Getting Started with IDL

IDL Version 7.0
November 2007 Edition

1107IDL70GS

Contents

Getting Started with IDL

Chapter 1
The Power of IDL

IDL, the *Interactive Data Language*, is the ideal software for data analysis, visualization, and cross-platform application development. IDL integrates a powerful, array-oriented language with numerous mathematical analysis and graphical display techniques, thus giving you incredible flexibility.

Interactive Analysis

A few lines of IDL can do the job of hundreds of lines of Java, FORTRAN, or C — without losing flexibility or performance. Using IDL, tasks that require days or weeks of programming with traditional languages can be accomplished in hours. Explore data interactively using IDL commands and then create complete applications by writing IDL programs.
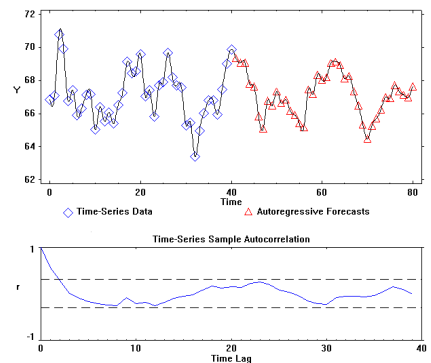
Data Analysis and Signal Processing
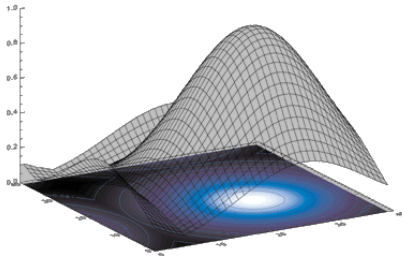
Use IDL to read data in a wide variety of formats — from simple ASCII to structured data formats like HDF, CDF, and NetCDF to modern image formats such as JPEG2000. Fit irregularly-sampled data to a regular grid, and use IDL's signal processing routines to extract and analyze the signals contained therein, using techniques from traditional filtering and transform operations to statistical methods such as prediction analysis. Use IDL's powerful graphical visualization tools to view the results of your analysis in two- and three-dimensional visualizations.

Image Processing and Display

IDL reads most common image files with a single command. Once you've imported image data into IDL, use a wide variety of image processing techniques to filter out noise, expose anomalies, and highlight true data characteristics. Create publication-quality, fully-annotated image displays.
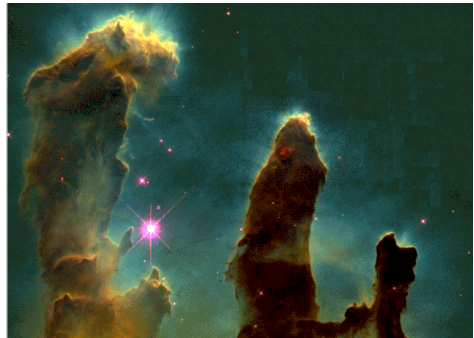
Combine Data and Maps
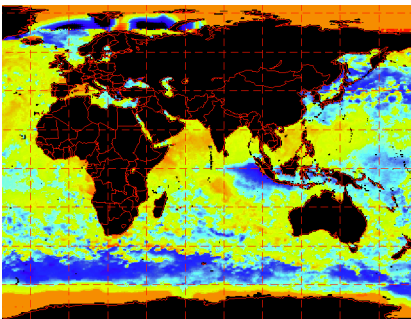
Easily overlay sampled data on a map display to extract geographical information from your data. Modify the map projection and coordinates to inspect any location on the globe.

Rapid Application Development
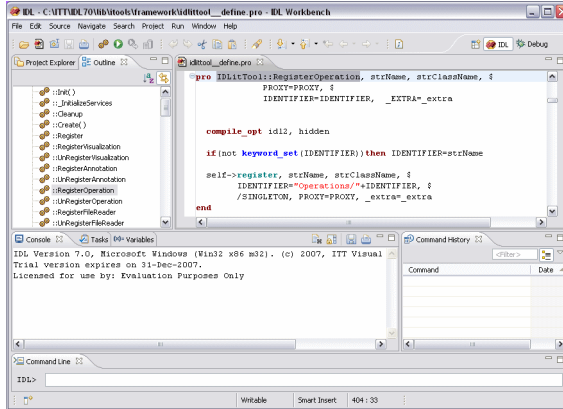
Use the powerful code development and debugging tools of the IDL Workbench to rapidly create complex applications in the IDL language. Distribute your code to other IDL users, or provide a compiled version that runs in the freely-available IDL Virtual Machine.

User Interface Toolkit

IDL's user interface toolkit allows you to quickly develop graphical user interfaces entirely in IDL. Create simple interfaces with only a few lines of code using IDL's built-in widgets, or use the iTools framework to build complex interactive applications in a fraction of the time you would spend creating a similar interface in other languages.
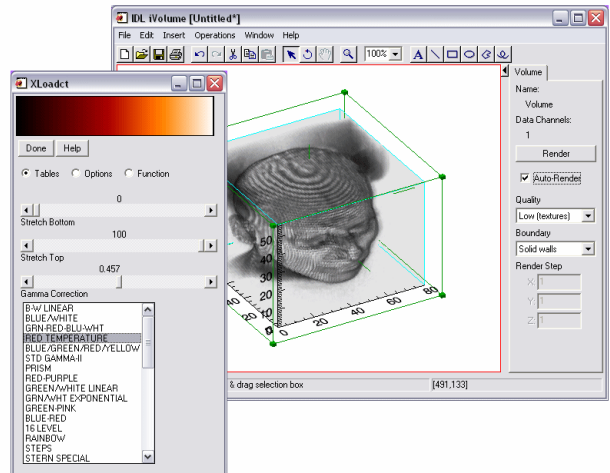
Using this Manual

The chapters included in this manual provide a "hands-on" way to learn basic IDL concepts and techniques. *Getting Started with IDL* demonstrates a number of common IDL applications; each section introduces basic IDL concepts and highlights some of the commonly-used IDL commands.

Each chapter functions similarly to a tutorial and is a demonstration of a particular IDL feature. It is recommended that you walk through each short, interactive chapter to preserve continuity, since many commands rely upon previous commands. Each chapter assumes the most basic level of IDL experience.

A Note on the Example Code

You don't have to read all of the descriptive passages that accompany each chapter. Simply enter the IDL commands shown in `courier` type at the IDL Command Line (the "IDL>" prompt) and observe the results. Unless otherwise noted, each line shown is a complete IDL command (press RETURN after typing each command). If you want more information about a specific command, you can read the explanations or consult IDL's online help system by selecting **Help → Help Contents** in the IDL Workbench.

Tip ———————————————————————————————————
The dollar sign ($) at the end of a line is the IDL continuation character. It allows you to enter long IDL commands as multiple lines.

───────────────————————————————————————————————————

A Note on the Graphics Displays

Many of the examples in this manual use IDL's iTools, which provide an interactive graphical interface to visualizations such as plots or images. The iTools use IDL's *Object graphics* system, and will automatically adjust to display correctly on any computer running IDL.

Other examples use IDL's *Direct graphics* routines (which have names like PLOT, CONTOUR, or TV). The Direct graphics system is simpler to use in some situations, but lacks some of the display management features of the Object graphics system. As a result, on most newer systems you will want to tell IDL to use a maximum of 256 colors in its graphics displays by entering the following command at the IDL command prompt:

```
DEVICE, RETAIN=2, DECOMPOSED=0
```

Other Resources

This manual provides examples that will give you a glimpse of the many ways IDL can speed your data analysis, visualization, and cross-platform development tasks. The following are some additional resources that can help you continue learning about IDL.

IDL Documentation

The IDL documentation set is installed along with IDL in hypertext format. To view the documentation, enter "?" at the IDL> prompt or select **Help Contents** from the **Help** menu of the IDL Workbench.

IDL's online help system is fully hyperlinked and indexed, and includes a powerful full-text searching mechanism. See "Getting Help" on page 33 and the Using IDL Help topic in the IDL Online Help system for information on using the help system itself.

In addition, Adobe Portable Document Format (PDF) versions of most books in the IDL documentation set are included in the info/docs directory of the IDL distribution disk.

ITT Visual Information Solutions Web Site

The ITT Visual Information Solutions web site (www.ittvis.com) provides additional information about IDL and other ITT Visual Information Solutions products. On our web site you will find:

- User forums, which allow you to correspond directly with other users of IDL to discuss problems, solutions, and techniques.

- The ITT Visual Information Solutions Code Contribution Library, which allows you to share source code, images, data, and more with other IDL users.

- Tips and Tricks for using IDL.

- Technical Support resources, including a database of common IDL questions and answers.

- Stories about customers' innovative uses of IDL.

IDL Workbench Welcome Page

Numerous local and web-based resources are available within the IDL Workbench interface. Select **Welcome** from the **Help** menu to display the Welcome screen, then click on **What's New** to gain access to up-to-date information from ITT Visual Information Solutions, including news and announcements of new downloadable

modules for IDL. Click on **Web Resources** to get quick access to developer news items, user forums, and other network resources. Click on **Tutorials** to access short lessons describing how to accomplish common tasks in IDL and the IDL Workbench.
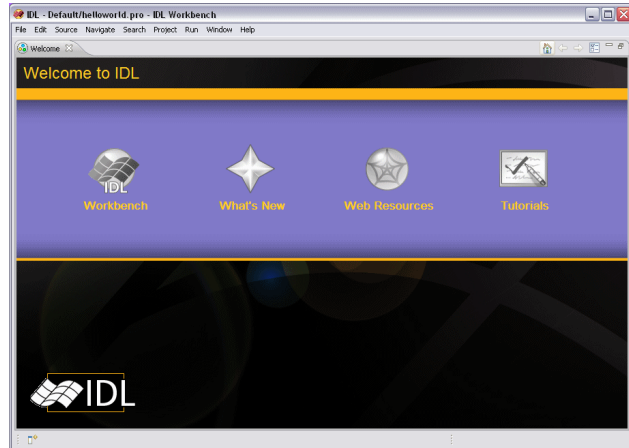
IDL Newsgroup

The IDL newsgroup is an independent forum for IDL users to discuss problems and solutions in IDL. Point your news reading software at the `comp.lang.idl-pvwave` USENET newsgroup, or use a web-based reader such as

```
http://groups.google.com/group/comp.lang.idl-pvwave
```

to read and join in the discussion.

Chapter 2
Super Quick Start

If you'd like to begin experimenting with IDL right away, before reading any more, try the following things:

Open the IDL Workbench

The IDL Workbench is a graphical interface and code development environment for IDL. To start the IDL Workbench:

- On Windows platforms, use the **Start** menu to select **IDL** from the **IDL 7.0** program group.

- On Macintosh platforms, click on the **IDL Workbench** icon in the **IDL 7.0** folder or launch an X11 terminal window and type idlde at the prompt.

- On UNIX and Linux systems, type idlde at the shell prompt.

The IDL Workbench is described in more detail in "The IDL Workbench" on page 19.

Display an Image

To quickly display an image using IDL:

1. Enter `iimage` at the IDL command prompt. The iImage window appears.

2. Select **File** → **Open** in the iImage window and browse to the `examples/data` subdirectory of your IDL installation.

3. Select the file `rose.jpg` and click **Open**. The image is displayed in the iImage window.

IDL's image display and image processing facilities are described in more detail in "Images" on page 73.
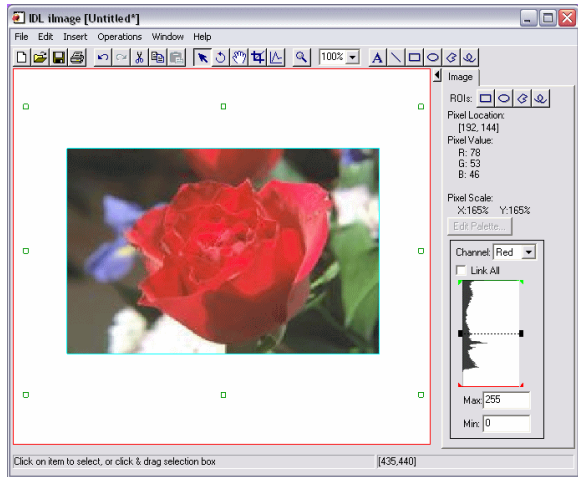
Create a 2-D Plot

To further analyze the image displayed in the previous section, you might want to create a plot showing the values of selected pixels plotted against their positions in the image (a *line profile*). You can do this in two ways:

- Click the **Line Profile** button in the iImage tool and draw a line interactively across the image. Three line profiles — one each for the red, blue, and green channels of the image, are displayed in an iPlot window.

- Select a line in the image array numerically. To do this, we'll have to create an image array directly in IDL (not using the iImage tool):

```
file = FILEPATH('rose.jpg', SUBDIR=['examples', 'data'])
READ_JPEG, file, rose
```

These two commands locate the `rose.jpg` file and create an IDL variable named `rose` that contains the image data. Use the Variables view in the IDL Workbench to inspect the `rose` variable, or type

```
HELP, rose
```

at the IDL command prompt. This shows us that the `rose` variable is a `[3, 227, 149]` byte array consisting of red, green, and blue image planes,

each of which is a 227 x 149 pixel array. To extract a single vector of data, we'll use IDL's array indexing syntax:

```
rose_slice = REFORM(rose[0,200,*])
```

This tells IDL to create a new variable named `rose_slice` that contains a vector consisting of the 149 elements found in column 200 of the red image plane (image plane 0, which is the first image plane in the array).

Finally, enter

```
iplot, rose_slice
```

at the IDL command prompt. The iPlot window displays the selected line profile.
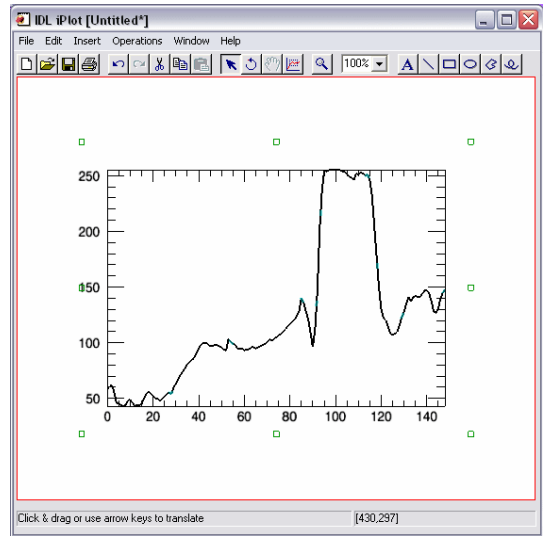
Clearly, creating a line profile interactively using the iImage tool is quicker in this example, but numerically selecting a vector from within an array is more precise and can be accomplished without mouse interaction.

Tip ───

You can quickly modify the appearance of your plot using the iTools property sheet controls. Simply double-click on an item (the plot line, for example) to display the property sheet. Change the selected options and see the results immediately.

───

IDL provides many tools for creating, annotating, and modifying two-dimensional plots. See "Line Plots" on page 39 for additional details.

Overlay an Image on a Map

If your data is associated with geographic coordinates, you can easily overlay your data on a map using any of several map projections available in IDL:

1. Enter `imap` at the IDL command prompt. The iMap window appears.

2. Select **File → Open** in the iMap window and browse to the `examples/data` subdirectory of your IDL installation.

3. Select the file avhrr.png and click **Open**. The iMap Register Image wizard appears, allowing you to specify how the pixels in the image map to geographic coordinates.

4. Click **Next**, then **Finish** in the **Register Image** wizard to accept the default values.

5. On the **Map** tab, click **Edit Projection** and select Mollweide in the Projection field and click **OK**.

6. Select **Insert → Map → Countries (low res)** to overlay country boundaries.
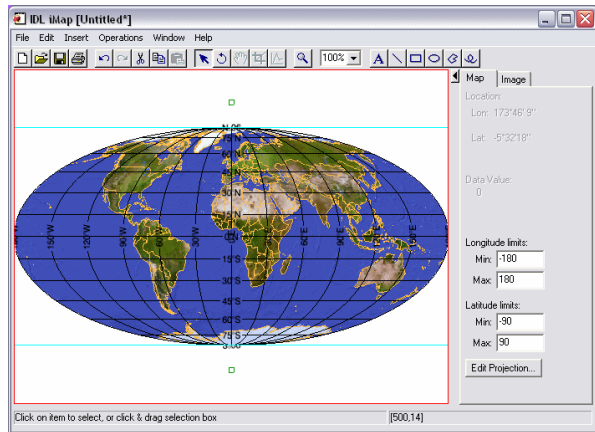
See "Mapping" on page 93 for more on working with maps in IDL.

Create a Simple IDL Program

Creating and running a program in IDL can be as simple as this:

1. Create a new IDL source file in the IDL Workbench by clicking the **New IDL Source file** toolbar button 📄 .

2. Enter the following text in the editor window:

```
PRO helloWorld

PRINT, 'Hello, World!'

END
```

3. Select **File → Save** and then click **OK** in the **Save As** dialog that appears, accepting the default filename and location. (This saves your code in a file named helloworld.pro in your default IDL project directory.)

4. Select **Run → Run helloworld** or press **F8**. Your routine is compiled and the string Hello, World! is printed in the Console view.

Tip ───

You could also run your program by entering `helloworld` at the IDL command prompt, or from within another IDL program.

───

See "Programming in IDL" on page 109 for more on creating programs in the IDL language.

Get Help

IDL provides several ways to get help, depending on what sort of assistance you require:

- To launch the IDL online help system, which contains both reference and user documentation for IDL, select **Help Contents** from the **Help** menu or type "?" at the `IDL>` prompt. See the *Using IDL Help* topic in the IDL Online Help for more information.

- In the Editor view of the IDL Workbench, hover the mouse pointer over name of a procedure or function. After a second, text describing the syntax of the routine appears. For example, if you type `a = sin` in an editor window and hover the mouse pointer, you'll see something like this:


```
a = sin(
    Syntax:
    Result = SIN(X, TPOOL_MAX_ELTS = null, TPOOL_MIN_ELTS = null, TPOOL_NOTHREAD = null)
                                                                    Press 'F2' for focus
```

This display (known as "hover help") shows you the syntax for the SIN routine. Note that IDL keyword values are always shown as "null" in the hover help display.

- Click on a routine name within the editor window. Press **F1** (Windows), **Shift+F1** (Solaris/Linux), or **Help** (Macintosh). The full IDL help entry for the routine is displayed.

- Begin a new line in the editor and type the following:

```
file = dialog
```

Without moving the cursor from the end of the word "dialog", press **Ctrl+space**. The following windows appear:


```
file = dialog
END         DIALOG_DBCONNECT()        Syntax:
            DIALOG_MESSAGE()          Result = DIALOG_PICKFILE(DEFAULT_EXTENSION =
            DIALOG_PICKFILE()         null, DIALOG_PARENT = null, DIRECTORY = null,
            DIALOG_PRINTERSETUP()     DISPLAY_NAME = null, FILE = null, FILTER = null,
            DIALOG_PRINTJOB()         FIX_FILTER = null, GET_PATH = null, GROUP = null,
            DIALOG_READ_IMAGE()       MULTIPLE_FILES = null, MUST_EXIST = null,
                                      OVERWRITE_PROMPT = null, PATH = null, READ =
```

This display is known as "content assist." Use the arrow keys to select
DIALOG_PICKFILE() from the left-hand list, noting that the syntax for the
DIALOG_PICKFILE function is displayed. Press **Enter** and the function
name is inserted into the editor window. Press **Ctrl+space** again to see the list
of keywords for the routine, followed by functions whose values could be
entered as arguments.

Note

Content assist is also available in the IDL Command Line view.

Chapter 3
The IDL Workbench

This chapter introduces the IDL Workbench and its capabilities.

The IDL Workbench

IDL 7.0 features a graphical front-end called the *IDL Workbench* that provides sophisticated code management, development, and debugging tools. The Workbench is created using the *Eclipse framework* — an extensible cross-platform environment that appears as a native application on all platforms. The IDL Workbench looks and behaves like a Windows application on Windows machines, like a Macintosh application on Macintosh machines, and like a Linux or Solaris application on those systems.

Note that the Eclipse features that make up the IDL Workbench are just a front-end: IDL's powerful computational engine is still used to analyze and display your data.

For additional information on working with IDL and the IDL Workbench, refer to the *Using IDL* manual, located in the IDL Online Help.

Starting the IDL Workbench

Note

For information on installing and licensing IDL, see the IDL installation instructions for your platform.

To start the IDL Workbench, follow the instructions according to your Operating System:

Operating System	IDL Workbench Instructions
Windows	Select **Start** → **IDL** *<version>* → **IDL**
Solaris/Linux	At the shell prompt, enter `idlde`
MacOS X	In your IDL installation folder, double-click on the IDL Workbench icon OR At the X11 Terminal window shell prompt, enter `idlde`

Command Line Options

You can alter some IDL behaviors by supplying command-line switches along with the command used to invoke IDL. IDL's options are described in detail in *Command Line Options for IDL Startup* in the IDL Online help, but the following are among the most useful.

-batch

Syntax: idlde -batch *filename*

Specifies that *filename* should be executed in non-interactive "batch" mode. Note that *filename* should specify the full path to the batch file.

-e

Syntax: idlde -e *IDL_statement*

Specifies a single IDL statement to be executed. Once the statement has executed, IDL waits for any widget applications to exit, and then IDL itself exits. Only the last -e switch on the command line is honored.

Note ───

If the IDL statement includes spaces, it must be enclosed in quote marks.

───

-nl

Syntax: idlde -nl *locale*

Selects a different locale (language). *Locale* is a two-letter abbreviation, such as en (English), fr (French), it (Italian), or ja (Japanese).

Eclipse and the IDL Workbench are both internationalized, but do not share the same language list. If a language is chosen that both platforms do not support, there will be translation mismatches in the UI (Eclipse portions of the UI will be documented in one language, and IDL Workbench portions documented in another).

Starting IDL in Command Line Mode

In *command-line mode*, IDL uses a text-only interface and sends output to your terminal screen or shell window. Graphics are displayed in IDL graphics windows.

Note
Command-line mode is only available on UNIX-based platforms (Solaris, Linux, and Macintosh).

To start IDL in command-line mode, follow the instructions according to your Operating System:

Operating System	Command-line Mode Instructions
Solaris/Linux	At the shell prompt, enter `idl`.
MacOS X	In your IDL installation folder, double-click on the IDL icon OR At the X11 Terminal window shell prompt, enter `idl`.

For more information about using IDL in command-line mode, see the *Launching IDL* topic in the IDL Online Help.

Tip
The command line options described above are also useful in command-line mode.

IDL Workbench Tour
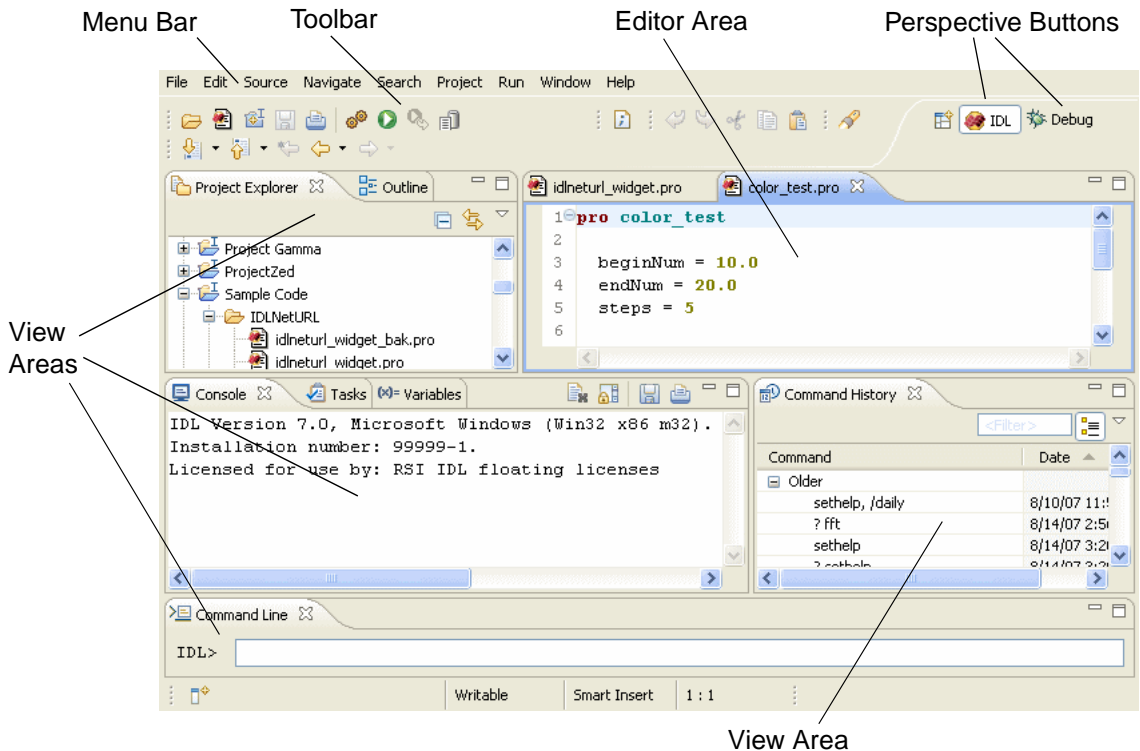
Figure 3-1: The IDL Workbench

The following sections discuss the components of the IDL Workbench.

