic for the pair which may be positive or negative
Fourth	The critical q' value
Fifth	0 if the conclusion is to reject H_0 or 1 to accept H_0
Sixth	The P-value

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsTukeyTest**, **StatsANOVA1Test**, **StatsScheffeTest**, and **StatsNPMCTest**.

StatsErlangCDF

StatsErlangCDF(*x*, *b*, *c*)

The StatsErlangCDF function returns the Erlang cumulative distribution function

$$F(x;b,c) = 1 - \frac{\Gamma\left(c, \frac{x}{b}\right)}{\Gamma(c)}.$$

where $b>0$ (also as $\lambda=1/b$) is the scale parameter, $c>0$ the shape parameter, $\Gamma(x)$ the **gamma** function, and $\Gamma(a,x)$ the incomplete gamma function **gammaInc**.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsErlangPDF**.

StatsErlangPDF

StatsErlangPDF(*x*, *b*, *c*)

The StatsErlangPDF function returns the Erlang probability distribution function

$$f(x;b,c)=\frac{\left(\frac{x}{b}\right)^{c-1}\exp\left(-\frac{x}{b}\right)}{b(c-1)!}.$$

where $b>0$ (also as $\lambda=1/b$) is the scale parameter and $c>0$ the shape parameter.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsErlangCDF**.

StatsErrorPDF

StatsErrorPDF(*x*, *a*, *b*, *c*)

The StatsErrorPDF function returns the error probability distribution function or the exponential power distribution

$$f(x;a,b,c)=\frac{\exp\left[-\frac{1}{2}\left(\frac{|x-a|}{b}\right)^{\frac{2}{c}}\right]}{b2^{\frac{c}{2}+1}\Gamma\left(1+\frac{c}{2}\right)}.$$

where a is the location parameter, $b>0$ is the scale parameter, $c>0$ is the shape parameter, and $\Gamma(x)$ is the **gamma** function.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsEValueCDF

StatsEValueCDF(*x*, μ, σ)

The StatsEValueCDF function returns the extreme-value (type I, Gumbel) cumulative distribution function

$$F(x;\mu,\sigma)=1-\exp\left(-\exp\left(\frac{x-\mu}{\sigma}\right)\right),$$

where $\sigma>0$. This is also known as the "minimum" form or distribution of the smallest extreme. To obtain the distribution of the largest extreme reverse the sign of σ.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsEValuePDF** and **StatsInvEValueCDF**.

StatsEValuePDF

StatsEValuePDF(*x*, μ, σ)

The StatsEValuePDF function returns the extreme-value (type I, Gumbel) probability distribution function

$$F(x;\mu,\sigma)=1-\exp\left(-\exp\left(\frac{x-\mu}{\sigma}\right)\right),$$

where σ>0. This is also known as the "minimum" form or the distribution of the smallest extreme. To obtain the distribution of the largest extreme reverse the sign of σ.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsEValueCDF** and **StatsInvEValueCDF**.

StatsExpCDF

StatsExpCDF(*x*, μ, σ)

The StatsExpCDF function returns the exponential cumulative distribution function

$$F(x;\mu,\sigma) = 1 - \exp\left(-\frac{x-\mu}{\sigma}\right),$$

where $x \geq \mu$ and $\sigma > 0$. It returns NaN for $\sigma = 0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsExpPDF** and **StatsInvExpCDF**.

StatsExpPDF

StatsExpPDF(*x*, μ, σ)

The StatsExpPDF function returns the exponential probability distribution function

$$f(x;\mu,\sigma) = \frac{1}{\sigma}\exp\left(-\frac{x-\mu}{\sigma}\right),$$

where μ is the location parameter and σ>0 is the scale parameter. Use μ=0 and σ=1 for the standard form of the exponential distribution. It returns NaN for σ=0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsExpCDF** and **StatsInvExpCDF**.

StatsFCDF

StatsFCDF(*x*, *n1*, *n2*)

The StatsFCDF function returns the cumulative distribution function for the F distribution with shape parameters *n1* and *n2*

$$F(x;n_1,n_2) = 1 - Betai\left(\frac{n_2}{2},\frac{n_1}{2},\frac{n_2}{n_2+n_1x}\right),$$

where *Betai* is the incomplete beta function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFPDF** and **StatsInvFCDF**.

StatsFPDF

StatsFPDF(*x*, *n1*, *n2*)

The StatsFPDF function returns the probability distribution function for the F distribution with shape parameters *n1* and *n2*

$$f(x;n_1,n_2)=\frac{\Gamma\left(\frac{n_1+n_2}{2}\right)\left(\frac{n_1}{n_2}\right)^{\frac{n_1}{2}}x^{\frac{n_1}{2}-1}}{\Gamma\left(\frac{n_1}{2}\right)\Gamma\left(\frac{n_2}{2}\right)\left(1+\frac{n_1x}{n_2}\right)^{\frac{n_1+n_2}{2}}}.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFCDF** and **StatsInvFCDF**.

StatsFriedmanCDF

StatsFriedmanCDF(*x, n, m, method, useTable*)

The StatsFriedmanCDF function returns the cumulative probability distribution of the Friedman distribution with *n* rows and *m* columns. The exact Friedman distribution is computationally intensive, taking on the order of $(n!)^m$ iterations. You may be able to use a range of precomputed exact values by passing a nonzero value for *useTable*, which will use *method* only if the value is not in the table. For large *m*, consider using the Chi-squared or the Monte-Carlo approximations. To abort execution, press Command-period (*Macintosh*) or Ctrl+Break (*Windows*).

method	What It Does
0	Exact computation.
1	Chi-square approximation.
2	Monte-Carlo approximation.
3	Use built-table only and return NaN if not in table.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvFriedmanCDF** and **StatsFriedmanTest**.

StatsFriedmanTest

StatsFriedmanTest [***flags***] [***wave1, wave2,… wave100***]

The StatsFriedmanTest operation performs Friedman's test on a randomized block of data. It is a nonparametric analysis of data contained in either individual 1D waves or in a single 2D wave. Output is to the M_FriedmanTestResults wave in the current data folder or optionally to a table.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/Q No results printed in the history area.

/RW Saves the ranking wave M_FriedmanRanks, which contains the rank values corresponding to each input datum.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The Friedman test ranks the input data on a row-by-row basis, sums the ranks for each column, and computes the Friedman statistic, which is proportional to the sum of the squares of the ranks.

Input waves can be a single 2D wave or a list of 1D numeric waves, which can also be specified in a string list with /WSTR. All 1D waves must have the same number of points. A 2D wave must not contain any NaNs.

The critical value for the Friedman distribution is fairly difficult to compute when the number of rows and columns is large because it requires a number of permutations on the order of (numColumns!)^numRows. A certain range of these critical values are supported by precomputed tables. When the exact critical value is not available you can use one of the two approximations that are always computed: the Chi-squared approximation or the Iman and Davenport approximation, which converts the Friedman statistic is converted to a new value Ff then compares it with critical values from the F distribution using weighted degrees of freedom.

With the /T flag, it displays the results in a table that contains the number of rows, the number of columns, the Friedman statistic, the exact critical value (if available), the Chi-squared approximation, the Iman and Davenport approximation, and the conclusion (1 to accept H_0 and 0 to reject it).

V_flag will be set to -1 for any error and to zero otherwise.

References

Iman, R.L., and J.M. Davenport, Approximations of the critical region of the Friedman statistic, *Comm. Statist*. *A9*, 571-595, 1980.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFriedmanCDF** and **StatsInvFriedmanCDF**.

StatsFTest

StatsFTest [***flags***] ***wave1, wave2***

The StatsFTest operation performs the F-test on the two distributions in *wave1* and *wave2*, which can be any real numeric type, must contain at least two data points each, and can have an arbitrary number of dimensions. Output is to the W_StatsFTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed.
	k=0: Normal with dialog (default).
	k=1: Kills with no dialog.
	k=2: Disables killing.
/TAIL=*tc*	Specifies the tail tested.
	tc=1: Lower one-tail test with H_a: sigma1>sigma2.
	tc=2: Upper one-tail test with H_a: sigma1<sigma2.
	tc=4: Default; the null hypothesis H_0: sigma1=sigma2 with H_a: sigma1!=sigma2.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The F statistic is the ratio of the variance of *wave1* to the variance of *wave2*. We assume the waves have equal wave variances and that H_0 is sigma1=sigma2. For the upper one-tail test we reject H_0 if F is greater than the upper critical value or if F is smaller than the lower critical value in the lower one-tail test. In the two-tailed test we reject H_0 if F is either greater than the upper critical value or smaller than the lower critical value. The critical values are computed by numerically solving for the argument at which the cumulative distribution function (CDF) equals the appropriate values for the tests. The CDF is given by

$$F(x,n_1,n_2) = 1 - betai\left(\frac{n_2}{2},\frac{n_1}{2},\frac{n_2}{n_2+n_1x}\right),$$

where the degrees of freedom n_1 and n_2 equal the number of valid (non-NaN) points in each wave -1, and *betai* is the incomplete beta function. To get the critical value for the upper one-tail test we solve F(x)=1-alpha. For the lower one-tail test we solve F(x)=alpha. In the two-tailed test the lower critical value is a solution for F(x)=alpha/2 and the upper critical value is a solution for F(x)=1-alpha/2.

The F-test requires that the two samples are from normally distributed populations.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsVariancesTest**, **StatsFCDF**, and **betai**.

StatsGammaCDF

StatsGammaCDF(*x*, μ, σ, γ)

The StatsGammaCDF function returns the gamma cumulative distribution function

$$F(x;\mu,\sigma,\gamma) = \frac{\Gamma_{inc}\left(\gamma, \frac{x-\mu}{\sigma}\right)}{\Gamma(\gamma)}. \qquad \begin{matrix} x \geq \mu \\ \sigma,\gamma > 0 \end{matrix}$$

where Γ is the gamma function and Γ_{inc} is the incomplete gamma function **gammaInc**.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGammaPDF** and **StatsInvGammaCDF**.

StatsGammaPDF

StatsGammaPDF(*x*, μ, σ, γ)

The StatsGammaPDF function returns the gamma probability distribution function

$$f(x;\mu,\sigma,\gamma) = \frac{\left(\frac{x-\mu}{\sigma}\right)^{\gamma-1} \exp\left(-\frac{x-\mu}{\sigma}\right)}{\sigma\Gamma(\gamma)}. \qquad \begin{matrix} x \geq \mu \\ \sigma,\gamma > 0 \end{matrix}$$

where μ is the location parameter, σ is the scale parameter, γ is the shape parameter, and Γ is the gamma function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGammaCDF** and **StatsInvGammaCDF**.

StatsGeometricCDF

StatsGeometricCDF(*x*, *p*)

The StatsGeometricCDF function returns the geometric cumulative distribution function

$$F(x,p) = 1-(1-p)^{x+1}.$$

where *p* is the probability of success in a single trial and *x* is the number of trials for $x \geq 0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGeometricPDF** and **StatsInvGeometricCDF**.

StatsGeometricPDF

StatsGeometricPDF(*x*, *p*)

The StatsGeometricPDF function returns the geometric probability distribution function

$$f(x,p) = p(1-p)^x,$$

where the p is the probability of success in a single trial and x is the number of trials $x \geq 0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGeometricCDF** and **StatsInvGeometricCDF**.

StatsHodgesAjneTest

StatsHodgesAjneTest [*flags*] *srcWave*

The StatsHodgesAjneTest operation performs the Hodges-Ajne nonparametric test for uniform distribution around a circle. Output is to the W_HodgesAjne wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/SA=*specAngle*	Uses the Batschelet modification of the Hodges-Ajne test to test for uniformity against the alternative of concentration around the specified angle. *specAngle* must be expressed in radians modulus 2π.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The input *srcWave* must contain angles in radians, can be any number of dimensions, can be single or double precision, and should not contain NaNs or INFs.

StatsHodgesAjneTest performs the standard Hodges-Ajne test, which simply tests for uniformity against the hypothesis that the population is not uniformly distributed around the circle. This test finds a diameter that divides the circle into two halves such that one contains the least number of data *m*, the test statistic.

Use /SA to perform the modified (Batschelet) test, which tests against the alternative that the population is concentrated somehow about the specified angle. The modified test counts the number of points *m*' in 90-degree neighborhoods around the specified angle. The test statistic is given by C=*n*-*m*' where *n* is the number of points in the wave. The critical value is computed from the binomial probability density.

In both cases H_0 is rejected if the statistic is smaller than the critical value.

V_flag will be set to -1 for any error and to zero otherwise.

References

Ajne, B., A simple test for uniformity of a circular distribution, *Biometrica*, *55*, 343-354, 1968.

See, in particular, Chapter 27 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsCircularMeans, **StatsCircularMoments**, **StatsWatsonUSquaredTest**, **StatsWatsonWilliamsTest**, and **StatsWheelerWatsonTest**.

StatsHyperGCDF

StatsHyperGCDF(*x*, *m*, *n*, *k*)

The StatsHyperGCDF function returns the hypergeometric cumulative distribution function, which is the probability of getting *x* marked items when drawing (without replacement) *k* items out of a population of *m* items when *n* out of the *m* are marked.

Details

The hypergeometric distribution is

$$F(x;m,n,k) = \sum_{L=0}^{x} \frac{\binom{n}{L}\binom{m-L}{k-L}}{\binom{m}{k}},$$

where $\binom{a}{b}$ is the **binomial** function. All parameters must be positive integers and must have $m>n$ and $x<k$; otherwise it returns NaN.

References

Klotz, J.H., *Computational Approach to Statistics*, <http://www.stat.wisc.edu/~klotz/Book.pdf>.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsHyperGPDF**.

StatsHyperGPDF

StatsHyperGPDF(*x, m, n, k*)

The StatsHyperGPDF function returns the hypergeometric probability distribution function, which is the probability of getting *x* marked items when drawing without replacement *k* items out of a population of *m* items where *n* out of the *m* are marked.

Details

The hypergeometric distribution is

$$f(x;m,n,k) = \frac{\binom{n}{x}\binom{m-n}{k-x}}{\binom{m}{k}},$$

where $\binom{a}{b}$ is the **binomial** function. All parameters must be positive integers and must have $m>n$ and $x<k$.

References

Klotz, J.H., *Computational Approach to Statistics*, <http://www.stat.wisc.edu/~klotz/Book.pdf>.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsHyperGCDF**.

StatsInvBetaCDF

StatsInvBetaCDF(*cdf, p, q* [*, a, b*])

The StatsInvBetaCDF function returns the inverse of the beta cumulative distribution function. There is no closed form expression for the inverse beta CDF; it is evaluated numerically.

The defaults (*a*=0 and *b*=1) correspond to the standard beta distribution.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBetaCDF** and **StatsBetaPDF**.

StatsInvBinomialCDF

StatsInvBinomialCDF(*cdf, p, N*)

The StatsInvBinomialCDF function returns the inverse of the binomial cumulative distribution function. The inverse function returns the value at which the binomial *CDF* with probability *p* and total elements *N*,

has the value 0.95. There is no closed form expression for the inverse binomial CDF; it is evaluated numerically.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBinomialCDF** and **StatsBinomialPDF**.

StatsInvCauchyCDF

StatsInvCauchyCDF(*cdf*, μ, σ)

The StatsInvCauchyCDF function returns the inverse of the Cauchy-Lorentz cumulative distribution function

$$x = \mu + \sigma \tan\left[\pi\left(cdf - \frac{1}{2}\right)\right].$$

It returns NaN for *cdf* <0 or *cdf*> 1.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCauchyCDF** and **StatsCauchyPDF**.

StatsInvChiCDF

StatsInvChiCDF(*x*, *n*)

The StatsInvChiCDF function returns the inverse of the chi-squared distribution of *x* and shape parameter *n*. The inverse of the distribution is also known as the percent point function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsChiCDF** and **StatsChiPDF**.

StatsInvCMSSDCDF

StatsInvCMSSDCDF(*cdf*, *n*)

The StatsInvCMSSDCDF function returns the critical values of the C distribution (mean square successive difference distribution), which is given by

$$f(C,n) = \frac{\Gamma(2m+2)}{a 2^{2m+1}\left[\Gamma(m+1)\right]^2}\left(1 - \frac{C^2}{a^2}\right)^m,$$

where

$$a^2 = \frac{\left(n^2 + 2n - 12\right)(n-2)}{\left(n^3 - 13n + 24\right)},$$

$$m = \frac{\left(n^4 - n^3 - 13n^2 + 37n - 60\right)}{2\left(n^3 - 13n + 24\right)}.$$

Critical values are computed from the integral of the probability distribution function.

References

Young, L.C., On randomness in ordered sequences, *Annals of Mathematical Statistics*, *12*, 153-162, 1941.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCMSSDCDF** and **StatsSRTest**.

StatsInvDExpCDF

StatsInvDExpCDF(*cdf*, μ, σ)

The StatsInvDExpCDF function returns the inverse of the double-exponential cumulative distribution function

$$x = \begin{cases} \mu + \sigma \ln(2cdf) & \text{when } cdf < 0.5 \\ \mu - \sigma \ln\left[2(1-cdf)\right] & \text{when } cdf \geq 0.5 \end{cases}$$

It returns NaN for *cdf* <0 or *cdf* > 1.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsDExpCDF** and **StatsDExpPDF**.

StatsInvEValueCDF

StatsInvEValueCDF(*cdf*, μ, σ)

The StatsInvEValueCDF function returns the inverse of the extreme-value (type I, Gumbel) cumulative distribution function

$$x = \mu - \sigma \ln(1 - cdf)$$

where σ>0. It returns NaN for *cdf*<0 or *cdf*>1. This inverse applies to the "minimum" form of the distribution. Reverse the sign of σ to obtain the inverse distribution of the maximum form.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsEValueCDF** and **StatsEValuePDF**.

StatsInvExpCDF

StatsInvExpCDF(*cdf*, μ, σ)

The StatsInvExpCDF function returns the inverse of the exponential cumulative distribution function

$$x = \mu - \sigma \ln(1 - cdf).$$

It returns NaN for *cdf* <0 or *cdf* > 1.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsExpCDF** and **StatsExpPDF**.

StatsInvFCDF

StatsInvFCDF(*x*, *n1*, *n2*)

The StatsInvFCDF function returns the inverse of the F distribution cumulative distribution function for *x* and shape parameters *n1* and *n2*. The inverse is also known as the percent point function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFCDF** and **StatsFPDF**.

StatsInvFriedmanCDF

StatsInvFriedmanCDF(*cdf*, *n*, *m*, *method*, *useTable*)

The StatsInvFriedmanCDF function returns the inverse of the Friedman distribution cumulative distribution function of *cdf* with *n* rows and *m* columns. Use this typically to compute the critical values of the distribution

```
Print StatsInvFriedmanCDF(1-alpha,n,m,0,1)
```

where alpha is the significance level of the associated test.

The complexity of the computation of Friedman CDF is on the order of $(n!)^m$. For nonzero values of *useTable*, searches are limited to the built-in table for distribution values. If *n* and *m* are not in the table the calculation may still proceed according to the *method*.

method	What It Does
0	Exact computation(slow, not recommended).
1	Chi-square approximation.
2	Monte-Carlo approximation (slow).
3	Use built-in table only and return a NaN if not in table.

For large *m* and *n,* consider using the Chi-squared or the Iman and Davenport approximations. To abort execution, press Command-period (*Macintosh*) or Ctrl+Break (*Windows*).

Note: Table values are different from computed values for both methods. Table values use more conservative criteria than computed values. Table values are more consistent with published values because the Friedman distribution is a highly irregular function with multiple steps of arbitrary sizes. The standard for published tables provides the X value of the next vertical transition to the one on which the specified P is found.

Precomputed tables use these values:

n	*m*
3	2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
4	2, 3, 4, 5, 6, 7, 8, 9
5	2, 3, 4, 5, 6
6	2, 3, 4, 5
7	2, 3, 4
8	2, 3
9	2, 3

References

Iman, R.L., and J.M. Davenport, Approximations of the critical region of the Friedman statistic, *Comm. Statist*., *A9*, 571-595, 1980.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFriedmanCDF** and **StatsFriedmanTest**.

StatsInvGammaCDF

StatsInvGammaCDF(*cdf*, μ, σ, γ)

The StatsInvGammaCDF function returns the inverse of the gamma cumulative distribution function. There is no closed form expression for the inverse gamma distribution; it is evaluated numerically.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGammaCDF** and **StatsGammaPDF**.

StatsInvGeometricCDF

StatsInvGeometricCDF(*cdf, p*)

The StatsInvGeometricCDF function returns the inverse of the geometric cumulative distribution function

$$x = \frac{\ln(1 - cdf)}{\ln(1 - p)} - 1$$

where *p* is the probability of success in a single trial and *x* is the number of trials.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGeometricCDF** and **StatsGeometricPDF**.

StatsInvKuiperCDF

StatsInvKuiperCDF(*cdf*)

The StatsInvKuiperCDF function returns the inverse of Kuiper cumulative distribution function.

There is no closed form expression. It is mapped to the range of 0.4 to 4, with accuracy of 1e-10.

References

See in particular Section 14.3 of

Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsKuiperCDF**.

StatsInvLogisticCDF

StatsInvLogisticCDF(*cdf*, *a*, *b*)

The StatsInvLogisticCDF function returns the inverse of the logistic cumulative distribution function

$$x = a + b\log\left(\frac{cdf}{1-cdf}\right).$$

where the scale parameter b>0 and the shape parameter is a.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogisticCDF** and **StatsLogisticPDF** functions.

StatsInvLogNormalCDF

StatsInvLogNormalCDF(*cdf*, *sigma*, *theta*, *mu*)

The StatsInvLogNormalCDF function returns the numerically evaluated inverse of the lognormal cumulative distribution function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogNormalCDF** and **StatsLogNormalPDF** functions.

StatsInvMaxwellCDF

StatsInvMaxwellCDF(*cdf*, *k*)

The StatsInvMaxwellCDF function returns the evaluated numerically inverse of the Maxwell cumulative distribution function. There is no closed form expression.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsMaxwellCDF** and **StatsMaxwellPDF** functions.

StatsInvMooreCDF

StatsInvMooreCDF(*cdf*, *N*)

The StatsInvMooreCDF function returns the inverse cumulative distribution function for Moore's R*, which is used as a critical value in nonparametric version of the Rayleigh test for uniform distribution around the circle. It supports the range $3 \le N \le 120$ and does not change appreciably for $N > 120$.

The inverse distribution is computed from polynomial approximations derived from simulations and should be accurate to approximately three significant digits.

References

Moore, B.R., A modification of the Rayleigh test for vector data, *Biometrica*, *67*, 175-180, 1980.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsCircularMeans** function.

StatsInvNBinomialCDF

StatsInvNBinomialCDF(*cdf*, *k*, *p*)

The StatsInvNBinomialCDF function returns the numerically evaluated inverse of the negative binomial cumulative distribution function. There is no closed form expression.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNBinomialCDF** and **StatsNBinomialPDF** functions.

StatsInvNCChiCDF

StatsInvNCChiCDF(*cdf*, *n*, *d*)

The StatsInvNCChiCDF function returns the inverse of the noncenteral chi-squared cumulative distribution function. It is computationally intensive because the inverse is computed numerically and involves multiple evaluations of the noncentral distribution, which is evaluated from a series expansion.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCChiCDF**, **StatsNCChiPDF**, **StatsChiCDF**, and **StatsChiPDF** functions.

StatsInvNCFCDF

StatsInvNCFCDF(*cdf*, *n1*, *n2*, *d*)

The StatsInvNCFCDF function returns the numerically evaluated inverse of the cumulative distribution function of the noncentral F distribution. *n1* and *n2* are the shape parameters and *d* is the noncentrality measure. There is no closed form expression for the inverse.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCFCDF** and **StatsNCFPDF** functions.

StatsInvNormalCDF

StatsInvNormalCDF(*cdf*, *m*, *s*)

The StatsInvNormalCDF function returns the numerically computed inverse of the normal cumulative distribution function. There is no closed form expression.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNormalCDF** and **StatsNormalPDF** functions.

StatsInvParetoCDF

StatsInvParetoCDF(*cdf*, *a*, *c*)

The StatsInvParetoCDF function returns the inverse of the Pareto cumulative distribution function

$$x = \frac{a}{\left(1 - cdf\right)^{(1/c)}}$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsParetoCDF** and **StatsParetoPDF** functions.

StatsInvPoissonCDF

StatsInvPoissonCDF(*cdf*, λ)

The StatsInvPoissonCDF function returns the numerically evaluated inverse of the Poisson cumulative distribution function. There is no closed form expression for the inverse Poisson distribution.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPoissonCDF** and **StatsPoissonPDF** functions.

StatsInvPowerCDF

StatsInvPowerCDF(*cdf, b, c*)

The StatsInvPowerCDF function returns the inverse of the Power Function cumulative distribution function

$$x = b / cdf^{(1/c)}.$$

where the scale parameter *b* and the shape parameter *c* satisfy *b*,*c*>0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPowerCDF**, **StatsPowerPDF** and **StatsPowerNoise** functions.

StatsInvQCDF

StatsInvQCDF(*cdf, r, c, df*)

The StatsInvQCDF function returns the critical value of the Q cumulative distribution function for *r* the number of groups, *c* the number of treatments, and *df* the error degrees of freedom (*df*=*r***c**(*n*-1) with sample size *n*).

Details

The Q distribution is the maximum of several Studentized range statistics. For a simple Tukey test, use *r*=1.

Examples

The critical value for a Tukey test comparing 5 treatments with 6 samples and 0.05 significance is:

```
Print StatsInvQCDF(1-0.05,1,5,5*(6-1))
```

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTukeyTest** function.

StatsInvQpCDF

StatsInvQpCDF(*ng, nt, df, alpha, side, sSizeWave*)

The StatsInvQpCDF function returns the critical value of the Q' cumulative distribution function for *ng* the number of groups, *nt* the number of treatments, and *df* the error degrees of freedom. *side*=1 for upper-tail or *side*=2 for two-tailed critical values.

sSizeWave is an integer wave of *ng* columns and *nt* rows specifying the number of samples in each treatment. If *sSizeWave* is a null wave (`$""`) StatsInvQpCDF computes the number of samples from *df*=*ng***nt**(*n*-1) with *n* truncated to an integer.

Details

StatsInvQpCDF is a modified Q distribution typically used with Dunnett's test, which compares the various means with the mean of the control group or treatment.

StatsInvQpCDF differs from other StatsInv*XXX* functions in that you do not specify a *cdf* value for the inverse (usually 1-*alpha* for the critical value). Here *alpha* selects one- or two-tailed critical values.

It is computationally intensive, taking longer to execute for smaller *alpha* values.

Examples

The critical value for a Dunnett test comparing 4 treatments with 4 samples and (upper tail) 0.05 significance is:

```
// here n=4 because 12=1*4*(4-1).
Print StatsInvQpCDF(1,4,12,0.05,1,$"")
  2.28734
```

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsDunnettTest** and **StatsInvQCDF** functions.

StatsInvRayleighCDF

StatsInvRayleighCDF(*cdf* [, *s* [, *m*]])

The **StatsInvRayleighCDF** function returns the inverse of the Rayleigh cumulative distribution functiongiven by

$$x = \mu + \sigma\sqrt{-2\ln(1-cdf)},$$

with defaults *s*=1 and *m*=0. It returns NaN for $s \le 0$ and zero for $x \le m$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRayleighCDF** and **StatsRayleighPDF** functions.

StatsInvRectangularCDF

StatsInvRectangularCDF(*cdf, a, b*)

The StatsInvRectangularCDF function returns the inverse of the rectangular (uniform) cumulative distribution function

$$x = a + cdf(b-a), \qquad a < b.$$

where $a < b$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRectangularCDF** and **StatsRectangularPDF** functions.

StatsInvSpearmanCDF

StatsInvSpearmanCDF(*cdf, N*)

The StatsInvSpearmanCDF function returns the inverse cumulative distribution function for Spearman's *r*, which is used as a critical value in rank correlation tests.

The inverse distribution is computed by finding the value of *r* for which it attains the *cdf* value. The result is usually lower than in published tables, which are more conservative when the first derivative of the distribution is discontinuous.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRankCorrelationTest**, **StatsSpearmanRhoCDF**, and **StatsKendallTauTest** functions.

StatsInvStudentCDF

StatsInvStudentCDF(*cdf, n*)

The StatsInvStudentCDF function returns the numerically evaluated inverse of Student cumulative distribution function. There is no closed form expression.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentCDF** and **StatsStudentPDF** functions.

StatsInvTopDownCDF

StatsInvTopDownCDF(cdf, N)

The StatsInvTopDownCDF function returns the inverse cumulative distribution function for the top-down distribution. For $3 \le N \le 7$ it uses a lookup table CDF and returns the next higher value of *r* for which the distribution value is larger than *cdf*. For $8 \le N \le 50$ it returns the nearest value for which the built-in distribution returns *cdf*. For $N > 50$ it returns the scaled normal approximation.

Tabulated values are from Iman and Conover who pick as the critical value the very first transition of the distribution following the specified *cdf* value. These tabulated values tend to be slightly higher than calculated values for 7<*N*<15.

References

Iman, R.L., and W.J. Conover, A measure of top-down correlation, *Technometrics*, *29*, 351-357, 1987.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRankCorrelationTest** and **StatsTopDownCDF** functions.

StatsInvTriangularCDF

StatsInvTriangularCDF(*cdf, a, b, c*)

The StatsInvTriangularCDF function returns the inverse of the triangular cumulative distribution function

$$x = \begin{cases} a + \sqrt{cdf(b-a)(c-a)} & 0 \le cdf \le \dfrac{c-a}{b-a} \\ b - \sqrt{(1-cdf)(b-a)(b-c)} & \dfrac{c-a}{b-a} \le cdf \le 1 \end{cases}$$

where *a*<*c*<*b*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTriangularCDF** and **StatsTriangularPDF** functions.

StatsInvUSquaredCDF

StatsInvUSquaredCDF(*cdf, n, m, method, useTable*)

The StatsInvUSquaredCDF function returns the inverse of Watson's U^2 cumulative distribution function integer sample sizes *n* and *m*. Use a nonzero value for *useTable* to search a built-in table of values. If *n* and *m* cannot be found in the table, it will proceed according to *method*:

method	**What It Does**
0	Exact computation using Burr algorithm (could be slow).
1	Tiku approximation using chi-squared.
2	Use built-in table only and return a NaN if not in table.

For large *n* and *m*, consider using the Tiku approximation. To abort execution, press Command-period (*Macintosh*) or Ctrl+Break (*Windows*). Because *n* and *m* are interchangeable, *n* should always be the smaller value. For *n*>8 the upper limit in the table matched the maximum that can be computed using the Burr algorithm. There is no point in using method 0 with *m* values exceeding these limits.

The inverse is obtained from precomputed tables of Watson's U^2 (see **StatsUSquaredCDF**).

Note: Table values are different from computed values. These values use more conservative criteria than computed values. Table values are more consistent with published values because the U^2 distribution is a highly irregular function with multiple steps of arbitrary sizes. The standard for published tables provides the X value of the next vertical transition to the one on which the specified P is found. See **StatsInvFriedmanCDF**.

References

Burr, E.J., Small sample distributions of the two sample Cramer-von Mises' W^2 and Watson's U^2, *Ann. Mah. Stat. Assoc.*, *64*, 1091-1098, 1964.

Tiku, M.L., Chi-square approximations for the distributions of goodness-of-fit statistics, *Biometrica*, *52*, 630-633, 1965.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWatsonUSquaredTest** and **StatsUSquaredCDF** functions.

StatsInvVonMisesCDF

StatsInvVonMisesCDF(*cdf*, *a*, *b*)

The StatsInvVonMisesCDF function returns the numerically evaluated inverse of the von Mises cumulative distribution function where the value of the integral of the distribution matches *cdf*. Parameters are as for **StatsVonMisesCDF**.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsVonMisesPDF** and **StatsVonMisesNoise** functions.

StatsInvWeibullCDF

StatsInvWeibullCDF(*cdf*, *m*, *s*, *g*)

The StatsInvWeibullCDF function returns the inverse of the Weibull cumulative distribution function

$$x = \mu + \sigma\left[-\ln\left(1 - cdf\right)\right]^{1/\gamma}.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWeibullCDF** and **StatsWeibullPDF** functions.

StatsJBTest

StatsJBTest [*flags*] *srcWave*

The StatsJBTest operation performs the Jarque-Bera test on *srcWave*. Output is to the W_JBResults wave in the current data folder.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed.
	k=0: Normal with dialog (default).
	k=1: Kills with no dialog.
	k=2: Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsJBTest computes the Jarque-Bera statistic

$$JB = \frac{n}{6}\left(S^2 + \frac{K^2}{4}\right),$$

where S is the skewness, K is the kurtosis, and n is the number of points in the input wave. We can express S and K terms of the jth moment of the distribution for n samples X_i

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^j$$

as

$$S = \frac{\mu_3}{\left(\mu_2\right)^{3/2}},$$

and

$$K = \frac{\mu_4}{\left(\mu_2\right)^{2}} - 3.$$

The Jarque-Bera statistic is asymptotically distributed as a Chi-squared with two degrees of freedom. For values of *n* in the range [7,2000] the operation provides critical values obtained from Monte-Carlo simulations. For further details or if you would like to run your own simulation to obtain critical values for other values of *n*, use the JarqueBeraSimulation example experiment.

StatsJBTest reports the number of finite data points, skewness, kurtosis, Jarque-Bera statistic, asymptotic critical value, and the critical value obtained from Monte-Carlo calculations as appropriate; it ignores NaNs and INFs.

References

Jarque, C., and A. Bera, A test of normality of observations and regression residuals, *International Statistical Review*, *55*, 163-172, 1987.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsKSTest**, **WaveStats**, and **StatsCircularMoments**.

StatsKendallTauTest

StatsKendallTauTest [***flags***] ***wave1*** [**,** ***wave2***]

The StatsKendallTauTest operation performs the nonparametric Mann-Kendall test, which computes a correlation coefficient τ (similar to Spearman's correlation) from the relative order of the ranks of the data. Output is to the W_StatsKendallTauTest wave in the current data folder.

Flags

/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Inputs may be a pair of XY (1D) waves of any real numeric type or a single 1D wave, which is equivalent to using a pair of XY waves where the X wave is monotonically increasing function of the point number. StatsKendallTauTest ignores wave scaling.

Kendall's τ is 1 for a monotonically increasing input and -1 for monotonically decreasing input. The significance of the test is computed from the normal approximation

$$Var(\tau) = \frac{4n+10}{9n(n-1)},$$

where *n* is the number of data points in each wave. The significance is expressed as a P-value for the null hypothesis of no correlation.

References

Kendall, M.G., *Rank Correlation Methods*, 3rd ed., Griffin, London, 1962.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsRankCorrelationTest**.

For small values of *n* you can compute the exact probability using the procedure WM_KendallSProbability().

StatsKSTest

StatsKSTest [***flags***] ***srcWave*** [**, *distWave***]

The StatsKSTest operation performs the Kolmogorov-Smirnov (KS) goodness-of-fit test for two continuous distributions. The first distribution is *srcWave* and the second distribution can be expressed either as the optional wave *distWave* or as a user function with /CDFF. Output is to the W_KSResults wave in the current data folder.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/CDFF=*func*	Specifies a user function expressing the cumulative distribution function. See Details.
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed.
	k=0: Normal with dialog (default).
	k=1: Kills with no dialog.
	k=2: Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The Kolmogorov-Smirnov (KS) goodness-of-fit test applies only to continuous distributions and cases where the compared distribution (expressed as a user function) is completely specified without estimating parameters from the data. It compares the cumulative distribution function (CDF) of two distributions and sets the test statistic D to the largest difference between the CDFs. Because CDFs are in the range [0,1], D is also bound by this range.

When specifying the distributions with two waves, StatsKSTest first sorts the data in the waves and then computes the CDFs and D. You can also specify one of the distributions with a user function. For example, the following function tests if the data in *srcWave* is normally distributed with zero mean and stdv=5:

```
Function getUserCDF(inX)
   Variable inX
   return statsNormalCDF(inX,0,5)
End
```

Outputs are the number of elements, the KS statistic D, and the critical value. When both distributions are specified by waves, the number of elements is the weighted value (n1*n2)/(n1+n2).

References

Critical values are based on:

Birnbaum, Z. W., and Fred H. Tingey, One-sided confidence contours for probability distribution functions, *The Annals of Mathematical Statistics*, *22*, 592–596, 1951.

A statistically more powerful modification of the classic KS test can be found in:

Khamis, H.J., The two-stage delta-corrected Kolmogorov-Smirnov test, *Journal of Applied Statistics*, *27*, 439-450, 2000.

StatsKSTest implements the original KS test. The difficulty in implementing the modified tests for all the cases defined by Stephens is in obtaining the critical values which have to be derived by time consuming Monte-Carlo simulations.

Critiques can be found in:

D'Agostino, R.B., and M. Stephens, eds., *Goodness-Of-Fit Techniques*, Marcel Dekker, New York, 1986.

NIST/SEMATECH, Kolmogorov-Smirnov Goodness-of-Fit Test, in *NIST/SEMATECH e-Handbook of Statistical Methods*, <http://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm>, 2005.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsJBTest**, **WaveStats**, and **StatsCircularMoments**.

StatsKuiperCDF

StatsKuiperCDF(*V*)

The StatsKuiperCDF function returns the Kuiper cumulative distribution function

$$F(V) = 1 - 2\sum_{j=1}^{\infty}\left(4j^2V^2 - 1\right)\exp\left(-2j^2V^2\right).$$

Accuracy is on the order of 1e-15. It returns 0 for values of V<0.4 or 1 for V>3.1.

References

See in particular Section 14.3 of

Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvKuiperCDF**.

StatsKWTest

StatsKWTest [***flags***] [***wave1, wave2,… wave100***]

The StatsKWTest operation performs the nonparametric Kruskal-Wallis test which tests variances using the ranks of the data. Output is to the W_KWTestResults wave in the current data folder.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/E Computes the exact P-value using the Klotz and Teng algorithm, which may require long computation times for large data sets. You can stop the calculation by pressing Command-period (*Macintosh*) or Ctrl+Break (*Windows*) after which all remaining results remain valid and the exact P-value is set to NaN.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

- *k*=0: Normal with dialog (default).
- *k*=1: Kills with no dialog.
- *k*=2: Disables killing.

/WSTR=*waveListString*
Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Inputs are two or more 1D numerical waves (one for each group of samples). Use NaNs for missing data or use waves with different number of points.

StatsKWTest always computes the critical values using both the Chi-squared and Wallace approximations. If appropriate (small enough data set) you can also use /E to obtain the exact P value. When the calculation involves many waves or many data points the calculation of the exact critical value can be very lengthy. All the results are saved in the wave W_KWTestResults in the current data folder and are optionally displayed in a table (/T). The wave contains the following information:

Row	Data
0	Number of groups
1	Number of valid data points (excludes NaNs)
2	Alpha

Row	Data
3	Kruskal-Wallis Statistic H
4	Chi-squared approximation for the critical value Hc
5	Chi-squared approximation for the P value
6	Wallace approximation for the critical value Hc
7	Wallace approximation for the P value
8	Exact P value (requires /E)

H_0 for the Kruskal-Wallis test is that all input waves are the same. If the test fails and the input consisted of more than two waves, there is no indication for possible agreement between some of the waves. See **StatsNPMCTest** for further analysis.

V_flag will be set to -1 for any error and to zero otherwise.

References

Klotz, J.H., *Computational Approach to Statistics*, <http://www.stat.wisc.edu/~klotz/Book.pdf>.

Klotz, J., and Teng, J., One-way layout for counts and the exact enumeration of the Kruskal-Wallis H distribution with ties, *J. Am. Stat. Assoc*, *72*, 165-169, 1977.

Wallace, D.L., Simplified Beta-Approximation to the Kruskal-Wallis H Test, *J. Am. Stat. Assoc*., *54*, 225-230, 1959.

