Introduction to IDL

Mark Piper, Ph.D.
Barrett M Sather
Exelis Visual Information Solutions

Version 2014-03-12

Contacting Us

The Exelis VIS Educational Services Group offers a spectrum of standard courses for IDL and ENVI, ranging from courses designed for beginning users to those for experienced application developers. We can develop a course tailored to your needs with customized content, using your data. We teach classes monthly in our offices in Boulder, Colorado and Herndon, Virginia, as well as at various regional settings in the United States and around the world. We can come to you and teach a class at your work site.

The Exelis VIS Professional Services Group offers consulting services. We have years of experience successfully providing custom solutions on time and within budget for a wide range of organizations with myriad needs and goals.

If you would like more information about our services, or if you have difficulty using this manual or finding the course files, please contact us:

Exelis Visual Information Solutions
4990 Pearl East Circle
Boulder, CO 80301 USA
+1-303-786-9900
+1-303-786-9909 (fax)
www.exelisvis.com
training@exelisvis.com

Contents

1. About This Course 1

Manual organization 2
The course files 3
Starting IDL 4

2. A Tour of IDL 7

Overview 8
Scalars and arrays 8
Reading data from files 9
Line plots 10
Surface plots 11
Contour plots 12
Displaying and processing images 14
Exercises 15
References 16

3. IDL Basics 17

IDL directory structure 18
The IDL workbench 19
Working directory 24
Preferences 24

Search path 26
The IDL Help system 27
References 28

4. Line, Bar and Scatter Plots 29

Graphics routines 30
Reflectance spectra 30
Boulder temperature data 33
The sunspot cycle 38
Exercises 41
References 42

5. Data Structures 43

Variables 44
Data types 45
Arrays 48
Lists and hashes 53
Structures 54
Strings 56
Pointers 58
Objects 59
Exercises 59
References 60

6. Programming 63

Programs 64
Parameters 69
Calling mechanism 70
Operators 72
Control statements 75
Batch files 82
Timing with TIC and TOC 82
Programming tips 83
Exercises 83
References 83

7. Images 85

What is an image? 86
HST imagery of the Carina Nebula 86
Truecolor JPEG image 90
Landsat 7 ETM+ image 92
Exercises 94
References 95

8. File Access 97

File types: text and binary 98
File manipulation routines 98
IDL SAVE files 101
Standard file types 102
Reading text and binary files 105
Low-level file routines 106
Exercises 114
References 114

9. Surface and Contour Plots 115

Graphics routines 116
Spatial rainfall distribution 116
Digital terrain elevations 121
Exercises 127
References 127

10. Analysis 129

Introduction 130
Interpolation 130
Curve fitting 134
Signal processing 138
Image processing 140
Exercises 146
References 146

11. Map Projections 149

Map projections 150
Graphics routines 150
A simple map 151
Gridded ASOS temperatures 152
Landsat 7 ETM+ image, georeferenced 154
General map projection routines 156
Exercises 157
References 157

Glossary 159

Index 165

Chapter 1:
About This Course

This chapter gives an overview of the style and use of this manual. Instructions are also given for downloading and installing the course files.

Why waste time learning when ignorance is instantaneous?

—Hobbes

Manual organization 2
The course files 3
Starting IDL 4

Manual organization

This is the manual for *Introduction to IDL*, a three-day course that provides scientists, engineers and developers with a working knowledge of the IDL programming language. IDL, developed by Exelis Visual Information Solutions of Boulder, Colorado, is the ideal software for data analysis, visualization and cross-platform application development. Thousands of technical professionals use IDL every day to rapidly develop algorithms, interfaces and visualizations and to quickly crunch large numerical problems.

Through a mixture of lectures, instructor-led exercises and challenge problems, this course significantly shortens the startup time for learning IDL, increasing efficiency. The course is focused on using IDL for data exploration, visualization and analysis, with examples from astronomy, atmospheric science, remote sensing and medical imaging.

This course, the first in the sequence of IDL courses developed by Exelis VIS, is intended for new users of IDL; no knowledge of IDL nor prior programming experience is required.

Programming style

Unlike languages such as C/C++, Java and Python, there is no set style standard for IDL programming. This is a blessing and a curse: it allows a user to quickly create programs; however, others may find those programs difficult to read. Here is a table that describes the styles used in this manual.

Table 1-1: Style guidelines used in this manual.

Area	Style guidelines
IDL program code	IDL statements are case insensitive. Lower case is used in this manual for IDL programs. Code written in a program file or a change to existing code is written in **bold** type.
Command line code	Code to be typed by the user at the IDL command line is prefixed with **IDL>**.
Program units	When referenced in the text, IDL and user-written programs are capitalized and italicized; e.g., *PLOT*, *HELP*, *PRINT*.
Files	Files and directories on a computer are listed in boldface; e.g., **pwd.pro**, **introduction**.
Variables	When referenced in the text, variables are displayed with a fixed-width font, capitalized; e.g., `PSTATE`, `!PI`.
Keywords	When referenced in the text, keywords and reserved words are capitalized, e.g., XSIZE, FOR, PRO.
Comments	Comments are marked with a semicolon (;).
Spacing	Indentation and empty lines are used liberally to set off related sections of code.
Line continuations	A dollar sign $ at the end of a line indicates that the current statement is continued on the following line. The $ can appear anywhere a space is legal except within a string constant or between a function name and the first open parenthesis. Any number of continuation lines are allowed.

A more detailed style guide created by Michael Galloy (http://www.michaelgalloy.com) can be found in the course files.

The examples in this manual require a minimal amount of coding. When necessary, instructions for when and where to enter code into a program are provided. For example, the following statement is meant to be entered into the program named *VIEW_DATA*.

view_data

```
tlb = widget_base(title='View Data', /column, /align_center, $
   space=10, mbar=menubase)
```

The color syntaxing indicates that this code is to be typed into the IDL workbench editor (or, when using the command-line version of IDL, into the editor of your choice), in the file or program unit specified by the bold, italic text above the code.

Existing code in a file is specified by a grey color. Here, we wish to add a field named `ANIMATE` to the existing structure variable `INFO` in the program VIEW_DATA.

view_data

```
info = { $
   rotate    : rotate, $
   color     : [255,0,0], $
   contour   : contour, $
   animate   : 'off' }
```

Find the line that contains the variable `INFO` in the file **view_data.pro** and add the color-syntaxed code exactly as it is typed in the manual.

Code prefixed with **`IDL>`** is intended to be typed at the IDL command line. Included may be IDL's response, such as:

```
IDL> print, !dir
C:\Program Files\Exelis\IDL83
```

Occasionally, code is used to explain a concept or recall previously written code, but it is not meant to be typed. For example,

```
event = {widget_timer, id:0L, top:0L, handler:0L}
```

This statement is not displayed in boldface type, it is not prefaced with **`IDL>`** and it does not reference a routine name, so it should not be typed.

The course files

The course files are a set of programs, data and other documents that have been collected for use in many of the examples in this course. Other courses in the IDL series, such as *Scientific Programming with IDL* and *Application Development I*, have their own course files. The files for this course can be downloaded from the Exelis VIS web site and installed on your machine. Instructions for doing so are given in the following sections.

Downloading the course files

The IDL course files are hosted on the Exelis VIS web site. You can access them through this shortened link:

`http://bit.ly/IDL_coursefiles`

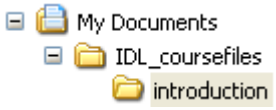
Select the *Introduction to IDL* course files for download. If you wish, feel free to download the entire IDL_coursefiles distribution.

This location for the course files is current as of this printing of the course manual; however, this hosting arrangement may change in the future. If you're unable to access these files, please e-mail us at `training@exelisvis.com` or call us at +1-303-786-9900. We will find a way to make the files available to you.

Installing the course files

Follow these steps to install the IDL_coursefiles distribution on your computer. In an Exelis VIS class, your instructor will have performed these steps for you.

1. Select a directory on your computer to install the course files. A suggested location is under your home directory on a UNIX-based platform or under your **My Documents** folder on Windows.
2. Unpack the course files. On Windows XP, Vista and 7, you can double-click the file's icon to extract its contents. On a UNIX-based platform, use a shell utility such as `unzip` to extract the files.
3. After extraction, the course files will be located in the directory **IDL_coursefiles** under the directory you selected for installation. For example, if you installed the files in your **My Documents** folder on Windows, the directory structure would appear as in the diagram on the right.



This completes the installation of the course files on your machine. Next, we need to tell IDL where to find them. This involves adding these files to your IDL path or creating a project for them in the IDL workbench. We'll address how to do this in Chapter 3, "IDL Basics."

Starting IDL

Table 1-2 gives instructions for starting IDL on its supported platforms. The IDL workbench (denoted *WB* in the Table) is available on Windows, Mac OS X and Linux. Additionally, on these platforms plus Solaris, a command-line version of IDL (denoted *CL* in the Table) is available.

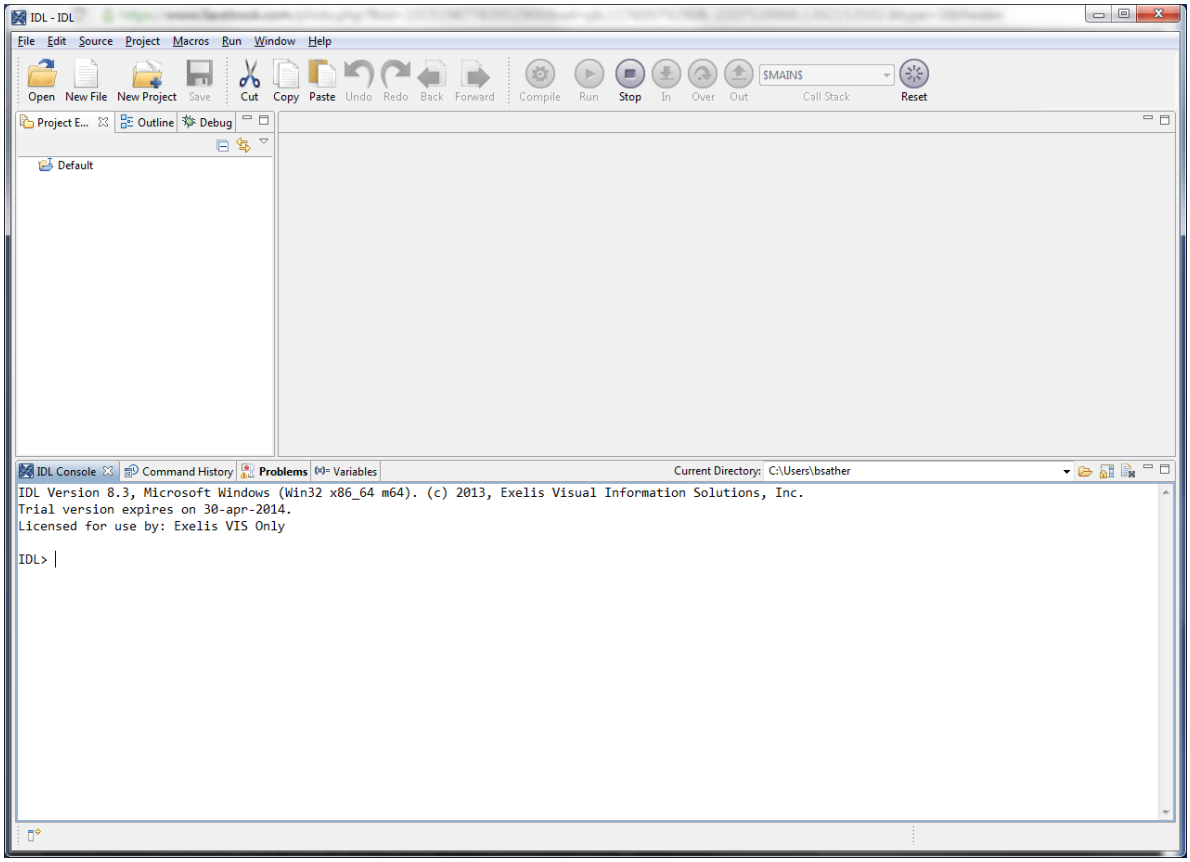
Please start IDL. You may use either the workbench or the command line version of IDL; this course is designed to work with either, though more emphasis is placed on using the workbench.

Table 1-2: Instructions for starting IDL on supported platforms.

Operating system	IDL app	Instructions
Windows	WB	From the **Start** menu, select **Start → All Programs → IDL X.X → IDL**, where **X.X** is the IDL version number; for example, 8.3.
	CL	From the **Start** menu, select **Start → All Programs → IDL X.X → Tools → IDL Command Line**.
Mac OS X	WB	Select the IDL application from the Dock.
	CL	Start the X11 application. From the X11 **Applications** menu, select **Terminal** to start an xterm. Type `idl` at the shell prompt in the xterm.
Linux	WB	Type `idlde` at a shell prompt.
	CL	Type `idl` at a shell prompt.
Solaris	CL	Type `idl` at a shell prompt.

Figure 1-1 shows what the workbench looks like on Windows. It looks the same (except for the operating system's window manager decoration) on Linux and Mac OS X.

Figure 1-1: The IDL workbench on Windows.



An introduction to the IDL workbench is given in Chapter 3, "IDL Basics", but let's first explore what you can do with IDL.

Chapter 2:
A Tour of IDL

This chapter provides a brief, interactive tour of IDL, demonstrating aspects of the language as well as IDL's built-in file access, analysis and visualization capabilities.

Oh, the places you'll go!

—Dr. Seuss

Overview 8
Scalars and arrays 8
Line plots 10
Surface plots 11
Contour plots 12
Displaying and processing images 14
Exercises 15
References 16

Overview

This chapter provides a set of guided exercises to help you get familiar with the basic syntax and functionality of IDL. You're not expected to understand every line of code at this point. The topics covered here will be explained in greater detail as the course progresses.

Please type the statements that are prefaced with the **IDL>** prompt at the IDL command line. A short explanation follows each statement or group of statements.

Scalars and arrays

We can use IDL interactively by typing statements and viewing the results.

```
IDL> print, 3*4
      12
```

The *PRINT* procedure displays the result of the expression `3*4` in IDL's console. Note that the comma is used to delimit arguments to an IDL procedure or function.

IDL allows you to make variables on the fly. Let's look at an example of a scalar variable. (A scalar represents a single value.)

```
IDL> a = 5*12
IDL> help, a
A               INT       =       60
```

We can get information about the scalar variable `A` with the *HELP* procedure. *HELP* is useful for obtaining diagnostic information from IDL. Notice that in addition to a value, `A` is associated with a type of number. By default, numbers without decimal points are treated as two-byte integers. Had we instead typed

```
IDL> a = 5.0*12.0
IDL> help, a
A               FLOAT     =       60.0000
```

the result would be a floating-point value.

Next, let's look at an example of an IDL array variable. (An array represents multiple values.)

```
IDL> b = fltarr(10)
IDL> help, b
B               FLOAT     = Array[10]
IDL> print, b
0.000000  0.000000     0.000000     0.000000     0.000000
0.000000  0.000000     0.000000     0.000000     0.000000
```

The *FLTARR* function is used to create floating-point arrays. Here, the variable `B` is a 10-element floating point array. By default, the values of `B` are initially set to zero. Note that the parameters to an IDL function are enclosed in parentheses.

IDL has control statements similar to those in other programming languages. For example, a FOR loop executes a statement, or a group of statements, a specified number of times. Here's an example of initializing the array `B` with values. Each element of `B` receives the value of its array index.

In IDL 8.3, the *PRINT* procedure is implied. IDL will automatically evaluate any expression or variable name on the command line and print out the result.

```
IDL> for i = 0, 9 do b[i] = i
IDL> b
0.00000      1.00000      2.00000      3.00000      4.00000
5.00000      6.00000      7.00000      8.00000      9.00000
```

The *FINDGEN* function can be used to create an indexed array identical to `B` but without the use of a loop:

```
IDL> c = findgen(10)
IDL> help, c
C               FLOAT     = Array[10]
IDL> c
0.00000      1.00000      2.00000      3.00000      4.00000
5.00000      6.00000      7.00000      8.00000      9.00000
IDL> array_equal(b, c)
       1
```

In IDL, native array functions are much faster than equivalent operations with control statements.

Arrays are handy because they store groups of numbers or text. We can access individual elements or groups of elements from an array by subscripting the array.

```
IDL> c[0:4]
0.000000     1.00000      2.00000      3.00000      4.00000
```

Here we have extracted a portion of the array `C`. Note that array index values start at zero in IDL.

New variables can be made from subscripted arrays.

```
IDL> d = c[1:6]
IDL> help, d
D               FLOAT     = Array[6]
```

The array `D` is created from elements 1 through 6 inclusive of the array `C`.

Reading data from files

By using native IDL programs, or by creating your own programs, you can read just about any type of file. For subsequent examples in this chapter, we'll use data from two files, both located in the **examples/data** directory of the IDL distribution.

The first is a flat binary file containing 512 byte values representing a sine wave with exponentially increasing frequency. With IDL routines, we locate this file and read its contents into the variable `CHIRP`:

```
IDL> file1 = filepath('chirp.dat', subdir=['examples','data'])
IDL> chirp = read_binary (file1, data_dims=512, data_type=1)
```

We can check the results of the read with *HELP*:

```
IDL> help, chirp
CHIRP           BYTE      = Array[512]
```

Note that the data are in the form of a one-dimensional array (also called a vector).

The second file is an IDL SAVE file containing elevations from a USGS digital elevation model (DEM) of the Maroon Bells peaks near Aspen, Colorado.

```
IDL> file2 = file_which('marbells.dat')
IDL> restore, file2
IDL> help, elev
ELEV            INT       = Array[350, 450]
```

The IDL variable `ELEV` is restored from this file into your current IDL session. Note that this variable is a two-dimensional array of type integer.

Line plots

What's in this variable `CHIRP`? With *PRINT*, we could view the data values in tabular form, but considering there are 512 values, the data may be easier to comprehend graphically. Display the data as a line plot with the *PLOT* function:

```
IDL> p = plot(chirp)
```

A graphics window appears:

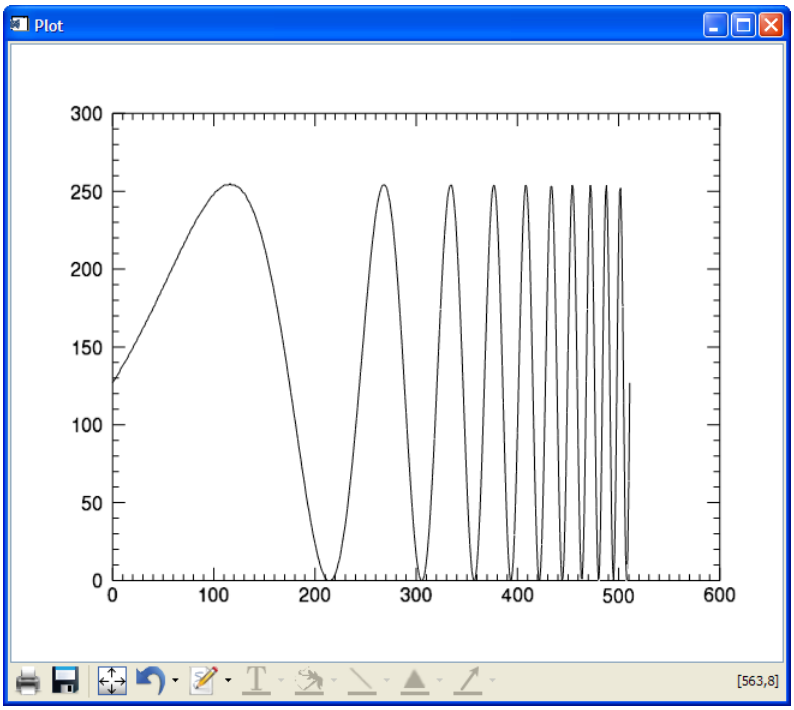


Figure 2-1:

A line plot of the `CHIRP` data.

IDL graphics windows are interactive: you can grab the edge of a window to resize it, or use the mouse to pan or zoom the data visualization.

PLOT returns a reference, stored in the variable `P`, that we can use to update the graphic we've created. Display the data with a dashed line by changing the LINESTYLE property of *PLOT*:

```
IDL> p.linestyle = 'long dash'
```

Note how the plot window automatically updates the plot with the new linestyle. Next, change the color of the plot line to green with the COLOR property:

```
IDL> p.color = 'green'
```

Properties allow us to customize and add detail to a plot, at either the creation of the plot or afterward. Properties are used in all of the graphics routines in IDL. Multiple properties can be set simultaneously. For example, we can change the line thickness (THICK property) and add a descriptive title (TITLE property) to our plot with:

```
IDL> p.setproperty, thick=2, $
>      title='Sine Wave with Exponentially Increasing Frequency'
```

`SetProperty` is a program (called a method) that is built in to every graphics routine.

Note that the last statement was too long to fit on a single line in the course manual. In IDL, a statement can be broken into multiple lines and connected with the continuation character `$`. The IDL prompt changes to a `>` to alert you that a statement is being continued. When typing at the IDL command line, a statement like this can be entered in one line and the continuation character can be omitted, the command line will scroll.

Surface plots

The variable `ELEV` that we read in to IDL earlier has two dimensions, representing a 350 x 450 array of digital elevation model values. We can view data stored in two-dimensional arrays like this with the *SURFACE* function:

```
IDL> s = surface(elev, title='Maroon Bells')
```

By default, *SURFACE* displays a filled surface that uses light source shading to give the appearance of depth. You can also display the surface as points or a wire mesh with the STYLE property:

```
IDL> s.style = 'points'
IDL> s.style = 'mesh'
```

Change the surface style back to filled and set its color to royal blue with `SetProperty`:

```
IDL> s.setproperty, style='filled', color='royal blue'
```

View the `ELEV` DEM from a vantage point high above the earth by rotating the surface with its `Rotate` method:

```
IDL> s.rotate, 15, /xaxis
IDL> s.rotate, -15, /zaxis
```

SURFACE really shines for interacting with data. Try grabbing the surface visualization with the mouse to rotate it. Grab a corner of the surface's bounding box to pan, grab a corner to zoom in/out.

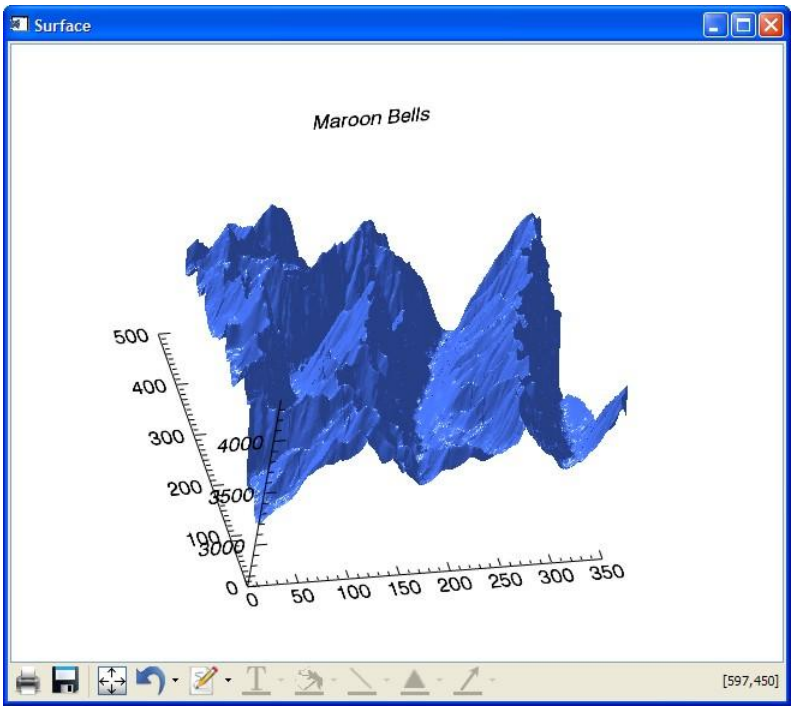


Figure 2-2:

The Maroon Bells DEM viewed with *SURFACE*.

The ability to interact with a data representation and view it from different angles can assist in understanding the features of the data.

Contour plots

A contour plot is another visualization type for data stored in two-dimensional arrays:

```
IDL> c = contour(elev, n_levels=12, rgb_table=5)
```

In this statement, we've overridden *CONTOUR*'s default number of isopleths (with the N_LEVELS property; the default is seven), as well as the color set used for the isopleths (the RGB_TABLE property; the default is black for all isopleths).

Allowing IDL to choose contour levels is useful for getting a first look at data, but a better technique is to calculate and assign contour levels given knowledge of the data. Start by determining the minimum and maximum values of the elevations in `ELEV` with the *MIN* and *MAX* functions:

```
IDL> print, min(elev), max(elev)
      2666    4241
```

Given this knowledge of the range of the elevation values (in meters), define a set of 16 contour levels starting at 2700 m with an increment of 100 m.

```
IDL> start_elevation = 2700     ; meters
IDL> increment = 100            ; meters
IDL> n_levels = 16
IDL> clevels = indgen(n_levels)*increment + start_elevation
```

Print the levels for a sanity check:

```
IDL> clevels
   2700      2800      2900      3000      3100      3200      3300
   3400      3500      3600      3700      3800      3900      4000
   4100      4200
```

Make a new contour plot and apply these contour levels to the plot with the C_VALUE and RGB_INDICES properties. You can use the up and down arrows on your keyboard to retrieve and edit a command you typed earlier in your IDL session.

```
IDL> d = contour(elev, rgb_table=15, c_value=clevels, $
>      rgb_indices=bytscl(clevels), title='Maroon Bells')
```

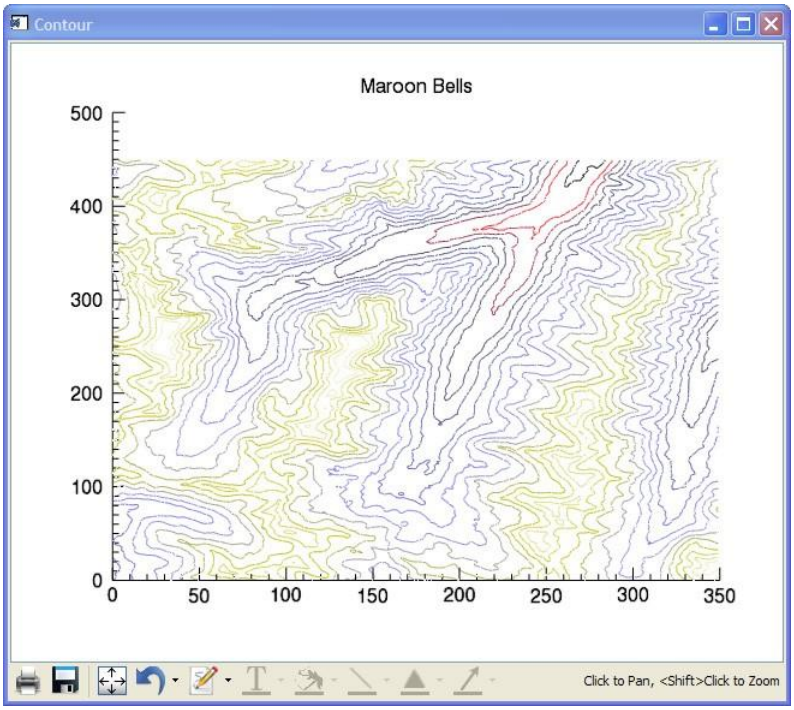


Figure 2-3:

The Maroon Bells DEM viewed with *CONTOUR*.

To see filled contours, set the FILL property on this graphic using its reference, `D`:

```
IDL> d.fill = 1
```

Make the contour plot three-dimensional by turning off the PLANAR property:

```
IDL> d.planar = 0
```

This makes a visualization with elevation proportional to the colors in the color palette.

Displaying and processing images

IDL can display arrays as images. Data values in an array are matched to grayscale intensities or colors.

Display the Maroon Bells DEM as an image using the *IMAGE* function:

```
IDL> i = image(elev)
```

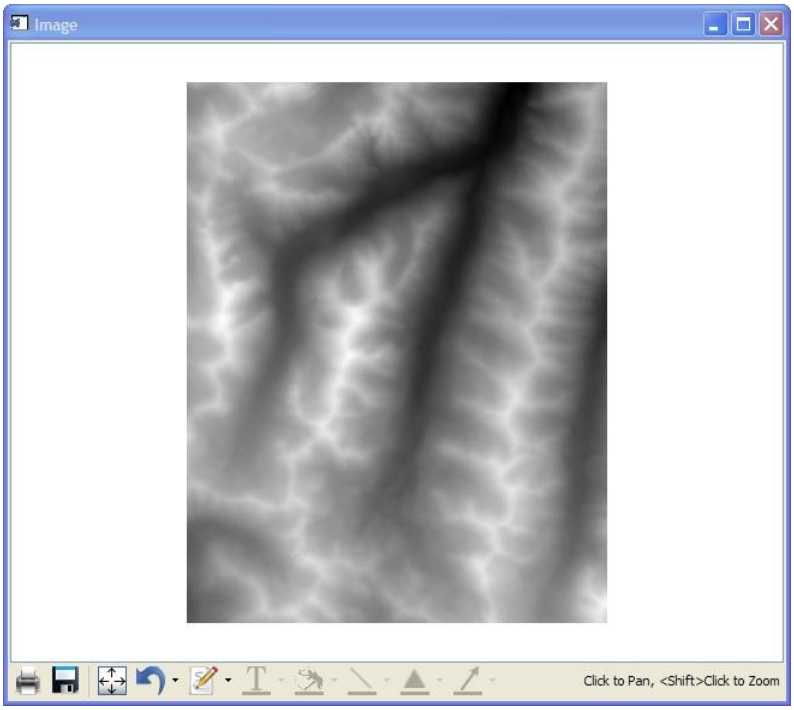


Figure 2-4:

The Maroon Bells DEM visualized with *IMAGE*.

The data are displayed as a 1:1 image—each element of the 350 x 450 `ELEV` array corresponds to a pixel on the display. A grayscale color palette is used by default. Data in this form can be displayed with different color palettes. The data values aren't altered; rather, the colors used to represent them are.

IDL excels at image processing. Differentiate the image using the *SOBEL* function:

```
IDL> elev_edge = sobel(elev)
```

SOBEL is an edge enhancer. Here, it gives an estimate of the elevation changes in the DEM. Normalize the elevation changes

```
IDL> elev_edge_normalized = elev_edge / float(max(elev_edge))
```

and view the result with *IMAGE*:

```
IDL> gradients = image(elev_edge_normalized, rgb_table=5)
```

Annotate the visualization with a title and a vertical colorbar using the *COLORBAR* and *TEXT* functions:

```
IDL> cbar = colorbar(orientation=1)
IDL> t = text(0.5, 0.95, 'Gradients: Maroon Bells DEM', /normal, $
>      alignment=0.5, target=gradients)
```

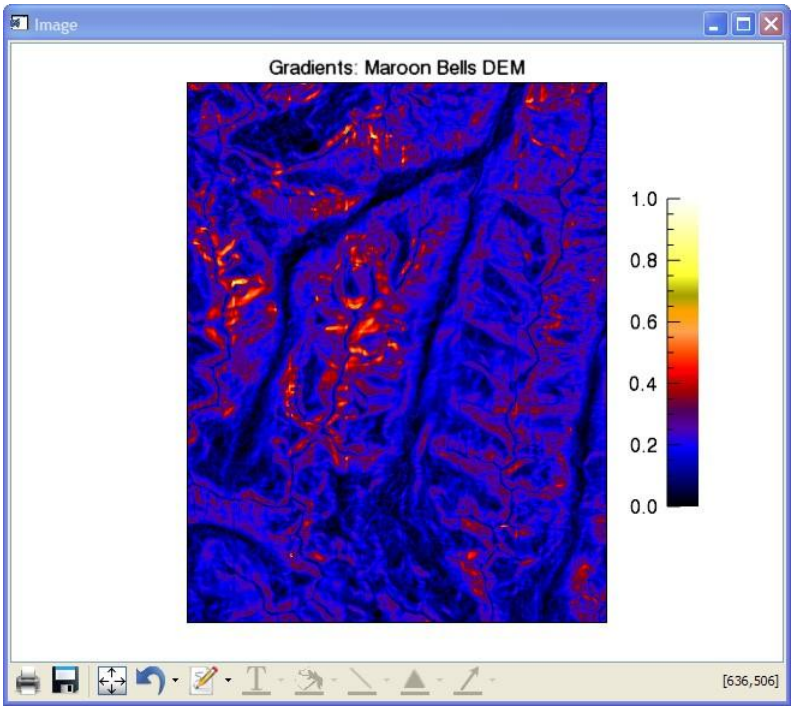


Figure 2-5:

An enhanced view of the Maroon Bells DEM depicting regions of sharp elevation changes.

In the processed image, the strikingly sharp peaks of the Maroon Bells have high pixel values—orange and red in this color table—because these regions of steep elevation change must have a derivative that is greater than zero in magnitude. The canyon floors have low pixel values—blue and black—because they're flatter, implying a smaller derivative.

Exercises

1. Make an IDL variable that holds the even integers between 0 and 10, inclusive.
2. Display the variable `ELEV` as a three-dimensional contour plot, but with lines/isopleths only (i.e., not filled).
3. Construct one cycle of a sine wave using the *SIN* function. Display it with *PLOT*.

For sample solutions to these problems, see the code in the file **tour_exercises.pro** in the IDL coursefiles **introduction/src** directory.

References

Bowman, K. P. *An Introduction to Programming with IDL*. Boston, Massachusetts: Academic Press, 2006.

Prof. Bowman's book is a distillation of his many years of teaching IDL to undergraduate students at Texas A&M University. It is filled with useful examples and exercises.

Fanning, David F. *IDL Programming Techniques*. Second Edition. Fort Collins, Colorado: Fanning Software Consulting, 2000.

This is the first book on using IDL, written by an original member of the Training Department at Research Systems. Dr. Fanning excels at explaining the idiosyncrasies of IDL to new and experienced users.

Galloy, Michael. *Modern IDL: A Guide to IDL Programming*. Boulder, Colorado, 2011. <http://modernidl.idldev.com/>

This is a great book for anyone using IDL, but it shines in its coverage of advanced topics and application development with IDL. It's also a useful reference guide, collecting tables and lists of items that are scattered throughout the IDL Help system.

Gumley, Liam E. *Practical IDL Programming*. San Francisco: Morgan Kaufmann, 2001.

An excellent book on general IDL programming recommended by David Stern, creator of IDL and founder of RSI.

Kling, Ronn. *IDL Primer*. Marshall, Virginia: KRS, Inc., 2007.

This handy, pocket-sized book summarizes the basics of using IDL.

Chapter 3: IDL Basics

Some of the basics of using IDL are presented in this chapter.

> *...the command line continued to exist as an underlying stratum – a sort of brainstem reflex – of many computer systems.*
>
> —Neal Stephenson, *In the Beginning was the Command Line*

IDL directory structure 18
The IDL workbench 19
Working directory 24
Preferences 24
Search path 26
The IDL Help system 27
References 28

IDL directory structure

By default, IDL is installed in the following locations on Windows and UNIX-based platforms:

- Windows (XP, Vista): **C:\Program Files\Exelis\IDLXX**, where **XX** is the IDL version number; 8.3, for example
- Mac OS X: **/Applications/itt/idl/idlXX**
- Solaris and Linux: **/usr/local/itt/idl/idlXX**

This installation directory is called the *main IDL directory*. It is used as a reference point for locating other files in the IDL distribution. Within IDL, the `!DIR` system variable stores the path to this directory. (System variables are discussed in Chapter 5, "Data Structures"). On UNIX-based systems, the environment variable `$IDL_DIR` also contains the path to this directory.

A list of the directories in the IDL distribution and an overview of their contents are given in Table 3-1.

Table 3-1: The directories in the IDL distribution.

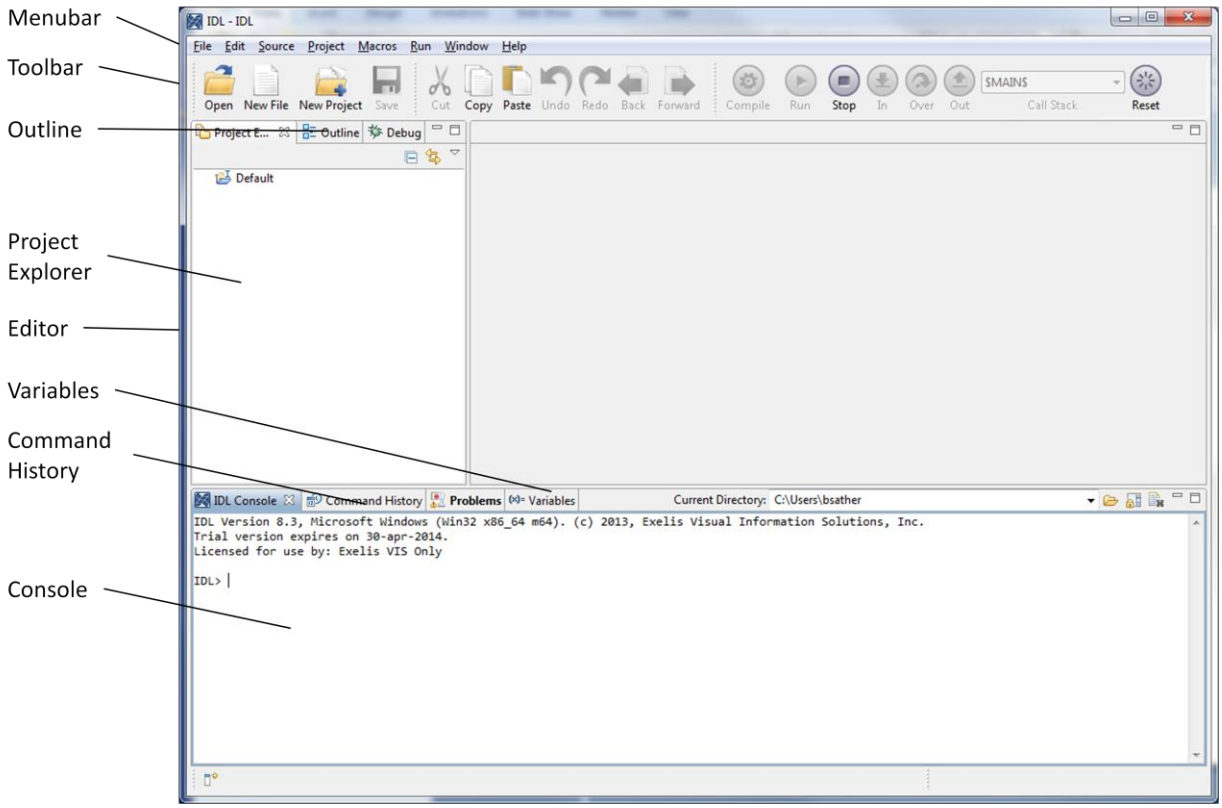
Name	Contents
bin	Stores the core IDL libraries as well as other shared libraries that are loaded dynamically into IDL. Also stores Eclipse components for building and adding to the workbench.
examples	Holds a collection of example IDL program files as well as many types of data files.
external	Contains information and examples of linking IDL with code written other languages, such as Fortran, C/C++ and Java.
help	The location of the IDL Help system files.
lib	Stores the IDL routines (in source code form) that are a part of the distribution, but are written in the IDL language itself.
resource	A catch-all directory that contains fonts IDL uses, the IDL mapping and color databases, as well as examples of IDL Bridge technologies.

With IDL's standardized directory structure, if you know the location of a file in the IDL distribution on one system, you'll know its location on another system, even if it's on another platform.

The IDL workbench

The IDL workbench is the application interface for IDL. It replaces the IDL Development Environment (IDLDE) included with IDL before version 7.0. The IDL workbench is built on Eclipse (`http://www.eclipse.org`), a powerful and popular open-source framework for building development environments. The workbench provides editing and debugging tools in an interface that looks and behaves the same way on all platforms supported by IDL. Figure 3-1 shows an annotated screenshot of the IDL workbench.

Figure 3-1: The IDL workbench.



A summary of the IDL workbench components (called *views* in the Eclipse terminology) displayed in Figure 3-1 is given below.

Table 3-2: IDL workbench views.

Workbench view	Function
Menubar	Controls for opening, editing, compiling and running IDL programs, as well as controls for interacting with other features of the workbench.
Project Explorer	A tool that shows the IDL programs, data and support files in a directory on the file system. Click on a file name to open it.
Toolbar	Graphical controls similar to those on the Menubar.
Outline	Presents a list of programs in a file; click on the program name to display it in the Editor.

Table 3-2: IDL workbench views.

Workbench view	Function
Editor	Where IDL programs are written, edited and debugged. Note the syntax coloring.
Variables	A list of the variables defined at the current level in the call stack.
Command History	A list of recent statements entered in the Console. You can double-click or drag-and-drop statements from here to the Console or the Editor.
Console	Where IDL statements are entered (at the **IDL>** prompt) and where IDL returns information to the user. Note the syntax coloring.

Views can be moved and docked in different locations in the workbench, or they can be torn off and placed elsewhere on your desktop.

Exploring the IDL workbench

The best way to learn about the IDL workbench is to experiment with it. Here's a list of examples to try. Note that these are first steps; we'll do much more as we move through the course.

1. In the Menubar, select **File → Open ...** (or select the Open icon in the Toolbar).

   Use the file picker to find a JPEG image; for example, **Day.jpg** in IDL's **examples/data** directory.

2. Open the image file. The image is read into IDL and displayed in the interactive *IMAGE* tool.

   Note that the following views are updated:

   - The Variables view lists the variable `DAY_JPG`, which holds the image pixel data.
   - The Console reports the IDL command that read and displayed the image.

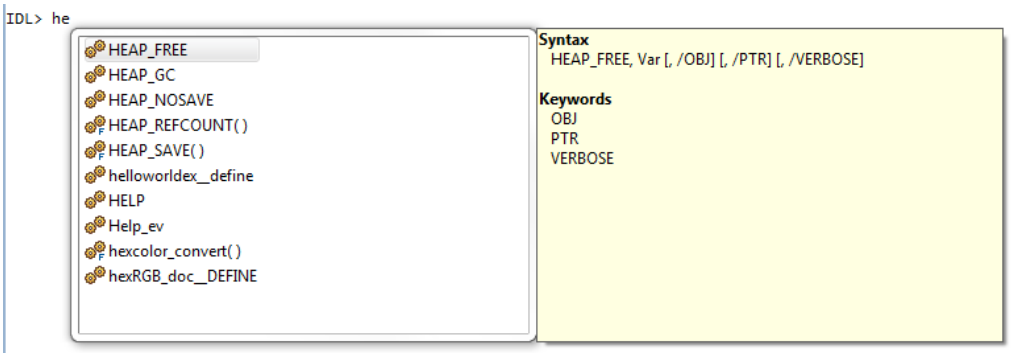
3. Get information on the variable `DAY_JPG` with the *HELP* procedure:

   ```
   IDL> help, day_jpg
   ```

   Note that *HELP* appears in a blue color in the Console. This is called *syntax coloring*; it helps code stand out when writing programs in the Editor or entering statements at the **IDL>** prompt, making the code easier to read.

4. The Content Assist feature saves time. At the **IDL>** prompt, type only the first two letters of the *HELP* procedure, then select `<Ctrl>-<Space>` on your keyboard.

A menu of possible completions appears:



Select the statement to complete from the Content Assist dialog using the arrow keys on your keyboard. Content Assist is also available in the Editor.

5. Look at the Command History view. (If you can't find the Command History view, select it from **Window → Show View** in the Menubar.) All the commands we've used recently are listed there. These commands can be dragged and dropped to the **IDL>** prompt or into the Editor. The Command History persists across sessions.

   At the **IDL>** prompt, you can also use the up and down arrows on the keyboard to retrieve statements typed earlier.

6. There are myriad keyboard shortcuts in the workbench. Keyboard shortcuts save time. For example, instead of using the mouse, use `<Ctrl>-i` to put focus at the **IDL>** prompt.

   The Table below displays a set of selected keyboard shortcuts available in the workbench. Shortcuts are given for both the Default and Emacs styles. In the table, "C" stands for the `<Ctrl>` key, while "A" stands for the `<Alt>` key.

Table 3-3: Selected workbench keyboard shortcuts.

Default style	Emacs style	Function
`C-o`	`C-x C-f`	Open a file.
`C-s`	`C-x C-s`	Save the file open in the editor window.
`C-F8`	`C-F8`	Compile the file open in the editor window.
`F8`	`F8`	Run the program shown in the editor window.
`C-x`	`C-w`	Cut
`C-c`	`A-w`	Copy
`C-v`	`C-y`	Paste
`C-z`	`C-x u`	Undo (very useful)
`C-f`	`A-r`	Find/Replace
`C-l`	`C-x g`	Goto line in editor window
`C-i`	`C-i`	Put focus in command line.
`F12`	`F12`	Put focus in current editor window.
`C-Space`	`C-Space`	Bring up Content Assist.