Menu Bar

The menu bar, located at the top of the IDL Workbench window, allows you to control various Workbench features.

You can display menu commands for each menu using the following methods:

- Clicking the menu on the menu bar.

- Pressing **Alt** (**Option** on the Macintosh) plus the underlined letter in the menu's title. For example, to display the **File** menu in Windows, press **Alt**+**F**.
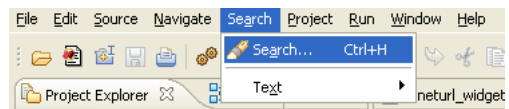
You can select or execute a menu command using the following methods:

- Clicking the item in the menu.

- Pressing **Alt** (**Option** on the Macintosh) plus the underlined letter in the menu's title, and then pressing the letter underlined in the menu item. For example, to select the menu item **Edit → Undo** in Windows, press **Alt**+**E**+**U**.

Note ───

Many menu items have alternate keyboard shortcuts. If a command has a keyboard shortcut, it is displayed to the right of the menu item.
───

Toolbar

The IDL Workbench toolbar buttons provide a shortcut to execute the most common tasks found in the main menu. When you position the mouse pointer over a toolbar button, a brief command description is displayed next to it (along with the keyboard shortcut, if applicable).

Views

Views are movable windows inside the IDL Workbench that display data, do analyses, and allow you to interact with the command line interpreter and compiled programs. For example, the Console view displays output from the command line and compiled programs.

Moving Views

A view might appear by itself or stacked with other views in a tabbed window. You can change the layout of a perspective by opening and closing views and by docking them in different positions in the Workbench window.

There are a number of options for moving views:

- You can move a view to another location by clicking on the view's tab and dragging it to another spot on the Workbench. You can move a view so that it occupies its own window, or you can move a view into a group of existing views (as a new tab).

- A view tab's context menu contains the **Detached** option, which allows you to detach the view into its own Workbench-independent window. The context menu's **Move → View** option lets you move a single view, and the **Move → Tab Group** option lets you move a group of views (as a collection of tabs).

Maximizing Views

You can maximize a view within the workbench interface by double-clicking on the view's tab. All other views are automatically minimized. Double-click on the view's tab again to restore it and all other views to the original size.

The following sections explain all the views that appear by default when you first run the IDL Workbench, but there are more views available. To display one of the other views, select it from the **Window → Show View** menu.

Project Explorer

The Project Explorer view provides a hierarchical view of the resources in the IDL Workbench. From here, you can open files for editing or select resources for operations such as exporting.
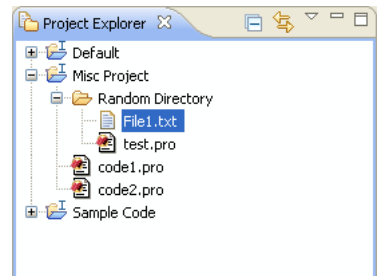
Outline

The Outline view displays a structural outline of the file that is currently open in the editor area. The Outline view shown at right displays a list of the procedures contained in a `.pro` file.
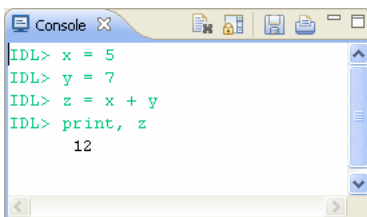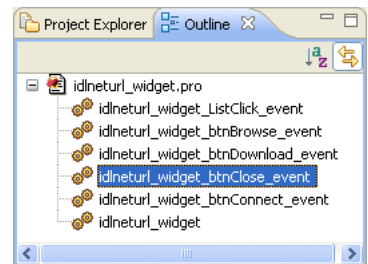
Console

The Console view displays output from both the IDL command line and compiled IDL programs. This output includes:

• Commands entered at the IDL command line

• IDL command output and errors

• IDL program output

• Compilation information and errors

Tasks

The Tasks view displays tasks inserted into code files. A task is a short text string explaining an action to be completed, in relation to a particular line of code. When you are done with a task, you can remove it or mark it as completed.
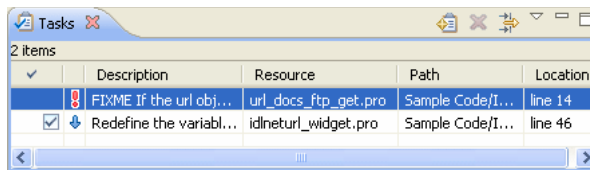
Variables

The Variables view displays the values of variables in the current execution scope. Used in conjunction with the Debug view, the Variables view allows you to see variable values in the routine in which execution halted. If the calling context changes during execution—as when stepping into a procedure or function—the variable list changes to reflect the current context.
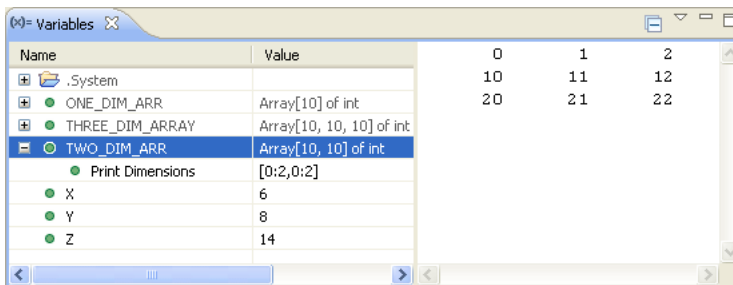
For more information on using the Variables view while debugging, see "Viewing Variable Values" on page 30.

Command History

The Command History view displays a list of commands entered at the IDL command line. You can copy one or more previous commands and paste or drag them to an editor or to the IDL command line. You can also double-click a single command to execute it immediately.
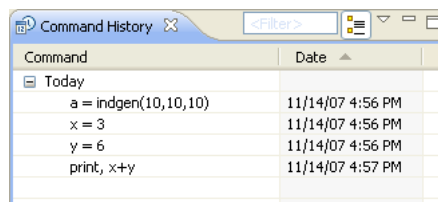
Command Line

The Command Line view displays the IDL command line, which is used to execute IDL statements, compile and launch applications, and create main-level programs.
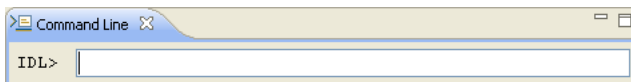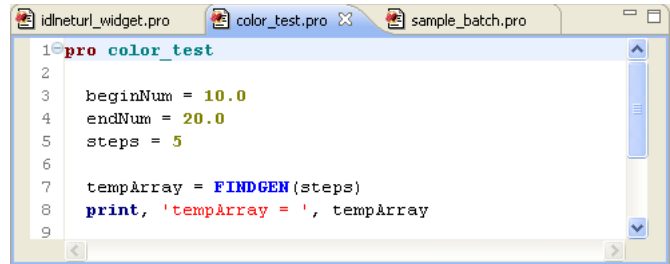
Files

IDL source files have the ".`pro`" extension. The IDL Workbench can host many different types of editors, but .`pro` files are edited using the IDL-supplied .`pro` file editor.

Editors

The editor area of the IDL Workbench contains the file editor windows. Any number of editors can be open at once, but only one can be active at a time. By default, editors are displayed as tabs inside the editor area. An asterisk (*) indicates that an editor has unsaved changes.

Although you will mostly use the IDL-supplied editor to work with .`pro` files, the IDL Workbench supports many types of popular editors.

Perspectives

A *perspective* defines the initial set and layout of views in the Workbench window. Each perspective provides a set of functionality aimed at accomplishing a specific type of task. For example, the IDL perspective combines views commonly used to edit IDL .`pro` source files, while the Debug perspective contains the views used to debug IDL programs. As you work in the Workbench, you will probably switch perspectives frequently.
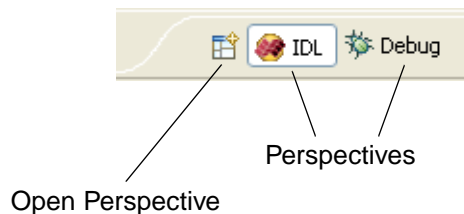
The **Perspectives** tab is located at the top-right of the IDL Workbench. To switch to a different perspective, click on one of the tab's perspective buttons, or click **Open Perspective** and select a perspective from the list.

Projects

In IDL, a *project* is a directory that contains source code files and other resources (data, image files, documentation, and so on). Projects are especially useful as logical containers for related source code and resource files. A project is saved within the IDL workspace (which is discussed later in the chapter).

While you can build and run individual `.pro` files, you can also build and run an entire project, as well as configure how the project is built.

Workspaces

A *workspace* is a directory that contains project directories, metadata about the contained projects, and information about the state of the IDL Workbench. Each workspace "remembers" the arrangement of the IDL Workbench views and perspectives. You can have as many different workspaces as you like, but only one workspace can be loaded at once.

You can select any location and directory name for your workspace. By default, the workspace directory is named `IDLWorkspace` and is located in your home directory (as defined by the `$HOME` environment variable on UNIX-like platforms and in `Documents and Settings\username` or `Users\username` on Windows platforms).

Compiling and Running an IDL Program

To compile and run an IDL program using the IDL Workbench:

1. Open the file in the IDL editor:

    *   Use the Project Explorer view to select a file located in one of your projects. Double-click on the file to open it in an editor.

    *   Alternately, use **File → Open File** to select a file from the file system. For example, you could open the file examples\demo\demosrc\d_uscensus.pro from the IDL installation directory.

2. Compile the file by clicking the **Compile** button   on the toolbar, or by selecting **Run → Compile** *filename*, where *filename* is the name of the file opened in the IDL editor.

3. Execute the file by clicking the **Run** button   on the toolbar, or by selecting **Run → run** *filename*, where *filename* is the name of the file opened in the IDL editor.

    Note ──
    You do not need to explicitly compile your program each time you run it. Clicking the **Run** button will compile the file automatically if it has not yet been compiled. You *do* need to recompile your program if you compiled it and then made changes to the source code.
    ──
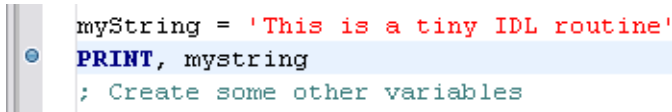
Breakpoints and Debugging

The IDL Workbench provides robust tools for finding and correcting problems in your code.

Breakpoints

To set a breakpoint, place the cursor on the line where you want the breakpoint to appear and press **Ctrl**-**Shift**-**B** or select **Run → Toggle breakpoint**. A blue dot appears in the left-hand margin of the editor window.

You can also toggle breakpoints on and off by double-clicking in the left-hand margin next to the line of code on which you want IDL to pause.

Debug Perspective

The IDL Workbench makes a distinction between editing and debugging code, and provides the ability to switch automatically to the *Debug perspective* when an error or breakpoint is encountered. The Debug perspective is a collection of the views most useful for debugging and analyzing code.

See the *Using the Debug Perspective* topic in the IDL Online Help for details.

Viewing Variable Values

When you run a routine that contains a breakpoint, IDL will halt execution when it reaches the breakpoint. When execution is halted, you can inspect the variable values in the current execution scope using the Variables view or by hovering the mouse pointer over a variable in the editor.

Stepping Through Code

When execution is halted due to a breakpoint or an error, you can execute single statements using the **Step** commands on the **Run** menu. See the *Stepping Through Code* topic in the IDL Online Help for details.

Debugging a Short Program

Let's walk through the process of debugging a short program. In this example, we'll create a program, set a breakpoint, inspect variable values, and step through the code.

1. To create a new .pro file, click the ![icon] icon or select
   File → New → IDL Source File.

   A new editor window appears.
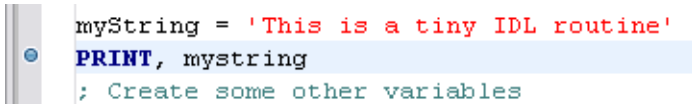
2. Enter the following text into the editor window:

   ```
   PRO tinyRoutine
   ; Create a string variable
   myString = 'This is a tiny IDL routine'
   PRINT, mystring
   ; Create some other variables
   myNumber = 4
   myResult = STRING(myNumber * !PI)
   ; Display the myResult variable
   void = DIALOG_MESSAGE('Result: '+myResult)
   END
   ```

3. To save your new .pro file, click the ![icon] icon or select **File → Save**.

4. Select "Default" as the parent folder and click **OK**, accepting the default filename (tinyroutine.pro).

5. To execute the program, click the ![icon] icon, press F8, or select
   Run → Run tinyroutine.

   The program prints a string to the Console view, displays a dialog, and ends. If you wanted to temporarily stop execution of your routine somewhere in the middle, you would set a *breakpoint*.

6. Click **OK** on the dialog to clear it.

7. Position the cursor on the words PRINT, myString in the editor window.

8. Press Ctrl-Shift-B or select **Run → Toggle breakpoint**. A blue dot appears in the left-hand margin of the editor window.

You can also toggle breakpoints on and off by double-clicking in the left-hand margin next to the line of code on which you want IDL to pause.

9. Run the `tinyroutine` program again (press F8, click the ⬤ icon, or select **Run → Run tinyroutine**).

    If this is the first time you have run a program with a breakpoint (or an error), you will see the **Confirm Perspective Switch** dialog.

10. Click **Yes** to display the IDL *Debug perspective*.

    The IDL Workbench interface is rearranged to add several new views at the top of the screen: Debug, Variables, and Breakpoints.

    Notice that the Variables view contains entries for the four variables defined in the `tinyroutine` routine, but that only the MYSTRING variable is defined. It is also instructive to examine the contents of the Debug and Console views when IDL stops at a breakpoint.

11. Press F6, click the 🔄 icon (on the toolbar in the Debug view), or select **Run → Step Over**.

    Note how the Debug, Console, Variables, and Editor views adjust as you repeatedly step through your code.

12. When you have stepped to the end of `tinyroutine` (you will see the text "`% Stepped to: $MAIN$`" in the Console view), click the IDL icon on the Perspective toolbar to return to the IDL perspective.
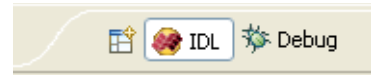
For additional information on debugging, see the *Debugging IDL Code* topic in the IDL Online Help.

Getting Help

There are several sources of user assistance in the IDL Workbench:

- Hover Help
- Content Assist
- Context-Sensitive Interface Help
- Online Help System

Hover Help

Hover Help is displayed in a pop-up window that appears when you hover the mouse cursor over the name of an IDL routine or variable. For routines, Hover Help displays the syntax documentation. For variables, Hover Help displays the current value of the variable (if execution is stopped in the routine in which the variable is defined).

Content Assist

Content Assistance is displayed in a pop-up window that appears when you place the mouse cursor in a full or partial IDL routine name and press **Ctrl**-**Space**. The Content Assist window displays a list of routine names that begin with the characters in the selected string.
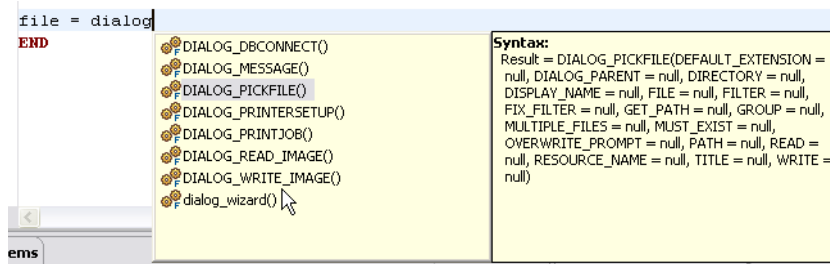
Highlighting an item from the Content Assist window displays the syntax for that routine. Selecting the item inserts it at the cursor location.

Context-Sensitive Interface Help

If you are working through a task and encounter a part of the IDL Workbench interface that you do not understand, you can summon context-sensitive help. By default, this displays the Help view and gives you some specific information about the view/editor/dialog you are using, and possibly some links to topics for further help.

Context-sensitive help can be accessed by clicking on the interface part in question and then selecting **Help → Dynamic Help** or pressing **F1** (Windows), **Shift**+**F1** (Linux and Solaris), or **Help** (Macintosh). Clicking the **?** icon in the lower left-hand corner of many IDL Workbench dialogs will also display context-sensitive help.

Online Help System

For more in-depth information, including general information, programming reference guides, and tutorials, refer to the IDL Online Help system. The Help system lets you browse, search, bookmark, and print Help documentation.

Tip ———
See the *Using IDL Help* topic in the IDL Online Help for complete information on using the IDL help system.
——

You can interact with the Workbench help system using either the Help view or a separate Help browser. The view and browser provide the same information but in different ways.

The Help View

The Help view provides help inside the Workbench. You can open the view from the main menu by selecting **Help → Dynamic Help** or **Help → Search**. The view opens showing the Related Topics or Search page, respectively. By default, typing ? followed by a search term at the IDL command line also displays help topics in the Help view. See the *Help View Interface* topic in the IDL Online Help for additional information.

The Help Browser

The Help browser provides the same content as the Help view, but in a *separate* browser application. You can open the window from the main menu by selecting **Help → Help Contents**. The first view shown in the window displays the table of contents for the product documentation. Click on one of the links to expand the navigation tree for a set of documentation. On some platforms, the Help browser can be either a stand-alone application or a web browser; on other platforms the Help browser is always a web browser. See the *Help Browser Interface* topic in the IDL Online Help for additional information.

Preferences

Preferences that apply to the IDL Workbench interface — editor settings, syntax coloring, and code templates, for example — are controlled via the IDL Workbench **Preferences** dialog. See the *IDL Preferences* topic in the IDL Online Help for details on IDL's workbench preferences. To display the **Preferences** dialog, select **Preferences** from the **Window** menu of the IDL Workbench interface. IDL Workbench preferences are grouped together under the heading **IDL**.
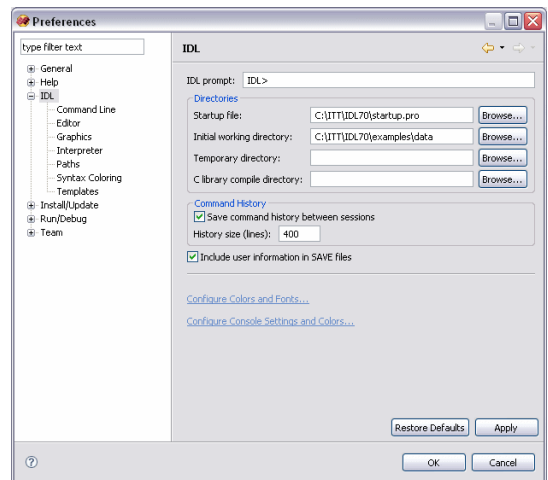
Note that the **Preferences** dialog also allows you to modify preferences for features that are not specifically related to IDL. These preferences include things like the external editors associated with specific file types, tasks to be invoked when starting up or shutting down the workbench, and keybindings. These non-IDL preferences are part of the Eclipse framework on which the IDL Workbench is built.

Use the filter text field at the top of the tree view in the **Preferences** dialog to locate specific items in the **Preferences** dialog.

Note ───

IDL Workbench preferences apply only when the IDL Workbench is running, and have no bearing on IDL programs. IDL System preferences, which control how IDL executes code, are set within IDL itself. See the *IDL Preferences* topic in the IDL Online Help for details.

──

Updating the IDL Workbench

The IDL Workbench provides an easy way to update and add to your IDL installation via the Internet. The **Software Updates** feature allows you to locate *plugins* that provide new features or revisions to existing features and install them automatically.

Types of features you might install include:

- Updates to the IDL Workbench itself, provided by ITT Visual Information Solutions.

- IDL features not included in the standard IDL distribution, such as file readers or even entire code libraries. These additional features may be available from the ITT Visual Information Solutions Code Contribution library or from other repositories set up by third parties.

- Eclipse features not included in the standard IDL Workbench distribution, such as source code managers (CVS, Subversion) or other productivity tools. These features may be available from ITT Visual Information Solutions (as is the plugin that integrates the CVS source code manager into the IDL Workbench) or from third parties.

Warning ——

While adding plugins provided by third parties *should* leave the IDL-specific features of the IDL Workbench unaltered, ITT Visual Information Solutions cannot vouch for the stability, quality, or usefulness of plugins from other sources. If you install a plugin that appears to adversely affect the IDL Workbench, uninstalling that single plugin should resolve the problem.

——

Installing New Features

To install new features, do the following:

1. Select **Help → Software Updates → Find and Install...**

2. Select **Search for new features to install** and click **Next>**.

3. Select an update site or add a new update site.

   The IDL Workbench includes the ITT Visual Information Solutions update site and some Eclipse-related sites by default. You can easily add other update sites to the list; see the *Installing new features with the update manager* topic in the IDL Online Help for details.

   For this example, select the **ITT Visual Information Solutions** update site.

4. Additions to the IDL
    Workbench itself are
    contained in the
    Workbench Updates
    section. In this section
    you will find plugins that
    provide additional or
    updated functionality to
    the workbench interface.

    The IDL Code
    Contributions section of
    the update site contains
    features that augment the
    IDL language. These
    plugins generally contain
    IDL source code, and
    may consist of single routines, entire applications, or even code libraries.
    Features available in this section have been created by IDL users either within
    or outside ITT Visual Information Solutions. While they have not received the
    rigorous quality assurance testing that IDL itself receives, they are of high
    quality and may be useful depending on your needs.

5. After selecting a feature to install, click **Next>**. Read and accept the license
    agreement for the feature, and click **Next>** again.

6. By default, plugins are installed in a subdirectory of the IDL installation
    directory, and IDL's search path is updated so that IDL can find the new plugin.
    You can install plugins in other directories by clicking **Change Location**; note,
    however, that if you install plugins in a different location you may need to
    manually modify IDL's search path.
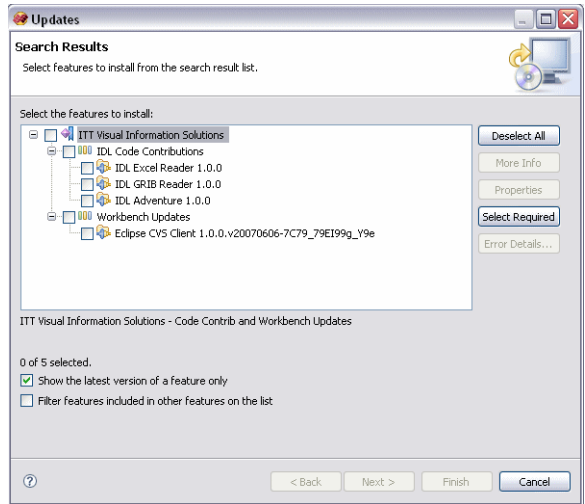
Updating Existing Features

To search for updates to features you have already installed, including the IDL
Workbench itself:

1. Select **Help → Software Updates → Find and Install...**

2. Select **Search for updates of the currently installed features** and click
    Next>.

3. The Update manager searches the network for updates to features in your
    current installation.

4. If it finds updates available, select the features you want to update and click **Next>**. Read and accept the license agreement for the feature, and click **Next>** again.