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWilcoxonRankTest**, **StatsNPMCTest**, and **StatsAngularDistanceTest**.

StatsLinearCorrelationTest

StatsLinearCorrelationTest [***flags***] ***waveA, waveB***

The StatsLinearCorrelationTest operation performs correlation tests on *waveA* and *waveB*, which must be real valued numeric waves and must have the same number of points. Output is to the W_StatsLinearCorrelationTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/CI	computes confidence intervals for the correlation coefficient.
/Q	No results printed in the history area.
/RHO=*rhoValue*	Tests hypothesis that the correlation has a nonzero value $\|r\| \le 1$.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors.

Details

The linear correlation tests start by computing the linear correlation coefficient for the *n* elements of both waves:

$$r = \frac{\sum_{i=1}^{n} X_i Y_i - \frac{1}{n}\sum_{i=1}^{n} X_i \sum_{i=1}^{n} Y_i}{\sqrt{\left(\sum_{i=1}^{n} X_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} X_i\right)^2\right)\left(\sum_{i=1}^{n} Y_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} Y_i\right)^2\right)}}$$

Next it computes the standard error of the correlation coefficient

$$sr = \sqrt{\frac{1-r^2}{n-2}}$$

The basic test is for hypothesis H_0: the correlation coefficient is zero, in which case *t* and *F* statistics are applicable. It computes the statistics:

$$t = r / sr$$

and

$$F = \frac{1+|r|}{1-|r|},$$

and then the critical values for one and two tailed hypotheses (designated by t_{c1}, t_{c2}, F_{c1}, and F_{c2} respectively). Critical value for *r* are computed using

$$\mathrm{rc}_i = \sqrt{\frac{t_c^2}{t_c^2+n}}$$

where *i* takes the values 1 or 2 for one and two tailed hypotheses. Finally, it computes the power of the test at the alpha significance level for both one and two tails (Power1 and Power2).

If you use /RHO it uses the Fisher transformation to compute

$$\mathrm{FisherZ}=\frac{1}{2}\ln\left(\frac{1+r}{1-r}\right)$$

$$\mathrm{zeta}=\frac{1}{2}\ln\left(\frac{1+\rho}{1-\rho}\right)$$

the standard error approximation

$$\mathrm{sigmaZ}=\sqrt{\frac{1}{n-3}},$$

$$\mathrm{Zstatistic}=\frac{FisherZ-zeta}{sigmaZ},$$

and the critical values from the normal distribution Z_{ci}.

The confidence intervals are calculated differently depending on the hypothesis for the value of the correlation coefficient. If /RHO is not used the confidence intervals are computed using the critical value F_{c2}, otherwise they are computed using the critical Z_{c2} and sigmaZ.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCircularCorrelationTest**, **StatsMultiCorrelationTest**, and **StatsRankCorrelationTest**.

StatsLinearRegression

StatsLinearRegression [***flags***] [***wave0, wave1,…***]

The StatsLinearRegression operation performs regression analysis on the input wave(s). Output is to the W_StatsLinearRegression wave in the current data folder or optionally to a table. Additionally, the M_DunnettMCElevations, M_TukeyMCSlopes, and M_TukeyMCElevations waves may be created as specified.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/B=*beta0* Tests the hypothesis that the slope *b*= *beta0* (default is 0). The results are expressed by the t-statistic, which can be compared with the tc value for the two-tailed test. Get the critical value for a one-tailed test using `StatsStudentCDF(1-alpha,N-2)`. It does not work with /MYVW.

/BCIW Computes two confidence interval waves for the high side and the low side of the confidence interval. The new waves are named with _CH and _CL suffixes respectively appended to the Y wave name and are created in the current data folder. For multiple runs a numeric suffix will also be appended to the names.

/BPIW[=*mAdditional*]
Computes prediction interval waves for the high side and the low side of the confidence interval on a single additional measurement (default). Use *mAdditional* to specify additional measurements. The new waves are named with _PH and _PL suffixes respectively appended to the Y wave name and are created in the current data folder. For multiple runs a numeric suffix will also be appended to the names.

/DET=*controlIndex* Performs Dunnett's multicomparison test for the elevations. The test requires more than two Y waves for regression, the test for the slopes should not reject the equal slope hypothesis, and the test for the elevations should reject the equal elevation hypothesis. *controlIndex* is the zero-based index of the Y wave representing the control (X waves do not count in the index specification). The test compares the elevation of every Y wave with the specified control.

Output is to the M_DunnettMCElevations wave in the current data folder or optionally to a table. For every Y wave and control Y wave combination, the results include SE, q, q' (shown as qp), and the conclusion with 1 to accept the hypothesis of equal elevations or 0 to reject it. Use /TAIL to determine the critical value and the sense of the test. If you use /TUK you will also get the Tukey test for the set of elevations.

/MYVW={*xWave*, *yWave*}
Specifies that the input consists of multiple Y values for each X value. It ignores all other inputs and the results are appropriate only for multiple Y values at each X point.

yWave is a 2D wave of values arranged in columns. Use NaNs for padding where rows do not have the same number of entries as others. It will use the X scaling of *yWave* when *xWave* is null, `/MYVW={*,yWave}`.

It first tests the hypothesis (H_0) that the population regression is linear in an analysis of variance calculation. It generates results 1-7 (see Details) as well as: Among Groups SS, Among Groups DF, Within Groups SS, Within Groups DF, Deviations from Linearity SS, Deviations from Linearity DF, F statistic defined by the ratio of Deviation from Linearity MS to Within Groups MS, and the critical value Fc.

Next, it tests the hypothesis that the slope beta=0. If the original H_0 was accepted, the new F statistic=regressionMS/residualMS. Otherwise the with the critical F=regressionMS/WithinGroupsMS with a corresponding critical value. Finally, it reports the values of the coefficient of determination r2 and the standard error of the estimate S_{YX}.

/PAIR Specifies that the input waves are XY pairs, where each pair must be an X wave followed by a Y wave.

/Q No results printed in the history area.

/RTO Reflects the regression through the origin.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/TAIL=*tCode* Sets the sense of the test when applying Dunnett's test (see /DET). *tCode* is 1 or 2 for a one-tail critical value and 4 for a two-tail critical value.

/TUK Performs a Tukey-type test on multiple regressions on two or more Y waves. There are two possible Tukey-type tests: The first is performed if the hypothesis of equal slopes is rejected. It compares all combinations of two Y waves to identify if some of the waves have equal slopes. Output is to the M_TukeyMCSlopes wave in the current data folder or optionally to a table. For every Y wave pair, the results include the difference between slopes (absolute value), q, the critical value qc, and the conclusion set to 1 for accepting the equality of the pair of slopes or 0 for rejecting the hypothesis.

The second Tukey-type test is performed if all the slopes are the same but the elevations are not. The test (see /DET) compares all possible pairs of elevations to determine which satisfy the hypothesis of equality. Output is to the M_TukeyMCElevations wave in the current data folder.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors.

Details

Inputs may consist of Y waves or XY wave pairs. If X data are not used, the X values are inferred from the Y wave scaling. For multiple waves where only some have pairs, use the /PAIR flag and enter * in each place where the X values should be computed.

For each input StatsLinearRegression calculates:

1. Least squares regression line y=a+b*x.
2. Mean value of X: xBar.
3. Mean value of Y: yBar.
4. Sum of the squares $(x_i\text{-xBar})^2$.
5. Sum of the squares $(y_i\text{-yBar})^2$.
6. Sum of the product $(x_iy_i\text{-xyBar})$.
7. Standard error of the estimate S_{YX}=.
8. F statistic for the hypothesis beta=0.
9. Critical F value Fc.
10. Coefficient of determination r2.
11. Standard error of the regression coefficient S_b.
12. t-statistic for the hypothesis beta=*beta0*, NaN if /B is not specified.
13. Critical value tc for the t-statistic above (used to calculate L1 and L2).
14. Lower confidence interval boundary (L1) for the regression coefficient.
15. Upper confidence interval boundary (L2) for the regression coefficient.

For two Y waves with the same slope, it computes a common slope (bc) and then tests the equality of the elevations (a). In both cases it computes a t-statistic and compares it with a critical value. If the elevations are also the same then it computes the common elevation (ac) and the pooled means of X and Y in (xp) and (yp).

For more than two Y waves it computes:

$$A_c = \sum_{j=1}^{W} A_j; \qquad A_j \equiv \sum x_i^2 = \sum_{i=0}^{n_j-1} X_i^2 - \frac{1}{n_j}\left(\sum_{i=0}^{n_j-1} X_i\right)^2$$

$$B_c = \sum_{j=1}^{W} B_j; \qquad B_j \equiv \sum xy = \sum_{i=0}^{n_j-1} XY - \frac{1}{n_j}\left(\sum_{i=0}^{n_j-1} X_i\right)\left(\sum_{i=0}^{n_j-1} Y_i\right)$$

$$C_c = \sum_{j=1}^{W} C_j; \qquad C_j \equiv \sum y^2 = \sum_{i=0}^{n_j-1} Y_i^2 - \frac{1}{n_j}\left(\sum_{i=0}^{n_j-1} Y_i\right)^2$$

$$SSp = \sum_{j=1}^{W} C_j - \frac{B_j^2}{A_j}$$

$$SSc = Cc - \frac{B_c^2}{A_c^2}$$

$$SSt = \sum_{j=1}^{W}\sum_{i=0}^{n_j} Y_{ji}^2 - \frac{1}{N}\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} Y_{ji}\right)^2 - \frac{\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} X_{ji}Y_{ji} - \frac{1}{N}\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} X_{ji}\right)\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} Y_{ji}\right)\right)^2}{\sum_{j=1}^{W}\sum_{i=0}^{n_j} X_{ji}^2 - \frac{1}{N}\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} X_{ji}\right)^2}$$

$$DFp = \sum_{j=1}^{W}\left(n_i - 2\right)$$

$$DFt = \sum_{j=1}^{W} n_i - 2$$

Here *W* is the number of Y-waves and $N = \sum_{j=1}^{W} n_j$ is the total number of data points in all Y-waves.

The test statistic F for equality of slopes is given by:

$$F = \left(\frac{SSc - SSp}{numWaves - 1}\right) \Big/ \frac{SSp}{DFp}.$$

Fc is the corresponding critical value.

Output is to the W_LinearRegressionMC wave in the current data folder.

V_flag will be set to -1 for any error and to zero otherwise.

References

See, in particular, Chapter 18 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; curvefit.

StatsLogisticCDF

`StatsLogisticCDF(x, a, b)`

The StatsLogisticCDF function returns the logistic cumulative distribution function

$$F(x;a,b)=\frac{1}{1+\exp\left(-\frac{x-a}{b}\right)}$$

where the scale parameter $b>0$ and the shape parameter is a.

See Also
Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogisticPDF** and **StatsInvLogisticCDF** functions.

StatsLogisticPDF

StatsLogisticPDF(*x*, *a*, *b*)

The StatsLogisticPDF function returns the logistic probability distribution function

$$f(x;a,b)=\frac{\exp\left(-\frac{x-a}{b}\right)}{b\left[1+\exp\left(-\frac{x-a}{b}\right)\right]^2},$$

where the scale parameter $b>0$ and the shape parameter is a.

See Also
Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogisticCDF** and **StatsInvLogisticCDF** functions.

StatsLogNormalCDF

StatsLogNormalCDF(*x*, σ [, θ, μ])

The StatsLogNormalCDF function returns the lognormal cumulative distribution function

$$F(x;\sigma,\theta,\mu)=\frac{1}{\sigma\sqrt{2\pi}}\int_0^x\frac{1}{t-\theta}\exp\left\{-\left[\ln\left(\frac{t-\theta}{\mu}\right)\right]^2\Big/2\sigma^2\right\}dt,$$

for $x \ge \theta$ and $\sigma, \mu>0$. The standard lognormal distribution is for $\theta=0$ and $\mu=1$, which are the optional parameter defaults.

See Also
Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogNormalPDF** and **StatsInvLogNormalCDF** functions.

StatsLogNormalPDF

StatsLogNormalPDF(*x*, σ [, θ, μ])

The StatsLogNormalPDF function returns the lognormal probability distribution function

$$f(x;\sigma,\theta,\mu)=\frac{1}{\sigma\sqrt{2\pi}}\frac{1}{x-\theta}\exp\left\{-\left[\ln\left(\frac{x-\theta}{\mu}\right)\right]^2\Big/2\sigma^2\right\},$$

for $x \ge \theta$ and $\sigma, \mu > 0$, where θ is the location parameter, μ is the scale parameter and, σ is the shape parameter. The standard lognormal distribution is for $\theta=0$ and $\mu=1$, which are the optional parameter defaults.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogNormalCDF** and **StatsInvLogNormalCDF** functions.

StatsMaxwellCDF

StatsMaxwellCDF(*x*, *k*)

The StatsMaxwellCDF function returns the Maxwell cumulative distribution function

$$F(x;k) = gammp\left(\frac{3}{2}, \frac{kx^2}{2}\right), \qquad x > 0.$$

where **gammp** is the regularized incomplete gamma function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsMaxwellPDF** and **StatsInvMaxwellCDF** functions.

StatsMaxwellPDF

StatsMaxwellPDF(*x*, *k*)

The StatsMaxwellPDF function returns Maxwell's probability distribution function

$$f(x;k) = \sqrt{\frac{2}{\pi}} k^{3/2} x^2 \exp\left(-\frac{kx^2}{2}\right), \qquad x > 0.$$

The Maxwell distribution describes, for example, the speed distribution of molecules in an ideal gas.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsMaxwellCDF** and **StatsInvMaxwellCDF** functions.

StatsMedian

StatsMedian(*waveName*)

The StatsMedian function returns the median value of a numeric wave *waveName*, which must not contain NaNs.

Example

```
Make/N=5 sample1={1,2,3,4,5}
Print StatsMedian(sample1)
3
Make/N=6 sample2={1,2,3,4,5,6}
Print StatsMedian(sample2)
3.5
```

See Also

Chapter III-12, **Statistics** for a function and operation overview; **WaveStats** and **StatsQuantiles**.

StatsMooreCDF

StatsMooreCDF(*x*, *N*)

The StatsMooreCDF function returns the cumulative distribution function for Moore's R*, which is used in a nonparametric version of the Rayleigh test for uniform distribution around the circle. It supports the range $3 \le N \le 120$ and does not change appreciably for $N>120$.

The distribution is computed from polynomial approximations derived from simulations and should be accurate to approximately three significant digits.

References

Moore, B.R., A modification of the Rayleigh test for vector data, *Biometrica*, *67*, 175-180, 1980.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsCircularMeans** function.

StatsMultiCorrelationTest

StatsMultiCorrelationTest [*flags*] *corrWave, sizeWave*

The StatsMultiCorrelationTest operation performs various tests on multiple correlation coefficients. Inputs are two 1D waves: *corrWave*, containing correlation coefficients, and *sizeWave*, containing the size (number of elements) of the corresponding samples. Although you can do all the tests at the same time, it rarely makes sense to do so.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/CON={*controlRow*,*tails*}

Performs a multiple comparison test using the *controlRow* element of *corrWave* as a control. It is one- or two-tailed test according to the tails parameter. Output is to the M_ControlCorrTestResults wave in the current data folder.

/CONT=*cWave* Performs a multiple contrasts test on the correlation coefficients. The contrasts wave, *cWave*, contains the contrast factor, c_i, entry for each of the n correlation coefficients r_i in *corrWave*, and satisfying the condition that the sum of the entries in *cWave* is zero. H_0 corresponds to

$$\sum_{i=0}^{n-1} c_i r_i = 0.$$

The test statistic S is

$$S = \frac{1}{\sqrt{\frac{c_i^2}{n_i - 3}}} \left| \sum_{i=0}^{n-1} c_i z_i \right|,$$

where z_i is the Fisher z transform of the correlation coefficient r_i:

$$z_i = \frac{1}{2} \ln\left(\frac{1 + r_i}{1 - r_i} \right).$$

It produces the SE value, the contrast statistic S, and the critical value, which are labeled ContrastSE, ContrastS, and Contrast_Critical, respectively, in the W_StatsMultiCorrelationTest wave.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/TUK Performs a Tukey-type multi comparison testing between the correlation coefficients by comparing every possible combination of pairs of correlation coefficients, computing the difference in their z-transforms, the SE, and the q statistic:

$$q = \frac{\left| z_j - z_i \right|}{\sqrt{\frac{1}{2}\left(\frac{1}{n_i - 3} + \frac{1}{n_j - 3} \right)}}.$$

The critical value is computed from the *q* CDF (**StatsInvQCDF**) with degrees of freedom *numWaves* and infinity. Output is to the M_TukeyCorrTestResults wave in the current data folder or optionally to a table.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Without any flags, StatsMultiCorrelationTest computes χ^2 for the correlation coefficients and compares it with the critical value.

$$\chi^2 = \sum_{i=0}^{n-1} z_i^2 \left(n_i - 3\right) - \frac{\left(\sum_{i=0}^{n-1} z_i \left(n_i - 3\right)\right)^2}{\sum_{i=0}^{n-1} \left(n_i - 3\right)},$$

where z_i is the Fisher's z transform of the correlation coefficients and n_i is the corresponding sample size. It computes the common correlation coefficient rw and its transform zw.

$$z_w = \frac{\sum_{i=0}^{n-1} z_i \left(n_i - 3\right)}{\sum_{i=0}^{n-1} \left(n_i - 3\right)}$$

These values are calculated even when not appropriate, such as when χ^2 exceeds the critical value and H_0 (all samples came from populations of identical correlation coefficients) is rejected.

The operation also computes ChiSquaredP (due to S.R. Paul), a different variant of χ^2 that is corrected for bias and should be compared with the same critical value. Output is to the W_StatsMultiCorrelationTest wave in the current data folder or optionally to a table.

References

See, in particular, Chapters 19 and 11 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsLinearCorrelationTest, **StatsCircularCorrelationTest**, **StatsDunnettTest**, **StatsTukeyTest**, **StatsInvQCDF**, and **StatsScheffeTest**.

StatsNBinomialCDF

StatsNBinomialCDF(*x*, *k*, *p*)

The StatsNBinomialCDF function returns the negative binomial cumulative distribution function

$$F(x;k,p) = Betai(k, x+1; p),$$

where **betai** is the regularized incomplete beta function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNBinomialPDF** and **StatsInvNBinomialCDF** functions.

StatsNBinomialPDF

StatsNBinomialPDF(*x*, *k*, *p*)

The StatsNBinomialPDF function returns the negative binomial probability distribution function

$$f(x;k,p) = \begin{pmatrix} x+k-1 \\ k-1 \end{pmatrix} p^k (1-p)^x, \qquad x = 0,1,2...$$

where $\binom{a}{b}$ is the **binomial** function.

The binomial distribution expresses the probability of the *k*th success in the *x*+*k* trial for two mutually exclusive results (success and failure) and *p* the probability of success in a single trial.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNBinomialCDF** and **StatsInvNBinomialCDF** functions.

StatsNCChiCDF

StatsNCChiCDF(*x*, *n*, *d*)

The StatsNCChiCDF function returns the noncentral chi-squared cumulative distribution function

$$F(x;n,d) = \sum_{i=1}^{\infty} \exp(d/2) \frac{(d/2)^i}{i!} F_c(x;n+2i),$$

where $n>0$ corresponds to degrees of freedom, $d \geq 0$ is the noncentrality parameter, and F_c is the central chi-squared distribution.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsChiCDF**, **StatsNCChiPDF**, and **StatsChiPDF** functions.

StatsNCChiPDF

StatsNCChiPDF(*x*, *n*, *d*)

The StatsNCChiPDF function returns the noncentral chi-squared probability distribution function

$$f(x;n,d) = \frac{\sqrt{d}\exp\left(-\frac{x+d}{2}\right)x^{(n-1)/2}}{2(dx)^{n/4}} I_{n/2-1}\left(\sqrt{dx}\right).$$

where $n>0$ is the degrees of freedom, $d \geq 0$ is the noncentrality parameter, and $I_k(x)$ is the modified Bessel function of the first kind, **bessI**.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCChiCDF**, **StatsInvNCChiCDF**, **StatsChiCDF**, and **StatsChiPDF** functions.

StatsNCFCDF

StatsNCFCDF(*x*, *n1*, *n2*, *d*)

The StatsNCFCDF function returns the cumulative distribution function of the noncentral F distribution. *n1* and *n2* are the shape parameters and *d* is the noncentrality measure. There is no closed form expression for the distribution.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCFPDF** and **StatsInvNCFCDF** functions.

StatsNCFPDF

StatsNCFPDF(*x, n1, n2, d*)

The StatsNCFPDF function returns the probability distribution function of the noncentral F distribution

$$f(x;n_1,n_2,d)=\frac{\exp(-d/2)}{B\left(\frac{n_1}{2},\frac{n_2}{2}\right)}x^{n_1/2-1}(xn_1+n_2)^{-(n_1+n_2)/2}n_1^{n_1/2}n_2^{n_2/2}\,{}_1F_1\left(\frac{n_1+n_2}{2},\frac{n_1}{2},\frac{xdn_1}{2(xn_1+n_2)}\right),$$

where $B()$ is the **beta** function and ${}_1F_1()$ is the hypergeometric function **hyperG1F1**.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCFCDF** and **StatsInvNCFCDF** functions.

StatsNCTCDF

StatsNCTCDF(*x, df, d*)

The StatsNCTCDF function returns the cumulative distribution function of the noncentral Student-T distribution. *df* is the degrees of freedom (positive integer) and *d* is the noncentrality measure. There is no closed form expression for the distribution.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentCDF**, **StatsStudentPDF**, and **StatsNCTPDF** functions.

StatsNCTPDF

StatsNCTPDF(*x, df, d*)

The StatsNCTPDF function returns the probability distribution function of the noncentral Student-T distribution. *df* is the degrees of freedom (positive integer) and *d* is the noncentrality measure.

$$f(x;n,\delta)=\frac{n^{n/2}n!}{2^n e^{\delta^2/2}(n+x^2)^{n/2}\Gamma\left(\frac{n}{2}\right)}\left\{\frac{\sqrt{2}\delta x\,{}_1F_1\left(\frac{n}{2}+1;\frac{3}{2};\frac{\delta^2x^2}{2(n+x^2)}\right)}{(n+x^2)\Gamma\left(\frac{n+1}{2}\right)}+\frac{{}_1F_1\left(\frac{n+1}{2};\frac{1}{2};\frac{\delta^2x^2}{2(n+x^2)}\right)}{\sqrt{(n+x^2)}\Gamma\left(\frac{n}{2}+1\right)}\right\}$$

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentPDF**, **StatsStudentCDF**, and **StatsNCTCDF** functions.

StatsNormalCDF

StatsNormalCDF(*x, m, s*)

The StatsNormalCDF function returns the normal cumulative distribution function

$$F(x,\mu,\sigma) = \frac{1}{2} + \frac{1}{2} erf\left(\frac{x-\mu}{\sigma\sqrt{2}}\right),$$

where *erf* is the error function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **erf**, **StatsNormalPDF** and **StatsInvNormalCDF** functions.

StatsNormalPDF

StatsNormalPDF(*x, m, s*)

The StatsNormalPDF function returns the normal probability distribution function

$$f(x,\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNormalCDF** and **StatsInvNormalCDF** functions.

StatsNPMCTest

StatsNPMCTest [***flags***] [***wave1, wave2,… wave100***]

The StatsNPMCTest operation performs a number of nonparametric multiple comparison tests. Output waves are saved in the current data folder according to the test(s) performed. Some tests are only appropriate when you have the same number of samples in all groups. StatsNPMCTest usually follows **StatsANOVA1Test** or **StatsKWTest**.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/CIDX=*controlIndex* Performs nonparametric multiple comparisons on a control group specified by the zero-based *controlIndex* wave in the input list. Output is to the M_NPCCResults wave in the current data folder or optionally to a table. The output column contents are: the first contains the difference between the rank sums of the control and each of the other waves; the second contains the standard error (SE); the third contains the statistic q, defined as the ratio of the difference in rank sums to SE; the fourth contains the critical value which also depends on the tails specification (see /TAIL); and the fifth contains the conclusion with 0 to reject H_0 and 1 to accept it. One version of this test applies when all inputs contain the same number of samples. When that is not the case, it uses the Dunn-Hollander-Wolfe approach to compute an appropriate SE and to handle possible ties.

/CONW=*cWave* Performs a nonparametric multiple contrasts tests. *cWave* has one point for each input wave. The *cWave* value is 1 to include the corresponding (zero based) input wave in the first group, 2 to include the wave in the second group, or zero to exclude the wave.

The contrast is defined as the difference between the normalized sum of the ranks of the first group and that of the second group. If *cWave*={0,1,1,1,2}, then the contrast is computed as

$$\text{contrast: } \frac{1}{3}[R_{n1}+R_{n2}+R_{n3}]-R_{n4}.$$

where R_{ni} is the normalized rank sum of the samples from the corresponding input wave. Note the significance of allowing zeros in the contrast wave because the actual ranking is performed on the pool of all the samples.

Output is to the M_NPMConResults wave in the current data folder or optionally to a table. The output column contents are: the first is the contrast value; the second is the standard error (SE); the third is the statistic S, which is the ratio of the absolute value of the contrast to SE; the fourth is the critical value (from χ^2 the approximation); and the fifth is the conclusion with 0 to reject H_0 and 1 indicating acceptance.

This test supports input waves with different number of samples and can also handle tied ranks. Note that the contrast wave used here is structured differently than for **StatsMultiCorrelationTest**.

/DHW Performs the Dunn-Holland-Wolfe test, which supports unequal number of samples and accounts for ties in the rank sums. Output is to the M_NPMCDHWResults wave in the current data folder or optionally to a table. The output column contents are: the first contains the difference between the means of the rank sums (rank sums divided by the number of samples in the group), the second contains the standard error (SE), the third contains the DHW statistic Q, the fourth contains the critical value, and the fifth contains the conclusion (0 to reject H_0 and 1 to accept).

/Q No results printed in the history area.

/SWN Creates a text wave, T_NPCCResultsDescriptors, containing wave names corresponding to each row of the comparison table (Save Wave Names). Use /T to append the text wave to the last column.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results.

/TAIL=*tc* Specifies H_0 with /CIDX.

tc=1: One tailed test ($\mu_c \leq \mu_a$).

tc=2: One tailed test ($\mu_c \geq \mu_a$).

tc=4: Default; two tailed test ($\mu_c = \mu_a$).

Code combinations are not allowed.

/SNK Performs a nonparametric variation on the Student-Newman-Keuls test where the standard error SE is a function of p (the rank difference). This test requires equal numbers of samples in all groups; use /DHW for unequal sizes.

Output is to the M_NPMCSNKResults wave in the current data folder. The output column contents are: the first contains the difference between rank sums, the second contains the standard error (SE), the third contains the p value (rank difference), the fourth the statistic, the fifth contains the critical value, and the sixth contains the conclusion (0 to reject H_0 and 1 to accept). This test is more sensitive to differences than the Tukey test (/TUK).

/TUK Perform a Tukey-type (Nemenyi) multiple comparison test using the difference between the rank sums. This is the default that is performed if you do not specify any of the test flags. This test requires equal numbers of points in all waves; use /DHW for unequal sizes.

Output is to the M_NPMCTukeyResults wave in the current data folder. The output column contents are: the first contains the difference between the rank sums, the second contains the SE values, the third contains the statistic q, the fourth contains the critical value for this specific alpha and the number of groups; and the last contains a conclusion flag with 0 indicating a rejection of H_0 and 1 indicating acceptance. H_0 postulates that the paired means are the same.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors.

Details

Inputs to StatsNPMCTest are two or more 1D numerical waves (one wave for each group of samples) containing two or more valid entries. The waves must have the same number of points for the use /SNK and /TUK tests, otherwise, for waves of differing lengths you must use the Dunn-Hollander-Wolfe test (/DHW).

V_flag will be set to zero for no execution errors. Individual tests may fail if, for example, there are different number of samples in the input waves for a test that requires an equal number of points. StatsNPMCTest skips failed tests and V_flag will be a binary combination identifying the failed test(s):

V_flag & 1 Tukey method failed (/TUK).

V_flag & 2 Student-Newman-Keuls failed (/SNK).

V_flag will be set to -1 for any other errors.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test** and **StatsKWTest**.

For multiple comparisons in parametric tests see: **StatsDunnettTest** and **StatsScheffeTest**.

StatsNPNominalSRTest

StatsNPNominalSRTest [***flags***] [***srcWave***]

The StatsNPNominalSRTest operation performs a nonparametric serial randomness test for nominal data consisting of two types. The null hypothesis is that the data are randomly distributed. Output is to the W_StatsNPSRTest wave in the current data folder.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/Q No results printed in the history area.

/P={*m*,*n*,*u*} Provides a summary of the data instead of providing the nominal series. *m* is the number of elements of the first type, *n* is the number of elements of the second type, and *u* is the number of runs or contiguous sequences of each type. Do not use *srcWave* with /P.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/Z Ignores errors.

Details

The input wave to StatsNPNominalSRTest is specified with *srcWave* or /P. The wave must contain exactly two values. If *srcWave* is a text wave, then each type can be designated by a letter or by a short string (less than 200 characters). If *srcWave* is numeric, you should avoid the usual floating point waves, which can give rise to internal representations of more than two distinct values. Output to W_StatsNPSRTest includes the total number of points (*N*), the number of occurrences (*m*) of the first variable, the number of occurrences (*n*) of the second variable, and the number of runs (*u*). When both *m* and *n* are less than 300, it computes the P value (probability P(*u'*<*u*)) and the critical values using the Swed and Eisenhart algorithm. When *m* or *n* are larger than 300, it computes the mean and standard deviation of an equivalent normal distribution with the corresponding critical value.

References

Swed, F.S., and C. Eisenhart, Tables for testing randomness of grouping in a sequence of alternatives, *Ann. Math. Statist.*, *14*, 66-87, 1943.

See, in particular, Chapter 25 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsSRTest**.

StatsParetoCDF

StatsParetoCDF(*x*, *a*, *c*)

The StatsParetoCDF function returns the Pareto cumulative distribution function

$$F(x;a,c) = 1 - \left(\frac{a}{x}\right)^c .$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsParetoPDF** and **StatsInvParetoCDF** functions.

StatsParetoPDF

StatsParetoPDF(*x*, *a*, *c*)

The StatsParetoPDF function returns the Pareto probability distribution function

$$f(x;a,c)=\frac{c}{x}\left(\frac{a}{x}\right)^{c}, \qquad \begin{array}{l} a,c>0 \\ x \geq a. \end{array}$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsParetoCDF** and **StatsInvParetoCDF** functions.

StatsPermute

StatsPermute(*waveA*, *waveB*, *dir*)

The StatsPermute function permutes elements in *waveA* based on the lexicographic order of *waveB* and the direction *dir*. It returns 1 if a permutation is possible and returns 0 otherwise. Use *dir*=1 for the next permutation and *dir*=-1 for a previous permutation.

Details

Both *waveA* and *waveB* must be numeric. The lexicographic order of elements in the index wave is set so that permutations start with the index wave *waveB* in ascending order and end in descending order. Elements of *waveA* are permuted in place according to the order of the indices in *waveB* which are clipped (after permutation) to the valid range of entries in *waveA*. *waveB* is also permuted in place in order to allow you to obtain sequential permutations. If *waveA* consists of real numbers you can permute them using the lexicographic value of the entries directly. To do so pass `$""` for *waveB*. Whenever it returns 0, neither *waveA* and *waveB* are changed.

Examples

```
Function allPermutations(num)
    Variable num

    Variable i,nf=factorial(num)
    Make/O/N=(num) wave0=p+1,waveA,waveB=p

    Print wave0
    for(i=0;i<nf;i+=1)
        waveA=wave0
        if(statsPermute(waveA,waveB,1)==0)
            break
        endif
        print waveA
    endfor
end

Executing allPermutations(3) prints:
  wave0[0]= {1,2,3}
  waveA[0]= {1,3,2}
  waveA[0]= {2,1,3}
  waveA[0]= {2,3,1}
  waveA[0]= {3,1,2}
  waveA[0]= {3,2,1}
```

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsPoissonCDF

StatsPoissonCDF(*x*, λ)

The StatsPoissonCDF function returns the Poisson cumulative distribution function

$$F(x;\lambda) = \sum_{i=0}^{x} \frac{\exp(-\lambda)\lambda^i}{i!}, \qquad x = 0,1,2...$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPoissonPDF** and **StatsInvPoissonCDF** functions.

StatsPoissonPDF

StatsPoissonPDF(*x*, λ)

The StatsPoissonPDF function returns the Poisson probability distribution function

$$f(x;\lambda) = \frac{\exp(-\lambda)\lambda^x}{x!}, \qquad x = 0,1,2...$$

where λ is the shape parameter.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPoissonCDF** and **StatsInvPoissonCDF** functions.

StatsPowerCDF

StatsPowerCDF(*x*, *b*, *c*)

The StatsPowerCDF function returns the Power Function cumulative distribution function

$$F(x;b,c) = \left(\frac{x}{b}\right)^c$$

where the scale parameter *b* and the shape parameter *c* satisfy $b,c > 0$ and $b \geq x \geq 0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPowerPDF**, **StatsInvPowerCDF** and **StatsPowerNoise** functions.

StatsPowerNoise

StatsPowerNoise(*b*, *c*)

The StatsPowerNoise function returns a pseudorandom value from the power distribution function with probability distribution:

$$f(x;b,c) = \frac{c}{x}\left(\frac{x}{b}\right)^c.$$

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use SetRandomSeed. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

The **StatsPowerPDF StatsInvPowerCDF** and **StatsInvPowerCDF** functions.

Noise Functions on page III-332.

Chapter III-12, **Statistics** for a function and operation overview.

StatsPowerPDF

StatsPowerPDF(*x, b, c*)

The StatsPowerPDF function returns the Power Function probability distribution function

$$f(x,b,c) = \frac{|c|}{x}\left(\frac{x}{b}\right)^c ,$$

where b is a scale parameter and c is a shape parameter.

For b,c > 0, x is drawn from b >= x >= 0.

For b>0, c<0, x is drawn from x>b.

For b<0, c>0, x is drawn from -b <= x <= 0.

For b<0, c<0, x is drawn from x<-b.

Note that for -1<c<0 the average diverges and the magnitude of a mean calculated from N samples will increase indefinitely with N.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPowerCDF**, **StatsInvPowerCDF** and **StatsPowerNoise** functions.

StatsQCDF

StatsQCDF(*q, r, c, df*)

The StatsQCDF function returns the value of the Q cumulative distribution function for *r* the number of groups, *c* the number of treatments, and *df* the error degrees of freedom (*f*=*rc*(*n*-1) with sample size *n*).

Details

The Q distribution is the maximum of several Studentized range statistics. For a simple Tukey test, use *r*=1.

References

Copenhaver, M.D., and B.S. Holland, Multiple comparisons of simple effects in the two-way analysis of variance with fixed effects, *Journal of Statistical Computation and Simulation*, *30*, 1-15, 1988.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTukeyTest** function.

StatsQpCDF

StatsQpCDF(*q, nr, nt, dt, side, sSizeWave*)

The StatsQpCDF function returns the the Q' cumulative distribution function associated with Dunnett's test.

Here *nr* is the number of groups (should be set to 1), *nt* is the number of treatments, *df* is the error degrees of freedom.

Set *side*=1 for upper-tail or *side*=2 for two-tailed CDF.

sSizeWave is an integer wave of nt rows specifying the number of samples in each treatment.

Details

StatsQpCDF is a modified Q distribution typically used with Dunnett's test, which compares the various means with the mean of the control group or treatment

References

"Algorithm AS 251: Multivariate Normal Probability Integrals with Product Correlations Structure", C. W. Dunnett, *Appl. Stat.*, 38 (1989) 564-579.

A short correction for the algorithm was published in: *Appl. Stat.*, 42 (1993) 709.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsDunnettTest**, **StatsInvQpCDF**, and **StatsInvQCDF** functions.

StatsQuantiles

StatsQuantiles [***flags***] ***srcWave***

The StatsQuantiles operation computes quantiles and elementary univariate statistics for a set of data in *srcWave*.

Flags

/ALL	Invokes all flags except /Q, /QM, and /Z.
/BOX	Computes parameters necessary to construct a box plot.
/iNaN	Ignores NaNs, which are sorted to the end of the array by default.
/IW	Creates an index wave W_QuantilesIndex. W_QuantilesIndex[*i*] corresponds to the position of *srcWave*[*i*] when sorted from minimum to maximum.
/Q	No information printed in the history area.
/QM=*qMethod*	Specifies the method for computing quartiles. *qMethod* has one of these values: 0: Tukey (default). 1: Minitab. 2: Moore and McCabe. 3: Mendenhall and Sincich. See Details for more information.
/QW	Creates a single precision wave W_QuantileValues containing the quantile value corresponding to each entry in *srcWave*.
/STBL	Uses a stable sort, which may require significant computation time for multiple entries with the same value.
/TM	Computes the tri-mean: 0.25*(V_Q25+2*median+V_Q75).
/TRIM=*tVal*	Computes the trimmed mean which is the mean value of the entries between the quantiles *tVal* (in %) and 100-*tVal*. By default *tVal*=25 and the trimmed mean corresponds to the midmean.
/Z	Ignores any errors.

Details

StatsQuantiles produces quick five-number summaries or more detailed results for univariate data. Values are returned in the wave W_StatsQuantiles and in the variables:

V_min	Minimum value.
V_max	Maximum value.
V_Median	Median value.
V_Q25	Lower quartile.
V_Q75	Upper quartile.
V_IQR	Inter-quartile range V_Q75-VQ25, which is also known as the H-spread.

Entries in the wave W_StatsQuantiles depend on your choice of flags. Each row has a row label explicitly defining its value. If you use the /ALL flag, W_StatsQuantiles will contain the following row labels:

minValue	lowerInnerFence
maxValue	lowerOuterFence
Median	upperInnerFence
Q25	upperOuterFence
Q75	triMean
IQR	trimmedMean

Otherwise, W_StatsQuantiles will contain the first five entries and any additionally requested value. You should always access values using the dimension labels (see **Dimension Labels** on page II-109).

There is frequently some confusion in comparing statistical results computed by different programs because each may use a different definition of quartiles. You can specify the method of computing the quartiles as you prefer with the /QM flag. If you neglect to choose a method, StatsQuantiles uses Tukey's method, which computes quartiles (also called hinges) as the lower and upper median values between the median of the data and the edges of the array. The Moore and McCabe method is similar to Tukey's method except you do not include the median itself in computing the quartiles. Mendenhall and Sincich compute the quartiles using 1/4 and 3/4 of (numDataPoints+1) and round to the nearest integer (if the fraction part is exactly 0.5 they round up for the lower quartile and down for the upper quartile). Minitab uses the same expressions but instead of rounding it uses linear interpolation.

StatsQuantiles uses a stable index sorting routine so that

```
IndexSort W_QuantilesIndex,srcWave
```

is a monotonically increasing wave.

References

Tukey, J. W., *Exploratory Data Analysis*, 688 pp., Addison-Wesley, Reading, Massachusetts, 1977.

Mendenhall, W., and T. Sincich, *Statistics for Engineering and the Sciences*, 4th ed., 1008 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1995.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **WaveStats**, **StatsMedian**, **Sort**, and **MakeIndex**.

StatsRankCorrelationTest

StatsRankCorrelationTest [*flags*] *waveA, waveB*

The StatsRankCorrelationTest operation performs Spearman's rank correlation test on *waveA* and *waveB*, 1D waves containing the same number of points. Output is to the W_StatsRankCorrelationTest wave in the current data folder.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors.