A quick reference of available keyboard shortcuts can be accessed from **Help → Key Assist...** in the workbench Menubar. An unabridged table of keyboard shortcuts is given in the IDL workbench preferences (see "Preferences" below). You can also define your own keyboard shortcuts there.

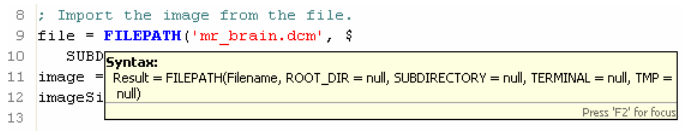
7. Open the IDL program **bytescaling.pro** using the `.edit` executive command:

   ```
   IDL>  .edit bytescaling
   ```

   The source code for the program *BYTESCALING* appears in the Editor. Note that, like in the Console, the source code in the Editor has syntax coloring.

   Executive commands are discussed in Chapter 6, "Programming."

8. The Editor has a Hover Help feature. Place your cursor over the *FILEPATH* function on line 9 of **bytescaling.pro**. A pop-up window appears, giving quick information about this routine:

9. Open another program file, **maskingimages.pro**, using the `.edit` command:

   ```
   IDL>  .edit maskingimages
   ```

   The source for this program appears in a separate tab in the Editor. You can cycle through open Editor tabs with the `<Ctrl>-F6` keyboard shortcut.

10. Double-click the **bytescaling.pro** tab on the Editor. This maximizes the Editor window. Double-click again to return the window to its original size.

11. Select the **maskingimages.pro** tab on the Editor. Compile and run the *MASKINGIMAGES* program by selecting the `F8` key.

    More on compiling and running programs is presented in Chapter 6, "Programming." More on displaying and processing images is presented in Chapter 10, "Analysis."

12. Close the Editor windows by clicking the "x" on the tabs.

Again, these are first steps; we'll see more as we move through the course. For more detailed information on using the IDL workbench, see the "References" section at the end of this chapter.

Projects

The IDL workbench uses *projects*, a concept inherited from Eclipse, to organize files.

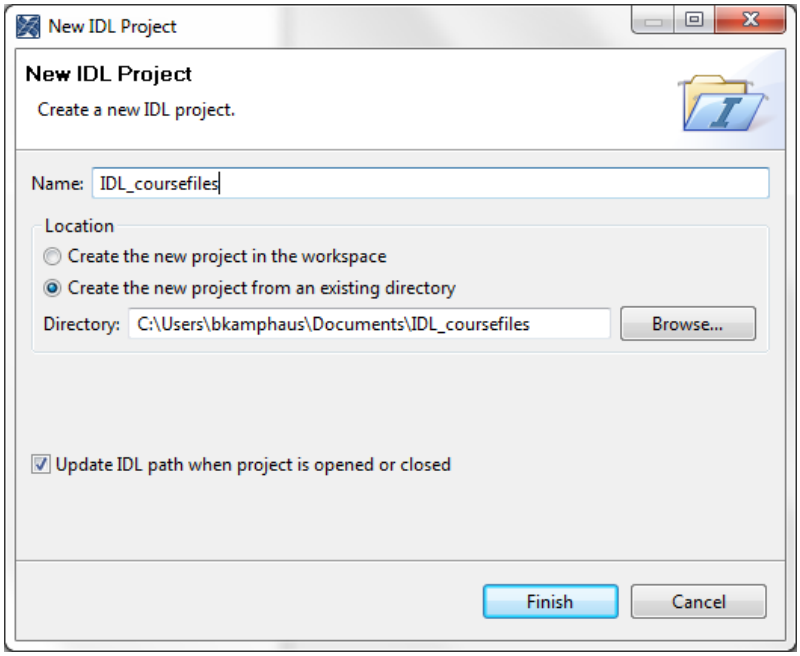
The Project Explorer view (see Figure 3-1) acts like an embedded file browser in the workbench (it looks suspiciously like Windows Explorer). With it, you can browse files related to your work in IDL. More importantly, projects also provide an easy way to manage IDL's *path*—the set of directories IDL searches to find programs—from the workbench. Path is discussed in more detail in the "Search path" section later in this chapter.

By default, a project named "Default" is provided. It maps to a directory named **Default** on the filesystem. You can verify it lives under **$HOME/IDLWorkspace83**, where **$HOME** is your home directory, by right-clicking on it in the Project Explorer and selecting **Properties** from the menu. A good use for this Default project is as a scratch directory for temporarily storing files. We'll use it in class as a place to store our work.

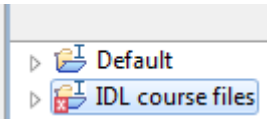
You can easily map existing directories on your computer's filesystem to new projects. For example, we'll set up the directory containing the IDL coursefiles as a new workbench project. Here are the steps to follow:

1. From the Menubar, select **File → New Project...**. The New IDL Project dialog appears:

Figure 3-2: The New IDL Project dialog.



2. Change the name of the project from "NewProject" to "IDL course files."
3. Use the **Browse** button to find and select the location of the **IDL_coursefiles** directory on your computer. (Recall we installed the coursefiles in the section "Installing the course files" on page 4.) Or, enter it manually.
4. Leave the button titled "Update IDL path when project is opened or closed" checked to allow IDL to manage the path for this project.
5. Click the **Finish** button. The new project will appear in the Project Explorer view of the workbench. Your Project Explorer should now look like this:



The IDL course files are used frequently in examples and exercises throughout this course. An IDL project provides a convenient way to browse and access these files. The red "X" appears on the course files due to a JSON file. Do not worry about it for this course.

Working directory

IDL has the concept of *current* or *working* directory. The working directory is the default location IDL looks to when opening or saving a file. The workbench's Console displays it in a droplist:

This is the home directory for the user "bsather" on Windows. On startup, the workbench defaults to the user's home directory (`$HOME` on UNIX-based systems or `%HOME%` on Windows).

In command-line mode, the working directory is the directory from which you start IDL. For example:

```
$ pwd
/home/bsather/stuff
$ idl
IDL> cd, current=c & print, c
/home/bsather/stuff
```

You can use the *CD* procedure to change directories. For example,

```
IDL> cd, '..'
```

changes IDL's working directory to the parent of the previous directory. *CD* can be used in the workbench or in IDL's command-line mode. The workbench also allows you to change directories using the Console drop list shown above.

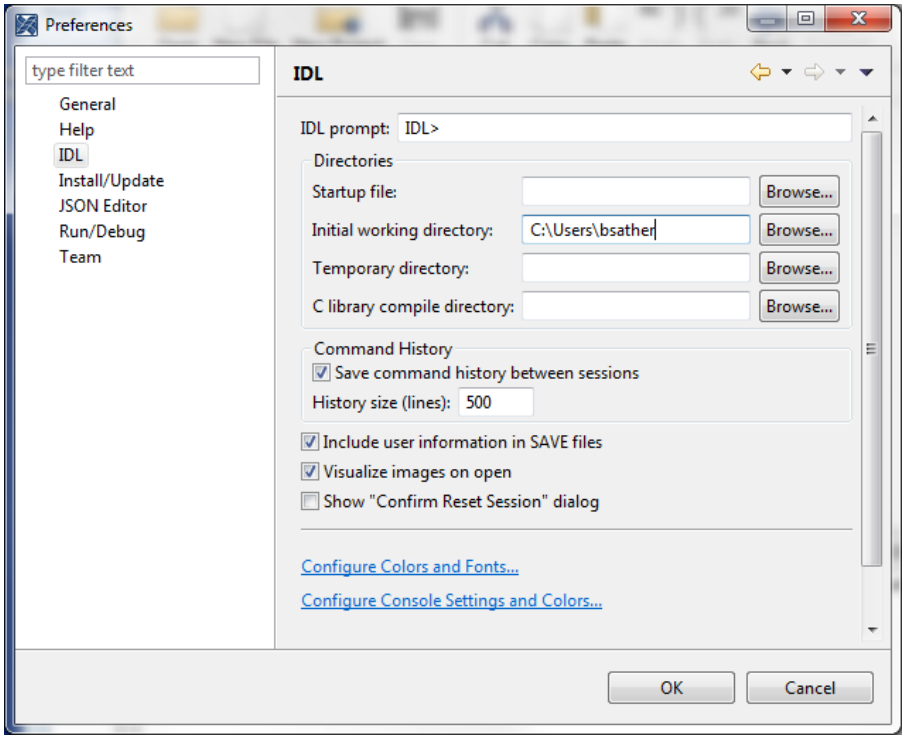
If you're using the workbench for this course, set the Default project to be the working directory by right-clicking on it in the Project Explorer and selecting **Set Selection as Current Working Directory** from the menu. If you're using IDL command-line mode, *CD* to a directory where you'll store results from the course.

Preferences

IDL has numerous configurable system preferences. Preferences can be set through the workbench or through command-line tools available in either the workbench or IDL command-line mode. Here, we set a preference that affects only the workbench, the initial working directory.

1. In the workbench Menubar, select **Window → Preferences** (on Mac OS X, select **IDL → Preferences**). Within the Preferences dialog, select the **IDL** item from the left pane. The dialog should look like Figure 3-3 below.

Figure 3-3: The workbench preferences, showing IDL preferences.



2. Next, set the "Initial working directory" field to your Default project directory, either by entering it manually or by using the **Browse** button.
3. Select **OK**. The workbench will use this setting for the initial working directory in subsequent IDL sessions.

IDL's system preferences can also be accessed through the routines listed in Table 3-4.

Table 3-4: IDL preference system routines.

Routine	Function
HELP	Setting the PREFERENCES keyword displays the current status of IDL's preferences, as well as any changes pending and the location of the user's **.pref** file.
PREF_SET	Sets new values for IDL preferences and optionally commits them. Set preferences exist in a pending state by default.
PREF_GET	Gets information about an IDL preference.
PREF_COMMIT	Commits preferences that exist in a pending state.

For example, list the preferences set in your current IDL session using *HELP*:

```
IDL> help, /preferences
```

More information on IDL system preferences can be found in the IDL Help system.

Search path

The search path is an ordered list of directories that IDL searches to find program, batch, SAVE and data files. Using a path is efficient: rather than looking through the entire directory tree of a file system, only a subset of directories where IDL might expect to find files is searched.

Path is very useful. IDL uses it not only to find its routines but also user-defined routines. A well-constructed path makes IDL much easier to use. An explanation of how IDL uses path is provided in "Calling mechanism" on page 70.

If you use the workbench, you can use projects to manage IDL's path, as shown in "Projects" on page 22. However, the paths managed in projects aren't available in IDL's command-line mode. If you use command-line mode, or if you alternate between the command line and the workbench, it's recommended that you instead use the path preference to manage IDL's path.

Let's query the preference system to find IDL's default path:

```
IDL> path = pref_get('idl_path')
IDL> path
<IDL_DEFAULT>
```

One entry, `<IDL_DEFAULT>`, exists; it's a special token that IDL replaces with paths to the **lib** and **examples** subdirectories of the IDL distribution. This is how IDL finds routines such as *PLOT*, *FILE_WHICH* and *READ_BINARY*, for example.

For user-defined paths, we'll need to modify this `IDL_PATH` preference. In the "Projects" section, users of the workbench set up the IDL coursefiles distribution as a project. Let's do the same for users of IDL command-line mode. (You don't have to perform this task if you've already set up the course files in a workbench project.)

If the **IDL_coursefiles** directory is located under your home directory, e.g., for a user "mpiper" in **C:\Users\mpiper\IDL_coursefiles**, then construct a new path using

```
IDL> new_path = path + path_sep(/search) $
IDL>     + expand_path('+\Users\mpiper\IDL_coursefiles')
```

In this statement, `PATH` is the starting path we retrieved from *PREF_GET* above, *PATH_SEP* makes the appropriate path delimiter (here, a colon ":"), *EXPAND_PATH*, in conjunction with the "+" sign, lists all the children of **/home/mpiper/IDL_coursefiles**.

Apply this new path to IDL's path preference:

```
IDL> pref_set, 'idl_path', new_path, /commit
```

then force IDL to recognize the new additions to its path by rebuilding the path cache:

```
IDL> path_cache, /rebuild
```

If you're using the workbench or IDL command-line mode, you'll now be able to access the programs and data located in the IDL coursefiles distribution.

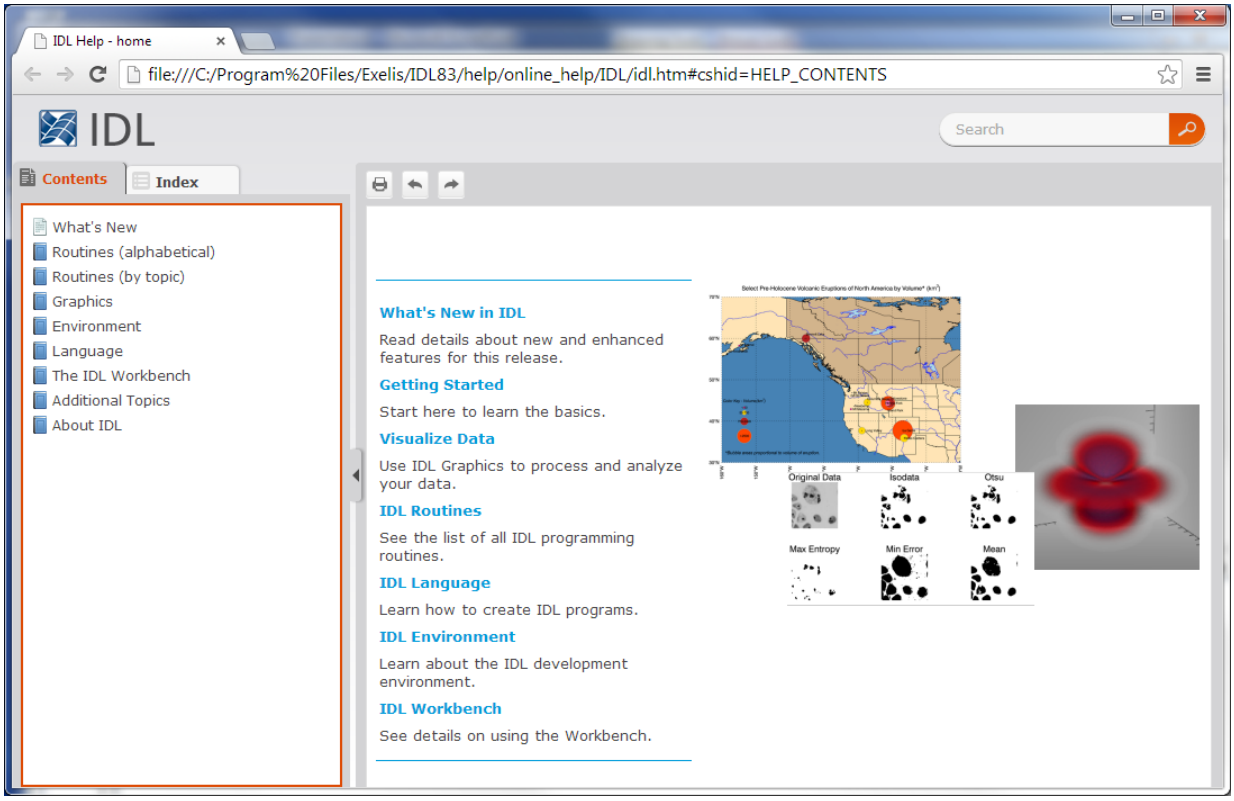
The IDL Help system

Every IDL installation has a link to the IDL help, which is now in a web compatible format. In the help system, you can find

- features of the IDL workbench
- IDL routines and their inputs
- programming interfaces for various file formats
- how to communicate with Java, C and Fortran programs

as well as a great deal of general information about IDL. The Help system is the ultimate reference tool for working with IDL. Always keep it open when working with IDL.

Figure 3-4: The IDL Help system browser.



Start the Help system by selecting **Help → Help Contents** from the IDL workbench Menu bar. On UNIX-based systems, the Help system can also be started from a shell prompt:

```
$ idlhelp
```

The Help system browser (see Figure 3-4 above) is divided into left and right panels. The right panel initially displays the Help system home page. The left panel has tabs for

1. browsing the contents of the Help system,
2. browsing the index of the Help system, and
3. searching the Help system.

Though the Help system has information on just about every conceivable IDL topic, finding the information you need can be difficult. In class, you'll see examples of efficient use of the Help system. A recommended technique is to use the Index tab if you have some idea of what you're looking for, and the Search tab otherwise.

You can quickly search the Help system from the IDL workbench command line by typing a question mark followed by the search term. For example, to get help on the *SURFACE* function, type

```
IDL> ?surface
```

The Help system page for *SURFACE* appears in the Help system browser.

References

About the Eclipse Foundation. The Eclipse Foundation, 2009. <http://www.eclipse.org/org/>. Accessed 2009-06-16.

The history of the Eclipse project and the Eclipse Foundation.

Bowman, K. P. *An Introduction to Programming with IDL*. Boston, Massachusetts: Academic Press, 2006.

Prof. Bowman's book is a distillation of his many years of teaching IDL to undergraduate students at Texas A&M University. It's filled with useful examples and exercises.

Fanning, David F. *IDL Programming Techniques*. Second Edition. Fort Collins, Colorado: Fanning Software Consulting, 2000.

A useful introductory book written by an original member of the Training Department at Research Systems. Dr. Fanning excels at explaining the idiosyncrasies of IDL.

Gumley, Liam E. *Practical IDL Programming*. San Francisco: Morgan Kaufmann, 2001.

An excellent book on general IDL programming recommended by David Stern, creator of IDL and founder of RSI.

Kling, Ronn. *IDL Primer*. Marshall, Virginia: KRS, Inc., 2007.

This handy, pocket-sized book summarizes the basics of using IDL.

Chapter 4: Line, Bar and Scatter Plots

This chapter describes how to create several types of point- and line-based graphics with IDL.

> *Imagination will often carry us to worlds that never were. But without it we go nowhere.*
>
> —Carl Sagan

Graphics routines 30
Reflectance spectra 30
Boulder temperature data 33
The sunspot cycle 38
Exercises 41
References 42

Graphics routines

Table 4-1 lists the IDL routines for creating various types of line, bar and scatter plots. Of these, the *PLOT* function is the primary routine for making line plots in IDL.

Table 4-1: IDL Graphics routines for producing line, bar and scatter plots.

Name	Description
PLOT	Makes line plots on two-dimensional Cartesian axes. You can control plot properties such as color, line style, line thickness, plot symbols, etc. *PLOT* can also be used to make scatter plots.
BARPLOT	Makes bar plots, including floating bar plots.
ERRORPLOT	Displays data with error bars. In addition to standard *PLOT* properties, you can control aspects of how the error bars are displayed.
PLOT3D	Makes line or scatter plots in three dimensions.
POLARPLOT	Makes line plots in two- or three-dimensional polar coordinates.

In addition to covering these routines in this chapter, we'll also introduce some helpers like *LEGEND*, *POLYLINE* and *POLYGON*.

To demonstrate how the graphics routines in Table 4-1 work, we'll use them in visualizing three datasets: a group of reflectance spectra, a year of daily temperature extrema in Boulder, Colorado and the last 40 years of sunspot activity.

Reflectance spectra

Reflectance spectroscopy is the study of light reflected from some material as a function of wavelength. Reflectance spectra can be measured in a laboratory for known minerals, for example, then used as a reference for terrestrial imagery captured from a multi- or hyperspectral sensor.

The file **spectra.txt** in the coursefiles **data** directory contains reference spectra from ENVI's JPL Mineral Spectral Library for three minerals: actinolite, alunite and antlerite. Read the contents of this file into IDL with the following statements:

```
IDL> file = file_which('spectra.txt')
IDL> readcol, file, wavelength, actinolite, alunite, antlerite
```

READCOL reads the four columns of data from the file into variables `WAVELENGTH`, `ACTINOLITE`, `ALUNITE` and `ANTLERITE`. Reading text files in covered in greater detail in Chapter 8, "File Access."

Visualize the actinolite reflectance spectrum with *PLOT*:

```
IDL> p = plot(actinolite)
```

PLOT operates on arrays. When *PLOT* is called with one parameter, the parameter is plotted versus its index values. *PLOT* can also be called with two parameters—the first

representing the independent data, the second the dependent data. To produce a more informative plot, display the measured actinolite spectrum versus its wavelength values:

```
IDL> q = plot(wavelength, actinolite)
```

Though the graphic contains more information, we still can't tell what's being plotted by looking at it. We at least need axis labels and a title:

```
IDL> r = plot(wavelength, actinolite, title='Actinolite Spectrum', $
>      xtitle='Wavelength ($\mu$m)', ytitle='Reflectance', $
>      xrange=[0.4, 2.5], yrange=[.48, .81])
```

The result is a basic, yet descriptive, graphic. Compare your result with the Figure. Note the use of the TEX-like formatting for making the Greek character *mu* on the *x*-axis title.

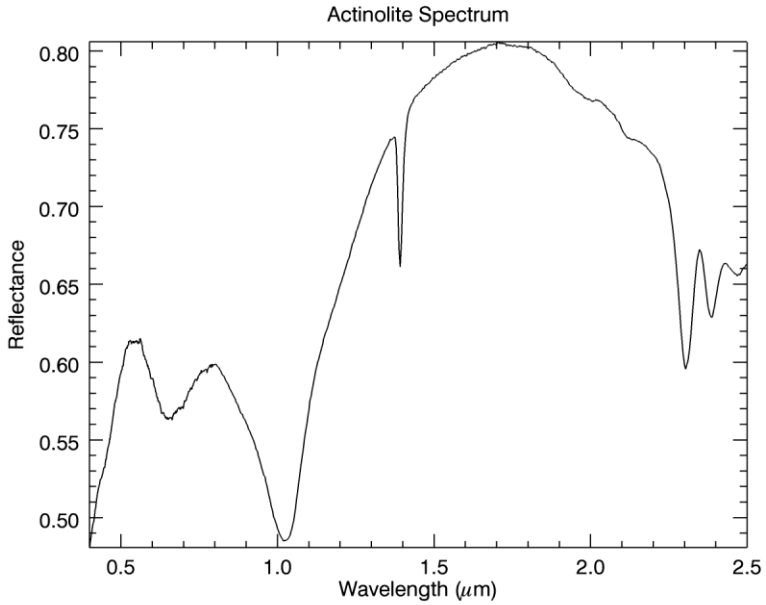


Figure 4-1:

A plot of the actinolite reflectance spectrum.

Before continuing, programmatically close the graphics windows with

```
IDL> foreach i, [p,q,r] do i.close
```

The FOREACH operator iterates over items in an array, list or hash. It's covered in greater detail in Chapter 6, "Programming."

Multiple data

Let's plot all three reflectance spectra on the same wavelength axis. Start with the actinolite spectrum, adding some additional detail:

```
IDL> s1 = plot(wavelength, actinolite, color='red', thick=2, $
>      xrange=[min(wavelength), max(wavelength)], yrange=[0,1], $
>      xtitle='Wavelength ($\mu$m)', ytitle='Reflectance', $
>      title='Reflectance Spectra of Three Minerals')
```

This spectrum is now plotted in red with a double-weight line. The YRANGE property is set to the minimum/maximum values of the *y*-axis; since reflectance is defined on a unit scale, this is `[0,1]`.

Next, use *PLOT* with the OVERPLOT property to add the alunite and antlerite spectra to the graphic:

```
IDL> s2 = plot(wavelength, alunite, color='green', thick=2, /overplot)
IDL> s2 = plot(wavelength, antlerite, color='blue', thick=2, /overplot)
```

The three spectra, displayed in red, green and blue, share the same wavelength axis.

Annotation

We can use *LEGEND* to add a legend for the mineral names to the graphic:

```
IDL> s1.name='Actinolite'
IDL> s2.name='Alunite'
IDL> s3.name='Antlerite'
IDL> legend = legend(position=[2.4,0.21], /data)
```

To produce labels, *LEGEND* uses the NAME property of each plot in the graphic. The legend is positioned in the data coordinate system: the *x*-location of the upper-left corner of the legend is 1.9 µm. The legend is placed, by default, on the current plot.

Next, use *POLYLINE* to display five light gray lines marking wavelengths every 0.5 µm, starting at 0.5 µm:

```
IDL> for j=0.5, 2.5, 0.5 do $
>      !null = polyline([[j,0],[j,1]], /data, color='light gray', target=s1)
```

The TARGET property tells *POLYLINE* in which graphic to draw the lines. FOR loops are covered in Chapter 6, "Programming." The null variable `!NULL` (discussed in Chapter 5, "Data Structures") is used to discard the references for these lines.

As a last step, shade the region of visible wavelengths (between 0.4 and 0.7 µm) with *POLYGON*:

```
IDL> xverts = [0.4, 0.7, 0.7, 0.4]
IDL> yverts = [0.0, 0.0, 1.0, 1.0]
IDL> !null = polygon(xverts, yverts, /data, target=s1, /fill_background, $
>      fill_color='light blue', transparency=80)
```

POLYGON walks around the vertices to produce a rectangle. Using the FILL_COLOR, FILL_BACKGROUND, and TRANSPARENCY properties, the rectangle is filled with a light blue color that is 80 percent transparent. Compare your results with Figure 4-2 on the next page.

Output

Finally, we can save the graphic we produced to a file:

```
IDL> s1.save, 'display_spectra.png'
```

The graphic is saved to a PNG file (the file type is determined by its extension) in the current directory using a default resolution of 600 dots per inch. Locate the file with your OS file browser and view the result.

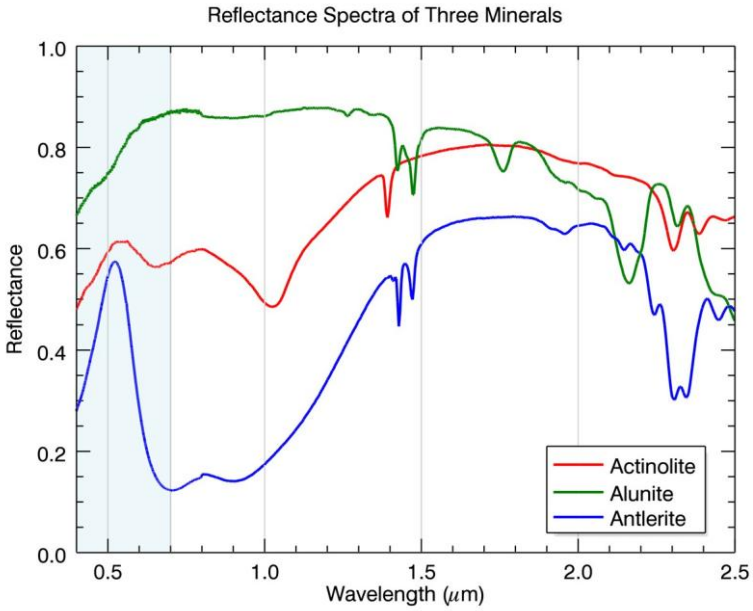


Figure 4-2:

Three reflectance spectra in one graphic.

The code used in this section is taken from the program *DISPLAY_SPECTRA* in the course files. You can examine the code and run the program to produce a similar result.

Boulder temperature data

Daily weather records have been kept in Boulder, Colorado since the late 1800s. The NOAA Earth System Research Laboratory / Physical Science Division in Boulder has collected these daily data from many sources over the years and made them freely available at http://www.esrl.noaa.gov/psd/boulder/getdata.html.

The file **boulderdaily1999.sav** contains one year of daily minimum and maximum temperatures (in Fahrenheit) in Boulder, derived from the ESRL data product. Restore the contents of this IDL SAVE file into your current IDL session with:

```
IDL> file = file_which('boulderdaily1999.sav')
IDL> restore, file
```

This file contains the variables DAY_OF_YEAR, TMIN and TMAX:

```
IDL> help, day_of_year, tmin, tmax
DAY_OF_YEAR     INT       = Array[365]
TMIN            FLOAT     = Array[365]
TMAX            FLOAT     = Array[365]
```

There's a more in-depth discussion of SAVE files (a very convenient way of storing and retrieving data in IDL) in Chapter 8, "File Access."

Visualize the daily maximum temperatures with a call to *PLOT*:

```
IDL> p = plot(day_of_year, tmax, color='red', $
>      title='Boulder Daily Temperature Extremes, 1999', $
>      xtitle='Day of Year', ytitle='Temperature ($\deg$F)')
```

Then, setting OVERPLOT, display the minimum temperatures with another call to *PLOT*:

```
IDL> q = plot(day_of_year, tmin, color='blue', /overplot)
```

In a slightly more complex use of *POLYGON*, shade the area between the curves, emphasizing the range of temperature values:

```
IDL> xverts = [day_of_year, reverse(day_of_year)]
IDL> yverts = [tmin, reverse(tmax)]
IDL> poly = polygon(xverts, yverts, /data, target=p, /fill_background, $
>    fill_color='light gray', transparency=60, linestyle='none')
```

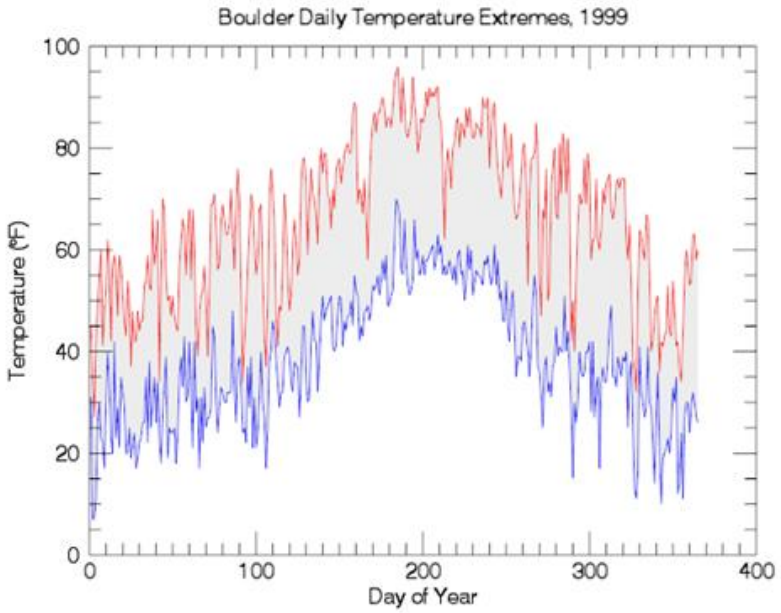


Figure 4-3:

One year of daily temperature extrema in Boulder.

As before, *POLYGON* walks around a set of vertices, but now the shape is defined by the minimum and maximum temperature traces. The *REVERSE* function simply reverses an array of values; e.g., `[1,2,3]` becomes `[3,2,1]`.

Scatter plots

As a sanity check for the Boulder temperature data, make sure that none of the minimum temperatures are higher than the maximum temperatures for a given day. We can perform this check graphically by making a scatter plot of the minimum versus maximum temperatures and observing that no point falls below the 1:1 line.

To make a scatter plot, use *PLOT* with the LINESTYLE property set to 'none' and your choice for the SYMBOL property:

```
IDL> scatter = plot(tmin, tmax, $
>      linestyle='none', symbol='star', /sym_filled, color='green', $
>      xminor=0, yminor=0, xtitle='$T_{min}$', ytitle='$T_{max}$', $
>      title='Minimum vs. Maximum Daily Temperature, 1999')
```

Here we use green, filled stars for the plot symbol.

Next, overplot the 1:1 line. There are several ways to do this. Here's one:

```
IDL> xmax = max(scatter.xrange)
IDL> ymin = min(scatter.yrange)
IDL> !null = plot([ymin,xmax],[ymin,xmax], color='blue', /overplot)
```

The Figure shows that minimum temperatures are never greater than the maximum temperatures.

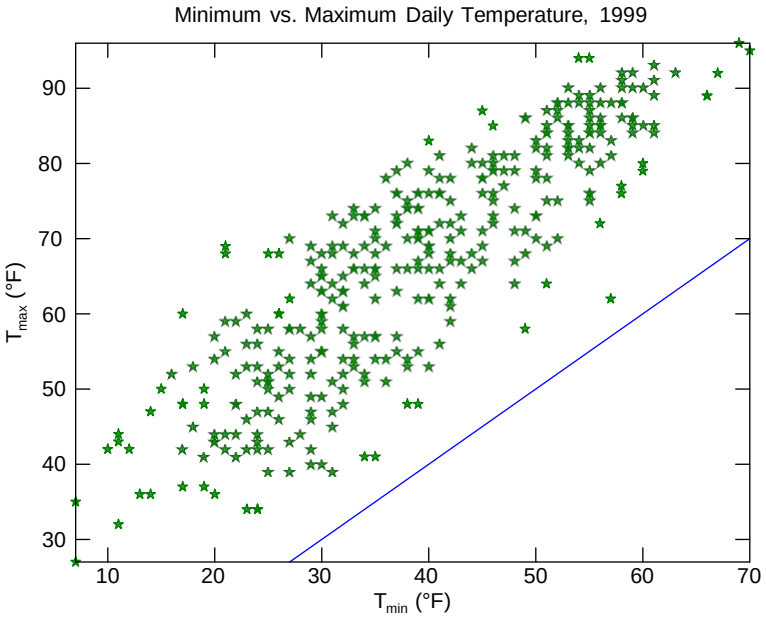


Figure 4-4:

A scatter plot of minimum versus maximum daily temperatures, with a 1:1 line displayed in blue.

Note that this result could also be obtained quickly with a relational statement:

```
IDL> total(tmin gt tmax)
   0.000000
```

The GT operator compares `TMIN` to `TMAX`, element-by-element, returning `1` for true, `0` for false. By summing the result of this expression with *TOTAL*, we find that in no case was `TMIN` larger than `TMAX` (otherwise the sum would be nonzero). The use of relational operators like GT is covered in more detail in Chapter 6, "Programming."

Bar plots

Monthly mean values of the Boulder temperature extrema can be represented visually with a bar plot or bar chart. The file **bouldermonthly1999.sav** contains one year of calculated monthly mean minimum and maximum temperatures, as well as their calculated standard deviations.

Restore the contents of this SAVE file into your current IDL session with:

```
IDL> file = file_which('bouldermonthly1999.sav')
IDL> restore, file
```

This file contains five variables:

```
IDL> help, months, mean_tmin, mean_tmax, stdev_tmin, stdev_tmax
MONTHS          STRING    = Array[12]
MEAN_TMIN       FLOAT     = Array[12]
MEAN_TMAX       FLOAT     = Array[12]
STDEV_TMIN      FLOAT     = Array[12]
STDEV_TMAX      FLOAT     = Array[12]
```

The MONTHS variable contains a list of abbreviated month names, as strings:

```
IDL> print, months
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Use *BARPLOT* to display the monthly mean maximum temperatures as a bar chart:

```
IDL> b0 = barplot(mean_tmax)
```

Let's make a more meaningful graphic by plotting the monthly mean minimums next to the maximums, and, further, plot these values versus their corresponding month names on the horizontal axis of the chart:

```
IDL> b1 = barplot(mean_tmax, nbars=2, index=0, fill_color='crimson', $
>      xtickname=months, xtickvalues=indgen(12), xminor=0, $
>      xtitle='Month', ytitle='Temperature ($\deg$F)', $
>      title='Boulder Monthly Mean Temperature Extremes, 1999')
IDL> !null = barplot(mean_tmin, nbars=2, index=1, fill_color='cornflower', $
>      /overplot)
```

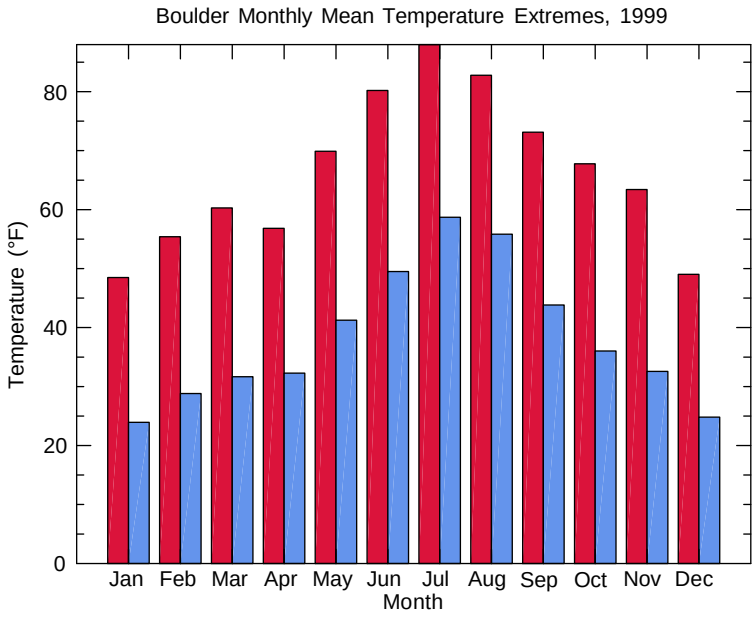


Figure 4-5:

Monthly mean temperature extremes displayed as a bar chart.

In these statements, the NBARS property sets the number of variables to be plotted in the chart. The INDEX property, then, is used to refer the individual variables. To get the month names displayed properly, the XTICKNAME and XTICKVALUES properties are needed.

For more examples of bar plots, check the *BARPLOT* entry in the IDL Help system.

Error plots

The monthly mean temperatures and a measure of their spread can be visualized with a line plot with error bars. The *ERRORPLOT* function is designed to do this.

Create an error bar plot of the monthly mean maximum temperatures in Boulder for 1999 with the following statements:

```
IDL> xmin = -1
IDL> xmax = 12
IDL> ymin = min(mean_tmax - stdev_tmax) * 0.95
IDL> ymax = max(mean_tmax + stdev_tmax) * 1.05
IDL> e = errorplot(mean_tmax, stdev_tmax, $
>      linestyle='none', symbol='square', /sym_filled, $
>      xrange=[xmin,xmax], yrange=[ymin,ymax], $
>      xminor=0, yminor=0, $
>      xtickname=months, xtickvalues=indgen(12), $
>      xtitle='Month', ytitle='Temperature ($\deg$F)', $
>      title='Boulder Monthly High Temperatures, 1999')
```

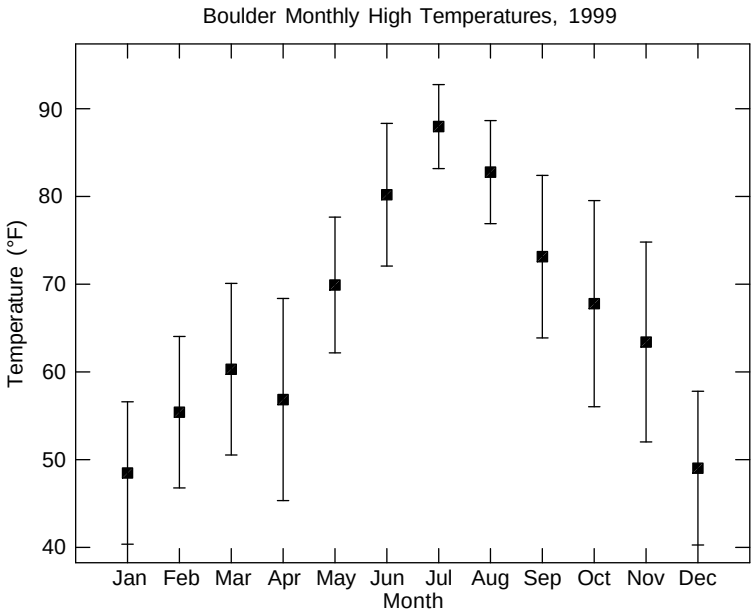


Figure 4-6:

Monthly mean high temperatures plus standard deviations.

The symbols (filled squares, as set by the SYMBOL and SYM_FILLED properties) show the mean. The error bars depict one standard deviation from the mean. As in Figure 4-5, to display the month names on the *x*-axis, the XTICKNAME and XTICKVALUES properties are needed. The XRANGE and YRANGE properties, with the multipliers on `YMIN` and `YMAX`, are used to provide extra space around the visualized data values. (To see the difference, try executing the *ERRORPLOT* statement without these properties set.) This is for appearances only.

For more examples of error bar plots, check the *ERRORPLOT* entry in the IDL Help system. IDL also has the *BOXPLOT* function, which creates a box and whiskers plot from data containing a minimum, lower quartile, median, upper quartile, and maximum.

The sunspot cycle

The waxing and waning of sunspots has been recorded since their discovery by Galileo in the 1600s. The Solar Physics Group at NASA's Marshall Space Flight Center provides a record of monthly sunspot numbers dating back to the year 1749. The file **spot_num.txt**, containing monthly averages and standard deviations of sunspot numbers, can be freely downloaded from http://solarscience.msfc.nasa.gov/greenwch/spot_num.txt. The numbers in this file are derived from the International Sunspot Numbers, compiled by the Solar Influences Data Analysis Center in Belgium.

Read the sunspot numbers file with *READCOL*:

```
IDL> file = file_which('spot_num.txt')
IDL> readcol, file, year, month, sunspots
IDL> help, year, month, sunspots
YEAR            FLOAT     = Array[3135]
MONTH           FLOAT     = Array[3135]
SUNSPOTS        FLOAT     = Array[3135]
```

Take a quick look at the sunspot numbers:

```
IDL> p = plot(sunspots, title='Sunspot Numbers')
```

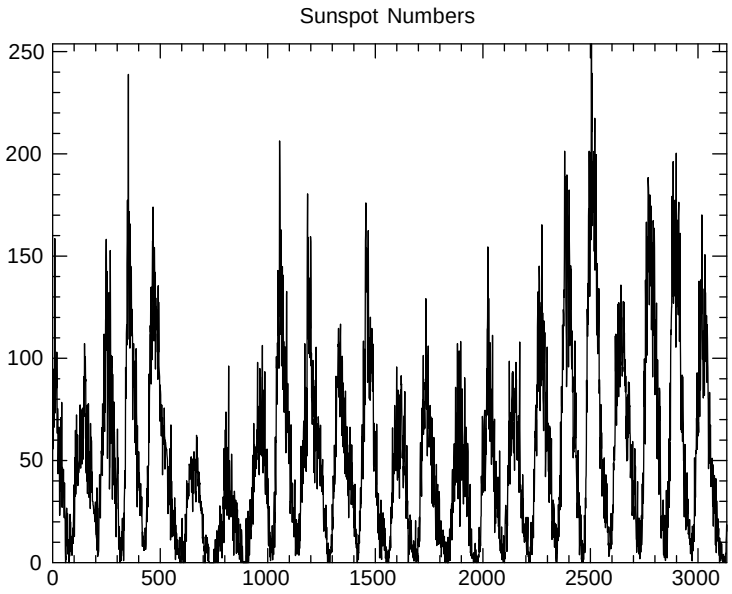


Figure 4-7:

A plot of the sunspot number time series.

The periodic pattern of sunspot activity is evident in the Figure. We'll explore the frequency-domain properties of this series in Chapter 10, "Analysis."

It would be better to plot the sunspot number versus time. Make a time vector from the YEAR and MONTH values read from the file:

```
IDL> time = year + (month-1.0)/12.0
```

Further, let's restrict the time interval we examine to the years 1970 to present.

There are several ways to do this in IDL, but one common technique is to use the *WHERE* function:

```
IDL> i = where(time ge 1970.0, n_sunspots)
IDL> print, n_sunspots
         483
```

WHERE returns the index locations that match its input expression. We'll cover *WHERE* in greater detail in Chapter 5, "Data Structures." Using the output from *WHERE*, plot the measured sunspot activity since 1970:

```
IDL> time_recent = time[i]
IDL> sunspots_recent = sunspots[i]
IDL> p = plot(time_recent, sunspots_recent, 'g-2o', $
>      xtitle='Year', ytitle='Sunspots', $
>      title='Sunspot Activity (1970-present)')
```

The string `'g-2o'` is an optional format parameter—it's used to provide a format for the graphic without verbose keywords. Here, the plot is green (g) using a solid line (-) that has a double-weight thickness (2) and a circle symbol (o). These short formats can appear in any order. For a complete list of short formats, see the IDL Help.

Histogram plots

Calculate a discrete frequency distribution of sunspot numbers in the selected time interval using the *HISTOGRAM* function:

```
IDL> sunspot_histogram = histogram(sunspots_recent, binsize=10.0, $
>      locations=sunspot_bins)
```

The binsize of 10.0 was empirically chosen; it gives the following bins:

```
IDL> sunspot_bins
   0.000000      10.0000      20.0000      30.0000      40.0000      50.0000
   60.0000      70.0000      80.0000      90.0000      100.000      110.000
   120.000      130.000      140.000      150.000      160.000      170.000
   180.000      190.000      200.000
```

where each value is the start of the bin; e.g., the first bin is defined on `[0.0, 10.0)`. The number of sunspots in each of these bins is given by `SUNSPOT_HISTOGRAM`:

```
IDL> print, sunspot_histogram
   59          72          38          32          34          24          26
   16          26          28          19          14          22          21
   13          12          13          10           3           1           1
```

As a sanity check, sum of the histogram should equal the number of sunspots:

```
IDL> print, total(sunspot_histogram), n_sunspots
      483.000         483
```

Visualize this histogram with *PLOT*:

```
IDL> h = plot(sunspot_bins, sunspot_histogram, /histogram, $
>      /fill_background, fill_color='lavender', fill_level=0, $
>      xtitle='Monthly Sunspot Number', ytitle='Frequency', $
>      title='Sunspots Per Month (1970-present)')
```

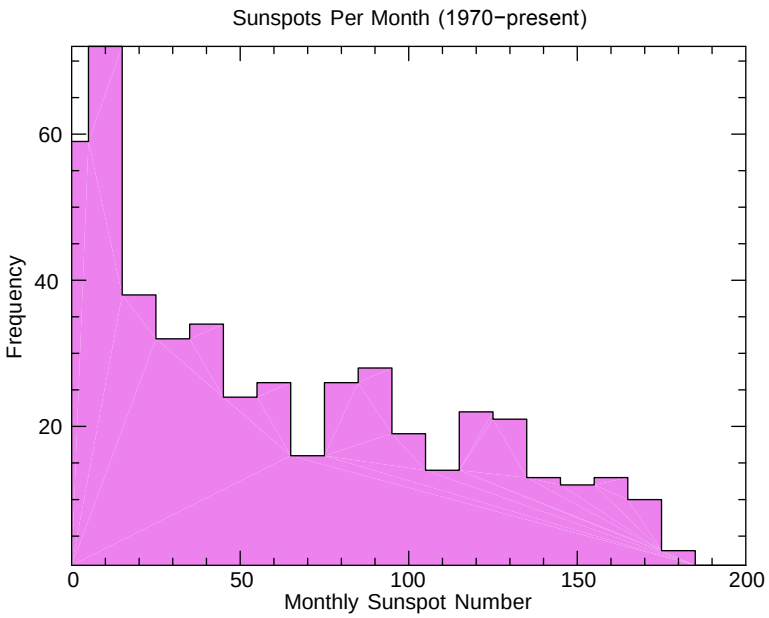


Figure 4-8:

Frequency distribution of monthly sunspot numbers from 1970 to present.

The HISTOGRAM property tells *PLOT* to draw lego-like lines instead of point-to-point lines.

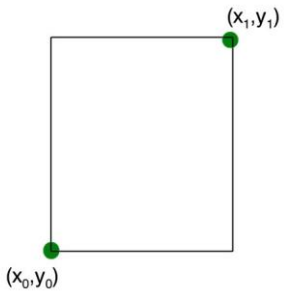
Positioning plots

To this point in this chapter, we've been displaying one graphic, one set of axes, per window. With the POSITION property it's possible to position multiple graphics in the same window.

Combine the sunspot series and its histogram in one window. First, plot the series:

```
IDL> xrange = [min(time_recent), max(time_recent)]
IDL> yrange = [min(sunspots_recent), max(sunspots_recent)]
IDL> series = plot(time_recent, sunspots_recent, $
>      position=[0.1, 0.15, 0.75, 0.90], dimensions=[800,600], $
>      xrange=xrange, yrange=yrange, $
>      xtitle='Year', ytitle='Sunspots', $
>      title='Sunspot Activity (1970-present)')
```

The POSITION property describes the lower left and upper right corners of the bounding box of the plot—the plot fits within this box. The syntax for POSITION is `[x0, y0, x1, y1]`, using the diagram on the right. The values for POSITION are in normalized coordinates, which range from `0` to `1` in each direction across a graphics window; e.g., `[0.5, 0.5]` would describe the center of a graphics window.



The DIMENSIONS property sets the size, in pixels, of the window containing this plot.

Now place the sunspot histogram next to the time series:

```
IDL> histoplot = plot(sunspot_histogram, sunspot_bins, /histogram, $
>      position=[0.80, 0.15, 0.95, 0.90], /current, $
>      yrange=yrange, ymajor=0, $
>      /fill_background, fill_color='light gray', $
>      xtitle='Frequency', title='Histogram')
```

Here, the values of POSITION set the bounding box for this plot on the far right of the window. The CURRENT property tells *PLOT* to reuse the current window. Setting YMAJOR to zero removes the axis labels from the plot.

Save the combined graphic to an encapsulated PostScript file:

```
IDL> series.save, 'sunspot-series-plus-histogram.eps'
```

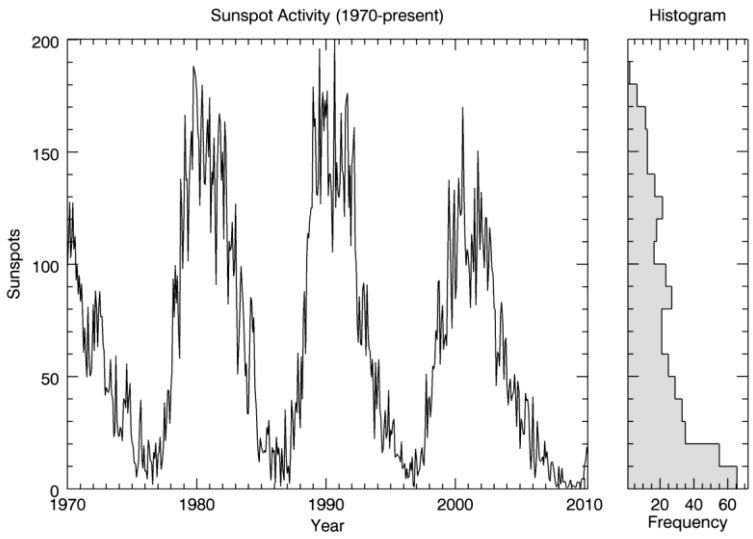


Figure 4-9:

The time series of recent sunspot activity and its histogram, in one graphic.

Swapping the order of the parameters in `HISTOPLOT` transposes the histogram. Moving horizontally across the series gives a visual estimate of the histogram value displayed in the plot on the right.

The code used to produce the plots in this section can be found the program *DISPLAY_SUNSPOT_SERIES* in the **introduction/src** directory.

Exercises

1. In oceanography, profiles of temperature and salinity are plotted with depth increasing in the negative *y*-direction. How could such a graphic be constructed in IDL? Use this statement to restore a temperature profile:

   ```
   IDL> restore, file_which('depth_profile.sav'), /verbose
   ```

   Use the variables `TEMPERATURE` and `DEPTH` in your graphic.

2. Display the cardioid given by the equation $\rho = 1 + \cos\theta$ in polar coordinates with the *POLARPLOT* function.
3. The file **iss_LLA_Int_beta.dat** in the IDL coursefiles **data** directory contains the position of the International Space Station over approximately 1.5 hr on 2008 September 10. Locate and read this file with:

```
IDL> file = file_which('iss_LLA_Int_beta.dat')
IDL> rdfloat, file, time, lat, lon, alt, skipline=9, /double
```

   Using the variables `LON`, `LAT` and `ALT`, display the position of the ISS with the *PLOT3D* function. For a nice example of using *PLOT3D*, see its entry in the IDL Help system.

For sample solutions to the Exercises, see the programs *DEPTH_PROFILE_EX*, *POLARPLOT_EX* and *PLOT3D_EX* in the IDL coursefiles **introduction/src** directory.

References

About Reflectance Spectroscopy. Roger N. Clark, U.S. Geological Survey, 1998. <http://speclab.cr.usgs.gov/aboutrefl.html>. Accessed 2010-04-13.

Information on reflectance spectra and how they're measured in the laboratory. The

Sunspot Cycle. David H. Hathaway, NASA Marshall Space Flight Center, 2010. <http://solarscience.msfc.nasa.gov/SunspotCycle.shtml>. Accessed 2010-04-14.

Information on sunspots and the historical record of sunspot numbers.

Tufte, Edward R. *The Visual Display of Quantitative Information*. Second edition. Cheshire, Connecticut: Graphics Press, 2002.

A modern classic on the theory and design of infographics.

Sparklines implementation. Michael D. Galloy, 2006. <http://michaelgalloy.com/2006/04/19/sparklines-implementation.html>. Accessed 2010-04-18.

Sparklines are "simple, word-size graphics" promoted by Edward Tufte. Mike has implemented sparklines (and dichotomous sparklines) in IDL. Mike has a variety of other cool visualization examples on his website.

Using IDL 8 Graphics (a.k.a. New Graphics). Mark Piper, 2011. <http://idldatapoint.com/2011/09/29/using-idl-8-graphics-a-k-a-new-graphics/>. Accessed 2011-10-14.

A short example of using (New) Graphics.

Chapter 5: Data Structures

This chapter describes how to create and work with variables in IDL.

> *The primary purpose of the Data statement is to give names to constants; instead of referring to pi as 3.141592653589793 at every appearance, the variable Pi can be given that value with a Data statement and used instead of the longer form of the constant. This also simplifies modifying the program, should the value of pi change.*
>
> — Fortran manual for Xerox Computers

Variables 44
Data types 45
Arrays 48
Lists and hashes 53
Structures 54
Strings 56
Pointers 58
Objects 59
Exercises 59
References 60

Variables

Variables are named containers for storing data. IDL variables have an associated *type* and *organization* (scalar, array, structure, list or hash). IDL has a rich set of data types for integer, floating point, string and abstract data. Conversion between the types is straightforward. Variables do not have to be declared in IDL: the type and organization of a variable is determined when it's created.

IDL is an array-based language. Operators and many library routines work on both arrays and scalars. The size of a variable (i.e., the amount of memory allocated to a variable) is limited only by the computer and operating system you are using, not by IDL.

Variable names

Variable names must start with a letter or an underscore. They may contain from 1 to 128 letters, digits, underscores, or dollar signs. Variable names are case-insensitive; IDL converts all alphabetic characters to uppercase internally.

Here are examples of valid variable names:

```
n_values            isFunction          _red
help                f0oBaR              read$_file5
```

and invalid variable names:

```
name.last           third%file          4th_list
$temp               case-1              Function
```

Can you explain why these are invalid?

System variables

System variables are a set of predefined variables available to all programs. They're identified by an exclamation point before their name. Some frequently used system variables are listed in Table 5-1, along with a short description of their purpose.

Table 5-1: A list of frequently used system variables.

System variable	Description
`!DTOR`	The conversion factor from degrees to radians.
`!PI, !DPI`	Single and double precision values of π.
`!DIR`	Stores the location of the main IDL directory.
`!NULL`	A null variable.
`!COLOR`	Stores a set of web-standard colors, by name, as RGB triples.
`!VALUES`	Holds machine representations of infinity and NaN (not a number).
`!VERSION`	Gives information about the version of IDL currently in use.

For a complete list of system variables, check the IDL Help system.

Data types

There are 15 built-in data types in IDL: seven integer, two floating point, two complex, a string type, two abstract types (pointers and objects), and undefined or null. Structures are considered their own type, lists and hashes are objects, while arrays are considered the same type as their elements.

An overview of built-in IDL data types is given in Table 5-2.

Table 5-2: IDL built-in data types.

Data type	Size (bytes)	Range	Scalar creation notation	Type code	Conversion function
byte	1	0–255	`a = 5B`	1	*BYTE*
integer	2	±2^15-1	`b = 0S ; b = 0`	2	*FIX*
unsigned integer	2	0–2^16-1	`c = 0U`	12	*UINT*
long integer	4	±2^31-1	`d = 0L`	3	*LONG*
unsigned long integer	4	0–2^32-1	`e = 0UL`	13	*ULONG*
64-bit integer	8	±2^63-1	`f = 0LL`	14	*LONG64*
unsigned 64-bit integer	8	0–2^64-1	`g = 0ULL`	15	*ULONG64*
float	4	±1E±38	`h = 0.0`	4	*FLOAT*
double	8	±1E±308	`i = 0.0D`	5	*DOUBLE*
complex	8	±1E±38	`j = complex(1.0,0.0)`	6	*COMPLEX*
double complex	16	±1E±308	`k = dcomplex(1.0,0.0)`	9	*DCOMPLEX*
string			`l = 'hello'` `l = "hello"`	7	*STRING/ STRTRIM*
pointer	4		`m = ptr_new()`	10	
object	4		`n = obj_new()`	11	
undefined or null	1			0	

A few notes on this table:

- The size in memory for each type is a nominal amount. There will be a slight overhead for the creation of an IDL variable.
- The default integer type in IDL is a two-byte or short integer.
- The type code of a variable or expression is returned by the *SIZE* function.
- There is no Boolean type in IDL. Typically, a value of 1B is used for true, 0B for false.
- The smallest/largest float and double values, as well as the resolution of these types, can be determined with the *MACHAR* function.

Type behaviors in IDL

IDL is a *dynamically typed* language. This means that an operation on a variable can change that variable's type.

In general, when variables of different types are combined in an expression, the result has the data type that yields the highest precision. For example, if an integer variable is added to a floating-point variable, the result is a floating-point variable:

```
IDL> a = 5 + 3.0
IDL> help, a
A               FLOAT     =       8.00000
```

This notion of type promotion is implicit in IDL. Given a mixed set of data types in an expression, IDL automatically promotes them to the highest type. No warning or error message is given when a type promotion occurs.

Be careful with IDL integer types. Examine the result of this innocuous statement:

```
IDL> b = 30000 + 10000
IDL> help, b
B               INT       =   -25536
```

How does IDL arrive at this incredible answer? Type promotion didn't occur since both operands are integers; the result therefore must also be an integer. The problem is the value that we expect, 40000, is outside the range of possible values for the IDL integer type. From Table 5-2, we see that the largest value we can create with the integer type is $2^{15} - 1 = 32767 < 40000$. (See also http://xkcd.com/571/.)

One way to parse this result is by looking at the binary representation of these numbers. Display the numbers 40000 and -25536 in base 2 with

```
IDL> print, 40000L, -25536, format='(B16.16)'
1001110001000000
1001110001000000
```

Note that they are the same binary number. However, in the long integer 40000, all of the digits are used for the range of the number, whereas in the short integer -25536, the first digit is the sign, with the value 1 indicating this is a negative number.

The upshot of this example is to warn you to be careful with type and, when in doubt, use long integers.

```
IDL> b = 30000L + 10000
IDL> help, b
B               LONG      =        40000
```

For information on altering this behavior in a more general fashion in IDL, see the section "Parameter passing" on page 70.

Be careful with floating-point data types, as well. Examine the following statement:

```
IDL> c = 10000.0 + 0.0001
IDL> c
   10000.0
```

The fractional value is missing! This is an example of an issue that's faced in IDL and other programming languages: it's a consequence of the way floating-point numbers are represented on a computer.

Floating-point numbers (both single and double, real and complex) are constructed so that they can represent a wide range of numbers, but at the cost of limiting precision. As a rule of thumb, you can expect to retain roughly 7 digits of precision in an operation with single precision float values and roughly 16 digits with double precision float values.

In general, it is recommended that double-precision arithmetic be used for all floating-point operations. At the end of a calculation, if less precision is needed, then the results can be cast back to single-precision floats.

A thorough discussion of this behavior is a bit outside the scope of this course. See the references at the end of this chapter for a better treatment, especially The Floating-Point Guide at http://floating-point-gui.de/.

Null variables

IDL has the concept of a null variable: a variable that is undefined, it has no data. A null variable can be expressed through the `!NULL` system variable:

```
IDL> help, !null
<Expression>    UNDEFINED = !NULL
IDL> !null
!NULL
```

or as a zero-length array:

```
IDL> a = [] ; same as !null
IDL> help, a
A               UNDEFINED = !NULL
```

A null variable can be handy, for example, in building an array by concatenation:

```
IDL> b = []
IDL> for i = 0, 9 do b = [i, b]  ; don't need to test whether "b" is defined
IDL> b
   9       8       7       6       5       4       3       2       1       0
```

Although this technique is highly inefficient in IDL, the penalty isn't great for small amounts of data.

A null variable can also be used to throw away the return value from a function:

```
IDL> file = filepath('image.tif', subdirectory=['examples','data'])
IDL> !null = query_image(file, info)
IDL> help, info
INFO            STRUCT    = -> <Anonymous> Array[1]
```

More information on null variables can be found in the IDL Help system (e.g., search for "null variables").

Arrays

IDL is an array-based language; it has useful syntax for indexing and operating on arrays. Appropriate use of the syntax makes code both faster and easier to read.

Arrays can be created by setting a variable equal to a list of values enclosed within square brackets. This statement creates a one-dimensional array, or vector, of six integers:

```
IDL> x = [4, 8, 15, 16, 23, 42]
IDL> help, x
X               INT       = Array[6]
```

Multidimensional arrays can also be created with square brackets, using nested sets of brackets, as in:

```
IDL> id4 = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]
IDL> help, id4
ID4             INT       = Array[4, 4]
```

The `ID4` variable becomes a 4×4 integer matrix.

Consider this statement:

```
IDL> y = [3.4, 6.7D, 45U, 90L]
```

Here, an array is created from four different variable types. IDL converts each value to the highest type in the group; in this case, double. The type of `Y` is double.

Arrays of the different IDL data types can be created with the routines listed in Table 5-3 on page 49. The first column in the table lists the data type, the second the array generator for the type, the last the index generator for the type. Array generators zero an array, while index generators fill an array with a sequence of integral values starting at zero. Here's a small example of their use:

```
IDL> vec1 = fltarr(5)    ; floating-point array generator
IDL> vec2 = findgen(5)   ; floating-point indexed array generator
```

The arrays have the same type and number of elements, so they look the same to *HELP*:

```
IDL> help, vec1, vec2
VEC1            FLOAT     = Array[5]
VEC2            FLOAT     = Array[5]
```

but printing the values of the arrays shows the difference:

```
IDL> print, vec1, vec2
   0.000000  0.000000     0.000000     0.000000     0.000000
   0.000000  1.000000     2.000000     3.000000     4.000000
```

The best way to initialize an array to a constant value is to use one of the array creation functions listed in Table 5-3 and add the value when the array is created. For example, to initialize an array to the integer value 1, use

```
IDL> ones = intarr(5) + 1
IDL> print, ones
   1       1       1       1       1
```

Table 5-3: Routines for creating arrays by IDL variable type.

Type	Array generating function	Index generating function
byte	*BYTARR*	*BINDGEN*
integer	*INTARR*	*INDGEN*
unsigned integer	*UINTARR*	*UINDGEN*
long integer	*LONARR*	*LINDGEN*
unsigned long integer	*ULONARR*	*ULINDGEN*
64-bit integer	*LON64ARR*	*L64INDGEN*
unsigned 64-bit integer	*ULON64ARR*	*UL64INDGEN*
float	*FLTARR*	*FINDGEN*
double	*DBLARR*	*DINDGEN*
complex	*COMPLEXARR*	*CINDGEN*
double complex	*DCOMPLEXARR*	*DCINDGEN*
string	*STRARR*	*SINDGEN*
pointer	*PTRARR*	
object	*OBJARR*	
undefined or null		

In IDL 8.3, arrays can also be generated using colon notation. The format for array creation with colons is [start, finish, step size]. If step size is not included, it is assumed to be 1.

```
IDL> [1:10]
      1       2       3       4       5       6       7       8       9      10
IDL> [10:0:-2]
      10       8       6       4       2       0
```

Subscripting

Array variables can be subscripted to access one element, a range of elements, or any number of non-sequential elements. Arrays are subscripted with square brackets `[]`, with indexing beginning at 0, the start of the array.

```
IDL> a = [0:9]
IDL> help, a
A               INT       = Array[10]
IDL> print, a[0], a[5]
   0       5
```

Subscripting can take place on the left or right side of the equal sign:

```
IDL> a[7] = a[4] * 12
```

A colon (`:`) is used to specify a contiguous range of subscripts:

```
IDL> data = a[4:7]
IDL> a[1:3] = [3, 84, 33]
```

The first statement creates a new four-element integer array, `DATA`, using elements 4 through 7 copied from `A`. The next statement sets elements 1 through 3 of `A` to the array of numbers on the right side of the equation.

An array range may be assigned an array of equal length, as above, or a single value. This statement sets elements 5 through 9 of `A` equal to the value -12:

```
IDL> a[5:9] = -12
```

An error occurs if the array range is set to an array with less than or greater than the number of elements in the range.

Strides may also be used in subscripting. Here, every second element of `A` between indices 5 and 9 is set to the value 3:

```
IDL> a[5:9:2] = 3
```

An asterisk (*) used in a subscript range indicates the end of the array. The following statement sets elements 3 to the end of the array (9) to the value 42:

```
IDL> a[3:*] = 42
```

Used alone, the asterisk represents the entire array:

```
IDL> a[*] = 42
```

Subscripted array elements don't have to be contiguous. Non-sequential elements of an array can be accessed by subscripting an array with another array:

```
IDL> array = [-14, 41, -90, 67, 10]
IDL> indices = [4, 2, 0, 1]
IDL> print, array[indices]
      10     -90     -14      41
```

The last statement prints elements 4, 2, 0 and 1 from `ARRAY`. This is a powerful method of subscripting that is used often in conjunction with the IDL *WHERE* function.

IDL also supports negative array indexing, allowing subscripting to occur from the end of the array. You can think of the negative indices wrapping around the end of the array from the start index at zero.

Let's reconstitute our scratch variable `A` for a few examples:

```
IDL> a = findgen(5)
```

Print the last element of `A` using standard indexing notation:

```
IDL> print, a[n_elements(a)-1]
```

Or, use negative array indexing:

```
IDL> print, a[-1]
```

Much nicer. Negative array indexing also works with the `:` and `*` subscripting operators. For example, print all array elements from the first up to the second-to-last:

```
IDL> print, a[0:-2]
```

and then print only the last three elements:

```
IDL> print, a[-3:*]
```

For more examples, see the code in the file **negative_array_indexing_ex.pro** in the IDL coursefiles. There's also more information on negative array indexing in the IDL Help system.

Multidimensional arrays

Arrays in IDL can have up to eight dimensions. How are such arrays subscripted?

```
IDL> m = indgen(4, 5)
IDL> print, m
   0       1       2       3
   4       5       6       7
   8       9      10      11
  12      13      14      15
  16      17      18      19
```

The method of subscripting in IDL is column major. Internally, arrays are laid out in a stream (the elements of `M` are numbered in the order they are stored in memory.) In the two-dimensional case, IDL array subscripting is specified as `[icolumn, irow]`:

```
IDL> print, m[1,0], m[0, 1]
   1       4
```

The subscripting operators `:` and `*` can also be used on multidimensional arrays:

```
IDL> m[*,4]         ; the fifth row
   16      17      18      19
IDL> m[2,*]         ; the third column
   2
   6
   10
   14
   18
IDL> m[2:3, 1:2] ; a block in the middle - col 3-4 row 2-3
   6         7
   10        11
```

Single-index subscripting

Any IDL variable can be subscripted with a single index. (This is even true for scalars, although the only valid index is 0.) The elements of a multidimensional array are indexed in the same order as they are numbered in the index generating function—the left-most index varies the fastest when traversing the array in order of the single index. For example, in the two-dimensional case above,

```
IDL> m[17]
   17
```

The *ARRAY_INDICES* function can be used to convert single index subscripts into dimensional subscripts.

```
IDL> array_indices(m, 17)
   1           4
```

The result is the `[column, row]` index values corresponding to element 17 in `M`.

The *WHERE* function

The *WHERE* function evaluates an array expression and returns the single-index subscript of each element for which the expression is true (it returns the locations, *not* the data values). If the expression is false, the return value is a one-element vector with the value -1. For example, say you have a set `N` of 15 normally distributed numbers:

```
IDL> n = randomn(123, 5, 3)
IDL> print, n
   0.147312     -1.27396     1.32809     -0.237652    -0.906046
   2.54712      -0.909852    0.478123    -0.375762    0.456295
   -0.496401    0.677828     -0.640487   0.462273     -1.09031
```

and you'd like to find the number and location of the negative values in the array. (Note that the first parameter to *RANDOMN* is a seed value for the random number generator.) You could accomplish this by looping over the elements of `N` and marking the negative values:

```
IDL> i_neg = []
IDL> for i = 0, n_elements(n)-1 do $
>      if n[i] lt 0.0 then i_neg = [i_neg, i]
IDL> n_neg = n_elements(i_neg)
```

The variable `N_NEG` gives the number of negative values, while `I_NEG` gives the location (using the single index subscript notation) of the negative values. What are they?

```
IDL> n_neg
   8
IDL> i_neg
   1      3      4      6      8      10     12     14
```

This technique works, but because of the FOR loop, it is inefficient in IDL (code performance is covered in greater detail in the *Scientific Programming with IDL* course). The *WHERE* function is vectorized. Using it:

```
IDL> i_neg = where(n lt 0.0, n_neg, /null)
```

is not only faster, but less code used to produce the result. Check that we get the same numbers:

```
IDL> n_neg
   8
IDL> i_neg
   1      3      4      6      8      10     12     14
```

Further, to recover the actual values of `N` that are less than zero, we simply subscript `N` with `I_NEG`:

```
IDL> n[i_neg]
   -1.27396  -0.237652    -0.906046    -0.909852
   -0.375762 -0.496401    -0.640487    -1.09031
```

The use of *WHERE* abounds in IDL. We'll see it several more times in this course.

Lists and hashes

Lists are containers that are similar to arrays. Like arrays, elements in a list can be accessed with indexed values. In fact, any of the array subscripting operations presented earlier in the chapter can be used with lists. Unlike arrays, lists are not restricted to a single data type.

Use *LIST* to create a new list. For example:

```
IDL> a = list('Homer', !pi, [0:9], '1123 6536 5321')
IDL> help, a
A               LIST  <ID=28977  NELEMENTS=4>
```

Note that the list `A` is a heterogeneous mix of types and organizations. *HELP* tells us that `A` is a list with four elements. Access the members of the list with array subscripts:

```
IDL> print, a[0] + ' ate ' + strtrim(2*a[1],2) + ' donuts.'
Homer ate 6.28319 donuts.
```

Empty lists are supported:

```
IDL> b = list()
IDL> help, b
B               LIST  <ID=28986  NELEMENTS=0>
```

Elements can be added to and removed from lists with the `Add` and `Remove` methods:

```
IDL> b.add, 'apple'
IDL> b.add, 'orange', 0 ; added before 'apple'
IDL> b.add, 'boysenberry'
IDL> b.remove ; on second thought, I don't like boysenberries
IDL> b
orange
apple
```

Lists can also be joined:

```
IDL> c = a + b
IDL> c[-1] ; the last element of the list apple
```

Like lists, hashes are also containers, but their elements are accessed with a key (typically a string) instead of an index. Hashes are more expensive to create and consume more memory than arrays or lists, but the key-value pair in a hash is a very convenient way to organize and access data.

Make a new hash table of colors with the *HASH* function:

```
IDL> colors = hash('red', [255,0,0], 'green', [0,255,0], 'blue', [0,0,255])
```

Use *HELP* and *PRINT* to get information on the hash:

```
IDL> help, colors
COLORS          HASH  <ID=29007  NELEMENTS=3>

IDL> print, colors
green:        0     255       0
blue:         0       0     255
red:        255       0       0
```

Note that the elements in a hash are not necessarily sequential. To access a value in the hash, use its key:

```
IDL> colors['red']
     255       0       0
```

To create an indexed hash, use *ORDEREDHASH*.

```
IDL> colors = orderedhash('red',[255,0,0],'green',[0,255,0],'blue',[0,0,255])
```

Now the hash keys will be organized in the order that they were entered. There is a bit of overhead to create ordered hashes, so if the order does not matter for what you are working on, *HASH* is a better option.

Note that, unlike lists, standard array indexing cannot be used with a hash.

Add a color to the hash:

```
IDL> colors['chartreuse'] = [127,255,0]
IDL> print, colors
red:       255       0       0
green:         0     255       0
blue:          0       0     255
chartreuse:      127     255       0
```

Test whether the key "orange" is in the hash with the `HasKey` method:

```
IDL> print, colors.haskey('orange')
   0
```

Remove the color "green" from the hash with the `Remove` method:

```
IDL> colors.remove, 'green'
IDL> print, colors
red:       255       0       0
blue:          0       0     255
chartreuse:      127     255       0
```

Lists and hashes can also be iterated with the FOREACH operator: examples are given in Chapter 6, "Programming." Lists and hashes are covered in greater depth, including where and why they might be used, in the *Application Development with IDL I* course.

Structures

Structures are containers that allow data of differing type and organization to be stored in a single variable, like lists and hashes. IDL supports two kinds of structures: *named* and *anonymous*. Named structures are used for fixed formats. Anonymous structures are used when the type and/or size of their components may change during the course of an IDL session.

For the examples in this section, use these structures; the first named, the second anonymous:

```
IDL> mycar = {car, make:'Honda', model:'Fit', year:2009}  ; named
IDL> star = {name:'Sirius', ra:6.75248, decl:-16.7161}    ; anonymous
```

Note the use of braces { } to make the structure variables. The structures are composed of fields (e.g., MAKE, MODEL, YEAR) with data assigned to them with a colon. What's in these variables? Use *HELP* with the STRUCTURES keyword set to find out:

```
IDL> help, mycar, star, /structures
** Structure CAR, 3 tags, length=28, data length=26:
   MAKE            STRING    'Honda'
   MODEL           STRING    'Fit'
   YEAR            INT           2010
** Structure <1b8ea6f0>, 3 tags, length=20, data length=20, refs=1:
   NAME            STRING    'Sirius'
   RA              FLOAT           6.75248
   DECL            FLOAT          -16.7161
```

The output from *HELP* clearly shows the field names, types and values.

Access the fields of a structure with the period (.) operator:

```
IDL> mycar.make
Honda
```

Structure fields can be used in expressions:

```
IDL> x = mycar.year * 15.5
IDL> x
   31155.0
```

We can change the data in a structure field, but not the type or size of the data:

```
IDL> mycar.year = 2010.5
IDL> help, mycar.year
<Expression>    INT       =     2010
```

The YEAR field of MYCAR remains integer.

Anonymous structures can be redefined or appended:

```
IDL> star = create_struct(star, 'mag', -1.46)
IDL> help, star
** Structure <19bb64d8>, 4 tags, length=24, data length=24, refs=1:
   NAME            STRING    'Sirius'
   RA              FLOAT           6.75248
   DECL            FLOAT          -16.7161
   MAG             FLOAT          -1.46000
```

but named structures can't. They're fixed in an IDL session:

```
IDL> mycar = {car, make:'Honda', model:'Fit', color:'blue'} ; #fail
% Conflicting or duplicate structure tag definition: YEAR.
% Execution halted at: $MAIN$
```

This is by design. With a named structure, the data within the fields may vary, but we want the same set of fields every time. Objects in IDL, for example, are based on named structures.

Arrays of structures can be created with *REPLICATE*. Make a parking lot with spaces for five cars:

```
IDL> lot = replicate({car}, 5)
IDL> help, lot
LOT             STRUCT    = -> CAR Array[5]
```

Park a few cars in the lot. The array subscripting operators : and * work here.

```
IDL> lot[0] = mycar
IDL> lot[1] = {car, 'Toyota', 'Avalon', 1999}
IDL> lot[2:*].make = 'Buick'
IDL> lot.make
Honda Toyota Buick Buick Buick
```

Arrays of structures can be handy, for example, in packaging a series of records from a database.

Strings

Strings are a primitive data type in IDL. They're used in storing and manipulating file names, parsing data in files and setting Graphics property values.

String literals are created with single (') or double (") quote marks, though single quote marks are recommended (see Exercises). Quotes can be nested one deep:

```
IDL> 'Robert "the Father of Modern Rocketry" Goddard'
Robert "the Father of Modern Rocketry" Goddard
```

Strings are case-sensitive in IDL. For example, these strings are not the same

```
IDL> 'Hello' eq 'hello'
   0
```

because of the ASCII codes for uppercase versus lowercase 'h':

```
IDL> print, byte('H'), byte('h')
   72     104
```

The + operator concatenates strings. It can be used with scalars or arrays. For example,

```
IDL> print, 'Goodnight' + ' ' + ['moon', 'stars'] + '.'
Goodnight moon. Goodnight stars.
```

IDL has a library of powerful routines for string processing, allowing string conversion, searching, splitting, joining, comparison and regular expression pattern matching. They're listed in Table 5-4 below. Several examples follow.

The *STRING* function converts a numeric type to string:

```
IDL> a = 42U
IDL> b = string(a)
IDL> help, a, b
A               UINT      =       42
B               STRING    = '      42'
```

Table 5-4: IDL string processing routines.

Function	Purpose
STRCMP	Compares two strings
STRCOMPRESS	Removes white space from a string
STREGEX	Performs string regular expression pattern matching
STRING	General string conversion function
STRJOIN	Collapses a string array into a merged scalar string
STRLEN	Returns the number of characters in a string
STRLOWCASE	Converts a string to lower case
STRMATCH	Compares a search string against an input string expression
STRMID	Extracts a substring from a string
STRPOS	Finds the first occurrence of a substring in a string, searched from the start or end of the string
STRPUT	Inserts the contents of one string into another
STRSPLIT	Splits its input string into an array of separate substrings
STRTRIM	Converts to string (if necessary) and removes leading and/or trailing blanks
STRUPCASE	Converts a string to upper case

STRING is often used with its FORMAT keyword. For example, add a leading zero to the integers in this array:

```
IDL> a = indgen(6)*2
IDL> b = string(a, format='(i2.2)')
IDL> print, b
00 02 04 06 08 10
```

These strings could be used to create an array of indexed filenames:

```
IDL> filenames = 'file' + b
IDL> print, filenames
file00 file02 file04 file06 file08 file10
```

STRING can use C-like or Fortran-like format codes; for details, see its in the IDL Help system.

Here is the first sentence of Abraham Lincoln's Gettysburg Address:

```
IDL> s = 'Four score and seven years ago our fathers brought forth on this' $
>      + ' continent a new nation, conceived in liberty, and dedicated' $
>      + ' to the proposition that all men are created equal.'
```

Note the use of the concatenation + and continuation $ characters in this statement. Find and extract 'liberty' from this string. Use *STRLEN* to determine the length of this string:

```
IDL> searchlen = strlen('liberty')
IDL> print, searchlen
   7
```

Use *STRPOS* to determine the location of this string, counting characters from the beginning of `S`, starting at zero.

```
IDL> loc = strpos(s, 'liberty') ; start counting at zero
IDL> print, loc
         102
```

So the 'l' in 'liberty' is the 103rd character in `S`. Extract 'liberty' from `S` with *STRMID*:

```
IDL> print, strmid(s, loc, searchlen)
liberty
```

More examples of using the string processing routines listed in Table 5-4 can be found in their respective entries in the IDL Help system. String processing is also covered in more detail in the *Scientific Programming with IDL* course.

Pointers

Pointers separate data from a reference to the data. A pointer's data can change, but the reference remains unchanged. Pointers are typically used in conjunction with structures: they allow for changes to type and organization in structure fields.

Note that IDL pointers are unlike pointers in C/C++ and Fortran: they do not point to a location in memory and memory cannot be traversed through the pointer.

Define a pointer to an important number:

```
IDL> p = ptr_new(42)
```

HELP and *PRINT* tell us information about this variable `P`, though *PRINT* is not so helpful:

```
IDL> help, p
P               POINTER   = <PtrHeapVar29017>
IDL> p
<PtrHeapVar29017>
```

To access the data in the pointer, we need to *dereference* `P`:

```
IDL> *p
```

The asterisk (`*`) is IDL's pointer dereference operator. Dereferenced pointers can be used in expressions like any ordinary variable:

```
IDL> x = *p / 6
IDL> x
   7
```

Assign new data to the pointer:

```
IDL> *p = 'hello'
IDL> *p
hello
```

The reference remains, but now points to the new data.

More information on pointers can be found in the IDL Help system (e.g., search for "pointers"). Pointers are also covered in more detail in the *Application Development with IDL I* course.

Objects

Objects wrap data and programs that act upon the data into a single variable. The topic of object-oriented programming is outside the scope of this course (it's covered in the *Application Development with IDL I & II* courses), but here we can talk about how to use objects, as well as some of the terminology for working with them.

Make an equilateral triangle with an `IDLgrPolygon` object:

```
IDL> x = [1, 0, -1]
IDL> y = [0, sqrt(3), 0]
IDL> triangle = idlgrpolygon(x, y)
```

The variable `TRIANGLE` is an object. It's an instance of `IDLgrPolygon`, which is a class in the IDL Object Graphics system. Polygons have *properties* such as color. What's the default color of a polygon? We can find out by calling its `GetProperty` *method* (methods are simply programs built in to objects):

```
IDL> triangle.getproperty, color=c
IDL> c
   0   0   0
```

The default color is black. Make the triangle green using the `SetProperty` method:

```
IDL> triangle.setproperty, color=!color.green
```

View the triangle with the *XOBJVIEW* helper program:

```
IDL> xobjview, triangle
```

With *XOBJVIEW*, you can pan, rotate and zoom in/out on the triangle. When you're finished, close *XOBJVIEW* and destroy the object by calling its `Cleanup` method:

```
IDL> triangle.cleanup
```

More information on using objects can be found in the IDL Help system (e.g., search for "objects").

Exercises

1. Why does IDL give a syntax error with the following statement?

   ```
   IDL>  a = "12 Monkeys is an awesome movie."
   a = "12 Monkeys is an awesome movie."
             ^
   % Syntax error.
   ```

   Hint: use the output from this statement

   ```
   IDL>  print, "12
   ```

to diagnose the problem.

2. Suppose the array TEST_ARR is defined as:

```
IDL> test_arr = [complex(0.0, 0.0), 5.0D, 0ULL, '123']
```

What is its type? Type it into IDL and use the *HELP* procedure to test your conclusion.

3. (Difficult) Without using a FOR loop, write a function that, given a two-dimensional array, returns a copy of the array with every other odd element changed to -1. For example, if

```
4       0       7       5       9
3       6       0       7       6
3       6       8       5       6
2       2       7       7       9
0       2       8       2       5
```

were passed to the function, it would return

```
4       0      -1       5      -1
3       6       0      -1       6
3       6       8      -1       6
2       2       7      -1       9
0       2       8       2      -1
```

To get a random 5 × 5 integer matrix with entries 0 through 9, use

```
IDL> array = fix(randomu(seed, 5, 5) * 10)
```

See the course program *SET_ALTERNATE_ODD* for one possible solution.

References

Bowman, K. P. *An Introduction to Programming with IDL*. Boston, Massachusetts: Academic Press, 2006.

Prof. Bowman's new book is a distillation of his many years of teaching IDL to undergraduate students at Texas A&M University. It is filled with useful examples and exercises. He devotes several chapters to variables.

Fanning, David F. *IDL Programming Techniques*. Second Edition. Fort Collins, Colorado: Fanning Software Consulting, 2000.

A useful introductory book written by an original member of the Training Department at Research Systems. Dr. Fanning excels at explaining the idiosyncrasies of IDL.

Galloy, Michael. *Modern IDL: A Guide to IDL Programming*. Boulder, Colorado, 2011. <http://modernidl.idldev.com/>

This is a great book for anyone using IDL, but it shines in its coverage of advanced topics and application development with IDL. It's also a useful reference guide, collecting tables and lists of items that are scattered throughout the IDL Help system.

Gumley, Liam E. *Practical IDL Programming*. San Francisco: Morgan Kaufmann, 2001.

An excellent book on general IDL programming recommended by David Stern, creator of IDL and founder of RSI. Chapter 2 gives information on IDL syntax.

Goldberg, David. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. ACM Computing Surveys, Vol 23, No 1, March 1991.

The Perils of Floating Point. Bruce M. Bush, Lahey Computer Systems. 1996. <http://www.lahey.com/float.htm>. Accessed 2009-06-16.

What Every Programmer Should Know About Floating-Point Arithmetic. The Floating-Point Guide, 2010. <http://floating-point-gui.de>. Accessed 2010-05-04.

Help! The Sky is Falling! David Fanning, Fanning Software Consulting, 2002. <http://www.idlcoyote.com/math_tips/sky_is_falling.html>. Accessed 2009-06-16.

Floating point. <http://en.wikipedia.org/wiki/Floating_point>. Accessed 2010-04-07.

These articles describe how floating-point arithmetic works on computers. (These are a few favorites; search for 'floating point arithmetic' for others.)

Chapter 6:
Programming

This chapter describes the programming components of IDL and how to use them.

> *If you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.*
>
> —Linus Torvalds (from `http://en.wikiquote.org`)

Programs 64
Parameters 69
Calling mechanism 70
Operators 72
Control statements 75
Batch files 82
Programming tips 82
Exercises 83
References 83

Programs

Statements are instructions for a computer. A statement is the syntactic equivalent of a complete sentence—it expresses a complete thought. Statements are the programmer's unit of communication with IDL. IDL statements are case insensitive, with the only exception being characters inside a string.

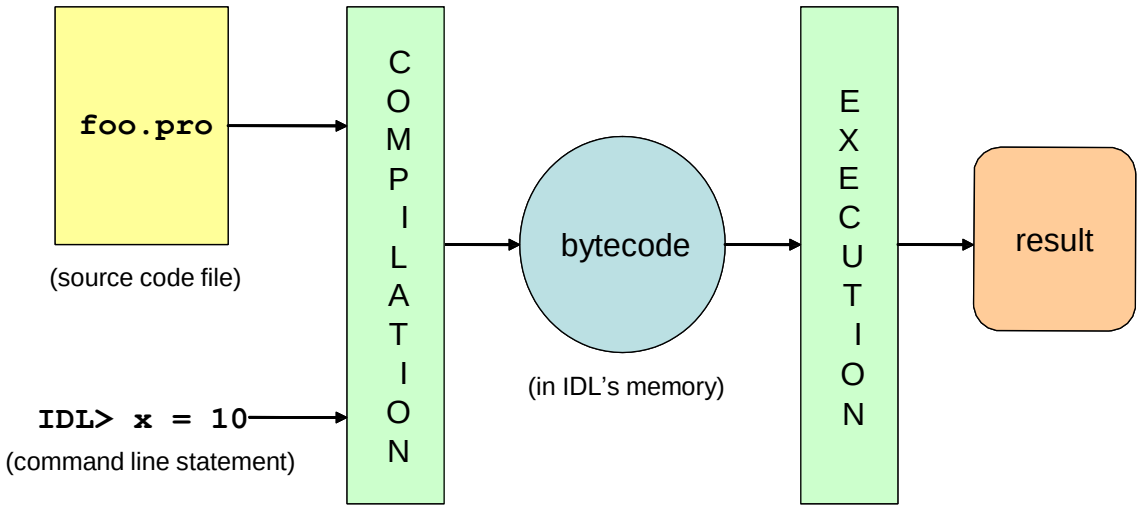
A program is a sequence of statements that are combined to act as a unit. IDL supports three program types:

- main
- procedure
- function

They are described below.

The IDL code execution model lists the steps needed to execute (run) a statement or a program. The compilation step takes IDL source code, existing either as a program in a file or a statement typed at the command line, and transforms it into machine-readable code, called bytecode, stored in IDL's memory. This bytecode is what is executed in order to produce a result.

Figure 6-1: The IDL code execution model.



When using the IDL workbench, programs can be compiled and executed with menu items or keyboard shortcuts, as described in the following sections. Programs can also be compiled and executed using the more fundamental executive commands, described next.

Executive commands

Executive commands are the statements used to compile, run, stop, and step through IDL programs. Executive commands are easily recognizable—they begin with a period. They may only be called from the command line or in IDL's noninteractive batch mode. The table below lists the executive commands and their purpose.

Table 6-1: Executive commands, grouped by function.

Command	Description
`.compile` *filename*	Compiles program modules. If called without a filename, it can be used to compose and compile a program from the command line.
`.run` *filename*	Compiles IDL procedures, functions and main programs, also executes main programs; if called without a filename, it can be used to compose, compile, and run a main-level program from the command line.
`.rnew` *filename*	Similar to `.run`, except that all user variables are erased before the new main program is executed.
`.go`	Executes a previously compiled main program.
`.reset_session`	Resets IDL, removing variables and compiled programs and reinitializing system variables.
`.full_reset_session`	Extends `.reset_session` by removing all system routines and libraries external to the IDL distribution.
`.edit` *filename*	Opens a file in an IDL workbench editor window.
`.continue`	Continues execution of a program that has been stopped because of an error, a *STOP* statement, or a keyboard interrupt.
`.out`	Continues program execution until the current program module returns to its caller.
`.return`	Continues execution until a *RETURN* statement is encountered.
`.skip` *n*	Skips over the specified number of statements in the current program; n = 0, 1, 2,...
`.step` *n*	Executes a specified number of statements in the current program, then stops; n = 0, 1, 2,...
`.stepover`	Steps over calls to other program units.
`.trace`	Similar to `.continue`, but it displays each line of code before it executes it.

An executive command may be abbreviated to the fewest characters that uniquely identify it. For example, `.comp` can be safely substituted for `.compile`.

The workbench toolbar and **Run** menu have buttons that match these executive commands. In the workbench, you have the option of using these buttons or typing executive commands directly at the Command Line.

Main

A main program is a sequence of IDL statements terminated with an END statement. The file **hello.pro** in the **introduction/src** directory gives an example:

```
print, 'Hello World'
end
```

Open this file in the workbench. Compile and execute this main program with the corresponding entries in the **Run** menu, or by using the `F8` keyboard shortcut:

```
% Compiled module: $MAIN$.
Hello World
```

Alternately, compile and execute the program with the `.run` executive command:

```
IDL> .run hello
% Compiled module: $MAIN$.
Hello World
```

Main programs have two features that differentiate them from procedures and functions:

1. Only one main program can exist in an IDL session: introducing a second main program kicks out the first. This is somewhat detrimental if you need to work with more than one program.
2. Any variables defined in a main program persist after the program ends. This can be useful for examining the results of the program. It can be detrimental, though, if leftover variables clog memory. Also, the possibility of overwriting variables increases.

Main programs are typically used as a quick way to collect a group of statements and execute them as a unit. They're also used as command programs to route data through a processing workflow.

Procedure

A procedure is a self-contained IDL program: any variable defined in a procedure is deleted when the program ends, unless it is passed out through a parameter. The file **date.pro** in the **introduction/src** directory gives an example:

```
pro date

   d = systime()
   print, 'The current date and time is ' + d
end
```

A procedure starts with a declaration consisting of the reserved word PRO, the name of the procedure, and any parameters for the procedure (this one has none). The body of the procedure follows. A procedure is terminated with an END statement.

Open this file in the workbench. Compile and execute this procedure with the entries in the **Run** menu, or use the F8 keyboard shortcut:

```
% Compiled module: DATE.
The current date and time is Wed May 19 14:31:24 2010
```

Alternately, compile the procedure by issuing the .compile executive command:

```
IDL> .compile date
```

and call it by name:

```
IDL> date
The current date and time is Wed May 19 14:33:47 2010
```

If the file containing this procedure had not been in IDL's search path (see "Search path" on page 26), it could still be compiled, either by opening it in the workbench, or by specifying its file path in the .compile statement. For example, if **date.pro** had been in **C:\temp**, which is typically not in IDL's path, the .compile statement would be:

```
IDL> .compile "C:\temp\date.pro"
```

IDL would locate the file and compile it. The procedure could then be executed by calling it by name, as before.

Function

A function is another self-contained IDL program, like a procedure, but a function returns information to its caller. The file **time.pro** in the **introduction/src** directory contains an example:

```
function time

   d = systime()
   return, 'The current date and time is ' + d
end
```

A function begins with a declaration consisting of the reserved word FUNCTION, the name of the function, and any parameters for the function (this one has none). The body of the function, which usually includes at least one *RETURN* statement, follows. A function is terminated with an END statement.

Open this file in the workbench. Compile the function with the **Compile time.pro** entry in the **Run** menu:

```
% Compiled module: TIME.
```

A function, however, cannot be executed directly through the **Run** menu. It must be called by name:

```
IDL> print, time()
The current date and time is Wed May 19 14:41:05 2010
```

Alternately, use the `.compile` executive command and call the function by name:

```
IDL> .compile time
IDL> print, time()
The current date and time is Wed May 19 14:42:27 2010
```

As with the procedure, if the file containing this function had not been in IDL's search path, it could still be compiled by opening it in the workbench or specifying its absolute file path in the `.compile` statement.

Advantages of procedures and functions

Unlike main programs, procedures and functions (together, subprograms) are

1. self-contained, with controlled inputs and outputs, and
2. reusable.

They allow a programmer to separate and group logically related code. This is desirable because the same code can then be used in different places, or even in different programs, multiple times without copying it.

Subprograms are like building blocks from which larger, more complex programs can be constructed. Because procedures and functions are self-contained, they can be written and tested with knowledge only of what goes into and out of them through their parameters (covered in "Parameters" below). Larger programs written using subprograms are typically easier to read and debug, since each procedure and function can be studied independently. This well-accepted programming practice is called structured programming.

The *Scientific Programming with IDL* course dives deeper into how procedures and functions (and main programs) are used.

The COMPILE_OPT statement

The COMPILE_OPT statement is used to alter IDL's compile-time behavior. The statement is locally scoped; i.e., it changes behavior only within the routine in which it is declared. Exelis VIS recommends the IDL2 option (a shorthand for combining the DEFINT32 and STRICTARR options, discussed below) be used in all new procedures and functions.

The DEFINT32 option makes the default IDL integer long (32-bit signed) instead of short (16-bit signed). This is a compatibility measure: it brings IDL into line with languages such as Fortran, C, Python and Java, where a four-byte signed integer is the default.

The STRICTARR option enforces the use of square brackets for subscripting. This is an efficiency measure: though it is possible to subscript arrays with parentheses, it creates an ambiguity with respect to function calls. IDL must resolve the ambiguity, which hurts performance.

More on the COMPILE_OPT statement can be found in its entry in the IDL Help system.

Parameters

Parameters are used to pass information in and out of procedures and functions. Almost every built-in routine in IDL uses them. IDL employs two kinds of parameters: positional and keyword. Though either type can be used to send or receive information, positional parameters are typically used for required information, whereas keywords are used for optional information.

Take *PLOT* as an example. It accepts positional parameters, as shown in these statements:

```
IDL> t = findgen(100)/5.0   ; independent variable
IDL> w = beselj(t)          ; dependent variable
IDL> p = plot(t, w)
IDL> q = plot(w, t)
```

Note that the order of the parameters is important (hence, they're "positional"). Both plots `P` and `Q` are correct, but they're inverse graphs. For built-in IDL routines, the order in which parameters are expected is listed in the routine's Help entry. When called with two positional parameters, *PLOT* expects the first to be the abscissas of the graph and the second to be the ordinates.

PLOT also accepts keywords. It's a style convention to list them after any positional parameters. [XYZ]TITLE are examples of input keywords for the *PLOT* function:

```
IDL> p1 = plot(t, w, xtitle='Time', ytitle='Displacement')
IDL> p2 = plot(t, w, ytitle='Displacement', xtitle='Time')
```

Keywords can be listed in any order; these statements produce the same result. Note the primary syntax of keywords is "*name* = *data*". With keywords, the equal sign "=" is used as a delimiter; it does not mean assignment!

TEST is an example of a behavior keyword for *PLOT*. Its state is on (1) or off (0, the default). The slash "/" turns the keyword on. These statements are equivalent:

```
IDL> p = plot(/test)
IDL> q = plot(test=1)
```

This "slash" notation is commonly used in IDL. Although it saves all of one keystroke, it's a handy visual clue that a keyword is being set.

CURRENT is an example of an output keyword for the *CD* procedure. For it to work, a variable must be placed as its data:

```
IDL> help, cdir
CDIR            UNDEFINED = <Undefined>
IDL> cd, current=cdir
IDL> help, cdir
CDIR            STRING    = '/home/mpiper/IDLWorkspace81/Default'
```

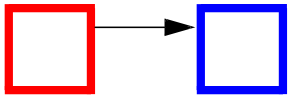
Here, the variable `CDIR` doesn't exist initially, but *CD* puts information into it through the CURRENT keyword. (Again, don't think of "=" as assignment!)

Keyword parameters can be abbreviated. This feature is useful when using IDL interactively, but it's not recommended for programming since it may confuse a reader.

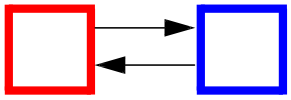
Parameter passing

Most programming languages support both "pass by value" and "pass by reference" methods of evaluating parameters passed into a program. IDL uses both.

Pass by value means that each parameter gets a copy of its argument's value, so that changes to the parameter don't affect the argument, even if the variables have the same name. Changes to the parameter are discarded when the program returns. By default, C, Java and Python use pass by value.



With pass by reference, an argument is not copied to its parameter. Any modifications to the parameter in the program change the argument because the two reference the same memory. By default, Fortran uses pass by reference.



In IDL, the parameter passing rules are:

1. named variables are passed by reference;
2. everything else—including function calls, subscripted arrays, lists or hashes, structure fields, system variables and constants—is passed by value.

Why is this distinction important?

Many IDL routines use parameters to pass information back to the caller. Take, for example, the call to *CD* in the previous section:

```
IDL> cd, current=cdir
```

Recall that `CDIR` was initially undefined, but because it's a variable, it is passed by reference through the CURRENT keyword into *CD*, which uses it to return the current directory path as a string. This parameter needs to be passed by reference or the information cannot be returned.

Note that IDL doesn't formally have a way to distinguish whether parameters are to be passed in or out of a program. Documentation, therefore, is key. In the Help system, output parameters for a routine will list the need for a named variable. For example, the CURRENT keyword for *CD* uses this text:

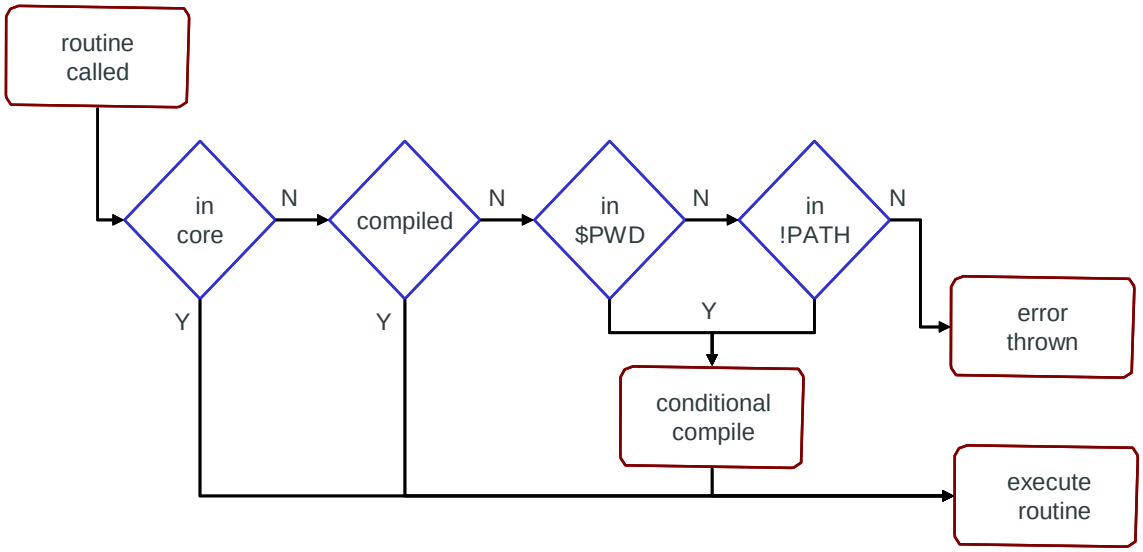
> *If CURRENT is present, it specifies a named variable into which the current working directory is stored as a scalar string.*

Calling mechanism

The calling mechanism is a series of steps that IDL performs to locate, compile and execute a procedure or function, when that routine is called by name from either the command line or another IDL program. The calling mechanism is invoked every time a routine is called, regardless of whether the routine is built in to IDL, or written by you, or written by another user.

Figure 6-2 diagrams the process of the calling mechanism.

Figure 6-2: The IDL calling mechanism.



The calling mechanism can be described in four steps:

1. Look for the routine in IDL's table of system routines (the core of IDL). If the routine is listed, execute it; otherwise, fall to the next step.
2. Check whether the routine has been compiled in the current IDL session. This can be diagnosed by calling *HELP* with its ROUTINES keyword set. If the program is already compiled, then run it. If not, then fall to the next step.
3. Search for a file on the file system that has the same base name as the routine with a **.pro** or **.sav** extension. The search starts in the current IDL directory, then proceeds through the directories listed in the `!PATH` system variable. If the file is found, IDL compiles routines it encounters in the file, starting at the top, until the called routine is found. This routine is then compiled and executed. If a file with the same name as the called routine is not found, or if a file with the same name is found, but it doesn't contain the called routine, then fall to the last step.
4. Issue an error message, stating that the requested procedure or function cannot be found.

As an example, call the *BIN_DATE* function from the command line:

```
IDL> now = bin_date()
% Compiled module: BIN_DATE.
```

Step 3 was reached in the calling mechanism. *BIN_DATE* is not a system routine, nor was it compiled earlier. However, the file **bin_date.pro** is in IDL's **lib** subdirectory, which is a part of the default search path. IDL opened the file, compiled the function *BIN_DATE* within and executed it.

Step 3 is frequently reached when working with user-defined IDL routines. Because of this, it's important to set up a search path (see page 26) that includes the directories containing IDL routines you wish to use.

Operators

Operators are programming elements that are used to combine values, expressions and variables into more complex values, expressions and variables. Table 6-2 lists the operators IDL supports, their order of precedence and examples of their use.

Table 6-2: Operators and operator precedence.

Priority	Operator	Description	Example
Highest	( )	parentheses for grouping	`IDL> print, 3*2+1, 3*(2+1)` `7 9`

	[ ]	brackets for concatenating arrays	`IDL> a = indgen(3)`
			`IDL> print, [a, -17]`
			`0 1 2 -17`

	( )	parentheses in a function call	`IDL> print, indgen(3)*2` `0 2 4`

	[ ]	brackets for subscripting arrays	`IDL> a = findgen(5)`
			`IDL> print, a[3]*2`
			`6`

	{ }	braces for structure creation	`IDL> s = {val:5}`
	.	structure field	`IDL> print, s.val*2` `10`

	*	pointer dereference	`IDL> p = ptr_new(1)`
			`IDL> print, *p`
			`1`

	-> or .	method invocation	`IDL> p = plot(/test)`
			`IDL> p.save, 'test1.png'`
			`IDL> p->save, 'test2.png'`

	++	increment	`IDL> a = 5`
			`IDL> print, a++, a, ++a`
			`5 7 7`

	--	decrement	`IDL> print, a--, a, --a` `7 5 5`

	^	exponentiation	`IDL> a = 5.0`
			`IDL> print, a^2, a^(-2)`
			`25.0000 0.0400000`

	# and ##	matrix multiplication	(see "Matrix multiplication" below)
	*	multiplication	`IDL> print, 5*2` `10`
	/	division	`IDL> print, 5/2, 5/2.` `2 2.50000`
	mod	modulo	`IDL> print, 5 mod 2` `1`

Table 6-2: Operators and operator precedence.

Priority	Operator	Description	Example
	+	addition / string concatenation	`IDL> print, 5 + 2` `7`
	-	subtraction / unary negation	`IDL> print, 5 - 2, -5` `3 -5`
	<	minimum	`IDL> print, 5 < 2` `2`
	>	maximum	`IDL> print, 5 > 2` `5`
	~	logical negation	`IDL> print, ~5` `0`
	not	bitwise complement	`IDL> print, not 5` `-6`
	eq	equal to	`IDL> print, 5 eq 2, 2.0 eq 2` `0 1`
	ne	not equal to	`IDL> print, 5 ne 2, 2.0 ne 2` `1 0`
	le	less than or equal to	`IDL> print, 5 le 2, 2.0 le 2` `0 1`
	lt	less than	`IDL> print, 5 lt 2, 2.0 lt 2` `0 0`
	ge	greater than or equal to	`IDL> print, 5 ge 2, 2.0 ge 2` `1 1`
	gt	greater than	`IDL> print, 5 gt 2, 2.0 gt 2` `1 0`
	and	bitwise and	`IDL> print, 5 and 6` `4`
	or	bitwise or	`IDL> print, 5 or 6` `7`
	xor	bitwise exclusive or	`IDL> print, 5 xor 6` `3`
	&&	logical and	`IDL> print, 5 && 6` `1`
	||	logical or	`IDL> print, 5 || 6` `1`
	? :	ternary (conditional expression)	`IDL> y = (x gt 0) ? x : -x`
Lowest	=	assignment	`IDL> a = 3`

For an interesting and informative visualization of these operators, see Michael Galloy's Periodic Table of IDL Operators: http://michaelgalloy.com/2006/11/01/periodic-table-of-idl-operators.html.

When operators at the same level of precedence are used without grouping in an expression, they are evaluated from left to right. For example:

```
IDL> print, 5 mod 3 * 2
       4
```

Compound operators

C-like compound operators are supported in IDL since version 6.0. The available operators are listed here:

Table 6-3: Compound operators.

*=	/=	+=	-=	#=
##=	<=	>=	and=	or=
xor=	^=	eq=	ne=	lt=
le=	gt=	ge=	mod=	

Compound operators combine assignment with another operator. A statement such as

```
a op= expression
```

is equivalent to

```
a = temporary(a) op expression
```

where *op* is an operator that can be combined with assignment to form one of the operators listed above, and *expression* is an IDL expression. For example:

```
IDL> a = 3
IDL> a += 5 * !pi
IDL> print, a
       18.7080
```

The final value of `A` is its initial value plus 5π.

Array operations

IDL operators can be used with arrays as well as with scalars. For example:

```
IDL> print, indgen(5) mod 2
       0       1       0       1       0
```

Here, the modulo operator MOD is applied to each element of the return from the *INDGEN* function. Other examples using IDL 8.3's new implied print and colon notation rather than *INDGEN*:

```
IDL> [0:4] > 2
       2       2       2       3       4
IDL> [0:4] gt 2
       0       0       0       1       1
```

In each case, the operator is applied on an element-by-element basis to the array. This powerful feature eliminates the need for loops. Looping still occurs, but at the C core of IDL, which is faster than looping at the interpreted level of IDL. Take advantage of this behavior as much as possible—it is both faster and easier to read.

Matrix multiplication

The # operator computes an array product by multiplying the columns of the first array by the rows of the second. The ## operator computes an array product by multiplying the rows of the first array by the columns of the second. The latter operation is equivalent to mathematical matrix multiplication.

For example:

```
IDL> a = indgen(3,2)     ; 3 x 2 array, in IDL syntax
IDL> a
       0       1       2
       3       4       5
IDL> b = indgen(2,3)     ; 2 x 3 array
IDL> b
       0       1
       2       3
       4       5
IDL> a # b
       3       4       5
       9      14      19
      15      24      33
IDL> a ## b
      10      13
      28      40
```

Note that for arrays `X` and `Y` of compatible dimensions, `X ## Y = Y # X`.

Control statements

Control statements are used to control how information flows through a program. IDL has a complete set of control statements with similar, if not identical, syntax to those in other programming languages. A list of control statements, with examples of their use, is provided in the following sections. Additional information can be found in the IDL Help.

Compound statements

In IDL, a control statement is a single execution step. Multiple steps can be grouped into a compound control statement with a BEGIN-END block. For example, this IF statement

```
if (x lt 0) then print, x
```

does not need a BEGIN-END block, whereas this one

```
if (x lt 0) then begin
   print, 'y takes the negative of x'
   y = -x
endif
```

does.

Conditional statements

The IF, CASE and SWITCH statements are used to branch control in a program.

IF

An IF statement evaluates a conditional expression. If the expression is true, the THEN clause of the statement is executed; otherwise, the ELSE clause is executed. If no ELSE clause is present, control falls out of the IF statement.

IF_EX, an example program in the IDL coursefiles **introduction/src** directory, uses IF to print the absolute value of a number generated from a normal distribution:

```
pro if_ex
   compile_opt idl2

   ; A number plucked from a Gaussian distribution.
   x = randomn(systime(/seconds), 1)
   print, 'x =', x, format='(a,7x,f8.5)'

   ; The one-line IF call.
   if (x gt 0.0) then print, 'x is greater than zero'

   ; The one-line IF-THEN-ELSE call.
   if (x gt 0.0) then y = x else y = -x

   ; The IF statement with a block.
   if (x eq 0.0) then begin
      z = 'This is highly unlikely!'
      print, z
   endif

   ; The IF-THEN-ELSE block.
   if (x gt 0.0) then begin
      y = x
   endif else begin
      y = -x
   endelse
   print, 'y = |x| =', y, format='(a,1x,f8.5)'
end
```

Run this program and examine its output:

```
IDL> if_ex
x =       -2.80085
y = |x| =  2.80085
IDL> if_ex
x =        0.58287
x is greater than zero
y = |x| =  0.58287
```

CASE

The CASE statement evaluates an expression to select, from a set of cases, one statement (which may be compound) for execution.

The program *CASE_EX* uses a CASE statement to determine the band interleaving of an RGB image (image interleaving is covered Chapter 7, "Images"):

```
pro case_ex, image
   compile_opt idl2

   ; Get the image dimensions.
   dims = size(image, /dimensions)

   ; Use CASE to test which dimension is 3. Note how the selector
   ; expressions have been moved into the body of the CASE.
   s = 'The image is interleaved '
   case 1 of
      dims[0] eq 3: print, s + 'by pixel (BIP)'
      dims[1] eq 3: print, s + 'by line (BIL)'
      dims[2] eq 3: print, s + 'band sequentially (BSQ)'
   else:
   endcase
end
```

For an example image, use *READ_IMAGE* (reading standard format image files, such as JPEG, is covered in Chapter 8, "File Access") to read the file **marsglobe.jpg**, located in the IDL **examples/data** directory. `MARS` is an RGB image. Use *CASE_EX* to state how its bitplanes are interleaved.

```
IDL> file = file_which('marsglobe.jpg')
IDL> mars = read_image(file)
IDL> help, mars
MARS            BYTE      = Array[3, 400, 400]
IDL> case_ex, mars
The image is interleaved by pixel (BIP)
```

The ELSE clause in a CASE statement is optional. It requires a colon, unlike the ELSE in an IF statement. When the case selector expression doesn't find a match among the cases, and there is no ELSE, an error is thrown.

SWITCH

Like CASE, a SWITCH is used to select from a set of statements for execution, depending on the value of a conditional expression. However, SWITCH differs from CASE in that if a match is found, all subsequent items in the set are executed until a BREAK (see "Jump statements" below) or an ENDSWITCH statement is found. While there is only one path through a CASE statement, there may be multiple paths through a SWITCH statement.

The program *SWITCH_EX* converts a band number (1, 2, 3, 4, 5, or 7) from a Landsat ETM+ image into a zero-based band index (in the range [0,5]) that IDL can use to subset the image by band:

```
pro switch_ex, band=band_number
   compile_opt idl2

   ; Check that the requested band number is in {1,2,3,4,5,7}. If
   ; not, issue a message and return. Note the accumulation of cases.
   switch band_number of
   1:
   2:
   3:
   4:
   5: begin
      band_index = band_number - 1
      break
      end
   7: begin
      band_index = band_number - 2
      break
      end
   else: begin
      print, 'Band number not available.'
      return
      end
   endswitch

   print, 'Landsat band number ' + strtrim(band_number,2) $
      + ' corresponds to band index ' + strtrim(band_index,2) + '.'
end
```

What's the band index for Landsat band number 7?

```
IDL> switch_ex, band=7
Landsat band number 7 corresponds to band index 5
```

What about (the nonexistent) band number -65?

```
IDL> switch_ex, band=-65 ; doesn't exist
Band number not available.
```

A SWITCH statement becomes logically equivalent to a CASE statement if a BREAK is included within every case. Unlike a CASE statement, the ELSE clause in a SWITCH is optional: if no matches are found for the selector expression, the SWITCH statement does nothing.

The definition of true and false

What is true and what is not, for the different IDL data types, is listed in Table 6-4. Note that IDL uses a two's complement for determining "true" for integer types. This can be disabled locally in a routine by setting the LOGICAL_PREDICATE option to COMPILE_OPT.

Table 6-4: The definition of truth, in IDL.

Variable or expression type	Truth statement
Integer (byte, integer, long)	Odd values are true, even values are false
Floating point (float, double, complex)	Nonzero values are true, zero values are false
String	Nonzero lengths are true, null strings are false
Heap variable (pointers, objects)	Valid heap variable data are true, null or undefined heap variable data are false

Loop statements

The FOR, WHILE, REPEAT and FOREACH statements can execute a statement, or a group of statements, multiple times.

FOR

The FOR loop executes a statement or group of statements a fixed number of times. It's possible to specify the step size and direction of the loop. The program *FOR_EX* gives two examples:

```
pro for_ex
   compile_opt idl2

   ; The one-line FOR loop.
   for j = 0, 9 do print, j

   ; A FOR loop with a block. The counter decrements with a step
   ; of 2 on each iteration until the loop limit is reached.
   sequence = findgen(21)
   for k = 20, 0, -2 do begin
      print, k, sequence[k], (sequence[k] mod 3)
   endfor
end
```

It's recommended to use, at minimum, a long integer (and avoid floating-point types) for the loop counter variable. Starting with IDL 8.0, the loop counter variable is automatically set to be a long integer. See Exercise 1 for an example of a problem with using a float for a loop counter.

WHILE

The WHILE loop executes a statement or group of statements as long as its test condition is true. The condition is checked at the entry point of the loop. The example program *WHILE_EX* uses a WHILE loop to generate a Fibonacci sequence:

```
function while_ex
   compile_opt idl2

   ; Generate a Fibonacci sequence.
   f = [0, 1] ; seeds
   while max(f) lt 100 do begin
      n = f[-1] + f[-2]
      f = [f, n]
   endwhile
   return, f
end
```

Execute this program with:

```
IDL> print, while_ex()
       0       1       1       2       3       5       8
      13      21      34      55      89     144
```

With a WHILE loop (and with a REPEAT loop, below) it's important to update the test condition on each iteration the loop; otherwise, an infinite loop could result. (If this would ever happen, stop the IDL interpreter by selecting `Ctrl-Break` on Windows and `Ctrl-C` on all UNIX-based systems.)

REPEAT

The REPEAT-UNTIL loop is similar to WHILE except that the condition is tested at the end of the loop. Like *WHILE_EX*, the example program *REPEAT_EX* generates a Fibonacci sequence:

```
function repeat_ex
   compile_opt idl2

   ; Generates a Fibonacci sequence.
   f = [0, 1] ; seeds
   repeat begin
      n = f[-1] + f[-2]
      f = [f, n]
   endrep until max(f) gt 100
   return, f
end
```

Execute this program with:

```
IDL> print, repeat_ex()
       0       1       1       2       3       5       8
      13      21      34      55      89     144
```

FOREACH

FOREACH loops through the elements of an array, structure, list or hash. However, unlike other loop statements in IDL, FOREACH doesn't use a loop counter, it just iterates through all the items in an input set.

The program FOREACH_EX uses a FOREACH statement to print out the names of all the animals contained in an array:

```
pro foreach_ex
   compile_opt idl2

   ; An array of zoo animals.
   zoo = ['lion', 'tiger', 'bear', 'red panda']

   print, 'The zoo has:'

   ; Use FOREACH to enumerate the animals in the zoo.
   foreach animal, zoo do $
      print, ' - ' + animal + 's'

   print, 'Oh my!'
end
```

Call this program with and evaluate the result:

```
IDL> foreach_ex
The zoo has:
 - lions
 - tigers
 - bears
 - red pandas
Oh my!
```

Jump statements

Jump statements like BREAK and CONTINUE move control to another location in a program, possibly skipping lines or even moving backward through a program.

BREAK

The BREAK statement provides a convenient way to immediately exit from a FOR, WHILE, REPEAT, FOREACH, CASE or SWITCH statement.

CONTINUE

The CONTINUE statement provides a convenient way to immediately start the next iteration of the enclosing FOR, WHILE, REPEAT or FOREACH loop. Whereas the BREAK statement exits from a loop, the CONTINUE statement exits only from the current loop iteration, proceeding immediately to the next iteration.

Batch files

A batch file is a sequence of individual IDL statements. A batch file is not a program—it cannot be compiled. Rather, each statement in the file is interpreted and executed sequentially by IDL.

These IDL commands:

```
print, systime()
pwd
```

are stored in the file **batch_ex.pro** in the **introduction/src** directory. To compile and execute the comands sequentially in batch mode, type:

```
IDL> @batch_ex
Fri Jun 24 09:59:39 2011
/home/mpiper/IDLWorkspace81/Default
```

Had **batch_ex.pro** not been in IDL's path, a quoted absolute or relative filepath could be specified after the @ symbol.

On UNIX-based systems, IDL's batch mode can be a powerful tool. For example, this command can be executed at a shell prompt:

```
$ idl < batch_ex.pro > batch.out &
```

With this shell command, IDL starts and runs in the background. It accepts the sequence of commands from the file **batch_ex.pro** and redirects the output to the file **batch.out**. When finished, IDL exits.

Timing with TIC and TOC

The TIC and TOC routines are used to check the run time of your IDL programs. TIC gets the initial time, and TOC gets the finish system time. TOC also takes the difference in the two times, and prints the result to the screen.

```
IDL> tic
IDL> toc
% Time elapsed: 2.3770001 seconds.
```

A specific clock variable can be assigned when TIC is called. To use the same clock, the varble has to then be specified when calling TOC.

```
IDL> clock = tic()
IDL> toc, clock
% Time elapsed: 7.9670000 seconds.
```

TOC can also be called multiple times on a single TIC.

```
IDL> tic
IDL> toc
% Time elapsed: 1.9879999 seconds.
IDL> toc
  % Time elapsed: 3.3700001 seconds.
```

Programming tips

Here are a few tips to help you successfully write programs in IDL.

1. Main programs are good, but procedures and functions are better.
2. Use COMPILE_OPT IDL2 in every new procedure and function. This implies the use of square brackets for subscripting arrays.
3. Use a consistent coding style. This will make it easier for others (and, sometimes, yourself) to read your programs.
4. Use descriptive variable and program names.
5. Place the primary routine last in a file. Name the file the same as the routine with a **.pro** extension. Use lowercase characters and underscores for file names. This technique works well with the calling mechanism and across operating systems.

IDL programming techniques are covered in greater detail in the *Scientific Programming with IDL* and *Application Development with IDL* courses offered by Exelis VIS. Further, Mike Galloy and David Fanning have collected their programming tips at

- http://michaelgalloy.com/2006/07/14/12-tips-for-beginning-idl-programmers.html and
- http://www.idlcoyote.com/documents/tips.html

Exercises

1. How many times will this FOR loop execute? Why?

```
IDL> for i=0.0, 1.0, 0.1 do print, i
```

2. Write a function to calculate the geometric mean

$$\langle x_i \rangle = \left(\prod_{i=1}^{N} x_i \right)^{1/N}$$

of an array of numbers. For a sample solution, see the program *GMEAN* in the IDL coursefiles **introduction/src** directory.

References

Procedural programming. Wikipedia, 2010. <http://en.wikipedia.org/wiki/Procedural_programming>. Accessed 2010-07-14.

Structured programming. Wikipedia, 2010. <http://en.wikipedia.org/wiki/Structured_programming>. Accessed 2010-07-14.

Information on programming architecture.

Two's complement. Wikipedia, 2011. <http://en.wikipedia.org/wiki/Two's_complement>. Accessed 2011-08-01.

Explains why IDL thinks `not 5` *is* `-6`.

File Naming Conventions. David Fanning, Fanning Software Consulting, 2003. <http://www.idlcoyote.com/tips/namefiles.html>. Accessed 2011-08-01.

An article on how to properly name programs and program files in IDL.

Galloy, Michael. *Modern IDL: A Guide to IDL Programming*. Boulder, Colorado, 2011. <http://modernidl.idldev.com/>

This is a great book for anyone using IDL, but it shines in its coverage of advanced topics and application development with IDL. It's also a useful reference guide, collecting tables and lists of items that are scattered throughout the IDL Help system.

Gumley, Liam E. *Practical IDL Programming*. San Francisco: Morgan Kaufmann, 2001.

An excellent book on general IDL programming recommended by David Stern, creator of IDL and founder of RSI.

Kernighan, Brian W. and Rob Pike. *The Practice of Programming*. Boston: Addison-Wesley, 1999.

McConnell, Steve. *Code Complete: A Practical Handbook of Software Construction*. Redmond, Washington: Microsoft Press, 1993.

These books discuss common techniques in designing, developing, testing and debugging program

Chapter 7: Images

This chapter describes how to use built-in IDL routines to manipulate and display image data.

> *For me the future of the image is going to be in electronic form... You will see perfectly beautiful images on an electronic screen.*
>
> —Ansel Adams (from `en.wikiquote.org`)

What is an image? 86
HST imagery of the Carina Nebula 86
Truecolor JPEG image 90
Landsat 7 ETM+ image 92
Exercises 94
References 95

What is an image?

A digital image is stored as an array in IDL. Each array element, or pixel, corresponds to a value of light intensity or color. Images can appear in a variety of formats, as listed here.

Table 7-1: Image formats.

Image format	Description
Binary image or mask	Each pixel has the value 1 or 0 (or on/off or white/black).
Intensity-mapped image	Each pixel has a value ranging over a set of intensities, usually 8-, 12- or 16- bit. The image is displayed in shades of gray corresponding to the intensities. Low values typically correspond to low intensities, high values to high intensities. However, this may be reversed (e.g., an X-ray image). This format is used by JPEG.
Color-mapped image	Each pixel has a value within an 8-bit range that corresponds not to an intensity, but rather to a color in a separate color lookup table. For example, a pixel value of 0 could correspond to black, 1 to purple, 2 to orange, etc. This format is used by GIF, PNG and TIFF.
Truecolor image	Each pixel corresponds to a 3- or 4-element array, giving the red, green, blue (and possibly alpha) components of a color. Such images are also called RGB or RGBA images. This format is used by JPEG, TIFF and PNG.
Multi- or hyperspectral image	Each pixel corresponds to a N-element array, where N is the number of spectral bands in which the sensor capturing the image samples. To visualize such imagery, three bands are typically selected to form an RGB color composite image.

Images are displayed in IDL with the *IMAGE* function. We'll explore how to work with and display image formats from the Table above at a high level in this chapter. For image processing, see Chapter 10, "Analysis." Also note that several file access routines are used in this chapter to read image data from files into IDL. File access is covered in greater detail in Chapter 8, "File Access."

HST imagery of the Carina Nebula

The Hubble Space Telescope (HST) marked its 20th anniversary in April 2010. As a part of the celebration, NASA and the Space Telescope Science Institute (STScI) released a new image of the Carina Nebula taken with Hubble's Wide Field Camera 3. The image shows the towering pillars of gas and dust of the nebula being torn apart by the new stars forming within it. Information (both general and technical) about this new image, including links to download it, can be found at:

`http://hubblesite.org/newscenter/archive/releases/2010/13`.

Here, courtesy of Max Mutchler, a research and instrument scientist at STScI, we have some of the unprocessed data that he used in making this image.

We'll use IDL to read the data from a file, visualize the data as a grayscale, intensity-mapped image, and do some light processing on the image to reduce noise and improve its appearance.

Use *DIALOG_PICKFILE* to locate the HST data file **ibdz11b0q_single_sci.fits** in the IDL coursefiles **data** directory. It's a FITS file, a binary file format used widely in astronomy.

```
IDL> hst_file = dialog_pickfile(filter='*.fits')
```

Use the astrolib *FITS_HELP* routine (the IDL Astronomy User's Library, or astrolib, is discussed in more detail on page 104) to get information on what's stored inside the file:

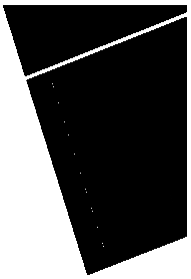
```
IDL> fits_help, hst_file

/home/mpiper/IDL/development/data/ibdz11b0q_single_sci.fits

      XTENSION  EXTNAME  EXTVER EXTLEVEL BITPIX GCOUNT  PCOUNT NAXIS  NAXIS*

   0                                      -32      0      0     2  2000 x 3000
```

FITS_HELP tells us the file contains a single 2000 x 3000 pixel image. Read the image data, and optionally the header, from the file with the astrolib *MRDFITS* function. The result is a two-dimensional IDL float array.

```
IDL> extension = 0 ; from FITS_HELP output
IDL> hst_data = mrdfits(hst_file, extension, hst_file_header)
MRDFITS: Image array (2000,3000)  Type=Real*4
IDL> help, hst_file_header, hst_data
HST_FILE_HEADER STRING    = Array[374]
HST_DATA        FLOAT     = Array[2000, 3000]
```

As described in Table 7-1, data stored as a two-dimensional array can be visualized as a grayscale intensity-mapped image, with low values mapped to darker shades of gray and high values mapped to lighter shades of gray. Visualize these HST data with *IMAGE*:

```
IDL> img = image(hst_data)
```



There isn't much to see: a black polygon with two white lines running through it, one thick, one dotted. Let's try to diagnose what it is we're seeing. Print the values of the first 10 pixels of the image:

```
IDL> print, hst_data[0:9]
   9999.90      9999.90      9999.90      9999.90
   9999.90      9999.90      9999.90      9999.90
   9999.90      9999.90
```

This value, 9999.9, is the image's fill value, used for pixels that don't represent data. These are the highest data values, so they're mapped to the white areas of the image. Use the *WHERE* function to find all the pixels that don't contain this fill value:

```
IDL> fill_value = 9999.9
IDL> i_nofill = where(hst_data lt fill_value, n_nofill)
IDL> float(n_nofill) / n_elements(hst_data)
   0.596079
```

The variable `I_NOFILL` holds the one-dimensional subscripts into the array `HST_DATA` that are valid, non-fill pixels. They constitute about 60 percent of the image. We'll use this subset of `HST_DATA` in further analyses.

The fill values explain the white stripes in the image, but what about the black pixels?

The *HISTOGRAM* function can be used to compute the histogram, or pixel distribution, of an image. This image's histogram is dominated by pixels near zero, with almost all of the pixel values falling below 0.2. Compute the histogram of the non-fill pixels in this image using 100 evenly spaced bins between 0.0 and 0.2. Display the result with *PLOT*.

```
IDL> nbins = 100
IDL> hmin = 0.0
IDL> hmax = 0.2
IDL> h_hst_data = histogram(hst_data[i_nofill], max=hmax, min=hmin, $
>      nbins=nbins)
IDL> hbins = findgen(nbins)/(nbins-1)*(hmax-hmin) + hmin
IDL> p = plot(hbins, h_hst_data, /histogram, $
>      /fill_background, fill_color='light steel blue', $
>      xtitle='Pixel value', ytitle='Occurrences', $
>      title='HST Carina Nebula Image Histogram')
```

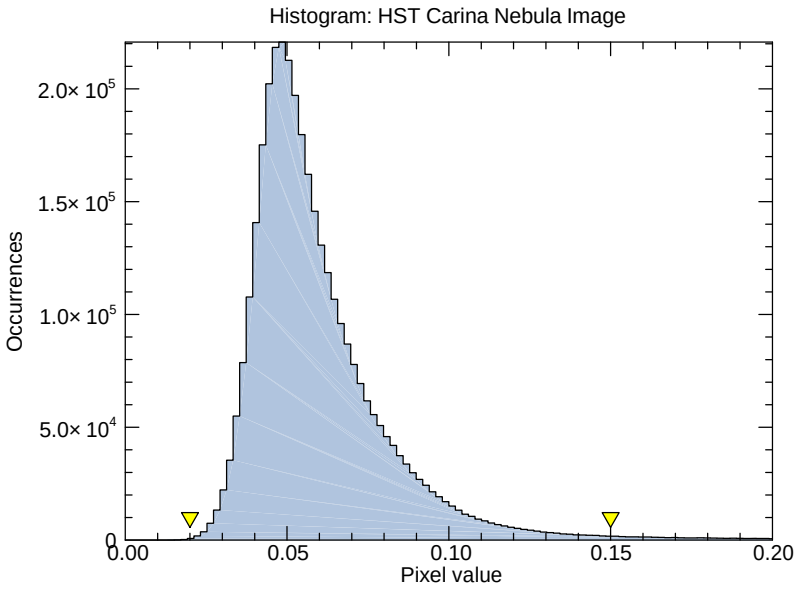


Figure 7-1:

The histogram of the raw HST Carina Nebula image.

The abscissa shows the histogram bins corresponding to intervals of pixel values found in the image, while the ordinate gives the number of times pixels within the intervals occur in the image. From inspection of this histogram—and with the help of the *WHERE* function—we see that 97 percent of the valid pixels have values between 0.02 and 0.15:

```
IDL> scale_min = 0.02
IDL> scale_max = 0.15
IDL> !null = where((hst_data ge scale_min) and (hst_data le scale_max), n)
IDL> float(n) / n_elements(i_nofill)
   0.970753
```

Use this range of pixel values as a rough estimate for byte scaling the image. Byte scaling is the process of interpolating the pixel values of an image into the byte range [0,255] for display purposes. For pixel values smaller than this range, this acts to increase the contrast in the image. Use the *BYTSCL* function to byte scale the image using the suggested lower and upper limits:

```
IDL> hst_data_scaled = bytscl(hst_data, min=scale_min, max=scale_max)
```

Any pixel value less than `SCALE_MIN` is mapped to 0 in `HST_DATA_SCALED`. Any pixel value greater than `SCALE_MAX`, including the fill value, is mapped to 255. Pixel values between these bounds are linearly interpolated to the interval [0,255]. Use *IMAGE* to visualize this bytescaled result as an intensity-mapped image:

```
IDL> carina1 = image(hst_data_scaled, title='HST WFC3 Carina Nebula')
IDL> txt = 'Byte scaled'
IDL> !null = text(0.1, 0.1, txt, /normal)
```

Compare your result with the left panel of Figure 7-2.

The Carina Nebula is now visible, but the image is noisy: it contains speckles from cosmic rays and detector artifacts such as chip gap, bad columns and CCD edges. Attempt to attenuate some of this noise by applying a median filter to the image. Median filtering replaces each pixel in an image with the median (the value in an ordered set with an equal number of values above and below it) of the two-dimensional neighborhood of a given width. It's similar to smoothing with a boxcar or top hat filter but it doesn't blur edges larger than the neighborhood. Median filtering is effective in removing "salt and pepper" noise (isolated high or low values) from an image.

Use the *MEDIAN* function to filter the image with a 7 x 7 kernel:

```
IDL> hst_data_filtered = median(hst_data_scaled, 7)
```

View the result with *IMAGE*:

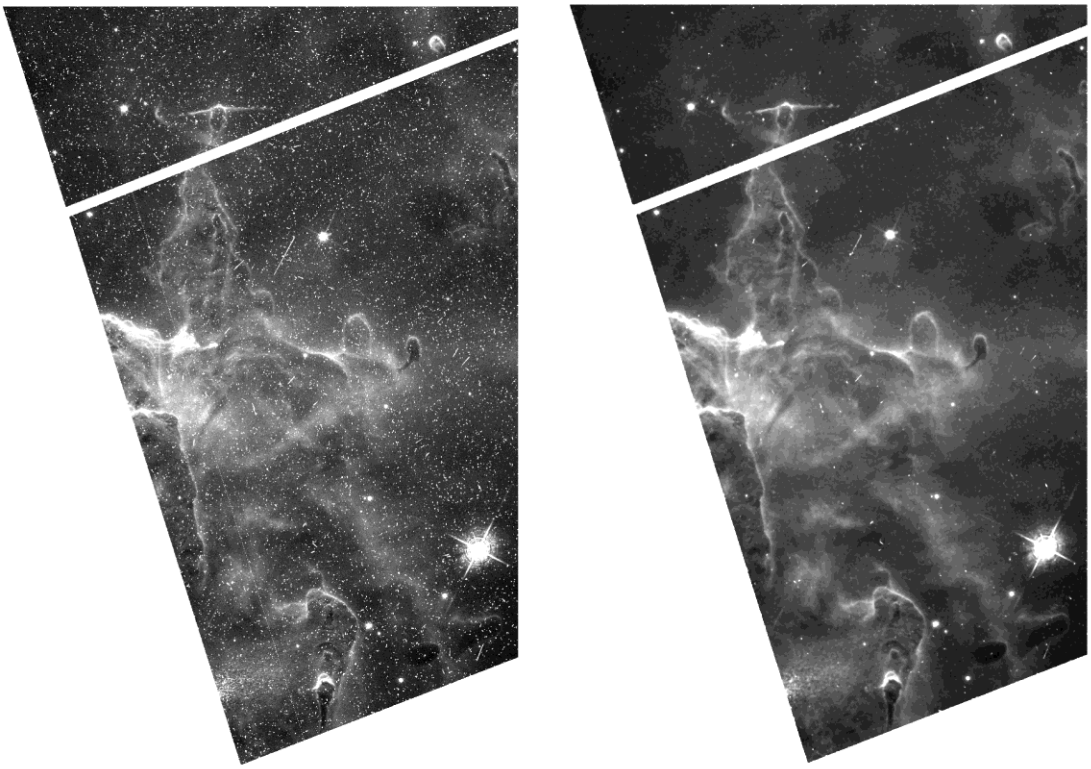
```
IDL> carina2 = image(hst_data_filtered, title='HST WFC3 Carina Nebula')
IDL> txt = [txt, 'Median filtered']
IDL> !null = text(0.1, 0.1, txt, /normal)
```

Compare your result with the image in the right panel of Figure 7-2.

This is a crude analysis of these image data. Max has a much more complicated algorithm for stitching and denoising these images to get a more robust result (see links to his work in the References).

A more detailed examination of these HST data can be found in the main program contained in **hst_carina_image.pro** in the IDL coursefiles.

Figure 7-2: The HST Carina Nebula image, byte scaled (left), then median filtered (right).



Truecolor JPEG image

The second image we'll examine is of a rose. It's stored in a JPEG file in the IDL **examples/data** directory. Read it with these statements:

```
IDL> rose_file = file_which('rose.jpg')
IDL> read_jpeg, rose_file, rose
IDL> help, rose
ROSE            BYTE      = Array[3, 227, 149]
```

The variable `ROSE` represents a Truecolor RGB (red + green + blue) image. Display it with *IMAGE*:

```
IDL> img = image(rose)
```

By default, the image is drawn in a Cartesian sense, from bottom to top, with the origin of the image in the lower left corner.

Note that the array `ROSE` has three dimensions, unlike the HST Carina image, which has only two. What is this extra dimension?

One way to think of an RGB image is as three intensity-mapped images, one for each color (these are called channels, bitplanes or bands), layered on top of each other, with their intensity values combined to form a single Truecolor image. `ROSE` is three 227 x 149 images combined to give single image represented as a three-dimensional array.

The manner in which the bitplanes of an RGB image are organized is called interleaving. There are three ways an image can be interleaved, as shown in the Table below.

Table 7-2: Truecolor image interleaving.

Interleaving	Image array dimensions
Band interleaved by pixel (BIP)	`3 x m x n`
Band interleaved by line (BIL)	`m x 3 x n`
Band sequential (BSQ)	`m x n x 3`

In this Table, *m* refers to the number of columns (or samples) in an image, *n* to the number of rows (or lines).

Interleaving describes how the image pixel data are ordered in the three-dimensional image array; the different interleaving techniques allow for preferential access to the samples, lines or bands of a Truecolor image. The `ROSE` image has dimensions 3 x 227 x 149. Therefore, we see that `ROSE` is BIP, which is typical for JPEG images.

Let's explore the interleaving of the rose image in an example. Start by decomposing `ROSE` into its individual bitplanes:

```
IDL> r = reform(rose[0,*,*])
IDL> g = reform(rose[1,*,*])
IDL> b = reform(rose[2,*,*])
IDL> help, r, g, b
R               BYTE      = Array[227, 149]
G               BYTE      = Array[227, 149]
B               BYTE      = Array[227, 149]
```

The interleaved dimension of `ROSE` is subscripted to extract the bitplanes. The *REFORM* function is used to remove the spurious first dimension from the subscripted results.

Store the original image and its bitplanes in a list, for convenience:

```
IDL> images = list(rose, r, g, b)
IDL> help, images
IMAGES          LIST  <ID=89741  NELEMENTS=4>
```

Make an array of descriptive titles for the four images in the list:

```
IDL> titles = ['Full image', 'Red Band', 'Green Band', 'Blue Band']
```

Make a window to hold a graphic:

```
IDL> w = window(window_title='A Rose, By Bands')
```

Looping over the elements of the `IMAGES` list, display the rose and its red, green and blue bitplanes in a 2 x 2 grid, using the LAYOUT property of *IMAGE*:

```
IDL> for i=0, n_elements(images)-1 do $
>      !null = image(images[i], layout=[2,2,i+1], /current, title=titles[i])
```

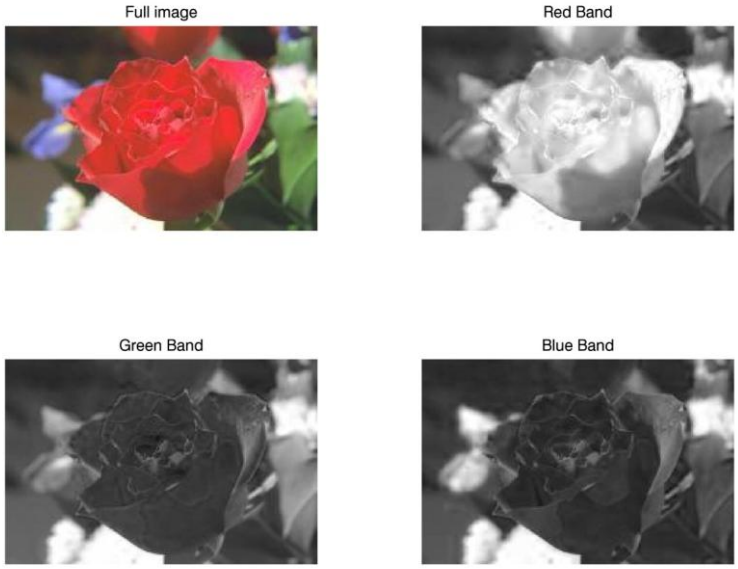


Figure 7-3:

An RGB image of a rose, displayed with its component red, green and blue bitplanes.

It's interesting to view the rose through the information in its bitplanes. For example, the blossom has high intensities in the red channel, but lower intensities in the green and blue channels. Likewise, the leaves have higher intensities in the green channel than in red or blue. The white flowers in the background have high intensities in all three channels.

Save this visualization to a TIFF file using pack bits compression with:

```
IDL> w.save, 'display_rose_bands.tif', compression=1
```

The complete code for this example can be found in the IDL coursefiles program *DISPLAY_ROSE_BANDS*.

Landsat 7 ETM+ image

The last image we'll explore is a Landsat 7 Enhanced Thematic Mapper Plus (ETM+) scene, obtained from the USGS EROS Data Center as a GeoTIFF file. The ETM+ sensor has six bands in visible and infrared wavelengths, as well as separate panchromatic and thermal bands. The data were read into ENVI (the Exelis VIS geospatial image exploration and analysis software built on IDL), spatially subset to the area around Boulder, Colorado, then exported to an ENVI-format file. Note that an ENVI file is actually two files—a data file (flat binary with BSQ interleaving) and a header file (plain text) that lists image metadata.

Examination of the header file **boulder-ETM.hdr**, located in the IDL coursefiles **data** directory, reveals that the image has 575 samples, 700 lines and 6 bands. Store this information in variables for later use:

```
IDL> n_samples = 575
IDL> n_lines = 700
IDL> n_bands = 6
```

Get the path to the data file with *FILE_WHICH*:

```
IDL> file = file_which('boulder-ETM.dat')
```

Read the data from the file into IDL with *READ_BINARY*. The DATA_DIMS keyword specifies the dimensions of the image, as well as the band interleaving (compare with Table 7-2).