Managing Your Configuration

You can also use the **Product Configuration** dialog to manage your installation, search for updates, and disable specific plugins.

Select **Help → Software Updates → Manage Configuration** to display the **Product Configuration** dialog.

Chapter 4
Line Plots

This chapter shows how to display and modify two- and three-dimensional plots with the iPlot tool and Direct graphics.

IDL and 2-D Plotting

This section demonstrates how to create and manipulate two-dimensional plots using the iPlot tool and IDL's Direct graphics system.

To learn more about plotting linear data in IDL, see the *Line Plots* topic in the IDL Online Help.

For more information on working with the iPlot tool, see the *Working with Plots* topic in the IDL Online Help.

For a list of Direct graphics plotting routines, refer to the *Plotting* section of the *Functional List of IDL Routines* topic in the IDL Online Help.

Plotting with iPlot

The IDL iPlot tool displays your data in plot form. The iPlot tool then allows you great flexibility in manipulating and visualizing plot data. iPlot can be used for any type of two- or three-dimensional plot, including scatter plots, line plots, polar plots, and histogram plots.

Creating a Simple 2-D Plot

To create a simple line plot in the iPlot tool, enter the following code at the IDL command line:

```
iPlot, RANDOMU(seed, 20)
```

In this case, we are using the RANDOMU function to return twenty uniformly-distributed, floating-point, pseudo-random numbers that are greater than 0, and less than 1.0.
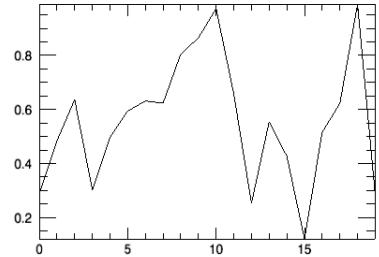
Creating a 2-D Overplot

In the iPlot tool, you may plot a new data set over a previously-drawn data set. This process (called overplotting) is useful for directly comparing multiple data sets.

In this example, we will plot a cosine wave on top of a sine wave.

1. The variable `theory` stores the points of a sine wave of decreasing amplitude.

    ```
    theory = SIN(2.0*FINDGEN(200)*!PI/25.0)*EXP(-0.02*FINDGEN(200))
    ```

2. Plot the sine wave in the iPlot tool.

    ```
    IPLOT, theory
    ```

3. Create the variable `newtheory` to contain cosine wave points.

    ```
    newtheory=COS(2.0*FINDGEN(200)*!PI/25.0)*EXP(-0.02*FINDGEN(200))
    ```
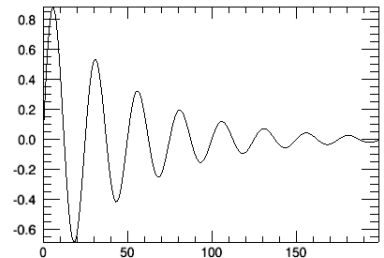
4. Overplot the cosine data in the iPlot tool.

   ```
   IPLOT, newtheory, /OVERPLOT, $
      LINESTYLE=2
   ```

   This plots the second line in the same iPlot window as the first. The LINESTYLE keyword changes the line style property of the plot to display a dashed line rather than a solid line. You can also overplot in the iPlot tool simply by loading new data over an older data set.
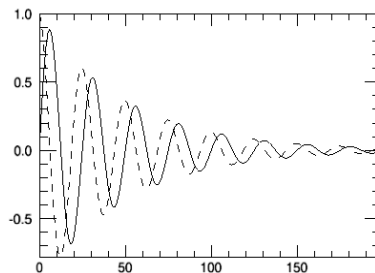
Plotting an ASCII Data Set

In this example, we will import an ASCII data set into IDL and plot it with the iPlot tool. Enter the following code at the IDL command line:

1. Create an ASCII template, which defines the format of a particular ASCII file. IDL will use this template to import the data. The plotTemplate variable contains the template.

   ```
   plotTemplate = ASCII_TEMPLATE( )
   ```

2. A dialog appears, prompting you to select a file. Select the plot.txt file in the examples/data subdirectory of the IDL distribution.

   After selecting the file, the **ASCII Template** dialog appears.

3. Select the **Delimited** field type, since the ASCII data is delimited by tabs (or spaces).

4. In the **Data Starts at Line** box, enter a value of 3. (The data does not start at line 1 because there are two comment lines at the beginning of the file.)

5. Click **Next**.

6. In the **Delimiter Between Data Elements** section, select **Tab**.

7. Click **Next**.

8. Name the ASCII file fields by selecting a row in the table at the top of the dialog and entering a value in the **Name** box.

   • Click on the table's first row (FIELD1). In the **Name** box, enter time.

   • Select the second row and enter temperature1.

   • Select the third row and enter temperature2.

9. Click **Finish**.

10. Enter the following code at the IDL command line to import the ASCII data
    file `plot.txt` using the custom template `plotTemplate`.

    ```
    plotAscii = READ_ASCII(FILEPATH('plot.txt', SUBDIRECTORY= $
        ['examples', 'data']), TEMPLATE=plotTemplate)
    ```

11. Plot the `temperature1` vs. `time` data.

    ```
    IPLOT, plotAscii.time, $
      plotAscii.temperature1
    ```

For more information on importing ASCII data,
see the *Reading ASCII Data* topic in the IDL
Online Help.
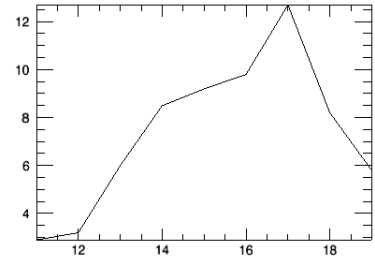
Adding Plot Titles

The iPlot tool allows you to modify your plots by
adding elements such as error bars, legends, and
axis titles. You can also manipulate the plot with
tools such as curve fitting or filtering.

In this example, we will add a main title and axis
titles to the ASCII data plot we created
previously. The VIEW_TITLE keyword adds a
main title, and the XTITLE and YTITLE
keywords add axis labels. If you have not already
done so this session, do the example "Plotting an
ASCII Data Set" on page 42.
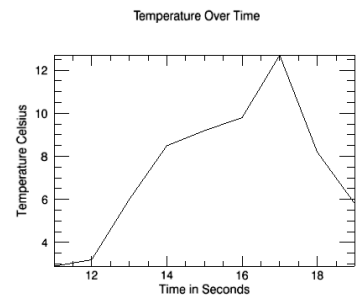
Enter the following code at the IDL command line, which will create a new iPlot
dialog and add titles to the plot.

```
IPLOT, plotAscii.time, plotAscii.temperature1, $
   VIEW_TITLE='Temperature Over Time', $
   XTITLE='Time in Seconds', $
   YTITLE='Temperature Celsius'
```

Alternately, you could add title annotations to an existing plot by selecting the Text
tool $\boxed{A}$, positioning the cursor at the location where you want the title to appear, and
typing the text. Double-clicking on the text displays the text annotation property
sheet, which allows you to modify the size, font, color, and other properties of the
annotation.

Changing the Data Range of a Plot

Your data set may contain more data than you want
to display in a particular plot. While you could use
IDL's array subscripting syntax to create a subset of
the original array, it is often easier to simply limit
the range used when creating the plot display.

For example, suppose you wanted to restrict the
range displayed in your plot to show only time
values (the X-axis) between 15 and 18 seconds, and
temperature values (the Y-axis) between 8 and 13 degrees. Using the [XYZ]RANGE
keywords to the IPLOT routine allows you to do this when creating the plot:
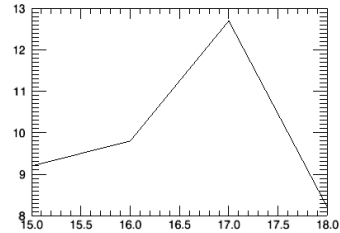
```
IPLOT, plotAscii.time, plotAscii.temperature1, $
    XRANGE=[15,18], YRANGE=[8,13]
```

Alternately, you could start by displaying the full data range in iPlot, and then alter
the Dataspace properties to reflect the new X and Y ranges:
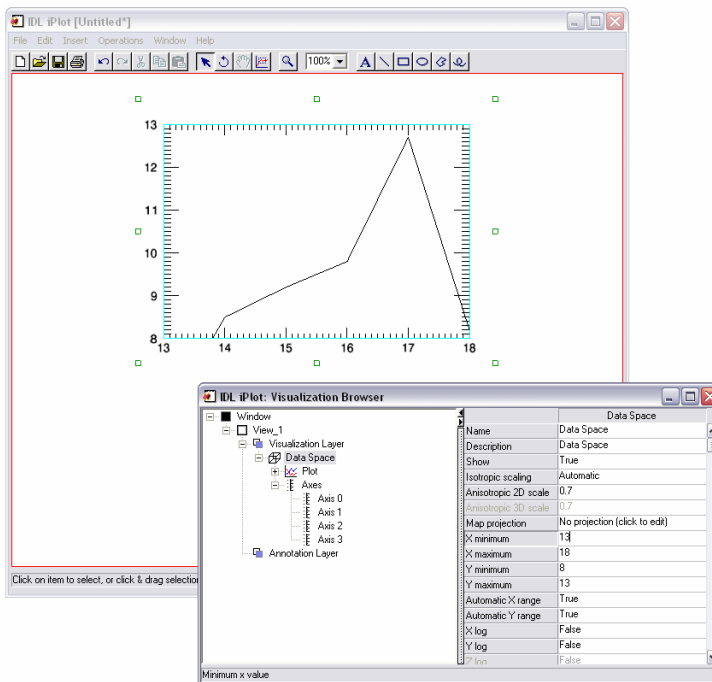
Using Plotting Symbols and Line Styles

When plotting several data sets in a single plot, it is often useful to use symbols, line styles, and legends to differentiate between the data sets. The following procedure created the plot shown at right.
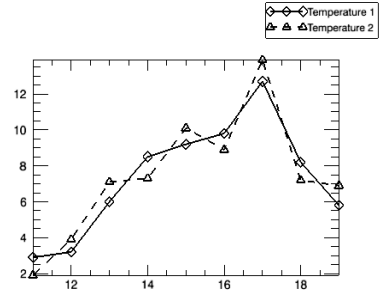
1. First, plot the temperature1 values using the standard (solid) line style and a diamond symbol to mark the data points:

   ```
   IPLOT, plotAscii.time, plotAscii.temperature1, SYM_INDEX=4
   ```

   Note that we set the SYM_INDEX keyword equal to four to create the diamond symbols.

2. Next, overplot the temperature2 values using a dashed line (LINESTYLE=2) and a triangle symbol to mark the data points (SYM_INDEX=5):

   ```
   IPLOT, plotAscii.time, plotAscii.temperature2, SYM_INDEX=5, $
       LINESTYLE=2, /OVERPLOT
   ```

3. To insert a legend, click in the plot area to select it (the axis lines around the plots will be highlighted), then select New Legend from the Insert menu. The legend is created using default names for the data sets (Plot and Plot1).

4. Double click on Plot in the legend to bring up the property sheet, and change the value in the Text field to Temperature 1. Similarly, change Plot1 to Temperature 2.

Adding Error Bars

You can add error bars to your plot using the [XYZ]ERROR keyword to IPLOT.

Suppose you know that the temperature values you have collected are only accurate within 0.3 degrees Celsius. To include error bars on your plot of temperature versus time, you would do the following:

1. Create an array with the same number of elements as you have temperature readings:
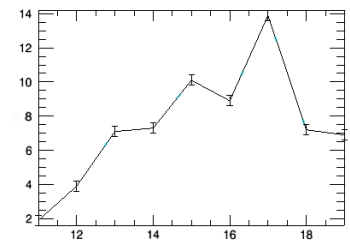
```
error_bars = FLTARR(N_ELEMENTS(plotAscii.temperature1))+0.3
```

This creates a floating-point array with the same number of elements as the plotASCII.temperature1 array, setting each element's value equal to 0.3.

Note
The size of the error bar does not need to be the same for every data point. Each element in the error_bars array could contain a different value.

2. Use the YERROR keyword to add the error bars:

```
IPLOT, plotAscii.time, plotAscii.temperature1, YERROR=error_bars
```

Plotting with Direct Graphics

In IDL's Direct graphics system, plots are created with the PLOT procedure.

Creating plots using Direct graphics is not as convenient as using the iPlot tool, but may be desirable if you are incorporating images into a larger widget-based application, or if you need to programatically create a large number of processed images.

Creating a Simple 2-D Plot

In this example, we will plot a sine wave using the Direct graphics PLOT procedure.

1. Since we are using Direct graphics, tell IDL to use a maximum of 256 colors and load a simple grayscale color map.

    ```
    DEVICE, RETAIN=2, DECOMPOSED=0
    LOADCT, 0
    ```

2. Create the X-axis values. The FINDGEN function creates an array of one hundred elements, with each element equal to the value of the element's subscript.

    ```
    X= 2*!PI/100 * FINDGEN(100)
    ```

3. Create the plot.

    ```
    PLOT, SIN(X)
    ```
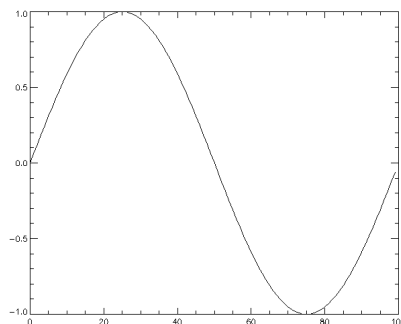
Creating a 2-D Overplot

As with the IDL iPlot tool, you can overlay plots in Direct graphics. This is accomplished with the OPLOT procedure.

It is often a good idea to change the color, line style, or line thickness parameters when calling OPLOT to distinguish the data sets. Refer to the *OPLOT* topic in the IDL Online Help for more information.

1. If you have not already done so, do the example "Creating a Simple 2-D Plot" on page 47.
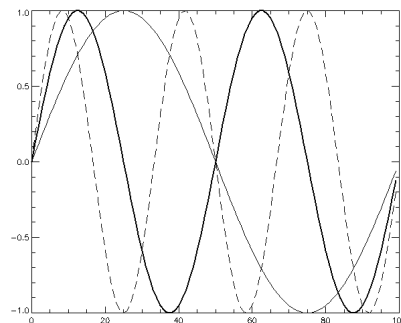
2. Overplot a new sine wave with twice the frequency. Make the line twice as thick.

```
OPLOT, SIN(2*X), THICK = 2
```

3. Overplot yet another sine wave with triple the frequency. Instead of a line, use long dashes.

```
OPLOT, SIN(3*X), LINESTYLE = 5
```

Printing a Direct Graphics Window

To print an image from an iTool window, you simply select Print from the iTool's Filemenu. Printing the contents of a Direct graphics window is more involved. To print a Direct graphics plot, enter the following commands at the IDL command line:

1. Save the current plotting environment variable to a local variable.

```
MYDEVICE=!D.NAME
```

2. Designate the printer as the plot destination.

```
SET_PLOT, 'printer'
```

3. Plot your data, with the output now directed to the printer.

```
PLOT, SIN(X)
```

4. Close the printing device.

```
DEVICE, /CLOSE
```

5. Restore the original output device for your plots.

```
SET_PLOT, MYDEVICE
```

See *The Printer Device* topic in the IDL Online Help for information on choosing a system printer for use when printing Direct graphics windows.

Note ──
If you experience problems printing on a UNIX platform, check that your printer is correctly configured. For more information, refer to the *IDL Printer Setup for UNIX or Mac OS X* topic in the IDL Online Help.
──

IDL and 3-D Plotting

IDL can also create three-dimensional line plots. As an example, enter the following code at the IDL command line to create a simple three-dimensional plot with the iPlot tool:

```
x = FINDGEN(200)
IPLOT, x * COS(x/10), $
    x * SIN(x/10), $
    x,SYM_INDEX=5
```

For more information on other types of three-dimensional plots, see
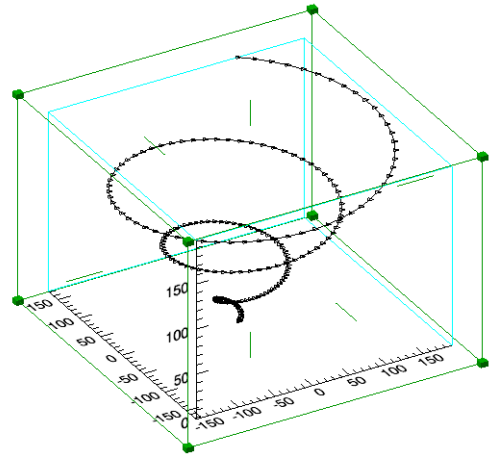
Chapter 5
Surfaces and Contours

This chapter shows how to display and process images with the iSurface and iContour tools, and with Direct graphics.

Surfaces and Contours in IDL

IDL provides tools for visualizing and manipulating many types of three-dimensional arrays, including contour plots, wire-mesh surfaces, and shaded surfaces. This chapter demonstrates how to visualize data in three dimensions using the IDL iTools and Direct graphics.

To learn more about visualizing and manipulating surfaces and contours in IDL, see the *Using IDL* user's guide in the IDL Online Help.

For more information on working with the iSurface and iContour tools, see the *Working with Surfaces* topic and the *Working with Contours* topic in the IDL Online Help.

For a description of the commonly-used Direct graphics routines used for visualizing 3-D data, refer to the *SURFACE*, *SHADE_SURF*, and *CONTOUR* topics in the IDL Online Help.

Displaying Surfaces with iSurface

The IDL iSurface tool allows you to visualize, modify, and manipulate surfaces in a single interactive environment.

Displaying Surfaces

In this example, we use the RESTORE procedure, which loads IDL variables and routines into memory that were previously saved to a file by the SAVE procedure.

To import surface data into the iSurface tool and visualize it, enter the following commands at the IDL command line:

1. Restore the `marbells.dat` SAVE file:

```
RESTORE, FILEPATH('marbells.dat', SUBDIRECTORY=['examples', 'data'])
```

By restoring `marbells.dat`, the array variable `elev` is loaded into memory.

2. Load the surface data into the iSurface tool and display it:

```
ISURFACE, elev
```
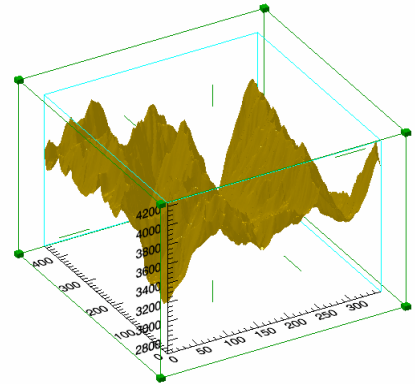
Displaying Shaded Surfaces

In the following example, we will add an external light source to a surface in the iSurface tool.

1. If you have not already done so, restore the `marbells.dat` SAVE file.

```
RESTORE, FILEPATH('marbells.dat', SUBDIRECTORY=['examples', 'data'])
```

2. Load the surface data into the iSurface
    tool and display it.

    `ISURFACE, elev`

3. Add a light source to the image by
    selecting **Insert → Light**.

4. Select the light bulb icon, and move it
    around the surface to see how the
    surface shadows change.
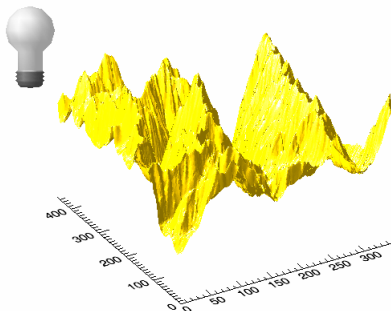
Modifying Surfaces

The iSurface tool allows you to manipulate and modify displayed surfaces. For
example, rotation tools are provided to make it easier to see all aspects of a 3-D
surface.

To rotate a surface freely or along an axis:

1. If you have not already done so, restore the marbells.dat SAVE file.

    `RESTORE, FILEPATH('marbells.dat', SUBDIRECTORY=['examples', 'data'])`

2. Load the surface data into the iSurface tool and display it:

    `ISURFACE, elev`

3. Select the surface in the iSurface window.

4. Click **Rotate** 🔄 on the window toolbar. The rotation sphere is displayed
    around the surface.

    •   To rotate the surface freely, position
        the mouse pointer over the surface so
        that it changes to a free rotation
        pointer ↻ . Click and drag to rotate
        the surface in the desired direction.

    •   To rotate the surface along an axis,
        position the mouse pointer over an
        axis so that it changes to an axis
        rotation pointer ⟳ . Click and drag
        to rotate the surface along the axis in
        the desired direction.
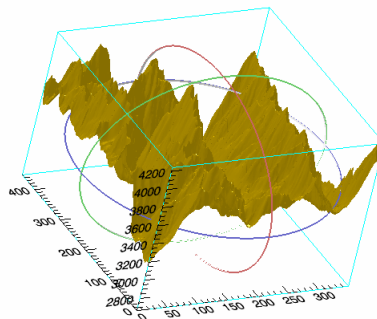
To rotate a surface in 90° increments left or right:

1. Select the surface in the iSurface window.

2. Select **Operations → Rotate → Rotate Left** or
   Operations → Rotate → Rotate Right.

To rotate a surface an arbitrary number of degrees:

1. Select the surface in the iSurface window.

2. Select **Operations → Rotate → Rotate by Angle**.

3. In the **Rotate Angle** dialog, enter the desired number of degrees to rotate the surface and click **OK**.

There are many other ways to modify a surface in the iSurface tool, including manipulating surface color, texture mapping, and surface annotation. For more information on working with the iSurface tool, see the *Working with Surfaces* topic in the IDL Online Help.

Displaying Surfaces with Direct Graphics

Working with images using Direct graphics is not as convenient as using the iImage tool, but may be desirable if you are incorporating images into a larger widget-based application, or if you need to programatically create a large number of processed images.

In this example, we will display three-dimensional surface data using IDL Direct graphics. Enter the following commands at the IDL command line:

1. If you have not already done so, restore the `marbells.dat` SAVE file.

    ```
    RESTORE, FILEPATH('marbells.dat', SUBDIRECTORY=['examples', 'data'])
    ```

2. Since we are using Direct graphics, tell IDL to use a maximum of 256 colors and load a simple grayscale color map.

    ```
    DEVICE, RETAIN=2, DECOMPOSED=0
    LOADCT, 0
    ```

3. Use the CONGRID procedure to resample the data set so that the grid can be displayed at a visible size. In this case, resample the array size to 35 x 45, or one-tenth its original size.

    ```
    MARBELLS=CONGRID(elev,35,45)
    ```

4. Visualize the grid.

    ```
    SURFACE, MARBELLS
    ```
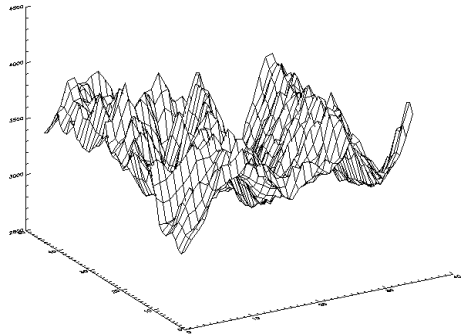
The SURFACE procedure can be used to view your data from different angles. The AX keyword specifies the surface angle of rotation (in degrees towards the viewer) about the X axis. The AZ keyword specifies the surface rotation in degrees, counterclockwise about the Z axis.

5. View the array from a different angle.

    ```
    SURFACE, MARBELLS, AX = 70, AZ = 25
    ```
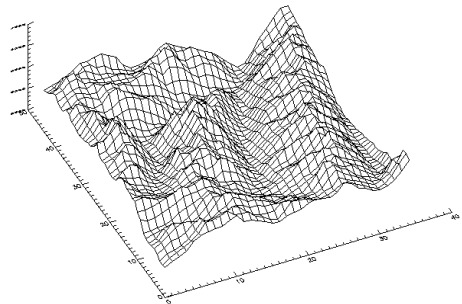
Displaying Contours with iContour

The iContour tool allows you to visualize, modify, and manipulate two-dimensional contour data in a single interactive environment.