Details

StatsRankCorrelationTest ranks *waveA* and *waveB* and then computes the sum of the squared differences of ranks for all rows. Ties are assigned an average rank and the corrected Spearman rank correlation coefficient is computed with ties. It reports the sum of the squared ranks (sumDi2), the sums of the ties coefficients (sumTx and sumTy respectively), the Spearman rank correlation coefficient (in the range [-1,1]), and the critical value. H_0 corresponds to zero correlation against the alternative of nonzero correlation. The critical value is usually lower than the one in published tables. When the first derivative of the CDF is discontinuous, tables tend to use a more conservative value by choosing the next transition of the CDF as the critical value. StatsRankCorrelationTest is not as powerful as **StatsLinearCorrelationTest**.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsLinearCorrelationTest, **StatsCircularCorrelationTest**, **StatsKendallTauTest**, **StatsSpearmanRhoCDF**, and **StatsInvSpearmanCDF**.

StatsRayleighCDF

StatsRayleighCDF(*x* [, *s* [, *m*]])

The StatsRayleighCDF function returns the Rayleigh cumulative distribution function

$$F(x;\sigma,\mu) = 1 - \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \qquad \sigma > 0, x > \mu.$$

with defaults *s*=1 and *m*=0. It returns NaN for $s \le 0$ and zero for $x \le m$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRayleighPDF** and **StatsInvRayleighCDF** functions.

StatsRayleighPDF

StatsRayleighPDF(*x* [, *s* [, *m*]])

The StatsRayleighPDF function returns the Rayleigh probability distribution function

$$f(x;\sigma,\mu) = \frac{x-\mu}{\sigma^2}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \qquad \sigma > 0, x > \mu.$$

with defaults *s*=1 and *m*=0. It returns NaN for $s \le 0$ and zero for $x \le m$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRayleighCDF** and **StatsInvRayleighCDF** functions.

StatsRectangularCDF

StatsRectangularCDF(*x*, *a*, *b*)

The StatsRectangularCDF function returns the rectangular (uniform) cumulative distribution function

$$F(x,a,b) = \begin{cases} 0 & x \le a \\ \dfrac{x-a}{b-a} & a \le x \le b \\ 1 & x \ge b \end{cases}$$

where *a*< *b*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRectangularPDF** and **StatsInvRectangularCDF** functions.

StatsRectangularPDF

StatsRectangularPDF(*x*, *a*, *b*)

The StatsRectangularPDF function returns the rectangular (uniform) probability distribution function

$$f(x;a,b) = \begin{cases} \dfrac{1}{b-a} & a \le x \le b \\ 0 & otherwise \end{cases}$$

where *a*< *b*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRectangularCDF** and **StatsInvRectangularCDF** functions.

StatsResample

StatsResample /N=*numPoints* [*flags*] *srcWave*

The StatsResample operation resamples *srcWave* by drawing (with replacement) *numPoints* values from *srcWave* and storing them in the wave W_Resampled or M_Resampled if /MC is used. You can iterate the process and compute various statistics on the data samples.

Flags

/ITER=*n* Repeats the resampling for *n* iterations, which is useful only when combined with /WS or /SQ.

/JCKN=*ufunc* Performs Jack-Knife analysis. Here ufunc is a user function of the format:

```
Function ufunc(inWave)
    wave inWave
    ... compute some statistic for inWave
    return someValue
End
```

The results are stored in the wave W_JackKnifeStats in the current data folder. Use

```
Edit W_JackKnifeStats.ld
```

to display the wave with dimension labels.

The idea behind this method is that *ufunc* returns some statistic z for inWave which is a subsample of srcWave of size (n-1). There are exactly n iterations and in each iteration the operation calls *ufunc* with one element of srcWave missing and stores the result in an internal array. At the end of iterations it uses the array to compute the various Jack-Knife estimates.

The standard estimator is defined as:

$$Z = ufunc(srcWave).$$

The Jack-Knife estimator is simply:

$$\hat{z} = \frac{1}{n}\sum_{i=1}^{n} z_i .$$

The Jack-Knife t-estimator is slightly less biased. It is given by:

$$t = nZ - (n-1)\hat{z},$$

and the estimate of the standard error is given by:

$$\hat{\sigma}_{\hat{z}} = \sqrt{\frac{n-1}{n}\sum_{i=1}^{n}(z_i - \hat{z})^2} .$$

/K Kills W_Resampled after passing it to WaveStats. When /ITER is used, W_Resampled is not saved.

/MC Use /MC when you want to sample random (complete) rows from a multi-column 2D *srcWave*. The combination of /N=n with /MC results in the wave M_Resampled in the current data folder. M_Resampled will have n rows, the same number of columns and the same data type as *srcWave*.

/N=*numPoints* Specifies the number of points sampled from *srcWave*.

/Q No information printed in the history area.

/SQ=*m* Uses StatsQuantiles to compute the data quartiles. The methods are:

m=0: Tukey (default).

m=1: Minitab.

m=2: Moore and McCabe.

m=3: Mendenhall and Sincich.

See Details for information about how the results are stored. The default trim value is 25%.

/WS=*m* Uses WaveStats operation to calculate data statistics.

m=0: Creates a new wave containing the samples (default).

m=1: Creates the new wave and passes it to `WaveStats/Q/M=1`.

m=2: Creates the new wave and passes it to `WaveStats/Q/M=2`.

See Details for information about how the results are stored.

/Z Ignores any errors.

Details

StatsResample can perform Bootstrap Analysis, permutations tests, and Monte-Carlo simulations. It draws the specified number of data points (with replacement) from *srcWave* and places them in a destination wave W_Resampled.

Specify /WS or /SQ to use the WaveStats or StatsQuantiles operations, respectively, to compute results directly from the data. StatsResample normally creates the wave W_Resampled and, optionally, the M_WaveStats and W_StatsQuantiles waves. Both options also create various V_ variables described below. If you use more than one iteration, StatsResample creates instead the waves M_WaveStatsSamples and M_StatsQuantilesSamples for the results.

M_WaveStatsSamples (with /WS) contains a column for each iteration. Each column is equivalent to the contents of M_WaveStats for that iteration. You can use the command

```
Edit M_WaveStatsSamples.ld
```

to display the results in a table using row labels, and, for example, to display a graph of the rms of the samples as a function of iteration number execute:

```
Display M_WaveStatsSamples[5][]
```

M_StatsQuantilesSamples (with /SQ) contains a column for each iteration. Each column consists of the contents of W_StatsQuantiles for the corresponding data. Here again you can execute the command

```
Edit M_StatsQuantilesSamples.ld
```

to display the wave in a table using row labels. To display a graph of the median as a function of iteration execute:

```
Display M_statsQuantilesSamples[2][]
```

Output Variables

StatsResample creates the following variables: V_Median, V_Q25, V_Q75, V_IQR, V_min, V_max, V_numNaNs, V_numINFs, V_avg, V_sdev, V_rms, V_adev, V_skew, V_kurt, and V_Sum.

These variables are valid only if you use either /SQ or /WS, but not both, and only if you do not use /ITER. Unused variables are set to NaN.

If you use /SQ the operation sets V_Median, V_Q25, V_Q75, V_IQR, V_min, and V_max.

If you use /WS the operation sets V_min, V_max, V_numNaNs, V_numINFs, V_avg, V_sdev, V_rms, V_adev, V_skew, V_kurt, and V_Sum.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsSample**, **WaveStats** and **StatsQuantiles**.

StatsSample

StatsSample /N=*numPoints* [*flags*] *srcWave*

StatsSample creates a random, non-repeating sample from *srcWave*.

It samples *srcWave* by drawing without replacement *numPoints* values from *srcWave* and storing them in the output wave W_Sampled or M_Sampled if /MC or /MR are used.

Flags

/N=*numPoints* Specifies the number of points sampled from *srcWave*. When combined with /MC, *numPoints* is the number of sampled rows and when combined with /MR, it is the number of sampled columns.

/MC Use /MC (multi-column) to randomly sample full rows from *srcWave*, i.e., the output consists of all columns of each selected row. /MC and /MR are mutually exclusive flags.

/MR — Use /MR (multi-row) to randomly sample full columns from *srcWave*, i.e., the output consists of all rows of each of the selected columns. /MC and /MR are mutually exclusive flags.

/Z — Ignores errors.

Details

If you omit /MC and /MR, the output is a 1D wave named W_Sampled where the samples are chosen from *srcWave* without regard to its dimensionality.

If you use either /MC or /MR the output is a 2D wave named M_Sampled which will have either the same number of columns (/MC) as *srcWave* or the same number of rows (/MR) as *srcWave*.

See Also

Chapter III-12, **Statistics**, **StatsResample**

StatsRunsCDF

StatsRunsCDF(*n*, *r*)

The StatsRunsCDF function returns the cumulative distribution function for the up and down runs distribution for total number of runs *r* in a random linear arrangement of *n* unequal elements. There is no closed form expression. It is computed numerically from the recursion of the probability density

$$f(r,n) = \frac{rf(r,n-1) + 2f(r-1,n-1) + (n-r)f(r-2,n-1)}{n},$$

with the initial condition

$$f(1,n) = \frac{2}{n!}.$$

References

Bradley, J.V., *Distribution-Free Statistical Tests*, Prentice Hall, Englewood Cliffs, New Jersey, 1968.

Olmstead, P.S., Distribution of sample arrangements for runs up and down, *Annals of Mathematical Statistics*, *17*, 24-33, 1946.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsSRTest** function.

StatsScheffeTest

StatsScheffeTest [***flags***] [***wave1, wave2,… wave100***]

The StatsScheffeTest operation performs Scheffe's test for the equality of the means. It supports two basic modes: the default tests all possible combinations of pairs of waves; the second tests a single combination where the precise form of H_0 is determined by the coefficients of a contrast wave (see /CONT). Output is to the M_ScheffeTestResults wave in the current data folder.

Flags

/ALPH=*val* — Sets the significance level (default 0.05).

/CONW=*cWave* — Performs a multiple contrasts test. *cWave* has one point for each input wave. The *cWave* value is 1 to include the corresponding (zero based) input wave in the first group, 2 to include the wave in the second group, or zero to exclude the wave.

The contrast is defined as the difference between the normalized sum of the ranks of the first group and that of the second group. If *cWave*={0,1,1,1,2}, then the contrast hypothesis H_0 corresponds to:

$$\frac{\bar{X}_1 + \bar{X}_2 + \bar{X}_3}{3} - \bar{X}_4 = 0.$$

For each pair of waves (*i*, *j*) with $i \neq j$, it computes

$$SE_{ij} = \sqrt{s^2\left(\frac{1}{n_j} + \frac{1}{n_i}\right)}, \qquad s^2 = \sum_{i=1}^{W}\sum_{j=0}^{n_j-1} X_j^2 - \sum_{i=1}^{W}\frac{\left(\sum_{j=0}^{n_j-1} X_j\right)^2}{n_j},$$

the statistic

$$S = \frac{\left|\sum_{i=0}^{n-1} c_i \overline{X}_i\right|}{SE},$$

the critical value, and a result field which is set to 1 if H_0 should be accepted or 0 if it should be rejected. W is the total number of waves, n_i and $\bar{X}_i$ are respectively the number of data points and the average of wave i.

/Q No results printed in the history area.

/SWN Creates a text wave, T_ScheffeDescriptors, containing wave names corresponding to each row of the comparison table (Save Wave Names). Use /T to append the text wave to the last column.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The default of StatsScheffeTest (also known as the S test) tests the hypotheses of equality of means for each possible pair of samples. It is not as powerful as Tukey's test (**StatsTukeyTest**) and is more useful for hypotheses formulated as multiple contrasts (see /CONT).

References

See, in particular, Chapter 11 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test**, **StatsDunnettTest** and **StatsTukeyTest**.

StatsSignTest

StatsSignTest [***flags***] ***wave1, wave2***

The StatsSignTest operation performs the sign test for paired-sample data contained in *wave1* and *wave2*.

Flags

/ALPH=*val* Sets the significance level (default 0.05).

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The input waves must be the equal length, real numeric waves and must not contain any NaNs or INFs. Results are saved in the wave W_SignTest and are optionally displayed in a table. StatsSignTest computes the differences in each pair and counts the total number of entries with positive and negative differences, and tests the results using a binomial distribution. When the number of data pairs exceeds 1024 it uses a normal approximation to the binomials for calculating the probabilities and the power of the test.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsSpearmanRhoCDF

StatsSpearmanRhoCDF(*r*, *N*)

The StatsSpearmanRhoCDF function returns the cumulative distribution function for Spearman's *r*, which is used in rank correlation test. It is valid for $N>1$ and $-1 \le r \le 1$. The distribution is mostly computed using the Edgeworth series expansion.

References

Algorithm AS 89, *Appl. Statist.*, *24*, 377, 1975.

van de Wiel, M.A., and A. Di Bucchianico, Fast computation of the exact null distribution of Spearman's rho and Page's L statistic for samples with and without ties, *J. of Stat. Plan. and Inference*, *92*, 133-145, 2001.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRankCorrelationTest**, **StatsInvSpearmanCDF**, and **StatsKendallTauTest** functions.

StatsSRTest

StatsSRTest [***flags***] ***srcWave***

The StatsSRTest operation performs a parametric or nonparametric serial randomness test on *srcWave*, which must contain finite numerical data. The null hypothesis of the test is that the data are randomly distributed. Output is to the W_StatsSRTest wave in the current data folder.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/GCD Tests the output of a random number generator (RNG). *srcWave* consists of values between 0 and 2^{32} (converted to unsigned 32-bit integers). GCD computes the gcd for consecutive pairs of data in *srcWave*. The number of steps in the GCD and the distribution of the GCD's are compared with ideal distributions and corresponding P values are reported. This test is part of Marsaglia's Die-Hard battery of tests. P-values close to either 0 or 1 indicate a nonideal RNG. You should use the reported minimum and maximum values to check that the input is indeed in the proper range. Typically *srcWave* consists of at least1e6 entries.

/NAPR Use the normal approximation even when the number of points is below 150.

/NP Performs a nonparametric serial randomness test by counting the numbers of runs up and down and computing the probability that such a value is obtained by chance.

/P Performs a parametric serial randomness test.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

- *k*=0: Normal with dialog (default).
- *k*=1: Kills with no dialog.
- *k*=2: Disables killing.

/Z Ignores errors.

Details

The parametric test for serial randomness is according to Young. *C* is given by

$$C = 1 - \frac{\sum_{i=0}^{n-2}\left(X_i - X_{i+1}\right)^2}{2\sum_{i=0}^{n-1}\left(X_i - \overline{X}\right)^2},$$

where $\overline{X}$ is the mean and n is the number of points in *srcWave*. The critical value is obtained from mean square successive difference distribution **StatsInvCMSSDCDF**. For more than 150 points, StatsSRTest uses the normal approximation and provides the critical values from the normal distribution. For samples from a normal distribution, *C* is symmetrically distributed about 0 with positive values indicating positive correlation between successive entries and negative values corresponding to negative correlation.

The nonparametric test consists of counting the number of runs that are successive positive or successive negative differences between sequential data. If two sequential data are the same it computes two numbers of runs by considering the two possibilities where the equality is replaced with either a positive or a negative difference. The results of the operation include the number of runs up and down, the number of unchanged values (the number of places with no difference between consecutive entries), the size of the longest run and its associated probability, the number of converted equalities, and the probability that the number of runs is less than or equal to the reported number (**StatsRunsCDF**). When equalities are encountered the operation computes the probabilities that the computed number of runs or less can be found in an equivalent random sequence.

Converted equalities are those with the same sign on both sides so that when we replace the equality by the opposite sign we increase the number of runs. The equalities that are not converted are found between two different signs and therefore regardless of the sign that we give them they do not affect the total number of runs. We implicitly assume that the data does not contain more than one sequential equalities.

The longest run is determined without taking into account equalities or their conversions. The probability of the longest run is computed from Equation 6 of Olmstead, which is accurate within 0.001 when the number of runs is 5 or more. This probability applies to either positive or negative differences and should be divided by two if a specific sign is selected.

References

Bradley, J.V., *Distribution-Free Statistical Tests*, Prentice Hall, Englewood Cliffs, New Jersey, 1968.

Olmstead, P.S., Distribution of sample arrangements for runs up and down, *Annals of Mathematical Statistics*, *17*, 24-33, 1946.

Wallis, W.A., and G.H. Moore, A significance test for time series, *J. Amer. Statist. Assoc*., *36*, 401-409, 1941.

Young, L.C., On randomness in ordered sequences, *Annals of Mathematical Statistics*, *12*, 153-162, 1941.

See, in particular, Chapter 25 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

<http://www.csis.hku.hk/cisc/projects/va/index.htm>

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsNPNominalSRTest** and **StatsRunsCDF**.

StatsStudentCDF

StatsStudentCDF(*t*, *n*)

The StatsStudentCDF function returns the Student (uniform) cumulative distribution function

$$F(t,n)=\begin{cases}\frac{1}{2}\left\{1+I\left(\frac{n}{2},\frac{1}{2};1\right)-I\left(\frac{n}{2},\frac{1}{2};\frac{n}{n+t^2}\right)\right\} & t>0\\ \frac{1}{2}\left\{1+I\left(\frac{n}{2},\frac{1}{2};\frac{n}{n+t^2}\right)-I\left(\frac{n}{2},\frac{1}{2};1\right)\right\} & t<0\\ \frac{1}{2} & t=0\end{cases}$$

where n>0 is degrees of freedom and is the incomplete beta function **betai**.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentPDF** and **StatsInvStudentCDF** functions.

StatsStudentPDF

StatsStudentPDF(*t*, *n*)

The StatsStudentPDF function returns the Student (uniform) probability distribution function

$$f(t,n)=\frac{\left(\frac{n}{n+t^2}\right)^{(n+1)/2}}{\sqrt{n}B\left(\frac{n}{2},\frac{1}{2}\right)}.$$

where n>0 is degrees of freedom and $B()$ is the **beta** function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentCDF** and **StatsInvStudentCDF** functions.

StatsTopDownCDF

StatsTopDownCDF(r, N)

The StatsTopDownCDF function returns the cumulative distribution function for the top-down correlation coefficient. It is computationally intensive because it must evaluate many permutations [$O((n!)^2)$]. It exactly calculates the distribution for $3 \le N \le 7$; outside this range it uses Monte-Carlo estimation for $8 \le N \le 50$ and asymptotic Normal approximation for N>50. The Monte-Carlo estimate uses 1e6 random permutations fitted with two 9-order polynomials for the range [-1,0] and [0,1]. The results are within 0.2% of exact values where known.

References

Iman, R.L., and W.J. Conover, A measure of top-down correlation, *Technometrics*, *29*, 351-357, 1987.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRankCorrelationTest** and **StatsInvTopDownCDF** functions.

StatsTriangularCDF

StatsTriangularCDF(x, a, b, c)

The StatsTriangularCDF function returns the triangular cumulative distribution function

$$F(x;a,b,c) = \begin{cases} \dfrac{(x-a)^2}{(b-a)(c-a)} & a \le x \le c \\ 1 - \dfrac{(b-x)^2}{(b-a)(c-a)} & c \le x \le b. \end{cases}$$

where *a*<*c*<*b*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTriangularPDF** and **StatsInvTriangularCDF** functions.

StatsTriangularPDF

StatsTriangularPDF(*x, a, b, c*)

The StatsTriangularPDF function returns the triangular probability distribution function

$$f(x;a,b,c) = \begin{cases} \dfrac{2(x-a)}{(b-a)(c-a)} & a \le x \le c \\ \dfrac{2(b-x)}{(b-a)(c-a)} & c < x < b \\ 0 & otherwise. \end{cases}$$

where *a*<*c*<*b*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTriangularCDF** and **StatsInvTriangularCDF** functions.

StatsTrimmedMean

StatsTrimmedMean(*waveName, trimValue*)

The StatsTrimmedMean function returns the mean of the wave *waveName* after removing *trimValue* fraction of the values from both tails of the distribution. *trimValue* is a number in the range [0, 0.5]. *waveName* can be any real numeric type.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsQuantiles** and **mean**.

StatsTTest

StatsTTest [***flags***] ***wave1*** [**,** ***wave2***]

The StatsTTest operation performs two kinds of T-tests: the first compares the mean of a distribution with a specified mean value (/MEAN) and the second compares the means of the two distributions contained in *wave1* and *wave2*, which must contain at least two data points, can be any real numeric type, and can have an arbitrary number of dimensions. Output is to the W_StatsTTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/CI Computes the confidence intervals for the mean(s).

/DFM=*m* Specifies method for calculating the degrees of freedom.

- *m*=0: Default; computes equivalent degrees of freedom accounting for possibly different variances.
- *m*=1: Computes equivalent degrees of freedom but truncates to a smaller integer.

m=2: Computes degrees of freedom by $DF=n_1+n_2-2$, where *n* is the sum of points in the wave. Appropriate when variances are equal.

/MEAN=*meanV* Compares *meanV* with the mean of the distribution in *wave1*. Outputs are the number of points in the wave, the degrees of freedom (accounting for any NaNs), the average, standard deviation (σ),

$$s_{\overline{X}} = \frac{\sigma}{\sqrt{DF+1}},$$

the statistic

$$t = \frac{\overline{X} - meanV}{s_{\overline{X}}}$$

and the critical value, which depends on /TAIL.

/PAIR Specifies that the input waves are pairs and computes the difference of each pair of data to get the average difference $\bar{d}$ and the standard error of the difference $S_{\bar{d}}$. The t statistic is the ratio of the two

$$t = \frac{\bar{d}}{s_{\bar{d}}}.$$

In this case H_0 is that the difference $\bar{d}$ is zero.

This mode does not support /CI and /DFM.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/TAIL=*tc* Specifies H_0 for /CIDX.

tc=1: One tailed test ($\mu_1 \le \mu_2$).

tc=2: One tailed test ($\mu_1 \ge \mu_2$).

tc=4: Default; two tailed test ($\mu_1 != \mu_2$).

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

When comparing the mean of a single distribution with a hypothesized mean value, you should use /MEAN and only one wave (*wave1*). If you use two waves StatsTTest performs the T-test for the means of the corresponding distributions (which is incompatible with /MEAN).

When comparing the means of two distributions, the default t-statistic is computed from Welch's approximate t:

$$t' = \frac{\overline{x}_1 - \overline{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}},$$

where s_i^2 are variances, n_i the number of samples, and $\bar{X}_i$ the averages of the respective waves. This expression is appropriate when the number of points and the variances of the two waves are different. If you want to compute the t-statistic using pooled variance you can use the /AEVR flag. In this case the pooled variance is given by

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2},$$

and the t-statistic is

$$t = \frac{\overline{x}_1 - \overline{x}_2}{s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}.$$

The different test are:

H_0	Rejection Condition
$\mu_1 = \mu_2$	$\lvert t \rvert \geq Tc(alpha,\nu)$
$\mu_1 > \mu_2$	$t \leq Tc(alpha, \nu)$
$\mu_1 < \mu_2$	$t \geq Tc(alpha, \nu)$

Tc is the critical value and ν is the effective number of degrees of freedom (see /DFM flag).When accounting for possibly unequal variances, ν is given by

$$\nu = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2 - 1}}.$$

The critical values (Tc) are computed by numerically by solving for the argument at which the cumulative distribution function (CDF) equals the appropriate values for the tests. The CDF is given by

$$F(x) = \begin{cases} \frac{1}{2} betai\left(\frac{\nu}{2}, \frac{1}{2}, \frac{\nu}{\nu + x^2}\right) & x < 0 \\ 1 - \frac{1}{2} betai\left(\frac{\nu}{2}, \frac{1}{2}, \frac{\nu}{\nu + x^2}\right) & x \geq 0. \end{cases}$$

To get the critical value for the upper one-tail test we solve F(x)=1-alpha. For the lower one-tail test we solve for x the equation F(x)=alpha. In the two-tailed test the lower critical value is a solution for F(x)=alpha/2 and the upper critical value is a solution for F(x)=1-alpha/2.

The T-test assumes both samples are randomly taken from normal population distributions.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsStudentCDF**, **StatsStudentPDF**, and **StatsInvStudentCDF**.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999. See in particular Section 8.1.

StatsTukeyTest

StatsTukeyTest [***flags***] [***wave1, wave2,… wave100***]

The StatsTukeyTest operation performs multiple comparison Tukey (HSD) test and optionally the Newman-Keuls test. Output is to the M_TukeyTestResults wave in the current data folder. StatsTukeyTest usually follows **StatsANOVA1Test**.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/NK Computes the Newman-Keuls test.

/Q No results printed in the history area.

/SWN Creates a text wave, T_TukeyDescriptors, containing wave names corresponding to each row of the comparison table (Save Wave Names). Use /T to append the text wave to the last column.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors.

Details

Inputs to StatsTukeyTest are two or more 1D numeric waves (one wave for each group of samples) containing any numbers of points but with at least two or more valid entries.

The contents of the M_TukeyTestResults columns are: the first contains the difference between the group means $\bar{X}_i - \bar{X}_i$, the second contains SE (supports unequal number of points), the third contains the q statistic for the pair, and the fourth contains the critical q value, the fifth contains the conclusion with 0 to reject H_0 ($\mu_i == \mu_j$) or 1 to accept H_0, with /NK, the sixth contains the *p* values

$$p = rank[\bar{X}_i] - rank[\bar{X}_j] + 1,$$

the seventh contains the critical values, and the eighth contains the Newman-Keuls conclusion (with 0 to reject and 1 to accept H_0). The order of the rows is such that all possible comparisons are computed sequentially starting with the comparison of the group having the largest mean with the group having the smallest mean.

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test**, **StatsScheffeTest**, and **StatsDunnettTest**.

StatsUSquaredCDF

StatsUSquaredCDF(*u2, n, m, method, useTable*)

The StatsUSquaredCDF function returns the cumulative distribution function for Watson's U^2 with parameters *u2* (U^2 statistic) and integer sample sizes *n* and *m*. The calculation is computationally intensive, on the order of binomial(*n*+*m*, *m*). Use a nonzero value for *useTable to* search a built-in table of values. If *n* and *m* cannot be found in the table, it will proceed according to *method*:

method	What It Does
0	Exact computation using Burr algorithm (could be slow).
1	Tiku approximation using chi-squared.
2	Use built-in table only and return a NaN if not in table.

For large *n* and *m*, consider using the Tiku approximation. To abort execution, press Command-period (*Macintosh*) or Ctrl+Break (*Windows*).

Precomputed tables, using the algorithm described by Burr, contain these values:

n	*m*
4	4-30
5	5-30
6	6-30
7	7-30
8	8-26
9	9-22
10	10-18
11	11-16
12	12-14
13	13

Because *n* and *m* are interchangeable, *n* should always be the smaller value. For *n*>8 the upper limit in the table matched the maximum that can be computed using the Burr algorithm. There is no point in using method 0 with *m* values exceeding these limits.

References

Burr, E.J., Small sample distributions of the two sample Cramer-von Mises' W^2 and Watson's U^2, *Ann. Mah. Stat. Assoc.*, *64*, 1091-1098, 1964.

Tiku, M.L., Chi-square approximations for the distributions of goodness-of-fit statistics, *Biometrica*, *52*, 630-633, 1965.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWatsonUSquaredTest** and **StatsInvUSquaredCDF** functions.

StatsVariancesTest

StatsVariancesTest [***flags***] [***wave1, wave2,… wave100***]

The StatsVariancesTest operation performs Bartlett's or Levene's test to determine if wave variances are equal. Output is to the W_StatsVariancesTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/METH=*m* Specifies the test type.

- *m*=0: Bartlett test (default).
- *m*=1: Levene's test using the mean.
- *m*=2: Modified Levene's test using the median.
- *m*=3: Modified Levene's test using the 10% trimmed mean.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

- *k*=0: Normal with dialog (default).
- *k*=1: Kills with no dialog.
- *k*=2: Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/WSTR=*waveListString*
Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

All tests define the null hypothesis by

$$H_0: \quad \sigma_1^2 = \sigma_2^2 = ... = \sigma_k^2,$$

against the alternative

$$H_a: \quad \sigma_i^2 \neq \sigma_j^2 \ \ for\ at\ least\ one\ i \neq j.$$

Bartlett's test computes:

$$T = \frac{(n-k)\ln\left(\sigma_w^2\right) - \sum_{i=1}^{k}(n_i - 1)\ln\left(\sigma_i^2\right)}{1 + \frac{1}{3(k-1)}\left[\sum_{i=1}^{k}\frac{1}{n_i - 1} - \frac{1}{N-k}\right]}.$$

Here $\sigma^2{}_i$ is the variance of the *i*th wave, *N* is the sum of the points of all the waves, n_i is the number of points in wave *i*, and *k* is the number of waves. The weighted variance is given by

$$\sigma_w^2 = \sum_{i=1}^{k}\frac{(n_i - 1)\sigma_i^2}{N-k}.$$

H_0 is rejected if T is greater than the critical value taken from the χ^2 distribution computed by solving for *x*:

$$1 - alpha = 1 - gammq\left(\frac{k-1}{2}, \frac{x}{2}\right).$$

Levene's test computes:

$$W = \frac{(N-k)\sum_{i=1}^{k} n_i\left(\overline{Z}_i - \overline{Z}\right)^2}{(k-1)\sum_{i=1}^{k}\sum_{j=1}^{k}\left(Z_{ij} - \overline{Z}_i\right)^2},$$

where

$$Z_{ij} = \left|Y_{ij} - \overline{Y}_i\right|,$$

$$\overline{Z}_i = \frac{1}{n_i}\sum_{j=1}^{k} Z_{ij},$$

$$\overline{Z} = \frac{1}{N}\sum_{i=1}^{k}\sum_{j=1}^{k} Z_{ij}.$$

$\overline{Y}_i$ depends on /METH.

H_0 is rejected if *W* is greater than the critical value from the F distribution computed by solving for *x*:

$$1 - alpha = 1 - betai\left(\frac{\nu_2}{2}, \frac{\nu_1}{2}, \frac{\nu_2}{\nu_2 + \nu_1 x}\right).$$

References

NIST/SEMATECH, Bartlett's Test, in *NIST/SEMATECH e-Handbook of Statistical Methods*, <http://www.itl.nist.gov/div898/handbook/eda/section3/eda357.htm>, 2005.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsVonMisesCDF

StatsVonMisesCDF(*x*, *a*, *b*)

The StatsVonMisesCDF function returns the von Mises cumulative distribution function

$$F(\theta; a, b) = \frac{1}{2\pi I_0(b)} \int_0^{\theta} \exp\left(b\cos(x - a)\right) dx.$$

where $I_0(b)$ is the modified Bessel function of the first kind (**bessI**), and

$$0 < \theta \le 2\pi$$

$$0 < a \le 2\pi$$

$$b > 0.$$

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsVonMisesPDF**, **StatsInvVonMisesCDF**, and **StatsVonMisesNoise** functions.

StatsVonMisesNoise

StatsVonMisesNoise(*a*, *b*)

The StatsVonMisesNoise function returns a pseudo-random number from a von Mises distribution whose probability density is

$$f(\theta; a, b) = \frac{\exp\left[b\cos(\theta - a)\right]}{2\pi I_0(b)},$$

where I_0 is the zeroth order modified Bessel function of the first kind.

References

Best, D.J., and N. I. Fisher, Efficient simulation of von Mises distribution, *Appl. Statist*., *28*, 152-157, 1979.

See Also

StatsVonMisesCDF, **StatsVonMisesPDF**, and **StatsInvVonMisesCDF**.

Noise Functions on page III-332.

Chapter III-12, **Statistics** for a function and operation overview

StatsVonMisesPDF

StatsVonMisesPDF(*q*, *a*, *b*)

The StatsVonMisesPDF function returns the von Mises probability distribution function

$$f(\theta;a,b) = \frac{\exp\left(b\cos\left(\theta - a\right)\right)}{2\pi I_0(b)}.$$

where $I_0(b)$ is the modified Bessel function of the first kind **bessI**, and

$$0 < \theta \le 2\pi$$
$$0 < a \le 2\pi$$
$$b > 0.$$

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsVonMisesCDF**, **StatsInvVonMisesCDF**, and **StatsVonMisesNoise** functions.

StatsWaldCDF

StatsWaldCDF(*x*, *m*, *l*)

The StatsWaldCDF function returns the numerically evaluated inverse Gaussian or Wald cumulative distribution function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWaldPDF** function.

StatsWaldPDF

StatsWaldPDF(*x*, *m*, *l*)

The StatsWaldPDF function returns the inverse Gaussian or Wald probability distribution function

$$f(x;\mu,\lambda) = \sqrt{\frac{\lambda}{2\pi x^3}}\exp\left[-\frac{\lambda\left(x-\mu\right)^2}{2\mu^2 x}\right]$$

where x, m, $l > 0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWaldCDF** function.

StatsWatsonUSquaredTest

StatsWatsonUSquaredTest [*flags*] *srcWave1*, *srcWave2*

The StatsWatsonUSquaredTest operation performs Watson's nonparametric two-sample U^2 test for samples of circular data. Output is to the W_WatsonUtest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.
/Z	Ignores errors.

Details

The input waves, *srcWave1* and *srcWave2*, each must contain at least two angles in radians (mod 2π), can have any number of dimensions, and can be single or double precision. They must not contain any NaNs or INFs.

The Watson U^2 H_0 postulates that the two samples came from the same population against the different populations alternative. In the calculation, StatsWatsonUSquaredTest ranks the two inputs, accounts for possible ties, computes the test statistic U^2, and compares it with the critical value. Because of the difficulty of computing the critical values, it always computes first the approximation due to Tiku and if possible it computes the exact critical value using the method outlined by Burr. You can evaluate the U2 CDF to get more information about the critical region.

V_flag will be set to -1 for any error and to zero otherwise.

References

We have found that this method leads to slightly different results depending on the compiler and the system on which it is implemented:

Burr, E.J., Small sample distributions of the two sample Cramer-von Mises' W2 and Watson's U2, *Ann. Mah. Stat. Assoc.*, *64*, 1091-1098, 1964.

Tiku, M.L., Chi-square approximations for the distributions of goodness-of-fit statistics, *Biometrica*, *52*, 630-633, 1965.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWatsonWilliamsTest**, **StatsWheelerWatsonTest**, **StatsUSquaredCDF**, and **StatsInvUSquaredCDF**.

StatsWatsonWilliamsTest

StatsWatsonWilliamsTest [***flags***] [**srcWave1, srcWave2, srcWave3,…**]

The StatsWatsonWilliamsTest operation performs the Watson-Williams test for two or more sample means. Output is to the W_WatsonWilliams wave in the current data folder or optionally to a table.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors.

Details

The StatsWatsonWilliamsTest must have at least two input waves, which contain angles in radians, can be single or double precision, and can be of any dimensionality; the waves must not contain any NaNs or INFs.

The Watson-Williams H_0 postulates the equality of the means from all samples against the simple inequality alternative. The test computes the sums of the sines and cosines from which it obtains a weighted r value (rw). According to Mardia, you should use different statistics depending on the size of rw: for rw>0.95 use the simple F statistic, but for 0.95>rw>0.7 you should use the F-statistic with the K correction factor. Otherwise you should use the t-statistic. StatsWatsonWilliamsTest computes both the (corrected) F-statistic and the t-statistic as well as their corresponding critical values.

V_flag will be set to -1 for any error and to zero otherwise.

References

See, in particular, Section 6.3 of:

Mardia, K.V., *Statistics of Directional Data*, Academic Press, New York, New York, 1972.

See, in particular, Chapter 27 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWatsonUSquaredTest** and **StatsWheelerWatsonTest**.

StatsWeibullCDF

StatsWeibullCDF(*x, m, s, g*)

The StatsWeibullCDF function returns the Weibull cumulative distribution function

$$F(x;\mu,\sigma,\gamma) = 1 - \exp\left[-\left(\frac{x-\mu}{\sigma}\right)^{\gamma}\right], \qquad x \ge \mu \text{ and } \sigma,\gamma > 0.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWeibullPDF** and **StatsInvWeibullCDF** functions.

StatsWeibullPDF

StatsWeibullPDF(*x, m, s, g*)

The StatsWeibullPDF function returns the Weibull probability distribution function

$$f(x;\mu,\sigma,\gamma) = \frac{\gamma}{\sigma}\left(\frac{x-\mu}{\sigma}\right)^{\gamma-1}\exp\left[-\left(\frac{x-\mu}{\sigma}\right)^{\gamma}\right],$$

where *m* is the location parameter, *s* is the scale parameter, and *g* is the shape parameter with $x \ge m$ and $s, g > 0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWeibullCDF** and **StatsInvWeibullCDF** functions.

StatsWheelerWatsonTest

StatsWheelerWatsonTest [***flags***] [**srcWave1, srcWave2, srcWave3,…**]

The StatsWheelerWatsonTest operation performs the nonparametric Wheeler-Watson test for two or more samples. Output is to the W_WheelerWatson wave in the current data folder or optionally to a table.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors.

Details

The StatsWatsonWilliamsTest must have at least two input waves, which contain angles in radians (mod 2π), can be single or double precision, and can be of any dimensionality; the waves must not contain any NaNs or INFs.

The Wheeler-Watson H_0 postulates that the samples came from the same population. The extension of the test to more than two samples is due to Mardia. The Wheeler-Watson test is not valid for data with ties, in which case you should use Watson's U^2 test.

V_flag will be set to -1 for any error and to zero otherwise.

References

Mardia, K.V., *Statistics of Directional Data*, Academic Press, New York, New York, 1972.

See, in particular, Chapter 27 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWatsonUSquaredTest** and **StatsWheelerWatsonTest**.

StatsWilcoxonRankTest

StatsWilcoxonRankTest [***flags***] ***waveA, waveB***

The StatsWilcoxonRankTest operation performs the nonparametric Wilcoxon-Mann-Whitney two-sample rank test or the Wilcoxon Signed Rank test (for paired data) on *waveA* and *waveB*. Output is to the W_WilcoxonTest wave in the current data folder or optionally to a table.

waveA and *waveB* must not contain NaNs or INFs.

Flags

/ALPH = *val* — Sets the significance level (default *val*=0.05).

/APRX=*m* — Sets the approximation method. It computes an exact critical value by default.

m=1: Standard normal approximation with ties (Zar P. 151).

m=2: Improved normal approximation (Zar P. 152).

Approximations may be appropriate for large sample sizes when computation may take a long time.

/Q — No results printed in the history area.

/T=*k* — Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/TAIL=*tail* — Specifies the tail tested.

tail=1: Lower tail.

tail=2: Upper tail.

tail=4: Two tail.

You can perform any combination of tests by adding their corresponding tail values (/TAIL=7 tests all tail possiblities). Note that H0 changes according to the selected tail.

/WSRT — Performs the Wilcoxon Signed Rank Test for paired data. The testh computes statistics Tp and Tm, lower-tail, upper-tail, and two-tail P-values. If the number of samples is less than 200 it computes exact P-values, otherwise they are computed using the normal approximation. Do not use /ALPH, /APRX, and /TAIL with this flag.

/Z — Ignores errors.

Details

The Wilcoxon-Mann-Whitney test combines the two samples and ranks them to compute the statistic U. If *waveA* has *m* points and *waveB* has *n* points, then *U* is given by

$$U = mn + \frac{m(m+1)}{2} - R_1,$$

with the corresponding statistic *U*' given by

$$U' = nm + \frac{n(n+1)}{2} - R2.$$

where R_i is the ranks of data in the *i*th wave (ranked in ascending order).

The distribution of *U* is difficult to compute, requiring the number of possible permutations of *m* elements of *waveA* and *n* elements of *waveB* that give rise to *U* values that do not exceed the one computed. The distribution is computed according to the algorithm developed by Klotz. With increasing sample size one can avoid the time consuming distribution computation and use a normal approximation instead. Klotz recommends this approximation for *N*=*m*+*n*~100.

Use /APRX=2 for the best approximation. The two approximations are discussed by Zar.

The Wilcoxon Signed Rank Test, or Wilcoxon Paired-Sample Test, ranks the difference between pairs of values and computes the sums of the positive ranks (Tp) and the negative ranks (Tm). It calculates Tp and Tm and P-values for all tail combinations. The P-values are:

P_lower_tail P(Wp<=Tp)

P_upper_tail P(Wp>=Tp)

P_two_tail 2*Min(P_lower_tail,P_upper_tail)

Wp is the generic symbol for the sum of positive ranks for the given number of pairs.