```
IDL> cube = read_binaray(file, data_dims=[n_samples,n_lines,n_bands])
IDL> help, cube
CUBE            BYTE      = Array[575, 700, 6]
```

Because it has three dimensions, data from a multiband sensor such as ETM+ is commonly called an image cube.

Extract bands 1, 2, and 3 from the image cube and form them into a band (3,2,1) Truecolor composite image for display:

```
IDL> band321 = bytarr(n_samples, n_lines, 3, /nozero)
IDL> band321[*,*,0] = cube[*,*,2]
IDL> band321[*,*,1] = cube[*,*,1]
IDL> band321[*,*,2] = cube[*,*,0]
```

Display the result with *IMAGE*:

```
IDL> im1 = image(band321, /order)
```

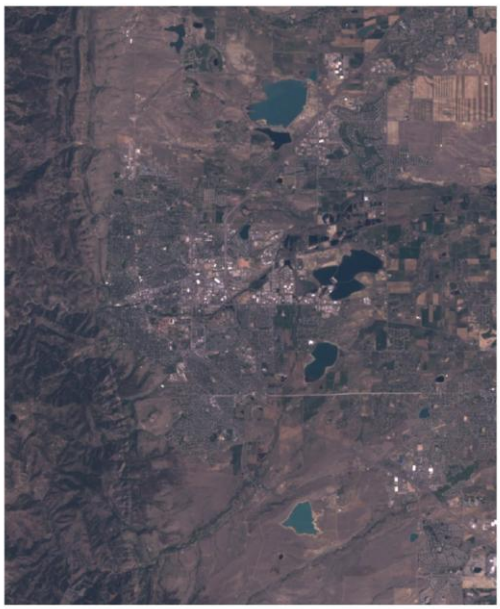


Figure 7-4:

A band (3,2,1) Truecolor composite image of Boulder, Colorado from Landsat 7 ETM+ data.

By default, IDL displays images in a Cartesian sense, placing the origin at the lower left corner and drawing from the bottom up. However, in remote sensing and medical imaging, images are drawn instead from the top down with the origin in the upper left corner. IDL's default draw order can be changed to use the remote sensing convention by setting the ORDER property for *IMAGE*. If the ORDER property is not set, the image would display, but it would be upside-down.

The colors in the image in Figure 7-4 appear washed out. Use *HIST_EQUAL* to apply a two percent linear stretch to the data (ENVI does this by default), to enhance the contrast in the image:

```
IDL> band321s = hist_equal(band321, percent=2)
```

In this operation, the pixel values between the 2nd and 98th percentiles of the image histogram are linearly interpolated to the byte range. Image pixels outside this percentile range are assigned the values 0 and 255, respectively. The relative distance between the interpolated pixels is maintained in a linear stretch. In other stretch types, such as Gaussian or square root, this isn't the case.

Display the result:

```
IDL> im2 = image(band321s, /order)
```

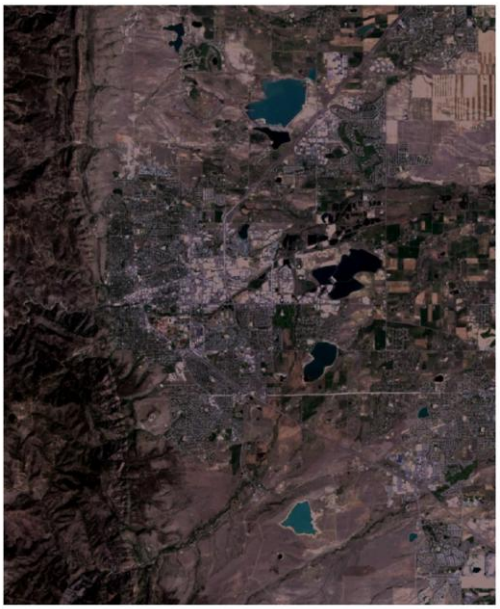


Figure 7-5:

A band (3,2,1) Truecolor composite image of Boulder, Colorado from Landsat 7 ETM+ data. A two percent linear stretch has been applied to the image.

Compare the images in Figure 7-4 and Figure 7-5. Note the fuller color range in the second image.

The code in this section is taken from *DISPLAY_BOULDER_LANDSAT_IMAGE* in the IDL coursefiles.

Exercises

1. Use the *TRANSPOSE* function to change the interleaving of the `ROSE` image from BIP to BSQ.
2. Can you make a mirror image (top or side reflection) using IDL routines?
3. Can you make an RGB image from a color-mapped image? Extra credit: can you also set the interleaving of the RGB image?

For sample solutions to these Exercises, see the programs *MAKE_RGB* and *MIRROR_IMAGE_EX* in the IDL coursefiles **introduction/src** directory.

References

HST 20th Anniversary Image is a WFC3 Mosaic of Carina Nebula HH 901. STScI, 2010. <http://archive.stsci.edu/prepds/carina>. Accessed 2010-07-20.

Hubble Space Telescope WFC3 and ACS mosaic images of HH 901 in the Carina Nebula. STScI, 2010. <http://archive.stsci.edu/pub/hlsp/carina/hlsp_carina_wfc3_HH901_readme.txt>. Accessed 2010-07-20.

Reference information on how Max Mutchler and the researchers at the Space Telescope Science Institute (STScI) processed the data from the Hubble Space Telescope to construct the 20th anniversary Carina Nebula image. The HST image data used in this course manual, as well as some notes on processing them, are courtesy Max Mutchler, STScI.

Fortner, Brand and Theodore E. Meyer. *Number by Colors: A Guide to Using Color to Understand Technical Data.* New York: Springer-Verlag, 1997.

This book explains the theory of color as well as the use of color in scientific data visualization.

How NOT to Lie with Visualization. Bernice E. Rogowitz and Lloyd A. Treinish, IBM Thomas J. Watson Research Center. <http://www.research.ibm.com/dx/proceedings/pravda/truevis.htm>. Accessed 2010-07-14.

An in-depth discussion of color table usage for scientific data visualization, including many examples.

Landsat 7. NASA, 2010. <http://landsat.gsfc.nasa.gov/about/landsat7.html>. Accessed 2010-07-14.

Landsat Enhanced Thematic Mapper Plus (ETM+). USGS EROS Data Center, 2010. <http://eros.usgs.gov/Find_Data/Products_and_Data_Available/ETM>. Accessed 2010-07-14.

Information on Landsat 7 and the Enhanced Thematic Mapper Plus sensor from the USGS and NASA.

Data available from the U.S. Geological Survey.

ENVI, the Environment for Visualizing Images. Exelis Visual Information Solutions, 2011. <http://www.exelisvis.com/ENVI>. Accessed 2011-10-14.

Information about the ENVI geospatial image exploration and analysis software developed by Exelis VIS.

Chapter 8:
File Access

This chapter describes how to read and write data files with IDL.

> *Computer files can be divided into two broad categories: binary and text. The distinction is subtle because to computers, any file is a sequence of digital bits.*
>
> —Wikipedia.org entry for "binary and text files"

File types: text and binary 98
File manipulation routines 98
IDL SAVE files 101
Standard file types 102
Reading text and binary files 105
Low-level file routines 106
Exercises 114
References 114

File types: text and binary

ASCII is a system for representing alphabetic, numeric and punctuation characters as numbers. A text file employs ASCII to translate the sequence of digital bits in a file into human-readable text. For example, the character `H` is represented by the number 72 in ASCII. Every computer understands ASCII, so it is a very portable file format. An ASCII file is human-readable; it can be viewed with text editors such as vi, (X)Emacs, TextWrangler or Wordpad.

Binary files employ no translation, they are simply streams of ones and zeros. Binary files are typically used to store numeric data or machine-readable instructions. Binary files are not human-readable. A text editor will attempt to translate the information in a binary file to ASCII characters, but the result is gibberish. Binary files are very compact; however, they are not as portable as text because of the byte ordering of different processors (see "Opening and closing files" below).

IDL can read and write text or binary files, including many types of standardized and hierarchical binary and text/binary combinations. In this chapter, we describe many tools and methods for reading and writing data files with IDL.

File manipulation routines

IDL has many built-in routines to perform actions on files at the file manager level (using Windows Explorer, Mac Finder, or commands at a shell prompt). A list of these routines, with an explanation of their purpose, is given here:

Table 8-1: File manipulation routines in IDL, loosely grouped by function.

Routine	Purpose
CD	Changes current working directory on the file system.
FILEPATH	Constructs a string containing the absolute file path to a file located relative to the main IDL directory.
FILE_BASENAME	This function returns the 'basename' of a file path. The basename is the final rightmost segment of the file path.
FILE_DIRNAME	This function returns the 'dirname' of a file path. The dirname is all of the file path except for the final rightmost segment.
FILE_EXPAND_PATH	Expands a given file or partial directory name to its fully qualified name regardless of the current working directory.
FILE_CHMOD	This procedure allows you to change the current access permissions (sometimes known as modes on UNIX platforms) associated with a file or directory.
FILE_COPY	Copies files, or directories of files, to a new location.
FILE_DELETE	Deletes files or empty directories.
FILE_INFO	This function returns information about a file, including its name, size, permissions, access and modification times.
FILE_LINES	Reports the number of lines of text in a file.

Table 8-1: File manipulation routines in IDL, loosely grouped by function.

Routine	Purpose
FILE_LINK	Creates UNIX file links, both regular (hard) and symbolic.[a]
FILE_MKDIR	Creates a new directory on the file system. *FILE_MOVE*
	Moves files, or directories of files, to a new location.
FILE_READLINK	Returns the path pointed to by a UNIX symbolic link.[a]
FILE_SAME	Determines whether two different file names refer to the same underlying file.
FILE_SEARCH	This function returns a string array containing the names of all files matching the input path specification, including wildcards.
FILE_TEST	Given a path to a file, returns 1 if the file exists, 0 if it doesn't.
FILE_WHICH	Modeled after the UNIX `which` command, this function separates a specified file path into its component directories and searches in turn each directory for a specific file.
FSTAT	Reports information about an open file.
DIALOG_PICKFILE	This routine allows the user to interactively pick a file, multiple files, or a directory using the platform's own native graphical file-selection dialog.
EOF	This function tests a specified file unit for the end-of-file condition.
FLUSH	This procedure forces all buffered output on the specified file units to be written.
PATH_SEP	Returns the proper file path delimiter for the current operating system.
PUSHD	Pushes a directory to the top of the directory stack.
POPD	Changes the current directory to the one on top of the directory stack, then deletes it.
PRINTD	Prints the directories in the directory stack.
COPY_LUN	Copies data between two open files.
POINT_LUN	Sets or obtains the current position of the file pointer in a specified file.
SKIP_LUN	This procedure reads data in an open file and moves the file pointer. It is used to skip over a known amount of data in a file without having the data available in an IDL variable.
TRUNCATE_LUN	This procedure truncates the contents of a file (which must be open for write access) at the current position of the file pointer.
SOCKET	*SOCKET* opens a client-side TCP/IP Internet socket as an IDL file unit. Such files can be used with any of IDL's file i/o routines.
FILE_POLL_INPUT	Blocks IDL command line until data are read from a file unit. Used primarily with *SOCKET*.
GETENV	Gets the value of an environment variable. *SETENV*
	Sets the value of an environment variable.
ROUTINE_FILEPATH	Returns the path to a compiled procedure or function.

a.Only available on UNIX-based platforms.

Here are some examples of using selected routines from Table 8-1.

Use *FILE_WHICH* to get the full path to the file containing the *HANNING* function:

```
IDL> file = file_which('hanning.pro')
IDL> print, file
C:\Program Files\ITT\IDL\IDL81\lib\hanning.pro
```

The file must reside in IDL's path for *FILE_WHICH* to work.

Extract the base name and directory name of this file with *FILE_BASENAME* and *FILE_DIRNAME*:

```
IDL> print, file_basename(file)
hanning.pro
IDL> print, file_dirname(file)
C:\Program Files\ITT\IDL\IDL81\lib
```

Extract the base name minus its file extension:

```
IDL> dotpos = strpos(file, '.', /reverse_search)
IDL> ext = strmid(file, dotpos)
IDL> print, ext
.pro
IDL> print, file_basename(file, ext)
hanning
```

Use *FILE_LINES* to count the number of lines in the file **hanning.pro**:

```
IDL> print, file_lines(file)
          91
```

Expand the current working directory with *FILE_EXPAND_PATH*:

```
IDL> print, file_expand_path('.')
```

Use *DIALOG_PICKFILE* to prompt a user to select all of the PNG files in the IDL coursefiles **data** directory:

```
IDL> start_dir = get_coursefiles_dir(/data)
IDL> png_files = dialog_pickfile(path=start_dir, $
>      filter='*.png', /multiple_files, $
>      title='Please select all PNG files')
IDL> print, png_files
C:\IDL_coursefiles\data\wind_rose.png
C:\IDL_coursefiles\data\idl_wavy.png
C:\IDL_coursefiles\data\wms.cgi.png
```

Print the names of the files starting with the letter *a* in IDL's **lib** subdirectory.

```
IDL> str = !dir + path_sep() + 'lib' + path_sep() + 'a*'
IDL> print, file_basename(file_search(str, /fold_case))
a_correlate.pro adapt_hist_equal.pro amoeba.pro annotate.pro
array_indices.pro arrow.pro ascii_template.pro
```

See the IDL Help system for more information on, and examples of, the routines listed in Table 8-1.

IDL SAVE files

SAVE files are the simplest file format to use in IDL. They're especially useful if the people you work and share data with also use IDL.

The *SAVE* procedure writes variables to a formatted binary file. The contents of a SAVE file can be read back into IDL at any time with the *RESTORE* procedure. SAVE files are platform-independent, so, e.g., a SAVE file created on Windows could be restored on a Mac. SAVE files are version-dependent, but a SAVE file created on an older version of IDL can be restored on any newer version.

SAVE files relieve the user of needing to know the details of how to access data in a file. For example, in the IDL coursefiles **data** directory, there's a netCDF file containing data from the NCAR Mesa Lab weather station. Read time and temperature data from the file:

```
IDL> ml_file = file_which('mlab.20020626.cdf')
IDL> ml_id = ncdf_open(ml_file)
IDL> tdry_id = ncdf_varid(ml_id, 'tdry')
IDL> time_id = ncdf_varid(ml_id, 'time_offset')
IDL> ncdf_varget, ml_id, tdry_id, tdry
IDL> ncdf_varget, ml_id, time_id, time
IDL> ncdf_close, ml_id
```

They're now in two IDL variables, `TIME` and `TDRY`:

```
IDL> help, time, tdry
TIME            FLOAT     = Array[288]
TDRY            FLOAT     = Array[288]
```

Write these variables to an IDL SAVE file:

```
IDL> save, time, tdry, filename='temperature_series.sav', /verbose
% SAVE: Portable (XDR) SAVE/RESTORE file.
% SAVE: Saved variable: TIME.
% SAVE: Saved variable: TDRY.
```

At another time or on another computer, read the contents of **temperature_series.sav** back into IDL with *RESTORE*:

```
IDL> restore, filename='temperature_series.sav', /verbose
% RESTORE: Portable (XDR) SAVE/RESTORE file.
% RESTORE: Save file written by mpiper@vonnegut
% RESTORE: IDL version 7.1 (linux, x86).
% RESTORE: Restored variable: TIME.
% RESTORE: Restored variable: TDRY.
IDL> help, time, tdry
TIME            FLOAT     = Array[288]
TDRY            FLOAT     = Array[288]
```

This is much easier than reading these data with the netCDF file API.

If the SAVE file is visible in the workbench Project Explorer, double-clicking on it will restore the variables into the IDL session. The same holds for selecting the SAVE file through the workbench **File > Open...** menu.

Standard file types

IDL has a host of routines for querying, reading and writing standard file formats, some of which are listed below. In this table, each format is listed with its typical file extension, query routine (used to retrieve file metadata), read routine and write routine. Some formats have an object-oriented instead of a procedural interface; some have both. For the scientific data formats listed at the end of the table, please refer to the IDL Help system. Scientific data formats are also covered in greater detail in the *Scientific Programming with IDL* course. For the geospatial file formats (ENVI, ArcGIS, GeoTIFF), see the ENVI Help. Geospatial file formats are covered in greater detail in the *Exploring ENVI* and *Extending ENVI with IDL* courses.

Table 8-2: Routines for accessing standard file formats (alpha by extension).

Format	Extension	Query routine	Read routine	Write routine
ArcGIS raster	**bil, bip, bsq**		*READ_BINARY, READU*	*WRITEU*
Windows Bitmap	**bmp**	*QUERY_BMP*	*READ_BMP*	*WRITE_BMP*
Comma Separated Value	**csv**	*QUERY_CSV*	*READ_CSV*	*WRITE_CSV*
ENVI image	**dat, img**		*READ_BINARY, READU*	*WRITEU*
Digital Imaging and Communications in Medicine	**dcm, dicom, img**	*QUERY_DICOM*	*READ_DICOM*	
		`IDLffDICOM` class (read and query)		
Drawing Exchange Format	**dxf**	`IDLffDXF` class (read, write and query)		
Graphics Interchange Format	**gif**	*QUERY_GIF*	*READ_GIF*	*WRITE_GIF*
Joint Photographic Experts Group	**jpg, jpeg**	*QUERY_JPEG*	*READ_JPEG*	*WRITE_JPEG*
JPEG2000	**jp2, jpx**	*QUERY_JPEG2000*	*READ_JPEG2000*	*WRITE_JPEG2000*
		`IDLffJPEG2000` class (read, write and query)		
Motion Picture Experts Group	**mpeg, mpg**			`IDLgrMPEG` class
Motion JPEG2000	**mj2**	`IDLffMJPEG2000` class (read, write and query)		
NCAR Raster Interchange Format	**nrif**			*WRITE_NRIF*
Portable Gray Map / Portable Pixmap	**pgm, ppm**	*QUERY_PPM*	*READ_PPM*	*WRITE_PPM*
Apple Picture Format	**pict**	*QUERY_PICT*	*READ_PICT*	*WRITE_PICT*
Portable Network Graphics	**png**	*QUERY_PNG*	*READ_PNG*	*WRITE_PNG*
ESRI Shapefiles	**shp**	`IDLffShape` class (read and query)		

Table 8-2: Routines for accessing standard file formats (alpha by extension).

Format	Extension	Query routine	Read routine	Write routine
Multi-resolution Seamless Image Database	**sid**	*QUERY_MRSID*	*READ_MRSID*	
		`IDLffMrSID` class (read and query)		
Symbolic Link Spreadsheet	**slk**		*READ_SYLK*	*WRITE_SYLK*
Sun Raster File	**srf**	*QUERY_SRF*	*READ_SRF*	*WRITE_SRF*
Tagged Image File Format (including GeoTIFF)	**tiff**	*QUERY_TIFF*	*READ_TIFF*	*WRITE_TIFF*
Windows Audio File	**wav**	*QUERY_WAV*	*READ_WAV*	*WRITE_WAV*
Wavefront Advanced Data Visualizer	**wave, bwave**		*READ_WAVE*	*WRITE_WAVE*
eXtensible Markup Language	**xml**	`IDLffXMLSAX` (read) and `IDLffXMLDOM` (read, write, query) classes		
X-Windows Dump	**xwd**		*READ_XWD*	
Common Data Format	**cdf**	See "CDF Routines" in the IDL Help. **nc,**		
Network Common Data Format	**cdf**	See "NCDF Routines" in the IDL Help.		
Hierarchical Data Format (v4)	**hdf**	See "HDF Routines" in the IDL Help.		
Hierarchical Data Format (v5)	**h5**, **hdf5**	See "HDF5 Routines" in the IDL Help.		
Hierarchical Data Format - Earth Observing System	**hdf**	See "HDF-EOS Routines" in the IDL Help.		

Working with the routines in the table is straightforward. For example, locate the file **image.tif** in the IDL **examples/data** directory and use *QUERY_TIFF* to get the file's metadata:

```
IDL> tiff_file = file_which('image.tif')
IDL> ok = query_tiff(tiff_file, info)
IDL> print, ok
       1
IDL> help, info, /structures
```

Read the image data into IDL with *READ_TIFF* and display them with *IMAGE*:

```
IDL> nyny = read_tiff(tiff_file)
IDL> help, nyny
NYNY            BYTE      = Array[768, 512]
IDL> im = image(nyny, title='New York, New York')
```

To write these data to a PNG file, use the *WRITE_PNG* procedure:

```
IDL> write_png, 'image.png', nyny
```

Or use *WRITE_JPEG* to write the data to a JPEG file instead:

```
IDL> write_jpeg, 'image.jpg', nyny
```

More information on the routines listed in Table 8-2 can be found in the IDL Help system.

Wrappers for standard format files

The routines *READ_IMAGE*, *WRITE_IMAGE* and *QUERY_IMAGE* are wrappers around the standard web image routines listed in Table 8-2. They can be used to work with image data when the file type isn't known until runtime. For example, *READ_IMAGE* can read an image without explicitly specifying its file type:

```
IDL> nyny = read_image(tiff_file)
```

The *IOPEN* procedure is a very high-level routine that opens, reads, and optionally visualizes data from a file. *IOPEN* can read many of the standard file formats shown in Table 8-2, including text, CSV, binary, web image formats (JPG, PNG, GIF, TIFF), ESRI Shapefiles and HDF5 files. For example, to read and visualize the TIFF file from the previous example, type:

```
IDL> iopen, tiff_file, nyny, /visualize
```

User-contributed routines

The IDL distribution doesn't have built-in routines to read and write all possible file formats, including some that are widely used. For example, the Flexible Image Transport System (FITS) format is used heavily in astronomy, but there aren't built-in routines in IDL for it (despite Dave Stern, the creator of IDL, being an astronomer).

To fill this gap, IDL users in the astronomy community have written their own IDL routines to read and write FITS files. These routines have been gathered into the The IDL Astronomy User's Library (or the "astrolib") hosted at `http://idlastro.gsfc.nasa.gov/` and available for anyone to download. A recent snapshot of the astrolib is included in the IDL coursefiles distribution in the **astrolib** directory. The astrolib also includes many general-use routines, including the handy *READCOL* procedure, which is used on several occasions in this course manual.

Another prominent IDL library is David Fanning's Coyote library. This library of general-use routines is hosted at `http://idlcoyote.com` and also available for anyone to download. A recent snapshot of this library is included in the IDL coursefiles distribution in the **coyote** directory.

In general, IDL has a strong user community that supports code sharing. When faced with the prospect of writing your own file reader for a specific file format, it might be worthwhile to browse the user libraries listed on the Related Sites & Links page `http://www.exelisvis.com/UserCommunity/RelatedSitesLinks.aspx` of the Exelis VIS website, or visit the Code Library `http://www.exelisvis.com/UserCommunity/CodeLibrary.aspx`, or do a quick Google search (`idl + search term` works well). Someone may have already written routines for your data format.

Reading text and binary files

Table 8-3 lists a pair of high-level routines that simplify the process of reading plain text and flat binary files into IDL. The trade-off is that these routines give less control over describing the format of the data in a file.

Table 8-3: High-level routines for reading generic text and binary files.

Routine	Purpose
READ_ASCII	Reads data from a text file into a structure variable. The formatting in the file can be specified with keywords or with a template returned from the *ASCII_TEMPLATE* function.
READ_BINARY	Reads data from a binary file into a structure or an array. The organization of the file can be specified with keywords or with a template returned from the *BINARY_TEMPLATE* function.

Open the file **ascii.txt** in IDL's **examples/data** subdirectory (see "IDL directory structure" on page 18 for a reminder of where this directory exists on your computer) with your favorite text editor. Note that it has four lines of header, followed by a blank line, followed by 7 columns and 15 rows of comma-delimited weather station data.

Use *READ_ASCII* to read the header and data from this file into IDL:

```
IDL> file_a = file_which('ascii.txt')
IDL> data = read_ascii(file_a, data_start=5, header=header)
IDL> help, header
HEADER          STRING    = Array[5]
IDL> help, data, /structures
** Structure <1b57808>, 1 tags, length=420, data length=420, refs=1:
   FIELD1          FLOAT     Array[7, 15]
```

The data from the file have been read into a structure variable containing one field: a 7 x 15-element float array. Extract the elevation values from this array and print them:

```
IDL> elevation = data.(0)[2,*]
IDL> print, elevation[0:5]
   399.000      692.000      1003.00      1333.00      811.000
   90.0000
```

IDL defaults to reading data into float arrays. However, the elevation data in the file are integers. It would be better to preserve the data type when reading from the file.

Try reading this file again, but this time with help from *ASCII_TEMPLATE*; it provides a graphical interface with a three-step process for describing the contents of a file. Use it to mark the first two columns of the file as floats and the remaining five as integers. *ASCII_TEMPLATE* returns a structure variable defining a reusable template:

```
IDL> file_a_template = ascii_template(file_a)
```

Use *READ_ASCII* with the template you created to read the contents of **ascii.txt** again:

```
IDL> data = read_ascii(file_a, template=file_a_template)
```

Confirm that the wind speed values have been read as integers:

```
IDL> wind_speed = data.(5)
IDL> help, wind_speed
WIND_SPEED      INT       = Array[15]
IDL> print, wind_speed[0:4]
   10       8      10       0       8
```

The file **convec.dat** in the **examples/data** subdirectory is an example of a flat binary file—it has no header, only data. Use *READ_BINARY* to read its contents into an IDL variable:

```
IDL> file_b = filepath('convec.dat', subdir=['examples','data'])
IDL> mantle = read_binary(file_b)
IDL> help, mantle
MANTLE          BYTE      = Array[61504]
```

The file was read, but what to make of the data? We need more information on what this file contains. From the file **index.txt** in the **examples/data** subdirectory, we see that it holds one 248 x 248-element byte array representing a model of Earth's mantle convection. This supplemental information is needed to understand the file's contents.

Use the DATA_DIMS and DATA_TYPE keywords to *READ_BINARY* to read the contents of **convec.dat** into a variable with the appropriate type and dimensions:

```
IDL> mantle = read_binary(file_b, data_dims=[248,248], data_type=1)
IDL> help, mantle
MANTLE          BYTE      = Array[248, 248]
```

View the data as an image with *IMAGE*:

```
IDL> i = image(mantle)
```

More information on using *READ_ASCII* and *READ_BINARY* can be found in the IDL Help system.

Low-level file routines

This section describes the use of routines in the *OPEN, CLOSE, READ,* and *WRITE* families. Use of these low-level file routines requires more knowledge of IDL, file types and operating systems, but as a user, you get greater control over how the actual file input/output is performed.

A note on process: when using IDL's low-level file routines, it is helpful to keep in mind a few simple steps:

1. Locate the file on the file system. This typically involves specifying a path to the file, either manually with a string literal, or with IDL routines like *FILE_WHICH*, *FILEPATH*, *FILE_SEARCH* or *DIALOG_PICKFILE*.
2. Define a data format for the file's contents by creating variables of appropriate type and dimensionality. Deciding on how to have IDL represent the data in a file is usually the most difficult part of using the low-level routines.

3. Open the file for reading, writing or updating. You may have to deal with byte ordering issues in binary files at this stage.
4. Perform file operations on the file (querying, reading, writing, positioning).
5. Close the file.

These steps are employed in the examples in this section.

Opening and closing files

The IDL routines for opening and closing files are listed here:

Table 8-4: Routines for opening and closing files.

Routine	Purpose
OPENR	Opens a file for reading.
OPENW	Opens a file for writing.
OPENU	Opens a file for updating (reading and/or writing).
CLOSE	Closes a file or a list of files.
FREE_LUN	Closes files and deallocates file units.

Use the *OPENR* procedure to open a file for reading:

```
IDL> file_a = filepath('ascii.txt', subdir=['examples','data'])
IDL> openr, 1, file_a
```

The value `1` is the logical unit number (see below) assigned to the file. Use the FILES keyword to *HELP* to see what files are currently open in your IDL session:

```
IDL> help, /files
Unit   Attributes   Name
1      Read         C:\Program Files\ITT\IDL\IDL81\examples\data\ascii.txt
```

Close the file with the *CLOSE* procedure:

```
IDL> close, 1
```

Logical unit numbers

A logical unit number (LUN) is a number IDL associates with an open file. All files are assigned a LUN when they are opened, and all input/output routines refer to files through this number. Multiple files can be open simultaneously in IDL, each with a different logical unit number.

Three logical unit numbers are reserved for use by the operating system:

0	*stdin*, the command line
-1	*stdout*, the output log
-2	*stderr*, the output log

There are 128 logical unit numbers available to a user:

1-99	Specified directly in an *OPEN* routine
100-128	Specified with *GET_LUN*, or the GET_LUN keyword to the *OPEN* routines

An example of directly specifying a LUN is given above. Once a LUN in the range 1-99 is assigned to a file, it cannot be reassigned until the file is closed, the IDL session is reset, or IDL is exited.

The *GET_LUN* procedure, or the GET_LUN keyword to *OPENR*, can be used to let IDL specify an available LUN in the range 100-128. This is particularly useful for preventing conflicts with other LUNs already in use.

For example, open the file **convec.dat** for reading, allowing IDL to select a LUN:

```
IDL> file_b = file_which('convec.dat')
IDL> openr, lun, file_b, /get_lun
```

Because the GET_LUN keyword has been set, a LUN is assigned by IDL to the variable `LUN`. Verify that the file is open with *HELP*:

```
IDL> help, /files
Unit   Attributes      Name
100    Read, Reserved  /usr/local/itt/idl81/examples/data/convec.dat
```

A file unit allocated with *GET_LUN* is freed with *FREE_LUN*. *FREE_LUN* closes the file and deallocates the LUN:

```
IDL> free_lun, lun
```

Compressed and XDR-format files

IDL can open GZIP compressed files with the COMPRESS keyword to *OPENR*. IDL can also open files saved in the eXternal Data Representation (XDR) format. XDR is a standard for the description and encoding of data that overcomes the byte-ordering problem in binary files, so it is useful for transferring data between different computer architectures. The IDL SAVE file format uses XDR.

Byte ordering in binary files

Byte ordering, or endianness, is an aspect of processor architecture that determines the order in which the bytes that compose integer and floating point numbers are stored in a binary file.

Consider an IDL long integer. IDL longs are composed of four bytes, which allows integer values between -2^{31} and $2^{31}-1$ to be expressed. Take, for example, the number 1000000L. In hexadecimal notation, this number is `000F4240`. With a processor that uses little endian addressing (e.g., Intel x86), the least significant byte is written first, so this number is represented in a binary file as `0x4240000F`. On the other hand, on a big endian processor (e.g., Sun SPARC and ULTRA, PowerPC), the most significant byte is written first, so the number is represented as `0x000F4240`.

Table 8-5: Byte ordering for common platforms supported by IDL.

Little endian	Big endian
Windows (Intel or AMD)	Mac (PowerPC)
Linux (Intel or AMD)	Sun Solaris (Sparc, Ultra)
Mac (Intel)	

When working with binary files, the byte ordering scheme of the processor used to write the file, as well as the byte ordering scheme of the computer on which you're using IDL, must be considered. To open binary files of differing endianness, use the keywords to *OPEN* described in Table 8-6. To change the endianness of numbers after reading them from a file, use the byte-swapping routines listed in Table 8-7.

Table 8-6: Keywords to *OPEN* routines for changing the endianness of data.

Keyword to *OPEN*	Purpose
SWAP_ENDIAN	Changes the byte order of multi-byte data read from a binary file.
SWAP_IF_LITTLE_ENDIAN	As SWAP_ENDIAN, but applied only if the host computer has a little endian processor.
SWAP_IF_BIG_ENDIAN	As SWAP_ENDIAN, but applied only if the host computer has a big endian processor.

Table 8-7: Post-read byte ordering routines.

Routine	Purpose
BYTEORDER	Reverses the ordering of bytes within integer and floating-point numbers. Works for scalars and arrays.
SWAP_ENDIAN	Like *BYTEORDER*, but also works for structures.
SWAP_ENDIAN_INPLACE	Like *SWAP_ENDIAN*, but doesn't make a copy in memory.

Reading and writing text files

Table 8-8 lists the low-level routines for reading and writing text files:

Table 8-8: Routines for reading and writing text files.

Routine	Purpose
READF	Reads data from a text files.
PRINTF	Writes data to a text file.

The *READF* procedure is used to read text files into IDL. *READF* provides two techniques for reading files: free format and explicit format. A free format read uses spaces, commas or tabs to distinguish elements in the file. An explicit format read distinguishes elements according to the instructions in a format statement.

Reading free format ASCII files

With free format input, IDL uses a few default rules to read the file, freeing the user of the chore of deciding how the data should be formatted.

1. Input is performed on scalar variables. Array and structure variables are treated as collections of scalar variables.
2. If the current input line is empty and there are variables left requiring input, read another line.
3. If the current input line is not empty but there are no variables left requiring input, the remainder of the line is ignored.
4. Input data must be separated by commas or white space (tabs, spaces, or new lines).
5. When reading into a variable of type string, all the characters remaining in the current input line are placed into the string.
6. When reading into numeric variables, effort is made to convert the input into a value of the expected type. Decimal points are optional and scientific notation is allowed. If a floating-point datum is provided for an integer variable, the value is truncated. The default type is float.
7. When reading into a variable of complex type, the real and imaginary parts are separated by a comma and surrounded by parentheses. If only a single value is provided, it is taken as the real part of the variable, and the imaginary part is set to zero.

Use **ascii.txt** from the IDL **examples/data** directory for an example of free format reading. Recall that this file contains four lines of header, followed by a blank line, followed by a 7 x 15 array of weather data. Start by getting a path to this file with *FILE_WHICH*:

```
IDL> file_a = file_which('ascii.txt')
```

Next, define variables to determine how the data should be stored once read into IDL. The first four lines of the file are header. The fifth is a blank line. Using rules 1 and 5 above, the header can be stored in a four-element string array and the blank line can be read into a scalar string and discarded.

```
IDL> header = strarr(4)
IDL> blank_line = '' ; the open-close quote makes an empty string
```

Determining how to read the weather data is a bit trickier. Reading the data into a floating-point array is the fastest way, but type information is not preserved. Rules 1, 2, 4 and 6 are used here.

```
IDL> data = fltarr(7, 15)
```

Open the file for reading with *OPENR*, allowing IDL to assign a LUN by setting the GET_LUN keyword:

```
IDL> openr, u, file_a, /get_lun
```

Read the contents of the file with *READF*:

```
IDL> readf, u, header, blank_line, data
```

Only one statement is needed to read the file because the variables HEADER, BLANK_LINE and DATA were set up to use the free format rules. Once the read is complete, close the file with *FREE_LUN*:

```
IDL> free_lun, u
```

As a last step, the array DATA can be parsed into properly typed array variables:

```
IDL> lon = reform(data[0,*])
IDL> lat = reform(data[1,*])
IDL> elev = fix(reform(data[2,*]))
IDL> temp = fix(reform(data[3,*]))
```

and so forth. Note the use of the *REFORM* function to convert the column vectors extracted from DATA into row vectors. Were the data read from the file correctly?

```
IDL> print, elev
   399       692       1003      1333      811       90     100       4
   1530      1206      4         968       99        415    1378
```

In addition to the example above, three more example programs, *ASCII_READ_EX[1-3]*, are included in the course files. They show alternate techniques for reading **ascii.txt** into IDL.

Reading explicit format ASCII files

The *READF* FORMAT keyword can be used to explicitly specify the appearance of data in a file operation. FORMAT takes as input a string containing format codes in either a Fortran-like (the default) or a C-like syntax. Tables of format codes for both syntaxes are listed in the IDL Help system; see the section "Using Explicitly Formatted Input/Output" referenced on the *READF* Help page. A list of rules for performing explicit-format file access is also given in this Help system entry.

The text file **cities.txt** in the IDL coursefiles **data** directory contains a list of cities and their locations in geographic coordinates. This file can be read using IDL's explicit formatting rules. The first five lines of the file are

```
Adelaide        -34.55   138.34
Anchorage        61.12  -149.48
Athens           38.00    23.38
Auckland        -36.53   174.45
Baghdad          33.14    44.22
```

Open this file for reading:

```
IDL> cities_file = filepath('cities.txt', root=get_coursefiles_dir(/data))
IDL> openr, lun, cities_file, /get_lun
```

Read the third line of the file into variables CITY, LAT1 and LON1 using *READF* with a FORMAT statement. Recall that by default *READF* attempts to read into arguments of type float, so we need to specify that the variable CITY is a string.

```
IDL> city = ''
IDL> readf, lun, city, lat1, lon1, format='(/,/,a15,f7.2,2x,f7.2)'
IDL> help, city, lon1, lat1
CITY            STRING    = 'Athens          '
LON1            FLOAT     =       38.0000
LAT1            FLOAT     =       23.3800
```

Note that a free-format read would have failed here because of rule 5, which specifies that everything remaining on a line is read into a string variable. An alternative to using explicit formatting would be to read the record as a string, then use IDL's string processing routines (see "Strings" on page 56) to parse the string and numeric information from the record.

Close the file and release the logical unit number:

```
IDL> free_lun, lun
```

An example of reading all the records in this file is given in the IDL coursefiles program *ASCII_READ_EX4*, located in the **introduction/src** directory.

Writing free and explicit format ASCII files

The *PRINTF* procedure is used to write text files from IDL, using free (the default) or explicit formatting rules. Explicit formatting uses the FORMAT keyword, as described above.

Use *ASCII_READ_EX4* to read the all the city data from the file **cities.txt**:

```
IDL> ascii_read_ex4, lon, lat, city
```

Use the *WHERE* function to determine which cities on the list are in the Southern hemisphere. Store those cities and their locations in new variables.

```
IDL> i_south = where(lat lt 0.0, n_south)
IDL> city_south = city[i_south]
IDL> lat_south = lat[i_south]
IDL> lon_south = lon[i_south]
```

Sort the Southern hemisphere cities by latitude. Note that the *SORT* function returns the sorted indices of an array, not the sorted array.

