Extending ENVI

An introduction to using the ENVI API in IDL

Revised for Print December 4, 2012

Produced by Outreach Services

Exelis Visual Information Solutions

4990 Pearl East Circle

Boulder, CO 80301

303-786-9900

Extending ENVI with IDL: Table of Contents

Chapter 1 About this Manual

This is the manual for *Extending ENVI with IDL,* a typically four-day course for remote sensing scientists, engineers, and developers who wish to incorporate their own algorithms and workflows into ENVI. ENVI, developed by Exelis Visual Information Solutions of Boulder, Colorado, is a powerful remote sensing software package with a tremendously flexible and capable Application Programming Interface (API).

This course provides an overview of the programming constructs available in IDL, the language in which ENVI is written, as well as the tools necessary for a user to extend ENVI with IDL, including ENVI library routines, custom file readers and writers, batch programs, application control, and extensions. Students should be familiar with ENVI and its functionality and have an understanding of very basic programming principles.

Although this is an IDL course, no prior specific experience in IDL is required.

Programming Style

Unlike languages such as C/C++, Java, or Python, there is no set style standard for IDL programming. This is a blessing and a curse: it allows users to quickly create programs, but others may find the program difficult to read. In order to increase readability and clarity of your IDL code, we recommend adopting a consistent style.

The following table describes the style standard 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**, **extending\src**
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
Spacing	Indentation and empty lines are used liberally to set off related sections of code and maintain code legibility in the manual. IDL is white-space agnostic, so white space will not impact code functionality (with very few exceptions).
Comments	Comments are marked with a semicolon (;).
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 is allowed. It is not necessary to type in line continuation characters - this is done to facilitate legibility in the manual.

The examples in this manual require that the user enter code into the workbench. When necessary, instructions and visual clues for when and where to enter code into a program are provided. For example, the following statement is meant to be entered into a routine:

```
  ; General error handler
  catch, envi_error
  if envi_error ne 0 then begin
    catch, /cancel
    if obj_valid(e) then $
      e.ReportError, 'ERROR: ' + !error_state.msg
    message, /reset
    return
  endif
```

The colored text indicates that this code is new and should be typed into an open IDL workbench editor menu in the position indicated. The presence of the drop-shadow border indicates that the code above should be typed into an editor window. Existing code in a file is specified instead by normal weighting and a gray color, rather than syntax highlighting. The following example indicates a one line adjustment to previous code:

```
  ; Get the input raster from the user.
  e = envi(/current)
  my_ui = e.ui
  my_raster = my_ui.SelectInputData(/raster)

  ; Make sure the input raster is acceptable
  if my_raster.nbands lt 2 then $
    message, "Image must be  multispectral!"
```

Code prefixed with ENVI> or IDL> is meant to be typed at the IDL command line in the workbench console. Code prefixed with ENVI> should be typed in when an ENVI session is active, IDL> when one is not active.

```
IDL> print, 'ok'
```

```
ENVI> bldr = e.OpenRaster(file_which('qb_boulder_pan'))
```

Occasionally code will be displayed as part of an explanation with no bold face or command prompt. This is meant to be used for illustration purposes only and should not be typed in (though of course you're always encouraged to test things out on your own or implement them in different circumstances as a student!)

The Course Files

The course files are a set of programs, documentation, data, and metadata that are packaged for use in examples and exercises as part of this course. Other courses in the ENVI and IDL series have their own course files. The files for this course are included on a CD in the directory 'extending.' If you do not have the CD available and wish to obtain the course files, please contact us at: `training@exelisvis.com`

Installing the Course Files

The following are the steps used to install the course files on your computer system. For instructions on mapping the course files to a project in the IDL workbench, consult Chapter 3: IDL Basics.

1. Select a directory on your computer to install the course files. Depending on your operating system and the permission settings in your user profile, this may need to be somewhere in "user space" – your 'home' directory on UNIX-based operating systems like Mac OS X and Linux, or somewhere like the 'My Documents' directory on Windows systems.
2. Copy the files from the CD to this location. If they have been distributed in a compressed format, use a decompression utility such as unzip, winzip, or 7-zip to extract the files to your chosen location.
3. After extraction, the course files should be installed in the directory **extending**. We recommend copying this directory to a parent directory entitled **ENVI_Coursefiles** or **IDL_Coursefiles** to keep all course materials for ENVI and IDL grouped in one location so they can be easily located.

This completes the installation of the course files on your machine. Next, we need to instruct IDL where to find them. As mentioned before, this will be addressed in the Projects section of Chapter 3.

Starting the IDL Workbench

This course uses the IDL workbench and assumes your version of IDL is one included with ENVI. While it is possible to use command line IDL or other editing environments (such as Emacs, Vi, or Notepad++) this course relies on workbench functionality for several exercises. Similarly, using IDL without an ENVI installation will fail for the ENVI programming material in this course, as ENVI libraries will not be available.

Under Windows, the ENVI linked IDL workbench can be started from the **Start** menu using the **ENVI** installation directory. The default installation path is **Start > Programs > ENVI X.X > IDL**.

Under Mac OS X, an IDL application should be available from the Dock.

On Linux and various other UNIX-based systems, the workbench can be launched by typing the command `idlde` at a shell prompt.

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.

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
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.

```
IDL> for i=0,9 do b[i] = i
IDL> print, b
0.000000      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> print, c
0.000000      1.00000      2.00000      3.00000      4.00000
5.00000       6.00000      7.00000
8.00000       9.00000
IDL> print, 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 s and represent things like sequences, images, and series. We can access individual elements or groups of elements from an array by subscripting the array.