Displaying Contours

To import data into the iContour tool and visualize it, enter the following commands at the IDL command line:

1. If you have not already done so, restore the marbells.dat SAVE file.

   ```
   RESTORE, FILEPATH('marbells.dat', $
     SUBDIRECTORY=['examples', 'data'])
   ```

2. Load the data into the iContour tool and display it.

   ```
   ICONTOUR, elev
   ```

3. To create filled contours:

   ```
   ICONTOUR, elev, /FILL, $
     RGB_TABLE=0, N_LEVELS=0
   ```
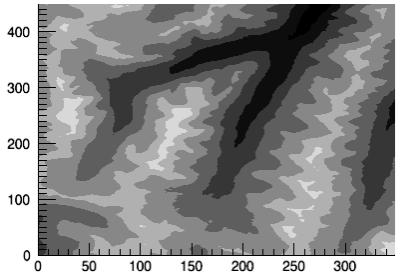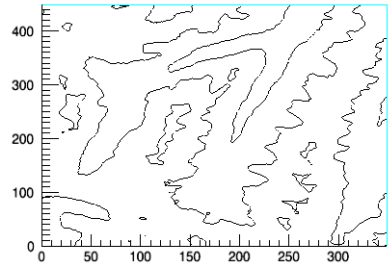

Modifying Contours

The iContour tool allows you to manipulate and modify displayed contours. For example, you can add a legend that shows the contour levels.

To add a legend, select **Insert → New Legend**. Double-click on the legend to display a dialog that allows you to modify the legend contents.

You might, for example, change the legend title, hide contour levels, or change the text style. For more information on working with the iContour tool, see the *Working with Contours* topic in the IDL Online Help.
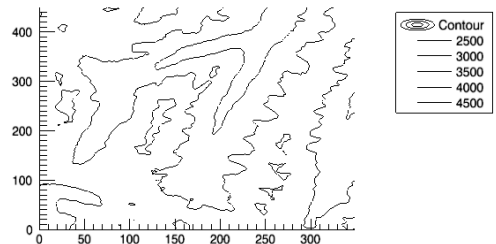
Displaying Contours with Direct Graphics

Working with images using Direct graphics is not as convenient as using the iImage tool, but may be desirable if you are incorporating images into a larger widget-based application, or if you need to programatically create a large number of processed images.

In this example, we will display a two-dimensional array as a contour plot using IDL Direct graphics. Enter the following commands at the IDL command line:

1. If you have not already done so, restore the `marbells.dat` SAVE file.

   ```
   RESTORE, FILEPATH('marbells.dat', $
     SUBDIRECTORY=['examples', 'data'])
   ```

2. Since we are using Direct graphics, tell IDL to use a maximum of 256 colors and load a simple grayscale color map.

   ```
   DEVICE, RETAIN=2, DECOMPOSED=0
   LOADCT, 0
   ```

3. Plot the contour.

   ```
   CONTOUR, elev
   ```

4. Create a customized CONTOUR plot with more contour lines.

   ```
   CONTOUR, elev, NLEVELS=8, $
     C_LABELS=[0,1]
   ```

   The NLEVELS keyword directs CONTOUR to plot eight equally-spaced contours. The C_LABELS keyword specifies which contour levels should be labeled (by default, every other contour is labeled).
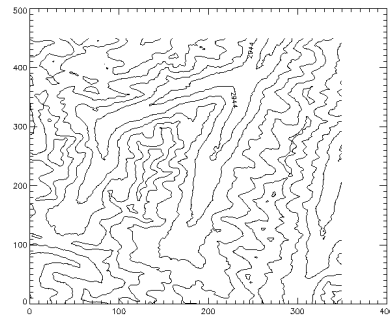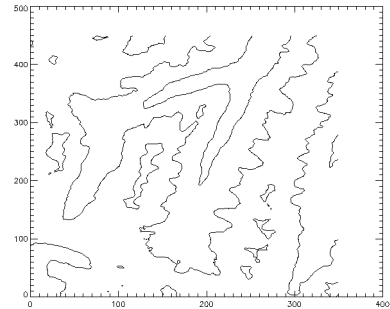
5. Similarly, you can create a filled contour plot where each contour level is filled with a different color (or shade of gray) by using the FILL keyword.

```
CONTOUR, elev, NLEVELS=8, /FILL
```

6. To outline the resulting contours, make another call to CONTOUR and use the OVERPLOT keyword to overlay the previous plot.

    You can add tickmarks that indicate the slope of the contours (the tickmarks point downhill) by using the DOWNHILL keyword.

```
CONTOUR, elev, NLEVELS=8, $
  /OVERPLOT, /DOWNHILL
```

7. CONTOUR can plot surface data in a three-dimensional perspective.

    First, set a three-dimensional viewing angle.

```
SURFR
```

8. By using the T3D keyword, the contours are drawn in a three-dimensional perspective.

```
CONTOUR, elev, NLEVELS=8, /T3D
```
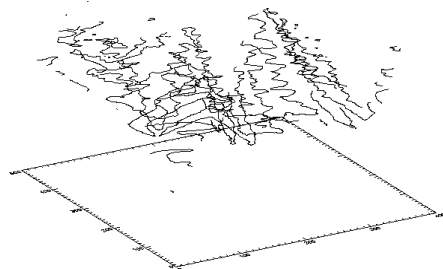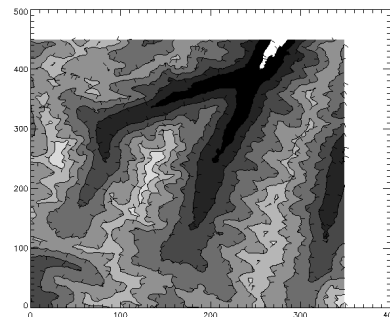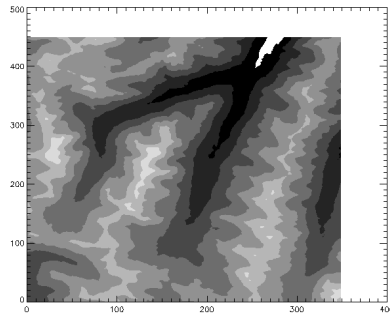
Working with Irregularly Gridded Data

The IDL routines TRIANGULATE and TRIGRID allow you to fit irregularly sampled data to a regular grid, allowing you to visualize the values using IDL's surface and contour visualization routines. This example creates surface plots of some irregularly sampled data.

1. First, we create a sample data set from some random values.

```
x = RANDOMU(seed, 32)
y = RANDOMU(seed, 32)
z = EXP(-3*((x-0.5)^2+(y-0.5)^2))
```

   (For more on IDL's random number generation, see the *RANDOMU* topic in the IDL Online Help.)

2. Use the TRIANGULATE procedure to construct a Delaunay triangulation of our set of randomly-generated points:

```
TRIANGULATE, x, y, tr
```

   The variable `tr` now contains a three-dimensional array listing the triangles in the Delaunay triangulation of the points specified by the X and Y arguments.

3. Use the TRIGRID function to create a regular grid of interpolated Z values, using the Delaunay triangulation:

```
grid_linear = TRIGRID(x, y, z, tr)
```

   By default, the TRIGRID function uses linear interpolation. To use quintic interpolation, set the QUINTIC keyword:
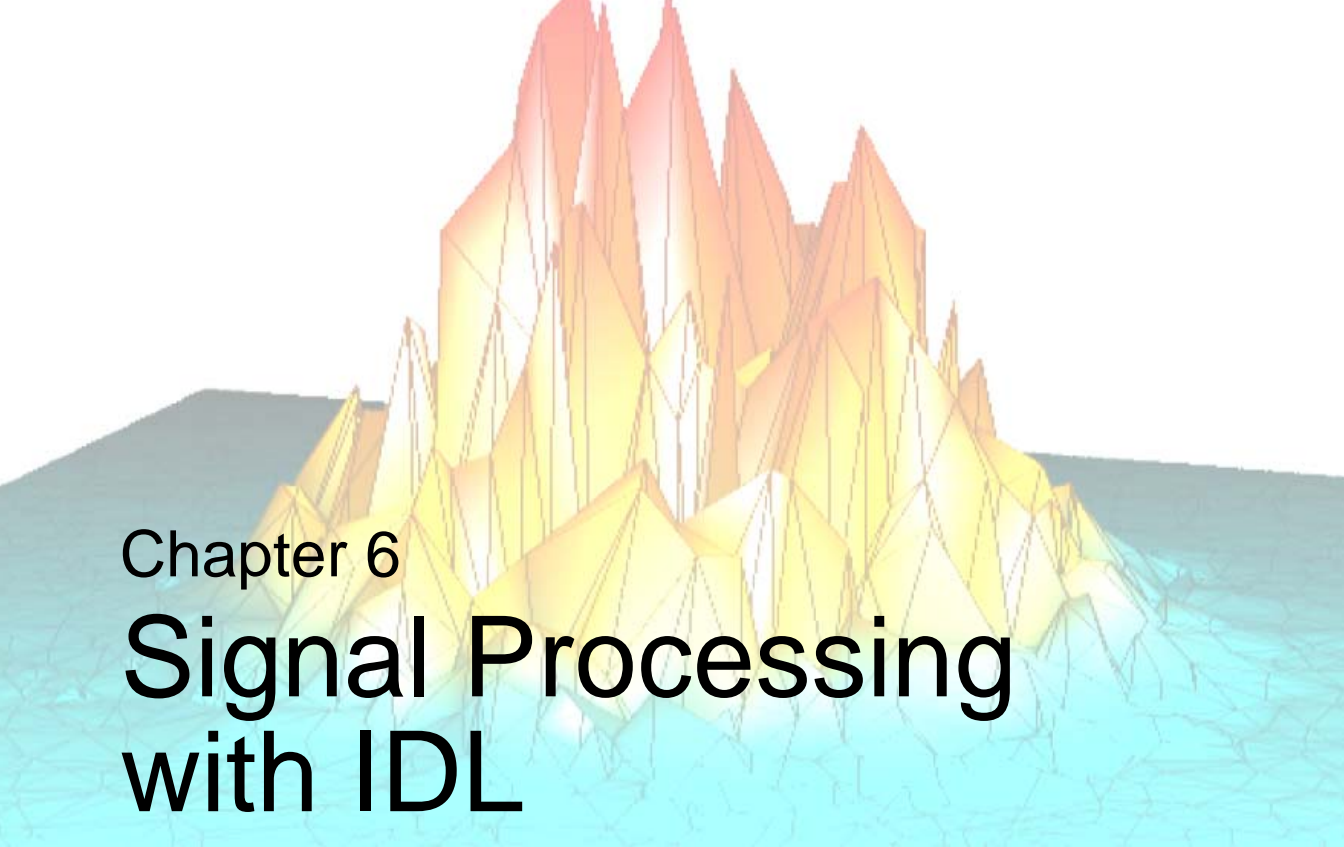
```
grid_quintic = TRIGRID(x, y, z, tr, /QUINTIC)
```

4. Display the interpolated surface values as wire-frame meshes side by side in the iSurface tool:

```
ISURFACE, grid_linear, STYLE=1, VIEW_GRID=[2,1]
ISURFACE, grid_quintic, STYLE=1, /VIEW_NEXT
```

For more information, see the *TRIANGULATE* and *TRIGRID* topics in the IDL Online Help.
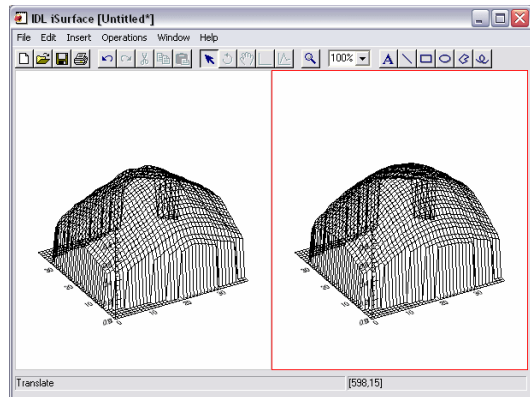
Chapter 6
Signal Processing with IDL

This chapter describes the following topics:

IDL and Signal Processing

This chapter introduces you to IDL's digital signal processing tools. First, we introduce some basic signal processing concepts such as removing noise, curve fitting, correlation, and transforms. Then we discuss the process of creating a data set and adding noise to appear like raw data. We use that "noisy" data set to understand different methods of removing noise. Finally, we view an existing data set consisting of damped sine wave data with severe high-frequency noise.

Most of the procedures and functions mentioned here work in two or more dimensions. For simplicity, only one-dimensional signals are used in the examples.

Using just a few IDL commands, you can perform some complex and powerful signal processing tasks. IDL has many more signal processing abilities than the ones shown in this chapter. To take advantage of all of IDL's powerful capabilities, look for more information in Chapter 8, "Signal Processing" (*Using IDL*) manual.

Signal Processing Concepts

This section introduces some basic signal processing concepts that you need to know before working with signal data.

Removing Noise

A signal, by definition, contains information. Any signal obtained from a physical process also contains unwanted frequency components (noise). IDL provides several digital filter routines to remove noise.

Some noise can simply be removed by smoothing or masking an image or masking it within the frequency domain, but some noise requires more filtering. (See the definition for Wavelet, below.)

See the *Digital Filtering* topic in the IDL Online Help for more information.

Curve Fitting

Curve fitting is the process of finding various ways to fit a curve to a series of data points that best represents all points. Curve-fitting can also estimate points between values along a continuum. Curve fitting allows you to find intermediate estimates for these values. IDL's CURVEFIT function uses a gradient-expansion algorithm to compute a non-linear least squares fit to a user-supplied function with an arbitrary number of parameters.

See the *Curve and Surface Fitting* topic in the IDL Online Help for more information.

Convolution and Correlation

The term *convolution* refers to the relationship between the input signal, output signal, and impulse response. Correlation is a method of detecting a known waveform in the noisy background signals. Signals of any given type travel at a known rate, and correlation determines if the signal also occurs in another signal.

Mathematically, convolution and correlation are similar. They both use two signals to produce a third signal. In correlation, this third signal is called the cross-correlation of the input signals.

See the *Correlation and Covariance* topic in the IDL Online Help for more information.

Transforms

It is often difficult or impossible to make sense of the information contained in a digital signal by looking at it in its raw form—that is, as a sequence of real values at discrete points in time. Signal analysis transforms offer natural, meaningful, alternate representations of the information contained in a signal. Transforms make signal processing easier by changing the *domain* in which the underlying signal is represented.

Most signals can be decomposed into a sum of discrete (usually sinusoidal) signal components. The result of such decomposition is a frequency spectrum that can uniquely identify the signal. IDL provides three transforms to decompose a signal and prepare it for analysis: the Fourier transform, the Hilbert transform, and the wavelet transform.

See the *Signal Analysis Transforms* topic in the IDL Online Help for more information.

Wavelet Analysis

Wavelet analysis is a technique to transform an array of *N* numbers from their actual numerical values to an array of N wavelet coefficients. Since the wavelet functions are compact, the wavelet coefficients measure the variations around just a small part of the data array. Wavelet analysis is useful for signal processing because the wavelet transform allows you to easily pick out features in your data, such as noise or discontinuities, discrete objects, edges of objects, etc.

See the *Using the IDL Wavelet Toolkit* topic in the IDL Online Help for more information.

Creating a Data Set

In this example, we create a data set and then introduce noise to make it appear more like real-world data. Then we plot the original data and the "noisy" data together in the same window to see the difference.

First, we need to create a data set to display.

1. Enter the following command to create a sine wave function with a frequency that increases over time and store it in a variable called DATA:

    ```
    data = SIN((FINDGEN(200)/35)^2.5)
    ```

    The FINDGEN function returns a floating-point array in which each element holds the value of its subscript, giving us the increasing "time" values upon which the sine wave is based. The sine function of each "time" value divided by 35 and raised to the 2.5 power is stored in an element of the variable DATA.

2. To view a quick plot of this data set, shown in the following diagram, enter:

    ```
    IPLOT, data
    ```

3. Add some uniformly-distributed random noise to this data set and store it in a new variable:

    ```
    noisy = data + ((RANDOMU $
        (SEED,200)-.5)/ 2)
    ```
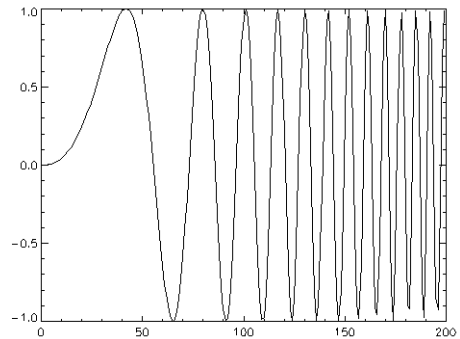
    The RANDOMU function creates an array of uniformly distributed random values. The original data set plus the noise is stored in a new variable called NOISY. When you plot this data set, it looks more like real-world test data.

4. Now plot the array:

    ```
    IPLOT, noisy
    ```

5. Display the original data set and the
    noisy version simultaneously by
    entering the following commands:

    ```
    IPLOT, data, XTITLE="Time",$
        YTITLE="Amplitude", THICK=3
    ```

6. Then overplot the previous data:

    ```
    IPLOT, noisy, /OVERPLOT
    ```
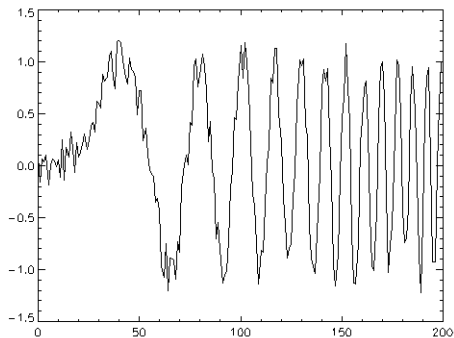
The XTITLE and YTITLE
keywords are used to create the X
and Y axis titles. The OVERPLOT
keyword plots the NOISY data set
over the existing plot of DATA.
Setting the THICK keyword causes
the default line thickness to be
multiplied by the value assigned to
THICK, so you can differentiate
between the data. This result can
be seen in the figure to the right.
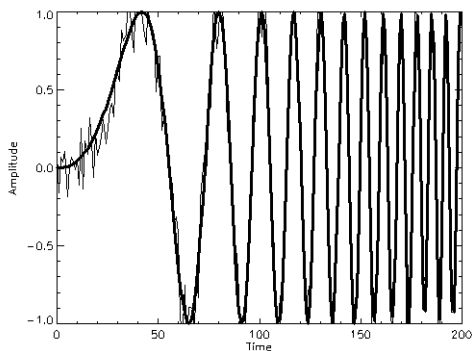
Signal Processing with SMOOTH

A simple way to smooth out the NOISY data set created in the previous example is to use IDL's SMOOTH function. It returns an array smoothed with a boxcar average of a specified width.

1. Create a new variable to hold the smoothed data set by entering the following command:

```
SMOOTHED = SMOOTH(noisy, 5)
```

2. Now plot your new data set:

```
IPLOT, SMOOTHED, VIEW_TITLE='Smoothed Data'
```

The TITLE keyword draws the title text centered over the plot. Notice that while SMOOTH did a fairly good job of removing noise spikes, the resulting amplitudes taper off as the frequency increases.
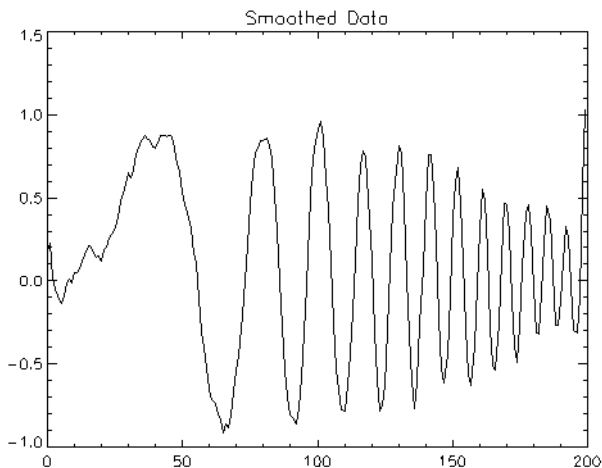
Figure 6-1: Using the SMOOTH Command

See the next example for another method of reducing noise in the data set.

Frequency Domain Filtering

Frequency domain filtering is another (perhaps better) way to eliminate noise. Noise is unwanted high-frequency content in sampled data. Applying a lowpass filter to the noisy data allows low-frequency components to remain unchanged while high frequencies are smoothed or attenuated. Construct a filter function by entering the following step-by-step commands:

1. Create a floating point array using FINDGEN which sets each element to the value of its subscript and stores it in the variable Y by entering:

    ```
    y=[FINDGEN(100),FINDGEN(100)-100]
    ```

2. Next, make the last 99 elements of Y a mirror image of the first 99 elements:

    ```
    y[101:199]=REVERSE(y[0:98])
    ```

3. Now, create a variable filter to hold the filter function based on Y:

    ```
    filter=1.0/(1+(y/40)^10)
    ```

4. Finally, plot:

    ```
    IPLOT, filter
    ```


    The next step applies the filter to the NOISY data. To filter data in the frequency domain, we multiply the Fast Fourier transform (FFT) of the data by the frequency response of a filter and then apply an inverse Fourier transform to return the data to the spatial domain.
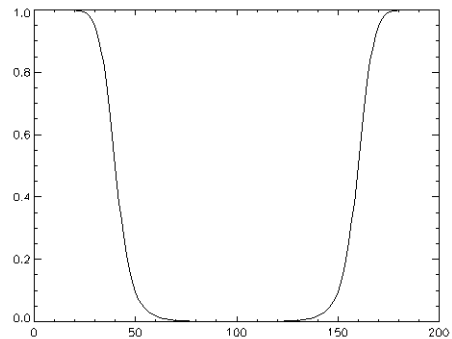
5. Now we can use a lowpass filter on the NOISY data set and store the filtered data in the variable lowpass by entering:

    ```
    lowpass = FFT(FFT(noisy,1)*filter,-1)
    ```

6. Then plot the filtered data:

```
IPLOT, lowpass
```


Note ———————————————

Your plots may look slightly different
due to the random number generator.
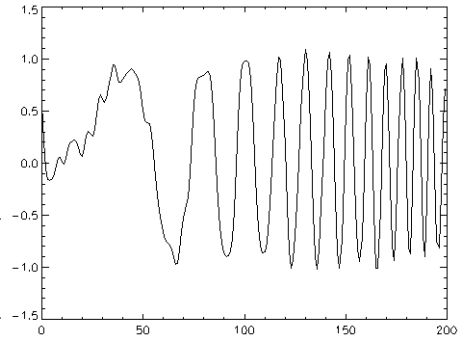
The same filter function can be used as a
high-pass filter (allowing only the high frequency or noise components through).