V_flag will be set to -1 for any error and to zero otherwise.

In both Wilcoxon-Mann-Whitney two-sample rank test and the Wilcoxon Signed Rank test H0 is that the data in the two input waves are statistically the same.

References

Cheung, Y.K., and J.H. Klotz, The Mann Whitney Wilcoxon distribution using linked lists, *Statistica Sinica*, *7*, 805-813, 1997.

See in particular Chapter 15 of:

Klotz, J.H., *Computational Approach to Statistics*, <http://www.stat.wisc.edu/~klotz/Book.pdf>.

Streitberg, B., and J. Rohmel, Exact distributions for permutations and rank tests: An introduction to some recently published algorithms, *Statistical Software Newsletter*, *12*, 10-17, 1986.

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsAngularDistanceTest** and **StatsKWTest**.

StatsWRCorrelationTest

StatsWRCorrelationTest [***flags***] ***waveA, waveB***

The StatsWRCorrelationTest operation performs a Weighted Rank Correlation test on *waveA* and *waveB*, which contain the ranks of sequential factors. The waves are 1-based, integer ranks of factors in the range 1-2^31.

StatsWRCorrelationTest computes a top-down correlation coefficient using Savage sums as well as the critical and P-values. Output is to the W_StatsWRCorrelationTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/Z Ignores errors.

Details

The StatsWRCorrelationTest input waves must be one-dimensional and have the same length. The waves are 1-based, integer ranks of factors corresponding to the point number. Ranks may have ties in which case you should repeat the rank value. For example, if the second and third entries have the same rank you should enter {1,2,2,4}. H_0 stipulates that the same factors are most important in both groups represented by *waveA* and *waveB*.

The top-down correlation is the sum of the product of Savage sums for each row:

$$r_{TD} = \frac{\sum_{i=1}^{n} S_{iA} S_{iB} - n}{n - S_1},$$

where n is the number of rows and the Savage sum S_i is

$$S_i = \sum_{j=i}^{n} \frac{1}{j},$$

and S_{iA} corresponds to the S_i value of the rank of the data in row (i-1) of *waveA*.

References

Iman, R.L., and W.J. Conover, A measure of top-down correlation, *Technometrics*, *29*, 351-357, 1987.

See, in particular, Chapter 19 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsLinearCorrelationTest**, **StatsRankCorrelationTest**, **StatsTopDownCDF**, and **StatsInvTopDownCDF**.

stopMSTimer

stopMSTimer(*timerRefNum*)

The stopMSTimer function frees up the timer associated with the *timerRefNum* and returns the number of elapsed microseconds since startMSTimer was called for this timer.

Parameters

timerRefNum is the value returned by startMSTimer or the special values -1 or -2. If *timerRefNum* is not valid then stopMSTimer returns 0. Passing -1 returns the clock frequency of the timer and, on the Macintosh, performs a calibration. Passing -2 returns the time in microseconds since the computer was started.

Details

If you want to make sure that all timers are free, call stopMSTimer ten times with *timerRefNum* equal to 0 through 9. It is OK to stop a timer that you never started.

On the Macintosh, the first call to stopMSTimer will take longer (approximately 0.1 sec. more) because of a clock frequency calibration. This will also happen when calling with a value of -1. If the extra time for the first call will cause problems, be sure to force the calibration before your first real timing run.

Examples

How long does an empty loop take on your computer?

```
Function TestMSTimer()
   Variable timerRefNum
```

```
    Variable microSeconds
    Variable n
    timerRefNum = startMSTimer
    if (timerRefNum == -1)
       Abort "All timers are in use"
    endif
    n=10000
    do
       n -= 1
    while (n > 0)
    microSeconds = stopMSTimer(timerRefNum)
    Print microSeconds/10000, "microseconds per iteration"
End
```

See Also

The **startMSTimer** and **ticks** functions.

str2num

str2num(*str*)

The str2num function returns a number represented by the string expression *str*.

Details

str2num returns NaN if *str* does not contain the text for a number.

str2num skips leading spaces and tabs and then reads up to the first non-numeric character.

See Also

The **char2num**, **num2char** and **num2str** functions.

The **sscanf** operation for more complex parsing jobs.

Strconstant

Strconstant *ksName*="*literal string*"

The Strconstant declaration defines the string *literal string* under the name *ksName* for use by other code, such as in a switch construct.

See Also

The **Constant** keyword for numeric types, **Constants** on page IV-40, and **Switch Statements** on page IV-34.

String

String [**/G**] ***strName*** [**=*strExpr***][**, *strName*** [**=*strExpr***]…]

The String declaration creates string variables and gives them the specified names.

Flags

/G Creates a global string. Overwrites any existing string with the same name.

Details

The string variable is initialized when it is created if you supply the =*strExpr* initializer. However, when String is used to declare a function parameter, it is an error to attempt to initialize it.

You can create more than one string variable at a time by separating the names and optional initializers with commas.

If used in a procedure, the new string is local to that procedure unless the /G (global) flag is used. If used on the command line, String is equivalent to String/G.

strName can optionally include a data folder path.

StringByKey

StringByKey(*keyStr, kwListStr* [, *keySepStr* [, *listSepStr* [, *matchCase*]]])

The StringByKey function returns a substring extracted from *kwListStr* based on the specified key contained in *keyStr*. *kwListStr* should contain keyword-value pairs such as `"KEY=value1,KEY2=value2"` or `"Key:value1;KEY2:value2"`, depending on the values for *keySepStr* and *listSepStr*.

Use StringByKey to extract a string value from a string containing a `"key1:value1;key2: value2;"` style list such as those returned by functions like **AxisInfo** or **TraceInfo**.

If the key is not found or if any of the arguments is `""` then a zero-length string is returned.

keySepStr, *listSepStr*, and *matchCase* are optional; their defaults are ":", ";", and 0 respectively.

Details

keyStr is limited to 255 characters.

kwListStr is searched for an instance of the key string bound by *listSepStr* on the left and a *keySepStr* on the right. The text up to the next *listSepStr* is returned.

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are usually case-insensitive. Setting the optional *matchCase* parameter to 1 makes the comparisons case sensitive.

Only the first characters of *keySepStr* and *listSepStr* are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *matchCase* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print StringByKey("BKEY", "AKEY:hello;BKEY:nok-nok")  // prints "nok-nok"
Print StringByKey("KY", "KX=1;ky=hello", "=")         // prints "hello"
Print StringByKey("KY", "KX:1,KY:joey,", ":", ",")    // prints "joey"
Print StringByKey("kz", "KZ:1st,kz:2nd,", ":", ",")   // prints "1st"
Print StringByKey("kz", "KZ:1st,kz:2nd,", ":", ",", 1)// prints "2nd"
```

See Also

The **NumberByKey**, **RemoveByKey**, **ReplaceNumberByKey**, **ReplaceStringByKey**, **ItemsInList**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

stringCRC

stringCRC(*inCRC*,*str*)

The stringCRC function returns 32-bit cyclic redundancy check value of bytes in *str* starting with *inCRC* which should be zero for the first (or only) set of bytes.

Details

Polynomial used is:

x^32+x^26+x^23+x^22+x^16+x^12+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x+1

See crc32.c in the public domain source code for zlib for more information.

See Also

The **WaveCRC** function.

StringFromList

StringFromList(*index, listStr* [, *listSepStr*])

The StringFromList function returns the *index*th substring extracted from *listStr*. *listStr* should contain items separated by the *listSepStr* character, such as "abc;def;".

Use StringFromList to extract an item from a string containing a list of items separated by a single character, such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

index is zero-based (set index to zero to get the first item in the string list).

If *index* < 0, or *index* ≥ the number of items in list, or if *listStr* or *listSepStr* is `""`, then a zero-length string is returned.

listSepStr is optional. When omitted, *listSepStr* is presumed to be ";".

Details

listStr is searched for an instance of the key string bound by *listSepStr* on the left and right. The text between the bounding *listSepStr*s is returned.

listStr is treated as if it ends with a *listSepStr* even if it doesn't.

Only the first character of *listSepStr* is used.

Examples

```
Print StringFromList(0, "wave0;wave1;")              // prints "wave0"
Print StringFromList(2, "wave0;wave1;")              // prints ""
Print StringFromList(1, "wave0;;wave2")              // prints ""
Print StringFromList(2, "fred\twilma\tbarney", "\t") // prints "barney"
```

A function that applies a unique line style to each trace in the top graph:

```
Function LineStyler()
   String traces= TraceNameList("",";",1)  // Traces in top graph
   String traceName
   Variable i=0
   do
      traceName= StringFromList(i,traces)
      if( strlen(traceName) == 0 )
         break
      endif
      ModifyGraph lstyle($traceName)= mod(i,18)   // 0 to 17, then repeat
      i += 1
   while (1)           // exit is via break statement
End
```

A function that converts a string list into a text wave:

```
Function List2Wave()
   String traces= TraceNameList("",";",1)  // Traces in top graph
   Variable n= ItemsInList(traces)
   Make/O/T/N=(n) textWave= StringFromList(p,traces)
End
```

See Also

The **AddListItem**, **ItemsInList**, **FindListItem**, **RemoveFromList**, **WaveList**, **WhichListItem**, **StringByKey**, **ListMatch**, **ControlNameList**, **TraceNameList**, **StringList**, **VariableList**, and **FunctionList** functions.

StringList

StringList(*matchStr, separatorStr*)

The StringList function returns a string containing a list of global string variables selected based on the *matchStr* parameter. The string variables listed are all in the current data folder.

Details

For a string variable name to appear in the output string, it must match *matchStr*. The first character of *separatorStr* is appended to each string variable name as the output string is generated.

The name of each string variable is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches all string variable names
"xyz"	Matches name xyz only
"*xyz"	Matches names which end with xyz
"xyz*"	Matches names which begin with xyz
"*xyz*"	Matches names which contain xyz
"abc*xyz"	Matches names which begin with abc and end with xyz

The list contains names only, without data folder paths. Thus, they are not suitable for accessing string variables outside the current data folder.

matchStr may begin with the ! character to return windows that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches variable names which *do not* end with xyz

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

Examples

`StringList("*",";")`	Returns a list of all string variables in the current data folder.
`StringList("S_*", ";")`	Returns a list of all string variables in the current data folder whose names begin with "S_".

See Also

See the **VariableList** and **WaveList** functions.

stringmatch

stringmatch(*string, matchStr*)

The stringmatch function tests *string* for a match to *matchStr*. You may include asterisks in *matchStr* as a wildcard character. Returns 1 to indicate a match, or 0 for no match.

Details

matchStr is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches any string.
"xyz"	Matches the string "xyz" only.
"*xyz"	Matches strings ending with "xyz", for instance "abcxyz".
"xyz*"	Matches strings beginning with xyz, for instance "xyzpqr".
"*xyz*"	Matches strings containing xyz, for instance "abcxyzpqr".
"abc*xyz"	Matches strings beginning with abc and ending with xyz, for instance "abcpqrxyz".

If *matchStr* begins with the ! character, a match is indicated if *string* does *not* match *matchStr*. For example:

"!*xyz"	Matches strings which *do not* end with xyz.

The ! character is considered to be a normal character if it appears anywhere else.

Note that matching is case-insensitive, so "xyz" also matches "XYZ" or "Xyz".

Also note that it is impossible to match an asterisk in *string*: use **GrepString** instead.

Among other uses, the stringmatch function can be used to build your own versions of the **WaveList** function, using **NameOfWave** and stringmatch to qualify names of waves found by **WaveRefIndexed**.

See Also

The **GrepString**, **cmpstr**, **strsearch**, **SplitString**, **ListMatch**, and **ReplaceString** functions and the **sscanf** operation.

strlen

strlen(*str*)

The strlen function returns the number of characters in the string expression *str*.

strlen returns NaN if the *str* is NULL. A local string variable or a string field in a structure that has never been set is NULL. NULL is not the same as zero length. Use **numtype** to test if the result from strlen is NaN.

Examples

```
String zeroLength = ""
String neverSet
Print strlen(zeroLength), strlen(neverSet)

// Test if a string is null
Variable len = strlen(neverSet)   // NaN if neverSet is null
if (numtype(len) == 2)            // strlen returned NaN?
   Print "neverSet is null"
endif
```

strsearch

strsearch(*str, findThisStr, start*[*, options*])

The strsearch function returns the numeric position of the string expression *findThisStr* in the string expression *str*.

Details

strsearch performs a case-sensitive search.

strsearch returns -1 if *findThisStr* does not occur in *str*.

The search starts from the character position in *str* specified by *start*; 0 is the first character in *str*.

strsearch limits *start* to one less than the length of *str*, so it is useful to use `Inf` for *start* when searching backwards to ensure that the search is from the end of *str*.

The optional *options* parameter is a bitmask specifying the search options:

1: Search backwards from *start*.

2: Ignore case.

3: Search backwards and ignore case.

Examples

```
String str="This is a test isn't it?"
Print strsearch(str,"test",0)                  // prints 10
Print strsearch(str,"TEST",0)                  // prints -1
Print strsearch(UpperStr(str),"TEST",0)        // prints 10
Print strsearch(str,"TEST",0,2)                // prints 10
Print strsearch(str,"is",0)                    // prints 2
Print strsearch(str,"is",3)                    // prints 5
Print strsearch(str,"is",Inf,1)                // prints 15
```

See Also

The **sscanf** operation and **FindListItem** and **ReplaceString** functions.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

strswitch-case-endswitch

```
strswitch(<string expression>)
   case <literal><constant>:
      <code>
      [break]
   [default:
      <code>]
endswitch
```

A strswitch-case-endswitch statement evaluates a string expression. If a case label matches *string expression*, then execution proceeds with *code* following the matching case label. When none of the cases match, execution will continue at the default label, if it is present, or otherwise the strswitch will be exited with no action taken. Note that although the break statement is optional, in almost all case statements it will be required for the strswitch to work correctly.

See Also

Switch Statements on page IV-34, **default** and **break** for more usage details.

STRUCT

```
STRUCT structureName localName
```

STRUCT is a reference that creates a local reference to a Structure accessed in a user-defined function. When a Structure is passed to a user function, it can only be passed by reference, so in the declaration within the function you must use `&localStructName` to define the function input parameter.

See Also

Structures in Functions on page IV-78 for further information.

See the **Structure** keyword for creating a Structure definition.

StructGet

StructGet [**/B=***b*] ***structVar, waveStruct***[[***colNum***]]

StructGet /S [**/B=***b*] ***structVar, strStruct***

The StructGet operation reads binary numeric data from a specified column of a wave or from a string variable and copies the data into the designated structure variable. The source wave or string will have been filled beforehand by **StructPut**.

Parameters

structVar is the name of an existing structure that is to be filled with new data values.

waveStruct is the name of a wave containing binary numeric data that will be used to fill *structVar*. Use the optional *colNum* parameter to specify a column from the structure wave. The contents of *waveStruct* are created beforehand using StructPut.

strStruct is the name of a string variable containing binary numeric data. The contents of *strStruct* are created beforehand using StructPut.

Flags

/B=*b* Sets the byte ordering for reading of structure data.

- *b*=0: Reads in native byte order.
- *b*=1: Reads bytes in reversed order.
- *b*=2: Default; reads data in big-endian, high-byte order (Motorola/Macintosh).
- *b*=3: Reads data in little-endian, low-byte order (Intel/Windows).

/S Reads binary data from a string variable, which was set previously with StructPut.

Details

The data that are stored in *waveStruct* and *strStruct* are in binary format so you can not directly view a meaningful representation of their contents by printing them or viewing the wave in a table. To view the contents of *waveStruct* or *strStruct* you must use StructGet to export them back into a structure and then retrieve the members.

If *colNum* is out of bounds it will be clipped to valid values and an error reported. If the row dimension does not match the structure size, as much data as possible will be copied to the structure.

By default, data are read in big-endian, high-byte order (Motorola/Macintosh). This allows data written on one platform to be read on the other.

See Also

The **StructPut** operation for writing structure data to waves or strings.

StructPut

StructPut [**/B=***b*] ***structVar, waveStruct***[[***colNum***]]

StructPut /S [**/B=***b*] ***structVar, strStruct***

The StructPut operation copies the binary numeric data in a structure variable to a specified column in a wave or to a string variable. The data in the wave or string can be read out into another structure using **StructGet**.

Parameters

structVar is the name of a structure from which data will be exported.

waveStruct is the name of an existing wave to which data will be exported. Use the optional *colNum* parameter to specify a column in *waveStruct* to contain the data. The first column of *waveStruct* will be filled if *colNum* is omitted.

strStruct is the name of an existing string variable to which data will be exported.

Flags

/B=*b* Sets the byte ordering for writing of structure data.

- *b*=0: Writes in native byte order.
- *b*=1: Writes bytes in reversed order.
- *b*=2: Default; writes data in big-endian, high-byte order (Motorola/Macintosh).
- *b*=3: Writes data in little-endian, low-byte order (Intel/Windows).

/S Writes binary data to a string variable.

Details

The structure to be exported must contain only numeric data in either integer, floating point, or double precision format. If the structure contains any objects such as String, NVAR, WAVE, etc., then an error will result at compile time.

If needed, StructPut will redimension *waveStruct* to unsigned byte format, will set the number of rows to equal the size of the structure, and set the column dimension large enough to accommodate the size specified by *colNum*. You can think of *waveStruct* as a one-dimensional array of structure contents indexed by *colNum* although the wave is actually two-dimensional with each column containing a copy of a separate structure.

By default, data are written in big-endian, high-byte order (Motorola/Macintosh). This allows data written on one platform to be read on the other.

After you have exported the structure data to *waveStruct* or *strStruct* they will contain binary data that you cannot inspect directly. To view the contents of *waveStruct* or *strStruct*, you must use the original structure or use StructGet to export them into another structure.

See Also

The **StructGet** operation for reading structure data from waves or strings.

Structure

```
Structure structureName
   memType memName [arraySize] [, memName [arraySize]]
   …
EndStructure
```

The Structure keyword introduces a structure definition in a user function. Within the body of the structure you declare the member type (*memType*) and the corresponding member name(s) (*memName*). Each *memName* may be declared with an optional array size.

Details

Structure member types (*memType*) can be any of the following Igor objects: Variable, String, WAVE, NVAR, SVAR, FUNCREF, or STRUCT.

Igor structures also support additional member types, as given in the next table, for compatibility with C programming structures and disk files.

Igor Member Type	C Equivalent	Byte Size
char	signed char	1
uchar	unsigned char	1
int16	short int	2
uint16	unsigned short int	2
int32	long int	4
uint32	unsigned long int	4
float	float	4
double	double	8

The Variable and double types are identical although Variable can be also specified as complex (using the /C flag).

Each structure member may have an optional *arraySize* specification, which gives the number of elements contained by the structure member. The array size is an integer number from 1 to 400 except for members of type STRUCT for which the upper limit is 100.

See Also

Structures in Functions on page IV-78 for further information.

See the **STRUCT** declaration for creating a local reference to a Structure.

StrVarOrDefault

StrVarOrDefault(*pathStr, defStrVal*)

The StrVarOrDefault function checks to see if *pathStr* points to a string variable and if so, it returns its value. If the string variable does not exist, returns *defStrVal* instead.

Details

StrVarOrDefault initializes input values of macros so they can remember their state without needing global variables to be defined first. Numeric variables use the corresponding numeric function, **NumVarOrDefault**.

Examples

```
Macro foo(nval,sval)
    Variable nval=NumVarOrDefault("root:Packages:mypack:nvalSav",2)
    String sval=StrVarOrDefault("root:Packages:mypack:svalSav","Hi")
    String dfSav= GetDataFolder(1)
    NewDataFolder/O/S root:Packages
    NewDataFolder/O/S mypack
    Variable/G nvalSav= nval
    String/G svalSav= sval
    SetDataFolder dfSav
End
```

StudentA

StudentA(*t, DegFree*)

The StudentA function returns the area from *-t* to *t* under the Student's T distribution having *DegFree* degrees of freedom. That is, it returns the probability that a random sample from Student's T is between *-t* and *t*.

Note that this is the *bi-tail* result. That is, it gives the area from *-t* to *t*, rather than the cumulative area from -× to *t*. It is this latter number that is commonly tabulated- StudentA returns the probability 1-α where the area from -× to t is the probability 1-α/2.

StudentA tests whether a normally-distributed statistic is significantly different from a certain value. You could use it to test whether an intercept from a line fit is significantly different from zero:

```
Make/O/N=20 Data=0.5*x+2+gnoise(1)      // line with Gaussian noise
Display Data
CurveFit line Data /D
print "Prob = ", StudentA(W_coef[0]/W_sigma[0], V_npnts-2)
```

Because the noise is random, the results will differ slightly each time this is tried. When we did it, the result was:

```
Prob = 0.999898
```

which indicates that the intercept of the line fit was different from zero with 99.99 per cent probability.

StudentT

StudentT(*Prob, DegFree*)

The StudentT function returns the t value corresponding to an area *Prob* under the Student's T distribution from *-t* to *t* for *DegFree* degrees of freedom.

Note that this is a *bi-tail* result, which is what is usually desired. Tabulated values of the Student's T distribution are commonly the one-sided result.

StudentT calculates confidence intervals from standard deviations for normally-distributed statistics. For instance, you can use it to calculate a confidence interval for the coefficients from a curve fit:

```
Make/O/N=20 Data=0.5*x+2+gnoise(1)      // line with Gaussian noise
Display Data
CurveFit line Data /D
print "intercept = ", W_coef[0], "±", W_sigma[0]*StudentT(0.95, V_npnts-2)
print "slope = ", W_coef[1], "±", W_sigma[1]*StudentT(0.95, V_npnts-2)
```

See Also

Chapter III-12, **Statistics** for a function and operation overview.

Submenu

Submenu *menuNameStr*

The Submenu keyword introduces a submenu definition. It is used inside a Menu definition. See Chapter IV-5, **User-Defined Menus** for further information.

sum

sum(*waveName* [, *x1, x2*])

The sum function returns the sum of the wave elements for points from x=*x1* to x=*x2*.

Details

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x2*. To use point indexing, replace *x1* with `pnt2x(waveName,pointNumber1)`, and a similar expression for *x2*.

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If the points nearest to *x1* or *x2* are not within the point range of 0 to numpnts(*waveName*)-1, sum limits them to the nearest of point 0 or point numpnts(*waveName*)-1.

If any values in the point range are NaN, sum returns NaN.

Examples

```
Make/O/N=100 data;SetScale/I x 0,Pi,data
data=sin(x)
Print sum(data,0,Pi)          // the entire point range, and no more
Print sum(data)               // same as -infinity to +infinity
Print sum(data,Inf,-Inf)      // +infinity to -infinity
```

The following is printed to the history area:

```
•Print sum(data,0,Pi)         // the entire point range, and no more
  63.0201
•Print sum(data)              // same as -infinity to +infinity
  63.0201
•Print sum(data,Inf,-Inf)     // +infinity to -infinity
  63.0201
```

See Also

mean and **area** functions.

SVAR

SVAR [/Z] *localName* [= *pathToStr*][, *localName1* [= *pathToStr1*]]…

SVAR is a declaration that creates a local reference to a global string variable accessed in a user-defined function.

The SVAR reference is required when you access a global string variable in a function. At compile time, the SVAR statement specifies a local name referencing a global string variable. At runtime, it makes the connection between the local name and the actual global variable. For this connection to be made, the global string variable must exist when the SVAR statement is executed.

When *localName* is the same as the global string variable name and you want to reference a global variable in the current data folder, you can omit *pathToStr*. Prior to Igor Pro 4.0, *pathToStr* was always required.

pathToStr can be a full literal path (e.g., root:FolderA:var0), a partial literal path (e.g., :FolderA:var0) or $ followed by string variable containing a computed path (see **Converting a String into a Reference Using $** on page IV-47).

You can also use a data folder reference or the /SDFR flag to specify the location of the string variable if it is not in the current data folder. See **Data Folder References** on page IV-61 and **The /SDFR Flag** on page IV-63 for details.

If the global variable may not exist at runtime, use the /Z flag and call **SVAR_Exists** before accessing the variable. The /Z flag prevents Igor from flagging a missing global variable as an error and dropping into the Igor debugger. For example:

```
SVAR/Z nv=<pathToPossiblyMissingStringVariable>
if( SVAR_Exists(sv) )
   <do something with sv>
endif
```

Note that to create a global string variable, you use the **String**/G operation.

Flags

/Z ignores string reference checking failures.

See Also

SVAR_Exists function.

Accessing Global Variables and Waves on page IV-50.

Converting a String into a Reference Using $ on page IV-47.

SVAR_Exists

SVAR_Exists(*name*)

The SVAR_Exists function returns 1 if the specified SVAR reference is valid or 0 if not. It can be used only in user-defined functions.

For example, in a user function you can test if a global string variable exists like this:

```
SVAR /Z str1 = gStr1       // /Z prevents debugger from flagging bad SVAR
if (!SVAR_Exists(str1))    // No such global string variable?
   String/G gStr1 = ""     // Create and initialize it
endif
```

See Also

WaveExists, **NVAR_Exists**, and **Accessing Global Variables and Waves** on page IV-50.

switch-case-endswitch

```
switch(<numeric expression>)
   case <literal><constant>:
      <code>
      [break]
   [default:
      <code>]
endswitch
```

A switch-case-endswitch statement evaluates a numerical expression. If a case label matches *numerical expression*, then execution proceeds with *code* following the matching case label. When no cases match, execution will continue at the default label, if present, or otherwise the switch will be exited with no action taken. Note that although the break statement is optional, in almost all case statements it will be required for the switch to work correctly.

See Also

Switch Statements on page IV-34, **default** and **break** for more usage details.

t

t

The t function returns the T value for the current chunk of the destination wave when used in a multidimensional wave assignment statement. T is the scaled chunk index while **s** is the chunk index itself.

Details

Unlike **x**, outside of a wave assignment statement, t does not act like a normal variable.

See Also

Waveform Arithmetic and Assignments on page II-93.

For other dimensions, the **p**, **q**, **r**, and **s** functions.

For scaled dimension indices, the **x**, **y**, and **z** functions.

TabControl

TabControl [/Z] *ctrlName* [*keyword = value* [, *keyword = value* …]]

The TabControl operation creates tab panels for controls.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the TabControl to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}
: Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

  kind can be one of `default`, `native`, or `os9`.

  platform can be one of `Mac`, `Win`, or `All`.

  See **DefaultGUIControls Default Fonts and Sizes** for how enclosed controls are affected by native TabControl appearance.

  See **Button** for more appearance details.

disable=*d*
: Sets user editability of the control.

  d=0: Normal.

  d=1: Hidden.

  d=2: Draw in gray state; disable control action.

font= "*fontName*"
: Sets the font used for tabs, e.g., `font="Helvetica"`.

fsize= *s*
: Sets the font size for tabs.

fstyle=*fs*
: *fs* is a binary coded number with each bit controlling one aspect of the font style as follows:

  bit 0: Bold.

  bit 1: Italic.

  bit 2: Underline.

  bit 3: Outline (*Macintosh only*).

  bit 4: Shadow (*Macintosh only*).

  See **Setting Bit Parameters** on page IV-12 for details about bit settings.

labelBack=(*r*,*g*,*b*) or 0
: Sets fill color for current tab and the interior. *r*, *g*, and *b* are integers from 0 to 65535. If not set, then interior is transparent and the current tab is filled with the window background. Note that if you use a fill color, draw objects can not be used because they will be covered up.

noproc
: Specifies that no function is to run when clicking a tab.

pos={*left*,*top*}
: Sets the position of the control in pixels.

pos+={*dx*,*dy*}
: Offsets the position of the control in pixels.

proc=*procName*
: Specifies the function to run when the tab is pressed. Your function must hide and show other controls as desired. The TabControl does not do this automatically.

size={*width*,*height*}
: Sets TabControl size in pixels.

tabLabel(*n*)=*lbl*
: Sets *n*th tab label to *lbl*. Set the label of the last tab to `""` to reduce the number of tabs.

userdata(*UDName*)=*UDStr*
: Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*
: Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

value=*v*
: Sets current tab number. Tabs count from 0.

win=*winName*
: Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

  When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

The action procedure for a TabControl can use a predefined structure WMTabControlAction as a parameter to the function. The control will use this more efficient method when the function properly matches the structure prototype for a TabControl, otherwise it will use the old-style method.

A TabControl action procedure using a structure has the format:

```
Function newActionProcName(TC_Struct) : TabControl
   STRUCT WMTabControlAction &TC_Struct
   …
End
```

The ": TabControl" designation tells Igor to include this procedure in the Procedure pop-up menu in the Tab Control dialog.

For a TabControl, the WMTabControlAction structure has members as described in the following table:

WMTabControlAction Structure Members

Member	Description
char ctrlName[MAX_OBJ_NAME+1]	Control name.
char win[MAX_WIN_PATH+1]	Host (sub)window.
STRUCT Rect winRect	Local coordinates of host window.
STRUCT Rect ctrlRect	Enclosing rectangle of the control.
STRUCT Point mouseLoc	Mouse location.
Int32 eventCode	Event that executed the procedure. **eventCode** / **Event** -1 Control being killed 2 Mouse up
Int32 eventMod	Bitfield of modifiers. See **Control Structure eventMod Field** on page III-385.
String userData	Primary (unnamed) user data. If this changes, it is written back automatically.
Int32 blockReentry	Prevents reentry of control action procedure. See **Control Structure blockReentry Field** on page III-386.
Int32 tab	Tab number.

Action functions should respond only to documented eventCode values. Other event codes may be added along with more fields. Although the return value is not currently used, action functions should always return zero.

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

When clicking a TabControl with the selector arrow, click in the title region. The control will not be selected if you click in the body. This is to make it easier to select controls in the body rather than the TabControl itself.

TabControls may be used to simplify complex control panels. If you are not already familiar with the concept of a TabControl, see **Modifying Axes** on page II-262 for an illustration of tabs as used in the Modify Axis dialog.

Designing a TabControl with all the accompanying interior controls can be somewhat difficult. Here is a suggested technique:

First, create and set the size and label for one tab. Then create the various controls for this first tab. Before starting on the 2nd tab, create the TabControl's procedure so that it can be used to hide the first set of controls. Then add the 2nd tab, click it to run your procedure and start adding controls for this new tab. When done, update your procedure so the new controls will be hidden when you start on the third tab. You may find it useful to create a recreation macro of your panel to get a convenient listing of the controls. Here is an example:

(The commands given include size and position information obtained by manual adjustment.)

1. Create a panel and a TabControl:

```
NewPanel /W=(150,50,478,250)
ShowTools
TabControl foo,pos={29,38},size={241,142},tabLabel(0)="first tab",value=0
```

2. Add a few controls to the interior of the TabControl:

```
Button button0,pos={52,72},size={50,20},title="First"
CheckBox check0,pos={52,105},size={102,15},title="Check first",value=0
```

3. Write a function:

```
Function fooProc(name,tab)
   String name
   Variable tab

   Button button0,disable= (tab!=0)
   CheckBox check0,disable= (tab!=0)
End
```

4. Set proc and add a new tab:

```
TabControl foo,proc=fooProc,tabLabel(1)="second tab"
```

5. Click the second tab (which hides the first tab's controls) and then add new controls like so:

```
Button button1,pos={58,73},size={50,20},title="Second"
CheckBox check1,pos={60,105},size={114,15},title="Check second",value= 0
```

6. Finally, change `fooProc` by adding these lines at the end:

```
Button button1,disable= (tab!=1)
CheckBox check1,disable= (tab!=1)
```

See Also

The **ControlInfo** operation for information about the control along with the **ModifyControl** and **ModifyControlList** operations. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

Table

Table

Table is a procedure subtype keyword that identifies a macro as being a table recreation macro. It is automatically used when Igor creates a window recreation macro for a table. See **Procedure Subtypes** on page IV-179 and **Killing and Recreating a Table** on page II-197 for details.

TableStyle

TableStyle

TableStyle is a procedure subtype keyword that puts the name of the procedure in the Style pop-up menu of the New Table dialog and in the Table Macros menu. See **Table Style Macros** on page II-229 for details.

TableInfo

TableInfo(*winNameStr, itemIndex*)

The TableInfo function returns a string containing a semicolon-separated list of keywords and values that describe a column in a table or overall properties of the table. The main purpose of TableInfo is to allow an advanced Igor programmer to write a procedure which formats or arranges a table or which manipulates the table selection.

Parameters

winNameStr is the name of an existing table window or `""` to refer to the top table.

itemIndex is one of the following:

itemIndex Value	Returns
-2	Information about the table as a whole.
-1	Information about the Point column
≥0	Information about a column other than the Point column. 0 refers to the first column after the Point column, 1 refers to the second column after the Point column, and so on.

TableInfo returns "" in the following situations:

winNameStr is "" and there are no table windows.

winNameStr is a name but there are no table windows with that name.

itemIndex not -2 and is out of range for an existing column.

Details

If *itemIndex* is -2, the returned string describes the table as a whole and contains the following keywords, with a semicolon after each keyword-value pair.

Keyword	Information Following Keyword
TABLENAME	The name of the table.
HOST	The host specification of the table's host window if it is a subwindow or "" if it is a top-level table window.
ROWS	Number of used rows in the table.
COLUMNS	Number of used columns in the table including the Point column.
SELECTION	A description of the table selection as you would specify it when invoking the ModifyTable operation's selection keyword.
FIRSTCELL	An identification of the first visible data cell in the top/left corner of the table in row-column format. The first data cell is at location 0, 0.
LASTCELL	An identification of the last visible data cell in the bottom/right corner of the table in row-column format.
TARGETCELL	An identification of the target (highlighted) data cell in row-column format.
ENTERING	1 if an entry has been started in the entry line, 0 if not.

If *itemIndex* is -1 up to but not including the number of used columns to the right of the Point column, the returned string describes the specified column and contains the following keywords, with a semicolon after each keyword-value pair.

Keyword	Information Following Keyword
TABLENAME	The name of the table.
HOST	The host specification of the table's host window if it is a subwindow or "" if it is a top-level table window.
COLUMNNAME	Name of the column as you would specify it to the Edit operation if you were creating a table showing just the column of interest.
TYPE	Column's type which will be one of the following: Unused, Point, Index, Label, Data, RealData, ImagData. "Index" identifies a index column such as the X values of a wave. "Label" identifies a column of dimension labels. "Data" identifies a data column of a scalar wave. RealData and ImagData identify a real or imaginary column of a complex wave.
INDEX	Column's position. -1 refers to the Point column, 0 to the first data column, and so on.

Keyword	Information Following Keyword
DATATYPE	Numeric data type of the wave or zero for text waves. See WaveType for a definition of data type codes.
WAVE	A full data folder path to the wave displayed in the column or "" for the Point column.
COLUMNS	The total number of columns in the table from the wave for the column for which you are getting information. This can be used to skip over all of the columns of a multidimensional wave.
HDIM	The wave dimension displayed horizontally as you move from one column to the next. 0 means rows, 1 means columns, 2 means layers, 3 means chunks.
VDIM	The wave dimension displayed vertically in the column. 0 means rows, 1 means columns, 2 means layers, 3 means chunks.
TITLE	As specified for the ModifyTable operation's title keyword.
WIDTH	Column's width in points.
FORMAT	As specified for the ModifyTable operation's format keyword.
DIGITS	As specified for the ModifyTable operation's digits keyword.
SIGDIGITS	As specified for the ModifyTable operation's sigDigits keyword.
TRAILINGZEROS	As specified for the ModifyTable operation's trailingZeros keyword.
SHOWFRACSECONDS	As specified for the ModifyTable operation's showFracSeconds keyword.
FONT	The name of the column's font.
SIZE	Column's font size.
STYLE	As specified for the ModifyTable operation's style keyword.
ALIGNMENT	0=left, 1=center, 2=right.
RGB	The column's color in R,G,B format.
ELEMENTS	As specified for the ModifyTable operation's elements keyword.

Examples

This example makes the table's target cell advance by one position within the range of selected cells each time it is called. To try it, create a table, select a range of cells and then run the function using the Macros menu.

```
Menu "Macros"
    "Test/1", /Q, AdvanceTargetCell("")
End

Function AdvanceTargetCell(tableName)
    String tableName              // Name of table or "" for top table.