```
IDL> print, 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')
% Compiled module: FILE_WHICH.
IDL> restore, file2
IDL> help, elev
ELEV            INT       = Array[350, 450]
```

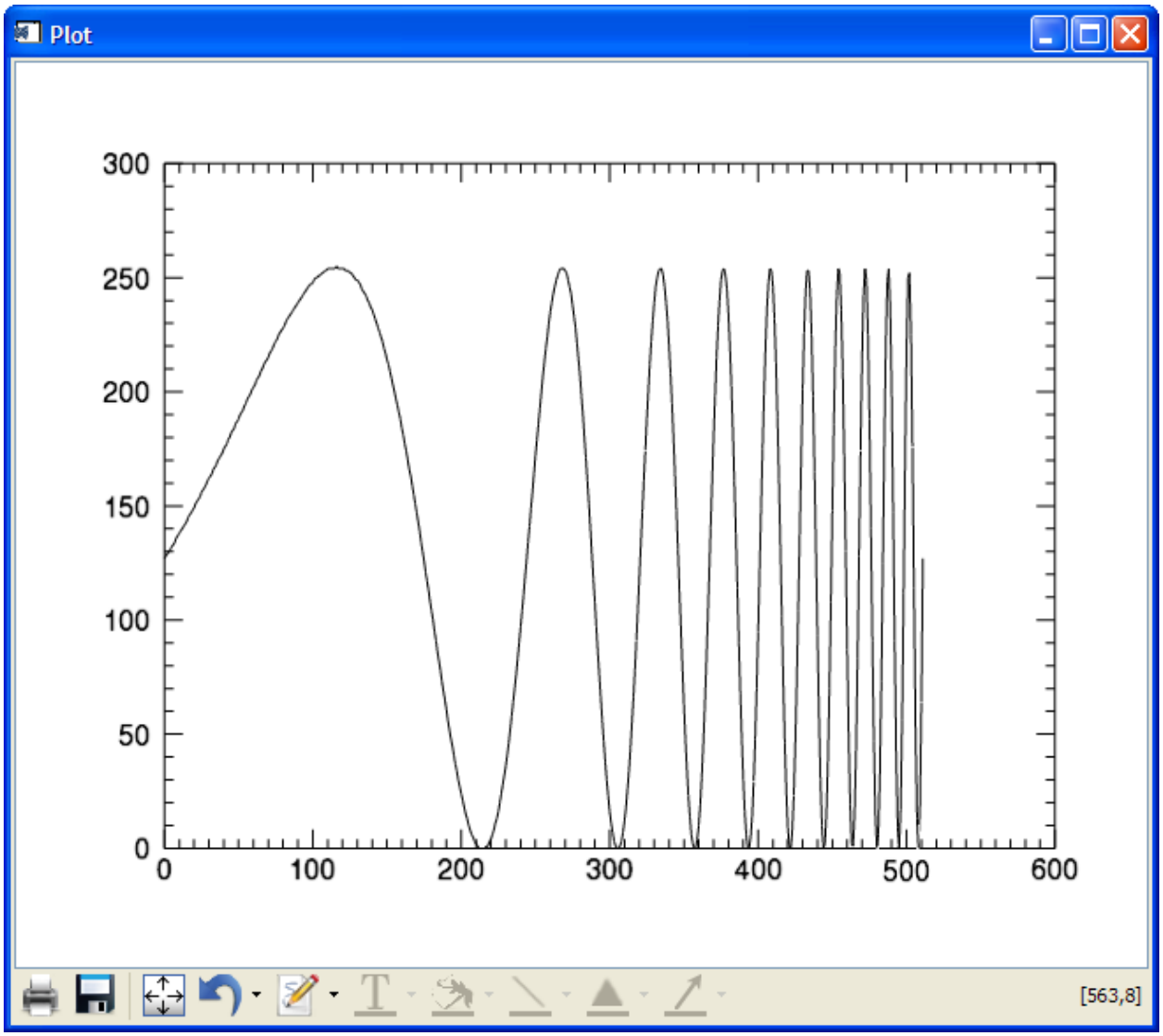
The IDL variable ELEV is restored from this file into your current IDL session. Note that this variable is a two-dimensional array of type integer.

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)
% Compiled module: FILE_WHICH.
```

A graphics window appears:



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 restored 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'
```

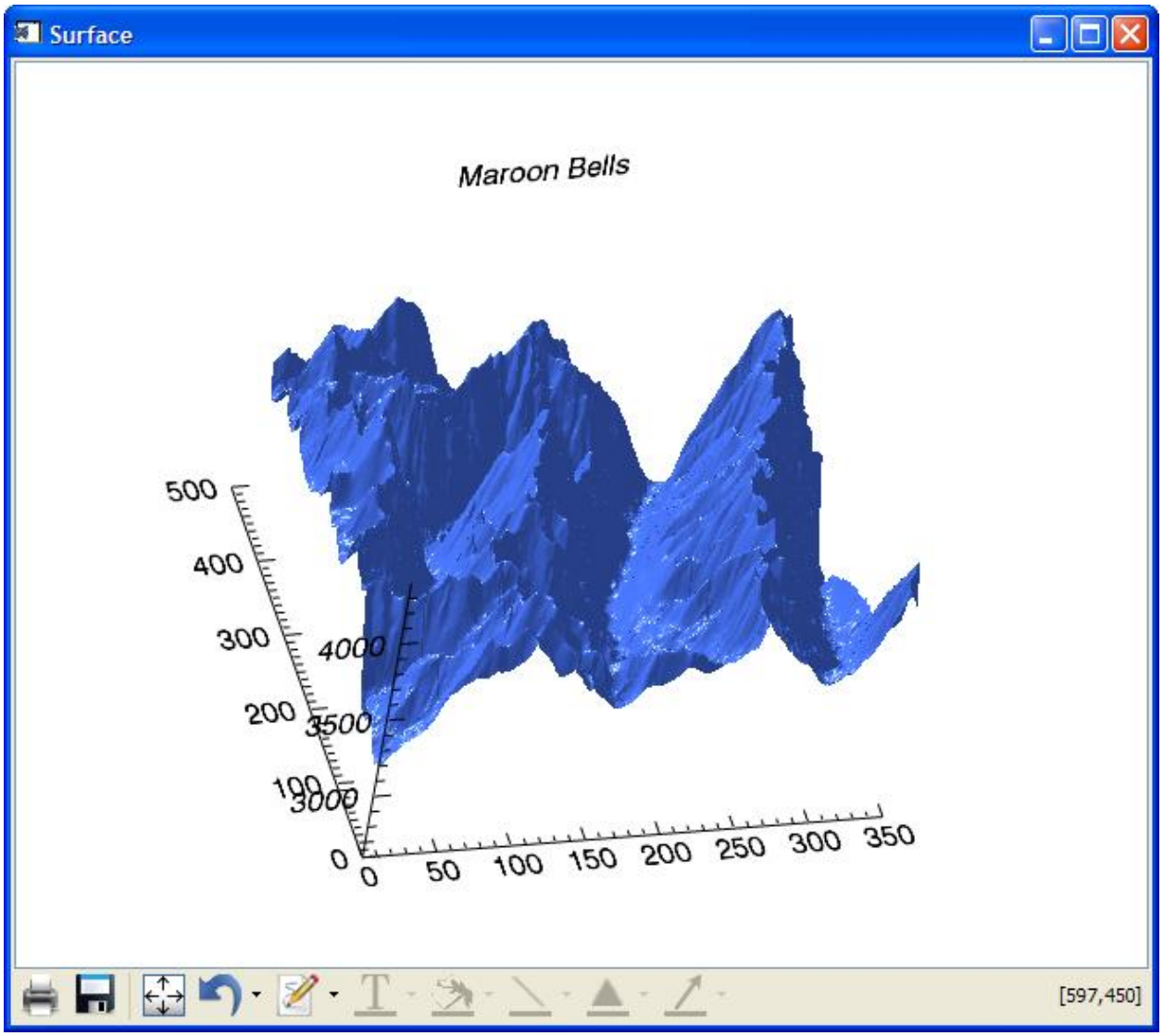
Change the surface style back to filled and set its color to royal blue with the method setProperty :

```
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.