7. To accomplish this, enter:

```
highpass = FFT(FFT(noisy,1)*(1.0-filter),-1)
```

8. Then plot the result:

```
IPLOT, highpass
```
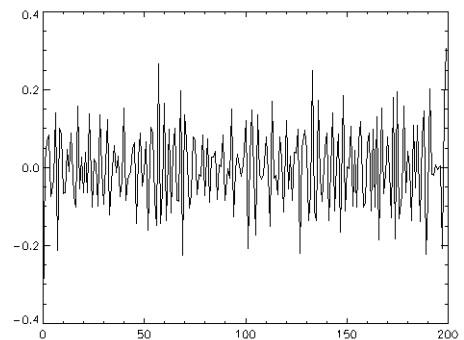
Displaying Multiple Plots in a Single Window

To display all the plots that were created in the previous sections, enter the following
IPLOT commands on the IDL command line:

```
IPLOT, noisy, VIEW_GRID=[2,2], VIEW_TITLE='Noisy Data Set'
IPLOT, filter, /VIEW_NEXT, VIEW_TITLE='Filter'
IPLOT, lowpass, /VIEW_NEXT, VIEW_TITLE='Low-Pass Filtered'
IPLOT, highpass, /VIEW_NEXT, VIEW_TITLE='High-Pass Filtered'
```

The resulting IPLOT window displays all four plots:
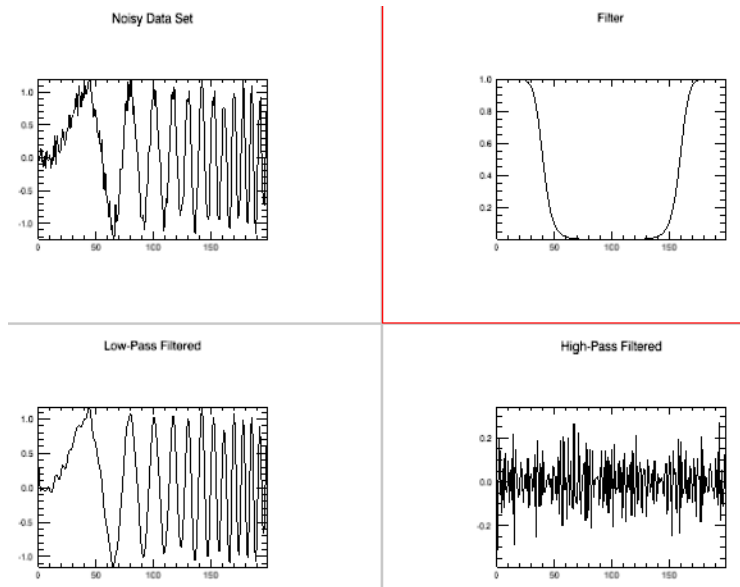
The following list describes the functionality of the IPLOT keywords used in the
previous commands:

- VIEW_GRID — Defines a plot grid by columns and rows.

- VIEW_TITLE — Defines the title for the current plot.

- VIEW_NEXT — Defines the next plot in relation to the current plot.

See the *IPLOT* topic in the IDL Online Help for complete information on these and
other keywords.

Creating Custom Filters

IDL provides the DIGITAL _FILTER function to allow you to compute your own data filters. This section gives a quick overview of how a Bandstop Finite Impulse Response (FIR) Filter works.

FIR filters are digital filters that have an impulse response that reaches zero in a finite number of steps. An FIR filter can be implemented non-recursively by convolving its impulse response with the time data sequence it is filtering. FIR filters are somewhat simpler than Infinite Impulse Response (IIR) filters, which contain one or more feedback terms and must be implemented with difference equations or some other recursive technique.

The DIGITAL_FILTER function constructs lowpass, highpass, bandpass, or bandstop filters. The figure at right plots a bandstop filter that suppresses frequencies between 7 cycles per second and 15 cycles per second for data sampled every 0.02 seconds.
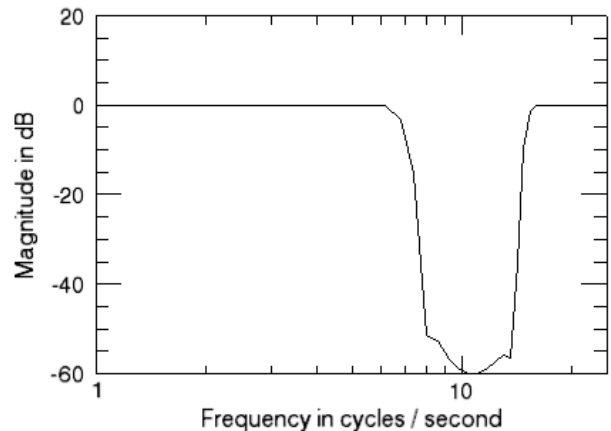
Type @sigprc10 at the IDL prompt to run the batch file that creates this display. The filter consists of 10 IDL statements, plus a call to the IPLOT routine for display. The source code is located in sigprc10, in the examples/doc/signal directory. View the code to start learning how to create your own custom filters. See the *Signal Processing* topic in the IDL Online Help for more information.

Wavelet Filtering Example

In this example, we use existing data rather than creating sample data. The example file is damped sine wave data with severe high-frequency noise.

1. Use the input variable to define the data to use:

    ```
    input = FILEPATH('damp_sn.dat', $
       SUBDIRECTORY=['examples', 'data'])
    ```

2. Create another variable to contain the output from the READ_BINARY function. READ_BINARY reads the contents of a file based on keywords or a predefined template. The DATA_DIMS keyword sets a scalar or array specifying the size of the data to be read and returned. (The array value of damp_sn.dat is provided in the index.txt file in examples/data.)

    ```
    output = READ_BINARY(input, DATA_DIMS=[512,1])
    ```

3. Plot the data:

    ```
    IPLOT, output
    ```

4. Use the wavelet transform to reduce the noise in the plot:

    ```
    smooth=WV_DENOISE(output, $
       'Coiflet', 3, PERCENT=50)
    ```
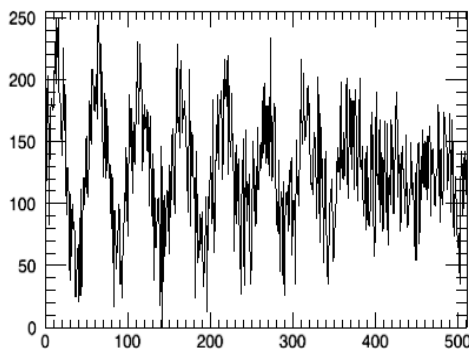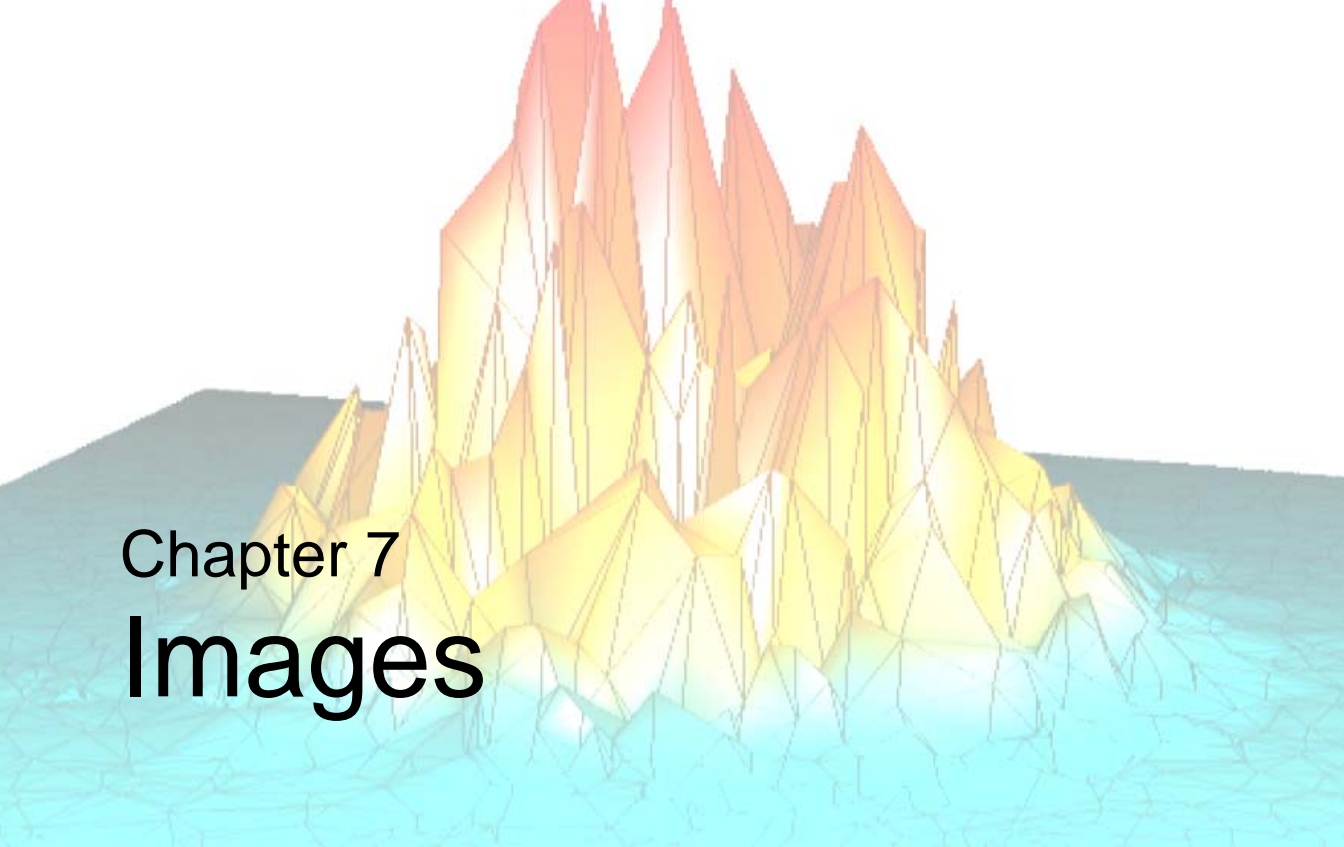

5. Plot the smoothed data:

    ```
    IPLOT, smooth, YRANGE=[0,255]
    ```

    Note that we specify the Y range to ensure that it is the same as in the previous plot.
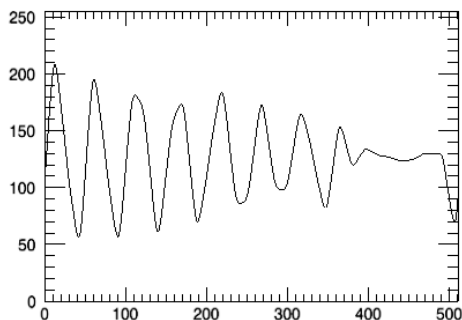
Chapter 7
Images

This chapter shows how to display and process images with the iImage tool and Direct graphics.

IDL and Images

IDL is ideal for working with image data because of its interactive operation, uniform notation, and array-oriented operators and functions. Images are easily represented as two-dimensional arrays in IDL and can be processed just like any other array. IDL also contains many procedures and functions specifically designed for image display and processing. In addition, the iImage tool allows you great flexibility in manipulating and visualizing image data.

To learn more about IDL's image processing capabilities, see the *Image Processing* user guide in the IDL Online Help.

Displaying Images with iImage

The iImage tool is a powerful option for visualizing and manipulating image data. Unlike command line-oriented visualizations, a single command with iImage gives you a fully interactive environment.

For more information on working with the iImage tool, see the *Working with Images* topic in the IDL Online Help.

Displaying Images

For this example, we will display a TIFF image of an aerial view above Manhattan.

1. At the IDL command line, enter **iImage**.

    The iImage tool displays.

2. On the iImage tool, select **File → Open**, and select `image.tif` from the `examples\data` subdirectory of your IDL installation.

3. Click **Open**, and the file is displayed in the iImage tool.

Resizing Images

There are several easy ways to resize an image in the iImage tool:

View Zoom   Size Menu

- On the toolbar, select a percent value from the **Size** menu (25%, 75%, and so on).

- On the toolbar, click the **View Zoom** button, click on the image, and use the mouse scroll wheel to increase or decrease the image magnification.
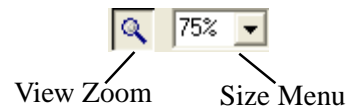
Contrast Enhancement

In order to improve the look of an image, sometimes changing how the colors are represented is all that is necessary. IDL provides several ways to manipulate the contrast.

Thresholding

The Thresholding operation takes an image containing a range of pixel values and produces a two-value image (effectively a black and white image). Specifically, all the pixel values up to a certain value are represented by either black or white pixels, and all the pixel values above the threshold value are represented by the opposite color. For example, a threshold value of 150 produces an image in which all the pixel values under 150 are represented by black pixels, and all pixel values of 150 and above are represented by white pixels.
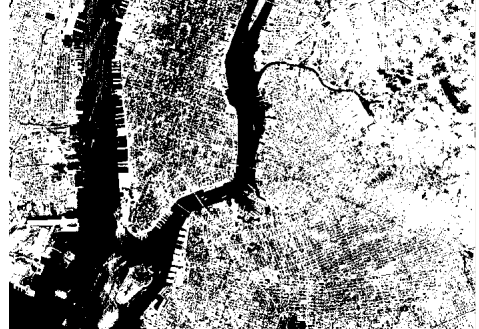
In the following example, we use the "greater than" operator (GT) to create a thresholded image in which pixel values greater than 140 are white and all others are black.

1. Load the `image.tif` file into an IDL variable:

```
img = READ_TIFF(FILEPATH('image.tif', $
   SUBDIRECTORY=['examples', 'data']))
```

2. The operation `img GT 140` creates an array of ones and zeros. The BYTSCL command transforms the array into values of 255 and zero.

   Scale the pixel values of the `image.tif` file to the entire range of a byte (0-256), and send all values greater than 140 to the iImage tool.

   ```
   IIMAGE, BYTSCL(img GT 140)
   ```

To create a thresholded image in which pixels with values less than 140 are white (the inverse of the previous example), enter the following code at the IDL command line:

```
   IIMAGE, BYTSCL(img LT 140)
```

In many images, the pixels have values that are only a small subrange of the possible values. By spreading the distribution so that each range of pixel values contains an approximately equal number of members, the information content of the display is maximized. In IDL, the HIST_EQUAL function performs this redistribution on an array.

To display a histogram-equalized version of image.tif, enter the following code at the IDL command line:

```
IIMAGE, HIST_EQUAL(img)
```

Smoothing and Sharpening

Images can be rapidly smoothed to soften edges or compensate for random noise in an image using IDL's Smooth filter. The Smooth filter performs an equally weighted smoothing using a square neighborhood of an arbitrary odd width, as shown below.

To smooth an image:

1. In the iImage tool, select **File → Open**, and select image.tif from the examples\data subdirectory of your IDL installation.

2. Click **Open**, and the file is displayed in the iImage tool.

3. Select **Operations → Filter → Smooth**.

    The **Smooth** dialog appears.

4. In the **Width** box, enter the value 7. This creates a 7 x 7 pixel-square smoothing area.

    The images at the bottom of the dialog show the displayed file before and after the filter is applied. The image shown at right is the smoothed image.

5. Click **OK**.
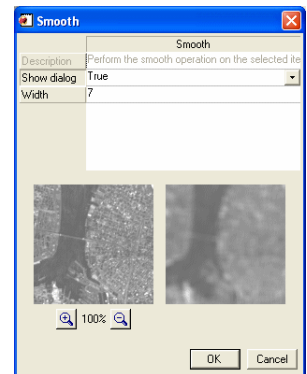
Unsharp Masking

The previous image looks a bit blurry because it contains only the low-frequency components of the original image. Often, an image needs to be sharpened so that edges or high spatial frequency components of the image are enhanced. One way to sharpen an image is to subtract a smoothed image containing only low-frequency components from the original image. This technique is called *unsharp masking*.

To unsharp mask an image:

1. Using the smoothed `image.tif` file used in the previous example, select
   Operations → Filter → Unsharp Mask.

   The **Unsharp Mask** dialog displays.

2. In the **Radius in Pixels** box, enter the value 7.

   The images at the bottom of the dialog show the displayed file before and after the filter is applied.

3. Click **OK**.
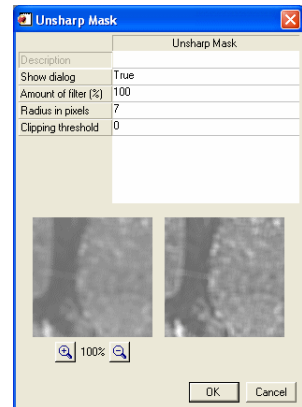
Sharpening Images with Differentiation

IDL has other built-in sharpening filters that use differentiation to sharpen images. The Roberts filter is one of these, and returns the Roberts gradient of an image.

To apply the Roberts filter to an image:

1. Select **File → Open**, and select image.tif from the examples\data subdirectory of your IDL installation.

2. Click **Open**.

    The file is displayed in the iImage tool.

3. Select **Operations → Filter → Roberts**.

    The Roberts filter is applied to the displayed image.

Another commonly-used gradient operation is the Sobel filter. IDL's Sobel filter operates over a 3 x 3 pixel region, making it less sensitive to noise than some other methods.

To apply the Sobel filter to an image:

1. Select **File → Open**, and select image.tif from the examples\data subdirectory of your IDL installation.

2. Click **Open**.

    The file is displayed in the iImage tool.

3. Select **Operations → Filter → Sobel**.

    The Sobel filter is applied to the displayed image.
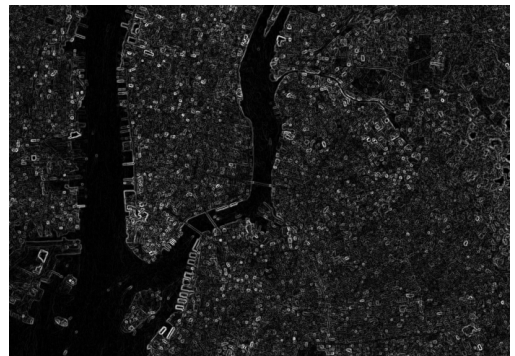
Loading Alternate Color Tables

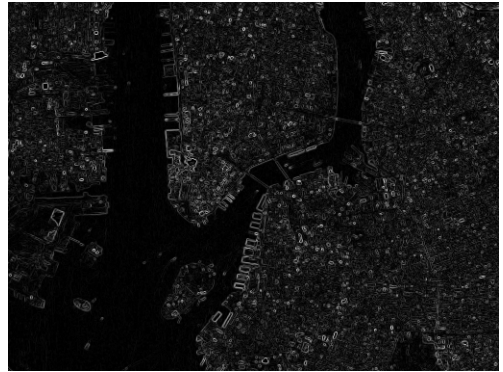Try loading some of the predefined IDL color tables to increase the contrast of the image.

1. After loading an image into the iImage tool, click **Edit Palette** (located on the **Image** tab).

    The **Palette Editor** dialog is displayed.

2. Click the **Load Predefined...** menu at the bottom of the dialog, and select a color table menu item.

    The loaded image will immediately incorporate the new color table. Go ahead and play with different color tables to observe their effect on the image.

3. When you are finished experimenting with different color tables, select the first color table in the menu, **B-W Linear** (the original black and white color table you have been working with), and click **OK**.

Cropping Images

To crop an image:

1. Select **File → Open**, and select image.tif from the examples\data subdirectory of your IDL installation.
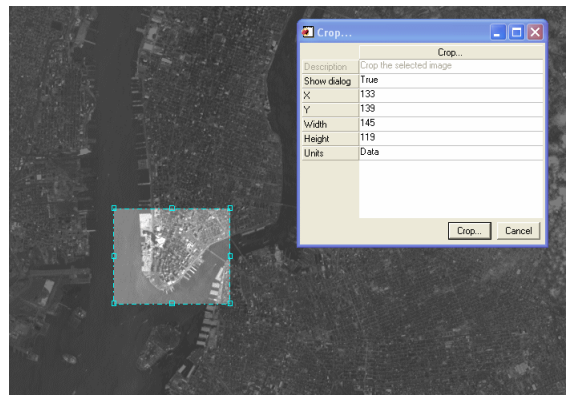
2. Click **Open**.

    The file is displayed in the iImage tool.

3. Select **Operations → Crop**.

    The **Crop** dialog displays.

4. Click on the image, and drag the box around the tip of the peninsula (actually Manhattan island).

5. On the **Crop** dialog, click **Crop**.

    The cropped portion of the original image is displayed
    (the lower tip of Manhattan, in this case).

You may also crop an image directly using the toolbar:

1. Click the **Crop** button 口 on the toolbar.

2. Click on the image and drag the box around the peninsula.

3. Double-click inside the box.

    The cropped portion of the original image is displayed.

Rotating Images

You can easily flip or rotate in image in the iImage tool.

To rotate an image 90 degrees clockwise:

1. If not already loaded, load the image.tif file and
    crop Manhattan island (the procedure is explained in
    "Cropping Images" on page 80).

2. Select **Operations → Rotate or Flip → Rotate Right**.

    The rotated image is displayed.

Extracting Profiles

The Line Profile tool plots image pixel values from
a line drawn over your image. The resulting 2-D
plot is displayed in a new iPlot window.

To create a line profile plot of an image:

1. On the iImage tool, select **File → Open**,
    and select image.tif from the
    examples\data subdirectory of your IDL
    installation.

2. Click **Open**, and the file is displayed in the iImage tool.

3. On the iImage toolbar, click the **Line Profile** ⌐ button.

4. Position the mouse pointer over the spot on the image where you want to start the line, and click.

5. Drag the pointer to the end point of your line, and release the mouse button.

    A new plot window displays showing a plot of the image pixel values that fall along the line.
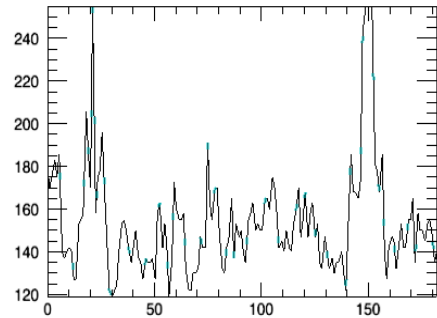
You can move the line around the image or change the endpoints, and the plot window continuously updates.

Displaying Images with Direct Graphics

The following sections show examples of reading and displaying image data using IDL's Direct graphics system. Working with images using Direct graphics is not as convenient as using the iImage tool, but may be desirable if you are incorporating images into a larger widget-based application, or if you need to programatically create a large number of processed images.

For a brief description of the IDL Direct graphics routines for displaying and manipulating images, refer to the *Direct graphics* and *Image Processing* sections of the *Functional List of IDL Routines* topic, found in the Online Help.

Displaying Images

Before processing an image, we must import the image into IDL.

For this example, we will continue to use the `image.tif` file.

1. Read the file by entering the following code at the IDL command line:

```
MYIMAGE=READ_TIFF(FILEPATH('image.tif', $
    SUBDIRECTORY=['examples', 'data']))
```

Using the IDL command line, you can view an image with two different routines. The TV procedure writes an array to the display in its original form. The TVSCL procedure displays the image and scales the color values so that all of the table colors are used (up to 256 colors).

2. Since we are using Direct graphics, tell IDL to use a maximum of 256 colors and load a simple grayscale color map.

```
DEVICE, RETAIN=2, DECOMPOSED=0
LOADCT, 0
```

3. Display the image with the TV procedure:

```
TV, MYIMAGE
```

4. Display the color-scaled image
    with the TVSCL procedure:

    ```
    TVSCL, MYIMAGE
    ```

5. Dismiss the graphics windows by
    clicking in the window's close
    icon or by entering WDELETE at
    the command line:

    ```
    WDELETE
    ```

Resizing Images

The REBIN function makes it easy to resize a vector or array to new dimensions. The
supplied dimensions must be proportionate (that is, integral multiples or factors) to
the dimensions of the original image. Since the original image array is 768 by 512,
we need to determine the correct dimensions of the resized image. To resize the
image to half the original size, simply take half of the array's original dimensions.

1. Create a new image with new
    dimensions using the REBIN
    function:

    ```
    NEWIMAGE=REBIN(MYIMAGE,384,256)
    ```

2. Display the image:

    ```
    TV, NEWIMAGE
    ```

Chapter 8
Volume Visualization

This chapter describes the following topics:

IDL and Volume Visualization

IDL can be used to visualize multi-dimensional volume data sets either at the command line or using the iVolume tool. Given a 3-D grid of density measurements, IDL can display a shaded surface representation of a constant-density surface (also called an isosurface). For example, in medical imaging applications, a series of 2-D images can be created by computed tomography or magnetic resonance imaging. When stacked, these images create a grid of density measurements that can be contoured to display the surfaces of anatomical structures.

This chapter introduces the iVolume tool for interactive exploration of volume data. It also presents some techniques for exploring volume data using IDL's Direct graphics routines.

For additional information on the iVolume tool's powerful capabilities for creating and manipulating volumes, see the *Working with Volumes* topic in the IDL Online Help. For a list of volume-related routines, see the "Volume Visualization" section of the *3D Visualization* topic in the IDL Online Help. See the *Creating Volume Objects* topic in the IDL Online Help for information on volume objects.

Volume Rendering with iVolume

The interactive iVolume tool allows you great flexibility in manipulating and visualizing true volume data.

Displaying a Volume

Here is a simple example of one way to visualize a volume using the iVolume tool.