    String info = TableInfo(tableName, -2)
    if (strlen(info) == 0)
        return -1                 // No such table
    endif

    String selectionInfo
    selectionInfo = StringByKey("SELECTION", info)

    Variable fRow, fCol, lRow, lCol, tRow, tCol
    sscanf selectionInfo, "%d,%d,%d,%d,%d,%d", fRow, fCol, lRow, lCol, tRow, tCol

    tCol += 1
    if (tCol > lCol)
        tCol = fCol
        tRow += 1
        if (tRow > lRow)
            tRow = fRow
        endif
    endif
```

```
   ModifyTable selection=(-1, -1, -1, -1, tRow, tCol)
End
```

See Also

The **ModifyTable** operation.

Tag

Tag [***flags***] [***traceOrAxisName, xAttach*** [**,** ***textStr***]]

The Tag operation puts a tag on the target or named graph window or subwindow. A tag is an annotation that is attached to a particular point on a trace, image, waterfall plot, or axis in a graph.

Parameters

traceOrAxisName is an optional trace or axis name. A trace name can be optionally followed by the # character and an instance number in order to distinguish multiple instances of the same wave in a graph. It identifies the trace or image to which the tag is to be attached. An axis name can be one of the standard axis names (Bottom, Top, Left, or Right) or a user-defined custom axis name.

A string containing *traceOrAxisName* must be used with the $ operator to specify *traceOrAxisName*.

xAttach is the X value of the point on the trace to which the tag is to be attached. For a multidimensional image, it is the linear index into the matrix array. For an axis, *xAttach* can be the X or Y point depending on the particular axis to which the tag is attached; specifying NaN for *xAttach* will center the tag on the axis.

textStr is the text that is to appear in the tag.

Flags

/A=*anchorCode* Specifies position of tag anchor point.

anchorCode	**Position**
LT	left top
LC	left center
LB	left bottom
MT	middle top
MC	middle center (default)
MB	middle bottom
RT	right top
RC	right center
RB	right bottom

anchorCode is a literal, *not* a string.

The anchor point is on the tag itself. Any line or arrow drawn from the tag to the wave starts at the tag's anchor point. The anchor point also determines the precise spot on the tag which represents its position.

/AO=*ao* Sets the text's auto-orientation mode. A non-zero *a0* value overrides the /O value.

/AO is for trace tags only. Setting /AO for any other kind of annotation has no effect.

An auto-oriented tag's text rotates whenever it is redrawn, usually when the underlying data changes, the graph is resized, or when the tag is attached to a new point.

ao=0: No auto-orientation. Use the /O value (default).

ao=1: Tangent to the trace line at the attachment point.

ao=2: Tangent to the trace line, snaps to vertical or horizontal if within 2 degrees of vertical or horizontal.

ao=3: Perpendicular to the trace line.

ao=4: Perpendicular to the trace line, snaps to vertical or horizontal if within 2 degrees of vertical or horizontal.

/B=(*r*,*g*,*b*) Sets color of the tag's background. *r*, *g*, and *b* specify the amount of red, green, and blue as an integer from 0 to 65535.

/B=*b*
- *b*=0: Opaque background.
- *b*=1: Transparent background.
- *b*=2: Same background as the graph plot area background.
- *b*=3: Same background as the window background.

/C Changes the existing tag.

/F=*frame*
- *frame*=0: No frame.
- *frame*=1: Underline frame.
- *frame*=2: Box frame.

/G=(*r*,*g*,*b*) Sets color of the text in the tag. *r*, *g*, and *b* specify the amount of red, green, and blue as an integer from 0 to 65535.

/H=*legendSymbolWidth*

legendSymbolWidth sets width of the legend symbol (the sample line or marker) in points. Use 0 for the default, automatic width.

/I=*i*
- *i* =1: Tag will be invisible if it is "off screen". "Off screen" means that its attachment point or any part of the tag's text is off screen. This is esthetically pleasing but gives you nothing to grab if you want to drag the tag back on screen.
- *i* =0: Tag will always be visible. If it is "off screen", it appears at the extreme edge of the graph.

/K Kills existing tag.

/L=*line*
- *line*=0: No line from tag to attachment point.
- *line*=1: Line connecting tag to attachment point.
- *line*=2: Line with arrow pointing from tag to attachment point.
- line =3: Line with arrow pointing from attachment point to tag.
- line =4: Line with arrows at both ends.

/LS= *linespace* Specifies a tweak to the normal line spacing where *linespace* is points of extra (plus or minus) line spacing. For negative values, a blank line may be necessary to avoid clipping the bottom of the last line.

/M[=*saMeSize*] /M or /M=1 specifies that legend markers should be the same size as the marker in the graph.

/M=0 turns same-size mode off so that the size of the marker in the legend is based on text size.

/N=*name* Specifies name of the tag to create or change.

/O=*rot* Sets the text's rotation. *rot* is in (integer) degrees, counterclockwise and must be a number from -360 to 360.

0 is normal horizontal left-to-right text, 90 is vertical bottom-to-top text.

If the tag is attached to a trace (not an image or axis), any non-zero /AO value will overwrite this rotation value.

/P=*tipOffset* Sets the offset from the tip of a tag's line or arrow to the point on the wave that it is tagging. *tipOffset* is a positive number from 0 to 200 in points. If *tipOffset*=0 (default), it automatically chooses an appropriate offset.

/Q[=*contourInstance*] Associates a tag with a particular contour level trace in a graph recreation macro. Of interest mainly to hard-core programmers.

When "=*contourInstance*" is present, /Q associates the tag with the contour wave. Igor will feel free to change or delete the tag, as appropriate, when it recalculates the contour (because you changed the contour data or appearance, the graph size or the axis range). *contourInstance* is a contour instance name, such as zWave or zWave#1 if you have the same wave contoured twice in the graph.

/Q by itself, with "=*contourInstance*" not present, disassociates the tag from the contour wave. Igor will no longer modify or delete the tag (unless the contour level to which it is attached is deleted). If you manually tweak a contour label, using the Modify Annotation dialog, Igor uses this flag.

/R=*newName* Renames the tag.

/S=*style*
- *style*=0: Single frame.
- *style*=1: Double frame.
- *style*=2: Triple frame.
- *style*=3: Shadow frame.

/T=*tabSpec* *tabSpec* is a single number in points, such as `/T=72`, for evenly spaced tabs or a list of tab stops in points such as `/T={50, 150, 225}`.

/TL=*extLineSpec* Specifies extended tag line parameters similar to the **SetDrawEnv** arrow settings.

extLineSpec = {*keyword* = *value*,…} or zero to turn off all extended specifications.

Valid *keyword-value* pairs are:

- len=*l* Length of arrow head in points (*l*=0 for auto).
- fat=*f* Width to length ratio of arrow head (default is 0.5 same as *f*=0).
- style=*s* Sets barb side mode (see **SetDrawEnv** astyle for values).
- shar =*s* Sets sharpness between -1 and 1 (default is 0; blunt).
- frame=*f* Sets frame thickness in outline mode.
- lThick=*l* Sets line thickness in points (default is 0.5 for *l*=0).
- lineRGB=(*r*,*g*,*b*)Sets color for lines. Default is all zeros (black); the same as the tag frame.
- dash=*d* Specifies dash pattern number between 0 and 17 (see **SetDashPattern** for patterns).

/W=*winName* Operates on the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/V=*vis*
- *vis*=0: Invisible annotation; not selectable. The annotation is still listed in **AnnotationList**.
- *vis*=1: Visible annotation (default).

/X=*xOffset* Distance from point to tag as percentage of graph width. For axis tags, the offsets are proportional to the size of the text used for the axis labels.

/Y=*yOffset* Distance from point to tag as percentage of graph height. For axis tags, the offsets are proportional to the size of the text used for the axis labels.

/Z=*freeZe*
- *freeZe*=1: Freezes tag position (you can't move it with the mouse).
- *freeZe*=0: Unfreezes it.

Details

If the /C flag is used, it must be the first flag in the command and must be followed immediately by the /N=*name* flag.

If the /K flag is used, it must be the first flag in the command and must be followed immediately by the /N=*name* flag with no further flags or parameters.

traceOrAxisName, *xAttach*, and *textStr* are all optional. If *traceOrAxisName* is specified, then *xAttach* must be specified, and vice versa. *textStr* may be specified only if *traceOrAxisName* and *xAttach* are specified.

This syntax allows changes to the tag to be made through the *flags* parameters without needing to respecify the other parameters. Similarly, the tag's attachment point can be changed without needing to respecify the *textStr* parameter.

xAttach is in terms of the wave's X scaling. If *traceOrAxisName* is displayed as an XY pair, we recommend that you use "point scaling" for the waves, so that *xAttach* can be a point number (because *xAttach* will not be an X axis value).

A tag can have at most 100 lines.

Annotation Escape Codes

textStr can contain the following escape codes which affect subsequent characters in string.

Note: These escape codes contain backslashes, which should themselves be preceded with another backslash. These are automatically inserted by the Add Annotation dialog, but must be manually added when writing a function or in the command line. See the discussion in the **TextBox** operation about backslashes in *textStr*.

The characters "<??>" in a tag indicate that you specified an invalid escape code or used a font that is not available.

The escape codes are:

\B	Use subscript (smaller type).
\F'*fontName*'	Use specified font (e.g., `\F'Helvetica'`).
\f*dd*	*dd* is a binary coded number with each bit controlling one aspect of the column's font style as follows: bit 0: Bold. bit 1: Italic. bit 2: Underline. bit 3: Outline (*Macintosh only*). bit 4: Shadow (*Macintosh only*). For example, bold underline is $2^0 + 2^2 = 1 + 4 = 5$. See **Setting Bit Parameters** on page IV-12 for details about bit settings.
\JR	Right aligned text.
\JC	Center aligned text.
\JL	Left aligned text.
\K(*r*,*g*,*b*)	Use specified color for text. *r*, *g*, and *b* are integers from 0 to 65535.
\M	Use normal (main) script (reverts to main line and size).
\$PICT$name=*pictName*$/PICT$	Inserts specified picture. *pictName* can be a ProcPict or the name of a picture as listed in the Pictures dialog (Misc menu). This is useful for inserting fancy math expressions created by another program. Available in new graphics only (see **Graphics Technology** on page III-421).
\S	Use superscript (uses smaller type).
\sa+*dd*	Adds extra space above line. *dd* is two digits in units of half points (1/144 inch). Can go anywhere in a line.
\sa-*dd*	Reduces space above line. *dd* is two digits in units of half points (1/144 inch). Can go anywhere in a line.
\sb+*dd*	Adds extra space below line. *dd* is two digits in units of half points (1/144 inch). Can go anywhere in a line.
\sb-*dd*	Reduces space below line. *dd* is two digits in units of half points (1/144 inch). Can go anywhere in a line.
\Z*nn*	Use font size *nn* (*nn* must be exactly two digits).
\Zr*nnn*	*nnn* is a 3 digit percentage by which to change the current font size. (*nnn* must be exactly three digits).

textStr can also contain escape codes to manipulate text info variables (see **About Text Info Variables** on page III-64).

When specifying an axis tag, you can also use additional dynamic escape codes that are available with the **Label** operation.

textStr can also contain the following escape codes which insert dynamically computed substrings:

\ON	Inserts name of wave to which tag is attached.
\On	Inserts name of wave and its instance number (if greater than 0).
\OP	Inserts point number to which tag is attached.
\OX	Inserts X value of point to which tag is attached (*xAttach)*.
\OY	Inserts Y value of point to which tag is attached.

\OZ	Inserts Z value of point to which tag is attached for contour level traces. Inserts NaN for other traces.
\{*dynText*}	*dynText* is dynamically evaluated text, which is discussed below.

Dynamically evaluated text (*dynText*) may contain numeric and string expressions. Igor automatically reevaluates *dynText* when a numeric or string variable or wave referenced in *dynText* changes.

Note: When used in *dynText* expressions, user-defined numeric and string variables in macros or functions must be declared as global variables for the expression to evaluate correctly.

dynText can take two forms, an easy one for a single numeric expression and a more complex form that provides precise control over the formatting of the result.

The easy form is:

\{*numeric-expression*}

This evaluates the expression and prints with generic ("%g") formatting.

The full form is:

\{*formatStr, list-of-numeric-or-string-expressions*}

formatStr and *list-of-numeric-or-string-expressions* are treated as for **printf**.

As an aid in typing the expressions, Igor considers carriage returns between the braces to be equivalent to spaces. Rather than typing (in the Add Annotation dialog):

```
\{"twice K0 is %g, and today is %s",2*K0,date()}
```

you can type:

```
\{
   "twice K0 is %g, and today is %s",
   2*K0,
   date()
}
```

Carriage returns can be typed directly in the Add Annotations dialog, or they can be typed as "\r" in a macro, function or the command line. Since \r is not a tag escape sequence, only one backslash is needed. See the examples.

Examples

```
Tag/C/N=t1/X=25/Y=50
```

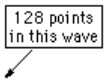
moves the tag named t1 to the location defined by X=25 and Y=50.

```
Tag/C/N=t1 wave1, 50
```

moves the tag named t1 to wave1 at x=50.

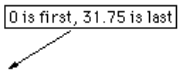
```
Tag/N=t2 wave1, 50,"\\JC\\{numpnts(wave1)} points\rin this wave"
```

creates a new tag on wave1 that looks like this:

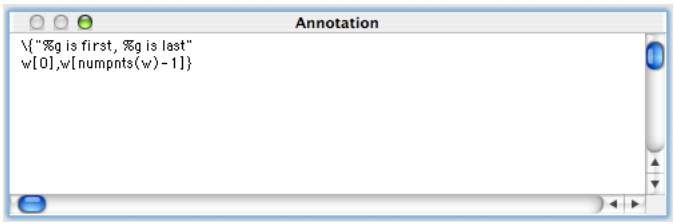



```
Tag w,0,"\\{\"%g is first, %g is last\"\rw[0],w[numpnts(w)-1]}"
```

creates a new tag on w that looks like this:



Observe how the "\r" in the *textStr* breaks the annotation text by examining the Modify Annotation dialog:



Following is an example of various ways in which axis tags can be used:

```
Make/O jack=sin(x/8)
SetScale x,0,14e9,"y" jack
Display jack
```

```
Label bottom "\\u#2"                  // turn off default axis label
ModifyGraph axOffset(bottom)=1.16667 // make room for tag (manual adustment)
Tag/N=text0/F=0/A=MT/X=0.20/Y=-4.29/L=0 bottom, Nan, "\\JCTime (\\U)\r2nd line"

// now a few "important location" tags...
Tag/N=text1/F=0/A=LB/X=1.20/Y=3.00 bottom, 0, "Big Bang"
Tag/N=text2/F=0/A=MB/X=0.00/Y=2.86 bottom, 8000000000, "Earth formed"
Tag/N=text3/F=0/A=RB/X=-0.80/Y=4.71 bottom, 13040000000, "Dinosaurs ruled"
```

See Also

See **About Text Info Variables** on page III-64 for *textStr* escape codes that manipulate text info variables and see the discussion in the **TextBox** operation about backslashes in *textStr*. The **AppendText** operation. For an explanation of graphing XY pairs, and X versus point scaling, see Chapter II-5, **Waves**.

For additional axis tag escape codes see the **Label** operation and **Axis Labels** on page II-280.

The **AnnotationList**, **AnnotationInfo**, **TagVal**, and **TagWaveRef** functions.

TagVal

TagVal(*code*)

TagVal is a very specialized function that is only valid when called from within the text of a tag as part of a \{} dynamic text escape sequence. It returns a number reflecting some property of the tag and helps you to display information about the tagged wave. The property is selected by the *code* parameter:

code	Return Value
0	Similar to \OP, returns the tag attach point number.
1	Similar to \OX, returns the X coordinate of tag attachment in the graph. When a tag is attached to an XY pair of traces, the X coordinate will most likely be different than the tag's X scaling attachment value specified in the Tag command.
2	Similar to \OY, returns the Y coordinate of tag attachment in the graph or the Y axis value in a Waterfall plot.
3	Similar to \OZ, returns the Z coordinate of tag attachment in a contour, image, or Waterfall plot.
4	Similar to \Ox, returns the trace x offset.
5	Similar to \Oy, returns the trace y offset.
6	Returns the X muloffset (with the not set value 0 translated to 1).
7	Returns the Y muloffset (with the not set value 0 translated to 1).

Because TagVal returns a numeric value, the result can be formatted any way you wish using the **printf** formatting codes. In contrast, the \O codes insert preformatted text, and you don't have control over the format.

TagVal is sometimes used in conjunction with the **TagWaveRef** function. For example, you might write a user-defined function that calculates a value as a function of a wave and a point number.

Examples

```
Tag wave0, 0, "Y value is \\{\"%g\",TagVal(2)}"
Tag wave0, 0, "Y value is \\{\"%g\",TagWaveRef()[TagVal(0)]}"
Tag wave0, 0, "Y value is \\OY"
```

These examples all produce identical results.

See Also

The **Tag** operation, the **TagWaveRef** function.

For a discussion of wave references, see **Wave Reference Functions** on page IV-173.

TagWaveRef

TagWaveRef()

TagWaveRef is a very specialized function that is only valid when called from within the text of a tag as part of a \{} dynamic text escape sequence. It returns a wave reference to the wave that the tag is on and helps you to display information about the tagged wave. It is often used in conjunction with the **TagVal** function. You can pass the result of TagWaveRef to any function that takes a Wave parameter.

Examples

Show the name of the data folder containing the tagged wave:

```
Tag wave0, 0,"\\ON is in \\{\"%s\",GetWavesDataFolder(TagWaveRef(),0)}"
```

See Also

The **Tag** operation, the **TagVal** function

For a discussion of wave references, see **Wave Reference Functions** on page IV-173.

tan

tan(*angle*)

The tan function returns the tangent of *angle* which is in radians.

In complex expressions, *angle* is complex, and tan(*angle*) returns a complex value:

$$\tan(x+iy)=\frac{\sin(\mathrm{x+iy})}{\cos(x+iy)}=\frac{\sin(2x)+i\sinh(2y)}{\cos(2x)+\cosh(2y)}.$$

See Also

sin, **cos**, **sec**, **csc**, **cot**

tanh

tanh(*num*)

The tanh function returns the hyperbolic tangent of *num*:

$$\tanh(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}.$$

In complex expressions, *num* is complex, and tanh(*num*) returns a complex value.

See Also

sinh, **cosh**, **coth**

TextBox

TextBox [***flags***] [***textStr***]

The TextBox operation puts a textbox on the target or named graph window. A textbox is an annotation that is not associated with any particular trace.

Parameters

textStr is the text that is to appear in the textbox. It is optional.

Flags

/A=*anchorCode* Specifies position of textbox anchor point.

anchorCode	**Position**	*anchorCode*	**Position**
LT	left top	RT	right top
LC	left center	RC	right center
LB	left bottom	RB	right bottom

anchorCode	Position	*anchorCode*	Position
MT	middle top		
MC	middle center		
MB	middle bottom		

anchorCode is a literal, *not* a string.

For interior textboxes, the anchor point is on the rectangular edge of the plot area of the graph window (where the left, bottom, right, and top axes are drawn).

For exterior textboxes, the anchor point is on the rectangular edge of the entire graph window.

/B=(*r*,*g*,*b*) Sets the color of the tag's background. *r*, *g*, and *b* specify the amount of red, green, and blue as an integer from 0 to 65535.

/B=*b*

b=0: Opaque background.

b=1: Transparent background.

b=2: Same background as the graph's plot area background.

b=3: Same background as the window background.

/C Changes existing textbox.

/D={*thickMult* [, *shadowThick* [, *haloThick*]]}

thickMult multiplies the normal frame thickness of a text-box. The thickness may be set using just /D=*thickMult*.

shadowThick, if present, overrides Igor's normal shadow thickness. It is in units of fractional points.

haloThick governs the annotation's halo thickness (a surrounding band of the annotation's background color), which can be -1 to 10 points wide.

The default *haloThick* value is -1, which preserves the behavior of previous versions of Igor where the halo of all annotations was set by the global variable root:V_TBBufZone. Any negative value of *haloThick* (-0.5, for example) will be overridden by V_TBBufZone if it exists, otherwise the absolute value of *haloThick* will be used. A zero or positive value overrides V_TBBufZone.

Any of the parameters may be missing. To set *haloThick* to 0 without changing other parameters, use `/D={,,0}`.

/E[=*exterior*] /E or /E=1 forces textbox (or legend) to be exterior to graph (provided *anchorCode* is not MC) and pushes the graph margins away from the anchor edge(s). /E=2 also forces exterior mode but does not push the margins.

/E=0 returns it to the default (an "interior textbox" which can be anywhere in the graph window).

/F=*frame*

frame=0: No frame.

frame=1: Underline frame.

frame=2: Box frame (default).

/G=(*r*,*g*,*b*) Sets color of the text in the tag. *r*, *g*, and *b* specify the amount of red, green, and blue as an integer from 0 to 65535.

/H=*legendSymbolWidth*

legendSymbolWidth sets width of the legend symbol (the sample line or marker) in points. Use 0 for the default, automatic width.

/K Kills existing textbox.

/LS= *linespace* Specifies a tweak to the normal line spacing where *linespace* is points of extra (plus or minus) line spacing. For negative values, a blank line may be necessary to avoid clipping the bottom of the last line.

/M[=*saMeSize*] /M or /M=1 specifies that legend markers should be the same size as the marker in the graph.

/M=0 turns same-size mode off so that the size of the marker in the legend is based on text size.

/N=*name* Specifies the name of the textbox to change or create.

/O=*rot* Sets the text's rotation. *rot* is in (integer) degrees, counterclockwise and must be a number from -360 to 360.

0 is normal horizontal left-to-right text, 90 is vertical bottom-to-top text.

/R=*newName* Renames the textbox.

/S=*style*
- *style*=0: Single frame.
- *style*=1: Double frame.
- *style*=2: Triple frame.
- *style*=3: Shadow frame.

/T=*tabSpec* *tabSpec* is a single number in points, such as `/T=72`, for evenly spaced tabs or a list of tab stops in points such as `/T={50, 150, 225}`.

/V=*vis*
- *vis* =0: Invisible annotation; not selectable. The annotation is still listed in **AnnotationList**.
- *vis* =1: Visible annotation.

/W=*winName* Operates in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

/X=*xOffset* For interior textboxes *xOffset* is the distance from anchor to textbox as a percentage of the plot area width.

For exterior textboxes *xOffset* is the distance from anchor to textbox as a percentage of the graph window width. See /E and /A.

/Y=*yOffset* *yOffset* is the distance from anchor to textbox as a percentage of the plot area height (interior textboxes) or graph window height (exterior textboxes). See /E and /A.

/Z=*freeZe*
- *freeZe*=1: Freezes annotation position (you can't move it with the mouse) in graphs but not in page layouts.
- *freeZe*=0: Unfreezes it. Annotations are always unfrozen in page layouts.

Details

Use the optional /W=*winName* flag to specify a specific graph or layout window. When used on the command line or in a Macro, Proc, or Window procedure, /W must precede all other flags.

If the /C flag is used, it must be the first flag in the command (except that if may follow an initial /W) and must be followed immediately by the /N=*name* flag.

If the /K flag is used, it must be the first flag in the command (or follow an initial /W) and must be followed immediately by the /N=*name* flag with no further flags or parameters.

textStr is optional. If missing, the textbox text is unchanged. This allows changes to the textbox to be made through the flags without changing the text.

A textbox can have at most 100 lines.

textStr can contain the following escape codes which affect subsequent characters in string. See **Notes about Backslashes**.

The characters "<??>" in a textbox indicate that you specified an invalid escape code or used a font that is not available.

The escape codes are:

\[<digit> Store info variable.

\]<digit> Recall info variable.

\B Use subscript (in smaller type above baseline).

\F'*fontName*' Use specified font (e.g., `\F'Helvetica'`).

\f*dd* *dd* is a binary coded number with each bit controlling one aspect of the column's font style as follows:
- bit 0: Bold
- bit 1: Italic

bit 2: Underline
bit 3: Outline (*Macintosh only*)
bit 4: Shadow (*Macintosh only*)

For example, bold underline is $2^0 + 2^2 = 1 + 4 = 5$. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

\JR	Right align text.
\JC	Center align text.
\JL	Left align text.
\K(*r*,*g*,*b*)	Use specified color for text. *r*, *g*, and *b* are integers from 0 to 65535.
\k(*r*,*g*,*b*)	Use specified color for marker stroke (line color). *r*, *g*, and *b* are integers from 0 to 65535. Use before \W*tdd* to change marker stroke color from the default of black (0,0,0).
\L*dtss*	Draws a line from x position specified in text info variable *d* to the current x position. Uses current text color. Thickness is encoded by digit *t* with values of 4,5,6 and 7 giving 0.25, 0.5, 1.0 and 1.5 pt. Line style is specified by 2 digit number *ss*.
\M	Use normal (main) script (reverts to main line and size).
\S	Use superscript (in smaller type below baseline).
\W*tdd*	Draws a marker symbol using current font size and color. The marker outline thickness is specified by the one-digit number *t* with 1, 4, 5 and 6 giving 0.0, 0.25, 0.5 and 1.0 point. A *t* value of 1, which sets the outline thickness to zero, is useful only for filled markers as it makes unfilled markers disappear. The marker symbol number is specified by the two-digit number *dd*.
\X<digit>	Recall X position.
\x+*dd*	Moves X position right by 2**dd* percent of the current font max width.
\x-*dd*	Moves X position left by 2**dd* percent of the current font max width.
\Y<digit>	Recall Y position.
\y+*dd*	Moves Y position up by 2**dd* percent of the current font height.
\y-*dd*	Moves Y position down by 2**dd* percent of the current font height.
\Z*nn*	Use font size *nn* (*nn* must be exactly two digits).
\Zr*nnn*	*nnn* is a 3 digit percentage by which to change the current font size. (*nnn* must be exactly three digits).

textStr can also contain escape codes to manipulate text info variables (see **About Text Info Variables** on page III-64).

textStr can also contain the following escape code which inserts the symbol used to display the named wave:

\s(*traceName***)**

\s is mainly used in Legends, where legend symbols are automatically created and removed, but it can also be used in Tags, Textboxes, and axis labels where the symbol is updated, but not automatically added or removed. See **Freezing the Legend Text** on page III-53.

textStr can also contain an escape code which inserts dynamically evaluated text:

\{*dynText*}

Dynamically evaluated text (*dynText*) may contain numeric and string expressions. Igor automatically reevaluates *dynText* when a numeric or string variable or wave referenced in *dynText* changes.

Note: When used in *dynText* expressions, user-defined numeric and string variables in macros or functions must be declared as global variables for the expression to evaluate correctly.

dynText can take two forms, an easy one for a single numeric expression and a more complex form that provides precise control over the formatting of the result.

The easy form is:

\{*numeric-expression*}

This evaluates the expression and prints with generic ("%g") formatting.

The full form is:

\{*formatStr, list-of-numeric-or-string-expressions*}

formatStr and *list-of-numeric-or-string-expressions* are treated as for **printf**.

As an aid in typing the expressions, Igor considers carriage returns between the braces to be equivalent to spaces. Rather than typing (in the Add Annotation dialog):

```
\{"twice K0 is %g, and today is %s",2*K0,date()}
```

you can type:

```
\{
   "twice K0 is %g, and today is %s",
   2*K0,
   date()
}
```

These carriage returns can be typed directly in the Add Annotations dialog, or be typed as "\r" in a macro, function or the command line.

Notes about Backslashes

Each backslash character in *textStr* should be preceded with another backslash when it appears in a macro, function or on the command line. This is because backslash is itself a special escape character for strings. Future versions of Igor may require this double-backslash syntax; currently Igor accepts either the single or double backslash as synonyms for inserting a single backslash into a string.

For example:

```
String myStr = "\\OK"
Print myStr
```

Prints the following in the history area:

```
  \OK
```

myStr contains a *single* backslash character. See **Escape Characters in Strings** on page IV-13 for more about the backslash character. This behavior affects how escape sequences in *textStr* should be written:

```
TextBox/C/N=text0 "\Z14Bigger"   // no: Future Igors may see "Z14Bigger"
TextBox/C/N=text0 "\\Z14Bigger"         // yes
TextBox/C/N=text0 "first line\rsecond line     // carriage return uses one \
```

Since \r (carriage return) is not a textbox escape sequence, only one backslash is needed.

You can see how backslashes in *textStr* should be entered by using the Add Annotation dialog and observing the command it creates. You will observe many double backslash sequences. Single backslash sequences were used by Igor 1.2, which Igor Pro currently accepts for backward compatibility reasons.

Examples

```
TextBox/C/N=t1/X=25/Y=50
```

moves the textbox named t1 to the location defined by X=25 and Y=50.

```
TextBox/C/N=t1 "New Text"
```

changes the text for t1.

See Also

See **About Text Info Variables** on page III-64 for *textStr* escape codes that manipulate text info variables. The **AppendText** operation. See the discussion above in **Notes about Backslashes** and **Escape Characters in Strings** on page IV-13 for still more about the backslash character. See the **printf** operation for formatting codes used in *formatStr*.

TextFile

TextFile(*pathName*, *index* [, *creatorStr*])

NOTE: TextFile is antiquated. Use **IndexedFile** instead.

The TextFile function returns a string containing the name of the *index*th TEXT file from the folder specified by *pathName*.

On Macintosh, TextFile returns only files whose file type property is TEXT, regardless of the file's extension.

On Windows, Igor considers files with ".txt" extensions to be of type TEXT.

Details

TextFile returns an empty string ("") if there is no such file.

pathName is the name of an Igor symbolic path; it is *not* a string.

index starts from zero.

creatorStr is an optional string argument containing four characters such as "IGR0". Only files of the specified Macintosh creator code are indexed. Set *creatorStr* to "????" to index all text files (or omit the argument altogether). This argument is ignored on Windows systems.

The order of files in a folder is determined by the operating system.

Examples

You can use TextFile in a procedure to sequence through each TEXT file in a folder, put the name of the text file into a string variable, and use this string variable as a parameter to the **LoadWave** or **Open** operations:

```
Function/S PrintFirstLineOfTextFiles(pathName)
   String pathName                  // Name of an Igor symbolic path.

   Variable refNum, index
   String str, fileName
   index = 0
   do
      fileName = TextFile($pathName, index)
      if (strlen(fileName) == 0)
         break                      // No more files
      endif
      Open/R/P=$pathName refNum as fileName
      FReadLine refNum, str         // Read first line including CR/LF
      Print fileName +":" + str     // Print file name and first line
      Close refNum
      index += 1                    // Next file
   while (1)
End
```

See Also

See the **IndexedFile** function, which is similar to TextFile but works on files of any type, and also **IndexedDir**. Also see the **LoadWave** and **Open** the operations.

ThreadGroupCreate

ThreadGroupCreate(*nt*)

The ThreadGroupCreate function creates a thread group containing *nt* threads and returns a thread ID number. Use the number of computer processors for *nt* when trying to improve computation speed using parallel threads. A background worker might use just one thread regardless of the number of processors.

See Also

ThreadSafe Functions on page IV-83 and **ThreadSafe Functions and Multitasking** on page IV-288.

ThreadGroupGetDF

ThreadGroupGetDF(*tgID, waitms*)

The ThreadGroupGetDF function retrieves a data folder path string from a thread group queue and removes the data folder from the queue.

When called from a preemptive thread it returns a data folder from the thread group's input queue. When called from the main thread it returns a data folder from the thread group's output queue.

tgID is a thread group ID returned by **ThreadGroupCreate**. You can pass 0 for *tgID* when calling ThreadGroupGetDF from a preemptive thread. You must pass a valid thread group ID when calling ThreadGroupGetDF from the main thread.

waitms is the maximum number of milliseconds to wait for a data folder to become available in the queue. Pass 0 to test if a data folder is available immediately. Pass INF to wait indefinitely or until a user abort.

ThreadGroupGetDF returns "" if the timeout period specified by *waitms* expires and no data folder is available in the queue.

For use with Igor Pro 6.20 or later, **ThreadGroupGetDFR** should be used instead of ThreadGroupGetDF which causes memory leaks.

See Also

ThreadSafe Functions on page IV-83 and **ThreadSafe Functions and Multitasking** on page IV-288.

The **ThreadGroupGetDFR** function.

ThreadGroupGetDFR

ThreadGroupGetDFR(*tgID, waitms*)

The ThreadGroupGetDF function retrieves a data folder reference from a thread group queue and removes the data folder from the queue. The data folder becomes a free data folder.

When called from a preemptive thread it returns a data folder from the thread group's input queue. When called from the main thread it returns a data folder from the thread group's output queue.

tgID is a thread group ID returned by **ThreadGroupCreate**. You can pass 0 for *tgID* when calling ThreadGroupGetDFR from a preemptive thread. You must pass a valid thread group ID when calling ThreadGroupGetDFR from the main thread.

waitms is the maximum number of milliseconds to wait for a data folder to become available in the queue. Pass 0 to test if a data folder is available immediately. Pass INF to wait indefinitely or until a user abort.

ThreadGroupGetDFR returns a NULL data folder reference if the timeout period specified by *waitms* expires and no data folder is available in the queue. You can test for NULL using **DataFolderRefStatus**.

ThreadGroupGetDFR was added in Igor Pro 6.20.

See Also

ThreadSafe Functions on page IV-83, **ThreadSafe Functions and Multitasking** on page IV-288 and **Free Data Folders** on page IV-75.

ThreadGroupPutDF

ThreadGroupPutDF *tgID, datafolder*

The ThreadGroupPutDF operation posts data to a preemptive thread group.

Parameters

tgID is thread group ID returned by **ThreadGroupCreate**, datafolder is the data folder you wish to send to the thread group.

datafolder can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

Details

When called from the main thread, **ThreadGroupPutDF** removes *datafolder* from main and posts to the input queue of the thread group specified by *tgID*. When called from a preemptive thread, use 0 for *tgID* and the data folder will be posted to the output queue of thread group to which thread belongs. Input and output data folders may be retrieved from the queues by calling the string function **ThreadGroupGetDF**.

Warning: Take care not to use any stale WAVE, NVAR, or SVAR variables that might contain references to objects in the data folder. Use **WAVEClear** on all WAVE reference variables that might contain references to waves that are in the data folder being posted before calling **ThreadGroupPutDF**. An error will occur if any waves in the data folder are in use or referenced in a WAVE variable.

Warning: Any DFREF variables that refer to the data folder (or any child thereof) must be cleared prior to executing this command. You can clear a DFREF using `dfr=$""`.

From the standpoint of the source thread, ThreadGroupPutDF is conceptually similar to KillDataFolder and, like KillDataFolder, if the current data folder is within *datafolder*, the current data folder is set to the parent of datafolder. You can not pass `root:` as *datafolder*.

See Also

The **ThreadGroupCreate** function, **ThreadSafe Functions** on page IV-83, and **ThreadSafe Functions and Multitasking** on page IV-288.

ThreadGroupRelease

ThreadGroupRelease(*tgID*)

The ThreadGroupRelease function releases thread group (and *tgID* is no longer valid). *tgID* is the thread group ID returned by **ThreadGroupCreate**.

If threads are still running, they are killed. An attempt is made to safely stop running threads but, if they continue to run, they will be force quit.

ThreadGroupRelease returns zero if successful, -1 if an error occurred (probably invalid *tgID*), or -2 if a force quit was needed. In the latter case, you should restart Igor Pro.

Any data folders remaining in the group's input or output queues will be discarded.

See Also

The **ThreadGroupCreate** function, **ThreadSafe Functions** on page IV-83, and **ThreadSafe Functions and Multitasking** on page IV-288.

ThreadGroupWait

ThreadGroupWait(*tgID, waitms*)

The ThreadGroupWait function returns index+1 of the first thread found still running after *waitms* milliseconds or returns zero if all are done.

tgID is the thread group ID returned by **ThreadGroupCreate** and *waitms* is milliseconds to wait.

If any of the threads of the group encountered a runtime error, the first such error will be reported now.

Use zero for *waitms* to just test or provide a large value to cause the main thread to sleep until the threads are finished. You can use INF to wait forever or until a user abort. If you know the maximum time the threads should take, you can use that value so you can print an error message or take other action if the threads don't return in time.

See Also

The **ThreadGroupCreate** function, **ThreadSafe Functions** on page IV-83, and **ThreadSafe Functions and Multitasking** on page IV-288.

ThreadProcessorCount

ThreadProcessorCount

The ThreadProcessorCount function returns the number of processors in your computer. For example, on a Macintosh Core Duo, it would return 2.

ThreadReturnValue

ThreadReturnValue(*tgID, index*)

The ThreadReturnValue function returns the value that the specified thread function returned when it exited. Returns NAN if thread is still running. *tgID* is the thread group ID returned by **ThreadGroupCreate** and *index* is the thread number.

See Also

The **ThreadGroupCreate** function, **ThreadSafe Functions** on page IV-83, and **ThreadSafe Functions and Multitasking** on page IV-288.

ThreadSafe

ThreadSafe Function *funcName*()

The ThreadSafe keyword declaration specifies that a user function can be used for preemptive multitasking background tasks on multiprocessor computer systems.

A ThreadSafe function is one that can operate correctly during simultaneous execution by multiple threads. Such functions are generally limited to numeric or utility functions. Functions that access windows are not ThreadSafe. To determine if an operation is ThreadSafe, use the Command Help tab of the Help Browser and choose ThreadSafe from the pop-up menu.

ThreadSafe functions can call other ThreadSafe functions but may not call non-ThreadSafe functions. Non-ThreadSafe functions can call ThreadSafe functions.

See Also

ThreadSafe Functions on page IV-83 and **ThreadSafe Functions and Multitasking** on page IV-288.

ThreadStart

ThreadStart *tgID, index, WorkerFunc*(*param1, param2,*…)

The ThreadStart operation starts the specified function running in a preemptive thread.

Parameters

tgID is thread group ID returned by **ThreadGroupCreate**, *index* is the desired thread of the group to set up to execute the specified ThreadSafe *WorkerFunc*.

Details

The worker function starts running immediately.

The worker function must be defined as ThreadSafe and must return a real or complex numeric result. The return value can be obtained after the function finishes by calling ThreadReturnValue.

The worker function can take variable and wave parameters. It can not take pass-by-reference parameters or data folder reference parameters.

Any waves you pass to the worker are accessible to both the main thread and to your preemptive thread. Such waves are marked as being in use by a thread and Igor will refuse to perform any manipulations that could change the size of the wave.

See Also

The **ThreadGroupCreate** and **ThreadReturnValue** functions; **ThreadSafe Functions** on page IV-83, and **ThreadSafe Functions and Multitasking** on page IV-288.

ticks

`ticks`

The ticks function returns the number of ticks (approximately 1/60 second) elapsed since the operating system was initialized.

See Also

The **stopMSTimer** function.

Tile

`Tile [`*`flags`*`] [`*`objectName`*` [, `*`objectName`*`]…]`

The Tile operation tiles the specified objects in the top page layout.

Parameters

objectName is the name of a graph, table, picture or annotation object in the top page layout.

Flags

/A=(*rows*,*cols*)	Specifies number of rows/columns in which to tile objects.
/G=*grout*	Specifies grout, the spacing between window tiles, in prevailing coordinates (points unless preceded by /I, /M or /R).
/I	Specifies coordinates in inches.
/M	Specifies coordinates in centimeters.
/O=*objTypes*	Adds objects of type(s) specified by bitwise mask to list of objects to be tiled: 1 (bit 0): Tile graphs. 2 (bit 1): Tile tables. 8 (bit 3): Tile pictures. 32 (bit 5): Tile textboxes. See **Setting Bit Parameters** on page IV-12 for details about bit settings.
/R	Specifies coordinates measured in percent of the printable page.
/S	Adds selected objects to objects to be tiled.
/W=(*left*,*top*,*right*,*bottom*)	Specifies page layout area in which to tile objects. Coordinates are in points unless /I, /M or /R are specified before /W.

Details

If /A=(*rows*,*cols*) is not used, Tile uses an appropriate number of rows and columns. If /A=(*rows*,*cols*) is used, objects are tiled in a grid of that many rows and columns. If *rows* or *cols* is zero, it substitutes an appropriate number for the zero parameter.

Objects to be tiled are determined by the /S and /O=*objTypes* flags and by any *objectName*s.

If no /S or /O flags are present and there are no *objectName*s, then all objects in the layout are tiled.

Otherwise the objects to be tiled are determined as follows:

- All *objectName*s are tiled.
- If the /S flag is present, the selected objects (if any) are also tiled.
- If the /O=*objTypes* flag is present then any objects specified by *objTypes* are also tiled. *objTypes* is a bitwise mask, so /O=3 tiles both graphs and tables.

See Also

The **Stack** operation.

TileWindows

TileWindows [***flags***] [***windowName*** [**, *windowName***]…]

The TileWindows operation tiles the specified windows on the desktop (*Macintosh*) or in the Igor frame window (*Windows*).

Flags

/A=(*rows*,*cols*) Specifies number of rows/columns in which to tile windows.

/C Adds the command window to the windows to be tiled.

/G=*grout* Specifies grout, the spacing between tiles, in prevailing units (points unless /I or /M are used).

/I Specifies coordinates in inches.

/M Specifies coordinates in centimeters.

/O=*objTypes* Adds windows of types specified by *objTypes* to windows to be tiled.

objTypes is a bitwise mask where:

bit 0: Graphs.
bit 1: Tables.
bit 2: Page layouts.
bit 4: Notebooks.
bit 6: Control panels.
bit 7: Procedure windows.
bit 9: Help windows.
bit 12: XOP target windows (e.g., Gizmo 3D plots).

Other bits should always be zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/P Adds the main procedure window to the windows to be tiled.

/R Specifies coordinates measured as % of tiling rectangle.

/W=(*left*,*top*,*right*,*bottom*)
Specifies tiling rectangle on the screen. Coordinates are in points unless /I, /M, or /R are specified before /W.

Details

The windows to be tiled are determined by the /C, /P, and /O=*objTypes* flags and by the *windowName*s. If no /C, /P or /O flags are present and there is no *windowName*s then all windows are tiled.

Otherwise the windows to be tiled are determined as follows:

- All named windows are tiled.
- If the /C flag is present, the command window is also tiled.
- If the /P flag is present, the procedure window is also tiled.
- If the /O=*objTypes* flag is present, any windows specified by *objTypes* are also tiled.

Examples

To tile all the procedure windows, including the main one, use:

```
TileWindows/P/O=128        // 2^7=128
```

See Also
The **StackWindows** operation.

time

time()
The time function returns a string containing the current time. The empty parentheses are required.

See Also
The **date**, **date2secs** and **DateTime** functions.

TitleBox

TitleBox [**/Z**] ***ctrlName*** [***keyword = value*** [**,** ***keyword = value*** **…**]]
The TitleBox operation creates the named title box in the target window.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters
ctrlName is the name of the TitleBox control to be created or changed.

The following keyword=value parameters are supported:

anchor= *hv*
: Specifies the anchor mode using a two letter code, *hv*. *h* may be L, M, or R for left, middle, and right. *v* may be T, C, or B for top, center and bottom. Default is LT.
If fixedSize=1, the anchor code sets the positioning of text within the frame.

appearance={*kind* [, *platform*]}
: Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.
kind can be one of `default`, `native`, or `os9`.
platform can be one of `Mac`, `Win`, or `All`.
See **Button** and **DefaultGUIControls** for more appearance details.

disable=*d*
: Sets user editability of the control.
d=0: Normal.
d=1: Hide.
d=2: Draw in gray state.

fColor=(*r*,*g*,*b*)
: Sets color of the titlebox. *r*, *g*, and *b* are integers from 0 to 65535.

fixedSize=*f*
: *f* =0: The titlebox automatically sizes itself to fit the title text (default).
f =1: The size settings are honored, and the titlebox does not automatically size itself to fit the title text.

font="*fontName*"
: Sets the font used for the control, e.g., `font="Helvetica"`.

frame= *f*
: Sets frame style.
f =0: No frame.
f =1: Default (same as *f*=3).
f =2: Simple box.
f =3: 3D sunken frame.
f =4: 3D raised frame.
f =5: Text well.

fsize=*s*
: Sets font size.

fstyle=*fs*
: *fs* is a binary coded number with each bit controlling one aspect of the font style as follows:
bit 0: Bold.
bit 1: Italic.
bit 2: Underline.
bit 3: Outline (*Macintosh only*).
bit 4: Shadow (*Macintosh only*).

	See **Setting Bit Parameters** on page IV-12 for details about bit settings.
labelBack=(*r*,*g*,*b*) or 0	Sets background color for title box. *r*, *g*, and *b* are integers from 0 to 65535. If not set (or labelBack=0), then background is transparent (not erased).
pos={*left*,*top*}	Sets the location of the top left corner in pixels.
pos+={*dx*,*dy*}	Offsets the position of the control in pixels.
size={*w*,*h*}	Set the width and height in pixels.
title=*titleStr*	Sets text of title box to *titleStr*. Limited to 63 characters.
variable= *svar*	Specifies an optional global string variable from which to get the TitleBox text.
win=*winName*	Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

The text can come from either the title=*titleStr* or variable=*svar* method. Whichever is used last is the current method. The maximum length of text with the title=*titleStr* method is 100 characters while the variable=*svar* method has no limit.

The text can contain escape codes to create fancy results with multiple lines, font changes etc. The escape codes are essentially the same as those for the TextBox operation. The easiest way to generate fancy text is to create a dummy TextBox, set up the text as desired and then click the To Cmd Line button followed by editing of the command line.

By default, the titlebox automatically resizes itself relative to the anchor point on the rectangle that encloses the text. Therefore you can specify a size of 0,0 along with a pos value in order to place the anchor point at the desired position. When fixedSize=1 is used, the titlebox does not resize itself and instead honors the values specified via the size keyword.

TitleBoxes can be used not only for titles but also as status or results readout areas, especially in conjunction with the variable= *svar* mode. When using a titlebox like this, you may find it useful to use fixedSize=1 so that the titlebox doesn't change size as the text changes.

Examples

```
NewPanel /W=(94,72,459,294)
DrawLine 150,32,150,140
DrawLine 70,100,213,100          // draw crossing lines at 150,100

// illustrate a default box
TitleBox tb1,title="A title box\rwith 2 lines",pos={150,100}

// Move center to 150,100
TitleBox tb1,pos={150,100},size={0,0},anchor=MC

// Set background color and therfore opaque mode
TitleBox tb1,labelBack=(55000,55000,65000)

// Now a few frame styles. Run these one at a time
TitleBox tb1,frame= 0            // no frame
TitleBox tb1,frame= 2            // plain frame
TitleBox tb1,frame= 3            // 3D sunken
TitleBox tb1,frame= 4            // 3D raised
TitleBox tb1,frame= 5            // text well

// Now some fancy text…
TitleBox tb1,frame= 1            // back to default (3D raised)
TitleBox tb1,title= "\Z18\[020 log\\B10\\M|[1 + 2K(jwt) + (jwt)\\S2\\M]|\\S-1"

// Create a string variable and hook up to the TitleBox
String s1= "text from a string variable"
TitleBox tb1,variable=s1

// Change string variable contents & note automatic update of TitleBox
s1= "something new"
```

See Also

The **TextBox** operation for additional text formatting escape codes and the **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

ToCommandLine

ToCommandLine ***commandsStr***

The ToCommandLine operation sends command text to the command line without executing the command(s).

The intended usage is for user-created panel windows with "To Cmd Line" buttons that are mimicking built-in Igor dialogs. You'll usually want to use Execute, instead.

Parameters

commandsStr The text of one or more commands.

Details

To send more than one line of commands, separate the commands with "\r" characters.

Note: ToCommandLine does not work when typed on the command line; use it only in a Macro, Proc, or Function.

Examples

```
Macro CmdPanel()
   PauseUpdate; Silent 1
   NewPanel /W=(150,50,430,229)
   Button toCmdLine,pos={39,148},size={103,20},title="To Cmd Line"
   Button toCmdLine,proc=ToCmdLineButtonProc
End

Function ToCmdLineButtonProc(ctrlName) : ButtonControl
   String ctrlName

   String cmd="MyFunction(xin,yin,\"yResult\")"// line 1: generate results
   cmd +="\rDisplay yOutput vs wx as \"results\"" // line 2: display results
   ToCommandLine cmd
End
```

See Also

The **Execute** and **DoIgorMenu** operations.

ToolsGrid

ToolsGrid [**/W=*winName***] ***keyword* = *value*** [**, *keyword* = *value*** …]

The ToolsGrid operation controls the grid you can use for laying out draw or control objects.

Parameters

ToolsGrid can accept multiple *keyword* = *value* parameters on one line.

snap=*n* Turns snap to grid on (*n*=1) or off (*n*=0).

visible=*n* Turns on grid visibility (*n*=1) or hides it (*n*=0).

grid=(*xy0*,*dxy*,*ndiv*) Defines both X and Y grids where *ndiv* is the number of subdivisions between major grid lines and *xy0* and *dxy* define the origin and spacing. Units are in points.

gridx=(*x0*,*dx*,*ndiv*) Defines the X grid where *ndiv* is the number of subdivisions between major grid lines and *x0* and *dx* define the origin and spacing. Units are in points.

gridy=(*y0*,*dy*,*ndiv*) Defines the Y grid where *ndiv* is the number of subdivisions between major grid lines and *y0* and *dy* define the origin and spacing. Units are in points.

Flags

/W=*winName* Sets the named window or subwindow for drawing. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

The default grid is 1 inch with 8 subdivisions. The grid is visible only in draw or selector mode and appears in front of the currently active draw layer.

TraceFromPixel

TraceFromPixel(*xpixel, ypixel, optionsString*)

The TraceFromPixel function returns a string based on an attempt to hit test the provided X and Y coordinates. Used to determine if the mouse was clicked on a trace in a graph.

When a trace is found, TraceFromPixel returns a string containing the following KEY:value; pairs:

TRACE:*tracename*

HITPOINT:*pnt*

tracename will be quoted if necessary and may contain instance notation. *pnt* is the point number index into the trace's wave when the hit was detected. If a trace is not found near the coordinate point, a zero length string is returned.

Parameters

xpixel and *ypixel* are the X and Y pixel coordinates.

optionsString can contain the following:

WINDOW:*winName*;

PREF:*traceName*;

ONLY:*traceName*;

DELTAX:*dx*;DELTAY:*dy*;

Use the WINDOW option to hit test in a graph other than the top graph. Use the ONLY option to search only for a special target trace. If the PREF option is used then the search will start with the specified trace but if no hit is detected, it will go on to the others.

When identifying a subwindow with WINDOW:*winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

The DELTAX and DELTAY values must both be specified to alter the region that Igor searches for traces. The *dx* and *dy* values are in pixels and the region searched is the rectangle from *xpixel-dx* to *xpixel+dx* and *ypixel-dy* to *ypixel+dy*.

If DELTAX or DELTAY are omitted, the search region depends on whether PREF or ONLY are specified. If either are specified then Igor first searches for the trace using *dx* = 3 and *dy* = 3. If the trace is not identified, Igor searches again using *dx* = 6 and *dy* = 6. If the trace is still not identified, Igor gives up and returns a zero-length result string.

If neither PREF nor ONLY are specified then Igor uses tries 3, 6, 12, and 24 for *dx* and *dy* until it finds a trace or gives up and returns a zero-length result string.

See Also

The **NumberByKey**, **StringByKey**, **AxisValFromPixel**, and **PixelFromAxisVal** functions.

ModifyGraph (traces) and **Instance Notation** on page IV-16 for discussions of trace names and instance notation.

TraceInfo

TraceInfo(*graphNameStr, ywavenameStr, instance*)

The TraceInfo function returns a string containing a semicolon-separated list of information about the trace in the named graph window or subwindow.

Parameters

graphNameStr can be "" to refer to the top graph.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

yWaveNameStr is either the name of a wave containing data displayed as a trace in the named graph, or a trace name (wave name with "#n" appended to distinguish the nth image of the wave in the graph). You might get a trace name from the **TraceNameList** function.

If *yWaveNameStr* contains a wave name, *instance* identifies which trace of *yWaveNameStr* you want information about. *instance* is usually 0 because there is normally only one instance of a given wave displayed in a graph. Set *instance* to 1 for information about the second trace of the wave named by *yWaveNameStr*, etc. If *yWaveNameStr* is `""`, then information is returned on the *instance*th trace in the graph.

If *yWaveNameStr* is a trace name, and *instance* is zero, the instance is extracted from *yWaveNameStr*. If *instance* is greater than zero, the wave name is extracted from *yWaveNameStr*, and information is returned concerning the *instance*th instance of the wave.

Details

The string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon. The keywords are as follows:

Keyword	Information Following Keyword
AXISFLAGS	Flags used to specify the axes. Usually blank because /L and /B (left and bottom axes) are the defaults.
AXISZ	Z value of a contour level trace or NaN if the trace is not a contour trace.
RECREATION	List of keyword commands as used by ModifyGraph command. The format of these keyword commands is: ***keyword*(x)=*modifyParameters*;**
XAXIS	X axis name.
XRANGE	Point subrange of the trace's X data wave in "[*startPoint*,þ*endPoint* : *increment*]" format. **Note**: Unlike the actual syntax of a trace subrange specification where increment is preceded by a semicolon character, here it is preceded by a colon character to preserve the notion that semicolon is what separates the keyword-value groups. If the entire X wave is displayed (the usual case), the XRANGE value is "[*]". If an X wave is not used to display the trace, then the XRANGE value is `""`.
XWAVE	X wave name if any, else blank.
XWAVEDF	Full path to the data folder containing the X wave or blank if no X wave.
YAXIS	Y axis name.
YRANGE	Point subrange of the trace's Y data wave or "[*]".

The format of the RECREATION information is designed so that you can extract a keyword command from the keyword and colon up to the ";", prepend "`ModifyGraph `", replace the "`x`" with the name of a trace ("`data#1`" for instance) and then **Execute** the resultant string as a command.

Note: The syntax of any subrange specifications in the RECREATION information are modified in the same way as for XRANGE and YRANGE. Currently only the zColor, zmrkSize, and zmrkNum keywords might have a subrange specification.

Examples

This example shows how to extract a string value from the keyword-value list returned by TraceInfo:

```
String yAxisName= StringByKey("YAXIS", TraceInfo("","",0))
```

This example shows how to extract a subrange and put the semicolon back:

```
String yRange= StringByKey("YRANGE", TraceInfo("","",0))
Print yRange                    // prints "[30,40:2]"
yRange= ReplaceString(":", yRange, ";")
Print yRange                    // prints "[30,40;2]"
```

The next example shows the trace information for the second instance of the wave "data" (which has an instance number of 1) displayed in the top graph:

```
Make/O data=x;Display/L/T data,data     // two instances of data: 0 and 1
Print TraceInfo("","data",1)[0,64]      // error if you try to print all
Print TraceInfo("","data",1)[65,128]
```

Prints the following in the history area:

```
XWAVE:;YAXIS:left;XAXIS:top;AXISFLAGS:/T;AXISZ:NAN(255);XWAVEDF:;
RECREATION:zColor(x)=0;zmrkSize(x)=0;zmrkNum(x)=0;textMarker(x)=
```

Following is a function that returns the marker code from the given instance of a named wave in the top graph. This example uses the convenient GetNumFromModifyStr() function provided by the #include <Readback ModifyStr> procedures, which are useful for parsing strings returned by TraceInfo.

```
#include <Readback ModifyStr>

Function MarkerOfWave(wv,instance)
    Wave wv
    Variable instance

    Variable marker
    String info = TraceInfo("",NameOfWave(wv),instance)

    marker = GetNumFromModifyStr(info,"marker","",0)

    return marker
End
```

See Also

The **Execute** operation.

TraceNameList

TraceNameList(*graphNameStr, separatorStr, optionsFlag*)

The TraceNameList function returns a string containing a list of trace names in the graph window or subwindow identified by *graphNameStr*.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

The parameter *separatorStr* should contain a single character such as "," or ";" to separate the names.

Details

The bits of *optionsFlag* have the following meanings:

Bit Number	Bit Value	Meaning
0	1	Include normal graph traces
1	2	Include contour traces
2	4	Omit hidden traces (the default is to list even hidden traces)

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

A trace name is defined as the name of the Y wave that defines the trace with an optional #ddd suffix that distinguishes between two or more traces that have the same wave name. Since the trace name has to be parsed, it is quoted if necessary.

Commands that take a trace name as a parameter or in a keyword can use a string containing a trace name with the $ operator to specify traceName. For instance, to change the display mode of a wave, you might use

```
ModifyGraph mode(myWave#1)=3
```

but

```
String myTraceName="myWave#1"
ModifyGraph mode($myTraceName)=3
```

will also work.

Examples

```
Make/O jack,'jack # 2';Display jack,jack,'jack # 2','jack # 2'
Print TraceNameList("",";",1)
Prints: jack;jack#1;'jack # 2';'jack # 2'#1;

// Generate a list of hidden traces
Make/O jack,jill,joy;Display jack,jill,joy
ModifyGraph hideTrace(joy)=1// hide joy
// (hidden + visible) - visible = hidden
String visibleTraces=TraceNameList("",";",1+4)// only visible normal traces
String allNormalTraces=TraceNameList("",";",1)// hidden + visible normal traces
```

```
String hiddenTraces= RemoveFromList(visibleTraces,allNormalTraces)
Print hiddenTraces
// Prints: joy;
```

See Also

For other commands related to waves and traces: **WaveRefIndexed**, **XWaveRefFromTrace**, **TraceNameToWaveRef**, **CsrWaveRef**, and **CsrXWaveRef**.

For a description of traces: **ModifyGraph**. For a discussion of contour traces: **All About Contour Traces** on page II-328.

For commands referencing other waves in a graph: **ImageNameList**, **ImageNameToWaveRef**, **ContourNameList**, and **ContourNameToWaveRef**.

ModifyGraph (traces) and **Instance Notation** on page IV-16 for discussions of trace names and instance notation.

TraceNameToWaveRef

TraceNameToWaveRef(*graphNameStr, traceNameStr*)

The TraceNameToWaveRef function returns a wave reference to the Y wave corresponding to the given trace in the graph window or subwindow named by *graphNameStr*.

Parameters

graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

The trace is identified by the string in *traceNameStr*, which could be a string determined using **TraceNameList**. Note that the same trace name can refer to different waves in different graphs.

Use **Instance Notation** (see page IV-16) to choose from traces in a graph that represent waves of the same name. For example, if *traceNameStr* is "myWave#2", it refers to the third instance of wave "myWave" in the graph ("myWave#0" or just "myWave" is the first instance).

See Also

For other commands related to waves and traces: **WaveRefIndexed**, **XWaveRefFromTrace**, **TraceNameList**, **CsrWaveRef**, and **CsrXWaveRef**.

For a description of traces: **ModifyGraph**. For a discussion of contour traces, see **All About Contour Traces** on page II-328.

For a discussion of wave references, see **Wave Reference Functions** on page IV-173.

For commands referencing other waves in a graph: **ImageNameList**, **ImageNameToWaveRef**, **ContourNameList**, and **ContourNameToWaveRef**.

Triangulate3D

Triangulate3D [/OUT=*format*] *srcWave*

The Triangulate3D operation creates a Delaunay "triangulation" of a 3D scatter wave. The output is a list of tetrahedra that completely span the convex volume defined by *srcWave*. Triangulate3D can also generate the triangulation needed for performing 3D interpolation for the same domain. Normally *srcWave* is a triplet wave (a 2D wave of 3 columns), but can use any 2D wave that has more than 3 columns (the operation ignores all but the first 3 columns).

Flags

/OUT=*format* Specifies how to save the output triangulation data.

format=1: Default; saves the triangulation result in the wave M_3DVertexList, which contains in each row, indices to rows in *srcWave* that describe the X, Y, Z coordinates of a single tetrahedral vertex. Each tetrahedron is described by one row in M_3DVertexList.

format=2: Saves the triangulation result in the wave M_TetraPath, which is a triplet path wave describing the tetrahedra edges. For each tetrahedron, there are four rows (triangles) separated by row of NaNs. The total number of rows in M_TetraPath is 20 times the number of tetrahedra in the triangulation.

format=4: Saves a wave containing internal diagnostic information generated during the triangulation process.

Details

Triangulate3D implements Watson's algorithm for tetrahedralization of a set of points in three dimensions. It starts by creating a very large tetrahedron which inscribes all the data points followed by a sequential insertion of one datum at a time. With each new datum the algorithm finds the tetrahedron in which the datum falls. It then proceeds to subdivide the tetrahedron so that the datum becomes a vertex of new tetrahedra.

The algorithm suffers from two known problems. First, it may, due to numerical instabilities, result in tetrahedra that are too thin. You can get around this problem by introducing a slight random perturbation in the input wave. For example:

```
srcWave+=enoise(amp)
```

Here amp is chosen so that it is much smaller than the smallest cartesian distance between two input points.

The second problem has to do with memory allocations which may exhaust available memory for some pathological spatial distributions of data points. The operation reports both problems in the history window.

Examples

```
Make/O/N=(10,3) ddd=gnoise(5)     // create random 10 points in space
Triangulate3d/out=2 ddd

// now display the triangulation in Gizmo:
Window Gizmo0() : GizmoPlot
    PauseUpdate; Silent 1
    if(exists("NewGizmo")!=4)
        DoAlert 0, "Gizmo XOP must be installed"
        return
    endif
    NewGizmo/N=Gizmo0 /W=(309,44,642,373)
    ModifyGizmo startRecMacro
    ModifyGizmo scalingMode=2
    AppendToGizmo Scatter=root:ddd,name=scatter0
    ModifyGizmo ModifyObject=scatter0 property={ scatterColorType,0}
    ModifyGizmo ModifyObject=scatter0 property={ Shape,2}
    ModifyGizmo ModifyObject=scatter0 property={ size,0.2}
    ModifyGizmo ModifyObject=scatter0 property={ color,0,0,0,1}
    AppendToGizmo Path=root:M_TetraPath,name=path0
    ModifyGizmo ModifyObject=path0 property={ pathColor,0,0,1,1}
    ModifyGizmo setDisplayList=0, object=scatter0
    ModifyGizmo setDisplayList=1, object=path0
    ModifyGizmo autoscaling=1
    ModifyGizmo compile
    ModifyGizmo endRecMacro
End
```

References

Watson, D.F., Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes, *The Computer J.*, *24*, 167-172, 1981.

Further information about this algorithm can be found in:

Watson, D.F., *CONTOURING: A guide to the analysis and display of spatial data*, Pergamon Press, 1992.

See Also

The **Interpolate3D** operation and the **Interp3D** function.

trunc

trunc(*num*)

The trunc function returns the integer closest to *num* in the direction of zero.

See Also

The **round**, **floor**, and **ceil** functions.

try-catch-endtry

```
try
   <possible abort code>
catch
   <code to handle abort>
endtry
```

A try-catch-endtry flow control statement provides a means for catching and responding to abort conditions in user functions.

When code executes in the try-catch area, any programmatic aborts will immediately jump to code within the catch-endtry area rather than jumping to the end of the user function. Normal flow (no aborts) will skip all code within the catch-endtry area.

Details

During execution of code in the catch-endtry area, user aborts are suppressed. This means that when the user attempts to abort procedure execution by holding down the appropriate abort key combination, it will not abort the catch code itself.

Whenever an abort occurs, information about the cause of the abort is returned via the following variable:

`V_AbortCode =`	-4:	Abort triggered by AbortOnRTE.
	-3:	Abort caused by Abort operation.
	-2:	Stack overflow abort.
	-1:	User abort.
	>=1:	Abort triggered by AbortOnValue.

See Also

Flow Control for Aborts on page IV-38 and **try-catch-endtry Flow Control** on page IV-38 for further details.

The **AbortOnRTE** and **AbortOnValue** keywords, and the **Abort** operation.

UniqueName

```
UniqueName(baseName, objectType, startSuffix [, windowNameStr])
```

The UniqueName function returns the concatenation of *baseName* and a number such that the result is not in conflict with any other object name.

windowNameStr is optional. If missing, it is taken to be the top graph, panel, layout, or notebook according to the value of *objectType*.

Details

baseName should be an unquoted name, such as you might receive from the user via a dialog or control panel.

objectType is one of the following:

1	Wave.	9	Control panel window.
2	Reserved.	10	Notebook window.
3	Numeric variable.	11	Data folder.
4	String variable.	12	Symbolic path.
5	XOP target window (e.g., surface plot).	13	Picture.
6	Graph window.	14	Annotation in the named or topmost graph or layout.
7	Table window.	15	Control in the named or topmost graph or panel.
8	Layout window.	16	Notebook action character in the named or topmost notebook.

startSuffix is the number used as a starting point when generating the numeric suffix that makes the name unique. Normally you should pass zero for startSuffix. If you know that names of the form base0 through baseN are in use, you can make UniqueName run a bit faster by passing `N+1` as the *startSuffix*.

The *windowNameStr* argument is used only with *objectTypes* 14, 15, and 16. The returned name is unique only to the window (other windows might have objects with the same name). If a named window is given but does not exist, UniqueName returns *baseName startSuffix*. *windowNameStr* is ignored for other *objectTypes*

Examples

```
String uniqueWaveName = UniqueName(baseWaveName, 1, 0)
String uniqueControlName = UniqueName("ctrl", 15, 0, "Panel0")
```

See Also

CheckName and **CleanupName**.

UnPadString

UnPadString(*str, padValue*)

UnPadString returns *str* after removing all trailing characters whose value is *padValue*. This undoes the action of PadString.

UnPadString can be used to recover the values of text waves created with Make/T=size. This is useful for speeding up access to very long text waves. Such waves are set by assigning the output of PadString to their elements.

UnPadString allows for the storage of variable length C strings (*padValue* of zero) in text waves created with preallocated string sizes.

See Also

The **PadString** function. The **Make** operation.

Unwrap

Unwrap *modulus, waveName* [*, waveName*]…

The Unwrap operation scans through each named wave trying to undo the effect of a modulus operation.

Parameters

modulus is the value applied to the named waves through the **mod** function as if the command:

```
waveName = mod(waveName,modulus)
```

had been executed. It is this calculation which Unwrap attempts to undo.

Details

The unwrap operation works with 1D waves only. See **ImageUnwrapPhase** for phase unwrapping in two dimensions.

Examples

If you perform an FFT on a wave, the result is a complex wave in rectangular coordinates. You can create a real wave that contains the phase of the result of the FFT with the command:

```
wave2 = imag(r2polar(wave1))
```

However, the rectangular to polar conversion leaves the phase information modulo 2π. You can restore the phase information with the command:

```
Unwrap 2*pi, wave2
```

Because the first point of a wave that has been FFTed has no phase information, in this example you would *precede* the Unwrap command with the command:

```
wave2[0] = wave2[1]
```

See Also

The **ImageUnwrapPhase** operation and **mod** function.

UpperStr

UpperStr(*str*)

The UpperStr function returns a string expression in which all lower-case characters in *str* are converted to upper-case.

See Also

The **LowerStr** function.

URLDecode

URLDecode(*inputStr*)

The URLDecode function returns a percent-decoded copy of the percent-encoded string *inputStr*. It is unlikely that you will need to use this function; it is provided for completeness.

For an explanation of percent-encoding, see **Percent Encoding** on page IV-239.

Example

```
String theURL = "http://google.com?key1=35%25%20larger"
theURL = URLDecode(theURL)
Print theURL
  http://google.com?key1=35% larger
```

See Also

URLEncode, **URLs** on page IV-239.

URLEncode

URLEncode(*inputStr*)

The URLEncode function returns a percent-encoded copy of *inputStr*.

Percent-encoding is useful when encoding the query part of a URL or when the URL contains special characters that might otherwise be misinterpreted by a web server. For an explanation of percent-encoding, see **Percent Encoding** on page IV-239.

Example

```
String baseURL = "http://google.com"
String key1 = "key1"
String value1 = URLEncode("35% larger")
String theURL = ""
sprintf theURL, "%s?%s=%s", baseURL, key1, value1
Print theURL
  http://google.com?key1=35%25%20larger
```

See Also

URLDecode, **URLs** on page IV-239.

ValDisplay

ValDisplay [**/Z**] ***ctrlName*** [***keyword = value*** [**,** ***keyword = value*** **…**]]

The ValDisplay operation creates or modifies the named control that displays a numeric value in the target window. The appearance of the control varies; see the **Examples** section.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the ValDisplay control to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}
: Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.
: *kind* can be one of `default`, `native`, or `os9`.
: *platform* can be one of `Mac`, `Win`, or `All`.
: See **Button** and **DefaultGUIControls** for more appearance details.

barBackColor=(*r*,*g*,*b*)
: Sets the background color under the bar (if any). *r*, *g*, and *b* range from 0 to 65535.

barBackColor=0
: Sets the background color under the bar to the default color, the standard document background color used on the current operating system, which is usually white.

barmisc={*lts*, *valwidth*}
: Sets the "limits text size" and the size of the type showing the bar limits. If *lts* is zero, the bar limits are not displayed. Otherwise, *lts* must be between 5 and 100.
: *valwidth* is the "value readout width". It claims the amount of horizontal space for the numeric part of the display.

If *valwidth* equals or exceeds the control width available to it, the numeric readout uses all the room, and prevents display of any bar.

If *valwidth* is zero, there is no numeric readout, and only the bar is displayed.

valwidth can range from zero to 4000, and it defaults to 1000 (which usually leaves no room for the display bar).

bodyWidth=*width*

Specifies an explicit size for the body (nontitle) portion of a ValDisplay control. By default (bodyWidth=0), the body portion is the amount left over from the specified control width after providing space for the current text of the title portion. If the font, font size, or text of the title changes, then the body portion may grow or shrink. If you supply a bodyWidth>0, then the body is fixed at the size you specify regardless of the body text. This makes it easier to keep a set of controls right aligned when experiments are transferred between Macintosh and Windows, or when the default font is changed.

disable=*d*

Sets user editability of the control.

d=0: Normal.

d=1: Hide.

d=2: Disable user input. Does not change control appearance because it is read-only.

fColor=(*r*,*g*,*b*)

Sets the initial color of the title. *r*, *g*, and *b* range from 0 to 65535. fColor defaults to black (0,0,0). To further change the color of the title text, use escape sequences as described for title=*titleStr*.

font="*fontName*"

Sets the font used to display the value of the variable, e.g., `font="Helvetica"`.

format=*formatStr*

Sets the numeric format of the displayed value. The default format is "%g". For a description of *formatStr*, see the **printf** operation.

frame=*f*

Sets frame style.

f=0: Value is unframed.

f=1: Default; value is framed (same as *f*=3).

f=2: Simple box.

f=3: 3D sunken frame.

f=4: 3D raised frame.

f=5: Text well.

fsize=*s*

Sets the size of the type used to display the value in the numeric readout. The default is 12 points.

fstyle=*fs*

fs is a binary coded number with each bit controlling one aspect of the font style as follows:

bit 0: Bold.

bit 1: Italic.

bit 2: Underline.

bit 3: Outline (*Macintosh only*).

bit 4: Shadow (*Macintosh only*).

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

help={*helpStr*}

Sets the help for the control. The help text is limited to a total of 255 characters. On Macintosh, help appears when you turn Igor Tips on. On Windows, help for the first 127 characters or up to the first line break appears in the status line. If you press F1 while the cursor is over the control, you will see the entire help text. You can insert a line break by putting "\r" in a quoted string.

highColor=(*r*,*g*,*b*)

Specifies the bar color when the value is greater than *base* in the limits keyword. *r*, *g*, and *b* are integers from 0 to 65535.

labelBack=(*r*,*g*,*b*) or 0

Specifies the background fill color for labels. *r*, *g*, and *b* are integers from 0 to 65535. The default is 0, which uses the window's background color.

limits={*low*,*high*,*base*}

Controls how the value is translated into a graphical representation when the display includes a bar (described fully in **Details**). Defaults are {0,0,0}, which aren't too useful.

limitsColor=(*r*,*g*,*b*) Sets the color of the limits text, if any. *r*, *g*, and *b* range from 0 to 65535. limitsColor defaults to black (0,0,0).

limitsBackColor=(*r*,*g*,*b*) Sets the background color under the limits text. *r*, *g*, and *b* range from 0 to 65535.

limitsBackColor=0 Sets the background color under the limits text to the default color, the standard document background color used on the current operating system, which is usually white.

lowColor=(*r*,*g*,*b*) Specifies the bar color when the value is less than *base* in the limits keyword. *r*, *g*, and *b* are integers from 0 to 65535.

mode=*m* Specifies the type of LED display to use, if any.

- *m*=0: Bar mode (default).
- *m*=1: Oval LED.
- *m*=2: Rectangular LED.
- *m*=3: Bar mode with no fractional part.
- *m*=4: Candy-stripe effect for the bar area to support indefinite-style progress windows. The value is taken to be the phase of the candy stripe. When using value= _NUM:n, n is taken as an increment value so you would normally just use 1. Uses the native platform appearance if the high and low colors are left as default. Note native formats may not fill vertical space. See **Progress Windows** on page IV-134 for an example.

pos={*left*,*top*} Sets the position of the display in pixels, from 0 to 32767.

pos+={*dx*,*dy*} Offsets the position of the display in pixels.

rename=*newName* Gives the ValDisplay control a new name.

size={*width*,*height*} Sets width and height of display in pixels. *width* can range from 10 to 200 pixels, *height* from 5 to 200 pixels. Default width is 50, default height is determined by the numeric readout font size.

title=*titleStr* Sets title of display to the specified string expression. The title appears to the left of the display. If this title is too long, it won't leave enough room to display the bar or even the numeric readout! Defaults to "" (no title).

titleStr can contain formatting escape codes in order to create fancy, styled results. The escape codes are the same as used by the **TextBox** operation. The easiest way to generate fancy text is to make selections from the Insert popup in the ValDisplay Control dialog.

value=*valExpr* Displays the numeric expression *valExpr*. It is *not* a string.

As of version 6.1, you can use the syntax _NUM:num to specify a numeric value without using a dependency.

valueColor=(*r*,*g*,*b*) Sets the color of the value readout text, if any. *r*, *g*, and *b* range from 0 to 65535. valueColor defaults to black (0,0,0).

valueBackColor=(*r*,*g*,*b*) Sets the background color under the value readout text.*r*, *g*, and *b* range from 0 to 65535.

valueBackColor=0 Sets the background color under the value readout text to the default color, the standard document background color used on the current operating system, which is usually white.

win=*winName* Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

zeroColor=(*r*,*g*,*b*) Governs the LED color (in LED mode only). *r*, *g*, and *b* are integers from 0 to 65535. Used in conjunction with the limits keyword such that zeroColor determines one endpoint color when base is between *low* and *high*, or LED color when the value is less than *low*.

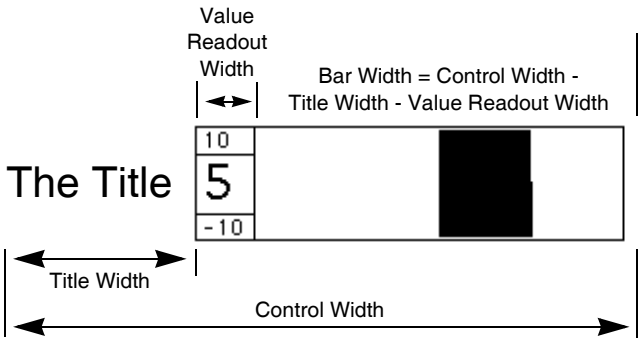
Flags

/Z No error reporting.

Details

The target window must be a graph or panel.

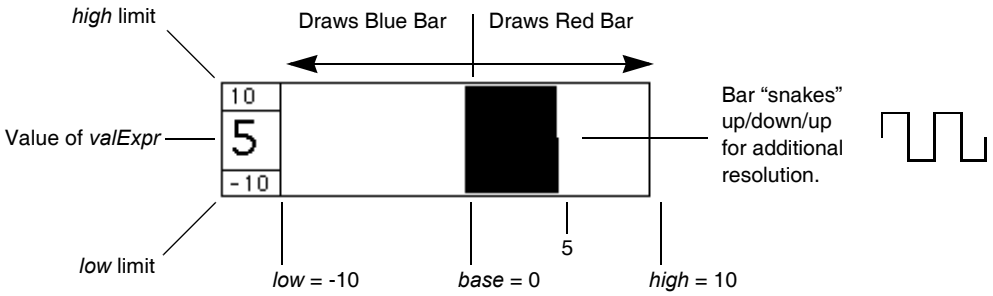
The appearance of the ValDisplay control depends primarily on the *width* and *valwidth* parameters and the width of the title. Space for the individual elements is allocated from left to right, with the title receiving first priority. If the control width hasn't all been used by the title, then the value display gets either *valwidth* pixels of room, or what is left. If the control width hasn't been used up, the bar is displayed in the remaining control width:



If you use the bodyWidth keyword, the value readout width and bar width occupy the body width. The total control width is then bodyWidth+title width, and the width from the size keyword is ignored.

The limits values *low*, *high*, and *base* and the value of *valExpr* control how the bar, if any, is drawn. The bar is drawn from a starting position corresponding to the *base* value to an ending position determined by the value of *valExpr*, *low* and *high*. *low* corresponds to the left side of the bar, and *high* corresponds to the right. The position that corresponds to the *base* value is linearly interpolated between *low* and *high*.

For example, with *low*= -10, *high*=10, and *base*= 0, a *valExpr* value of 5 will draw from the center of the bar area (0 is centered between -10 and 10) to the right, halfway from the center to the right of the bar area (5 is halfway from 0 to 10):



The *valExpr* must be executable at any time. The expression is stored and executed when the ValDisplay needs to be updated. However, execution will occur outside the routine that creates the ValDisplay, so you must not use local variables in the expression.

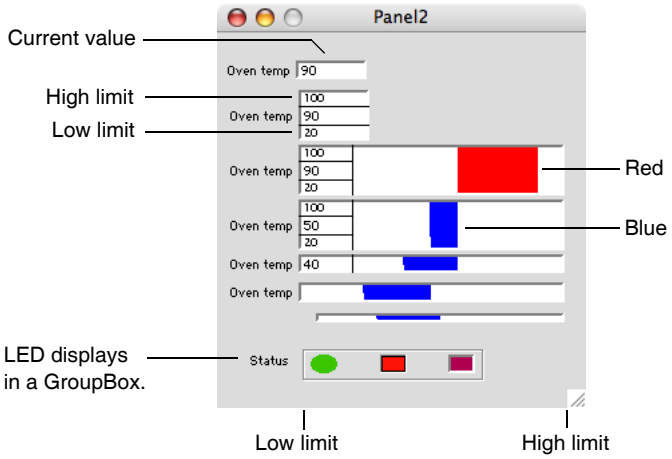
valExpr may be enclosed in quotes and preceded with a # character (see **When Dependencies are Updated** on page IV-206) to defer evaluation of the validity of the numeric expression, which may be needed if the expression references as-yet-nonexistent global variables or user-defined functions:

```
ValDisplay valdisp0 value=notAVar*2     // "unknown name or symbol" error
ValDisplay valdisp0 value=#"notAVar*2"     // still not valid, no error
Variable notAVar=3                      // now valid; ValDisplay works
```

In a ValDisplay, the #"" syntax permits use of a string expression. Normally, the # prefix signifies that the following text must be a literal quoted string. String expressions are evaluated at runtime to obtain the final expression for the ValDisplay. In other words, there is a level of indirection.

Examples

Here is a sampling of the various types of ValDisplay controls available:



You can use a ValDisplay to replace the bar mode with a solid color fill designed to look like an LED. Use the mode keyword with mode=1 to create an oval LED or mode=2 to create a rectangular LED. You can specify different frames with the rectangular LED but only a simple frame is available for the oval mode. Use mode=0 to revert to bar mode.

The color and brightness of the LED depends on the value that the ValDisplay is monitoring combined with the limits={*low*, *high*, *base*) setting, the two color settings used in bar mode along with a third color (zeroColor) that is used only in LED mode. When the value is between *low* and *high*, the color is a linear combination of endpoint colors. If *base* is between *low* and *high*, the endpoint colors are the low color and the zero color, or the zero color and the high color. For values outside the limits, the appropriate limiting color is chosen.

If *base* is less than the *low*, the endpoint colors are the low color and the high color. In this case, if the value is less than *low* the LED takes on the zero color.

You should use the bodyWidth setting in conjunction with LED mode to keep the LED from dramatically changing size or disappearing when the title is changed or if your experiment is moved to a different platform (Macintosh vs PC).

Try the ValDisplay Demo example experiment to see these different modes in action. The experiment file is in your Igor Pro folder, in the "Examples:Feature Demos" subfolder.

See Also

See **ValDisplay** on page III-380 for more examples, and the **printf** operation about *formatStr*.

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

Progress Windows on page IV-134 for an example of candy-stripe mode=4.

Variable

Variable [*flags*] *varName* [=*numExpr*][, *varName* [=*numExpr*]]…

The Variable declaration creates real or complex variables and gives them the specified name.

Flags

/C	Declares a complex variable.
/D	Obsolete, included only for backward compatibility (see **Details**).
/G	Creates a variable with global scope and overwrites any existing variable.

Details

The variable is initialized when it is created if you supply the initial value. However, when Variable is used to declare a function parameter, it is an error to attempt to initialize it.

You can create more than one variable at a time by separating the names and optional initializers for multiple variables with a comma.

As of Igor Pro 2.02 for Power Macintosh, Igor Pro 3.0 for all Macintoshes, and for all Windows versions, all numeric variables are double precision. Previously variables could be single or double precision and the /D flag meant double precision. The /D flag is allowed for backward compatibility but is no longer needed and should not be used in new code.

If used in a macro or function the new variable is local to that macro or function unless the /G (global) flag is used. If used on the command line, the new variable is global.

varName can include a data folder path.

Examples

To initialize a complex variable, use the **cmplx** function. For example:

```
Variable/C cv1 = cmplx(1,2)
```

sets the real part of cv1 to 1 and the imaginary part to 2.

Variance

Variance(*inWave* [, *x1*, *x2*])

Returns the variance of the real-valued *inWave*. The function ignores NaN and INF values in *inWave*.

Parameters

inWave is expected to be a real-valued numeric wave. If *inWave* is a complex or text wave, Variance returns NaN.

x1 and *x2* specify a range in *inWave* over which the variance is to be calculated. They are used only to locate the points nearest to x=*x1* and x=*x2* . The variance is then calculated over that range of points. The order of *x1* and *x2* is immaterial.

If omitted, *x1* and *x2* default to -∞ and +∞ respectively and the variance is calculated for the entire wave.

Details

The variance is defined by

$$\text{var} = \frac{\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2}{n-1}$$

where

$$\bar{x} = \frac{\sum_{i=1}^{n} X_i}{n}.$$

Examples

```
Make/O/N=5 test = p
SetScale/P x, 0, .1, test

// Print variance of entire wave
Print Variance(test)

// Print variance from x=0 to x=.2
Print Variance(test, 0, .2)