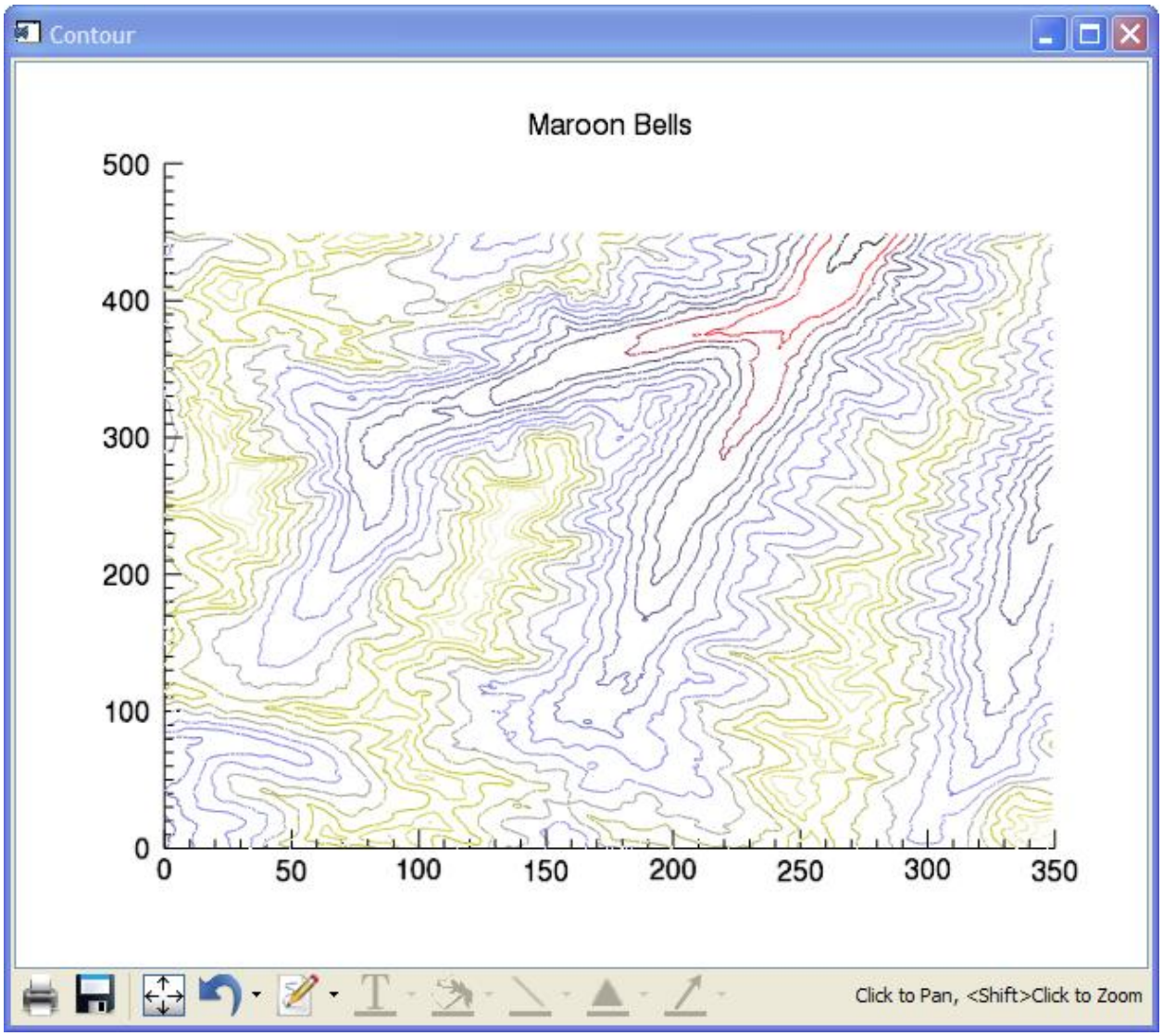
The ability to interact with a data representation and view it from different angles can assist in understanding the features of the data.

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:

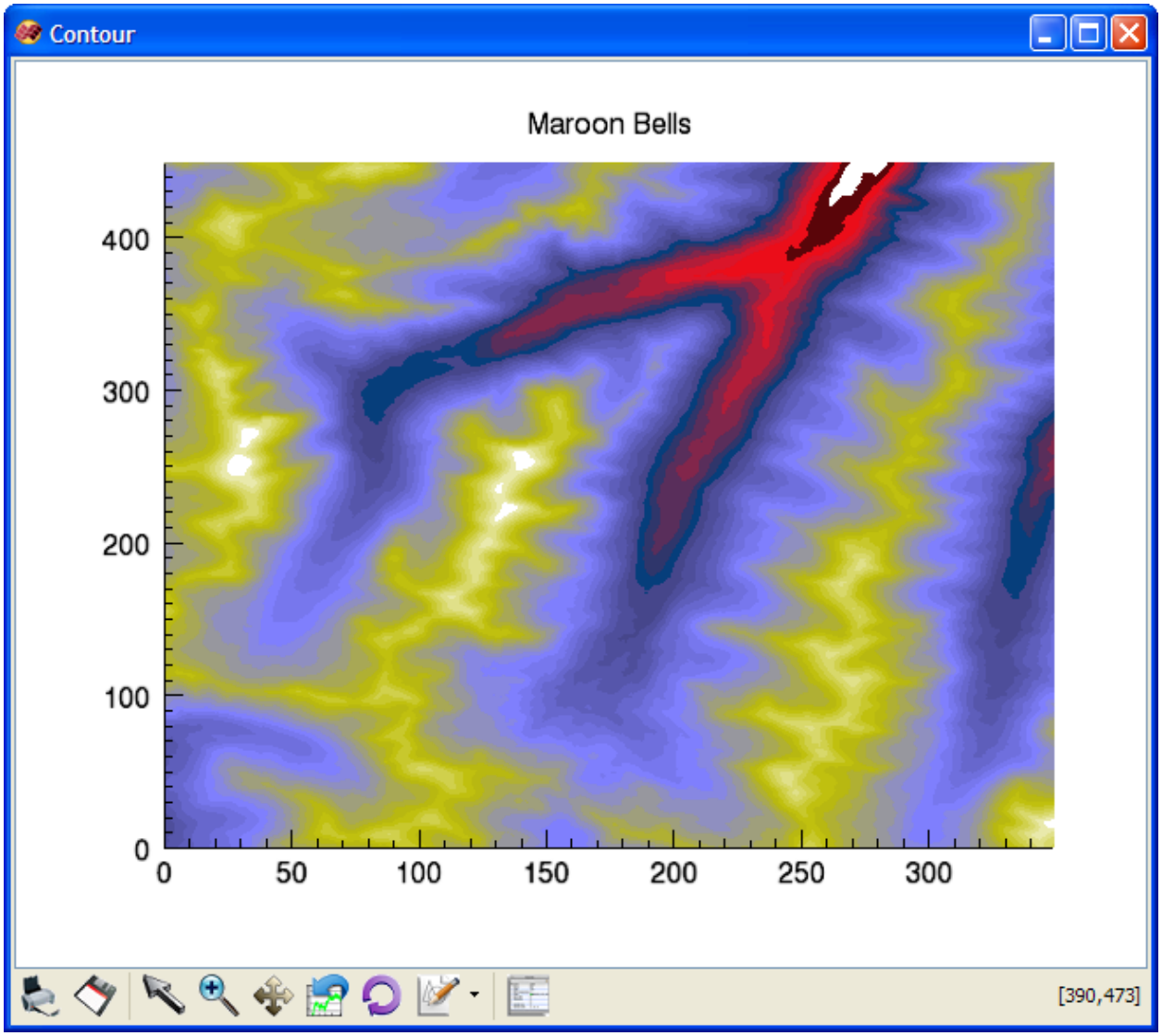
```
IDL> print, clevels
    2700    2800    2900    3000    3100    3200    3300    3400
    3500    3600    3700    3800    3900    4000    4100    4200
```

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')
```

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
```