```
IDL> sorted_city = city_south[sort(lat_south)]
IDL> sorted_lat = lat_south[sort(lat_south)]
IDL> sorted_lon = lon_south[sort(lat_south)]
```

Open a new file, **southern_hemisphere_cities.txt**, for writing using *OPENW*:

```
IDL> openw, lun, 'southern_hemisphere_cities.txt', /get_lun
```

Because a relative filepath is used, this file will be opened in the current directory.

Use *PRINTF* to display an informative header at the top of the new file:

```
IDL> printf, lun, format='("Cities in the Southern Hemisphere")'
IDL> printf, lun
IDL> printf, lun, format='(3x,"lat",5x,"lon",4x,"name")'
IDL> printf, lun, format='(3x,"---",5x,"---",4x,"----")'
```

Next, write the sorted cities (and their geocoordinates) to the file, using explicit formatting:

```
IDL> for i = 0, n_south-1 do $
>      printf, lun, sorted_lat[i], sorted_lon[i], sorted_city[i], $
>             format='(2(f7.2,1x),a15)'
```

Last, close the file:

```
IDL> free_lun, lun
```

See also *FORPRINT* in the astrolib—it safely replaces the FOR loop used above. The IDL coursefiles program *ASCII_APPEND_EX* gives an example of adding a new city and its geocoordinates to the **cities.txt** file.

Reading and writing binary files

Table 8-9 lists the low-level routines for reading and writing binary files:

Table 8-9: Routines for reading and writing binary files.

Routine	Purpose
READU	Reads data from a binary file.
WRITEU	Writes data to an unformatted binary file.

Data in a binary file is transferred directly between the file system and IDL without translation through a character lookup table. Binary files are smaller, and they support fast and efficient data transfer, but their portability suffers from the endianness issue ("Byte ordering in binary files" on page 108). Furthermore, because the contents of a binary file can't typically be understood by the unaided eye (unless you plan to spend some quality time with a hex editor), additional information is needed to explain what is contained in such a file.

There are several unformatted binary files in the IDL **examples/data** subdirectory. The file **index.txt** lists these files and describes their contents. For example, **index.txt** states that the file **convec.dat** contains a single 248 x 248 element byte array representing a model of Earth's mantle convection. Let's read this file into IDL.

First, set up the path to the file:

```
IDL> file_b = filepath('convec.dat', subdir=['examples','data'])
```

Next, define a data format for the contents of the file:

```
IDL> mantle = bytarr(248,248)
```

We know the data type and dimensions of the file from **index.txt**.

Open the file, read its contents with *READU*, then close it:

```
IDL> openr, lun_b, file_b, /get_lun
IDL> readu, lun_b, mantle
IDL> free_lun, lun_b
```

The data read from the file can be displayed as an image:

```
IDL> i = image(mantle)
```

Take the inverse of the mantle data and write them to a file:

```
IDL> openw, lun, 'antimantle.dat', /get_lun
IDL> writeu, lun, -mantle
IDL> free_lun, lun
```

Exercises

1. The file **people.dat** in the IDL **examples/data** directory is a flat binary file containing two 8-bit 192 x 192 images. How would you read only the second image from the file?
2. Use the astrolib *READCOL* procedure to read the header and data from the file **ascii.txt** used in the section "Reading text and binary files" on page 105.

For answers to these problems, see the programs *MEDIAN_FILTERING_EX1* (where a solution to Exercise 1 is used) and *READCOL_EX* in the **introduction/src** directory of the IDL coursefiles.

References

The IDL Astronomy User's Library. Wayne Landsman, NASA Goddard Space Flight Center, 2010. <http://idlastro.gsfc.nasa.gov/>. Accessed 2010-04-29.

The site of the extremely useful IDL astrolib, curated by Wayne Landsman. A recent snapshot of the astrolib is included in the IDL coursefiles distribution.

Coyote and Catalyst Program Libraries. David Fanning, Fanning Consulting, 2010. <http://www.idlcoyote.com/documents/programs.html>. Accessed 2010-04.

The site of David Fanning's Coyote library. His NCDF_BROWSER tool is really nice for browsing and extracting data from netCDF, HDF4 and HDF-EOS files.

Endianness. Wikipedia, 2010. <http://en.wikipedia.org/wiki/Endianness>. Accessed 2010-07-14.

The Wikipedia entry describing byte ordering.

Bowman, K. P. *An Introduction to Programming with IDL*. Boston, Massachusetts: Academic Press, 2006.

Prof. Bowman's book has many in-depth examples of file access, including working with netCDF files.

Chapter 9: Surface and Contour Plots

This chapter gives examples of displaying data as a wire mesh surface, a shaded surface, or a contour plot.

> *The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!', but 'That's funny...'*
>
> —Isaac Asimov (from `en.wikiquote.org`)

Graphics routines 116
Spatial rainfall distribution 116
Digital terrain elevations 121
Exercises 127
References 127

Graphics routines

Surface and contour plots are used to visualize the spatial variation of some variable. These plots can be created in IDL with the *SURFACE* and *CONTOUR* functions.

Table 9-1: IDL Graphics routines for producing surface and contour plots.

Name	Description
CONTOUR	Makes line or filled contour plots on two- or three-dimensional Cartesian axes. The input for *CONTOUR* is a two-dimensional array; isopleths are drawn for the array values. Properties such as color, labels, downhill tick marks, filled contours, etc., can be set.
SURFACE	Makes point, wire mesh, filled, ruled or lego surface plots. The input for *SURFACE* is a two-dimensional array; the height of the surface is given by the array values. Surfaces must be single-valued.

Many variations on surface and contour plots can be produced in IDL using the configurable properties of *SURFACE* and *CONTOUR*; they can even be combined in a graphic. In addition to these routines, we'll introduce helpers like *TEXT* and *COLORBAR*.

To demonstrate how these routines work, we'll use them in visualizing two datasets: gridded spatial rainfall distribution over the CONUS in a 24 hour period and digital elevation data for the area surrounding Denver, Colorado.

Spatial rainfall distribution

The NOAA/National Weather Service (NWS) River Forecast Centers provide gridded, multi-sensor (NEXRAD Doppler radar and rain gauge) observed precipitation estimates on a daily basis for the continental United States. These data can be downloaded in either netCDF or Shapefile format from `http://water.weather.gov/precip/download.php`.

The file **nws_precip_conus_20100416.nc** in the IDL coursefiles **data** directory is a netCDF file downloaded from this NWS site. Locate this file with *FILE_WHICH*

```
IDL> file = file_which('nws_precip_conus_20100416.nc')
```

then open the file, read the netCDF file variable 'amountofprecip' into the IDL variable `PPT` and close the file with the following statements:

```
IDL> file_id = ncdf_open(file)
IDL> ppt_id = ncdf_varid(file_id, 'amountofprecip')
IDL> ncdf_varget, file_id, ppt_id, ppt
IDL> ncdf_close, file_id
```

There's more information on working with netCDF files, including how they're structured, in the IDL Help system (and working with netCDF is also covered in greater detail in the *Scientific Programming with IDL* course).

We now have a variable containing the gridded precipitation values:

```
IDL> help, ppt
PPT             INT       = Array[1051, 813]
```

Data preparation

Before we visualize the precipitation data, let's take a few preparatory steps that aim to increase the data density (a term used by Edward Tufte) of the visualization, hopefully making it more informative.

First, take a quick look at the range of values in PPT:

```
IDL> print, min(ppt), max(ppt)
      -1   27224
```

The precipitation values are given in hundredths of a millimeter, while the value -1 indicates no precipitation. Make more convenient units for precipitation by converting them to millimeters. Also ensure there are no precipitation values less than zero.

```
IDL> ppt *= 0.01
IDL> ppt >= 0.0
```

Second, subset the spatial extent of the data. It's not raining over much of the CONUS in the time period captured in the file, but it did rain heavily in Texas. Focus on this area by subscripting the original array:

```
IDL> ppt_s = ppt[300:679, 0:519]
IDL> ppt_dims = size(ppt_s, /dimensions)
IDL> help, ppt_s
PPT_S           FLOAT     = Array[380, 520]
```

The spatial subset PPT_S ("s" for subset) is a 380 x 520 array. These dimensions are determined by the *SIZE* function and stored for later use in PPT_DIMS.

Third, smooth the subset. The data contain noise which will clutter the visualization. Smoothing attenuates noise, which should improve the legibility of the graphic.

```
IDL> ppt_ss = smooth(ppt_s, 5)
```

The *SMOOTH* function smooths the data with a 5 x 5 moving average filter.

Last, the data are georeferenced, with a grid of roughly 4 x 4 km over our subset. We'll discuss map projections and displaying data in them in Chapter 11, "Map Projections." For now, we can approximate the spatial dimensions of the data based on the grid size:

```
IDL> grid_spacing = [4.0, 4.0] ; km
IDL> xgrid = findgen(ppt_dims[0])*grid_spacing[0]
IDL> ygrid = findgen(ppt_dims[1])*grid_spacing[1]
```

The variables XGRID and YGRID are vectors that give the distance, in kilometers, from the lower left corner of the subset data array, located at 108.3° W longitude, 22.7° N latitude.

Shaded surface plot

Visualize the subset and smoothed rainfall totals as a shaded surface (light source shading produces the illusion of depth) with the *SURFACE* function:

```
IDL> s_precip = surface(ppt_ss, xgrid, ygrid, font_name='Times', $
>      xtitle='East (km)', ytitle='North (km)', $
>      ztitle='Precipitation (mm)')
```

Also add a descriptive title and subtitle to the graphic with *TEXT*:

```
IDL> str = ['Observed Precipitation', $
>      '24-hr total for the period ending 2010-04-16 12:00 UTC']
IDL> title = text(0.5, 0.9, str[0], alignment='center', $
>      font_name='Times', font_size=24)
IDL> subtitle = text(0.5, 0.85, str[1], alignment='center', $
>      font_name='Times')
```

After rotating the surface and moving the *z*-axis to the back, we get:

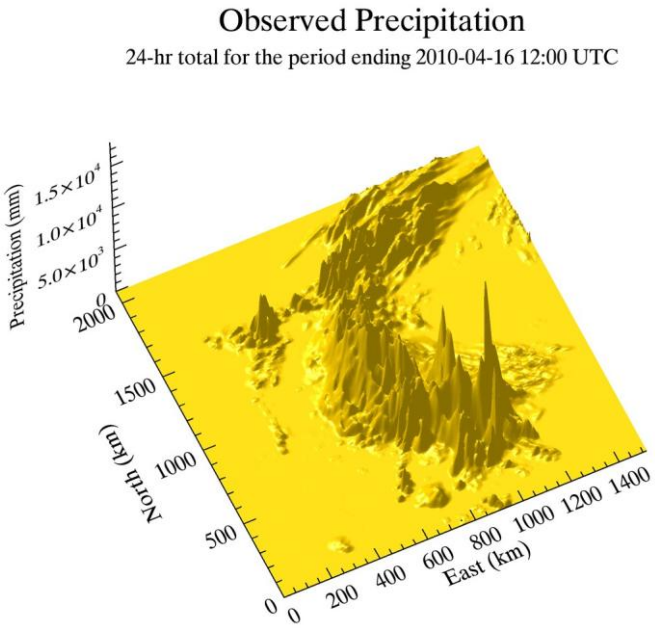


Figure 9-1:

Daily precipitation totals displayed as a shaded surface. Distances are approximate.

The FONT_NAME property can be set to any font available on your computer (e.g., Monaco on a Mac or Arial on Windows). IDL provides four TrueType fonts that are available on every platform: Helvetica, Times, Courier and Symbol. The default font (and FONT_SIZE) is 12 point Helvetica.

The shaded surface provides a striking view of the precipitation data, but it's difficult to quantitatively match the precipitation values to the surface's *z*-axis.

Filled contour plot

We can use the *CONTOUR* function to display the spatial distribution of the rainfall totals as a filled contour plot:

```
IDL> xticks = [0,500,1000,1500]
IDL> yticks = [0,500,1000,1500,2000]
IDL> pc = contour(ppt_ss, xgrid, ygrid, /fill, $
>      rgb_table=27, n_levels=31, rgb_indices=bytscl(indgen(31)), $
>      position=[0.15, 0.25, 0.90, 0.90], /xstyle, /ystyle, $
>      xminor=0, yminor=0, xtickvalues=xticks, ytickvalues=yticks, $
>      xtitle='Distance (km, east)', ytitle='Distance (km, north)', $
>      title='$\bf Observed Precipitation$')
```

The contour plot uses 31 levels of color from a built-in color palette (using the N_LEVELS, RGB_INDICES and RGB_TABLE properties, respectively). The major tick values in the *x*- and *y*-directions have been set to customized values ([XY]TICKVALUES property), with minor ticks completely removed ([XY]MINOR property) to reduce clutter on the graphic. The plot is explicitly positioned in the window with the POSITION property, using the default normalized coordinates.

Place a colorbar along the bottom of the graphic with a call to *COLORBAR*:

```
IDL> cb = colorbar(target=pc, title='Precipitation (mm)', font_size=12, $
>      ticklen=0, position=[300, -500, 1200, -450], /data)
```

Here, the DATA property forces the POSITION values into the coordinate system of the data. Setting TICKLEN to zero suppresses all tick marks.

Use *TEXT* and *ARROW* to include some descriptive annotation on the contour plot. Add a pair of text strings describing the interval over which data were collected

```
IDL> info1 = text(50, 200, '24-hr total', /data, font_size=10)
IDL> info2 = text(50, 100, 'Valid 2010-04-16 1200 UTC', /data, font_size=10)
```

and mark the location of maximum precipitation in this period with

```
IDL> maxval = text(1300, 450, 'max: 272 mm', /data, font_size=10)
IDL> maxarrow = arrow([1050,1280], [470,470], /data, $
>      arrow_style=2, head_size=0.5)
```

Note that these annotations are also positioned in the coordinate system of the data.

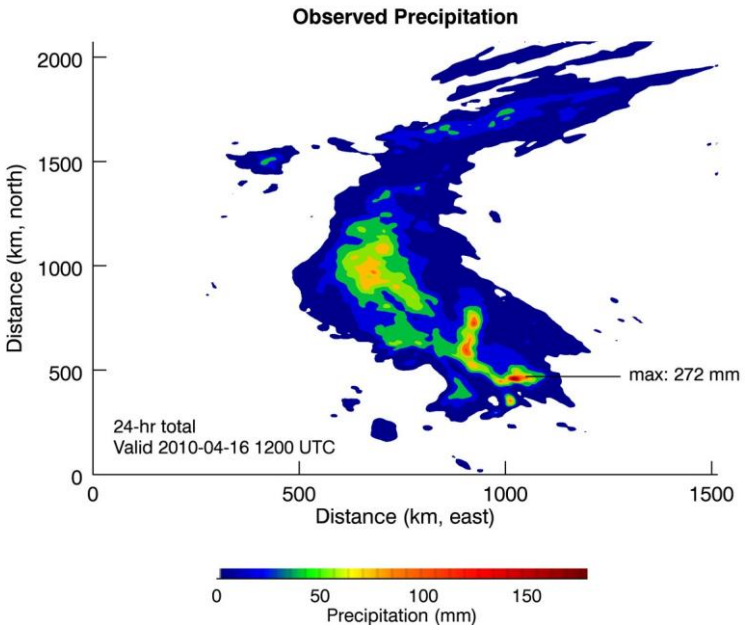


Figure 9-2:

Daily precipitation totals displayed as a filled contour plot

.

Save this graphic to a JPEG file with

```
IDL> pc.save, 'precip-contour.jpg'
```

and check the result with your OS image viewer.

Combined surface and contour plot

Surface and contour plots can be combined in a graphic. For visualization purposes, further reduce the dimensions of the precipitation data using bilinear interpolation. This makes it easier to see individual grid cells in a plot.

```
IDL> ppt_sss = rebin(ppt_ss, ppt_dims/10.0)
IDL> help, ppt_sss
IDL> PPT_SSS          FLOAT       = Array[38, 52]
```

In this statement, the *REBIN* function reduces the size of the subset and smoothed precipitation data array by a factor of 10 using bilinear interpolation.

Call *SURFACE* to display the subset, smoothed and subsampled (hence the "sss" suffix) precipitation data as a wire mesh surface:

```
IDL> gr1 = surface(ppt_sss, title='$\bf Observed Precipitation$', $
>      style=1, color=[64,64,64], aspect_z=0.1)
```

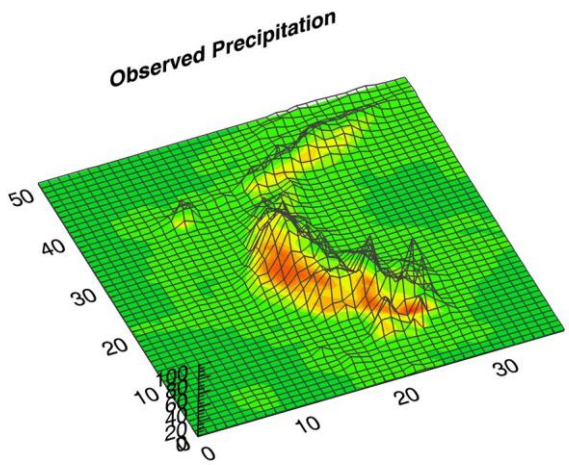
Setting STYLE to 1 makes this a wire mesh surface. The COLOR property can take an RGB triple as input; here, the color is a dark gray. The ratio of the z dimension to the x and y dimensions of the plot, in data coordinates, is set by the ASPECT_Z property.

Use *CONTOUR* to display a filled contour plot of the precipitation values directly beneath the wire mesh surface:

```
IDL> n_levels = 21
IDL> rgb_indices = indgen(n_levels)*10 + 50
IDL> gr2 = contour(ppt_sss, /fill,
>      rgb_table=27, rgb_indices=rgb_indices, n_levels=n_levels, $
>      zvalue=min(ppt_sss), /overplot, /xstyle, /ystyle)
```

Figure 9-3:

Precipitation displayed as a combined wire mesh surface and filled contour plot.



Setting OVERPLOT displays the filled contour plot in the same coordinate system as the wire mesh surface at the position on the vertical axis specified by the ZVALUE property.

More detail on these examples can be found in the course program *DISPLAY_PRECIP*.

Digital terrain elevations

The USGS EROS Data Center provides several global digital elevation data products, including GTOPO30, a global digital elevation model (DEM) with a grid spacing of 30 arc seconds (about 1 kilometer). These data are distributed by the USGS as gridded rasters, which are easily fed into the IDL Graphics routines *SURFACE*, *CONTOUR* and *IMAGE*.

The Satellite Geodesy group at the Scripps Institution of Oceanography runs a CGI script at `http://topex.ucsd.edu/cgi-bin/get_data.cgi` that conveniently allows you to subset and download elevation data from the GTOPO30 DEM. The result is a text file with columns of longitudes, latitudes and elevations. The DEM data are still gridded, but because they're organized as xyz-tuples, they can't be passed directly into *SURFACE* or *CONTOUR*. We'll need to reorganize the data for display.

The file **denver_metro_topo.txt** specifies a subset of the GTOPO30 DEM over the Denver, Colorado metro area, with the upper left corner at (106°W, 41°N) and the lower right corner at (104°W, 39°N). Use *FILE_WHICH* to locate and *READCOL* to read the columns of longitude, latitude and elevation values from this file:

```
IDL> file = file_which('denver_metro_topo.txt')
IDL> readcol, file, lon, lat, elev
IDL> help, lon, lat, elev
LON             FLOAT     = Array[19118]
LAT             FLOAT     = Array[19118]
ELEV            FLOAT     = Array[19118]
```

Attempting to pass the elevation array into *SURFACE* (for example) results in an error:

```
IDL> s = surface(elev)
% SURFACE: Input must be a two-dimensional array.
% Execution halted at: $MAIN$
```

The elevation array needs to be reorganized from a 19118-element vector into an *n* x *m*-element rectangular array, where we need to determine *n* and *m*.

Look at the first few values of these arrays:

```
IDL> for i=0,9 do print, lon[i], lat[i], elev[i]
      254.008      40.9988      2427.00
      254.025      40.9988      2483.00
      254.042      40.9988      2591.00
      254.058      40.9988      2689.00
      254.075      40.9988      2599.00
      254.092      40.9988      2449.00
      254.108      40.9988      2383.00
      254.125      40.9988      2337.00
      254.142      40.9988      2323.00
      254.158      40.9988      2323.00
```

Notice that while the longitude values change linearly, the latitude values repeat. We're moving along a line of latitude in the grid. The point where the repeating latitude value changes marks the edge of the grid.

The *UNIQ* function returns the subscripts of the unique values in an array:

```
IDL> i_unique_lat = uniq(lat)
IDL> print, i_unique_lat[0:4]
         120         241         362         483         604
```

Note that the index of the last element in each set of non-unique elements is returned. From this result, we see there are $n = 121$ longitude values along each line of latitude. Given the total number of elevation values, we calculate there are $m = 19118/n = 158$ latitude values in the grid. In code, this is:

```
IDL> n_lon = i_unique_lat[0] + 1
IDL> n_lat = n_elements(elev) / n_lon
IDL> print, n_lon, n_lat
       121         158
```

Now that we know the dimensions of the grid, we can reorganize the one-dimensional `ELEV` array into a two-dimensional array for visualization:

```
IDL> elev2d = reform(elev, n_lon, n_lat)
IDL> help, elev2d
ELEV2D          FLOAT     = Array[121, 158]
```

The *REFORM* function changes dimensions of an array, keeping the number of elements in the array unchanged.

With these steps, we've overcome the primary obstacle to visualizing these data. There are two details left to address.

We don't need all 19118 longitude and latitude values read from the file. Extract only the 121 unique longitude and 158 unique latitude values:

```
IDL> latvec = lat[i_unique_lat]
IDL> lonvec = lon[0:i_unique_lat[0]]
IDL> help, latvec, lonvec
LATVEC          FLOAT     = Array[158]
LONVEC          FLOAT     = Array[121]
```

These vectors define the nodes of the grid.

IDL mapping routines expect west longitudes to have negative values. The longitude values read from the file are offset by 360 degrees. Remove this offset:

```
IDL> lonvec -= 360.0
IDL> print, min(lonvec), max(lonvec)
      -105.992     -103.992
```

The variables `ELEV2D`, `LONVEC` and `LATVEC` are used to produce the visualizations in the following sections. These steps are reproduced in the course program *PREPARE_TOPO*.

Wire mesh surface plot

Visualize the gridded elevation values versus longitude and latitude as a wire mesh surface, using several customized options:

```
IDL> ss = get_screen_size()
IDL> zvalues = [1e3,3e3,5e3]
IDL> topo = surface(elev2d, lonvec, latvec, $
>      style='mesh', color=[64,64,64], dimensions=0.75*ss, /hidden_lines, $
>      xmajor=3, ymajor=3, ztickvalues=zvalues, $
>      xtitle='Longitude (deg W)', ytitle='Latitude (deg N)', $
>      ztitle='Elevation (m)', title='Denver Topography')
```

The *GET_SCREEN_SIZE* function returns the resolution, in pixels, of the display device. These dimensions are used to size the window used for this graphic with the DIMENSIONS property. Setting the HIDDEN_LINES property activates hidden line removal, so you can't see through the surface. Custom tick values are set on the *z*-axis; note the optional use of scientific notation for the numbers. The number of major tick marks on the *x*- and *y*-axes is prescribed, instead of letting IDL choose them.

The horizontal to vertical aspect ratio of this graphic is highly exaggerated. Compute an aspect ratio of 0.2 with the help of the RANGE course program and apply it:

```
IDL> xz_ratio = (range(lonvec)/range(zvalues))*0.2
IDL> topo.aspect_z = xz_ratio
```

Finally, rotate and scale the graphic into an orientation looking northwest:

```
IDL> topo.scale, 1.25, 1.25, 1.25
IDL> topo.rotate, -75, /zaxis
IDL> topo.rotate, 5, /yaxis
IDL> topo.rotate, 5, /xaxis
```

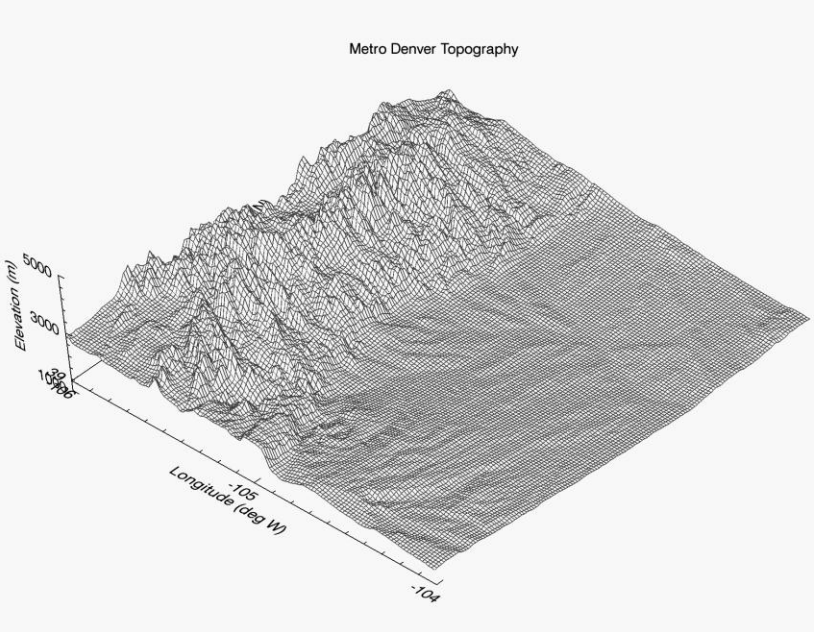


Figure 9-4:

The topography of metro Denver visualized as a wire mesh surface.

All Graphics functions have `Rotate`, `Scale` and `Translate` methods, allowing you to perform coordinate transformations after the graphic has been created. Here, the `Scale` method is used to magnify the graphic by a factor of 1.25 in each of the coordinate dimensions. The `Rotate` method takes its input angle in degrees, using a right-handed coordinate system. The rotation angle is applied to the axis specified by the XAXIS, YAXIS or ZAXIS keyword, with ZAXIS the default.

Save this surface visualization to a PNG file with:

```
IDL> topo.save, 'topo-wire-surface.png', bit_depth=1, resolution=300
```

The BIT_DEPTH property is format-dependent. For a PNG, setting it to `1` produces an 8-bit, palettized image. The RESOLUTION property specifies the size, in dots per inch (dpi), of the output image. The default is 600 dpi.

Three-dimensional contour plot

It's best to know your data, especially when trying to visualize them. The Eastern Plains of Colorado have little relief compared to the Front Range west of Denver. Make a set of custom contour levels to capture the elevation changes in the mountains and in the plains. First, for the mountains, a set of 12 levels, spaced 250 m apart, starting at 1500 m above sea level (about a mile high):

```
IDL> levels = indgen(12)*250 + 1500
```

Then, by array concatenation, prepend a set of elevations with a 50 m spacing for the plains:

```
IDL> levels = [1300, 1350, 1400, 1450, levels]
```

These custom contour levels reflect the slower change in elevation over the plains, and allow the South Platte River valley north and east of Denver to be visualized in the contour plot.

Visualize the topography in the vicinity of Denver as a three-dimensional, filled contour plot using custom contour levels and a built-in color palette:

```
IDL> c3d = contour(elev2d, lonvec, latvec, /fill, $
>      rgb_table=26, c_value=levels, rgb_indices=bytscl(levels), $
>      aspect_z=xz_ratio, planar=0, shading=1, $
>      xmajor=3, ymajor=3, ztickvalues=zvalues, $
>      xtitle='Longitude (deg W)', ytitle='Latitude (deg N)', $
>      ztitle='Elevation (m)', title='Denver Topography')
```

Then, for emphasis, overlay the custom contour levels in black:

```
IDL> c3do = contour(elev2d, lonvec, latvec, $
>      c_value=levels, /overplot, planar=0)
```

Setting the PLANAR property to zero makes a three-dimensional contour plot. Setting the SHADING property blends the colors of the mesh elements that compose the filled contour plot (Gouraud shading); the default is flat shading, where each mesh element has its own color.

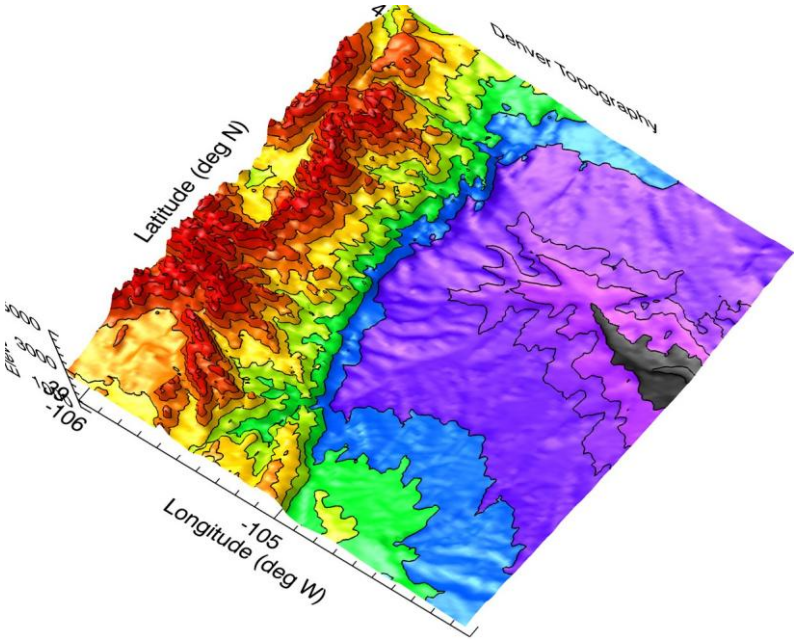


Figure 9-5:

A 3D filled contour plot of the topography around Denver.

Note that the *z*-axis tick values and aspect ratio from the wire-mesh surface example are used again in this graphic.

Texture-mapped surface plot

The TIFF file **denver_metro_nlcd.tif** in the IDL coursefiles **data** directory contains a land cover classification image derived from the USGS National Land Cover Data 1992 dataset for the same geographic area covered by the elevation data used in this chapter. Locate this file with *FILE_WHICH* and read it with *READ_IMAGE*:

```
IDL> file = file_which('denver_metro_nlcd.tif')
IDL> land_cover = read_image(file)
IDL> help, land_cover
LAND_COVER      BYTE      = Array[3, 500, 800]
```

With *SURFACE*, you can warp an image onto a three-dimensional filled surface, a process known as texture mapping. Display the elevation data as a surface and apply the land cover image as a texture map:

```
IDL> a = surface(elev2d, lonvec, latvec, $
>      aspect_z=xz_ratio, shading=1, $
>      texture_image=land_cover, /texture_interp, $
>      xmajor=3, ymajor=3, ztickvalues=zvalues, $
>      xtitle='Longitude (deg W)', $
>      ytitle='Latitude (deg N)', $
>      ztitle='Elevation (m)', $
>      title='Metro Denver Land Use / Topography')
```

Here, the TEXTURE_IMAGE property is used to specify the image to use in the texture map. Setting the TEXTURE_INTERP property tells *SURFACE* to use bilinear interpolation to warp the image to the surface (note the differing dimensions of the arrays `ELEV2D` and `LAND_COVER`); the default is nearest-neighbor sampling.

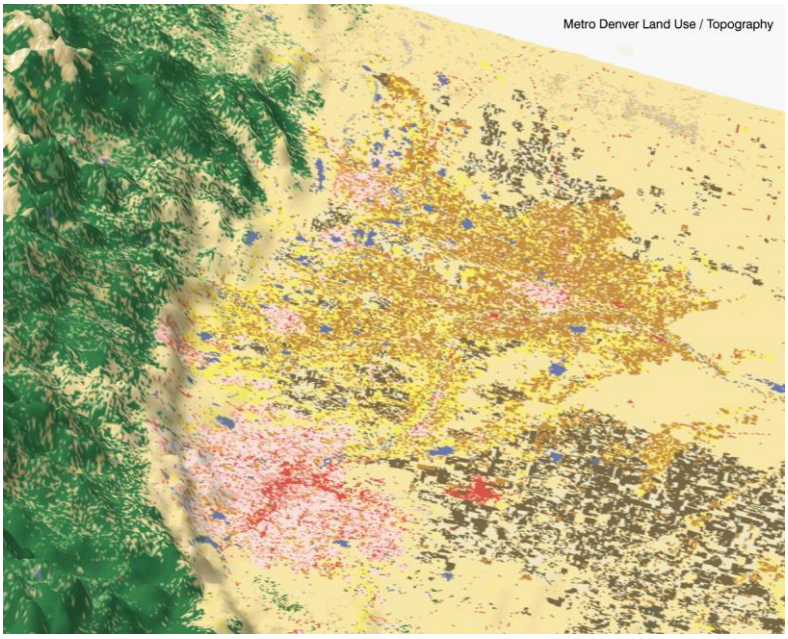


Figure 9-6:

USGS National Land Cover data texture mapped onto a surface representation of the topography near Denver.

Note that the surface has been rotated and magnified. The title has also been moved. Additional details (such as these) on the graphics created in this section can be found in the course program *DISPLAY_TOPO*.

A remark

One topic not addressed in this section (or even in this chapter) is how to visualize spatial data that are scattered, not gridded. For example, imagine you're given ozone concentration readings at a few U.S. cities, such as:

Table 9-2: Ozone concentration values for selected U.S. cities, 2010-04-27 5:00 pm LT.

City	Longitude (deg)	Latitude (deg)	Ozone Concentration (ppb)
Madison, WI	-89.39	43.08	45
Pittsburgh, PA	-79.98	40.44	46
Raleigh, NC	-78.66	35.82	44
Minneapolis, MN	-93.27	44.96	34
Huntsville, AL	-86.63	34.71	37
Albuquerque, NM	-106.62	35.12	52

How could you visualize these data as a surface or contour plot? Both *SURFACE* and *CONTOUR* require two-dimensional, gridded (regular or irregular) input.

The most common technique for addressing this situation is to the interpolate the scattered data to a regular grid. This topic is discussed in greater detail in Chapter 10, "Analysis."

Exercises

1. Display a wire-mesh surface where the color of the surface is proportional to the height of the surface. (Hint: use the VERT_COLORS property of *SURFACE*.)
2. Assuming the digital elevation model (DEM) stored in the IDL SAVE file **lvdem.sav** in the IDL coursefiles **data** directory has one kilometer pixels, and the values of the DEM are in meters, what would be an appropriate value for the ASPECT_Z property to give a 5x vertical exaggeration? Visualize the result as a surface or 3D contour plot.

For answers to these problems, see the programs *SURFACE_VERT_COLORS_EX* and *VERTICAL_EXAGGERATION_EX* in the **introduction/src** directory of the IDL coursefiles.

References

About the Precipitation Analysis Pages. NOAA/National Weather Service, 2010. <http://water.weather.gov/precip/about.php>. Accessed 2010-04-16.

Information about the NWS precipitation product used in this chapter.

Global 30 Arc-Second Elevation (GTOPO30). USGS EROS Data Center, 2009. <http://eros.usgs.gov/Find_Data/Products_and_Data_Available/GTOPO30> Accessed 2010-04-22.

A description of the USGS GTOPO30 digital elevation model, as well as links to download the data. A README file, in Word format, is available at this site.

Measured and Estimated Seafloor Topography. Scripps Institution of Oceanography, University of California San Diego, Satellite Geodesy Group, 2009. <http://topex.ucsd.edu/marine_topo/mar_topo.html>. Accessed 2010-04-23.

Smith, W. H. F., and D. T. Sandwell, *Global seafloor topography from satellite altimetry and ship depth soundings.* Science, **277**, 1957-1962, 1997.

The reference website and journal article for the gridded GTOPO30 DEM data used in the latter half of this chapter. Note that the seafloor topography data actually aren't used. Data: SIO, NOAA, U.S. Navy, NGA, GEBCO.

National Land Cover Data 1992 (NLCD 92). USGS EROS Data Center, 2009. <http://eros.usgs.gov/Find_Data/Products_and_Data_Available/NLCD_92> Accessed 2010-04-23.

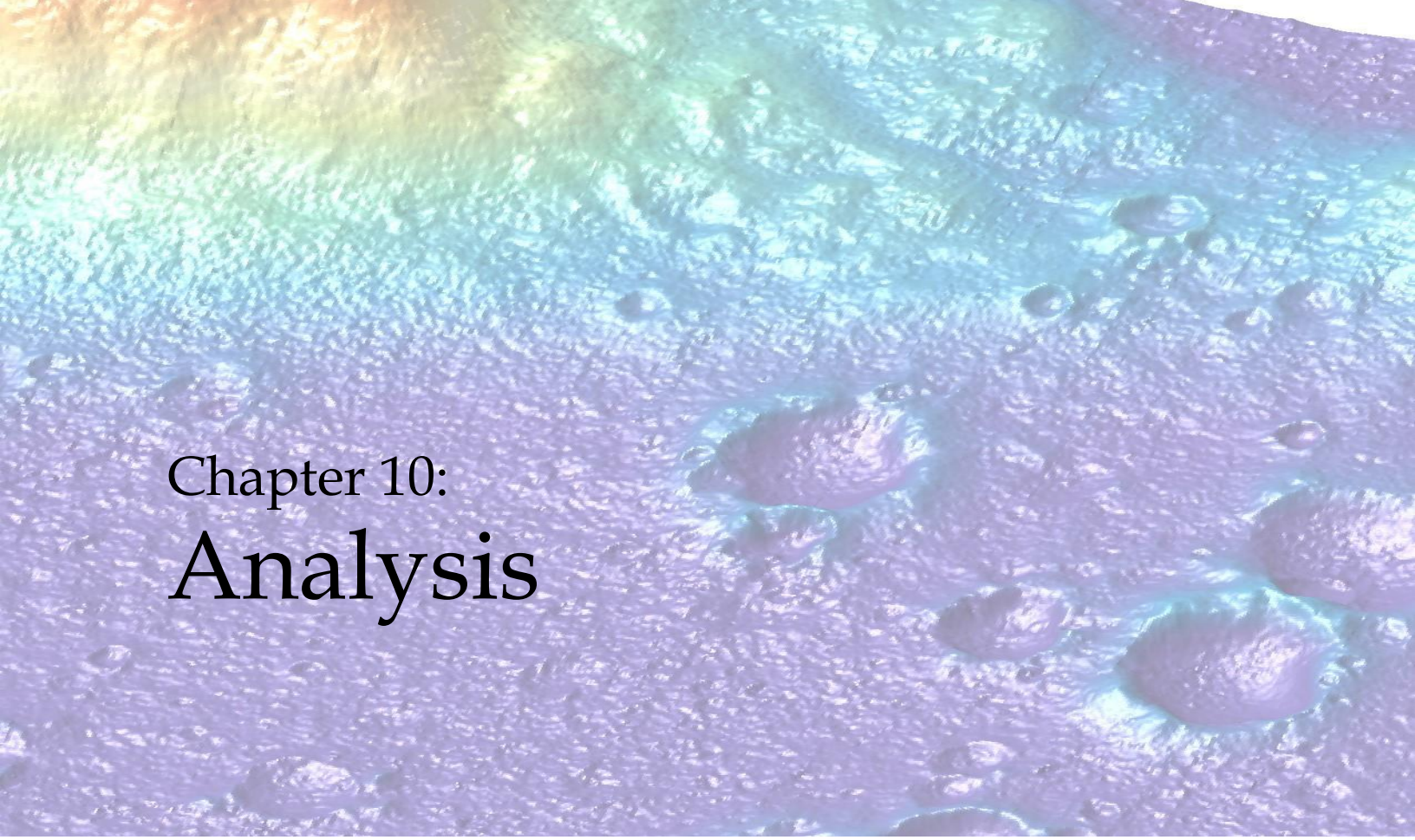
This site provides a description of the USGS National Land Cover Data 1992 land cover classification scheme, as well as links for download.

Air Quality Forecast Guidance. NOAA/National Weather Service, 2010. <http://www.nws.noaa.gov/aq/>. Accessed 2010-04-27.

The location of the NWS air quality data used in Table 9-2.

Tufte, Edward R. *The Visual Display of Quantitative Information*. Second edition. Cheshire, Connecticut: Graphics Press, 2002.

A modern classic on the theory and design of infographics.



Chapter 10: Analysis

This chapter gives examples of using built-in IDL routines for performing interpolation, curve fitting, signal processing and image processing.

If the numbers are boring, then you've got the wrong numbers.

—Edward Tufte, *Envisioning Information*

Introduction 130
Interpolation 130
Curve fitting 134
Signal processing 138
Image processing 140
Exercises 146
References 146

Introduction

IDL is a robust analysis tool, with an extensive library of built-in routines. These routines are documented, with examples, in the IDL Help system. In this chapter, we'll show examples of using analysis routines in four categories: interpolation, curve fitting, signal processing and image processing. These categories were selected because of their general nature and frequent use by IDL programmers.

An add-on to IDL, the IDL Math & Stats Module, includes an IDL interface to the IMSL numerical libraries under a license from Visual Numerics, Inc. These routines aren't discussed in this chapter, which focuses on the routines in the standard IDL distribution. For more information on the IDL Math & Stats Module, check the Exelis VIS website, http://www.exelisvis.com/ProductsServices/IDL/IDLModules/IDLMathandStats.aspx.

Interpolation

Observations of physical processes are often discrete. A problem that frequently arises is the need to estimate the value of a process at an intermediate, unsampled location. Say, for example, a person records the temperature outside every hour, on the hour, for a day. How can they estimate the temperature at 3:41 pm? This is where interpolation, the process of constructing a curve connecting a set of known values, is used.

By definition, an interpolant must intersect each known point. Interpolants are usually smooth and continuous. Interpolation with polynomials and splines, as well as with nearest-neighbor sampling and kriging, are among the methods available in IDL. Refer to the Help system for a complete list of interpolation routines included in IDL.

A pair of examples follows.

Interpolation with cubic splines

A cubic spline is a piecewise polynomial interpolating function. Between each original data point a third-order polynomial is constructed; these polynomials are connected to form the interpolant. The key feature is that the first and second derivatives of the piecewise polynomials must match across their endpoints. This gives cubic splines their advantage in interpolation—they are continuous and smooth.

Generate a sequence of 10 points that lie on a sine curve. Add Gaussian values with a standard deviation of 0.1 to the curve to simulate measurement error.