1. At the IDL command line, read some volume data with the following commands:

    ```
    file = FILEPATH('head.dat', SUBDIRECTORY = ['examples', 'data'])
    head_data = READ_BINARY(file, DATA_DIMS = [80, 100, 57])
    ```

2. Now invoke the iVolume tool to visualize the volume:

    ```
    iVolume, head_data
    ```


Volume data does not automatically appear in the tool window since volume rendering can be a time consuming operation. To render the volume, click the **Render** button on the tool. *Auto-render* is available but not set by default since it can slow down your iTool session if you have simultaneous operations.
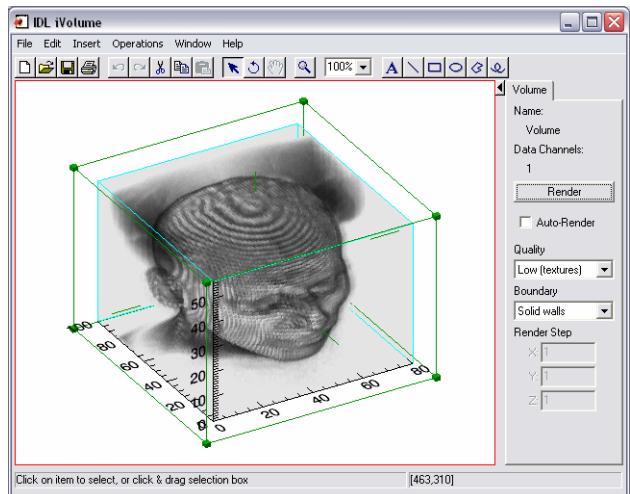
Note

Volume data is only rendered if the volume dataspace is selected. Thus, if you have multiple volumes in your dataspace, auto-render must be turned on in order to render both simultaneously.

Volume Rendering Quality

The volume can be rendered in two quality modes:

- **Low** — Done with a stack of 2D texture-mapped semi-transparent polygons. The polygons are oriented so that the flat sides face the viewer as directly as possible. On most systems, Low-quality mode renders faster than High-quality mode, but not as accurately.

- **High** — Done with the IDLgrVolume ray-casting volume renderer. This quality mode is CPU-intensive and will usually take much longer than the Low-quality mode.

Volume Boundary

Since the volume is not always rendered, iVolume draws volume boundaries to help you locate and select the volume. You can select the volume by selecting on the boundaries, without rendering the volume. By default, iVolume draws a translucent solid cube. You can also select a wire frame, or no boundaries.
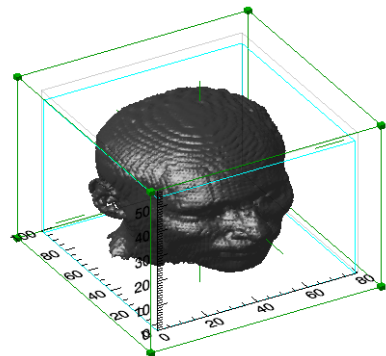
Note ───

Use caution when turning off boundaries. If boundaries are turned off and Auto-Render is also off, you may need to use the Visualization Browser to select the volume again.

───

Displaying an Isosurface

An *isosurface* is a set of points in a three-dimensional array that have the same value. In volume data, an isosurface generally defines a structure of some sort. To display an isosurface using the iVolume tool:

1. Click on the volume data to select it.

2. Select
   Operations → Volume → Isosurface
   from the iVolume menu.

3. Select the isosurface value and quality using the **Isosurface Value Selector** dialog. (Choose an isosurface value of 50 for a good result in this example.) Click **OK**.

Note

If you have checked the **Auto-Render** checkbox on the Volume tab, both the isosurface and the original volume will be rendered together. To see only the isosurface, uncheck the **Auto-Render** checkbox and click in the iVolume window.

Displaying Image Planes

An *image plane* is a two-dimensional slice taken through a three-dimensional volume. When presented as an image, image planes allow you to look at structures inside the volume.

To view an image plane:

1. Click on the volume data to select it. Make sure the **Auto-Render** checkbox on the Volume tab is unchecked.

2. Select **Operations → Volume → Image plane** from the iVolume menu.

3. Click on the image plane and drag back and forth to move the plane across the volume. To change the orientation of the image plane, double-click to display the image plane's property sheet, then select X, Y, or Z from the Orientation field.
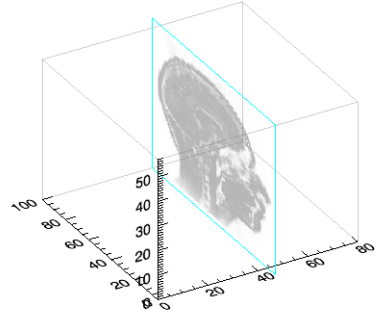
You can also display the image slice in an iImage tool:

1. Click on the image plane to select it.

2. Select **Operations → Image Plane → Launch iImage** from the iVolume menu.

A new iImage tool is created to contain the image. Moving the image plane or changing its orientation in the iVolume tool automatically updates the image displayed in the iImage tool.

For much more information on working with the iVolume tool, see the *Working with Volumes* topic in the IDL Online Help.

Volume Rendering with Direct Graphics

Visualizing volume data using IDL's Direct graphics routines (as opposed to the iTools) requires some additional work, but may be desirable if you are incorporating the volume visualization into a larger widget-based application. The following sections will guide you through the process of setting up a volume visualization using Direct graphics routines.

3-D Transformations in Direct Graphics

When creating three-dimensional plots (surface or volume visualizations, for example) using IDL Direct graphics, you must apply a three-dimensional transformation matrix to the data before display. The transformation applies a specified translation, rotation, and scaling to the three-dimensional data array before displaying it on the two-dimensional computer screen.

Three-dimensional transformations are especially important when using the POLYSHADE routine to display volume data. Unless the transformation is set up so that the entire volume is visible, the volume will not be rendered correctly. Once a 3-D transformation has been established, most IDL plotting routines can be made to use it by including the T3D keyword.

There are a number of ways to set up a transformation matrix in IDL:

- Modify the transformation matrix stored in the IDL system variable !P.T directly. This method is rather difficult, because you have to figure out the transformation yourself. More information about the transformation matrix can be found in the *Coordinate Conversions* topic in the IDL Online Help.

- Use the T3D, SCALE3, or SURFR routines to modify the !P.T transformation matrix.

- Use the SURFACE or SHADE_SURF routines to display some data. These routines calculate a transformation matrix based on the data you supply and update the !P.T system variable automatically.

In the following example, we will use the SCALE3 routine to update the !P.T transformation matrix.

Displaying an Isosurface

Two IDL routines, SHADE_VOLUME and POLYSHADE, are used together to create and display an isosurface. SHADE_VOLUME generates a list of polygons that define a 3-D surface given a volume data set and a contour (or *density*) level. The

function POLYSHADE creates a shaded-surface representation of the isosurface defined by those polygons.

Like many other IDL commands, POLYSHADE accepts the T3D keyword that makes POLYSHADE use a user-defined 3D transformation. Before you can use POLYSHADE to render the final image, you need to set up an appropriate three-dimensional transformation. The XRANGE, YRANGE, and ZRANGE keywords accept two-element vectors, representing the minimum and maximum axis values, as arguments. POLYSHADE returns an image based upon the list of vertices, V, and list of polygons, P. The T3D keyword tells POLYSHADE to use the previously-defined 3D transformation. The TV procedure displays the shaded-surface image.

Enter the following lines at the IDL command prompt:

1. If you created the `data` variable in the previous section, you can skip this step. If you have not yet created a `head_data` variable, read some volume data with the following commands:

    ```
    file = FILEPATH('head.dat', SUBDIRECTORY = ['examples', 'data'])
    head_data = READ_BINARY(file, DATA_DIMS = [80, 100, 57])
    ```

2. Since we are using Direct graphics, tell IDL to use a maximum of 256 colors. Load a simple grayscale colormap.

    ```
    DEVICE, RETAIN=2, DECOMPOSED=0
    LOADCT, 0
    ```

3. Create the polygons and vertices that define the isosurface with a value of 50. Return the vertices in the variable V and the polygons in the variable P:

    ```
    SHADE_VOLUME, head_data, 50, V, P, /LOW
    ```
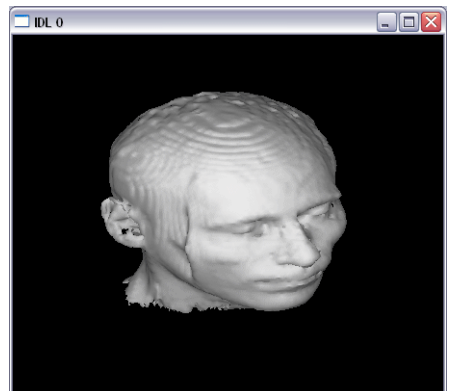
4. Set up an appropriate 3-D transformation matrix using the SCALE3 procedure:

    ```
    SCALE3, XRANGE=[0,80], YRANGE=[0,100], ZRANGE=[0,57]
    ```

5. Display a shaded-surface representation of the previously generated arrays of vertices and polygons:

    ```
    TV, POLYSHADE(V, P, /T3D)
    ```


This is the same isosurface created using the iVolume tool in "Displaying an Isosurface" on page 88.

Displaying an Image Plane

To display an image plane taken from volume data, use IDL's array indexing syntax to specify a "slice" through the three-dimensional data array.

1. The `head_data` array is 80 x 100 x 57 elements; to extract an X-Y slice from this array roughly in the middle:


   ```
   head_slice = head_data[40,*,*]
   ```

   This command creates a new variable named `head_slice` containing a 1 x 100 x 57 element array.

2. Reformat the 1 x 100 x 57 element array as a two-dimensional array with 100 x 57 elements:

   ```
   head_slice = REFORM(head_slice)
   ```
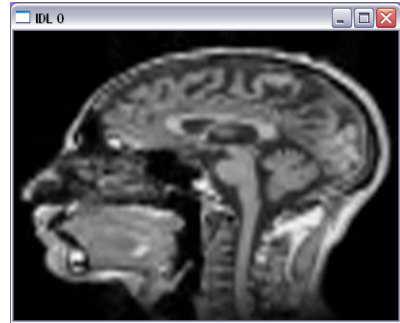
3. Resize the array to 500 x 400 elements, using cubic interpolation:
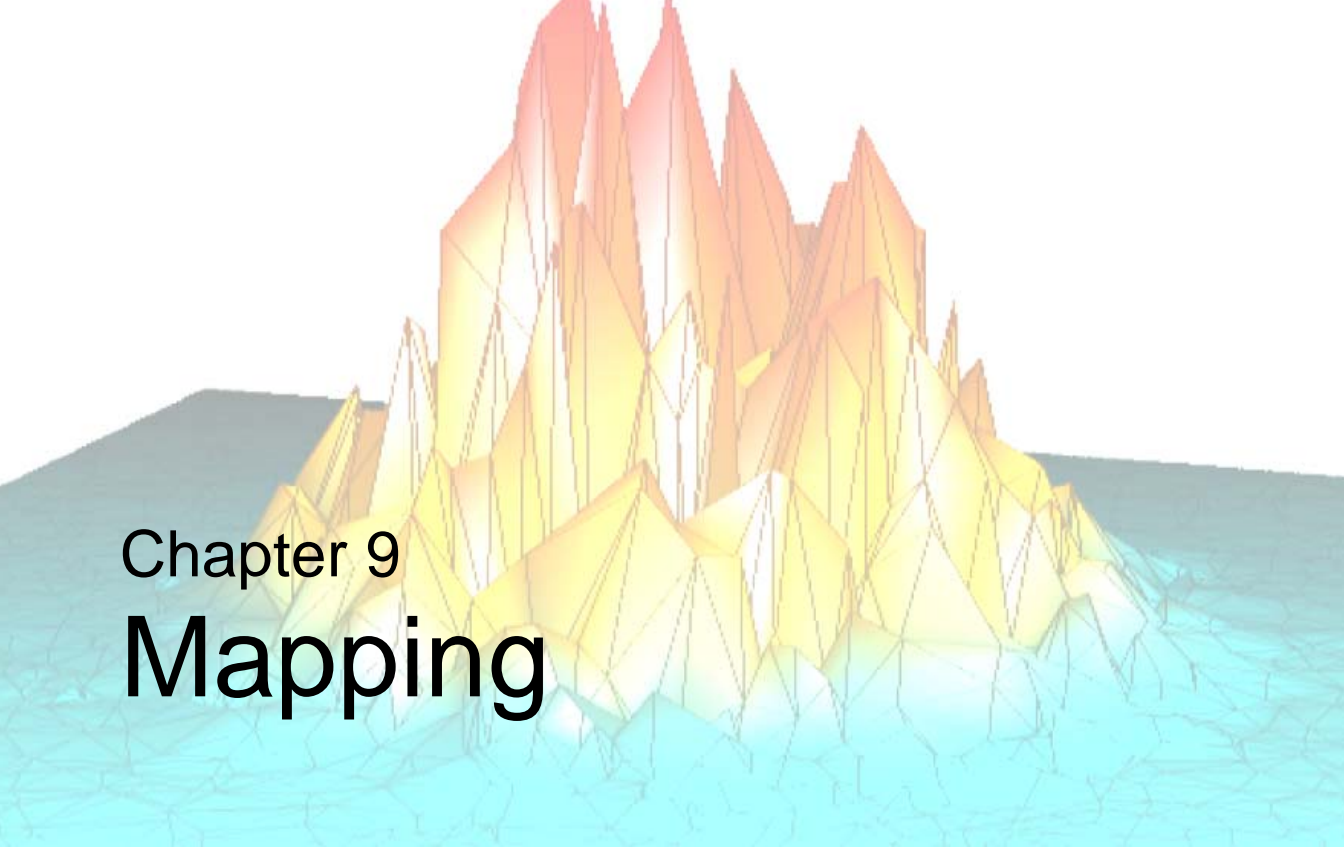
   ```
   head_slice = CONGRID(head_slice, 500, 400, CUBIC=-0.7)
   ```

4. Display the image:

   ```
   TV, head_slice
   ```

This is essentially the same process used by the iVolume tool in "Displaying Image Planes" on page 89.

Chapter 9
Mapping

This chapter describes the following topics:

IDL and Mapping

IDL's mapping facilities allow you to plot data over different projections of the globe. This chapter shows how to display various map projections and plot data over them.
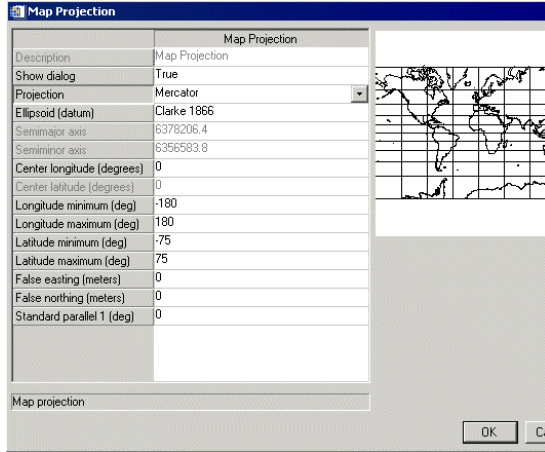
The first part of this chapter demonstrates the mapping capabilities of the iMap tool. The second part discusses how to work with direct graphics statements at the IDL command line to demonstrate IDL's interactive mapping capability.

Find more information on the IDL mapping routines in *Using IDL* and in the *IDL Reference Guide*. Also see "Working with Maps" (Chapter 15, *iTool User's Guide*).

Using the iMap Tool

The interactive iMap tool gives you great flexibility in manipulating and visualizing map data. The iMap tool also allows you to manipulate and edit individual components of a map display, such as rivers, lakes, or national or state boundaries.

We'll start by simply displaying a projection and then adding a map to the display:

1. To open an iMap window, type IMAP at the IDL command line. An empty iMap window appears.

2. To open and view a projection, select **Operations** → **Map Projection** from the iMap menu. This command opens the Map Projection dialog, shown here:

3. Choose a projection from the Projection pull-down list. A preview displays in the dialog. (The example here uses the Mercator projection.) Click **OK**.

4. The results in the iMap window show just the projection, without other mapping elements.

5. To show the continents against the projection, select **Insert** → **Map** → **Continents**. The continent outlines now display in the iMap window against the Mercator projection:
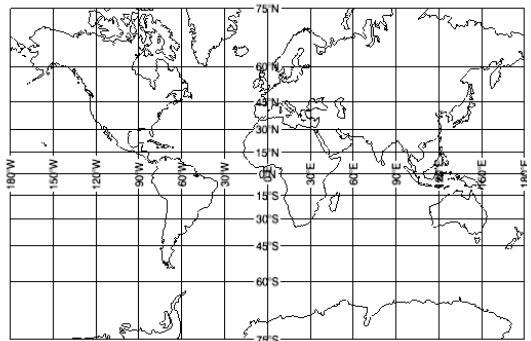

Figure 9-1: Mercator Projection

The iMap tool is a good way to familiarize yourself with projections. For example, with a basic global map such as this, you can easily change the projection by clicking **Edit Projection** in the Map tab on the right side of the iMap window. In the Map Projection window, select another projection and it displays in the preview window. This example shows the Mollweide projection.
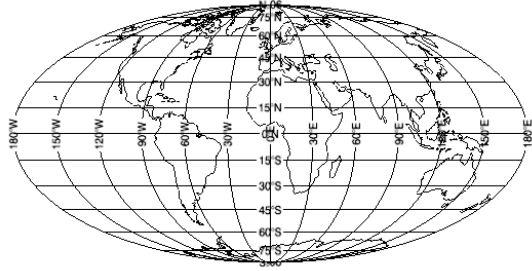

Figure 9-2: Mollweide Projection

To draw a map that looks more like a globe, use the Orthographic projection. Choose other projections to understand the differences in how they display maps.

See the *Map Projections* topic in the IDL Online Help for more on the map projections that IDL supports.

Modifying Map Data

The iMap tool allows you to make numerous modifications to maps. This section shows a simple exercise that guides you through a couple of easy modifications: adding available land features and changing map feature colors.

1. To open an iMap window, type IMAP at the IDL command line. An empty iMap window appears.

2. To open and view a global map, select **Insert → Map → Continents** from the iMap menu. A map of the world appears.

3. Now add countries to the global map. Select **Insert → Map → Countries (high res)** from the iMap menu. The countries of the world appear on the map.

4. Choose a part of the map that you want to fill with color. Right-click on that part and select **Properties** from the pop-up menu that appears. (This example changes the display properties of Australia.)

5. The IDL iMap Visualization Browser appears. Change any properties you like. (This example changes the line thickness and fill color properties, as shown in the dialog to the right.)
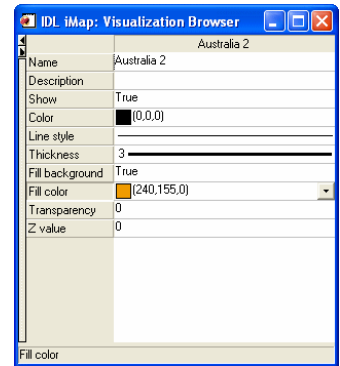
Changes are made to the map immediately, so you can see the results and change them to fit your needs. When you are finished with changes, close the IDL iMap Visualization Browser.
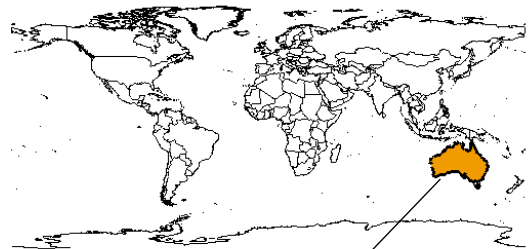
Result of modifications to Australia

Note

For more on mapping, see the *Working with Maps* topic in the IDL Online Help.

Fitting an Image to a Projection

IDL gives you the ability to open image data within a map projection through iMap. IDL warps the image to fit the map automatically. In this example, we'll use IDL's automatic capabilities to open an image containing 1 km resolution global land cover data. (Image data courtesy of Reto Stöckli, NASA/Goddard Space Flight Center.)
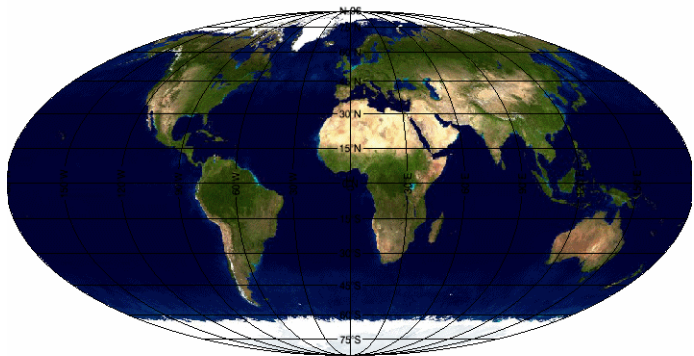
1. Open an iMap window by typing IMAP at the IDL command line.

2. Select **Operations →
Map Projection**. This command opens the Map Projection dialog, shown here:

3. Choose the Mollweide projection from the Projection pull-down list, then click **OK**.
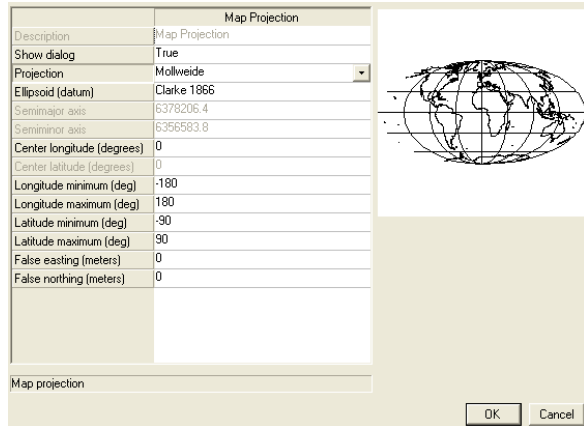
4. Now, open image data from the IDL examples files. Select **File → Open** and navigate to the IDL examples/data directory. Choose the file named Day.jpg.

   The IDL Map Register Image Step 1 dialog appears.

5. Click **Next** to accept the default settings. Step Two of the IDL Map Register Image dialog appears, displaying the default data that IDL uses to fit the image to the projection.

6. Click **Finish** to accept the default settings. In the iMap window, the image data display in the Mollweide projection:

See also "Warping Images to Maps" on page 104.

Plotting a Portion of the Globe

You do not always have to display the entire globe, you can view just a section of the globe by defining an area by latitude and longitude in the Map Panel. This example displays the North American Continent using the Miller Cylindrical projection.

1. Open an iMap window by typing IMAP at the IDL command line.

2. Select **Operations → Map Projection**. This command opens the Map Projection dialog, shown below.

3. Choose the Hammer projection from the Projection pull-down list.

4. Change the Longitude and Latitude settings as follows (and shown to the right):

   • Longitude minimum -130

   • Longitude maximum -70

   • Latitude minimum 10

   • Latitude maximum 55
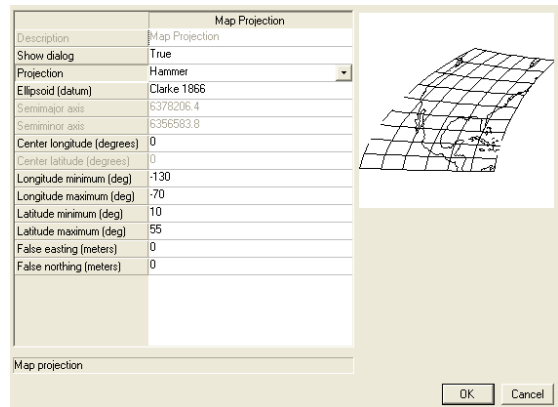
5. Click **OK**. The iMap window now displays a grid:
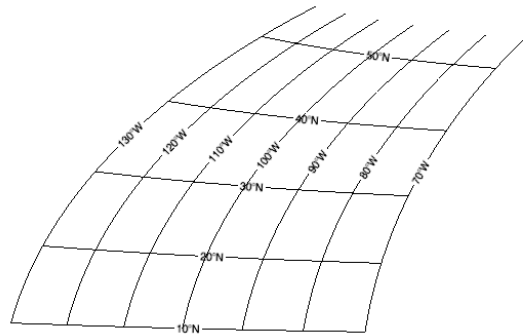
6. Now add a map to display on this grid. From the iMap menu, select **Insert → Map → Countries (high res)**. The countries within the specified area appear on the grid.