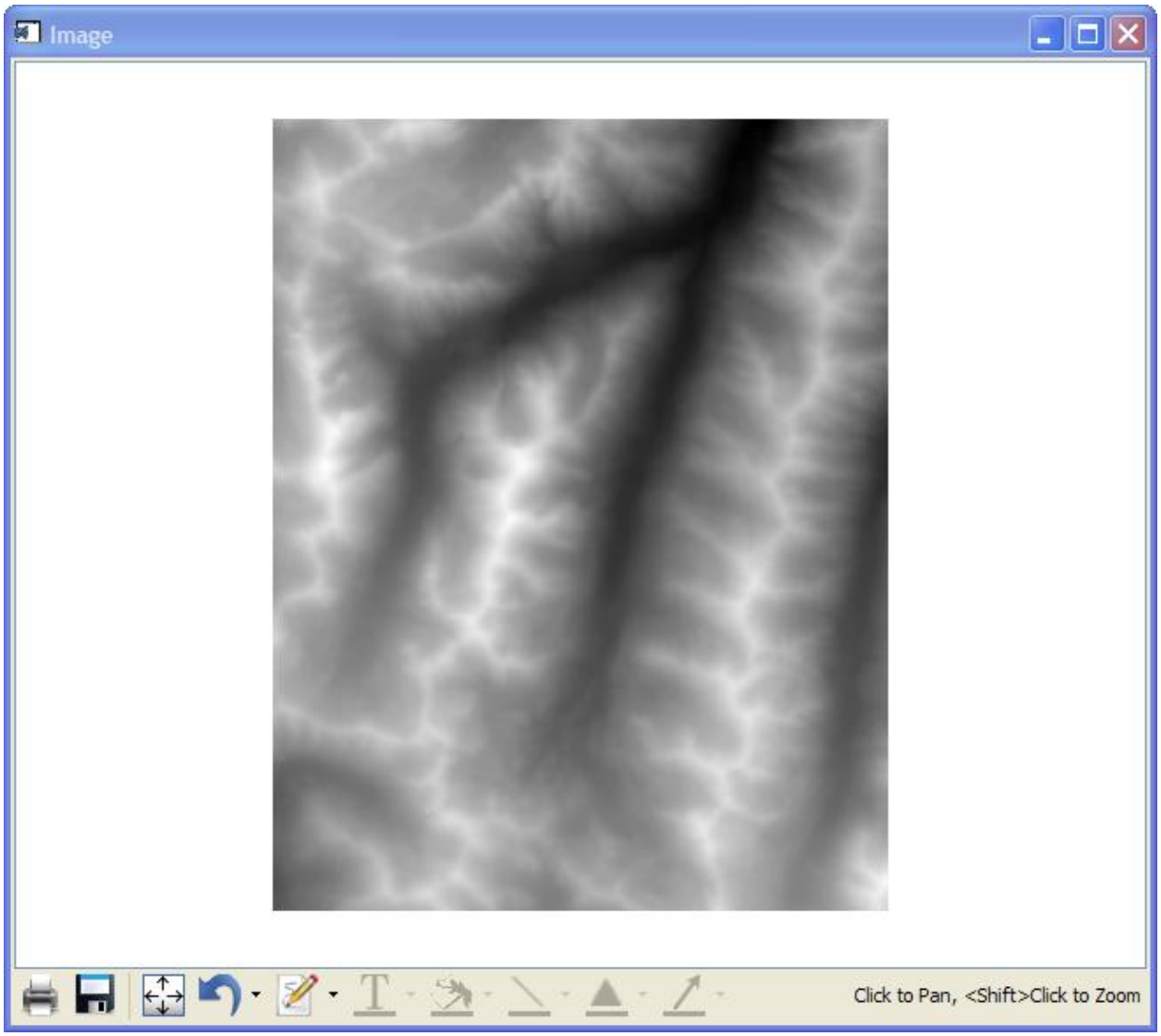
This makes a visualization with elevation proportional to the colors in the color palette.

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)
```



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))
```

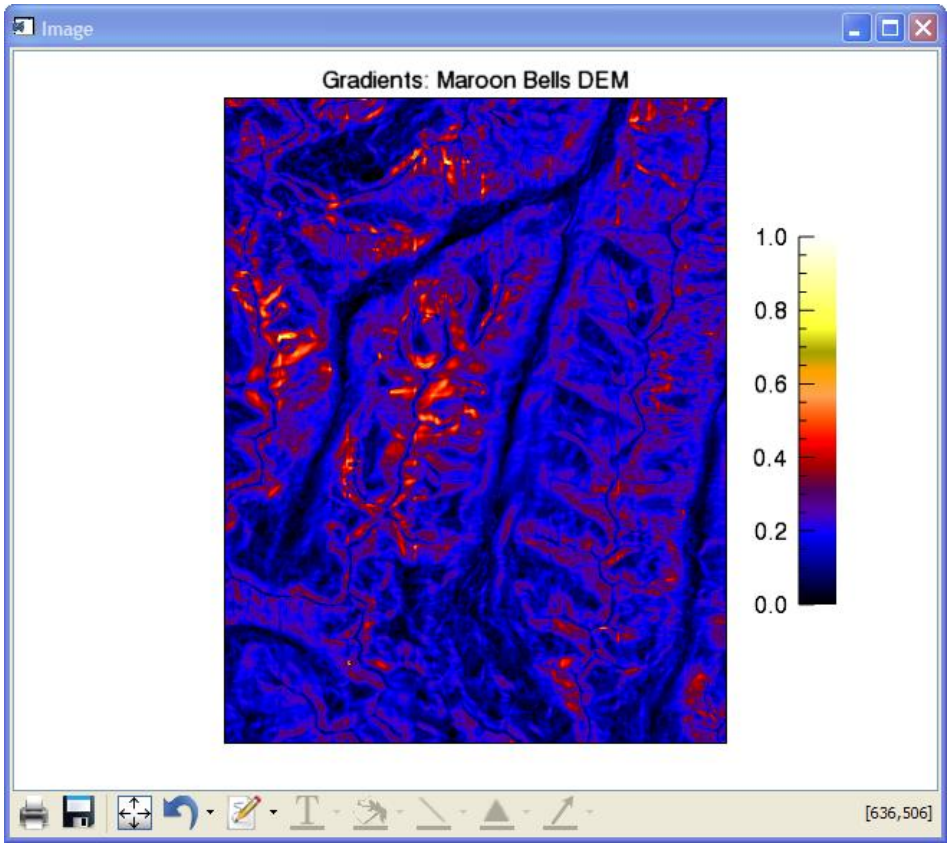
and view the result with IMAGE :

```
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)
```

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

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. With recent changes to IDL, this is the only manual we currently recommend as both comprehensive and up to date.

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.

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.

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; 8.1, for example.

Mac OS X: **/Applications/exelis/idlXX**

Linux: **/usr/local/exelis/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 the following table:

The directories in the IDL distribution

Directory	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.
products	Where other Exelis VIS software built on IDL, such as ENVI (prior to version 5.0), are stored. **NOTE**: the **products** directory is only present prior to ENVI version 5.0.
resource	A catch-all directory that contains fonts IDL uses, the IDL mapping and color databases, as well as examples of IDL Bridge technologies.