// Print variance for points 1 through 3
Variable x1=pnt2x(test, 1)
Variable x2=pnt2x(test, 3)
Print Variance(test, x1, x2)
```

See Also

mean, **WaveStats**

VariableList

VariableList(*matchStr, separatorStr, variableTypeCode*)

The VariableList function returns a string containing a list of global variables selected based on the *matchStr* and *variableTypeCode* parameters. The variables listed are all in the current data folder.

Details

For a variable name to appear in the output string, it must match *matchStr* and also must fit the requirements of *variableTypeCode*. The first character of *separatorStr* is appended to each variable name as the output string is generated.

The name of each variable is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches all variable names.
"xyz"	Matches variable name xyz only.
"*xyz"	Matches variable names which end with xyz.
"xyz*"	Matches variable names which begin with xyz.
"*xyz*"	Matches variable names which contain xyz.
"abc*xyz"	Matches variable names which begin with abc and end with xyz.

matchStr may begin with the ! character to return windows that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches variable names which *do not* end with xyz.

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

variableTypeCode is used to further qualify the variable. The variable name goes into the output string only if it passes the match test and its type is compatible with *variableTypeCode*. *variableTypeCode* is any one of:

2: System variables (K0, K1 . . .)
4: Scalar variables
5: Complex variables

Examples

`VariableList("*",";",4)`	Returns a list of all scalar variables.
`VariableList("!V_*", ";",5)`	Returns a list of all complex variables except those whose names begin with “V_”.

See Also

See the **StringList** and **WaveList** functions.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

vcsr

vcsr(*cursorName* [, *graphNameStr*])

The vcsr function returns the Y (vertical) value of the point which the specified cursor (A through J) is attached to in the top (or named) graph.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

The result is computed from the coordinate system of the graph’s Y axis. The Y axis used is the one used to display the wave on which the cursor is placed.

See Also

The **Cursor** operation and the **CsrInfo**, **hcsr**, **pcsr**, **qcsr**, **xcsr**, and **zcsr** functions.

version

#pragma version = *versNum*

In the File Information dialog, #pragma version=*versNum* provides file version information that is displayed next to the file name in the dialog. This line must not be indented and must appear in the first fifty lines of the file. See **Procedure File Version Information** on page IV-145.

See Also

The **The version Pragma** on page IV-42, **Procedure File Version Information** on page IV-145, the **IgorInfo** function, and #**pragma**.

WAVE

WAVE [**/C**][**/T**][**/Z**] ***localName*** [**=*pathToWave***][**, *localName1*** [**=*pathToWave1***]]…

WAVE is a declaration that identifies the nature of a user-defined function parameter or creates a local reference to a wave accessed in the body of a user-defined function.

The optional parameter *pathToWave* is used only in the body of a function, not in a parameter declaration.

The WAVE reference is required when you use a wave in an assignment statement in a function. At compile time, the WAVE statement specifies that the local name references a wave. At runtime, it makes the connection between the local name and the actual wave. For this connection to be made, the wave must exist when the WAVE statement is executed.

When *localName* is the same as the global wave name and you want to reference a wave in the current data folder, you can omit the *pathToWave*. Prior to Igor Pro 4.0, *pathToWave* was always required.

pathToWave can be a full literal path (e.g., root:FolderA:wave0), a partial literal path (e.g., :FolderA:wave0) or $ followed by string variable containing a computed path (see **Converting a String into a Reference Using $** on page IV-47).

You can also use a data folder reference or the /SDFR flag to specify the location of the wave if it is not in the current data folder. See **Data Folder References** on page IV-61 and **The /SDFR Flag** on page IV-63 for details.

If the wave may not exist at runtime, use the /Z flag and call **WaveExists** before accessing the wave. The /Z flag prevents Igor from flagging a missing wave as an error and dropping into the debugger. For example:

```
WAVE/Z wv=<pathToPossiblyMissingWave>
if( WaveExists(wv) )
   <do something with wv>
endif
```

Note that to create a wave, you use the **Make** operation.

Flags

/C	Complex wave.
/T	Text wave.
/Z	Ignores wave reference checking failures.

See Also

WaveExists function.

WAVE Reference Type Flags on page IV-58 for additional wave type flags and information.

Accessing Global Variables and Waves on page IV-50.

Accessing Waves in Functions on page IV-65.

Converting a String into a Reference Using $ on page IV-47.

WAVEClear

WAVEClear *localName* [**, *localName1*** **…**]

The WAVEClear operation clears out a WAVE reference variable. WAVEClear is equivalent to WAVE/Z localName= $"".

Details

Use WAVEClear to avoid unexpected results from certain operations such as **Duplicate** or **Concatenate**, which will reuse the contents of a WAVE reference variable and may not generate the wave in the desired data folder or with the desired name.

WAVEClear ensures that memory is deallocated after waves are killed as in this example:

```
Function foo()
   Make wave1
   FunctionThatKillsWave1()
   WAVEClear wave1
   AnotherFunction()
End
```

Although memory used for wave1 will be deallocated when `foo` returns, that memory will not be automatically released while the function executes because the WAVE variable still contains a reference to the wave. In this example, WAVEClear deallocates that memory before `AnotherFunction` executes.

You can also use WAVEClear before passing a data folder to preemptive threads using **ThreadGroupPutDF**.

See Also

Accessing Waves in Functions on page IV-65, **Wave Reference Counting** on page IV-180, and **ThreadSafe Functions and Multitasking** on page IV-288.

WaveCRC

WaveCRC(*inCRC*, *waveName* [, *checkHeader*])

The WaveCRC function returns 32 bit cyclic redundancy check value of the bytes in the named wave starting with *inCRC*, which should be zero for the first (or only) set of bytes.

waveName may be a numeric or text wave.

The optional *checkHeader* parameter determines how much of the wave is checked:

checkHeader	What It Does
0	Check only the wave data (default).
1	Check only the internal binary header.
2	Check both.

Details

Polynomial used is:

x^32+x^26+x^23+x^22+x^16+x^12+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x+1

See crc32.c in the public domain source code for zlib for more information.

See Also

The **stringCRC** function.

WaveDims

WaveDims(*wave*)

The WaveDims function returns the number of dimensions used by *wave*.

Returns zero if wave reference is null. See **WaveExists** for a discussion of null wave references.

Also returns zero if wave has zero rows. A matrix will return 2.

WaveExists

WaveExists(*wave*)

The WaveExists function returns one if wave reference is valid or zero if the wave reference is null. For example if, in a user function, you have:

```
Wave w= $"no such wave"
```

then `WaveExists(w)` will return zero.

WaveExists should be used in functions only. In macros, use the exists function instead.

See Also
exists, **NVAR_Exists**, **SVAR_Exists**, and **Accessing Global Variables and Waves** on page IV-50.

WaveInfo

WaveInfo(*waveName*, 0)

The WaveInfo function returns a string containing a semicolon-separated list of information about the named wave.

Details
The string contains six kinds of information. Each group is prefaced by a keyword and colon, and terminated with a semicolon. The keywords are:

Keyword	Information Following Keyword
DUNITS	The data units (y scaling units, zero to three characters).
FULLSCALE	Three numbers indicating whether the wave has any data full scale information, and the min and max data full scale values. The format of the FULLSCALE description is: FULLSCALE:*validFS*,*minFS*,*maxFS*; *validFS* is 1 if *minFS* and *maxFS* have been set via a `SetScale d` command; otherwise it is 0.
LOCK	Reads back the value set by **SetWaveLock**.
MODIFIED	1 if the wave has been modified since the experiment was last saved, else 0.
MODTIME	The date and time that the wave was last modified in seconds since January 1, 1904.
NUMTYPE	A number denoting the numerical type of the wave: 1: Complex, added to one of the following: 2: 32-bit (single precision) floating point 4: 64-bit (double precision) floating point 8: 8-bit signed integer 16: 16-bit signed integer 32: 32-bit signed integer 64: Unsigned, added to 8, 16, or 32 if wave is unsigned For example, the number denoting a complex double precision wave is 5 (i.e., 1+4). See **Setting Bit Parameters** on page IV-12 for details about bit settings.
PATH	The name of the symbolic path in which the wave file is stored (e.g., PATH:home;) or nothing if there is no path for the wave (PATH:;).
XUNITS	The X units (X scaling units, zero to three characters).

Always pass 0 as the second input parameter. In future versions of Igor, this parameter may request other kinds of information to be returned.

A null wave reference returns a zero-length string. This might be encountered, for instance, when using WaveRefIndexed in a loop to act on all waves accessed by **WaveRefIndexed**, and the loop has incremented beyond the highest valid index.

Examples
```
Make/O wave1;SetScale x,0,1,"dyn",wave1;SetScale y,3,20,"v",wave1
String info = WaveInfo(wave1,0)
Print NumberByKey("NUMTYPE", info)          // Prints 2
Print StringByKey("DUNITS", info)           // Prints "v"
```

See Also
The functions and operations listed under "About Waves" categories in the Command Help tab of the Igor Help Browser; among them are **CreationDate**, **modDate**, **WaveType**, **note**, and **numpnts**.

NumberByKey and **StringByKey** functions for parsing the returned keyword list.

WaveInfo lacks information about multidimensional waves. Individual functions are provided to return dimension-related information: **DimDelta**, **DimOffset**, **DimSize**, **WaveUnits**, and **GetDimLabel**.

WaveList

WaveList(*matchStr*, *separatorStr*, *optionsStr*)

The WaveList function returns a string containing a list of waves selected from the current data folder based on *matchStr* and *optionsStr* parameters. See **Details** for information on listing waves in graphs, and for references to newer, data folder-aware functions.

Details

For a wave name to appear in the output string, it must match *matchStr* and also must fit the requirements of *optionsStr* and it must be in the current data folder. The first character of *separatorStr* is appended to each wave name as the output string is generated.

The name of each wave is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything.

For example:

`"*"`	Matches all wave names in current data folder.
`"xyz"`	Matches wave name xyz only, if xyz is in the current data folder.
`"*xyz"`	Matches wave names which end with xyz and are in the current data folder.
`"xyz*"`	Matches wave names which begin with xyz and are in the current data folder.
`"*xyz*"`	Matches wave names which contain xyz and are in the current data folder.
`"abc*xyz"`	Matches wave names which begin with abc and end with xyz and are in the current data folder.

matchStr may begin with the ! character to return windows that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches wave names which do not end with xyz.

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

optionsStr is used to further qualify the wave.

Use `""` to accept all waves in the current data folder that are permitted by *matchStr*.

Set *optionsStr* to one or more of the following comma-separated keyword-value pairs:

optionsStr	Selection Criteria
`"BYTE:0"` *or* `"BYTE:1"`	Waves that are not 8-bit integer (if 0) or only waves that are 8-bit integer (if 1).
`"CMPLX:0"` *or* `"CMPLX:1"`	Waves that are not complex (if 0) or only waves that are complex (if 1).
`"DIMS:`*numberOfDims*`"`	All waves in current data folder that have *numberOfDims* dimensions. This is the number of dimensions reported by **WaveDims**. Use `"DIMS:0"` for all waves having no points (`numpnts(w)==0`). Use "DIMS:1" for graph traces (or one of the X, Y, and Z waves of a contour plot). Use "DIMS:2" for false color and indexed color images (see **Indexed Color Details** on page II-356). Use "DIMS:3" for direct color images (see **Direct Color Details** on page II-358).
`"DP:0"` *or* `"DP:1"`	Waves that are not double precision floating point (if 0) or only waves that are double precision floating point (if 1).
`"INTEGER:0"` *or* `"INTEGER:1"`	Waves that are not 32-bit integer (if 0) or only waves that are 32-bit integer (if 1).
`"MAXCHUNKS:`*max*`"`	Waves having no more than *max* chunks.
`"MAXCOLS:`*max*`"`	Waves having no more than *max* columns.

optionsStr	Selection Criteria
"MAXLAYERS:*max*"	Waves having no more than *max* layers.
"MAXROWS:*max*"	Waves having no more than *max* rows.
"MINCHUNKS:*min*"	Waves having at least *min* chunks.
"MINCOLS:*min*"	Waves having at least *min* columns.
"MINLAYERS:*min*"	Waves having at least *min* layers.
"MINROWS:*min*"	Waves having at least *min* rows.
"SP:0" *or* "SP:1"	Waves that are not single precision floating point (if 0) or only waves that are single precision floating point (if 1).
"TEXT:0" *or* "TEXT:1"	Waves that are not text (if 0) or only waves that are text (if 1).
"UNSIGNED:0" *or* "UNSIGNED:1"	Waves that are not unsigned integer (if 0) or only waves that are unsigned integer (if 1).
"WIN:"	All waves in the current data folder that are displayed in the top graph or table.
"WIN:*windowName*"	All waves in the current data folder that are displayed in the named table or graph window or subwindow.
"WORD:0" *or* "WORD:1"	Waves that are not 16-bit integer (if 0) or only waves that are 16-bit integer (if 1).

You can specify more than one option by separating the options with a comma. See the **Examples**.

Note: Even when *optionsStr* is used to list waves used in a graph or table, the waves must be in the current data folder.

Note: In addition to waves displayed as normal graph traces, WaveList will list matrix waves used with **AppendImage** or NewImage and the X, Y, and Z waves used with **AppendXYZContour**.

Note: Individual contour traces are not listed because they have no corresponding waves. See **All About Contour Traces** on page II-328.

There are several functions that are more useful for listing waves in graphs and tables.

WaveList with WIN:*windowName* gives only the names of the waves in the graph or table and does not include the data folder for each wave. If you need to know what data folder the waves are in, use **WaveRefIndexed** to get the wave itself and then if needed use **GetWavesDataFolder** to get the path.

When identifying a subwindow with WIN:*windowName*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

To list the actual waves used in a graph, or to distinguish two or more instances of the same named wave in a graph, use **TraceNameList**. This function can be used in conjunction with **TraceNameToWaveRef**, and **XWaveRefFromTrace**.

Use **ContourNameList** to list contour plots in a given window and **ContourNameToWaveRef** to access the waves used to generate the contour plot.

To list the contour traces (that is, the contour lines themselves) use **TraceNameList** with the appropriate option.

Use **ImageNameList** to list images in a given window and **ImageNameToWaveRef** to access the waves used to generate the images.

Examples

```
// Returns a list of all waves in the current data folder.
WaveList("*",";","")

// Returns a list of all waves in the current data folder and displayed in the top table or graph.
WaveList("*", ";","WIN:")

// Returns a list of waves in the current data folder whose names
// end in "_bkg" and which are displayed in Graph0 as 1D traces.
WaveList("*_bkg", ";", "WIN:Graph0")

// Returns a list of waves in the current data folder whose names do not
// end in "X" and which are displayed in Graph0 as 1D traces or as one
```

```
// of the X, Y, and Z waves of an AppendXYZContour plot.
WaveList("!*X", ";", "WIN:Graph0,DIMS:1")
```

Contrary to what you might expect, you can *not* use the output of WaveList directly with operations that have a list of waves as their parameters. See **Processing Lists of Waves** on page IV-174 for ways of dealing with this.

See Also

Chapter II-6, **Multidimensional Waves**.

Execute, **ContourNameList**, **ImageNameList**, **TraceNameList**, and **WaveRefIndexed**.

WaveMax

WaveMax(*waveName* [, *x1*, *x2*])

The WaveMax function returns the maximum value in the wave for points between x=*x1* to x=*x2*, inclusive.

Details

If *x1* and *x2* are not specified, they default to -inf and +inf, respectively.

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x2*. To use point indexing, replace *x1* with `pnt2x(waveName,pointNumber1)`, and a similar expression for *x2*.

If the points nearest to *x1* or *x2* are not within the point range of 0 to numpnts(*waveName*)-1, WaveMax limits them to the nearest of point 0 or point numpnts(*waveName*)-1.

For a floating-point wave, WaveMax runs about three times faster than getting the same information using WaveStats. For an integer wave, WaveMax runs about ten times faster than WaveStats. The advantage may not hold for short waves.

See Also

The **WaveMin** function and **WaveStats** operation.

WaveMeanStdv

WaveMeanStdv *srcWave binSizeWave*

The WaveMeanStdv operation calculates the standard deviation of the means for the specified bin distribution saving the result in the wave W_MeanStdv.

For each entry in *binSizeWave*, *srcWave* is divided into the specified number of bins. Values in each bin are averaged and then the mean and standard deviation of the averages (among all bins) are calculated. The value of the standard deviation of the bin averages divided by the mean is then stored in W_MeanStdv corresponding to the bin size entry in *binSizeWave*.

All entries in *binSizeWave* must be positive integers.

Details

When the number of points in *srcWave* does not divide evenly into the bin size entry from *binSizeWave*, the last bin will have a smaller number of data points. In order not to skew the results the values corresponding to the last bin will be dropped. If your data set is small compared to the bin size you might want to pad *srcWave* with additional values (e.g., duplicate values from the beginning of the wave).

This operation does not support NaNs. If you get a NaN as an entry in the output wave then there is either a NaN in *srcWave* or something is wrong with the calculation for that entry.

WaveMin

WaveMin(*waveName* [, *x1*, *x2*])

The WaveMin function returns the minimum value in the wave for points between x=*x1* to x=*x2*, inclusive.

Details

If *x1* and *x2* are not specified, they default to -inf and +inf, respectively.

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x2*. To use point indexing, replace *x1* with `pnt2x(waveName,pointNumber1)`, and a similar expression for *x2*.

If the points nearest to *x1* or *x2* are not within the point range of 0 to numpnts(*waveName*)-1, WaveMin limits them to the nearest of point 0 or point numpnts(*waveName*)-1.

For a floating-point wave, WaveMin runs about three times faster than getting the same information using WaveStats. For an integer wave, WaveMin runs about ten times faster than WaveStats. The advantage may not hold for short waves.

See Also
The **WaveMax** function and **WaveStats** operation.

WaveName

WaveName(*winNameStr, index, type*)

The WaveName function returns a string containing the name of the *index*th wave of the specified *type* in the named window or subwindow.

Parameters
winNameStr can be "" to refer to the top graph or table.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details
WaveName works on waves displayed in a graph, in a table or on the list of waves in the current data folder. If the window is a table, WaveName returns the column name (e.g., "wave0.d"), rather than the name of the wave itself (e.g., "wave0").

For most uses, we recommend that you use **WaveRefIndexed** instead of WaveName. WaveName returns a string containing the wave name only, with no data folder path qualifying it. Thus, you may get erroneous results if the wave referred to in the graph has the same name as a different wave in the current data folder. Likewise, if the named wave resides in a data folder that is not the current data folder, you will not be able to refer to the named wave.

winNameStr is a string expression containing the name of a graph or table or an empty string (""). If the string is empty and *type* is 4 then WaveName works on the list of all waves in the current data folder. If the string is empty and the type parameter is not 4 then WaveName works on the top graph or table.

index starts from zero.

type is a number from 1 to 4. When type is 4 and *winNameStr* is "", WaveName works on the list of all waves in the current data folder.

For graph windows, *type* is 1 for y waves, 2 for x waves, 3 for either y or x waves.

For table windows, *type* is 1 for data columns, 2 for index or dimension label columns, 3 for either data or index or dimension label columns.

WaveName returns an empty string ("") if there is no wave matching the parameters.

Examples
```
WaveName("",0,4)    // Returns name first wave current data folder.
WaveName("",0,1)    // Returns name of first Y wave in the top graph.
WaveName("Graph0",1,2)     // Returns name of second X wave in Graph0.
WaveName("Table0",1,3)     // Returns name of second column in Table0.
```

WaveRefsEqual

WaveRefsEqual(*w1, w2*)

The WaveRefsEqual function returns the truth the two wave references are the same.

Requires Igor Pro 6.20 or later.

See Also
Wave Reference Functions on page IV-173

WaveRefIndexed

WaveRefIndexed(*winNameStr, index, type*)

The WaveRefIndexed function returns a wave reference to the *index*th wave of the specified *type* in the named window or subwindow.

Parameters

winNameStr can be "" to refer to the top graph or table window or the current data folder.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details

WaveRefIndexed is analogous to WaveName but works better with data folders. We recommend that you use it instead of WaveName.

winNameStr is a string expression containing the name of a graph or table or an empty string (""). If the string is empty and *type* is 4 then WaveRefIndexed works on Igor's list of all waves in the current data folder. If the string is empty and the type parameter is not 4 then WaveRefIndexed works on the top graph or table.

index starts from zero.

type is a number from 1 to 4. When type is 4 and *winNameStr* is "", WaveRefIndexed works on the list of all waves in the current data folder.

For graph windows, *type* is 1 for y waves, 2 for x waves, 3 for either y or x waves.

For table windows, *type* is 1 for data columns, 2 for index or dimension label columns, 3 for either data or index or dimension label columns.

WaveRefIndexed returns a null reference (see **WaveExists**) if there is no wave matching the parameters.

Examples

```
WaveRefIndexed("",0,4)     // Returns first wave in current data folder.
WaveRefIndexed("",0,1)     // Returns first Y wave in the top graph.
WaveRefIndexed("Graph0",1,2)      // Returns second X wave in Graph0.
WaveRefIndexed("Table0",1,3)      // wave in second column in Table0.
```

See Also

The **NameOfWave** and **GetWavesDataFolder** functions.

For a discussion of wave references, see **Wave Reference Functions** on page IV-173.

WaveStats

WaveStats [***flags***] ***waveName***

The WaveStats operation computes several statistics on the named wave.

Flags

/ALPH=*val* Sets the significance level for the confidence interval of the mean (default *val*=0.05).

/C=*method* Calculates statistics for complex waves only. Does not affect real waves.

You can use *method* in various combinations to process the real, imaginary, magnitude, and phase of the wave. The result is stored in the wave M_WaveStats (see **Details** for format).

method =0: Default; ignores the imaginary part of *waveName*. Use /W to also store statistics in M_WaveStats.

method =1: Calculates statistics for real part of *waveName* and stores it in M_WaveStats.

method =2: Calculates statistics for imaginary part of *waveName* and stores the result in M_WaveStats.

method =4: Calculates statistics for magnitude of *waveName*, i.e., `sqrt(real^2 +imag^2)`, and stores the result in M_WaveStats.

method =8: Calculate statistics for phase of *waveName* using `atan2(imag,real)`.

If you use a single *method* the results are stored both in M_WaveStats and in the standard variables (e.g., V_avg, etc.). If you specify *method* as a combination of more

than one binary field then the variables reflect the results for the lowest chosen field and all results are stored in the wave M_WaveStats.

For example, if you use /C=12, the variables will be set for the statistics of the magnitude and M_WaveStats will contain columns corresponding to the magnitude and to the phase.

In this mode V_numInfs will always be zero.

Note: if you invoke this operation and M_WaveStats already exists in the current data folder, it will be either overwritten or initialized to NaN.

/M=*moment* Calculates statistical moments.

moment =1: Calculates only lower moments: V_avg, V_npnts, V_numInfs, and V_numNaNs. Use it if you do not need the higher moments.

moment =2: Default; calculates both lower moments and higher order quantities: V_sdev, V_rms, V_adev, V_skew, and v_kurt.

/Q Prevents results from being printed in history.

/R=(*startX*,*endX*) Specifies an X range of the wave to evaluate.

/R=[*startP*,*endP*] Specifies a point range of the wave to evaluate.

If you specify the range as /R=[*startP*] then the end of the range is taken as the end of the wave. If /R is omitted, the entire wave is evaluated.

/W Stores results in the wave M_WaveStats in addition to the various V_ variables when /C=0.

/Z No error reporting.

/ZSCR Computes z scores, $z_i = (Y_i - \overline{Y})/\sigma$, saved in W_ZScore.

Details

WaveStats uses a two-pass algorithm to produce more accurate results than obtained by computing the binomial expansions of the third and fourth order moments.

WaveStats returns the statistics in the automatically created variables:

`V_npnts` Number of points. Does not include points whose value is NaN or INF.

`V_numNans` Number of NaNs.

`V_numINFs` Number of INFs.

`V_avg` Average of Y values.

`V_sdev` Standard deviation of Y values, $\sigma = \sqrt{\frac{1}{V\_npnts - 1}\sum (Y_i - V\_avg)^2}$.

("Variance" is V_sdev^2.)

`V_sem` Standard error of the mean $sem = \sigma / \sqrt{V\_numPnts}$.

`V_rms` RMS (Root Mean Square) of Y values $= \sqrt{\frac{1}{V\_npnts}\sum Y_i^2}$.

`V_adev` Average deviation $= \frac{1}{V\_npnts}\sum_{i=0}^{V\_npnts-1} |x_i - \bar{x}|$.

`V_skew` Skewness $= \frac{1}{V\_npnts}\sum_{i=0}^{V\_npnts-1} \left[\frac{x_i - \bar{x}}{\sigma}\right]^3$.

`V_kurt` Kurtosis $= \frac{1}{V\_npnts}\sum_{i=0}^{V\_npnts-1} \left[\frac{x_i - \bar{x}}{\sigma}\right]^4 - 3$.

`V_minloc` X location of minimum Y value.

V_min	Minimum Y value.
V_maxloc	X location of maximum Y value.
V_max	Maximum Y value.
V_minRowLoc	Row containing minimum data value.
V_maxRowLoc	Row containing maximum data value.
V_minColLoc	Column containing minimum Z value (2D or higher waves).
V_maxColLoc	Column containing maximum Z value (2D or higher waves).
V_minLayerLoc	Layer containing minimum Z value (3D or higher waves).
V_maxLayerLoc	Layer containing maximum Z value (3D or higher waves).
V_minChunkLoc	Chunk containing minimum Z value (4D waves only).
V_maxChunkLoc	Chunk containing maximum Z value (4D waves only).
V_startRow	First wave point. Zero if you do not use /R.
V_endRow	Last wave point. Last point if you do not use /R.

WaveStats prints the statistics in the history area unless /Q is specified. The various multidimensional min and max location variables will only print to the history area for waves having the appropriate dimensionality.

The format of the M_WaveStats wave is:

Row	Statistic	Row	Statistic	Row	Statistic
0	numPoints	9	minLoc	18	maxColLoc
1	numNaNs	10	min	19	maxLayerLoc
2	numInfs	11	maxLoc	20	maxChunkLoc
3	avg	12	max	21	startRow
4	sdev	13	minRowLoc	22	endRow
5	rms	14	minColLoc	23	sum
6	adev	15	minLayerLoc	24	meanL1
7	skew	16	minChunkLoc	25	meanL2
8	kurt	17	maxRowLoc	26	sem

meanL1 and meanL2 are the confidence intervals for the mean

$$MeanL1 = V\_avg - t_{\alpha,v}\frac{V\_sdev}{\sqrt{V\_npnts}} \text{ and } MeanL1 = V\_avg + t_{\alpha,v}\frac{V\_sdev}{\sqrt{V\_npnts}}$$

where $t_{a,v}$ is the critical value of the Student T distribution for *alpha* significance and degree of freedom v=*V_npnts*-1.

Use Edit M_WaveStats.ld to display the results in a table with dimension labels identifying each of the row statistics.

WaveStats is not entirely multidimensional aware. Even so, much of the information computed by WaveStats is useful. See **Analysis on Multidimensional Waves** on page II-110 for details.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

The **ImageStats** operation for calculating wave statistics for specified regions of interest in 2D matrix waves.

The **WaveMax**, **WaveMin**, **mean** and **Variance** functions.

WaveTransform

WaveTransform [*flags*] *keyword srcWave*

The WaveTransform operation transforms *srcWave* in various ways. If the /O flag is not specified then unless otherwise indicated the output is stored in the wave W_WaveTransform, which will be of the same data type as *srcWave* and saved in the current data folder.

Parameters

keyword is one of the following:

abs	Calculates the absolute value of the entries in *srcWave*. It stores results in W_Abs if *srcWave* is 1D or M_Abs otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.
acos	Calculates the inverse cosine of the entries in *srcWave*. It stores results in W_Acos if *srcWave* is 1D or M_Acos otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.
asin	Calculates the inverse sine of the entries in *srcWave*. It stores results in W_Asin if *srcWave* is 1D or M_Asin otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.
atan	Calculates the inverse tangent of the entries in *srcWave*. It stores results in W_Atan if *srcWave* is 1D or M_Atan otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.
cconjugate	Calculates the complex conjugate of *srcWave*. Stores results in W_CConjugate or M_CConjugate, depending on wave dimensionality, or overwrites *srcWave* if /O is used.
cos	Calculates the cosine of the entries in *srcWave*. It stores results in W_Cos if *srcWave* is 1D or M_Cos otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.
crystalToRect	Converts triplet (three column {x,y,z}) waves from nonorthogonal crystallographic coordinates to rectangular cartesian system. The parameters provided in the /P flag are the crystallographic definition of the coordinate system given by {a, b, c, alpha, beta, gamma}. The three angles are assumed to be expressed in radians unless the /D flag is specified. The transformation sets the first component parallel to the vector a and the third component parallel to c*. The output is stored in the current data folder in the wave M_CrystalToRect which has the same data type. If the /O flag is specified, the output overwrites the original data.
flip	Flips the data in *srcWave* about its center. If /O flag is used, *srcWave* is overwritten. Otherwise a new wave is created in the current data folder. The wave is named W_flipped or M_flipped according to the dimensionality of *srcWave*.
index	Fills *srcWave* as in `jack=p`. If /P is specified then `jack=p+`*`param1`*. The /O flag does not apply here.
inverse	Computes `1/srcWave[i]` for each point in *srcWave* and stores it in W_Inverse or M_Inverse depending on the dimensionality of *srcWave*.
inverseIndex	Fills *srcWave* as in `jack=numPnts-1-p`. If /P is specified the `jack=numPnts-1-p+`*`param1`*.
magnitude	Creates a real-valued wave that is the magnitude of *srcWave*. If you do not specify the /O flag, the output is stored in W_Magnitude or M_Magnitude depending on the dimensionality of *srcWave*; the output precision will be the same as *srcWave*.
magsqr	Creates a real-valued wave that is the magnitude squared of *srcWave*. If *srcWave* is a double precision complex wave, the output is also double precision, otherwise the output is a single precision wave. Stores the result in wave W_MagSqr or M_MagSqr, depending on the dimensionality of *srcWave*, or overwrites *srcWave* if /O is used.
max	Calculates the maximum of a point in *srcWave* and a fixed number specified as a single parameter with the /P flag. It stores results in W_max if *srcWave* is 1D or M_max otherwise. It will overwrite *srcWave* when used with the /O flag. See also the min keyword and the example below.

min	Calculates the minimum of a point in *srcWave* and a fixed number specified as a single parameter with the /P flag. It stores results in W_min if *srcWave* is 1D or M_min otherwise. It will overwrite *srcWave* when used with the /O flag. See also the max keyword and the example below.
normalizeArea	Calculates the area under the curve and rescales the wave so that the area is 1. Note that waves with negative areas will be rescaled to positive values. Applies to 1D real-valued waves. It does not affect wave scaling. Stores the result in the wave W_normalizedArea or overwrites *srcWave* if /O is used.
phase	Creates a real-valued wave containing the phase of the complex input wave. If the /O flag is not used, the output is stored in W_Phase or M_Phase depending on the dimensionality of *imageMatrix*. You can also use /P={*norm*} to divide the output wave by the value of *norm*.
rectToCrystal	Converts triplet (three column {x,y,z}) waves from cartesian coordinates to nonorthogonal crystallographic coordinate system. The parameters provided in the /P flag are the crystallographic definition of the coordinate system given by {a, b, c, alpha, beta, gamma}. The three angles are assumed to be expressed in radians unless the /D flag is specified. The transformation assumes the first component parallel to the vector a and the third component parallel to c*. The output is stored in the current data folder in the wave M_RectToCrystal which has the same data type. If the /O flag is specified, the output overwrites the original data.
sgn	Sets the value to -1 if the entry is negative, 1 otherwise. Stores the results in W_Sgn or overwrites *srcWave* if /O is used. This operation will not work on UNSIGNED waves.
shift	Shifts the position of data in *srcWave* by the specified number of points. Unlike Rotate, WaveTransform discards data points that shift outside existing wave boundaries. After the shift, vacated wave points are set to the specified *fillValue*. The shift and the *fillValue* are specified with the /P flag using the syntax: /P={*numPoints*, *fillValue*}. If you do not provide a fill value, it will be 0 for integer waves and NaN for SP and DP.
sqrt	Calculates the square root of the entries in *srcWave*. It stores results in W_sqrt if *srcWave* is 1D or M_sqrt otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single- or double-precision real wave.
tan	Calculates the tangent of the entries in *srcWave*. The results are stored in W_tan if *srcWave* is 1D or M_tan otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single- or double-precision real wave.
zapINFs	Deletes elements whose value is infinity or -infinity. This is relevant for 1D single-precision and double-precision floating point waves only and does nothing for other types of 1D waves. It is not suitable for multi-dimensional waves and returns an error if *srcWave* is multi-dimensional. Use **MatrixOp** `replace` for multi-dimensional waves.
zapNaNs	Deletes elements whose value is NaN. This is relevant for 1D single-precision and double-precision floating point waves only and does nothing for other types of 1D waves. It is not suitable for multi-dimensional waves and returns an error if *srcWave* is multi-dimensional. Use **MatrixOp** `replaceNaNs` for multi-dimensional waves.

Flags

/D	If present, angles in wave data are interpreted as in degrees. Otherwise they are interpreted as in radians.
/O	Overwrites input wave.
/P={*param1*…}	Specifies parameters as appropriate for the keyword that you are using. The number of parameters and their order depends on the keyword.

Examples

```
// Produce output values in the range [-1,1]:
WaveTransform /P={(pi)} phase complexWave

// Faster than myWave=myWave>1 ? 1 : myWave
WaveTransform /P={1}/O min myWave
```

See Also

The **Rotate** operation.

References

Shmueli, U. (Ed.), International Tables for Crystallography, Volume B: 3.3, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996.

WaveType

WaveType(*waveName* [,*selector*])

The WaveType function returns the type of data stored in the wave.

The optional *selector* parameter requires Igor Pro 6.1 or later.

If *selector* = 1, WaveType returns 0 for a null wave, 1 if numeric, 2 if string, 3 if the wave holds data folder references or 4 if the wave holds wave references.

If *selector* = 2, WaveType returns 0 for a null wave, 1 for a normal global wave or 2 for a free wave or a wave that is stored in a free data folder.

If *selector* is omitted or zero, the returned value for non-numeric waves (text waves, wave-reference waves and data folder-reference waves) is 0.

If *selector* is omitted or zero, the returned value for numeric waves is a combination of bit values shown in the following table:

Type	Bit Number	Decimal Value	Hexadecimal Value
complex	0	1	1
32-bit float	1	2	2
64-bit float	2	4	4
8-bit integer	3	8	8
16-bit integer	4	16	10
32-bit integer	5	32	20
unsigned	6	64	40

The unsigned bit is used only with the integer types while the complex bit can be used with any numeric type. Only one of the bits 1-5 are set at a time. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Examples

```
Variable waveIsComplex = WaveType(wave) & 0x01
Variable waveIs32BitFloat = WaveType(wave) & 0x02
Variable waveIs64BitFloat = WaveType(wave) & 0x04
Variable waveIs8BitInteger = WaveType(wave) & 0x08
Variable waveIs16BitInteger = WaveType(wave) & 0x10
Variable waveIs32BitInteger = WaveType(wave) & 0x20
Variable waveIsUnsigned = WaveType(wave) & 0x40
```

See Also

For concepts related to selector = 1 or 2, see **Free Waves** on page IV-71, **Wave Reference Waves** on page IV-60 and **Data Folder Reference Waves** on page IV-65.

WaveUnits

WaveUnits(*waveName, dimNumber*)

The WaveUnits function returns a string containing the units for the given dimension.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers, and 3 for chunks. Use -1 to get the data units. If the wave is just 1D, *dimNumber*=0 returns X units and 1 returns data units. This behavior is just like the WaveMetrics procedure WaveUnits found in the WaveMetrics Procedures folder in previous versions of Igor Pro.

wfprintf

wfprintf *refNumOrStr*, *formatStr* [*flags*] *waveName* [, *waveName*]…

The wfprintf operation is like the **printf** operation except that it prints the contents of the named waves to a file whose file reference number is in *refNum*.

The **Save** operation also outputs wave data to a text file. Use Save unless you need the added flexibility provided by wfprintf.

Parameters

refNumOrStr is a numeric expression, a string variable or an SVAR pointing to a global string variable.

If a numeric expression, then it is a file reference number returned by the **Open** operation or an expression that evaluates to 1.

If *refNumOrStr* is 1, Igor prints to the history area instead of to a file.

If *refNumOrStr* is the name of a string variable, the wave contents are "printed" to the named string variable. *refNumOrStr* can also be the name of an SVAR to print to a global string:

```
SVAR sv = root:globalString
wfprintf sv, "", wave0
```

refNumOrStr can not be an element of a text wave.

The value of each named wave is printed to the file according to the conversion specified in *formatStr*.

formatStr contains one numeric conversion specification per column. See **printf**. If *formatStr* is `""`, wfprintf uses a default format which gives tab-delimited columns.

Flags

Note: /R must follow the *formatStr* parameter directly without an intervening comma.

/R=(*startX*,*endX*) Specifies an X range in the wave(s) to print.

/R=[*startP*,*endP*] Specifies a point range in the wave(s) to print.

Details

A complex wave in the wavelist is treated as two separate columns and requires two separate conversion specifications if you do not use the default `""` specification.

The number of conversion characters must exactly match the number of columns (one column per real wave, two columns per complex wave) in the wave list. The wave list is limited to 100 waves. You can output up to 200 columns by using complex waves instead of real waves.

The only conversion characters allowed are: fFeEgdouxXcs (the floating point, integer and string conversion characters). You cannot use an asterisk to specify field width or precision.

If any of these restrictions is intolerable, you can use fprintf in a loop.

The wfprintf operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-110 for details.

Examples

```
Function Example1()
   Make/O/N=10 wave0=sin(p*pi/10)                  // test numeric wave
   Make/O/N=10/T textWave= "row "+num2istr(p)      // test text wave
   Variable refNum
   Open/P=home refNum as "output.txt"// open file for write
   wfprintf refNum, "%s = %g\r"/R=[0,5], textWave, wave0    // print 6 values each
   Close refNum
End
```

The resulting output.txt file contains:

```
row 0 = 0
row 1 = 0.309017
row 2 = 0.587785
row 3 = 0.809017
row 4 = 0.951057
row 5 = 1
```

```
Function/S NumericWaveToStringList(w)
   Wave w                          // numeric wave (if text, use /T here and %s below)
```

```
    String list
    wfprintf list, "%g;" w        // semicolon-separated list
    return list
End

Print NumericWaveToStringList(wave0)
  0;0.309017;0.587785;0.809017;0.951057;1;0.951057;0.809017;0.587785;0.309017;
```

See Also

The **printf** operation for complete format and parameter descriptions and **Creating Formatted Text** on page IV-230. The **Open** operation about *refNum* and for another way of writing wave files.

The **Save** operation.

WhichListItem

WhichListItem(*itemStr*, *listStr* [, *listSepStr* [, *startIndex* [, *matchCase*]]])

The WhichListItem function returns the index of the first item of *listStr* that matches *itemStr*. *listStr* should contain items separated by the *listSepStr* character, such as "abc;def;". If the item is not found in the list, -1 is returned.

Use WhichListItem to locate an item in a string containing a list of items separated by a single character, such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

listSepStr, *startIndex*, and *matchCase* are optional; their defaults are ";", 0, and 1 respectively.

Details

WhichListItem differs from **FindListItem** in that WhichListItem returns a list index, while FindListItem returns a character offset into a string.

listStr is searched for *itemStr* bound by *listSepStr* on the left and right.

listStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *listSepStr* are always case-sensitive. The comparison of *itemStr* to the contents of *listStr* is usually case-sensitive. Setting the optional *matchCase* parameter to 0 makes the comparison case insensitive.

If *itemStr* is not found, if *listStr* is `""`, or if *startIndex* is not within the range of 0 to ItemsInList(*listStr*)-1, then -1 is returned.

Only the first character of *listSepStr* is used.

Items can be empty. The lists `"abc;def;;ghi"` and `";abc;def;;ghi;"` have four items (the third item is `""`). Use `WhichListItem("", listStr)` to find the first empty item.

If *startIndex* is specified, then *listSepStr* must also be specified. If *matchCase* is specified, *startIndex* and *listSepStr* must be specified.

Examples