7. For better readability, you can change the way the grid and annotations appear. From the iMap menu, select **Window → Visualization Browser**.
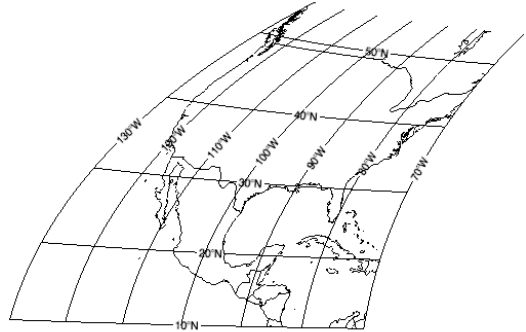
8. In the Visualization Browser, expand the tree view until you can see the Map Grid item. Double-click on **Map Grid** to expand the browser and display the Map Grid properties. Make the following changes:
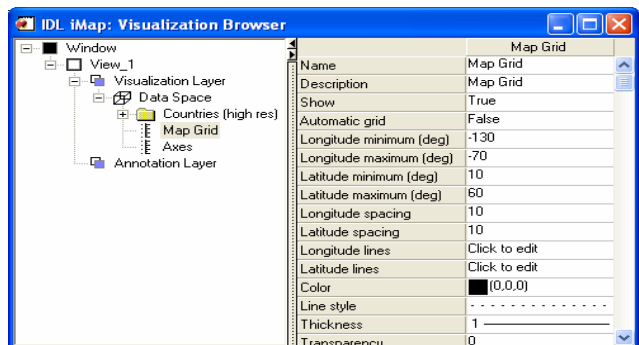
    • Automatic grid = False

    • Line style = dotted

    • Label position = 0.0

The iMap window now displays the map and grid projection with dotted lines and the grid annotations on the edges:
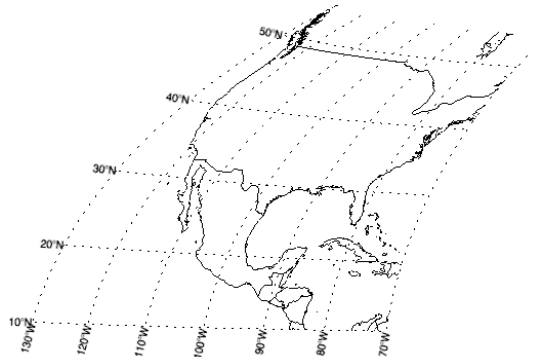
Note

You can add a title and other annotations using the Text tool on the iMap tool bar.

Plotting Data on Maps

You can annotate plots easily using the Direct graphics programming capabilities of IDL. Creating map displays using Direct graphics routines is not as convenient as using iMap, but may be desirable if you are incorporating maps into a larger, widget-based application.

To plot the location of the five cities as shown in the following figure, create three arrays for the data to plot: one each to hold latitude and longitude locations, and one to hold the names of the cities.

1. From the IDL command line, type the following command to create a five-element array of floating-point values representing latitudes in degrees North of zero.

   ```
   lats=[40.02,34.00,38.55,48.25,17.29]
   ```

2. The values in LONS are negative because they represent degrees West of zero longitude.

   ```
   lons=[-105.16,-119.40,-77.00,-114.21,-88.10]
   ```

3. Create a five-element array of string values. Text strings can be enclosed in either single or double quotes.

   ```
   cities=['Boulder, CO','Santa Cruz, CA',$
       'Washington, DC','Whitefish, MT','Belize, Belize']
   ```

4. Since we are using Direct graphics, tell IDL to use a maximum of 256 colors. Load a gray scale color table and set the background to white and the foreground to black:

   ```
   DEVICE, RETAIN=2, DECOMPOSED=0
   LOADCT, 0
   !P.BACKGROUND=255
   !P.COLOR=0
   ```

5. Draw a Mercator projection and define an area that encompasses the United States and Central America.

   ```
   MAP_SET, /MERCATOR, /GRID, /CONTINENT, LIMIT=[10,-130,60,-70]
   ```

6. Place a plotting symbol at the location of each city. The PSYM keyword creates diamond-shaped plotting symbols. SYMSIZE controls the size of the plotting symbols.


```
PLOTS, lons, lats, $
   PSYM=4, SYMSIZE=1.4, $
   COLOR=120
```

The result shows a window with the map projection of the area with the plotting symbols (shown here).
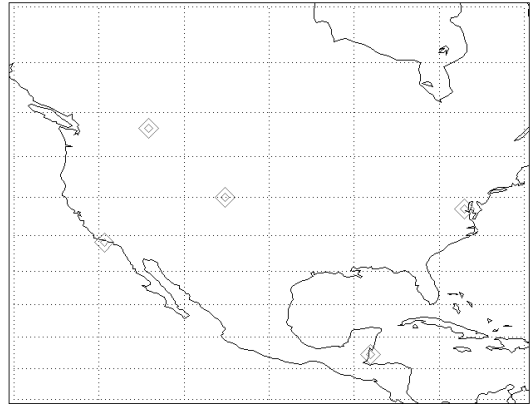
7. Place the names of the cities near their respective symbols. XYOUTS draws the characters for each element of the array CITIES at the corresponding location specified by the array elements of LONS and LATS. The CHARTHICK keyword controls the thickness of the text characters and the CHARSIZE keyword controls their size (1.0 is the default size). Setting the ALIGN keyword to 0.5 centers the city names over their corresponding data points.

```
XYOUTS, lons, lats, cities, COLOR=80, $
   CHARTHICK=2, CHARSIZE=1.25, ALIGN=0.5
```

Now the plotting symbols and city names display on the map:
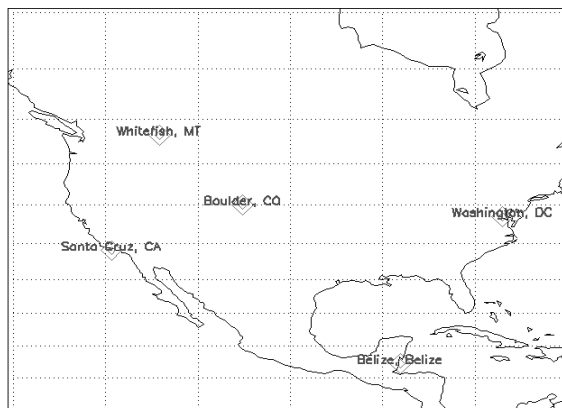
Reading Latitudes and Longitudes

If a map projection is displayed, IDL can return the position of the cursor over the map in latitude and longitude coordinates.

1. Enter the command:

   ```
   CURSOR, lon, lat & PRINT, lat, lon
   ```

   The CURSOR command reads the "X" and "Y" positions of the cursor when the mouse button is pressed and returns those values in the LON and LAT variables. Use the mouse to move the cursor over the map window and click on any point. The latitude and longitude of that point on the map are printed in the Output Log.

2. When you are finished with your map, close the graphics window.

Warping Images to Maps

Image data can also be displayed on maps using Direct graphics. The MAP_IMAGE function returns a warped version of an original image that can be displayed over a map projection. In this example, elevation data for the entire globe is displayed as an image with continent outlines and grid lines overlaid.

1. Define the map that you want to display, using WORLD as the variable in which to store the map data. Define the data dimensions as a 360 by 360 square array using the DATA_DIMS function. In the IDL command line, enter:

```
world = READ_BINARY(FILEPATH('worldelv.dat', $
    SUBDIRECTORY=['examples', 'data']), DATA_DIMS=[360,360])
```

2. Since we are using Direct graphics, tell IDL to use a maximum of 256 colors and load a color table.

```
DEVICE, RETAIN=2, DECOMPOSED=0
LOADCT, 26
```

3. View the data as an image using the variable WORLD that you defined above.

```
TV, world
```


The first column of data in this image corresponds to 0 degrees longitude. Because we'll use MAP_IMAGE later and it assumes that the first column of the image corresponds to -180 degrees, we'll use the SHIFT function on the data set before proceeding.

4. Shift the array 180 elements in the row direction and 0 elements in the column direction to make -180 degrees the first column in the array.
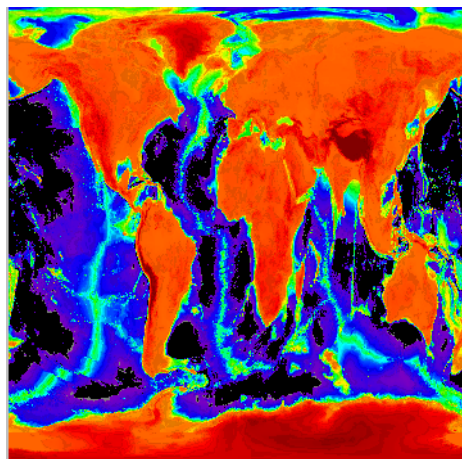
```
world = SHIFT(world, 180, 0)
```
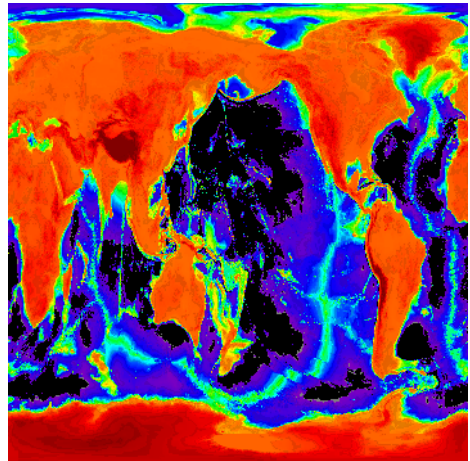
5. View the data as an image again, noting the difference made by the shift.

    ```
    TV, world
    ```

    From the image contained in the data, you can create a warped image to fit any of the available map projections. Define a map projection before using MAP_IMAGE, because this routine uses the currently defined map parameters.

6. Create a Mollweide projection with continents and gridlines.

    ```
    MAP_SET, /MOLLWEIDE, /CONT, /GRID, COLOR=100
    ```

7. Warp the image using bilinear interpolation using the BLIN command to smooth the warped image and save the result in the variable NEW.
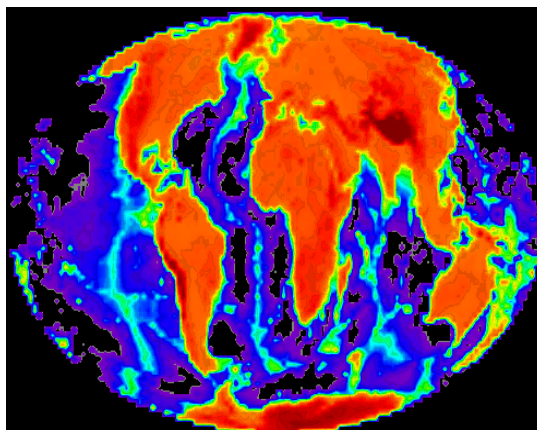
    ```
    new = MAP_IMAGE(world, sx, sy, /BILINEAR)
    ```

    The SX and SY output variables in the command above contain the X and Y starting positions for displaying the image.

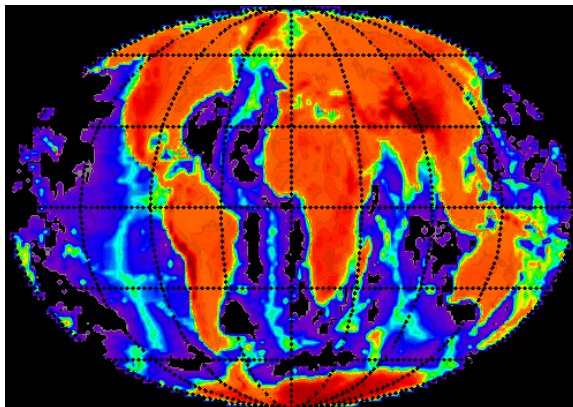8. Display the new image over the map:

    ```
    TV, new, sx, sy
    ```

    See the map in the previous figure and note that the warped image now displays over the existing continent and grid lines.

9. The continent outlines and thick grid lines can be displayed, as shown next, by
    entering:

    ```
    MAP_CONTINENTS
    MAP_GRID, GLINETHICK=3
    ```

Displaying Vector Data on a Map

You can use the iVector tool along with the iMap tool to easily add vector data in a map display.

1. Load some vector data representing global wind patterns:

   ```
   RESTORE, FILEPATH('globalwinds.dat', SUBDIR=['examples','data'])
   ```

   This command creates four variables — u, v, x, and y— that contain the vector data.

2. Create a map display of the globe, using the Mollweide projection:

   ```
   IMAP, MAP_PROJECTION='Equirectangular', LIMIT=[-35, -90, 35, 90]
   ```

3. Select **Insert → Map → Continents** to display the continental outlines.

4. Double-click on the continental outlines to display their property sheet. Set the Transparency value to zero and select a light grey fill color.

5. Finally launch iVector to create the vector display, coloring the wind vectors according to their magnitude:

   ```
   IVECTOR, u, v, x, y, /OVERPLOT, X_SUBSAMPLE=3
   ```

For additional information on creating vector displays, see the *Working with Vectors* and *IVECTOR* topics in the IDL Online Help.
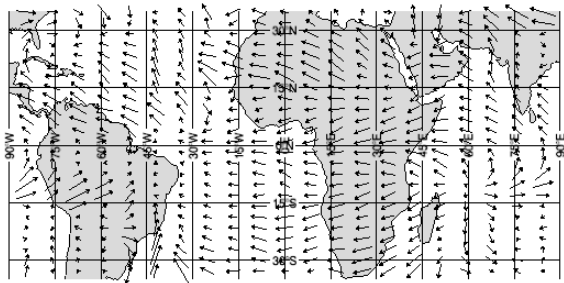
Chapter 10
Programming in IDL

This chapter describes the following topics:

Programming in IDL

IDL applications range from the simple (a short program entered at the IDL command line) to the complex (large programs with graphical user interfaces). Whether you are writing a small program to analyze a single data set or a large-scale application for commercial distribution, you'll need to understand the programming concepts used by the IDL language.

Programming in IDL feels familiar to developers already familiar with C, C++, or FORTRAN. Like these languages, IDL is a high-level programming language with similar syntax and operation.

While the programming environment is similar enough to make the transition easy, IDL's structure and tools make programming faster and more efficient. The following list outlines the benefits IDL offers over other programming languages:

- **Array operations**—using arrays creates more efficient code by eliminating the need for loops to perform operations on each data element.

- **Dynamic data types**—variables do not need to be explicitly typed because IDL determines the data type from the code context. Variables can be created or changed at any time, even within the same program.

- **IDL Workbench Development Environment**—provides the interactivity to speed up development, including chromacoding, coding tools, automatic compilation, and visual debugging tools. Programmers can quickly compile and run programs for testing and immediately view any problem areas that cause errors.

- **Interactive programming modes**—interactive mode allows you to run commands from the command line to immediately test code lines.

- **Graphical User Interface (GUI) Tools**—IDL provides several ways to develop GUI applications. These tools include:

  - Widget Programming—use IDL's library of widget tools to create simple controls such as buttons and sliders. Widget programming provides complete control over user interface design and functionality.

  - iTools—use IDL's built-in iTools to quickly visualize data with a minimum of programming, or create your own custom iTool application.

- **Built-in routines**—IDL provides a huge library of routines for graphical user interface (GUI) programming, numerical analysis, and data visualization.

- **Integrated development**—IDL is able to make calls to external programs written in other development languages, and provides the ability to call from external programs in IDL.

- **Distribution**—IDL provides tools that allow you to distribute your applications either as source code or in a compiled binary format called a SAVE file. Anyone with an IDL development license can execute IDL source code. If your colleagues or customers do not have an IDL development license, they can run most compiled IDL applications in the free IDL Virtual Machine. If your application uses features available only with an IDL license, you have the option of purchasing and distributing runtime licenses or embedding a license directly in the compiled application code.

This chapter gives a very brief introduction and overview into programming in IDL. To continue to learn to program in IDL, see the *Application Programming* manual and the documentation for specific routines in the *IDL Reference Guide*.

Types of IDL Programs

There are multiple ways of writing and executing programs within IDL. These involve varying levels of complexity and include $MAIN$ programs, procedures, and functions.

$Main$

You typically create a $MAIN$ program at the IDL command line when you have a few commands you want to run without creating a separate file. $MAIN$ programs are not explicitly named by a procedure (PRO) or function (FUNCTION) heading. They do require an END statement, just as procedures and functions do. Since they are not named, $MAIN$ programs cannot be called from other routines and cannot be passed arguments. $MAIN$ programs can be run from the command line using the `.RUN` command or saved in a file and run later.

When IDL encounters a main program either as the result of a .RUN command or in a text file, it compiles the code into the special program named $MAIN$ and immediately executes it. Afterwards, it can be executed again using the .GO command.

Note ───
Only one main program unit may exist within an IDL project window at any time.
──

Named Programs: Procedures and Functions

Procedures and functions are both modular programs that can be run individually and called from other programs. A program may include multiple procedures and functions and call as many other programs as necessary. Developers can choose whether to save many individual procedures and functions or to combine them in the same file. Reusing the same procedure or function for multiple programs can be a deciding factor in saving them separately. See the following sections for more information about how procedures and functions differ and when to use them.

Note ───
To view programs written in IDL, use the demo programs that come with IDL, found in the `\examples\demo\demosrc` of the IDL distribution. Open and view these programs to help you understand procedures and functions, but don't save any changes you make, as you and other users may use these programs for demonstration and training purposes.
──

Procedure

A procedure is a self-contained sequence of IDL statements that performs a well-defined task. A procedure is identified by a procedure definition statement (PRO <procedure_name>), where the procedure name is the name of the IDL statement you are creating. Parameters are named variables that are used in the procedure.

Use procedures when you are working on data "in place" or when no value is returned from the program. For example, a procedure could create a plot display on the screen but return no values back to IDL.

Function

A function is a self-contained sequence of IDL statements that performs a well-defined task and returns a value to the calling program unit when it is executed. All functions return a function value which is given as a parameter in the RETURN statement used to exit the function. A function is identified by a function definition statement (FUNCTION <function_name>), where the function name is the name of the IDL statement you are creating.

Use functions when you need easy access to a returned value, since functions create a new variable by default.

See *Overview of IDL Program Types* topic in the IDL Online Help for more information.

IDL Language Elements

The basic language elements of IDL are a bit different from other programming languages such as FORTRAN, C, and C++. These elements include dynamic data types, array operations, positional parameters (arguments), keywords, and automatic compilation. The following sections introduce the basics of these IDL language elements, along with how to avoid naming conflicts.

Variables and Data Types

IDL is different from other languages that require programmers to specifically designate a particular data type for each variable. IDL interprets variable types by their usage. This "loose" or dynamic data typing gives IDL flexibility and the ability to redefine variable data types at the command line or within programs. With this flexibility comes the need to keep track of the data types. IDL's built-in HELP procedure is an easy tool to use to return the data type of any variable, as shown in the following command-line example:

```
IDL> varvalue = 7.99
IDL> help, varvalue
VARVALUE        FLOAT     =        7.99000
```

Another example shows how the same variable is redefined as another data type:

```
IDL> varvalue = '7.99'
IDL> help, varvalue
VARVALUE        STRING    = '7.99'
```

Notice that since the variable value is within single quotes, IDL interprets it as a string. IDL does not hold the previous value of the variable in memory, so it can be changed at any time.

The three valid IDL variable organizations are scalars, arrays, and structures:

- Scalars—contain single values

- Arrays—contain multiple values arranged in an n-dimensional "grid."

- Structures—are collections of scalars, arrays, or other structures.

There are 16 variable types in IDL, which are shown in the following table:

| IDL Variable Types	
Undefined	Structure
Unsigned byte	Double precision complex
16-bit integer	Pointer heap variable
32-bit integer	Object reference heap variable
Single precision floating	Unsigned 16-bit integer
Double precision floating	Unsigned 32-bit integer
Single precision complex	64-bit integer
String	Unsigned 64-bit integer

Arguments

Arguments in IDL are positional parameters that pass information to a routine. In other words, each argument must be given in the order specified by the syntax of the routine. Some arguments are required, while others are optional, depending on the routine.

For example, the IDL system routine PLOT has two arguments: x and y. The arguments must be given in the correct order or the resulting plot's axes will be incorrect. If the y argument is not given, the routine plots y as a function of x, as shown in the following example:

```
IPLOT, EXP(FINDGEN(10))
```

The result of this command is the following plot:
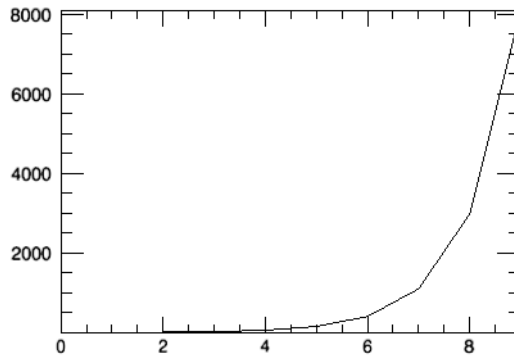
Keywords

Keywords are optional parameters that consist of keyword-value pairs. They can be passed to a routine in any order. If a keyword is not specified, the default value of that keyword is passed to the routine. A routine may have many available keywords to choose from. You can use as many or as few as you need.

Continuing with the PLOT example, we adds keywords to label the x and y axes:

```
IPLOT, EXP(FINDGEN(10)), XTITLE='Time', YTITLE='Velocity'
```
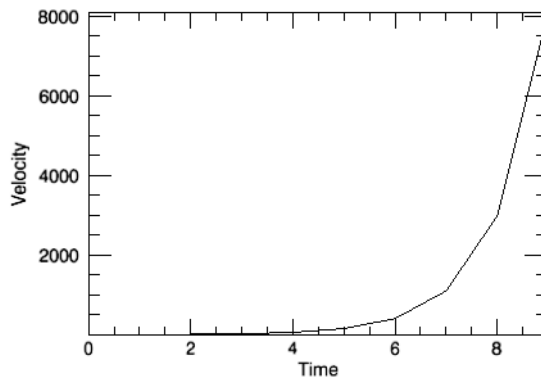
Automatic Compilation

IDL compiles routines any time it encounters a routine name, whether it is typed at the command line or called from another routine. If the routine is already in IDL's memory, IDL runs it. If the routine is not in memory, IDL searches each directory in the path definition for (`filename.pro`) and automatically compiles it. (For more on the IDL path, see the *IDL_PATH* topic in the IDL Online Help.)

Note ──

IDL routines all have specific names, which can conflict with user-written routines if those routines have the same name. When IDL encounters this conflict, the automatic compilation mechanism ignores the user-written routine. For more information, see the *Advice for Library Authors* topic in the IDL Online Help.

Code Written in Other Programming Languages

IDL allows you to incorporate routines written in other programming languages using the following methods:

- Call Java or COM objects and methods using the Import Bridge technology. See the *IDL Import Bridge* topic in the IDL Online Help for information.

- Call other types of external sharable object code (C or FORTRAN, for example). See the CALL_EXTERNAL, LINKIMAGE, MAKE_DLL routines, and the *External Development Overview* topic in the IDL Online Help for information.

- You can also export IDL objects for use by Java or COM programs. See the *IDL Export Bridge* topic in the IDL Online Help for information.

Arrays and Efficient Programming

IDL has been specifically designed to process arrays easily and naturally. You can get excellent performance in your applications by using the built-in array processing routines, which allow an operation to be performed on every element in an array, without having to explicitly create a loop. This functionality makes for simpler coding and faster computing. For more information, see the *Writing Efficient IDL Programs* topic in the IDL Online Help.

Using Operators on Arrays

IDL has a large number of different operators. In addition to the usual operators — addition, subtraction, multiplication, division, exponentiation, relations (EQ, NE, GT, etc.), and logical arithmetic (&&, ||, ~, AND, OR, NOT, and XOR) — other operators exist to find minima, maxima, select scalars and subarrays from arrays (subscripting), and to concatenate scalars and arrays to form new arrays.