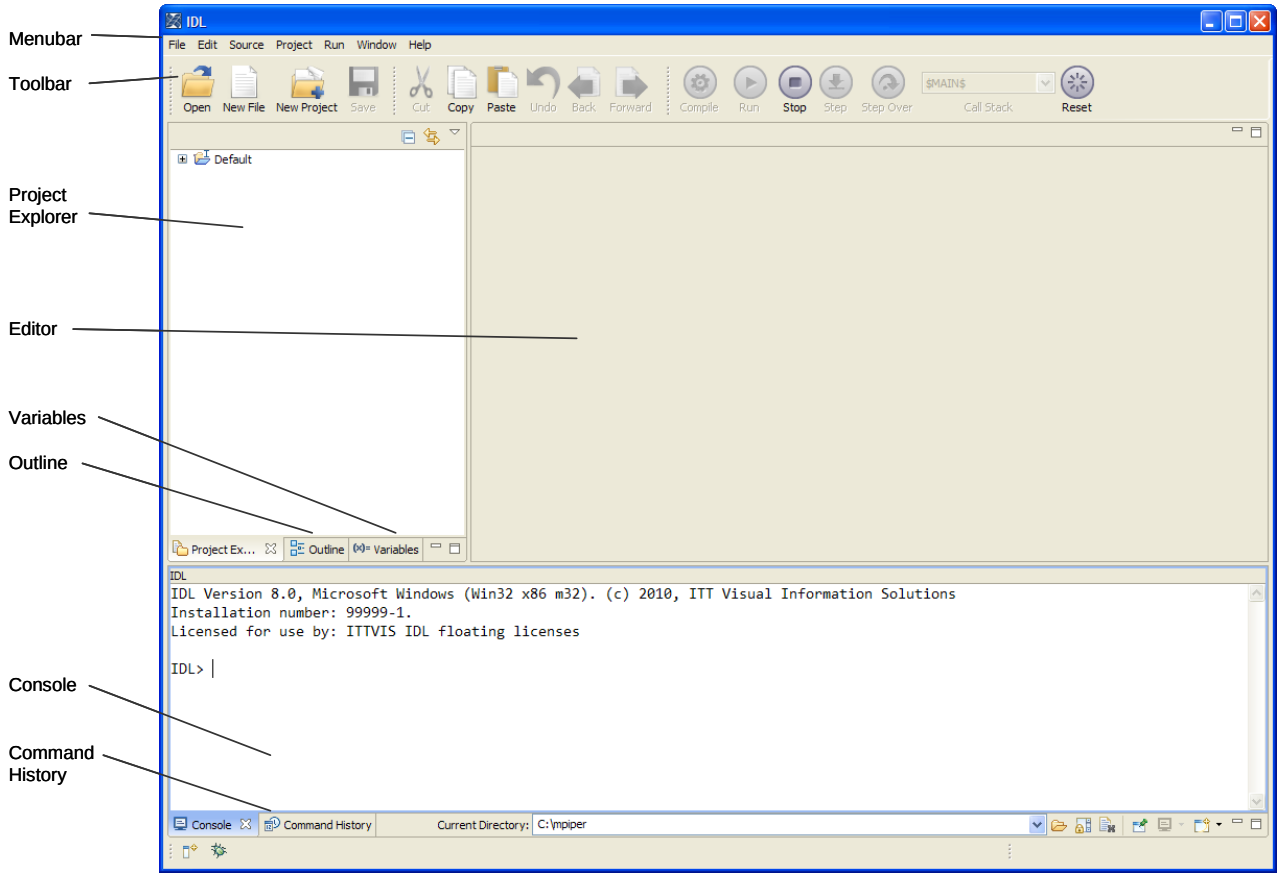
With IDL's standardized directory structure, if you know the location of a file in the IDL distribution on one system, you'll know its location on another system, even if it's on another platform.

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. The following figure shows an annotated screenshot of the IDL workbench.

The IDL workbench

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



IDL workbench views

View	Function
Menubar	Controls for opening, editing, compiling and running IDL programs, as well as controls for interacting with other features of the workbench.
Toolbar	Graphical controls similar to those on the Menubar.
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.
Outline	Presents a list of programs in a file; click on the program name to display it in the Editor.
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.
Console	Where IDL statements are entered (at the IDL> prompt) and where IDL returns information to the user. Note the syntax coloring.
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.

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

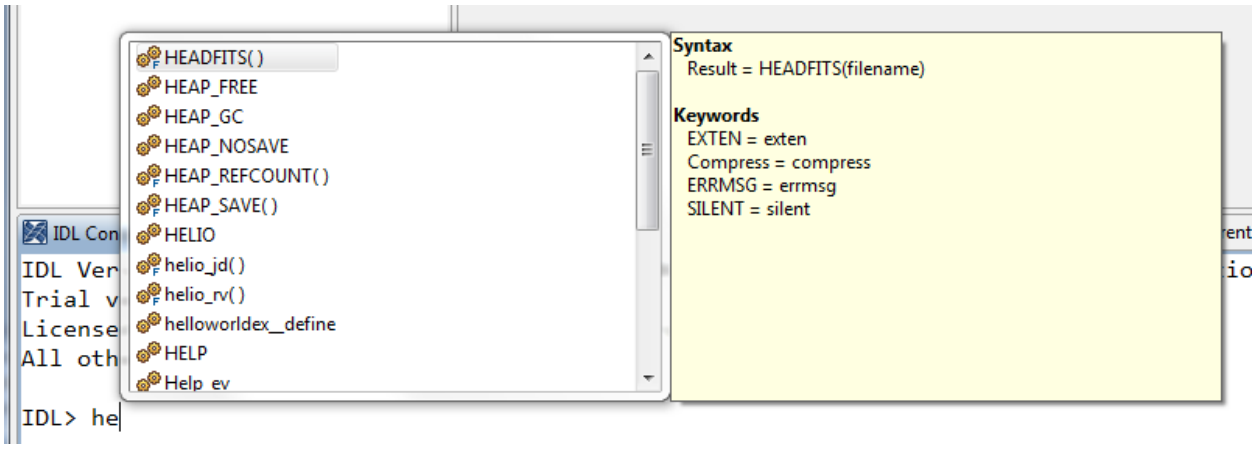
The best way to learn about the IDL workbench is to experiment with it. Here's a list of examples to try. Note that these are first steps; we'll do much more as we move through the course.

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 "M" stands for the<Alt>key.

Selected workbench keyboard shortcuts

Default style	Emacs style	Function
C-o	C-x	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	M-w	Copy
C-v	C-y	Paste
C-z	C-x u	Undo
C-f	M-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 5, "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 5, "Programming."

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.

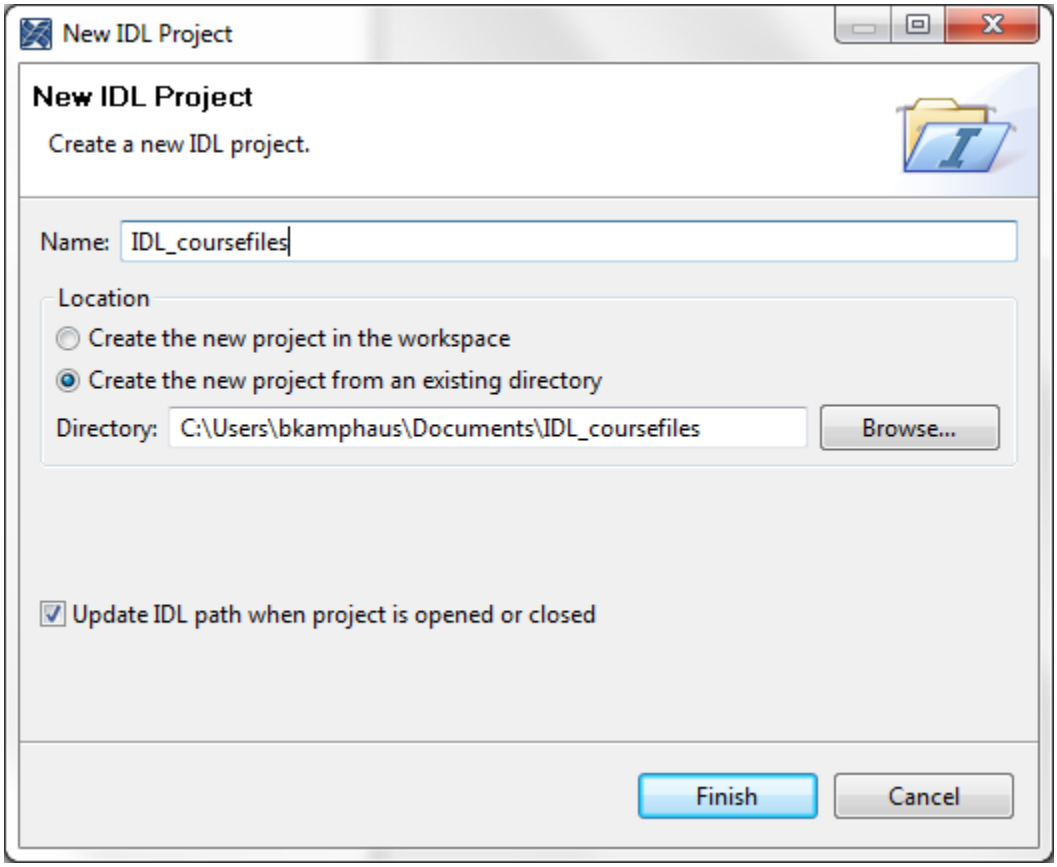
The Project Explorer view acts like an embedded file browser in the workbench (it looks suspiciously like Windows Explorer). With it, you can browse files related to your work in IDL. More importantly, projects also provide an easy way to manage IDL’s path —the set of directories IDL searches to find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

By default, a project appropriately named “Default” is provided. It maps to a directory named Default on the file system. You can verify it lives under **$HOME/IDLWorkspace81** , where **$HOME** is your home directory, by right-clicking on it in the **Project Explorer** and selecting **Properties** from the menu. A good

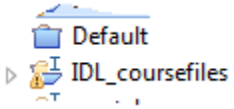
use for this Default project is as a scratch directory for temporarily storing files. We'll use it in class as a place to store our work.