```
Print WhichListItem("wave0", "wave0;wave1;")          // prints 0
Print WhichListItem("c", "a;b;")                      // prints -1
Print WhichListItem("", "a;;b;")                      // prints 1
Print WhichListItem("c", "a,b,c,x,c", ",")            // prints 2
Print WhichListItem("c", "a,b,c,x,c", ",", 3)         // prints 4
Print WhichListItem("C", "x;c;C;")                    // prints 2
Print WhichListItem("C", "x;c;C;", ";", 0, 0)         // prints 1
```

See Also

The **AddListItem**, **FindListItem**, **FunctionList**, **ItemsInList**, **RemoveFromList**, **RemoveListItem**, **StringFromList**, **StringList**, **TraceNameList**, **VariableList**, and **WaveList** functions.

WignerTransform

WignerTransform [**/Z**] [**/WIDE=*wSize***] [**/GAUS=*gaussianWidth***] [**/DEST=*destWave***] ***srcWave***

The WignerTransform operation computes the Wigner transformation of a 1D signal in *srcWave*, which is the name of a real or complex wave. The result of the WignerTransform is stored in *destWave* or in the wave M_Wigner in the current data folder.

Flags

/DEST=*destWave* Creates by default a real wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-56 for details.

/GAUS=*gWidth* Computes the Gaussian Wigner Transform, which is a convolution of the Wigner Transform with a two-dimensional Gaussian (in the two parameters of the transform). The computation of the transform simplifies significantly when the product of the widths of the two Gaussians is unity (minimum uncertainty ellipse).

gWidth uses the same units as the *srcWave* scaling.

/WIDE=*wSize* Computes Wigner Transform and sets the transform width to *wSize*. This is the default transformation with *wSize* set to the size of *srcWave*.

/Z No error reporting.

Details

The Wigner transform maps a time signal $U(t)$ into a 2D time-frequency representation:

$$W(t,\nu) = \int_{-\infty}^{\infty} U\left(t+\frac{x'}{2}\right)U^*\left(t-\frac{x'}{2}\right)e^{-2\pi i x'\nu}dx' .$$

The computation of the Wigner transform evaluates the offset product

$$U\left(t+\frac{x'}{2}\right)U^*\left(t-\frac{x'}{2}\right)$$

over a finite window and then Fourier transforms the result. The offset product can be evaluated over a finite window width, which can vary from a few elements of the input wave to the full length of the wave. You can control the width of this window using the /WIDE flag. If you do not specify the output destination, WignerTransform saves the results in the wave M_Wigner in the current data folder.

The input wave can be of any numeric data type, and the output data type and precision will be the same as the input. Although the Wigner transform is real, the output will be complex when *srcWave* is complex. By inspecting the complex wave you can gain some insight into the numerical stability of the algorithm. The X-scaling of the output wave is identical to the scaling of *srcWave*. The Y-scaling of the input wave is taken from the Fourier Transform of the offset product, which in turn is determined by the X-scaling of *srcWave*. Specifically, if `dx=DimDelta(srcWave,0)` and *srcWave* has N points then `dy=DimDelta(M_Wigner,1)=1/(dx*N)`. WignerTransform does not set the units of the output wave.

The Ambiguity Function is related to the Wigner Transform by a Fourier Transform, and is defined by

$$A(\tau,\nu) = \int_{-\infty}^{\infty} U\left(t+\frac{\tau}{2}\right)U^*\left(t-\frac{\tau}{2}\right)e^{-2\pi i t\nu}dt .$$

Convolving the Wigner Transform with a 2D Gaussian leads to what is sometimes called the Gaussian Wigner Transform or GWT. Formally the GWT is given by the equation:

$$GWT(t,\nu;\delta_t,\delta_\nu) = \frac{1}{\delta_t\delta_\nu}\int\int dt'd\nu' W(t',\nu')e^{-2\pi\left(\frac{t-t'}{\delta_t}\right)^2}e^{-2\pi\left(\frac{\nu-\nu'}{\delta_\nu}\right)^2} .$$

Computationally this equation simplifies if the respective widths of the two Gaussians satisfy the minimum uncertainty condition $\delta_t*\delta_\nu=1$. The /Gaus flag calculates the Gaussian Wigner Transform using your specified width, δ_t, and it selects a δ_ν such that it satisfies the minimum uncertainty condition.

See Also

CWT, **FFT**, and **WaveTransform** operations.

For further discussion and examples see **Wigner Transform** on page III-245.

References

Wigner, E. P., On the quantum correction for thermo-dynamic equilibrium, *Physics Review*, *40*, 749-759, 1932.

Bartelt, H.O., K.-H. Brenner, and A.W. Lohman, The Wigner distribution function and its optical production, *Optics Communications*, *32*, 32-38, 1980.

Window

Window *macroName*([*parameters*]) [:*macro type*]

The Window keyword introduces a macro that recreates a graph, table, layout, or control panel window. The macro appears in the appropriate submenu of the Windows menu. Window macros are automatically created when you close a graph, table, layout, control panel, or XOP target window (e.g., surface plot). You should use **Macro**, **Proc**, or **Function** instead of Window for your own window macros. Otherwise, it works the same as **Macro**.

See Also

The **Macro**, **Proc**, and **Function** keywords. **Data Folders and Window Macros** on page II-125 for details.

Macro Syntax on page IV-98 for further information.

WindowFunction

WindowFunction [/FFT[=*f*] /DEST=*destWave*] *windowKind, srcWave*

The WindowFunction operation multiplies a one-dimensional (real or complex) *srcWave* by the named window function.

By default the result overwrites *srcWave*.

Parameters

srcWave A one-dimensional wave of any numerical type. See **ImageWindow** for windowing two-dimensional data.

windowKind Specifies the windowing function, which are listed in bold (some windows, in italics, have multiple options for different constant values). The equations assume that /FFT=1:

Bartlet: a synonym for Bartlett.

Bartlett: $$w(n) = \begin{cases} \frac{n}{N/2} & n = 0, 1, \ldots, \frac{N}{2} \\ W(N-n) & n = \frac{N}{2}, \ldots, N-1 \end{cases}.$$

Blackman: $$w(n) = a_0 - a_1\cos\left(\frac{2\pi}{N}n\right) + a_2\cos\left(\frac{2\pi}{N}2n\right) - a_3\cos\left(\frac{2\pi}{N}3n\right) \quad .$$

$$n = 0,1,2,\ldots N-1$$

windowKind	a_0	a_1	a_2	a_3
Blackman367	0.42323	0.49755	0.07922	
Blackman361	0.44959	0.49364	0.05677	
Blackman492	0.35875	0.48829	0.14128	0.01168
Blackman474	0.40217	0.49703	0.09392	0.00183

Cos: $$w(n) = \cos\left(\frac{n}{N}\pi\right)^{\alpha} \qquad n = -\frac{N}{2}, \ldots, -1, 0, 1, \ldots, \frac{N}{2}$$

Cos1: $\alpha = 1.$

Cos2: $\alpha = 2.$

Cos3: $\alpha = 3.$

Cos4: $\alpha = 4.$

Hamming: $$w(n) = \begin{cases} 0.54 + 0.46\cos\left(\frac{2\pi}{N}n\right) & n = -\frac{N}{2}, \ldots, -1, 0, 1, \ldots, \frac{N}{2} \\ 0.54 - 0.46\cos\left(\frac{2\pi}{N}n\right) & n = 0, 1, 2, \ldots, N-1 \end{cases}.$$

Hanning: $$w(n) = \begin{cases} \frac{1}{2}\left[1 + \cos\left(\frac{2n}{N}\pi\right)\right] & n = -\frac{N}{2}, \ldots, -1, 0, 1, \ldots, \frac{N}{2} \\ \frac{1}{2}\left[1 - \cos\left(\frac{2n}{N}\pi\right)\right] & n = 0, 1, 2, \ldots, N-1 \end{cases}.$$

Kaiser Bessel: $$w(n) = \frac{I_0\left(\pi\alpha\sqrt{1-\left(\frac{n}{N/2}\right)^2}\right)}{I_0(\pi\alpha)} \qquad 0 \le |n| \le \frac{N}{2}.$$

$$I_0(X) = \sum_{k=0}^{\infty}\left(\frac{(x/2)^2}{k!}\right)^2$$

KaiserBessel20: $\alpha = 2.$

KaiserBessel25: $\alpha = 2.5.$

KaiserBessel30: $\alpha = 3.$

Parzen: $$w(n) = 1 - \left|\frac{n}{N/2}\right|^2 \qquad 0 \le |n| \le \frac{N}{2}.\text{WWN}$$

Poisson: $$w(n) = \exp\left(-\alpha\frac{|n|}{N/2}\right) \qquad 0 \le |n| \le \frac{N}{2}.$$

Poisson2: $\alpha = 2.$

Poisson3: $\alpha = 3.$

Poisson4: $\alpha = 4.$

Riemann: $$w(n) = \frac{\sin\left(\frac{n}{N}2\pi\right)}{\left(\frac{n}{N}2\pi\right)} \qquad 0 \le |n| \le \frac{N}{2}.$$

Flags

/DEST=*destWave* Creates or overwrites *destWave* with the result of the multiplication of *srcWave* and the window function.

When used in a function, the WindowFunction operation by default creates a real wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-56 for details.

/FFT [=1] The window interval is `0...N=numpnts(`*`srcWave`*`)`. This sets the first value of *srcWave* to zero, but not the last value. This is appropriate for windowing data in preparation for Fourier Transforms, and is the same algorithm used by **FFT**.

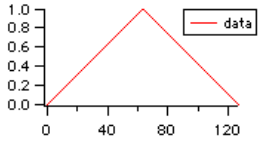
The window interval is `0...N=numpnts(`*`srcWave`*`)-1` if /FFT is missing or /FFT=0. This sets the first and last value of *srcWave* to 0. This is the (only) algorithm that the Hanning operation uses.

Details

A "window function" alters the input data by decreasing values near the start and end of the data smoothly towards zero, so that when the FFT of the data is computed the effects of nonintegral-periodic signals are diminished. This improves the ability of the FFT to distinguish among closely-spaced frequencies. Each window function has advantages and disadvantages, usually trading off rejection of "leakage" against the ability to discriminate adjacent frequencies. For more details, see the **References**.

WindowFunction stores the window function's normalization value (the average squared window value) in V_value. This is the value you would get from WaveStats's `V_rms*V_rms` for a wave of *srcWave*'s length whose values were all equal to 1:

```
Make/O data = 1
WindowFunction Bartlet, data      // Bartlet allowed as synonym for Bartlett
Print V_value              // Prints 0.330709, mean of squared window values
```




```
WaveStats/Q data
Print V_rms*V_rms              // Prints 0.330709
```

See Also

FFT and **ImageWindow** operations.

References

For more information about the use of window functions see:

Harris, F.J., On the use of windows for harmonic analysis with the discrete Fourier Transform, *Proc, IEEE, 66*, 51-83, 1978.

Wikipedia entry: <http://en.wikipedia.org/wiki/Window_function>.

WinList

WinList(*matchStr, separatorStr, optionsStr*)

The WinList function returns a string containing a list of windows selected based on the *matchStr* and *optionsStr* parameters.

Details

For a window name to appear in the output string, it must match *matchStr* and also must fit the requirements of *optionsStr*. The first character of *separatorStr* is appended to each window name as the output string is generated.

The name of each window is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches all window names
"xyz"	Matches window name xyz only
"*xyz"	Matches window names which end with xyz
"xyz*"	Matches window names which begin with xyz
"*xyz*"	Matches window names which contain xyz
"abc*xyz"	Matches window names which begin with abc and end with xyz

matchStr may begin with the ! character to return windows that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches window names which *do not* end with xyz

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

optionsStr is used to further qualify the window. The acceptable values for *optionsStr* are:

`""`	Consider all windows.
`"WIN:"`	The target window.
`"WIN:`*`windowTypes`*`"`	Consider windows that match *windowTypes*.
`"INCLUDE:`*`includeTypes`*`"`	Consider procedure windows that match *includeTypes*.
`"INDEPENDENTMODULE:1"`	If `SetIgorOption independentModuleDev=1`, then consider procedure windows that are part of any independent module. If `SetIgorOption independentModuleDev=0` or if `INDEPENDENTMODULE:0`, then procedure windows that are part of any independent module are not listed.
"FLT:1"	Return only panels that were created with NewPanel/FLT=1. Specifying "FLT" also implies "WIN:64". Omit FLT or use "FLT:0" to return windows that do not float (and most do not).
"FLT:2"	Return only panels that were created with NewPanel/FLT=2. Specifying "FLT" also implies "WIN:64".

"VISIBLE:1" Return only visible windows (ignore hidden windows).

windowTypes is a literal number. The window name goes into the output string only if it passes the match test and its type is compatible with *windowTypes*. *windowTypes* is one of:

```
  1:  Graphs
  2:  Tables
  4:  Layouts
 16:  Notebooks
 64:  Panels
128:  Procedure windows
512:  Help windows
4096: XOP target windows (e.g., Gizmo 3D plots)
```

or a bitwise combination of the above for more than one type of inclusion. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Procedure windows and help windows don't have names. WinList returns the window title instead.

includeTypes is also a literal number. The window name goes into the output string only if it passes the match test and its type is compatible with *includeTypes*. *includeTypes* is one of:

1: Procedure windows that are not #included.

2: Procedure windows included by `#include "someFileName"`.

4: Procedure windows included by `#include <someFileName>`.

or a bitwise combination of the above for more than one type of inclusion.

You can combine the `WIN`, `INCLUDE` and `INDEPENDENTMODULE` options by separating them with a comma.

When executing `SetIgorOption independentModuleDev=1` after opening the current experiment, the title of any procedure window in an independent module will be followed by `[nameOfIndependentModule]`.

For example, if a procedure file contains:

```
#pragma IndependentModule=myIndependentModule
#include <Axis Utilities>
```

A call to WinList like this:

```
String procedureWindows = WinList("*", ";", "WIN:128,INDEPENDENTMODULE:1")
```

will store "Axis Utilities.ipf [myIndependentModule];" into the `procedureWindows` string, along with any other procedure windows that are part of that independent module. The procedure window that contains the

```
#pragma IndependentModule=myIndependentModule
```

statement is also listed with the independent module name.

To get a list of procedure windows that comprise a particular independent module, you can use a call such as:

```
SetIgorOption IndependentModuleDev=1
list= WinList("* [myIndependentModule]",";","WIN:128,INDEPENDENTMODULE:1")
```

Examples

Command	Returned List
`WinList("*",";","")`	All existing non-floating windows.
`WinList("*", ";","WIN:3")`	All graph and table windows.
`WinList("Result_*", ";", "WIN:1")`	Graphs whose names start with "Result_".
`WinList("*", ";","WIN:64,FLT:1,FLT:2")`	All floating panel windows.
`WinList("*", ";","INCLUDE:6")`	All #included procedure windows.
`WinList("*", ";","WIN:1,INCLUDE:6")`	All graphs and #included procedure windows.

See Also

Independent Modules on page IV-214. The **ChildWindowList** and **WinType** functions.

WinName

WinName(*index, windowTypes* [, *visibleWindowsOnly* [, *floatKind*]])

The WinName function returns a string containing the name of the *index*th window of the specified *type*, or an empty string ("") if no window fits the parameters.

If the optional *visibleWindowsOnly* parameter is nonzero, only visible windows are considered. Otherwise both visible and hidden windows are considered.

If the optional *floatKind* parameter is 1, only floating windows created with NewPanel/FLT=1 are considered. If *floatKind* is 2, only NewPanel/FLT=2 windows are considered. *windowTypes* must contain at least 64 (panels).

If *floatKind* is omitted or is 0 only non-floating ("normal") windows are considered.

Procedure windows don't have names. WinName returns the procedure window title instead.

Details

index starts from zero, and returns the top-most window matching the parameters.

The window names are ordered in window-stacking order, as returned by WinList.

`DoWindow/B` moves the window to the back and changes the index needed to retrieve its name to the greatest index that returns any name.

Hiding or showing a window (with `SetWindow hide=1` or `Notebook visible=0` or by manual means) does not affect the index associated with the window.

windowTypes is one of:

1:	Graphs.
2:	Tables.
4:	Layouts.
16:	Notebooks.
64:	Panels.
128:	Procedure windows.
4096:	XOP target windows (e.g., Gizmo 3D plots).

or a bitwise combination of the above for more than one type of window. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Examples

```
Print WinName(0,1)          // Prints the name of the top graph.
Print WinName(0,3)          // Prints the name of the top graph or table.
String win=WinName(0,1)     // The name of the top visible graph.
SetWindow $win hide=1       // Hide the graph (it may already be hidden).
Print WinName(0,1)          // Prints the name of the now-hidden graph.
Print WinName(0,1,1)        // Prints the name of the top visible graph.
Print WinName(0,64,1,1)     // Name of the top visible NewPanel/FLT=1 window.
```

See Also

WinList, **DoWindow** (/F and /B flags), **SetWindow** (hide keyword), **Notebook (Miscellaneous)** (visible keyword), **NewPanel** (/FLT flag).

WinRecreation

WinRecreation(*winStr, options*)

The WinRecreation function returns a string containing the window recreation macro (or style macro) for the named window.

Parameters

winStr is the name of a graph, table, page layout, panel, notebook, or XOP target window or the title of a procedure window or help file. If *winStr* is "" and options is 0 or 1, information for the top graph, table, page layout, panel, notebook, or XOP target window is returned.

The meaning of *options* depends on the type of window as described in the following sections.

Target Window Details

Target windows include graphs, tables, page layouts, panels, notebooks, and XOP target windows.

If *options* is 0, WinRecreation returns the window recreation macro.

If *options* is 1, WinRecreation returns the style macro or an empty string if the window does not support style macros.

Graphs Details

If *options* is 2, WinRecreation returns a recreation macro in which all occurrences of wave names are replaced with an ID number having the form ##<number>## (for instance, ##25##). These ID numbers can be found easily using the **strsearch** function. This is intended for applications that need to alter the recreation macro by replacing wave names with something else, usually other wave names. The ID numbers are the same as those returned by the **GetWindow** operation with the wavelist keyword.

Graphs and Panels Details

If *options* is 4, WinRecreation returns the window recreation macro without the default behavior of causing the graph to revert to "normal" mode (as if the GraphNormal operation had been called). This allows the use of WinRecreation when a graph or panel is in drawing tools mode without exiting that mode. For windows other than graphs or panels, this is equivalent to an *options* value of 0.

Notebooks Details

If *options* is -1, WinRecreation returns the same text that the Generate Commands menu item would generate with the Selected paragraphs radio button selected and all the checkboxes selected (includes text commands).

If *options* is 0, WinRecreation returns the same text that the Generate Commands menu item would generate with the Entire document radio button selected and all the checkboxes *except* "Generate text commands" selected).

If *options* is 1, WinRecreation returns the same text that the Generate Commands menu item would generate with the Entire document radio button selected and all the checkboxes selected (includes text commands).

Regardless of the value of *options* the text returned by WinRecreation for notebook always ends with 5 lines of file-related information formatted as comments:

```
// File Name: MyNotebook.txt
// Path: "Macintosh HD:Desktop Folder:"
// Symbolic Path: home
// Selection Start: paragraph 100, position 31
// Selection End: paragraph 100, position 31
```

Help Windows Details

WinRecreation returns the same 5 lines of file-related information as described above for notebooks.

Set *options* to -3 to ensure that *winStr* is interpreted as a help window title (help windows have only titles, not window names).

Procedures Details

WinRecreation returns the same 5 lines of file-related information as described above for notebooks.

Set *options* to -2 to ensure that *winStr* is interpreted as a procedure window title (procedure windows have only titles, not window names).

If `SetIgorOption IndependentModuleDev=1` is in effect, *winStr* can also be a procedure window title followed by a space and, in brackets, an independent module name. In such cases WinRecreation returns text from or information about the specified procedure file which is part of that independent module. (See **Independent Modules** on page IV-214 for independent module details.)

For example, in an experiment containing:

```
#pragma IndependentModule=myIM
#include <Axis Utilities>
```

code like this:

```
String text=WinRecreation("Axis Utilities.ipf [myIM]",-2)
```

will return the file-related information for the Axis Utilities.ipf procedure window, which is normally a hidden part of the myIM independent module.

To get the text content of a procedure window, use the **ProcedureText** function.

Examples

```
WinRecreation("Graph0",0)     // Returns recreation macro for Graph0.
WinRecreation("",1)                     // Style macro for top window.
String win= WinName(0,16,1)                // top visible notebook
String str= WinRecreation(str,-1)          // Selected Text commands
Variable line= itemsInList(str,"\r")-5     // First file info line
Print StringFromList(line, str,"\r")       // Print File Name:
Print StringFromList(line+1, str,"\r")     // Print Path:
Print StringFromList(line+2, str,"\r")     // Print Symbolic Path:
Print StringFromList(line+3, str,"\r")     // Selection Start:
Print StringFromList(line+4, str,"\r")     // Selection End:
```

See Also

Saving a Window as a Recreation Macro on page II-61.

WinType

WinType(*winNameStr*)

The WinType function returns a value indicating the type of the named window.

Details

winNameStr is a string or string expression containing the name of a window or subwindow, or "" to signify the target window. When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

WinType returns the following values:

0: No window by that name.
1: Graph.
2: Table.
3: Layout.
5: Notebook.
7: Panel.
13: XOP target window (e.g., surface plot).

Because command and procedure windows do not have *names* (they only have *titles*), WinType can not even be asked about those windows.

See Also

The **WinName**, **ChildWindowList**, and **WinList** functions.

WMAxisHookStruct

See **NewFreeAxis** for further explanation of WMAxisHookStruct.

```
Structure WMAxisHookStruct
   char win[200]       // Host window or subwindow name
   char axName[32]     // Name of axis
   char mastName[32]   // Name of controlling axis or ""
   char units[50]      // Axis units.
   Variable min        // Current axis range minimum value
   Variable max        // Current axis range maximum value
EndStructure
```

WMBackgroundStruct

See **CtrlNamedBackground**, **Background Tasks** on page IV-279, and **Preemptive Background Task** on page IV-292 for further explanation of WMBackgroundStruct.

```
Structure WMBackgroundStruct
   char name[32]       // Background task name
   UInt32 curRunTicks  // Tick count when task was called
   Int32 started       // TRUE when CtrlNamedBackground start is issued
   UInt32 nextRunTicks // Precomputed value for next run
                       // but user functions may change this
EndStructure
```

WMButtonAction

See **Button** for further explanation of WMButtonAction.

```
Structure WMButtonAction
    char ctrlName[32]    // Control name
    char win[200]        // Host window or subwindow name
    Rect winRect         // Local coordinates of host window
    Rect ctrlRect        // Enclosing rectangle of the control
    Point mouseLoc       // Mouse location
    Int32 eventCode      // -1: Control being killed
                         //  1: Mouse down
                         //  2: Mouse up
                         //  3: Mouse up outside control
                         //  4: Mouse moved
                         //  5: Mouse enter
                         //  6: Mouse leave
    Int32 eventMod       // See Control Structure eventMod Field on page III-385
    String userData      // Primary unnamed user data.
    Int32 blockReentry   // See Control Structure blockReentry Field on page III-386
EndStructure
```

WMCheckboxAction

See **CheckBox** for further explanation of WMCheckboxAction.

```
Structure WMCheckboxAction
    char ctrlName[32]    // Control name
    char win[200]        // Host window or subwindow name
    Rect winRect         // Local coordinates of host window
    Rect ctrlRect        // Enclosing rectangle of the control
    Point mouseLoc       // Mouse location
    Int32 eventCode      // -1: Control being killed
                         //  2: Mouse up
    Int32 eventMod       // See Control Structure eventMod Field on page III-385
    String userData      // Primary unnamed user data
    Int32 blockReentry   // See Control Structure blockReentry Field on page III-386
    Int32 checked        // Checkbox state
EndStructure
```

WMCustomControlAction

See **CustomControl** for further explanation of WMCustomControlAction.

```
Structure WMCustomControlAction
    char ctrlName[32]    // Control name
    char win[200]        // Host window or subwindow name
    Rect winRect         // Local coordinates of host window
    Rect ctrlRect        // Enclosing rectangle of the control
    Point mouseLoc       // Mouse location
    Int32 eventCode      // See CustomControl for details
    Int32 eventMod       // See Control Structure eventMod Field on page III-385
    String userData      // Primary unnamed user data
    Int32 blockReentry   // See Control Structure blockReentry Field on page III-386
    Int32 missedEvents   // TRUE when events occurred but the user
                         // function was not available for action
    Int32 mode           // General purpose
    Int32 isVariable     // TRUE if varName is a variable
    Int32 isWave         // TRUE if varName referenced a wave
    Int32 isString       // TRUE if varName is a String type
    NVAR nVal            // Valid if isVariable and not isString
    SVAR sVal            // Valid if isVariable and isString
    WAVE nWave           // Valid if isWave and not isString
    WAVE/T sWave         // Valid if isWave and not isString
    Int32 rowIndex       // If isWave, this is the row index
                         // unless rowLabel is not empty
    char rowLabel[32]    // Wave row label
    Int32 kbChar         // Keyboard key character code
    Int32 kbMods         // Keyboard key modifiers bit field
                         // Bit 0: Command pressed (Macintosh)
                         // Bit 1: Shift pressed
                         // Bit 2: Alpha pressed lock
                         // Bit 3: Option (Mac) or Alt (Windows) pressed
                         // Bit 4: Control pressed
    Int32 curFrame       // Input and output, used with kCCE_frame event
```

```
    Int32 needAction    // See CustomControl for details
EndStructure
```

WMFitInfoStruct

See **The `WMFitInfoStruct` Structure** on page III-228 for further explanation of WMFitInfoStruct.

```
Structure WMFitInfoStruct
    char IterStarted    // Nonzero on the first call of an iteration
    char DoingDestWave  // Nonzero when called to evaluate autodest wave
    char StopNow        // Fit function sets this to nonzero to
                        // indicate that a problem has occurred
                        // and fitting should stop
    Int32 IterNumber    // Number of iterations completed
    Int32 ParamPerturbed// See The WMFitInfoStruct Structure on page III-228 for details
EndStructure
```

WMGizmoHookStruct

See ModifyGizmo in the Gizmo Reference help file for further explanation of WMGizmoHookStruct.

```
Structure WMGizmoHookStruct
    Int32 version
    char winName[32]
    char eventName[32]
    Int32 width
    Int32 height
    Int32 mouseX
    Int32 mouseY
    Variable xmin
    Variable xmax
    Variable ymin
    Variable ymax
    Variable zmin
    Variable zmax
    Variable eulerA
    Variable eulerB
    Variable eulerC
    Variable wheelDx
    Variable wheelDy
EndStructure
```

WMListboxAction

See **ListBox** for further explanation of WMListboxAction.

```
Structure WMListboxAction
    char ctrlName[32]   // Control name
    char win[200]       // Host window or subwindow name
    Rect winRect        // Local coordinates of host window
    Rect ctrlRect       // Enclosing rectangle of the control
    Point mouseLoc      // Mouse location
    Int32 eventCode     // -1: Control being killed
                        //  1: Mouse down
                        //  2: Mouse up
                        //  3: Double click
                        //  4: Cell selection (mouse or arrow keys)
                        //  5: Cell selection plus Shift key
                        //  6: Begin edit
                        //  7: Finish edit
                        //  8: Vertical scroll. See ListBox for details.
                        //  9: Horizontal scroll
                        // 10: Top row set or first column set
                        // 11: Column divider resized
                        // 12: Keystroke, char code is in row field
    Int32 eventMod      // See Control Structure eventMod Field on page III-385
    String userData     // Primary unnamed user data
    Int32 blockReentry  // See Control Structure blockReentry Field on page III-386
    Int32 eventCode2    // Obsolete
    Int32 row           // Selection row. See ListBox for details.
    Int32 col           // Selection column. See ListBox for details.
    WAVE/T listWave     // List wave specified by ListBox command
    WAVE selWave        // Selection wave specified by ListBox command
    WAVE colorWave      // Color wave specified by ListBox command
```

```
    WAVE/T titleWave    // Title wave specified by ListBox command
EndStructure
```

WMMarkerHookStruct

See **Custom Marker Hook Functions** on page IV-274 for further explanation of WMMarkerHookStruct.

```
Structure WMMarkerHookStruct
    Int32 usage         // 0= normal draw, 1= legend draw
    Int32 marker        // Marker number minus start
    float x, y          // Location of desired center of marker
    float size          // Half width/height of marker
    Int32 opaque        // 1 if marker should be opaque
    float penThick      // Stroke width
    RGBColor mrkRGB     // Fill color
    RGBColor eraseRGB   // Background color
    RGBColor penRGB     // Stroke color
EndStructure
```

WMPopupAction

See **PopupMenu** for further explanation of WMPopupAction.

```
Structure WMPopupAction
    char ctrlName[32]   // Control name
    char win[200]       // Host window or subwindow name
    Rect winRect        // Local coordinates of host window
    Rect ctrlRect       // Enclosing rectangle of the control
    Point mouseLoc      // Mouse location
    Int32 eventCode     // -1: Control being killed
                        //  2: Mouse up
    Int32 eventMod      // See Control Structure eventMod Field on page III-385
    String userData     // Primary unnamed user data
    Int32 blockReentry  // See Control Structure blockReentry Field on page III-386
    Int32 popNum        // Item number currently selected (1-based)
    char popStr[400]    // Contents of current popup item
EndStructure
```

WMSetVariableAction

See **SetVariable** for further explanation of WMSetVariableAction.

```
Structure WMSetVariableAction
    char ctrlName[32]   // Control name
    char win[200]       // Host window or subwindow name
    Rect winRect        // Local coordinates of host window
    Rect ctrlRect       // Enclosing rectangle of the control
    Point mouseLoc      // Mouse location
    Int32 eventCode     // -1: Control being killed
                        //  1: Mouse up
                        //  2: Enter key
                        //  3: Live update
                        //  4: Scroll wheel up if increment is zero
                        //  5: Scroll wheel down if increment is zero
                        //  6: Value changed by dependency update
    Int32 eventMod      // See Control Structure eventMod Field on page III-385
    String userData     // Primary unnamed user data
    Int32 blockReentry  // See Control Structure blockReentry Field on page III-386
    Int32 isStr         // TRUE for a string variable
    Variable dval       // Numeric value of variable
    char sval[400]      // Value of variable as a string
    char vName[106]     // Name of variable or wave
    WAVE svWave         // Valid if using wave
    Int32 rowIndex      // Row index for a wave if rowLabel is empty
    char rowLabel[32]   // Wave row label
    Int32 colIndex      // Column index for a wave if colLabel is empty
    char colLabel[32]   // Wave column label
EndStructure
```

WMSliderAction

See **Slider** for further explanation of WMSliderAction.

```
Structure WMSliderAction
    char ctrlName[32]   // Control name
```

```
    char win[200]       // Host window or subwindow name
    Rect winRect        // Local coordinates of host window
    Rect ctrlRect       // Enclosing rectangle of the control
    Point mouseLoc      // Mouse location
    Int32 eventCode     // -1 if control about to be killed
                        // or bit field:
                        // Bit 0: Value set
                        // Bit 1: Mouse down
                        // Bit 2: Mouse up
                        // Bit 3: Mouse moved
    Int32 eventMod      // See Control Structure eventMod Field on page III-385
    String userData     // Primary unnamed user data
    Int32 blockReentry  // See Control Structure blockReentry Field on page III-386
    Variable curval     // Value of slider
EndStructure
```

WMTabControlAction

See **TabControl** for further explanation of WMTabControlAction.

```
Structure WMTabControlAction
    char ctrlName[32]   // Control name
    char win[200]       // Host window or subwindow name
    Rect winRect        // Local coordinates of host window
    Rect ctrlRect       // Enclosing rectangle of the control
    Point mouseLoc      // Mouse location
    Int32 eventCode     // -1: Control being killed
                        //  2: Mouse up
    Int32 eventMod      // See Control Structure eventMod Field on page III-385
    String userData     // Primary unnamed user data
    Int32 blockReentry  // See Control Structure blockReentry Field on page III-386
    Int32 tab           // Tab number
EndStructure
```

WMWinHookStruct

See **Named Window Hook Functions** on page IV-265 for further explanation of WMWinHookStruct.

```
Structure WMWinHookStruct
    char winName[200]   // Host window or subwindow name
    Rect winRect        // Local coordinates of the affected (sub)window
    Point mouseLoc      // Mouse location
    Variable ticks      // Tick count when event happened
    Int32 eventCode     // See Named Window Hook Events on page IV-265
    char eventName[32]  // See Named Window Hook Events on page IV-265
    Int32 eventMod      // See Control Structure eventMod Field on page III-385
    char menuName[256]  // Name of the menu item as for SetIgorMenuMode
    char menuItem[256]  // Text of the menu item as for SetIgorMenuMode
    char traceName[32]  // See Named Window Hook Functions on page IV-265
    char cursorName[2]  // Cursor name A through J
    Variable pointNumber// See Named Window Hook Functions on page IV-265
    Variable yPointNumber  // See Named Window Hook Functions
    Int32 isFree        // 1 if the cursor is not attached to anything
    Int32 keycode       // ASCII value of key struck
    char oldWinName[32] // Simple name of the window or subwindow
    Int32 doSetCursor   // Set TRUE to change cursor to cursorCode
    Int32 cursorCode    // Standard cursor as set by hookcursor=number
    Variable wheelDx    // Vertical lines to scroll
    Variable wheelDy    // Horizontal lines to scroll
EndStructure
```

wnoise

wnoise(*shape, scale*)

The wnoise function returns a pseudo-random value from the two-parameter Weibull distribution characterized by the *shape* and *scale,* the respective *gamma* and *alpha* parameters. The two-parameter Weibull probability distribution function is

$$f(x;\alpha,\gamma) = \frac{\gamma}{\alpha} x^{\gamma-1} \exp\left[-\frac{1}{\alpha}x^{\gamma}\right] \quad \begin{matrix} x \geq 0 \\ \alpha > 0 \\ \gamma > 0 \end{matrix}$$

The mean of the Weibull distribution is

$$\alpha^{\frac{1}{\gamma}}\Gamma\left(1+\frac{1}{\gamma}\right),$$

and the variance is

$$\alpha^{\frac{2}{\gamma}}\Gamma\left(1+\frac{2}{\gamma}\right) - \alpha^{\frac{2}{\gamma}}\left[\Gamma\left(1+\frac{1}{\gamma}\right)\right]^2.$$

Note that this definition of the PDF uses different scaling than the one used in StatsWeibullPDF. To match the scaling of StatsWeibullPDF multiply the result from Wnoise by the factor scale^(1-1/shape).

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-332.

Chapter III-12, **Statistics** for a function and operation overview.

x

x

The x function returns the scaled row index for the current point of the destination wave in a wave assignment statement. This is the same as the X value if the destination wave is a vector (1D wave).

Details

Outside of a wave assignment statement, x acts like a normal variable. That is, you can assign a value to it and use it in an expression.

See Also

The **p** function and **Waveform Arithmetic and Assignments** on page II-93.

x2pnt

x2pnt(*waveName, x1*)

The x2pnt function returns the point number on the wave whose X value is closest to *x1*.

Details

There are no equivalent functions for multidimensional waves. To calculate this information for other dimensions, use this expression:

```
(ScaledDimPos - DimOffset(waveName, dim))/DimDelta(waveName,dim)
```

This expression calculates the number of an element in the dimension *dim* (p, q, r, or s). *ScaledDimPos* is the scaled position in that dimension (x, y, z, or t). *dim* is 0 for rows, 1 for columns, 2 for layers or 3 for chunks. Setting *dim* =0 is equivalent to using x2pnt. The point number includes a fractional part; you may wish to use **ceil**, **round**, **trunc**, or **floor** to calculate an integer point number.

See Also
The functions **DimDelta**, and **DimOffset**.

For an explanation of waves and X scaling, see **Changing Dimension and Data Scaling** on page II-83.

xcsr

xcsr(*cursorName* [, *graphNameStr*])
The xcsr function returns the X value of the point which the named cursor (A through J) is on in the top or named graph.

Parameters
cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details
The result is derived from the wave that the cursor is on, not from the X axis of the graph. If the wave is displayed as an XY pair, the X axis and the wave's X scaling will usually be different.

See Also
The **Cursor** operation and the **CsrInfo**, **hcsr**, **pcsr**, **qcsr**, **vcsr**, and **zcsr** functions.

XWaveName

XWaveName(*graphNameStr, waveNameStr*)
The XWaveName function returns a string containing the name of the wave supplying the X coordinates against which the named wave is displayed in the named graph window or subwindow.

Parameters
graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details
XWaveName returns an empty string ("") if the wave is not plotted versus an X wave.

For most uses, we recommend that you use **XWaveRefFromTrace** instead of WaveName. XWaveName returns a string containing the wave name only, with no data folder path qualifying it. Thus, you may get erroneous results if the X wave referred to in the graph has the same name as a different wave in the current data folder. Likewise, if the named wave resides in a folder that is not the current data folder, you will not be able to refer to the named wave.

graphNameStr and *waveNameStr* are strings, *not* names.

Examples
```
Display ywave vs xwave              // XY graph
Print XWaveName("","ywave")         // prints xwave
```

XWaveRefFromTrace

XWaveRefFromTrace(*graphNameStr, traceNameStr*)
The XWaveRefFromTrace function returns a wave reference to the wave supplying the X coordinates against which the named trace is displayed in the named graph window or subwindow.

Parameters
graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Details
XWaveRefFromTrace returns a null reference (see **WaveExists**) if the wave is not plotted versus an X wave.

graphNameStr and *traceNameStr* are strings, not names.

Examples

```
Display ywave vs xwave                    // XY graph
Print XWaveRefFromTrace("","ywave")[50] // prints value of xwave at point 50
```

See Also

For other commands related to waves and traces: **WaveRefIndexed**, **TraceNameToWaveRef**, **TraceNameList**, **CsrWaveRef**, and **CsrXWaveRef**.

For a description of traces: **ModifyGraph**.

For a discussion of contour traces see **All About Contour Traces** on page II-328.

For commands referencing other waves in a graph: **ImageNameList**, **ImageNameToWaveRef**, **ContourNameList**, and **ContourNameToWaveRef**.

For a discussion of wave references, see **Wave Reference Functions** on page IV-173.

y

y

The y function returns the Y value for the current column of the destination wave when used in a multidimensional wave assignment statement. Y is the scaled column index whereas **q** is the column index itself.

Details

Unlike **x**, outside of a wave assignment statement, y does not act like a normal variable.

See Also

x, **z**, and **t** functions for other dimensions.

p, **q**, **r**, and **s** functions for the scaled indices.

z

z

The z function returns the Z value for the current layer of the destination wave when used in a multidimensional wave assignment statement. z is the scaled layer index whereas **r** is the layer index itself.

Details

Unlike **x**, outside of a wave assignment statement, z does not act like a normal variable.

See Also

x, **y**, and **t** functions for other dimensions.

p, **q**, **r**, and **s** functions for the scaled indices.

zcsr

zcsr(*cursorName* [, *graphNameStr*])

The zcsr function returns a Z value when the specified cursor is on a contour, image, or waterfall plot. Otherwise, it returns NaN.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-95 for details on forming the window hierarchy.

Examples

```
Print zcsr(A)              // not zcsr("A")
Print zcsr(A,"Graph0")     // specifies the graph
```

See Also

The **Cursor** operation and the **CsrInfo**, **hcsr**, **pcsr**, **qcsr**, **vcsr**, and **xcsr** functions.

ZernikeR

ZernikeR(*n*,*m*,*r*)

The ZernikeR function returns the Zernike radial polynomials of degree *n* that contains no power of *r* that is less than *m*. Here *m* is even or odd according to whether *n* is even or odd, and *r* is in the range 0 to 1.

Note that the full circle polynomials are complex. For any angle *t* (theta), they are given by: `ZernikeR(`*n*`,`*m*`,`*r*`)*exp(i`*mt*`)`.

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z