```
IDL> n = 10
IDL> x = findgen(n)
IDL> y = sin(x) + randomn(seed, n)*0.1
```

Plot these 10 points as plus signs, using the short Graphics format:

```
IDL> p = plot(x, y, '+ ', xtitle='x', ytitle='y')
```

Generate a new set of abscissas that are dense (here, by a factor of four) within the original points. Interpolate the original data at these new points using the *SPLINE*

function. The return from *SPLINE* is an array of the interpolated values at the new abscissas.

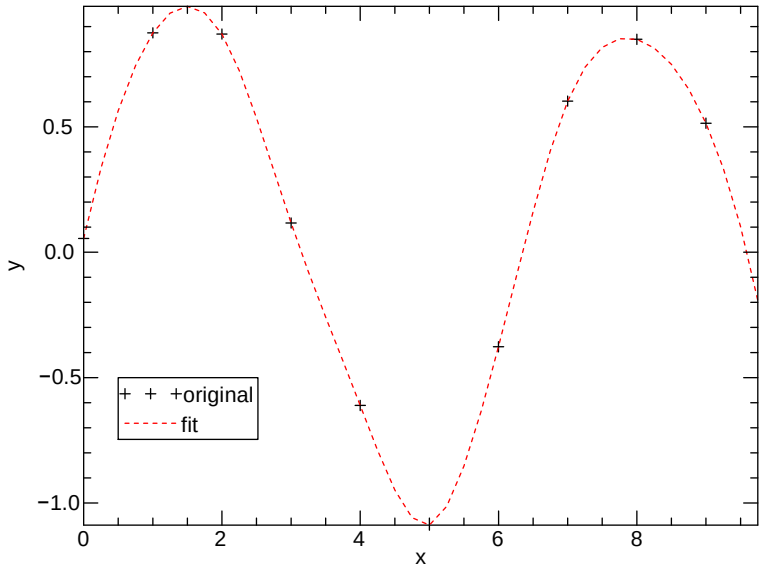
```
IDL> new_x = n*findgen(n*4)/(n*4)
IDL> new_y = spline(x, y, new_x)
```

Plot the interpolated points on top of the originals with a red, dotted line:

```
IDL> q = plot(new_x, new_y, 'r:', /overplot)
```

Figure 10-1:

An example of cubic spline interpolation with *SPLINE*.



Note that the interpolant is smooth as it passes through each of the original points. For more detail, see *SPLINE_EX1* in the IDL coursefiles **introduction** directory.

Gridding irregularly spaced data

The National Weather Service Automated Surface Observing System (ASOS) provides observations of temperature, pressure, dew point, wind, precipitation and other meteorological variables at several hundred airports throughout the country.

Here we attempt to visualize the surface pressure field from a group of ASOS stations spaced irregularly over the Great Plains. Start by using the *READ_ASOS* course program to read the data from an ASOS file included in the IDL coursefiles **data** directory:

```
IDL> asos_file = file_which('20_02.dat')
IDL> asos = read_asos(asos_file)
```

The variable `ASOS` is a structure with fields containing arrays of data from 56 ASOS stations. The fields give the identifier and position for each station, as well as meteorological data. Display a planview of the locations of the stations, marking each station with a cross:

```
IDL> stations = plot(asos.lon, asos.lat, 'X ', $
>      xtitle='Longitude (deg)', ytitle='Latitude (deg)')
```

The field PRESSURE contains surface pressure measurements in units of Pascals. There are several bad data points in this field marked with the value 999999.0. Find them using the *WHERE* function:

```
IDL> tolerance = 1.0
IDL> bad_value = 999999.0
IDL> bad = where(abs(asos.pressure - bad_value) lt tolerance, n_bad)
IDL> print, n_bad
      14
```

Because of the way floating-point numbers are constructed (see "Type behaviors in IDL" on page 46), it may be difficult to find where values of PRESSURE exactly match the bad data value of 999999.0. A recommended technique is to use *WHERE* to find the values that fall within a user-defined neighborhood of the bad data value.

The *GRID_INPUT* procedure is used to preprocess data for gridding. It can remove bad and duplicate values. Pass the locations of the stations and their corresponding pressure values:

```
IDL> grid_input, asos.lon, asos.lat, asos.pressure, $
>     lon_qc, lat_qc, pressure_qc, exclude=bad, epsilon=0.25
```

Output is to three new variables: LON_QC, LAT_QC and PRESSURE_QC, representing the quality-checked station locations and pressure values. The EXCLUDE keyword is set to the indices of the bad data points. If more than one station is within the radius value set with the EPSILON keyword, only the first station's data are used.

Compare the positions of the quality checked data points with the original points:

```
IDL> keepers = plot(lon_qc, lat_qc, 'sg ', /sym_filled, /overplot)
```

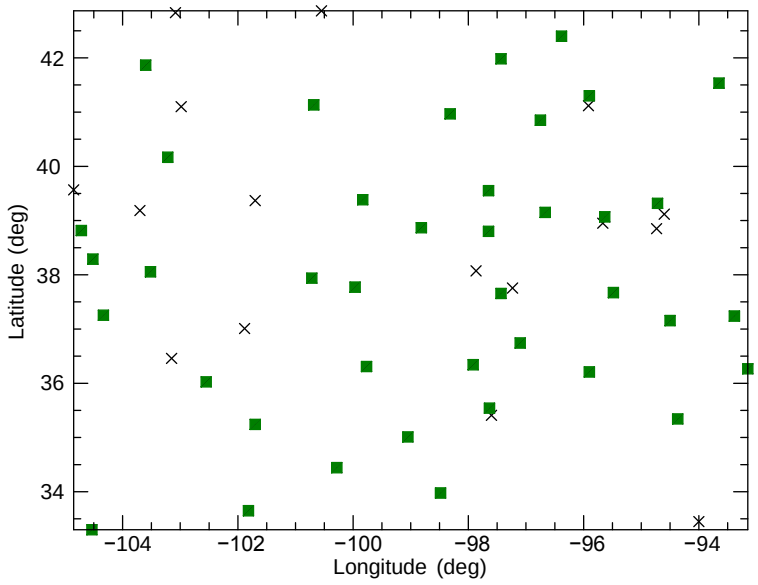


Figure 10-2:

The locations of the ASOS stations. Stations marked with a box are included in the analysis.

The ASOS stations with viable data are highlighted with a green box. Stations without boxes are removed from further analysis.

Next, set up a grid. These statements define a 21 x 17 grid with 0.5 degree resolution, on the domain [104°W < longitude < 94°W] and [34°N < latitude < 42°N]:

```
IDL> n_lon = 21
IDL> n_lat = 17
IDL> resolution = 0.5 ; degrees
IDL> start_lon = -104.0
IDL> start_lat = 34.0
IDL> lon_vec = findgen(n_lon)*resolution + start_lon
IDL> lat_vec = findgen(n_lat)*resolution + start_lat
```

Finally, call *GRIDDATA* to interpolate the pressure values from the irregularly spaced stations to a regular grid:

```
IDL> pressure_grid = griddata(lon_qc, lat_qc, pressure_qc, $
>      /inverse_distance, power=3, /sphere, /degrees, $
>      /grid, xout=lon_vec, yout=lat_vec)
```

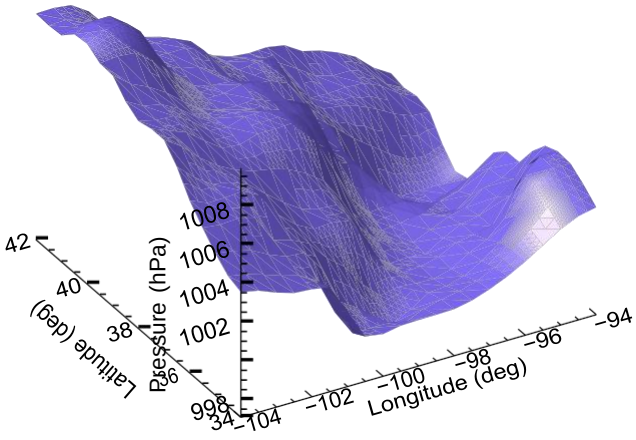
GRIDDATA takes the quality-checked variables from *GRID_INPUT*, as well as the longitude and latitude grid nodes through the XOUT, YOUT and GRID keywords. The inverse distance method with a cubic power law is used for interpolation. A better method could be chosen based on knowledge of the data (pressure is a smooth field variable at this scale in the atmosphere). SPHERE and DEGREES are set because the inputs are latitudes and longitudes; i.e., spherical coordinates, in decimal degrees.

Display the gridded surface pressure data after converting the pressure values from Pa to hPa, a meteorological convention:

```
IDL> pressure_grid *= 1.0e-2
IDL> p = surface(pressure_grid, lon_vec, lat_vec, $
>      color='slate blue', shading=1, xtitle='Longitude (deg)', $
>      ytitle='Latitude (deg)', ztitle='Pressure (hPa)')
```

Figure 10-3:

The pressure surface from the gridded ASOS data.



See the program *IRREGULAR_GRID_EX* in the IDL coursefiles for more information.

Curve fitting

Curve fitting is the process of representing a set of observations (e.g., measurements from an instrument) by a model with a smaller set of tunable parameters. The selection of the model is critical; it's usually based on the theory underlying the observations. The model parameters are adjusted to optimize a statistic of merit, which can be used to quantify how well the model fits the data. Additional goodness-of-fit tests can also be performed to verify the appropriateness of the model.

The IDL Help system has a list of curve and surface fitting routines, as well as goodness-of-fit tests, included in IDL.

Two examples of curve fitting are presented in the following sections.

Least-squares linear fit

A resistance temperature device (RTD) makes use of the linear resistivity of some metals, such as copper and platinum, to measure temperature. The relationship between resistance (R) and temperature (T) in an RTD can be expressed, to first order, as:

$$R(T) = a_0 + a_1 T$$

where the coefficients a_0 and a_1 are determined by calibration. One technique for calibrating an RTD involves passing a constant current through it, and, by Ohm's Law ($V=IR$), recording its temperature response to known voltages; in effect:

$$T(V) = b_0 + b_1 V$$

where b_0 and b_1 can be determined by a linear least-squares regression of the collected temperatures to the input voltages.

The file **rtdcalib.dat**, located in the IDL coursefiles **data** directory, is a text file with a set of voltages (in volts) and response temperatures (in degrees Celsius) that were used to calibrate an RTD. Read this file with *READCOL*:

```
IDL> file = file_which('rtdcalib.dat')
IDL> readcol, file, voltage, temperature, skipline=4
IDL> help, voltage, temperature
VOLTAGE         FLOAT     = Array[2000]
TEMPERATURE     FLOAT     = Array[2000]
```

Take a quick look at these variables with *PLOT*:

```
IDL> p = scatterplot(voltage, temperature, SYMBOL='dot')
```

Use the *LINFIT* function to determine the linear least-squares fit to these data:

```
IDL> coefficients = linfit(voltage, temperature, chisq=chisq, $
>     prob=prob_chisq, sigma=sigma_coefficients)
```

The authors of *Numerical Recipes in C* state that a fitting routine should provide coefficient values, error estimates for the coefficients and a statistical measure of the goodness-of-fit. These are calculated in *LINFIT* and returned through keyword parameters.

Print the coefficients, their error estimates, the chi-square value and its confidence:

```
IDL> print, coefficients ; intercept, slope
       20.0080      311.830
IDL> print, sigma_coefficients
       0.00573102     0.258579
IDL> print, chisq, prob_chisq ; unreduced chi-square
       42.1272      1.00000
```

The chi-square value is much greater than 1.0, so the model fit is believable. The probability of the chi-square value being this large or larger is 1.0, indicating that we have confidence in the goodness-of-fit of this calculation.

Display the original points and the linear fit as a scatter plot, using the POSITION property to leave room at the bottom of the graphic for a second plot:

```
IDL> original = plot(voltage, temperature, $
>      linestyle='none', symbol='dot', $
>      position=[0.15, 0.35, 0.95, 0.90], $
>      ytitle='Temperature ($\deg C$)', $
>      title='Resistance Temperature Device Calibration')
```

Using the calculated fit coefficients, make a line of regression (consisting of two points) to demonstrate the fit. Overplot it on the original data.

```
IDL> v_min = min(voltage, max=v_max)
IDL> v_fit = [v_min, v_max]
IDL> t_fit = coefficients[0] + coefficients[1]*v_fit
IDL> print, v_fit, t_fit
     0.000000    0.0398590
      20.0080      32.4373
IDL> fit = plot(v_fit, t_fit, /overplot, thick=2, color='crimson')
```

Calculate the residuals of the linear regression. Display the residuals in a second plot on the bottom of the graphic.

```
IDL> residuals = temperature - (coefficients[0] + coefficients[1]*voltage)
IDL> res = plot(voltage, residuals, /current, $
>      position=[0.15, 0.10, 0.95, 0.30], ytickvalues=[-0.4, 0.0, 0.4], $
>      color='slate blue', xtitle='Voltage (V)')
```

Overplot a zero line on the residuals:

```
IDL> zero_line = plot(res.xrange, res.xrange*0.0, color='gray', /overplot)
```

Note how the actual *x*-axis plot range (the XRANGE property) was extracted through `RES`, its parent.

Use the HIDE property to hide the bottom axis of `ORIGINAL` and the top axis of `RES`:

```
IDL> (original.axes)[0].hide = 1
IDL> (res.axes)[2].hide = 1
```

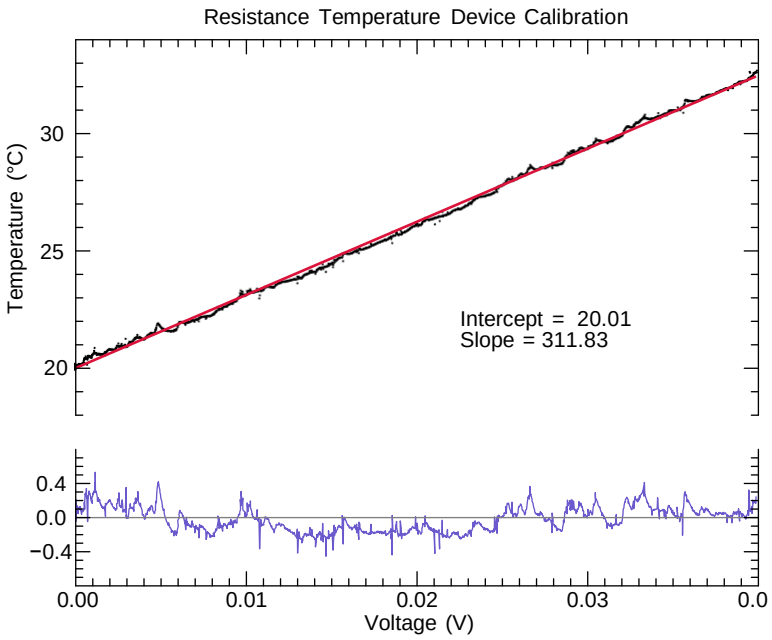


Figure 10-4:

An example of linear least-squares regression with *LINFIT*, including a plot of the residuals.

For more, see *LINFIT_EX* in the IDL coursefiles **introduction/src** directory.

Least-squares user-defined fit

Often, a nonlinear fit is needed. The *SVDFIT* function computes a least-squares fit to a user-supplied model using a singular value decomposition technique. The model must be written as an IDL function.

Create a vector of heights (in meters) and 15-minute mean and standard deviation wind speed values (in meters per second) at those heights:

```
IDL> heights = [0.1, 3.0, 5.0, 10.0] ; meters
IDL> wind_mean = [0.0, 4.83, 5.34, 6.14] ; m/s
IDL> wind_stdev = [0.0, 0.99, 1.01, 0.96] ; m/s
```

The first height is an estimate of the aerodynamic roughness length, the level above the ground at which the wind speed is zero. The remaining heights are the levels at which the instruments—propeller-vane anemometers—are mounted. These data were captured at a field site in eastern Kansas.

Display an error plot of the wind profile, with the independent variable `HEIGHTS` on the vertical axis and the mean and standard deviation wind profiles displayed on the horizontal axis:

```
IDL> pdata = errorplot(wind_mean, heights, wind_stdev, wind_stdev*0.0, $
>      symbol='circle', /sym_filled, sym_color='red', $
>      xrange=[0,10], linestyle='none', name='Data', $
>      xtitle='Wind speed ($m s^{-1}$)', ytitle='Height (m)', $
>      title='Mean Vertical Wind Profile')
```

The mean wind profile is displayed with filled red circles at the four heights. The standard deviation wind speed values are displayed with error bars.

The "law of the wall" from fluid mechanics suggests the use of a logarithmic or logsquare model to describe these measurements. If U is wind speed and z is height, the logsquare model can be written

$$U(z) = a_0 + a_1(\ln z) + a_2(\ln z)^2$$

which can be solved for the coefficients a_0, a_1 and a_2. In IDL, we represent this model with the function *WIND_PROFILE_MODEL*, found in the file of the same name in the IDL coursefiles **introduction/src** directory.

```
function wind_profile_model, x, m

   return, [1.0, alog(x), alog(x)^2]
end
```

Note that the return array has three elements, one for each term of the logsquare model. The two parameters, `X` and `M`, are required by *SVDFIT*.

Call *SVDFIT* to perform the curve fit:

```
IDL> c = svdfit(heights, wind_mean, chisq=chisq, sigma=sigma, $
>      a=[1.0, 1.0, 1.0], function_name='wind_profile_model')
```

The keyword A is used to provide initial guesses for the model coefficients. The FUNCTION_NAME keyword specifies the name of the IDL function representing the model. CHISQ and SIGMA are output keywords, giving the chi-square value of the fit and the standard deviations of the coefficients, respectively. The values of the coefficients are returned in the variable `C`. Print the coefficients, the error estimates on the coefficients and the chi-square value:

```
IDL> print, c
      3.42586      1.32973   -0.0684753
IDL> print, sigma
      0.0695748    0.0164516    0.0164704
IDL> print, chisq
      0.00301076
```

Use the coefficients returned from *SVDFIT* to make a vector describing the fit:

```
IDL> n_fit_points = 50
IDL> max_height = 12 ; meters
IDL> z_fit = findgen(n_fit_points)/n_fit_points*max_height + heights[0]
IDL> u_fit = c[0] + c[1]*alog(z_fit) + c[2]*alog(z_fit)^2
```

Plot the fit over the original data points and include a legend:

```
IDL> pfit = plot(u_fit, z_fit, color='blue', name='Fit', /overplot)
IDL> plegend = legend(target=[pdata, pfit], position=[0.75, 0.85])
```

LEGEND uses the NAME properties of the two plots, `PDATA` and `PFIT`, for its labels. The POSITION property is specified in normalized coordinates, the default.

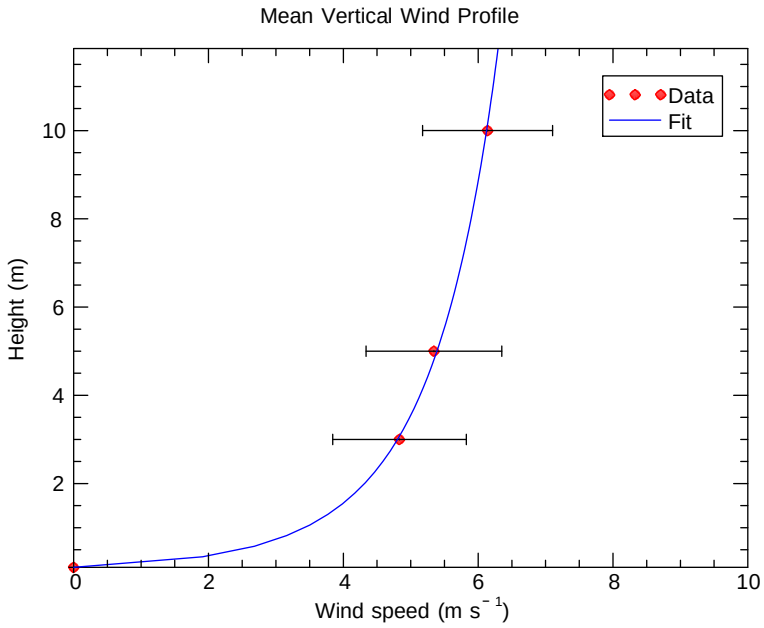


Figure 10-5:

Wind profile data fit with a logsquare model using *SVDFIT*.

For an expanded version of this example, see the program *SVDFIT_EX* in the IDL coursefiles **introduction** directory.

Signal processing

A signal is the record of a process that occurs over an interval of time, space or some arbitrary measure. A signal, by definition, contains information, but any signal obtained from a physical process also contains noise. It is often difficult to make sense of the information contained in a signal by looking at it in its raw form. Signal processing is the body of methods used to analyze and extract quantitative information from signals. Various disciplines in science and engineering have developed techniques to extract information from the signals they study.

IDL has a variety of tools for the processing of one- and higher-dimensional signals. Most are based on the fast Fourier transform (FFT) or the wavelet transform (WT). Refer to the IDL Help system for a complete list of signal processing routines included in IDL.

Here's an example of using the *FFT* function to compute a power spectrum of a time series. The data are the sunspot series used in Chapter 4, "Line, Bar and Scatter Plots", a record of monthly sunspot numbers dating back to the year 1749, downloaded from the web site of the NASA MSFC Solar Physics Group. Read the sunspot numbers file:

```
IDL> file = file_which('spot_num.txt')
IDL> readcol, file, year, month, sunspots
IDL> help, year, month, sunspots
YEAR            FLOAT     = Array[3135]
MONTH           FLOAT     = Array[3135]
SUNSPOTS        FLOAT     = Array[3135]
```

For a depiction of these data in the time domain, see Figure 4-7.

Store as variables the number of values in the sunspot series and the time interval between the measurements:

```
IDL> n_samples = n_elements(sunspots)
IDL> sampling_interval = 1/12.0 ; one month, in years
```

Use these values to derive the period, the fundamental frequency and the minimum resolvable, or Nyquist, frequency of the series:

```
IDL> period = n_samples*sampling_interval
IDL> fundamental_freq = 1.0/period
IDL> nyquist_freq = 0.5/sampling_interval ; years^{-1}
```

Transform the sunspot series to the frequency domain with the *FFT* function. The variable SUNSPOTS_HAT gives the Fourier coefficients of the series. It is of type complex.

```
IDL> sunspots_hat = fft(sunspots)
```

Compute magnitude and power spectra from the coefficients:

```
IDL> sunspots_mag = abs(sunspots_hat)
IDL> sunspots_pwr = sunspots_mag^2
```

Make a zero-based frequency vector for the positive Fourier modes. (IDL stores positive Fourier modes first, followed by negative modes.)

```
IDL> n_positive_modes = n_samples/2
IDL> frequency = findgen(n_positive_modes)*fundamental_freq
```

For visualization, set up a one-sided power spectrum by folding in the negative Fourier modes. Omit the zero, or DC, frequency from the visualization.

```
IDL> frequency_nodc = frequency[1:n_positive_modes-1]
IDL> sunspots_pwr_nodc = 2.0*sunspots_pwr[1:n_positive_modes-1]
```

Plot the power spectral density versus frequency on logarithmic axes by setting the XLOG and YLOG properties:

```
IDL> p = plot(frequency_nodc, sunspots_pwr_nodc, /xlog, /ylog, $
>      xtitle='Frequency ($years^{-1}$)', ytitle='Spectral Density', $
>      title='Power Spectrum of Sunspot Numbers, 1749-2010')
```

The power spectrum has a peak near 0.1 yr^{-1}. Locate the peak frequency with *MAX* and mark it on the plot with *POLYLINE*.

```
IDL> sunspots_pwr_nodc_peak = max(sunspots_pwr_nodc, i_peak)
IDL> frequency_nodc_peak = frequency_nodc[i_peak]
IDL> print, sunspots_pwr_nodc_peak, frequency_nodc_peak
       388.082    0.0919540
IDL> !null = polyline([1.0,1.0]*frequency_nodc_peak, p.yrange, $
>      color='orange', /data)
```

The inverse of the peak frequency gives the period of the sunspot cycle: approximately 11 years. Mark this on the plot with *TEXT*:

```
IDL> sunspot_peak_period = 1.0/frequency_nodc_peak
IDL> str = '$T_{peak}$ = ' + strtrim(sunspot_peak_period,2) + ' years'
IDL> !null = text(1e-1, 1e-4, str, /data, font_size=12)
```

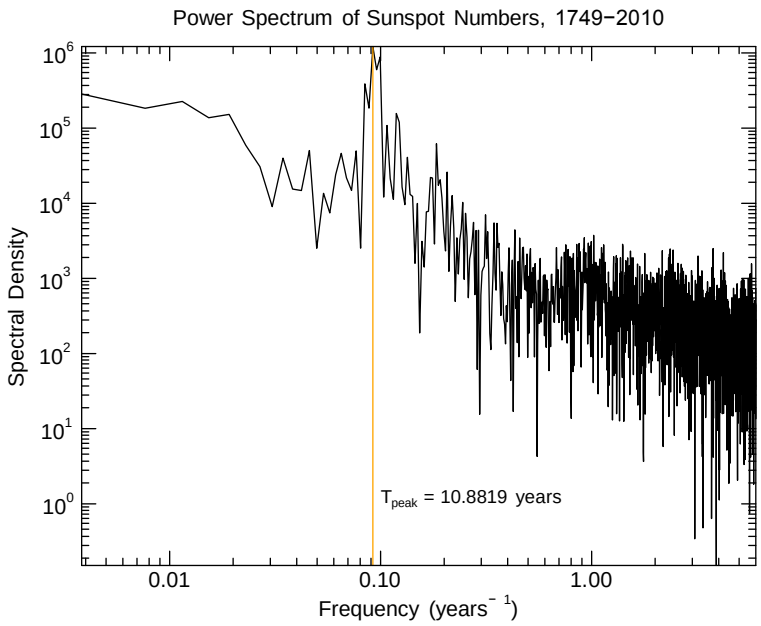


Figure 10-6:

The power spectrum of the sunspot series. The frequency associated with the sunspot cycle is marked in orange.

More detail on this signal processing example can be found in the program *ANALYZE_SUNSPOT_SERIES* in the IDL coursefiles **introduction** directory.

Image processing

Much scientific information, in fields ranging from cellular biology to medicine to astronomy, is communicated through the use of imagery. To extract information from imagery, researchers rely on the body of methods called image processing. Here, we look at a few examples of image processing.

Note that the Exelis VIS Documentation Group produces a guide, *Image Processing*, as a part of the IDL documentation set. It can accessed, in PDF form, from the **help/pdf** directory under the main IDL directory on your computer. It's a great resource that contains almost 300 pages of examples of image processing with IDL.

Histogram equalization

Histogram equalization rearranges image pixel values to spread the image's histogram over the byte range. This typically enhances the contrast between neighboring regions. Some pixels with values that are different initially may be assigned the same value. Other pixel values may be missing entirely. Histogram equalization is an example of global contrast enhancement.

Start by reading an image from a file. This image represents a model of convection in the Earth's mantle.

```
IDL> file = filepath('convec.dat', subdir=['examples','data'])
IDL> mantle = read_binary(file, data_dims=[248,248])
```

Display the image as the first item in a 2 x 2 grid of graphics using *IMAGE* and the LAYOUT property:

```
IDL> orig = image(mantle, layout=[2,2,1], title='Original Image', $
>      window_title='Histogram Equalization Example')
```

Use *HISTOGRAM* to calculate the image's histogram, or frequency distribution; recall this is the graph of the number of times a pixel value occurs in an image:

```
IDL> orig_histogram = histogram(mantle)
```

Display the histogram in the second position in the grid:

```
IDL> hist = plot(orig_histogram, layout=[2,2,2], /current, $
>      color='red', max_value=5e3, $
>      xtitle='Pixel value', ytitle='Frequency', title='Histogram')
```

Any value greater than MAX_VALUE is not plotted. In this case, it excludes the white (value 255) and black (value 0) pixel counts in the image, which otherwise dominate the histogram. Also note the use of scientific notation: 5e3 = 5000. IDL can use either scientific or standard notation for numbers.

The histogram displays a narrow peak, indicative of low contrast in the image. Apply *HIST_EQUAL* to the image to spread its histogram, thereby increasing the contrast in the mantle region between the core and surface. Display the result in the third position of the grid.

```
IDL> equ_mantle = hist_equal(mantle)
IDL> equ = image(equ_mantle, layout=[2,2,3], /current, $
>      title='Equalized Image')
```

Last, calculate and display the histogram of the equalized image in the fourth position of the grid:

```
IDL> equ_histogram = histogram(equ_mantle)
IDL> equ_hist = plot(equ_histogram, layout=[2,2,4], /current, $
>      color='blue', max_value=5e3, $
>      xtitle='Pixel value', ytitle='Frequency')
```

Compare your result with Figure 10-7 below.

Save this graphic to an encapsulated PostScript file with

```
IDL> orig.save, 'hist_equal_ex1.eps'
```

This statement is used to produce the EPS file used in the Figure.

For another take on visualizing this result, see *HIST_EQUAL_EX1* in the IDL coursefiles **introduction** directory.

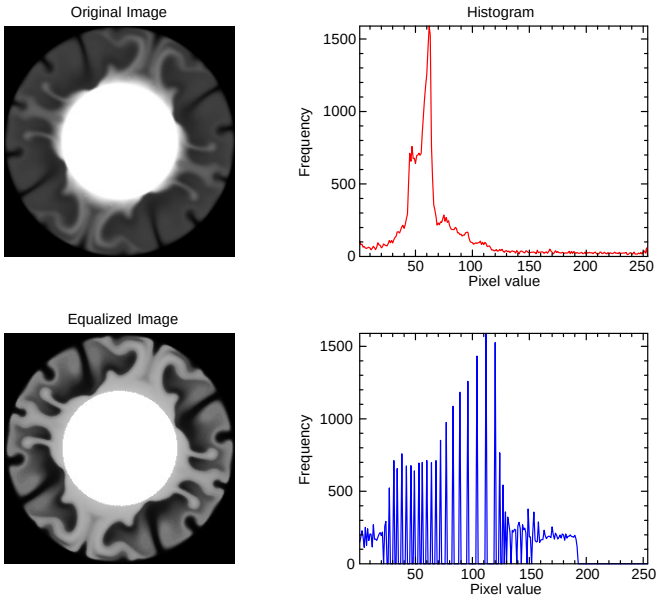


Figure 10-7:

An example of image contrast enhancement with histogram equalization.

Image sharpening

Image sharpening locally enhances contrast in an image by boosting the image's high-frequency components. One technique, unsharp masking, increases the visual acuity of an image by highpass filtering it and adding back a fraction of the filtered image to the original.

Read in an image from a JPEG file. The image is of cultured Neocosmospora vasinfecta, a common plant fungal pathogen.

```
IDL> file = file_which('n_vasinfecta.jpg')
IDL> read_jpeg, file, fungus
IDL> help, fungus
FUNGUS          BYTE      = Array[3, 410, 300]
```

The image FUNGUS is a 410 x 300 pixel-interleaved RGB image. Use the *UNSHARP_MASK* function to sharpen the image:

```
IDL> sharpend = unsharp_mask(fungus, amount=1.0, radius=3)
IDL> help, sharpend
SHARPEND        BYTE      = Array[3, 410, 300]
```

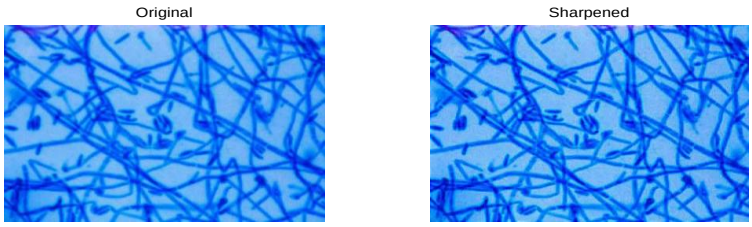
The AMOUNT keyword specifies, on a unit scale, how much of the highpass image to add back to the original, while RADIUS specifies the width, in pixels, of the Gaussian smoothing kernel used to construct the highpass image.

Display the original and sharpened images, side-by-side, with *IMAGE*:

```
IDL> img1 = image(fungus, layout=[2,1,1], title='Original', $
>      window_title='Image Sharpening Example')
IDL> img2 = image(sharpend, layout=[2,1,2], /current, title='Sharpened')
```

Figure 10-8:

An example of increasing contrast in an image by unsharp masking.



Edge enhancement

IDL has several built-in edge enhancement routines. *LAPLACIAN* can be used to emphasize regions of sharp change in images, but because of its symmetry it's insensitive to direction. More specialized operators can be used to emphasize the direction of edges, such as *SOBEL* and *ROBERTS*, which work by estimating derivatives in certain directions. An example of using a few of the IDL edge enhancing routines is given here.

Read the first image from the **people.dat** file and store it in a hash:

```
IDL> filename = filepath('people.dat', subdir=['examples', 'data'])
IDL> ali = hash('Original', read_binary(filename, data_dims=[192,192]))
IDL> help, ali['Original'] ; Note hash key is case-sensitive
<Expression>    BYTE      = Array[192, 192]
```

This is an image of Ali Bahrami, Dave Stern's first employee at RSI. Ali ported the core of IDL from FORTRAN 77 to C.

Apply five edge enhancers from the IDL library to the image:

```
IDL> ali['Sobel'] = sobel(ali['Original'])
IDL> ali['Roberts'] = roberts(ali['Original'])
IDL> ali['Canny'] = canny(ali['Original']) - 1B
IDL> ali['Laplacian'] = laplacian(ali['Original'])
IDL> ali['Prewitt'] = prewitt(ali['Original'])
```

With *IMAGE*, display the original and edge-enhanced images in a 3 x 2 grid:

```
IDL> display = hash() ; to use references later
IDL> index = 1 ; for layout
IDL> w = window(window_title='Edge Enhancement Examples')
IDL> foreach enhanced_image, ali, technique do $
>      display[technique] = image(enhanced_image, layout=[3,2,index++], $
>            title=technique, /current)
```

Note that the ordering of the images is munged by the hash, and that *IMAGE* automatically byte scales non-byte data into the byte range for display.

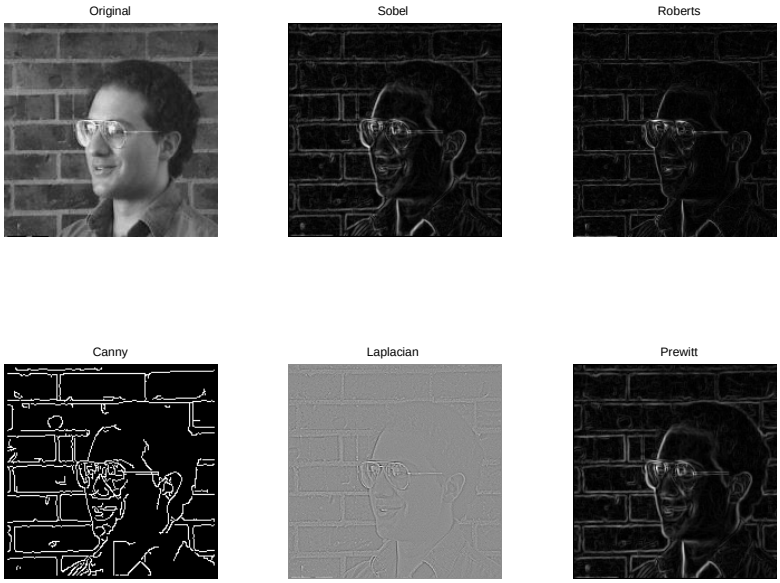


Figure 10-9:

Examples of applying a set of edge enhancement routines to an image.

Fourier filtering

IDL's *FFT* function can operate on arrays of up to 8 dimensions. Here, use it to lowpass filter an intensity mapped image with an ideal filter.

Use *READ_DICOM* to read an MR image of a person's knee, taken in the sagittal plane:

```
IDL> file = file_which('mr_knee.dcm')
IDL> knee = read_dicom(file)
IDL> help, knee
KNEE            INT       = Array[256, 256]
```

Using *IMAGE*, display the image, and also set up space to display three more images:

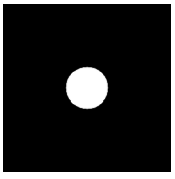
```
IDL> im1 = image(knee, /order, layout=[4,1,1], dimensions=[1000,400], $
>      title='Image', window_title='FFT Lowpass Filtering Example')
```

Medical images are drawn from top to bottom, hence the need for the ORDER property.

Set up an ideal lowpass filter with a cutoff frequency at one-quarter of the image's Nyquist mode in either dimension (since the image is square):

```
IDL> xsize = (size(knee, /dimensions))[0]
IDL> ysize = (size(knee, /dimensions))[1]
IDL> cutoff = (xsize < ysize) / 8 ; 1/4 of Nyquist mode
IDL> ideal_filter = shift(dist(xsize, ysize), xsize/2, ysize/2) le cutoff
```

The *DIST* function builds a distance map: an array in which each value is proportional to its frequency. This is particularly useful for creating image filters. The distance map is shifted by half its width and height to place the lowest frequencies at the center. A thresholding operation is then applied to make the filter—a binary image mask where values less than the cutoff are set to 1 (white) while the rest are 0 (black).



Transform the image to the Fourier domain with *FFT*:

```
IDL> knee_hat = fft(knee, /center)
```

Prepare power spectra to help visualize how the filter works:

```
IDL> power = abs(knee_hat)^2
IDL> power_display = alog10(power)
IDL> power_filtered_display = bytscl(power_display) * ideal_filter
```

A common technique for visualizing image power spectra is to display the base 10 logarithms of the power spectral amplitudes; otherwise, the power spectrum is overwhelmed by the low-frequency modes—you'd see one white pixel in a field of black. The filtered power spectrum is only used to show what Fourier modes are passed and which are blocked by the filter. Display these power spectra with *IMAGE* in the window we've set up:

```
IDL> im2 = image(power_display, /order, layout=[4,1,2], /current, $
>      title='Power Spectrum')
IDL> im3 = image(power_filtered_display, /order, layout=[4,1,3], /current, $
>      title='Filtered Power Spectrum')
```

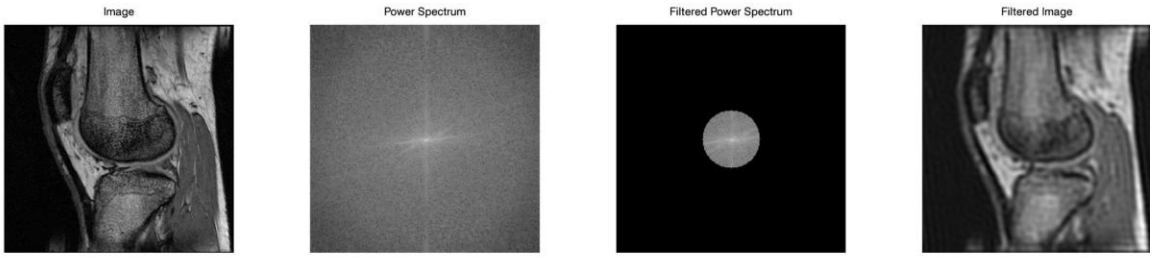
Finally, lowpass filter the image by convolving it with the filter in the Fourier domain and backtransforming with *FFT*:

```
IDL> knee_lowpass = fix(fft(knee_hat*ideal_filter, /inverse, /center))
```

Images values within the filter's cutoff are passed, while those outside are blocked. Display the result with *IMAGE*:

```
IDL> im4 = image(knee_lowpass, /order, layout=[4,1,4], /current, $
>      title='Filtered Image')
```

Figure 10-10: An image, lowpass filtered.



The filtered image is blurred by the lowpass filtering operation. High-frequency information has been removed, making it easier to identify large-scale features in the image.

An ideal filter is the simplest type of filter but because it zeros Fourier modes outside of its width it produces ringing. We'd get better results with an exponential or a Butterworth filter.

Exercises

1. Try constructing a polynomial interpolant to the data presented in "Interpolation with cubic splines" on page 130 using the *POLY_FIT* function.
2. Construct and visualize the negative of an intensity-mapped image. Can you do the same for a color-indexed and an RGB image?

For sample solutions to these exercises, see the programs *POLY_FIT_EX* and *NEGATIVE_IMAGE_EX* in the IDL coursefiles **introduction/src** directory.

References

Automated Surface Observing System. NOAA/National Weather Service, 2001. <http://www.weather.gov/asos/index.html>. Accessed 2010-05-05.

Information about the NWS/FAA/DoD ASOS program.

Piper, M., J.C. Wyngaard, W.H. Snyder, and R.E. Lawson, Jr., *Top-down, bottom-up diffusion experiments in a water convection tank*. J. Atmos. Sci., **52**, 3607-3619, 1995.

A reference for the resistance temperature device calibration data used in the curve fitting section.

The Sunspot Cycle. David H. Hathaway, NASA Marshall Space Flight Center, 2010. <http://solarscience.msfc.nasa.gov/SunspotCycle.shtml>. Accessed 2010-04-14.

Information on sunspots and the historical record of sunspot numbers.

Press, W.H., B.P. Flannery, S.A. Teukolsky and W.T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge, England: Cambridge University Press. 1988.

A handbook of scientific computing in C. Many analysis routines in the IDL distribution are derived from the programs in this text.

Bendat, J.S. and A.G. Piersol. *Random Data; Analysis and Measurement Procedures*. New York, New York: Wiley, 1971.

Priestley, M.B. *Spectral Analysis and Time Series*. San Diego, California: Academic Press, 1981.

These texts deal with signal processing techniques. Priestley's book is very dense.

Gonzales, R.C. and R.E. Woods. *Digital Image Processing*. Reading, Massachusetts: Addison-Wesley, 1992.

Russ, J.C. *The Image Processing Handbook*. Second edition. Boca Raton, Florida: CRC Press, 1995.

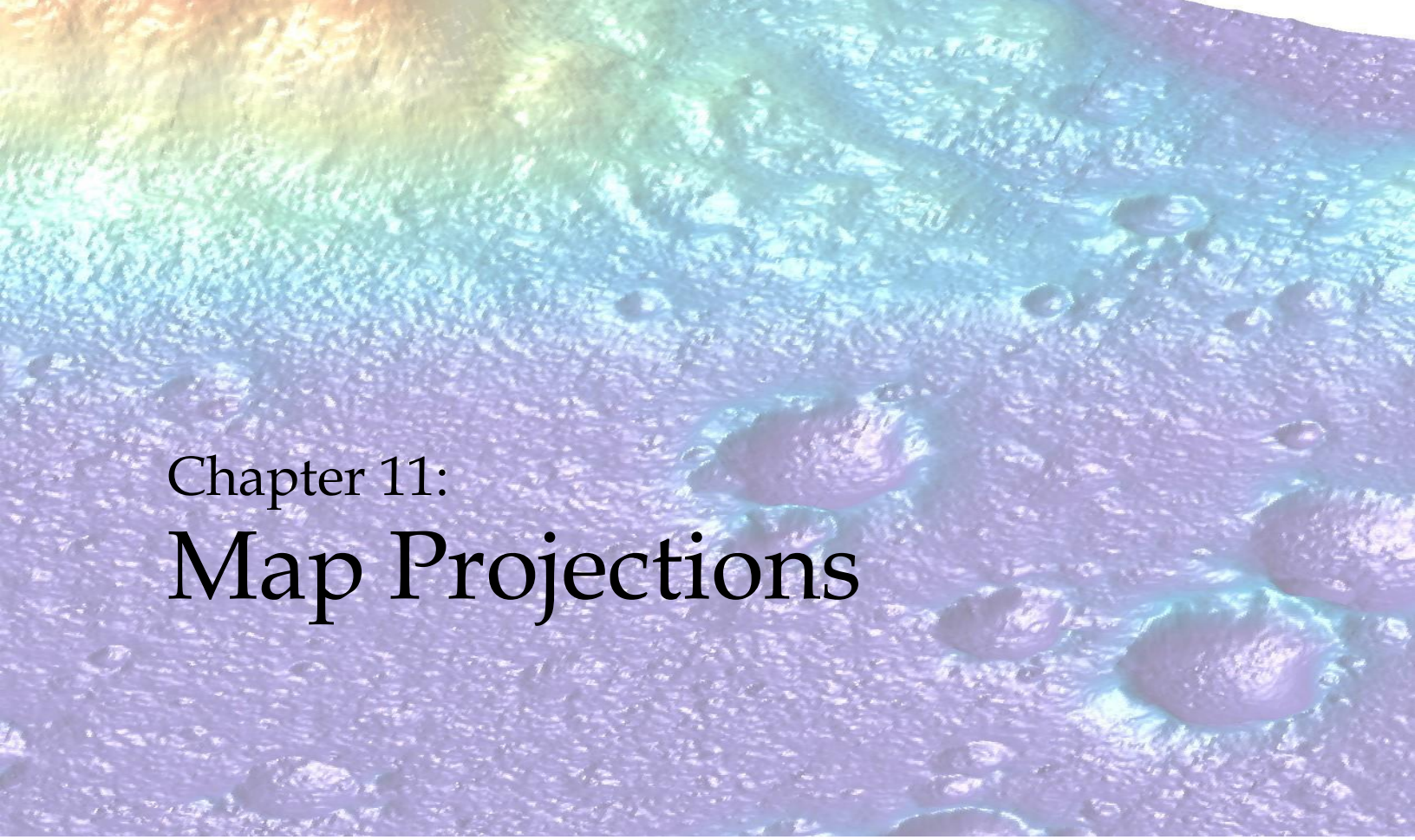
These texts deal with image processing techniques.

Kling, R.L. *Application Development with IDL: Combining analytical methods with widget programming*. Warrenton, Virginia: Ronn Kling Consulting, 1999.

This useful text implements several analysis techniques in IDL, including active contours, simulated annealing and watershed segmentation.

Bowman, K. P. *An Introduction to Programming with IDL.* Boston, Massachusetts: Academic Press, 2006.

Prof. Bowman discusses many of the same topics covered in this chapter.



Chapter 11: Map Projections

This chapter describes how to create map projections and display data in them.

> *Clarke's Third Law: Any sufficiently advanced technology is indistinguishable from magic.*
>
> —Arthur C. Clarke

Map projections 150
Graphics routines 150
A simple map 151
Gridded ASOS temperatures 152
Landsat 7 ETM+ image, georeferenced . . . 154
General map projection routines 156
Exercises 157
References 157

Map projections

A map projection is a transformation from coordinates on a curved surface to coordinates on a plane. Historically, the idea arose from the need to represent locations on the (approximately) spherical surface of the Earth on a piece of paper.

Sizes, shapes and distances can be distorted in the process of transforming from curved to planar coordinates. To address this problem, a variety of projections have been developed, employing various techniques to deform a spherical surface into a planar surface by stretching or cutting the surface, thus minimizing distortion in particular locations.

Map projections are useful for displaying scientific data with a wide geographical extent; e.g., in geophysics, map projections are used to display global sea surface temperatures, polar sea ice distributions and surface weather maps.

Graphics routines

Table 11-1 lists the IDL routines for creating and annotating map projections.

Table 11-1: IDL Graphics routines for map projections.

Name	Description
MAP	Makes a graphics window that uses a map projection for its coordinate system. For projections, IDL uses the USGS GCTP (see References); the IDL Help page for *MAP* includes a list of available projections and their keywords. By default, a longitude-latitude grid is displayed over the projection.
MAPCONTINENTS	Displays continental outlines, rivers and political boundaries (including U.S. states and Canadian provinces) on a map graphic.
MAPGRID	Displays a longitude-latitude grid on a map graphic. Various aspects of the grid are configurable.

MAP is the primary routine for setting up a map projection. *MAPCONTINENTS* and *MAPGRID* can be used to annotate an existing projection. Data can be visualized in a projection with *CONTOUR*, *IMAGE* and *PLOT*. The data can be in Cartesian (projected) or geographic (lat-lon) coordinates. Reprojection, the act of converting data from one map projection to another, is covered in the *Scientific Programming with IDL* course.

To demonstrate how the graphics routines in Table 11-1 work, we'll use them in visualizing two datasets: the gridded ASOS weather data used in Chapter 10, "Analysis" and the Landsat 7 ETM+ scene of Boulder from Chapter 7, "Images." First, though, is a simple example of using the mapping routines.

A simple map

Start by using *MAP* to create a Mercator projection in a new Graphics window:

```
IDL> m = map('Mercator', limit=[-15, 90, 30, 150], $
>      fill_color='dark sea green', title='Southeast Asia')
```

The LIMIT property places bounds on the map projection. Its values are, in order:

- minimum latitude
- minimum longitude
- maximum latitude
- maximum longitude

Note that south latitudes and west longitudes are represented by negative numbers. *MAP* also automatically creates a grid, which we can manipulate later.

Next, use *MAPCONTINENTS* to display political boundaries with black outlines and a light brown fill color:

```
IDL> countries = mapcontinents(/countries)
IDL> countries.fill_color='wheat'
```

The properties of the map grid can be accessed through the map reference. Use the map reference `M` to change the line style and color of the grid. Also push the grid behind the continents (map annotations behave like layers).