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

1. From the Menubar, select File g New Project.. . The New IDL Project dialog appears: 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 **extending** directory on your computer. (Recall we installed the course files in the section "Installing the course files" in Chapter 1). Or, enter it manually.



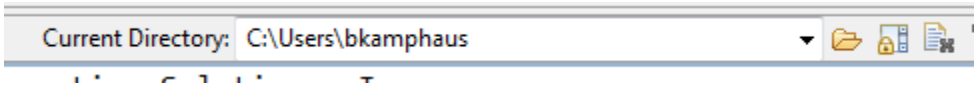
4. Leave the button titled "Update IDL path when project is opened or closed" checked to allow IDL to manage the path for this project.
5. Click the Finish button. The new project will appear in the Project Explorer view of the workbench. Your Project Explorer should now look like this:



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.

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 drop list:



This is the home directory for the user "bkamphaus" 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/bkamphaus/stuff
$ idl
IDL> cd, current=c & print, c
C:\Users\bkamphaus
```

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.

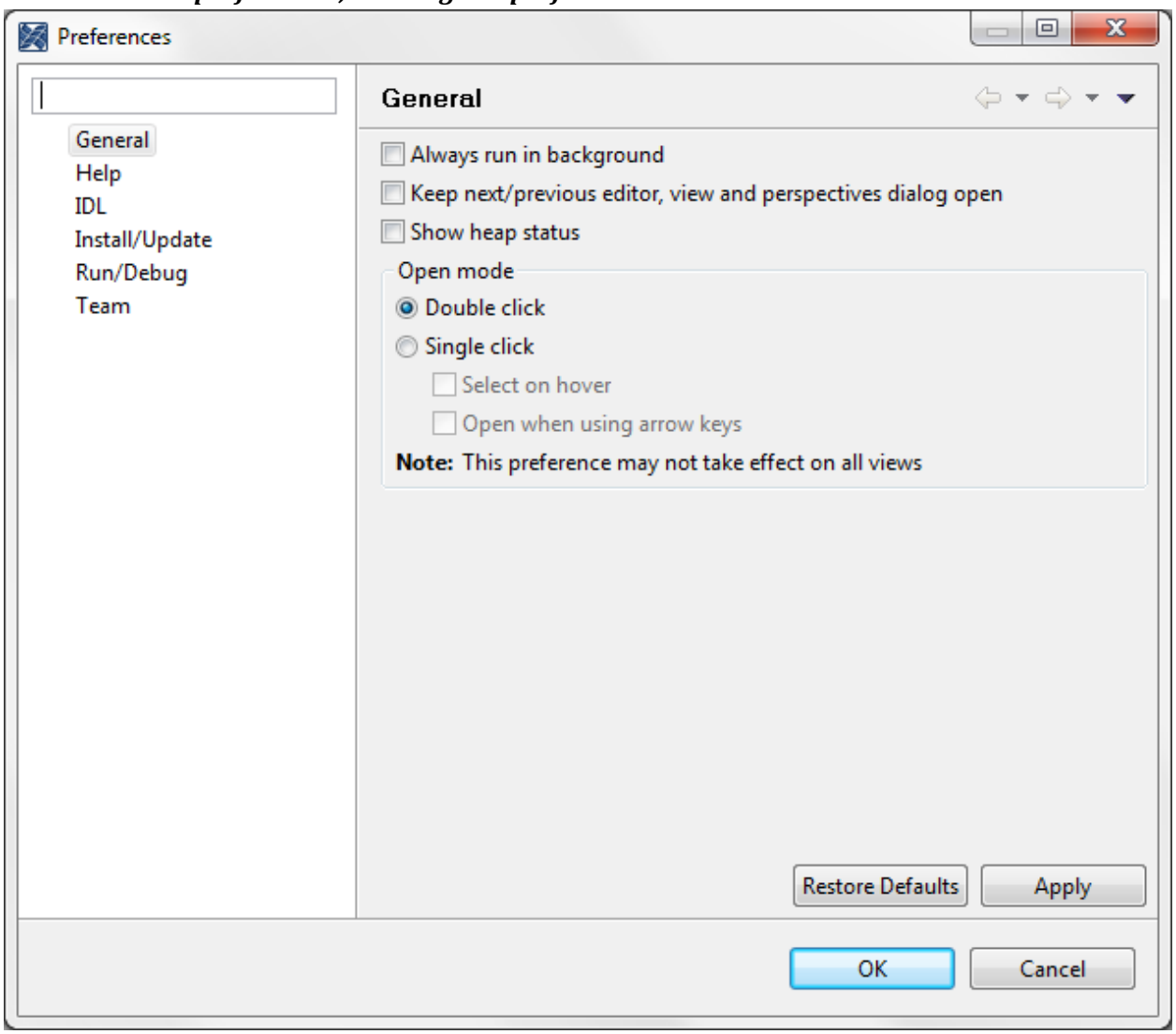
If you're using the workbench for this course, set the Default project to be the working directory by right-clicking on it in the Project Explorer and selecting **Set Selection as Current Working Directory** from the menu. If you're using IDL command-line mode, CD to a directory where you'll store results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

In the workbench Menubar, select **Window-> Preferences** (on Mac OS X, select IDL g Preferences). Within the Preferences dialog, select the IDL item from the left pane. The dialog should look like the figure opposite:

The workbench preferences, showing IDL preferences.



Next, set the "Initial working directory" field to your Default project directory, either by entering it manually or by using the Browse button.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 the following table.

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 Chapter 5, "Programming."

If you use the workbench, you can use projects to manage IDL's path, as shown previously in this chapter. 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> print, 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 course files 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 "bkamphaus" in **/home/bkamphaus/IDL_coursefiles** , then construct a new path using:

```
IDL> new_path = path + path_sep(/search) $
> + expand_path( '+/home/bkamphaus/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/bkamphaus/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 course files distribution.