All of IDL's operators can be applied to arrays as well as to scalars.

IDL's ability to perform operations directly on entire arrays makes it ideal for processing array data. For example, suppose you had an array A consisting of 100 floating point integers, and you wanted to create a corresponding array containing the absolute value of each array element. Most languages would require you to write a loop to create the new array. In IDL, the following single statement suffices:

```
B = ABS(A)
```

The array B is created as a 100-element array, each element of which contains the absolute value of the corresponding element in array A.

Similarly, multiplying each element of array C by the corresponding element of array D is simple:

```
E = C * D
```

See the *Arrays* and *Expressions and Operators* topics in the IDL Online help for additional details.

Subscripts

Subscripts retrieve or modify individual array elements, and are also referred to as array indices. In IDL subscripts, the first array index element is always zero. This is different from FORTRAN, where indices start by default with one. In a one-dimensional array, elements are numbered starting at 0 with the first element, 1 for the second element, and running to n - 1, the subscript of the last element.

You can use array subscripts to access one element, a range of elements, or a number of non-sequential elements in an array. You can also use subscripts to designate new values for array elements.

For example, the following expression gives the value of the seventh element of the variable `arr` (remember that array subscripts start at zero, not 1).

```
arr[6]
```

The next statement stores the number three at the seventh element of `arr`, with no changes to other array elements.

```
arr[6] = 3
```

Using Array Operators to Avoid IF Statements

Suppose you want to add all positive elements of array B to array A:

- Using a loop will be slow:

```
FOR I=0, (N-1)DO IF B[I]GT 0 THEN A[I]=A[I] + B[I]
```

- Fast way: Mask out negative elements using array operations.

```
A = A + (B GT 0) * B
```

- Faster way: Add B > 0

```
A = A + (B > 0)
```

When an IF statement appears in the middle of a loop with each element of an array in the conditional, the loop can often be eliminated by using logical array expressions.

Using Array Operators and the WHERE Function

In the example below, each element of the array C is set to the square-root of the corresponding element of array A if A[i] is positive; otherwise, C[i] is set to minus the square-root of the absolute value of A[i].

- Using an IF statement is slow:

```
FOR I=0,(N-1) DO IF A[I] LE 0 THEN C[I]=-SQRT(-A[I]) ELSE
   C[I]=SQRT(A[I])
```

- Fast way:

```
C = ((A GT 0) * 2 - 1 ) * SQRT(ABS(A))
```

The expression (A GT 0) has the value 1 if A[I] is positive and has the value 0 if A[I] is not. (A GT 0)* 2 - 1 is equal to +1 if A[I] is positive or -1 if

A[I] is negative, accomplishing the desired result without resorting to loops or IF statements.

Another method is to use the WHERE function to determine the subscripts of the negative elements of A and negate the corresponding elements of the result.

• Get subscripts of negative elements.

```
negs = WHERE(A LT 0)
```

• Take root of absolute value.

```
C = SQRT(ABS(A))
```

• Negate elements in C corresponding to negative elements in A.

```
C[negs] = -C[negs]
```

Using Vector and Array Operations

Whenever possible, vector and array data should be processed with IDL array operations instead of scalar operations in a loop. For example, consider the problem of flipping a 512×512 image. This problem arises because approximately half the available image display devices consider the origin to be the lower-left corner of the screen, while the other half recognize it as the upper-left corner.

Note ——————————————————————————————

The following example is for demonstration only. The IDL system variable !ORDER and corresponding features in the iTools and Direct graphics image display routines are easier to use and more efficient.

————————————————————————————————————

A programmer without experience in using IDL might be tempted to write the following nested loop structure to solve this problem:

```
FOR I = 0, 511 DO FOR J = 0, 255 DO BEGIN
```

• Temporarily save pixel:

```
TEMP=IMAGE[I, J]
```

• Exchange pixel in same column from corresponding row at bottom.

```
image[I, J] = image[I, 511 - J]
image[I, 511-J] = temp
ENDFOR
```

A more efficient approach to this problem capitalizes on IDL's ability to process arrays as a single entity.

- Enter at the IDL Command Line:

```
FOR J = 0, 255 DO BEGIN
```

- Temporarily save current row.

```
temp = image[*, J]
```

- Exchange row with corresponding row at bottom.

```
image[*, J] = image[*, 511-J]
image[*, 511-J] = temp
ENDFOR
```

At the cost of using twice as much memory, processing can be simplified even further by using the following statements:

- Get a second array to hold inverted copy.

```
image2 = BYTARR(512, 512)
```

- Copy the rows from the bottom up.

```
FOR J = 0, 511 DO image2[*, J] = image[*, 511-J]
```

- Even more efficient is the single line:

```
image2 = image[*, 511 - INDGEN(512)]
```

that reverses the array using subscript ranges and array-valued subscripts.

- Using the built-in ROTATE function:

```
image = ROTATE(image, 7)
```

This works because inverting the image is equivalent to transposing it and rotating it 270 degrees clockwise.

- Using the built-in REVERSE function:

```
image = REVERSE(image, 2)
```

IDL Programming Concepts and Tools

IDL's programming environment provides tools that help you organize and accelerate code development. The IDL workspace and projects provide the basic framework for IDL programming. The different programming tools include object and GUI programming, including the iTools library. These concepts are introduced in the following sections, along with how to distribute your IDL applications.

Workspace

The Workbench in IDL uses the concept of a *workspace*, where IDL stores all projects, folders, and files in a single directory. All projects reside in the workspace. You can choose where the workspace physically resides on your system, and you can create multiple workspaces, but only one can be open at a time.

Projects

An IDL project is a virtual collection of folders, files, and metadata. Projects are not required by IDL, but the benefits of saving programs in projects include cross-project searching and easy navigation. You can also use projects for customizing builds, version management, sharing, and resource organization.

In the IDL workflow, you first create a project using the Workbench and specify a location for it in the file system. Code files are then associated with the project.

Object Programming

Object-oriented programming blurs the lines between routines and the data that they act upon. The benefits of using object-oriented programming include reusable classes and more modular code (easier to find and fix errors). Object-oriented applications can also be easier to maintain and extend.

IDL began as a procedural language, but object-oriented programming was introduced in IDL 5.0. One of the driving reasons was to simplify 3D graphics capabilities (known as Object graphics in IDL). The IDL Object graphics system is a collection of pre-defined object classes that act as building blocks. To build a useful application, you must use several of these building blocks together. Compared to IDL's Direct graphics, object graphics are more complex, but produce robust, 3-D visualizations. Another difference is that object graphics are meant for application development rather than for command-line users.

For more information, see *The Basics of Using Objects in IDL* topic in the IDL Online Help.

Graphical User Interface (GUI) Programming

IDL provides several programming options for creating user interfaces. The following list shows the options in order from simplest to most complex:

- Command-line—using the IDL command line, you can display data in the IDL output log or direct graphics visualizations. For example, using statements such as PLOT, PRINT, and TV.

- iTool Interface—using an existing iTool allows you to quickly display data and manipulate images. Existing iTools include iPlot, iImage, iContour, iSurface, and iVolume.

- Custom Widget Interface—IDL provides a library of widget tools to create simple controls such as buttons and sliders. Using widgets offers you complete control over user interface design, but you must code all the underlying functionality. It is possible for more advanced programmers to create applications that combine iTools and widgets.

- Custom iTool—The most complex programming option is creating custom iTool interfaces. This option allows you to expand the capabilities and appearance of the standard iTools.

iTool Programming

The term iTools stands for intelligent tools, which are a collection of IDL applications that share a common framework. The iTools all have a graphical user interface (GUI) that allows you to program applications with custom toolbars, menus, buttons, etc. IDL provides several predefined iTools, and you can develop your own using iTool programming.

Programming in iTools uses the iTools Component Framework, which is a set of class files and utilities that help you create new tools or extend the existing iTools.

For more information, see the *Creating an iTool* topic in the IDL Online Help.

Distributing Programs

Once you have completed your application, you can quickly and easily create a distribution of your software product. See the *Running and Building Projects* topic in the IDL Online Help for information on packaging your application for distribution.

IDL Workbench Editor

The Editor is the IDL Workbench area where you create, view, and edit code. Within the Editor, you'll find tools that help you format, comment, test, and debug code. The Editor provides tools that help you create code faster and more efficiently than you could in a simple text editor.

Some of the features of the IDL Editor that are discussed in this chapter are:

- Writing Code—The Editor provides tools such as content assist, key bindings, and commenting code to help you speed up code development.

- Formatting Code—Tools such as syntax coloring and other formatting options help you quickly format the code for readability.

- Viewing and Finding Code—Features such as code folding, open declaration, and hover help allow you to quickly scan your code. Finding what you need is easy with the find and replace and parentheses matching tools.

- Organizing Code—The bookmark and task markers help you easily annotate and organize your code.

- Testing and Debugging Code—The debugging features of the IDL Workbench include setting breakpoints in the Editor.

- Code Versioning—The Editor allows you to compare the current version of a file with previous versions to restore any necessary revisions.

For more information, see the *IDL Editor Tips and Tricks* topic in the IDL Online Help.

Executing a Simple IDL Program

To show IDL's programming capabilities, the following program example uses the iVolume iTool to display volume data. This example uses Black Hole volume data provided by the University of North Carolina.

1. From the IDL Workbench, open a new IDL Editor window by selecting **File → New → IDL Source File**.

2. Type (or copy) the following lines of code into the new Editor window to form a program:

```
PRO my_ivolume, _EXTRA=_extra

; Set the variable fname to the black hole volume data file
fname = FILEPATH('cduskcD1400.sav', SUBDIR=['examples', 'data'])
RESTORE, fname
; load a color table and supress the color table message using
; the keyword /SILENT
LOADCT, 15, /SILENT
; Return the Red, Green, Blue values from the
; internal color tables
; to the variables r, g, b
TVLCT, r, g, b, /GET
; Display the data using the iVolume iTool
  IVOLUME, density, RGB_TABLE0=[[r],[g],[b]], $
    /AUTO_RENDER, /NO_SAVEPROMPT
END
```

Note ───

Semicolons (;) in IDL code are indicators of the beginning of comment lines, which explain what the actual code lines are doing and/or help you understand your code (while being ignored by IDL itself).

───

Note ───

The dollar sign ($) at the end of a line is the IDL continuation character. It allows you to enter long IDL commands as multiple lines.

───

Saving, Compiling, and Running

To view the program at work, IDL requires a few additional steps:

1. Save the file as my_ivolume.pro by selecting **File → Save As** and then entering "my_ivolume.pro".

2. Run the program by selecting **Run → Run my_ivolume.pro** (IDL automatically compiles the program if it is in the IDL path).

The resulting iVolume window displays the following image:
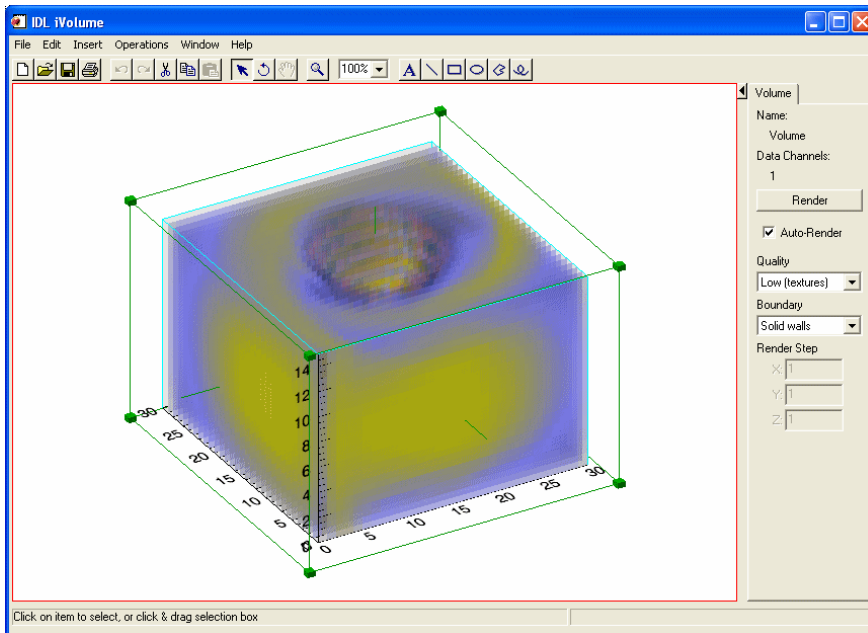
Note ───

If your program encounters an error while executing, be sure to check your code for typographical errors.

───

Debugging

Debugging is the process of finding and correcting errors or undesirable behavior in your code. The IDL Workbench supplies tools that let you monitor the execution of your program, stop and re-start execution, step through the program one statement at a time, and inspect or change the values of variables.

The debugging process begins when IDL temporarily stops execution before it reaches the end of a program. There are two ways this can happen: when IDL encounters an error that forces either compilation or execution to halt, or when IDL encounters a breakpoint you have set in the code to cause a temporary halt. Note that not every error in your code will cause IDL to halt execution; many problems involve code that runs correctly to completion but creates incorrect results.

For complete information, see the *Debugging and Error-Handling* topic in the IDL Online Help.

Chapter 11
User Interfaces in IDL

This chapter describes the following topics:

User Interface Options in IDL

If you create an application that requires user interaction, you will need to supply a *user interface*. IDL gives you several options for supplying an interface. In order of increasing complexity, you can use any of the following:

- Non-graphical Interface — You can use the IDL command line as a non-graphical user interface to request simple textual user input, display Direct graphics visualizations, or display data in the IDL Console view. See "Non-Graphical User Interfaces" on page 131 for more information.

- Existing iTool Interface — You can use an existing iTool to provide quick data display and manipulation capabilities for image, plot, surface, volume and map data. See "Existing iTool Interfaces" on page 132 for more information.

- Graphical Interface — You can use IDL widgets to build a complete graphical interface of your own design. Creating a user interface from scratch (as opposed to using the iTools framework) gives you complete control over the appearance and functionality of the interface, but you must code all underlying functionality. You can also create a hybrid widget-iTool application, but this requires additional programming expertise. See "Graphical Interfaces with IDL Widgets" on page 133 for more information.

- Custom iTool — You can create a custom iTool interface that allows you to expand on the capabilities of the standard iTool design and configure the appearance of your iTool. This option requires the most programming expertise. It is likely that one of the other options will meet the needs of the majority of applications, but this level of customization is available for those who require it. See "Custom iTool Interfaces" on page 136 for more information.

Non-Graphical User Interfaces

If your application requires little interaction from the user and runs in a full IDL installation (that is, if your user has a licensed copy of IDL and can use either the IDL Workbench or the IDL command line version), you may not need to create a graphical user interface at all. For example, if you have created a simple program that requires the user to enter a small number of data values and returns a numerical result, you may not want the overhead of a graphical interface.

The READ routine allows you to use the IDL command line to prompt the user for values, providing a record of both the prompts and the values entered in the Console view. See the *READ/READF* topic in the IDL Online Help for details.

This kind of textual interface behaves identically on all systems that run IDL and requires very little programming, but it does require that a licensed version of IDL be available. Applications using graphical interfaces can run using a runtime IDL license or in the IDL Virtual Machine, neither of which requires that the application's end user have a full IDL license.
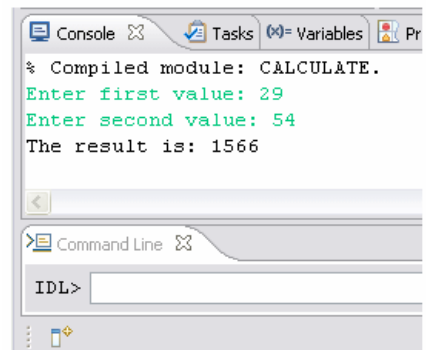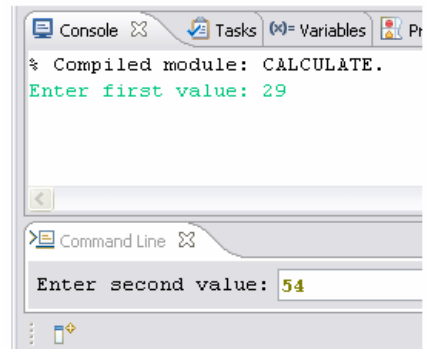
Existing iTool Interfaces

The IDL Intelligent Tools (iTools) are a set of interactive utilities that combine data analysis and visualization with the task of producing presentation quality graphics. Based on the IDL Object graphics system, the iTools are designed to help you get the most out of your data with minimal effort. They allow you to continue to benefit from the control of a programming language, while enjoying the convenience of a point-and-click environment.

Using an existing iTool user interface for data display and modification is the easiest way to allow your user to access, visualize, and modify plot, volume, surface, map, and image data.

The example exercises in this manual use the iTools extensively. Trying the examples and experimenting with the iTools should give you a good idea of whether an existing iTool can provide the interface your application needs. See *Introducing the IDL iTools* topic in the IDL Online Help for information on using the iTools.
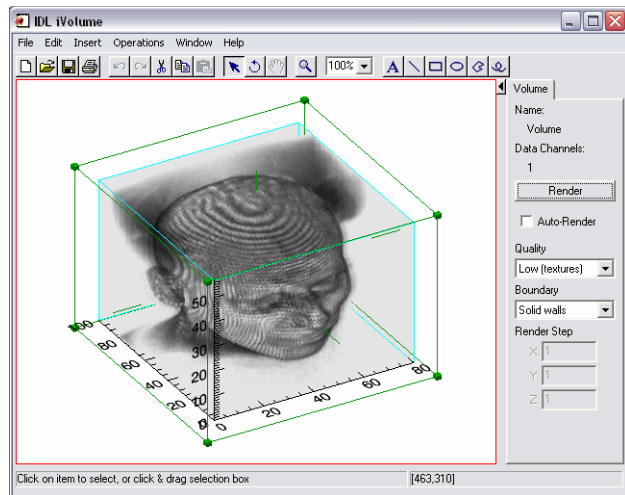
If you need functionality beyond that provided by an existing iTool, you can expand the functionality by adding:

- Custom operations or manipulators to standard visualization types

- Custom file writers or file readers

- Custom messages

Using an existing iTool lets you provide your users with a great deal of pre-built functionality. For information on expanding the iTool functionality mentioned above, see the *iTool Programming* topic in the IDL Online Help.

Graphical Interfaces with IDL Widgets

IDL allows you to construct and manipulate graphical user interfaces using *widgets*. Widgets are simple graphical objects such as pushbuttons or sliders that allow user interaction via a pointing device (usually a mouse) and a keyboard. Widget applications can be simple or complex; the IDL iTools are examples of sophisticated applications with a graphical user interface constructed from IDL widgets.

While creating an interface using IDL widgets is significantly simpler than building a similar interface using native window system graphical interface toolkits, the style of programming required is fundamentally different than in other IDL programs. A program written to be used from the IDL command line generally accepts its inputs when the program is invoked. The program then proceeds in a well-defined order to process those inputs and provide some output — a calculated value, a plot, an image, *etc*. In contrast, widget applications are *event-driven*.

In an event-driven system, the program creates an interface and then waits for messages (events) to be sent to it from the window system. Events are generated in response to user manipulation, such as pressing a button or moving a slider. The program responds to events by carrying out the action or computation specified by the programmer, and then waiting for the next event.

See "A Simple Widget Example" on page 134 for code that creates a very simple widget application that allows you to choose and display a JPEG image file from the IDL distribution.
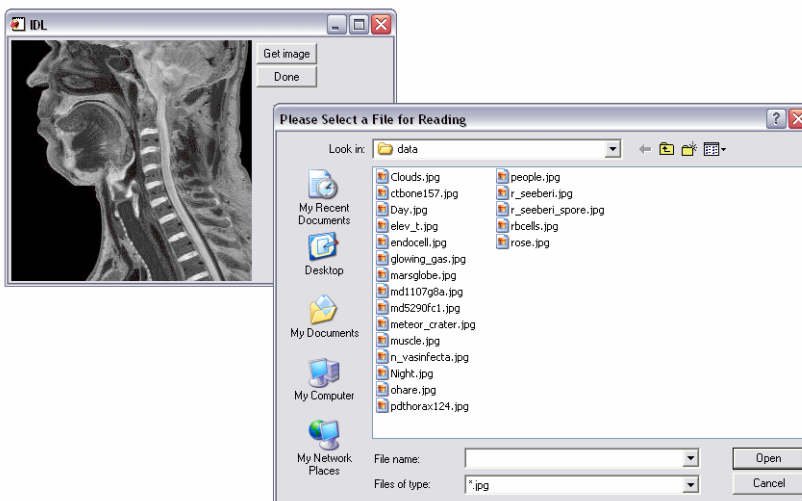
A Simple Widget Example

The following lines of IDL code create the simple application described in
"Graphical Interfaces with IDL Widgets" on page 133.

```
PRO simple_image_viewer_event, ev

    WIDGET_CONTROL, ev.TOP, GET_UVALUE=drawid
    WIDGET_CONTROL, ev.ID, GET_UVALUE=uval
    CASE uval OF
       'get_image' : BEGIN
          path = !DIR+'/examples/data'
          file = DIALOG_PICKFILE(PATH=path, /READ, $
             FILTER='*.jpg', /FIX_FILTER)
          image = READ_IMAGE(file)
          WSET, drawid
          ERASE
          IF (SIZE(image, /N_DIMENSIONS) EQ 3) THEN BEGIN
             TV, image, /TRUE
          ENDIF ELSE BEGIN
             TV, image
          ENDELSE
       END
       'done' : WIDGET_CONTROL, ev.top, /DESTROY
    ENDCASE

END

PRO simple_image_viewer

    main_base = WIDGET_BASE(/ROW, XSIZE=400, YSIZE=255)
    draw = WIDGET_DRAW(main_base, XSIZE=250, YSIZE=250)
    button_base = WIDGET_BASE(main_base, /COLUMN)
    button = WIDGET_BUTTON(button_base, VALUE='Get image', $
       UVALUE='get_image')
    button = widget_button(button_base, VALUE='Done', $
       UVALUE='done')

    WIDGET_CONTROL, main_base, /REALIZE
    WIDGET_CONTROL, draw, GET_VALUE=drawid
    WIDGET_CONTROL, main_base, SET_UVALUE=drawid

    XMANAGER, 'simple_image_viewer', main_base, /NO_BLOCK

END
```

It is beyond the scope of this manual to describe every detail of the
simple_image_viewer program. But as you can see, it requires fewer than 40 lines

of IDL code to create a program that allows you to quickly select and view the contents of an image file.

If you are interested in trying the `simple_image_viewer` program, do the following:

1. Open the IDL Workbench.

2. Select **File → New → IDL Source File** to create a new editor window.

3. Enter the code lines from the previous page in the editor window.

4. Select **File → Save** and then click **OK** in the **Save As** dialog that appears, accepting the default filename and location. (This saves your code in a file named `simple_image_viewer.pro` in your default IDL project directory.)

5. Select **Run → Run simple_image_viewer** or press F8. The program is executed and the graphical interface is displayed.

6. Click **Browse** to choose a JPEG image to be displayed. Click **Done** to dismiss the application.

For more on using the IDL Workbench to create programs, see "The IDL Workbench" on page 19. For more on creating graphical user interfaces using IDL widgets, see the *Creating Widget Applications* topic in the IDL Online Help.

Custom iTool Interfaces

Each of the standard iTools (such as the iPlot or iImage tools) have the same basic interface style. Beyond adding operations or manipulators, you can modify the existing iTool interface by adding:

- Modal dialogs, implemented through a user interface service

- iTool panels, which provide a set of controls that are attached to a visualization window and are always available

Beyond this, you have the option of modifying the standard iTool interface. Standard iTools are constructed of a number of compound widgets designed to work explicitly within the iTool architecture. You can modify the standard iTool interface by creating a custom iTool-widget interface, a hybrid tool that combines traditional widget functionality and iTool compound widgets. This requires knowledge of widget programming, how to create an iTool, how to create a UI service, and how to use the iTool compound widgets. For more information on the previous topics, see the *iTool Programming* topic in the IDL Online Help.

Index

E

F

G

H

I

J

K

L

M