```
IDL> m.mapgrid.linestyle = 'dotted'
IDL> m.mapgrid.color = 'white'
IDL> m.mapgrid.order, /send_backward
```

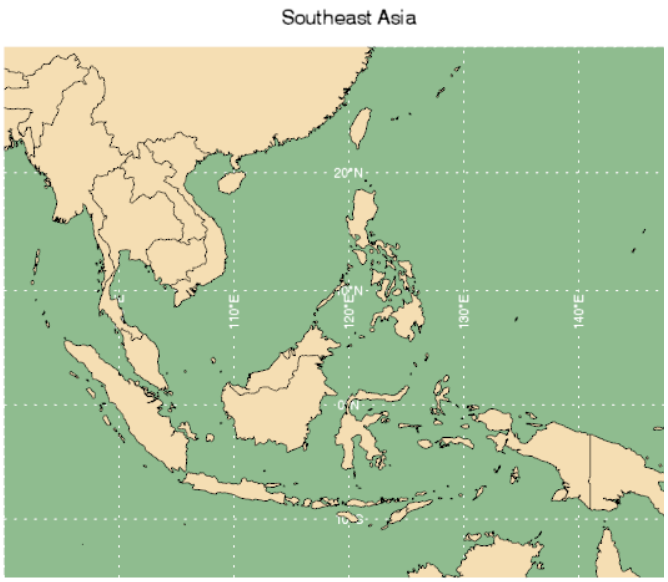


Figure 11-1:

A map of Southeast Asia generated with IDL mapping routines.

Gridded ASOS temperatures

In the section "Gridding irregularly spaced data" on page 131, we used the IDL routine *GRIDDATA* to interpolate Automated Surface Observing System (ASOS) weather data to a regular grid. Gridded surface temperature data from the ASOS file used in this exercise are stored in an IDL SAVE file, **gridded_temperature.sav**, in the IDL coursefiles **data** directory. Restore the contents of this file to your IDL session:

```
IDL> restore, file_which('gridded_temperature.sav')
```

This SAVE file contains IDL variables G_TEMP, G_LON, and G_LAT, corresponding to the gridded temperature data, the latitude and longitude grid for these data, and LON and LAT, the original locations of the ASOS stations. The grid is 21 x 17, with 0.5 degree resolution, starting at 104° W longitude, 34° N latitude, increasing north and east.

```
IDL> help, g_temp, g_lon, g_lat, lon, lat
G_TEMP          DOUBLE    = Array[21, 17]
G_LON           FLOAT     = Array[21]
G_LAT           FLOAT     = Array[17]
LON             FLOAT     = Array[50]
LAT             FLOAT     = Array[50]
```

Use *MAP* to create a new orthographic projection in a Graphics window:

```
IDL> limit = [33.0, -106.0, 43.0, -92.0]
IDL> m1 = map('Orthographic', limit=limit, color='gray', transparency=50, $
>     center_longitude=mean(limit[1:*:2]), $
>     center_latitude=mean(limit[0:*:2]), $
>     title='ASOS-Derived Surface Temperatures')
```

The projection is limited to the geographic area bounded by 33° N latitude, 106° W longitude, 43° N latitude and 92° W longitude. The orthographic projection can optionally set the CENTER_LONGITUDE and CENTER_LATITUDE properties to center the projection. The COLOR and TRANSPARENCY properties apply to the map grid.

Use *MAP_CONTINENTS* to overlay U.S. state boundaries on the projection:

```
IDL> states = mapcontinents(/usa, transparency=25)
```

Now display the gridded temperature data on the map projection with a contour plot:

```
IDL> min_level = 278   ; Kelvin
IDL> interval = 2      ; Kelvin
IDL> n_levels = 11
IDL> c_levels = indgen(n_levels)*interval + min_level
IDL> sfctemp1 = contour(g_temp, g_lon, g_lat, $
>     c_value=c_levels, /overplot, $
>     rgb_table=33, grid_units='degrees')
```

The longitude and latitude vectors for the grid, G_LON and G_LAT, must be passed to *CONTOUR* in addition to the gridded temperature data. The units of the grid vectors are described by the GRID_UNITS property. The contour levels to be displayed are passed into the C_VALUE property. Note the OVERPLOT property must be set so that *CONTOUR* doesn't erase the graphics already present in the window.

Last, display a colorbar along the bottom of the graphic:

```
IDL> sfctempcb = colorbar(target=sfctemp1, title='Temperature (K)', $
>      position=[0.25, 0.1, 0.75, 0.125])
```

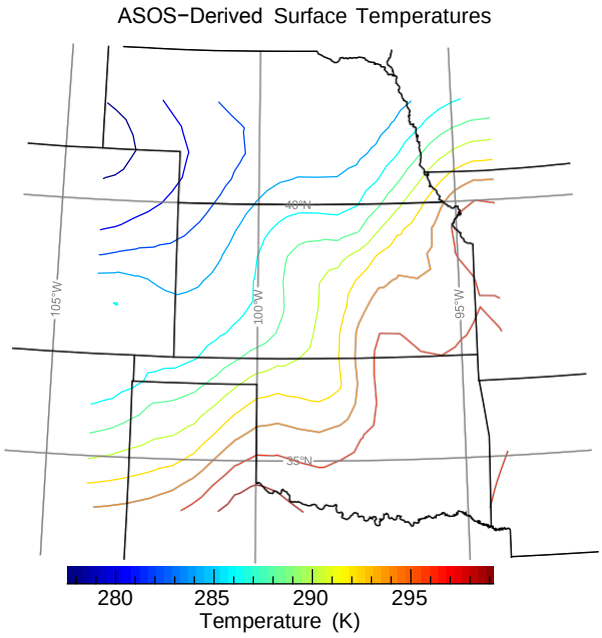


Figure 11-2:

Surface temperatures over the U.S. Great Plains displayed as a contour plot.

A professional meteorologist would probably adjust the plot slightly, packing the isotherms more closely around the cold front stretching from southwest to northeast across Kansas.

Alternately, we can display these gridded surface temperatures as an image in a map projection. As we saw in Chapter 7, "Images", images are fundamentally rectangular. To display an image in a map projection, it needs to be transformed to the coordinate system of the projection. Depending on the geolocation of the data and the type of projection, extreme warping may occur.

Start by recreating the projection used in the previous example (recall you can use the up arrow on your keyboard to retrieve statements typed earlier, or copy-paste from the command history view in the workbench):

```
IDL> m2 = map('Orthographic', limit=limit, color='gray', transparency=50, $
>      center_longitude=mean(limit[1:*:2]), $
>      center_latitude=mean(limit[0:*:2]), $
>      title='ASOS-Derived Surface Temperatures')
IDL> states = mapcontinents(/usa, transparency=25)
```

Next, use *IMAGE* to display the data warped to the map projection:

```
IDL> sfctemp2 = image(g_temp, g_lon, g_lat, $
IDL>     /interpolate, /overplot, $
IDL>     rgb_table=33, grid_units='degrees')
```

Setting the INTERPOLATE property forces bilinear interpolation to be used instead of nearest neighbor sampling when warping the image. Toggle this property off to see the uninterpolated grid cells.

The image is now layered on top of the states. Bring the state boundaries to the front of the graphic with the ORDER method:

```
IDL> states.order, /bring_to_front
```

Last, display a colorbar along the bottom of the graphic:

```
IDL> sfctempcb = colorbar(target=sfctemp2, title='Temperature (K)', $
>      position=[0.25, 0.1, 0.75, 0.125])
```

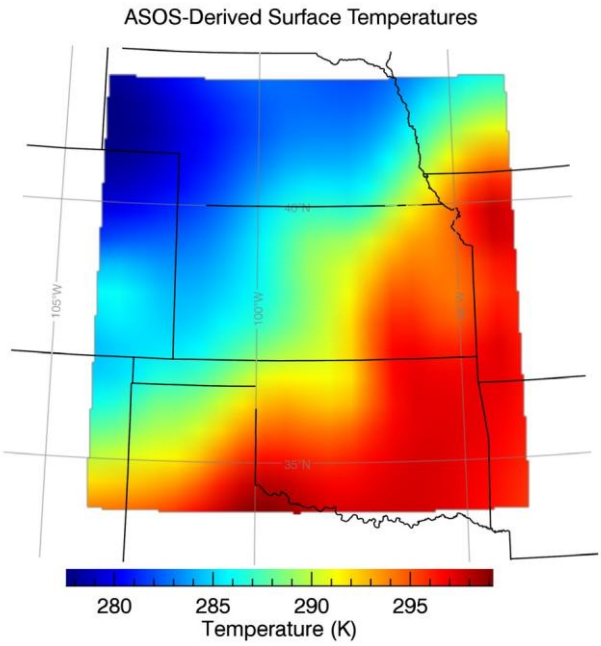


Figure 11-3:

Surface temperatures over U.S. Great Plains displayed as an image.

Compare this result with Figure 11-2. For more detail, see *MAP_ASOS_TEMPERATURES* in the IDL coursefiles **introduction/src** directory.

Landsat 7 ETM+ image, georeferenced

In Chapter 7, "Images", we looked at data from the ENVI file **boulder-ETM.dat**, a Landsat 7 ETM+ scene spatially subset to the area around Boulder. These data are georeferenced, though we didn't use the file's georeferencing information in that example. Let's use it here to display an image from the scene in its native projection.

Start, as before, by reading the image cube from the file:

```
IDL> file = file_which('boulder-ETM.dat')
IDL> n_samples = 575
IDL> n_lines = 700
IDL> n_bands = 6
IDL> cube = read_binary(file, data_dims=[n_samples,n_lines,n_bands])
```

Recall that the parameters N_SAMPLES, N_LINES and N_BANDS are from the ENVI header file **boulder-ETM.hdr**.

From the image cube, form a band (3,2,1) Truecolor composite image (compare this technique with the one used on page 93):

```
IDL> band321 = cube[*,*,[2,1,0]]
```

Now we need map projection information from the ENVI header file. Define variables for the location of the upper-left corner of the image and its pixel size, both in meters:

```
IDL> ul_map = [471375.0, 4439865.0]   ; meters
IDL> pixel_size = [30.0, 30.0]        ; meters
```

Next, set up parameters to be used by *IMAGE* to display the composite image in its native projection:

```
IDL> xsize = n_samples * pixel_size[0]
IDL> ysize = n_lines * pixel_size[1]
IDL> startx = ul_map[0]
IDL> starty = ul_map[1] - ysize
```

XSIZE and YSIZE give the dimensions of the image, while STARTX and STARTY give the position of the lower left corner of the image.

Estimate map projection limits based on knowledge of where Boulder is, geographically speaking:

```
IDL> limit = [39.90, -105.35, 40.15, -105.10]
```

For a more exact technique of determining a projection limit, based on where the corners of the image are located in its native projection, see the program on which this example is based: *MAP_BOULDER_LANDSAT_IMAGE* in the IDL coursefiles **introduction/src** directory.

Set up the image's native projection (Universal Transverse Mercator, zone 13 North, taken from the ENVI header file), then display the image with a two percent linear stretch:

```
IDL> m = map('UTM', zone=13, limit=limit)
IDL> i = image(hist_equal(band321, percent=2), $
>      /order, /overplot, grid_units='meters', $
>      image_dimensions=[xsize,ysize], $
>      image_location=[startx,starty])
```

Note that the GRID_UNITS property indicates the units of the data are in meters. The ORDER property is set to display the image using the remote sensing convention, where the origin is the top left corner.

Try zooming in on the image—see how the grid follows. Compare your result with Figure 11-4 below.

Figure 11-4:

A Landsat 7 ETM+ band (3,2,1) Truecolor composite image of Boulder, displayed in its native UTM projection.

General map projection routines

IDL has general, built-in routines for creating and working with map projections. The routines listed below are what do the work in the Graphics mapping routines listed in Table 11-1.

Table 11-2: General IDL routines for working with map projections.

Name	Description
MAP_PROJ_INIT	Defines a map projection. All projection information is stored locally in the `!MAP` structure variable returned from this function.
MAP_PROJ_FORWARD	Maps spherical/geographic *(lon,lat)* coordinates to projected/Cartesian *(x,y)* coordinates.
MAP_PROJ_INVERSE	The inverse of *MAP_PROJ_FORWARD*.
MAP_PROJ_IMAGE	Warps image data to a projection. The image data can be in geographic or Cartesian coordinates.
MAP_PROJ_INFO	Gives information about a projection variable returned from *MAP_PROJ_INIT* in a human-readable format.

There's more on using these routines (and on using them in the context of reprojection) in the *Scientific Programming with IDL* course.

Exercises

1. Try employing different map projections (other than Mercator and Orthographic) in the chapter examples.
2. Try representing data on a map projection by combining a contour plot and an image.
3. Can you represent a straight line on the Earth's surface on a map projection? Hint: a straight line on the Earth's surface is a great circle.

References

General Cartographic Transformation Package (GCTP). U.S. Geological Survey, 2010. <http://gcmd.nasa.gov/records/USGS-GCTP.html>. Accessed 2010-05-28.

This page at NASA Goddard gives a description of the GCTP. It includes links to obtain the C source code (which is what IDL uses) and a README.

Snyder, J.P. *Map Projections – A Working Manual.* U.S. Geological Survey Professional Paper 1395. Washington, D.C.: U.S. Government Printing Office, 1987.

The canonical USGS map projection reference.

Map Projections. U.S. Geological Survey, 2006. <http://egsc.usgs.gov/isb/pubs/MapProjections/projections.html>. Accessed 2010-05-28.

An annotated list of map projections, with references for each projection. Includes projection properties and suggested uses for each projection. (Author note: I have a full-size poster of this page.)

Glossary

8-bit color A system for displaying color that can only use 256 unique colors simultaneously. Older GPUs only support 8-bit color.

24-bit color A system for displaying color that can employ the entire 256^3 colors in the RGB color system. Most modern GPUs support 24-bit color.

actual parameter The actual variable or expression that gets passed to a subprogram. Cf. **formal parameter**.

algorithm A logical set of steps for accomplishing a computational goal.

anonymous structure A structure that lacks a user-defined name; it can be redefined in an IDL session.

argument See **positional parameter**.

array A variable that contains one or more values of the same type.

array dimension An IDL array may have between one and eight dimensions.

array operation An operation that is performed on some or all of the elements of an array in a single statement. Cf. **scalar operation**.

array subscript An index for the position of a single element in an array.

ASCII The acronym for American Standard Code for Information Interchange. Used in general for describing a commonly used character set for mapping numbers to displayed characters.

associated variable A variable linked to an open file that maps a repeating array structure onto the file's contents.

batch file A file containing IDL statements, with no declaration or terminating END statement.

batch mode A noninteractive mode for executing batch files.

big endian The method of byte ordering where the most significant byte goes first; i.e., the big end in.

binary file The most basic file type on a computer, composed of byte data. Displays as a jumble of random characters in a text editor.

bit The fundamental unit of information on a computer, with a value of 0 or 1.

bug A specific instance of a syntax or logic error.

byte A unit consisting of a series of eight bits.

byte code The executable form of IDL code that results from compiling source code.

byte ordering The scheme a processor uses to order the bytes that make a multiple-byte number (e.g., float, long, complex). See **big endian** and **little endian**.

byte range The integer values between 0 and 255. Eight bits per byte gives 2^8 possible values.

byte scaling An operation that proportions a set of numbers into the byte range.

call stack The current layered sequence of subprogram calls in an IDL session. The call stack is rooted at the main program level.

calling program A program that calls another program.

code The act of programming, or the program itself.

color flashing The disruptive flashing that occurs when IDL gains and loses focus when a private color table is enabled.

color mode In Direct Graphics, the method by which IDL specifies colors in the RGB color system.

color table A lookup table whose elements are colors defined in the RGB color system. Synonyms include color lookup table, LUT and color palette.

command line The IDL component where statements can be entered.

command prompt See **command line**.

comment Explanatory text written alongside code. In IDL, lines prefaced with a semicolon are interpreted as comments.

compile The computational process of converting source code into byte code in an IDL session.

concatenation The act of joining two variables into one. Used often with arrays and strings.

conditional A branching control statement.

control statement A statement used to control the execution of other statements in a program.

data type See **variable type**.

declaration A statement that defines a subprogram's type, name and parameters.

decomposed color The color mode in which IDL uses long integers to directly specify colors in the RGB color system.

device-copy technique A method of copying data from the graphics card to an IDL Direct Graphics window.

Direct Graphics The original graphics system in IDL. It uses a non-persistent method for displaying graphics.

display device In Direct Graphics, the device that supplies the windowing system; e.g., Windows or X.

dynamic typing A property of IDL whereby a variable's type can change as the result of an operation.

Eclipse An open-source software development environment. The IDL workbench is built on Eclipse.

error A code execution or syntax problem that prevents further execution. Cf. **warning**.

endianness See **byte ordering**.

executive command A type of statement used in compiling, executing and debugging programs. Executive commands can only be used at the command line.

expression A computational statement that returns a value.

file path A string that gives the location of a file on a file system.

file system A system for organizing files on an operating system.

floating point The machine representation of a decimal number.

formal parameter The parameter listed in the definition of a subprogram, a placeholder for an actual parameter.

function A self-contained program that begins with a function declaration and ends with an END statement.

global variable A variable whose scope is unlimited in an IDL session.

GPU Acronym for graphics processing unit, the component of a computer that controls what is displayed on a monitor. Also graphics card.

Graphics The graphics system introduced in IDL 8.0. It's based on Object Graphics, but is scriptable like Direct Graphics.

graphics device In Direct Graphics, a peripheral where graphics can be directed and displayed.

graphics system A framework for displaying graphics, such as plots and imagery. IDL has two graphics systems: Direct Graphics and Object Graphics.

graphics window An area on the display device where graphics can be viewed.

hash A container variable that holds values of any data type or organization; the values are unordered and accessed with a key.

IDE Integrated Development Environment.

IDL Interactive Data Language.

IDLDE IDL Development Environment. See **IDL workbench**.

IDL Runtime A reduced version of IDL that can be used to execute compiled IDL programs. Requires special licensing.

IDL Virtual Machine A version of IDL that uses IDL Runtime but without a license. There is a click-through splash screen, however.

IDL workbench The Eclipse-based graphical code development environment for IDL introduced in IDL 7.0, replacing the IDLDE.

iTool An interactive IDL application used to analyze and visualize data, then produce publication-quality output.

image A visual representation of an object discretized into a rectangular array whose elements (pixels) represent the colors or intensities of the representation.

indexed color The color mode in which IDL uses a color table to address subsets of the RGB color system. (Emulation of an 8-bit system.)

interpreter The part of IDL that converts source code to byte code.

JimP An omniscient IDL programmer.

key A string or number used to identify a value in a hash.

keyword parameter A type of parameter that is preceded by a name (the "keyword") and an equal sign.

list A container variable that holds values of any data type or organization; the values are ordered and accessed with an index.

little endian The method of byte ordering where the least significant byte goes first; i.e., the little end in.

local variable A variable whose scope is limited to the program or subprogram in which it was created.

logical unit number A number associated with an open file on the file system.

long integer An integer type that occupies 32 bits in memory.

loop A control statement that can be executed multiple times.

LUT An acronym for color lookup table. Also CLUT.

main IDL directory The directory in which IDL is installed. It is the root of the IDL distribution.

main program A program that has no declaration but is terminated with an END statement.

main program level The level in the call stack where a main program resides. This is the default start point in an IDL session.

method A program built into an object. Used often with Graphics; e.g., the save method of a plot.

Metafile format A file format used by applications in the Windows operating system.

named structure A structure that has a name. Its definition is fixed within an IDL session.

null variable An undefined variable that can be used in assignment and concatenation.

Object Graphics IDL's three-dimensional, device-independent graphics system based OpenGL. Graphical displays created with Object Graphics are persistent in memory.

operating system The software responsible for the basic operations of a computer.

operator A tool in a programming language that combines variables and expressions to make new variables and expressions.

parameter A variable or expression passed to a subprogram.

pass-by-reference A method of parameter passing where a reference to the parameter is passed to a subprogram. The subprogram and the calling program operate on the same data.

pass-by-value A method of parameter passing where a copy of the parameter is passed to a subprogram. The subprogram and the calling program operate on separate data.

path See **search path**.

perspective A grouping of views in the IDL workbench. The workbench comes with two perspectives, "IDL" and "Debug." You can make your own perspectives by rearraging views.

pixel Short for "picture element", the fundamental unit of an image.

positional parameter A type of parameter whose order in the actual parameter list must match that in the formal parameter list in the definition of a subprogram.

PostScript An interpreted language used in printing, developed by Adobe Systems.

private color table A color table used only by IDL on an 8-bit system. IDL "steals" colors from other applications when it has focus, but returns them when focus is lost.

procedure A self-contained program that begins with a procedure declaration and ends with an END statement.

program An organized set of statements meant to be compiled and executed to perform a task. In IDL, programs must be terminated with an END statement.

project A project is a directory that contains source code and other files.

property The data built in to an object. Used often in Graphics; e.g., the color property of a plot.

PseudoColor An industry name for 8-bit color.

reference (Graphics) The variable returned from a call to a Graphics function. A reference can be used to control the Graphic after it has been created.

regular expression A pattern that describes a string or a set of strings.

RGB color system A system for specifying colors based on combining, additively, intensities of red, green and blue light.

RGB triple A three-element array that gives the red, green and blue components of a color.

routine A synonym for **program**.

SAVE file A binary file type used for storing IDL byte code.

scalar A single value, as opposed to an array or a vector.

scalar operation An operation that is performed on a single value.

scope See **variable scope**.

search path A set of directory paths to be searched by IDL when an unidentified routine needs to be compiled.

short integer An integer type that occupies 16 bits in memory.

signed integer An integer type that allows both positive and negative values.

source code Code written by a programmer. In IDL, like other languages, source code is plain text.

statement A command or instruction in a programming language.

Stern, David Creator of IDL and founder of RSI. Dave has a Stern factor of 1.

Stern factor A measure of one's IDL geekiness. Defined on a unit scale.

string A variable type composed of character, as opposed to numeric, information.

string processing The techniques for manipulating string variables.

structure A composite variable that can store information of differing type and organization in a single entity.

structure tag or field An element of a structure.

subprogram A procedure or function.

subroutine A synonym for **subprogram**.

subscripting The operation whereby values of an array are accessed.

syntax The rules defining the proper ordering of parameters, expressions and operators in a statement to produce code that can be compiled.

system routine An IDL program that is written in ANSI C. Its source code is not available to a user.

system variable A special class of predefined global variables. System variables always begin with an exclamation point.

TEX A typesetting language developed by Donald Knuth.

text file A file composed of character data that is human-readable. Cf. **binary file**.

TrueColor An industry name for 24-bit color.

type See **variable type**.

type code A value associated with an IDL variable type.

undefined variable A variable that doesn't exist; It has no type or value.

unformatted file A synonym for a **binary file**.

unsigned integer An integer type that allows only positive values.

user routine Any IDL program that is written in the IDL language.

variable A named repository for storing information.

variable scope The range in the call stack over which a variable is defined.

variable type The sort of information that can be stored in a particular variable.

vector A one-dimensional array.

view A component of the IDL workbench. The Command Line, Console and Editor are examples of views.

warning An execution-time problem that doesn't halt execution.

workbench See **IDL workbench**.

working directory The directory on the file system that IDL is working in—file paths are relative to this directory.

workspace A workspace is a directory that contains project directories, metadata about the contained projects, and information about the state of the IDL workbench.

Index

Symbols

^ exponentiation 72
; comment character 2
: array indexing operator 49
!color system variable 44
!dir system variable 18, 44
!dpi system variable 44
!dtor system variable 44
!null system variable 32, 44
!path system variable 71
!pi system variable 44
!values system variable 44
!version system variable 44
? : ternary operator 73
. period operator 55, 72
' string literal delimiter 56
" string literal delimiter 56
( ) grouping operator 72
[ ] for subscripting and concatenating arrays 72
{} for structure creation 55
* array indexing operator 50
    in multi-dimensional arrays 51
* pointer dereference operator 72
&& logical 'and' operator 73
# and ## matrix multiplication operators 75
+ string concatenation operator 56
-- decrement operator 72
++ increment operator 72
< minimum operator 73
= assignment operator 73
= keyword data delimiter 69
> maximum operator 73
-> method invocation operator 72
|| logical 'or' operator 73
~ logical 'not' operator 73
$ continuation character 2, 11

Numerics

64-bit integer data type 45
    arrays of 49

A

ABS function 132, 139
ALOG function 137
AND operator 73
ARRAY_INDICES function 51
arrays 48–52
    concatenation 47
    creation function 48
    creation functions 49
    generator functions 49
    index generator functions 48
    internal storage order 51

multidimensional 51
negative array indexing 50
non-sequential subscripting 50
of type 64-bit integer 49
of type byte 49
of type complex 49
of type double 49
of type double complex 49
of type float 49
of type integer 49
of type long integer 49
of type object 49
of type pointer 49
of type string 49
of type unsigned 64-bit integer 49
of type unsigned integer 49
of type unsigned long integer 49
single-index subscripting 51
subscripting 9, 49, 51
subscripting with strides 50
using square brackets 48, 49
vector 10
ARROW function 119
ASCII files 98
explicit format 111
free format 110
READ_ASCII function 105
reading 105, 109–112
text editor 98
writing 112
ASCII_TEMPLATE function 105
ASPECT_Z property 120, 123
astrolib, see IDL Astronomy User's Library

B

bar plots/charts 36
BARPLOT function **30**, 36
INDEX property 36
NBARS property 36
beauty 86
BEGIN statement 75
behavior keyword 69
BIN_DATE function 71
binary files 98
byte order or endianness 108
READ_BINARY function 105
reading 105, 113
writing 113
binary image mask 144
BINARY_TEMPLATE function 105
BINDGEN function **49**
BIT_DEPTH property 124
BREAK statement 77, 81
BYTARR function 113, **49**
byte data type 45
arrays of 49
BYTE function 45
byte order in binary files 108
BYTEORDER procedure 109
SWAP_ENDIAN function 109
SWAP_ENDIAN_INPLACE procedure 109
bytecode 64
BYTEORDER procedure 109
BYTSCL function **89**

C

calling mechanism 70–71
CANNY function 143
CASE statement 77
CD procedure 24, **98**
CURRENT keyword 69
CINDGEN function **49**
CLOSE procedure **107**
code execution model 64
COLOR property 11
using RGB triple 120
COLORBAR function 15, 119
comment character ; 2
COMPILE_OPT statement 68
DEFINT32 option 68
LOGICAL_PREDICATE option 79
STRICTARR option 68
complex data type 45
arrays of 49
COMPLEX function 45
COMPLEXARR function **49**
compound operators 74
continuation character $ 2
CONTINUE statement 81
CONTOUR function 12, **116**, 118
C_VALUE property 13, 152
FILL property 13
GRID_UNITS property 152
N_LEVELS property 12
PLANAR property 13, 124
RGB_TABLE property 12
control statements **75**, 75–79
BEGIN-END block 75
CASE statement 77
FOR loop 79
IF statement 76
REPEAT loop 80
SWITCH statement 77
WHILE loop 80
coordinates

for Earth 53
normalized 137
COPY_LUN procedure 99
course files **3**, 26
downloading 4
IDL workbench project 23
installing 4
course manual
downloading course files 4
how it is organized 2
how to use 2
style conventions 2
course programs
ANALYZE_SUNSPOT_SERIES 140
DISPLAY_PRECIP 121
DISPLAY_TOPO 126
GET_COURSEFILES_DIR function 111
PREPARE_TOPO 122
RANGE function 123
READ_ASOS function 131
WIND_PROFILE_MODEL function 137
Coyote library 104
CREATE_STRUCT function 55
cubic splines, see interpolation
current directory 71, 112, **24**
CURRENT property 41, 91
curve fitting 134–138
chi-square statistic 134, 137
fit coefficients 134
linear least-squares 134
logsquare model 137
nonlinear least-squares fit 136

D

data
24-hr precipitation totals 116
ASOS weather observations 131, 152
daily Boulder temperature 33
GTOPO30 digital elevations 121
International Space Station position 42
Maroon Bells DEM 10
NCAR Mesa Lab weather station 101
ozone concentration 126
propeller-vane anemometer 136
reflectance spectra 30
resistance temperature device (RTD) 134
sunspot numbers 38, 138
USGS National Land Cover Data 125
DATA property 32, 119
data types 45
range 45
scalar creation notation 45
sizes 45
type code 45
DBLARR function **49**
DCINDGEN function **49**
DCOMPLEX function 45
DCOMPLEXARR function **49**
dereferencing pointers 58
DIALOG_PICKFILE function 87, 100, **99**
DIMENSIONS property 40, 123
DINDGEN function **49**
directories
search path 71
directory
Default project 23
main IDL 18
project 23
working or current 24
DIST function 144
double complex data type 45
arrays of 49
double data type 45
arrays of 49
DOUBLE function 45

E

ELSE reserved word 76
END statement 66, 67, 75
ENVI 30, 92, 154
file format 92
environment variables
$IDL_DIR 18
EOF function 99
EQ operator 73
error bar plots 37
ERRORPLOT function 37, 136, **30**
executive commands 64, 65
Exelis VIS
Codebank 104
contacting ii
Documentation Group 140
Educational Services Group ii
IDL and ENVI classes ii
Professional Services Group ii
website ii, 4, 104
EXPAND_PATH function 26

F

FFT function 138, 139, 144
power spectrum 145
see also signal processing
file extension 100
file manipulation routines 98–100
FILE_* routines, table of 98

FILE_BASENAME function 100, **98**
FILE_CHMOD procedure 98
FILE_COPY procedure 98
FILE_DELETE procedure 98
FILE_DIRNAME function 100, **98**
FILE_EXPAND_PATH function 100, **98**
FILE_INFO function 98
FILE_LINES function 100, **98**
FILE_LINK procedure 99
FILE_MKDIR procedure 99
FILE_MOVE procedure 99
FILE_POLL_INPUT function 99
FILE_READLINK function 99
FILE_SAME function 99
FILE_SEARCH function 100, **99**
FILE_TEST function **99**
FILE_WHICH function 10, 30, 100, 110, **99**
FILEPATH function 9, 47, **98**, 107, 113, 140
files
  ASCII, see ASCII files
  binary, see binary files
  closing 107
  ENVI file 92
  ENVI files 154
  FILE_* routines 98
  FITS format 87, **104**
  GZIP compressed 108
  logical unit number (LUN) 107
  opening 107
  process for input/output with low-level routines 106
  SAVE files 101
  scientific data formats 102
  standard image formats 102
  XDR format 108
FILL_BACKGROUND property 32, 39
FILL_COLOR property 32, 39
FILL_LEVEL property 39
FINDGEN function 9, 48, 117, **49**
FIX function 45, 111
float data type 45
  arrays of 49
FLOAT function 45
floating-point precision 47
FLTARR function 8, 48, **49**
FLUSH procedure 99
FONT_NAME property 118
FONT_SIZE property 118
FOR loop 9, 32, 52, 79, 113
FOREACH loop 31, 143, **81**
FORMAT keyword 46, 111
FORPRINT procedure 113
FREE_LUN procedure 108, 111, **107**
FSTAT function 99
function **67**
  compiling 67
  declaration statement 67
  executing 67
function declaration statement 67
FUNCTION reserved word 67

G

GE operator 73
GET_COURSEFILES_DIR function 100
GET_LUN procedure **108**
GET_SCREEN_SIZE function 123
GETENV function **99**
Gouraud shading 124
Graphics
  method 11
  properties 11
  reference 11
GRID_INPUT procedure 132
  EPSILON keyword 132
  EXCLUDE keyword 132
GRID_UNITS property 155
GRIDDATA function 133
  DEGREES keyword 133
  SPHERE keyword 133
GRIDDATA procedure 152
gridding, see interpolation
GT operator 35, 73

H

HANNING function 100
HASH function 53, 143
hashes **53**
  accessing values 54
  adding and removing values 54
  creating 53
  remove key 54
  test for key 54
HELP procedure **8**, 53, 58, 60
  FILES keyword 107
  STRUCTURES keyword 55
HIDDEN_LINES property 123
HIDE property 135
HIST_EQUAL function 141
HISTOGRAM function 88, 141, **39**

I

IDL
  batch mode 65
  code execution model 64
  command-line version 4

coursefiles, see course files 23
Help system, see IDL Help system
Math & Stats Module 130
program types 64
search path 22, 26, 67
style conventions 2
supported platforms 4
workbench 4, 19
IDL Astronomy User's Library 87, **104**
IDL Help system 27–28
browser 28
hover help 22
starting 27
techniques for using 28
workbench quick search 28
IDL version 44
IDL Workbench
projects 4
syntax coloring 20
IDL workbench 4, 19
command history 20
console 20
content assist 20
editor 20
hover help 22
keyboard shortcuts 21
outline view 19
preferences 24
project explorer view 19, 23
projects 22
Default project 23
IDL_coursefiles project 23
statement recall 21
syntax coloring 22
variables view 20
view 19
workspace 23
IDLDE, see IDL workbench
IDLffDICOM class 102
IDLffDXF class 102
IDLffJPEG2000 class 102
IDLffMJPEG2000 class 102
IDLffMrSID class 103
IDLffShape class 102
IDLffXMLDOM class 103
IDLffXMLSAX class 103
IDLgrMPEG class 102
IF statement 52, **76**
IMAGE function 14, **86**, 87, 103, 143, 144
INTERPOLATE property 154
ORDER property 144
image processing 140–146
contrast enhancement 94, 142
histogram equalization 140
median filtering 89
sharpening 142
unsharp masking 142
images **86**, 86–95 binary
image **86** bitplanes or
channels 90 byte scaling
89
color-mapped or color indexed **86**
composite image 93
contrast enhancement 89, 94, 140, 142
draw order 93
histogram 88, 141
image cube 93
in a map projection 153
in map projections 155
intensity-mapped 87, **86**
interleaving 93, 94, **91**
multi- or hyperspectral 93, **86**
negative 146
ORDER keyword 93
origin and draw order 90
positioning 91
processing, see image processing
QUERY_IMAGE function 104
READ_IMAGE function 104
reading and writing standard formats 102
Truecolor 90, **86**
two percent linear stretch 94
WRITE_IMAGE function 104
important number 58
index generator function 48
INDGEN function 124, **49**
input keyword 69
INTARR function **49**
integer overflow 46
integers 45
64-bit 45
arrays of 49
byte 45
long 45
short 45
unsigned 45
interpolation 130–133
cubic splines 130
GRIDDATA function 133
gridding 133
SPLINE function 130
IOPEN procedure **104**

K

keyboard shortcuts 21
keyword parameters 69
"slash" notation 69

behavior keyword 69
input keyword 69
output keyword 69

L

L64INDGEN function **49**
LAPLACIAN function 143
LAYOUT property 91, 143
LE operator 73
LEGEND function 137, **32**
LINDGEN function **49**
line plots
see PLOT function
LINESTYLE property 11
LINFIT function **134**
CHISQ keyword 134
PROB keyword 134
SIGMA keyword 134
LIST function 53, 91
lists **53**
adding elements 53
creating 53
empty 53
joining 53
removing elements 53
subscripting 53
LOG property 139
logical unit number (LUN) 107
LON64ARR function **49**
LONARR function **49**
LONG function 45
long integer data type 45
arrays of 49
LONG64 function 45
loop statements 79
lowpass filter 144
LT operator 73

M

MACHAR function 45
magnitude spectrum 139
main IDL directory 18, 140
main program **66**
compiling 66
executing 66
MAJOR property 123
MAP function **150**, 152
LIMIT property 151
map projections 150–157
displaying images 153
with CONTOUR function 152
MAP_CONTINENTS procedure 152
MAP_PROJ_FORWARD function **156**
MAP_PROJ_IMAGE function **156**
MAP_PROJ_INFO function **156**
MAP_PROJ_INIT function **156**
MAP_PROJ_INVERSE function **156**
MAPCONTINENTS function **150**
MAPGRID function **150**
MAX function 12
maximum subscript parameter 139
MAX_VALUE property 141
MEDIAN function 89
MIN function 12, 135
MINOR property 119
MOD operator 72
multidimensional arrays 51

N

N_ELEMENTS function 52, 122
N_LEVELS property 119
NAME property 32, 137
NE operator 73
negative array indexing 50
netCDF 116
normalized coordinate system 137
NOT operator 73
null variable 32, 45, **47**

O

OBJ_NEW function 45
OBJARR function **49**
Object Graphics system 59
objects 45, **59**
arrays of 49
creating 59
destroying 59
methods 59
properties 59
OF reserved word 77
online help, see IDL Help system
OPENR procedure **107**, 108, 110
COMPRESS keyword 108
GET_LUN keyword 108, 110
OPENU procedure 107
OPENW procedure 112, **107**
operators 72–75
array operations 74
assignment = 73
bitwise
NOT, AND, OR, XOR 73
compound 74
exponentiation ^ 72
grouping ( ) 72

increment ++ and decrement -- 72
logical
&&, ||, ~ 73
matrix multiplication # and ## 75
method invocation -> 72
minimum < and maximum > 73
modulo 72
pointer dereference * 72
precedence 72
relational 73
ternary ? : 73
OR operator 73
ORDER method 154
Order method 151
output keyword 69
OVERPLOT property 32, 34, 120, 152

P

parameters 69–70
pass by reference 70
pass by value 70
passing 70
positional vs keyword 69
PATH_CACHE procedure 26
PATH_SEP function 26, 100, **99**
path, see search path
pi 44
PLANAR property 124
PLOT function 10, **30**, 135
format parameter 39
HISTOGRAM property 40
LINESTYLE property 34
output to a file 32
POSITION property 135
properties 11
scatter plot 34
SYMBOL property 34
TeX-like character formatting 31
PLOT3D function 42, **30**
POINT_LUN procedure 99
pointers 45, **58**
arrays of 49
dereference operator 58, 72
dereferencing 58
PTR_NEW function 58
POLARPLOT function 42, **30**
POLYGON function 34, **32**
POLYLINE function 139, **32**
POPD procedure 99
POSITION property 40, 119, 137
positional parameters 69
positioning graphics 40
LAYOUT property 91
POSITION property 119
PostScript
encapsulated 41
power spectrum 139
PREF_GET function 26
PREF_SET procedure 26
preferences 24, 26
PREWITT function 143
PRINT procedure **8**, 53, 54, 55, 58, 59
FORMAT keyword 46
PRINTD procedure 99
PRINTF procedure 112, **109**
PRO reserved word 66
procedure **66**
compiling 67
declaration statement 66
executing 67
procedure declaration statement 66
programs 64–68
bytecode 64
calling mechanism 70
code execution model 64
COMPILE_OPT statement 68
function 67
helpful tips 82
list of types 64
main 66
parameter passing 70
procedure 66
source code 64
structured programming 68
projects, see IDL workbench
PTR_NEW function 45, **58**
PTRARR function **49**
PUSHD procedure 99

Q

QUERY_BMP function 102
QUERY_CSV function 102
QUERY_DICOM function 102
QUERY_GIF function 102
QUERY_IMAGE function 47, **104**
QUERY_JPEG function 102
QUERY_JPEG2000 function 102
QUERY_MRSID function 103
QUERY_PICT function 102
QUERY_PNG function 102
QUERY_PPM function 102
QUERY_SRF function 103
QUERY_TIFF function 103
QUERY_WAV function 103

R

RANDOMN function 52, 130
RANGE property 31, 135
READ_ASCII function 105, **105**
READ_BINARY function 9, 93, 102, **105**, 106, 140
    DATA_DIMS keyword 106
    DATA_TYPE keyword 106
READ_BMP function 102
READ_CSV function 102
READ_DICOM function 102, 144
READ_GIF function 102
READ_IMAGE function 77, 125, **104**
READ_JPEG procedure 90, 102, 142
READ_JPEG2000 function 102
READ_MRSID function 103
READ_PICT procedure 102
READ_PNG procedure 102
READ_PPM procedure 102
READ_SRF procedure 103
READ_SYLK function 103
READ_TIFF function 103
READ_WAV function 103
READ_WAVE procedure 103
READ_XWD function 103
READCOL procedure 30, 104, 121
READF procedure **109**, 111, 112
    FORMAT keyword 111
READU procedure 102, **113**
REBIN function 120
REFORM function 91, 111, 122
regression, see curve fitting
REPEAT loop 80
REPLICATE function 56
reprojection 150
RESOLUTION property 124
RESTORE procedure 10, 33, 152, **101**
RETURN procedure 67
REVERSE function 34
RGB_INDICES property 13, 119
RGB_TABLE property 119
ROBERTS function 143
Rotate method 123, **124**
ROUTINE_FILEPATH function 99
RSI, see Exelis VIS

S

SAVE files 10, 33, 35, 101
Save method 32, 92
SAVE procedure **101**
scalars
    creation 45
45
Scale method 123, **124**
scatter plot 34
scientific data file formats 102
search path 4, 22, **26**, 67, 71
    EXPAND_PATH function 26
    PATH_SEP function 26
    preference 26
SETENV procedure 99
SETPROPERTY method 11
SHADING property 124
signal 138
signal processing 138–140
    fast Fourier transform (FFT) 138
    fundamental frequency 139
    magnitude spectrum 139
    Nyquist frequency 139
    period 139
    power spectrum 139
    wavelet transform (WT) 138
SIN function 15
SINDGEN function **49**
SIZE function 45, 117
SKIP_LUN procedure 99
SMOOTH function 117
SOBEL function 14, 143
SOCKET procedure 99
SORT function 112
source code 64
SPLINE function 130
    example 131
splines, see interpolation
statement recall 21
statements 64
Stern, David 104
STRARR function 110, **49**
STRCMP function 57
STRCOMPRESS function 57
STREGEX function 57
strides 50
STRING function 45, 56, 57, **57**
    FORMAT keyword 57
strings 45, **56**, 56–58
    arrays of 49, 118
    concatenating 56
    converting to 45
    creation 45
    null string 110
    STRING function 57
    type code 45
STRJOIN function 57
STRLEN function 57, **57**
STRLOWCASE function 57
STRMATCH function 57

STRMID function 58, 100, **57**
STRPOS function 58, 100, **57**
STRPUT function 57
STRSPLIT function 57
STRTRIM function 45, 139, **57**
structured programming 68
structures **54**, 54–56
  accessing fields 55
  anonymous vs. named 54
  arrays 56
  CREATE_STRUCT function 55
  creation 55
  fields 55
STRUPCASE function 57
STYLE property 120
subprograms 68
subscripting arrays 9, 49, 51
  single-index notation 51
SURFACE function 11, **116**, 117
  HIDDEN_LINES property 123
  Rotate method 11
  STYLE property 11, 120, 123
  texture mapping 125
  TEXTURE_IMAGE property 125
  TEXTURE_INTERP property 125
SVDFIT function **136**, 137
  CHISQ keyword 137
  FUNCTION_NAME keyword 137
  SIGMA keyword 137
SWAP_ENDIAN function 109
SWAP_ENDIAN_INPLACE procedure 109
SWITCH statement 77
SYM_FILLED property 37
SYMBOL property 37
system variables 44
SYSTIME function 66, 67

T

TARGET property 32
TEMPORARY function 74
text file, see ASCII files
TEXT function 15, 118, 119, 139
  ALIGNMENT property 118
texture mapping 125
TEXTURE_IMAGE property 125
TEXTURE_INTERP property 125
the answer 58
THEN reserved word 76
THICK property 11, 31
thresholding operation 144
TICKLEN property 119
TICKNAME property 36
TICKVALUES property 36, 119, 123
time series analysis, see signal processing
TITLE property 11
TOTAL function 35
Translate method 124
TRANSPARENCY property 32
TRANSPOSE function 94
TRUNCATE_LUN procedure 99
truth, definition of 79
  see also beauty
type codes 45
type conversion functions 45

U

UINDGEN function **49**
UINT function 45
UINTARR function **49**
UL64INDGEN function **49**
ULINDGEN function **49**
ULON64ARR function **49**
ULONARR function **49**
ulong data type 45
ULONG function 45
ulong64 data type 49
ULONG64 function 45
undefined data type 45, 49
undefined variable 45
UNIQ function 122
UNSHARP_MASK function 142
unsigned 64-bit integer data type 45
  arrays of 49
unsigned integer data type 45, 49
unsigned long integer data type
  arrays of 49
UNTIL reserved word 80
USGS GCTP 150

V

variables 44–60
  arrays 48
  data types 45
  dynamic typing 46
  floating-point precision 47
  hashes 53
  integer behavior 46
  integer overflow 46
  lists 53
  null variable 47
  objects 59
  organization 44
  pointers 58
  strings 56
  structures 54

system variables 44
type promotion 46
types 44
variable names 44
vector variable 10, 48
views, see IDL workbench

W

WHERE function 39, **52**, 87, 112
tolerance example 132
WHILE loop 80
WINDOW function 91
workbench, see IDL workbench
working directory 24
WRITE_BMP procedure 102
WRITE_CSV procedure 102
WRITE_GIF procedure 102
WRITE_IMAGE function 104
WRITE_JPEG procedure 102, 104
WRITE_JPEG2000 procedure 102
WRITE_NRIF procedure 102
WRITE_PICT procedure 102
WRITE_PNG procedure 102, 103
WRITE_PPM procedure 102
WRITE_SRF procedure 103
WRITE_SYLK procedure 103
WRITE_TIFF procedure 103
WRITE_WAV procedure 103
WRITE_WAVE procedure 103
WRITEU procedure 102, 114, **113**

X

XDR format files 108
XOR operator 73
31, 36, 119, 123, 135, 139

Y

YMAJOR property 41
YRANGE property 40

Z

zero-length array 47
ZVALUE property 120