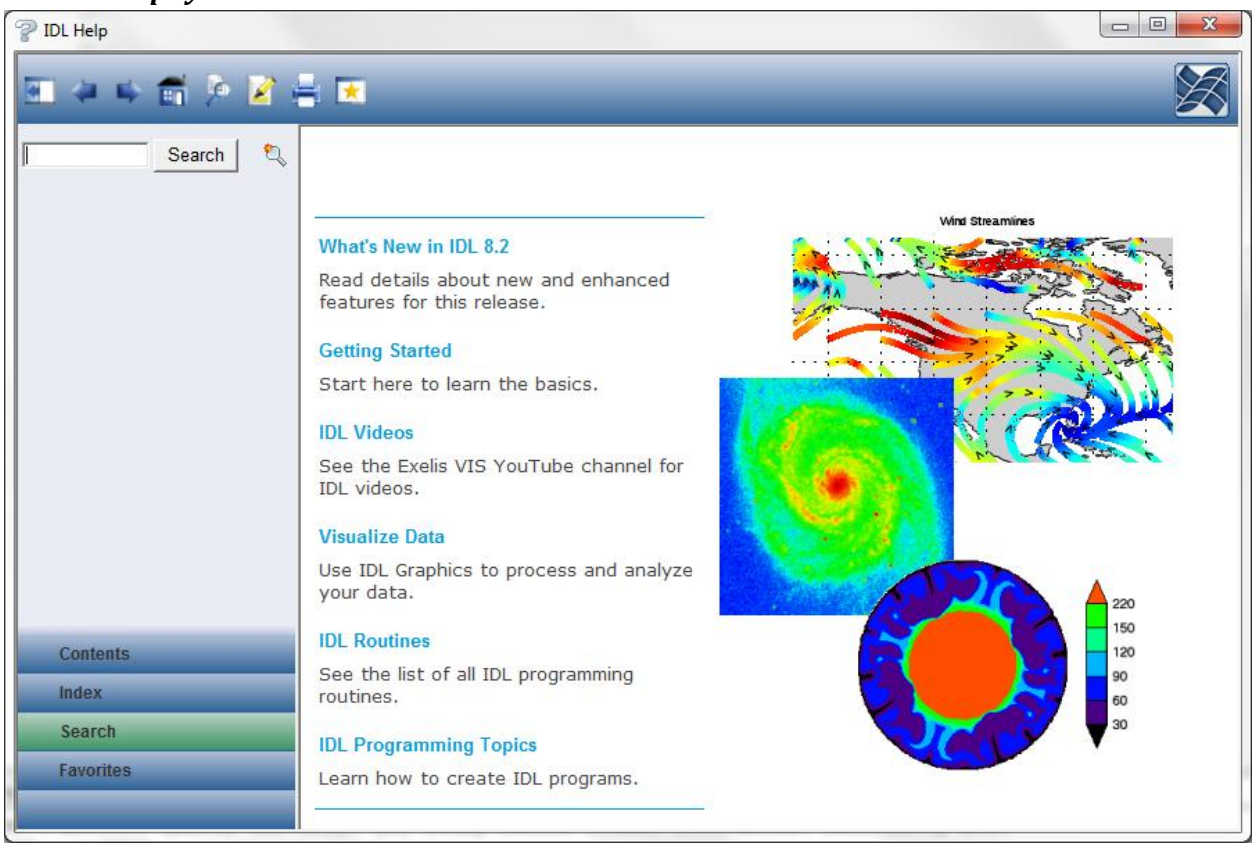
The IDL Help system

Every IDL installation is equipped with extensive built-in documentation in the IDL Help system. With the Help system, you can learn about:

- 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.

The IDL Help system browser.



Start the Help system by selecting Help g 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 above) is divided into left and right panels. The right panel initially displays the Help system home page. The left panel has tabs for browsing the contents of the Help system, browsing the index of the Help system, and searching the Help system.

Though the Help system has information on just about every conceivable IDL topic, finding the information you need can sometimes 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.

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 Data Structures

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
_redhelp
f0oBaR
read$_file5
```

and invalid variable names:

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

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 the following table, along with a short description of their purpose.

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) - IEEE standard values.
!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 the following table:

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	ULON64
float	4	±1E±38	`h = 0.0`	4	FLOAT
double	8	±1E±308	`i = 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	varies		`l = 'hello'` `l = "hello"`	7	STRING / STRTRIM
structure	varies		`m = {field1:0}`	8	CREATE_STRUCT
pointer	4		`n = ptr_new()`	10	
object	varies		`o = obj_new()` `o = IDLgrPolygon()`	11	
undefined	1		Use without defining, e.g.: `cd, current=o`	0	
null	1		`p = !null`	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 the previous table, we see that the largest value we can create with the integer type is 215 - 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.

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

```
IDL> c = 10000.0 + 0.00001
IDL> print, 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> print, !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 a separate condition when 'b'
                             is defined vs. undefined
IDL> print, 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)
```

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 an integer typed identity 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.

```
IDL> help, y
Y               DOUBLE    = Array[4]
```

Arrays of the different IDL data types can be created with the routines listed below . 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:

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		

```
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      1.00000      2.00000      3.00000      4.00000
```

The best way to initialize an array to a constant value is to use one of the array creation functions listed in the above table 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
```

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 = indgen(10)
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]
      4.00000
```

Or, use negative array indexing:

```
IDL> print, a[-1]
      4.00000
```

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> print, m[*,4] ;the fifth row
      16      17      18      19

IDL> print, m[2,*] ;the third column
       2
       6
      10
      14
      18

IDL> print, m[2:3,1:2] ;a block in the middle
       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> print, m[17]
      17
```

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

```
IDL> print, 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. In a lower level language, you would accomplish this with a FOR loop by iterating through the array and marking locations in a second array. IDL by contrast includes the WHERE function, a vectorized tool that returns the indices of elements in an array where a condition is met. The syntax is as follows:

```
INDICES_RETURNED = WHERE(ARRAY_WITH_CONDITION, COUNT_RETURNED)
```

For our array, this looks as follows:

```
IDL> i_neg = where(n lt 0.0, n_neg)
IDL> print, i_neg
           1           3           4           6           8
          10          12          14
IDL> print, n_neg
           8
```

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

```
IDL> print, 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.

Note, in the case that a condition is not met, WHERE has two possible outputs. The default is to return a value of -1 (set in versions of IDL before negative subscripting was allowed). Optionally it can be set to return !NULL instead. The two uses are illustrated below:

Default:

```
IDL> i_none = where(n gt 100.0)
IDL> print, i_none
          -1
IDL> print, n[i_none]
     -1.09031
```

With NULL keyword set:

```
IDL> i_none = where(n gt 100.0, /null)
IDL> print, i_none
!NULL
IDL> print, n[i_none]
!NULL
```

Note that the above example prevents the subsequent subscript from mistakenly returning a location in which the condition is not actually met.

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, indgen(10), '1123 6536 5321')
IDL> help, a
A               LIST  <ID=5700  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=5709  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'
IDL> b.add, 'peach', 0 ;added before 'apple'
IDL> b.add, 'boysenberry'
IDL> b.remove ;will remove last element w/o argument
```

```
IDL> print, b
peach
apple
orange
```

Lists can also be joined:

```
IDL> c = a + b
IDL> print, c
Homer
      3.14159
0      1      2      3       4       5       6       7       8       9
1123 6536 5321
peach
apple
orange
```

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=5733  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> print, colors['red']
     255       0       0
```

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
green:        0     255       0
blue:         0       0     255
chartreuse:       127     255       0
red:      255       0       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
blue:         0       0     255
chartreuse:       127     255       0
red:      255       0       0
```

We can retrieve all the keys in a hash with:

```
IDL> keys = colors.keys()
IDL> help, keys
KEYS            LIST  <ID=5745  NELEMENTS=3>
```

Lists and hashes can also be iterated with the FOREACH operator: examples are given in Chapter 5, “Programming.” Lists and hashes are covered in greater depth, including where and why they might be used, in the Application Development with IDL 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
MYCAR           STRUCT    = -> CAR Array[1]
STAR            STRUCT    = -> <Anonymous> Array[1]
IDL> help, mycar, star, /structures
** Structure CAR, 3 tags, length=28, data length=26:
   MAKE            STRING    'Honda'
   MODEL           STRING    'Fit'
   YEAR            INT           2009
** Structure <442b208>, 3 tags, length=20, data length=20, refs=1:
   NAME            STRING    'Sirius'
   RA              FLOAT           6.75248
   DECL            FLOAT          -16.7161
```

The output from HELP with /STRUCTURES set clearly shows the field names, types and values.

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

```
IDL> print, mycar.make
Honda
```

Structure fields can be used in expressions:

```
IDL> x = mycar.year * 15.5
IDL> print, x
      31139.5
```

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 <26851e0>, 4 tags, length=24, data length=24, refs=1:
   NAME            STRING    'Sirius'
   RA              FLOAT           6.75248
   DECL            FLOAT          -16.7161
   MAG             FLOAT          -1.46000
```

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> print, 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> print, '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> print, '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 the table below. Several examples follow.

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 to lower case.
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 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.

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'
```

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 the help topic on STRING 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')
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
```

We could use a similar process to parse a metadata field read from a text file to figure out the day on which a particular scene was imaged:

```
IDL> s = 'Collection date: 2012-08-17 14:20:23'
IDL> searchlen = strlen('Collection date:')
IDL> print, searchlen
          16
IDL> loc = strpos(s, 'Collection date:')
IDL> print, loc
           0
IDL> print, strmid(s, loc + searchlen, 12)
 2012-08-17
```

More examples of using the string processing routines listed in the previous table 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   = <PtrHeapVar5752>
IDL> print, p
<PtrHeapVar5752>
```

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

```
IDL> print, *p
      42
```

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

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

Assign new data to the pointer:

```
IDL> *p = 'hello'
IDL> print, *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 course), 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> print, 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 statementprint , "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 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

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.

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 5 Programming

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.

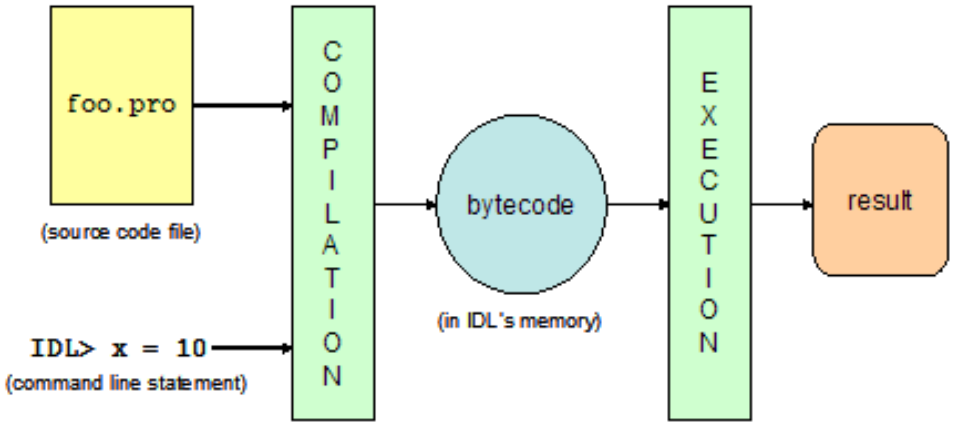
A program is a sequence of statements that are combined to act as a unit. IDL supports three program types:

- 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.

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 non-interactive batch mode. The table below lists the executive commands and their purpose.

Command	Description
.compile filename	Compiles program modules. If called without a filename it will 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 the reset 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, ore 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, then stops; 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:

```
IDL> .compile -v 'C:\IDL_coursefiles\introduction\src\hello.pro'
```

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

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:

- 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.
- 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 pwd
% Compiled module: PWD.
```

and call it by name:

```
IDL> pwd
C:\Users\bkamphaus
```

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\pwd.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 Sep 19 12:06:15 2012
```

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

```
IDL> .compile time
% Compiled module: TIME.
IDL> print, time()
The current date and time is Wed Sep 19 12:06:34 2012
```

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:

- Self-contained
- Reusable.
- Have controlled inputs and outputs,

They allow a programmer to separate and logically group 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 var
IDL> w = beselj(t) ;dependent var
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 adelimiter; 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    = 'C:\Users\bkamphaus'
```

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:

- Named variables are passed by reference
- 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.

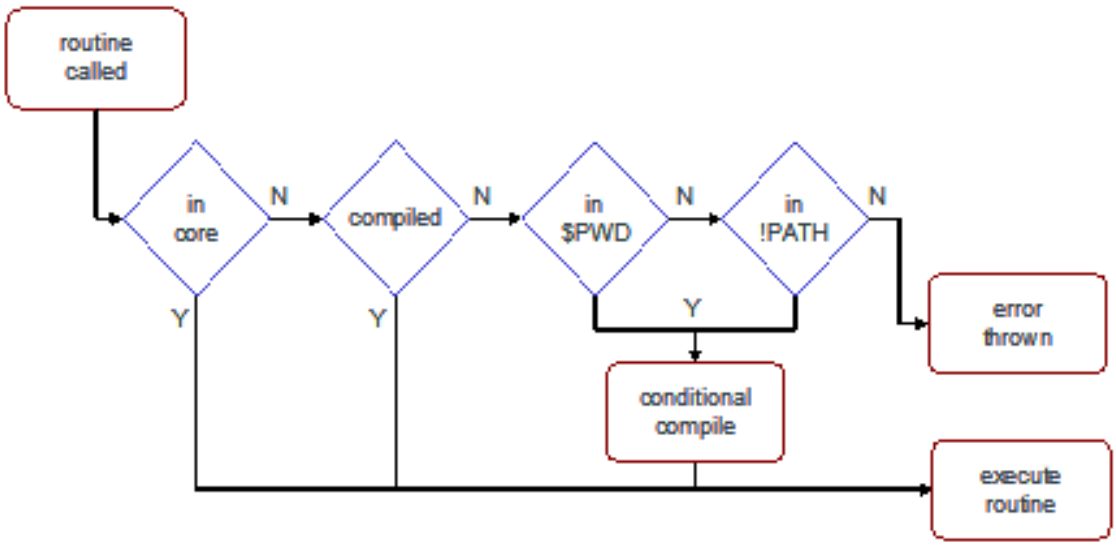
Note that IDL doesn't formally have a way to distinguish whether parameters are to be passed in or out of a program. Documentation, therefore, is key. In the Help system, output parameters for a routine will list the need for a named variable. For example, the CURRENT keyword for CD uses this text:

> *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.

The following figure diagrams the process of the 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 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. The following table lists the operators IDL supports, their order of precedence and examples of their use.

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`
			`6.00000`

	{}	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`

Priority	Operator	Description	Example
	+	Addition / string concatenation	`IDL> print, 5 + 2` `7`
	-	Subtraction / unary negation	`IDL> print, 5 - 2, -5` `3 -5`
	<	Minimum of	`IDL> print, 5 < 2` `2`
	>	Maximum of	`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 * 24 / 2
      24
```

Compound operators

Compound operators are supported in IDL from version 6.0. The available operators are listed here:

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

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

```
a operator= expression
```

is equivalent to:

```
a = temporary(a) operator 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 5 π .

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:

```
IDL> print, indgen(5) > 2
       2       2       2       3       4
IDL> print, indgen(5) 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)
IDL> print, a
       0       1       2
       3       4       5
IDL> b = indgen(2, 3)
IDL> print, b
       0       1
       2       3
       4       5
IDL> print, a # b
           3           4           5
           9          14          19
          15          24          33
IDL> print, 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 =       -0.54909
y = |x| =  0.54909
IDL> if_ex
x =        0.28984
x is greater than zero
y = |x| =  0.28984
```

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:

```
pro case_ex, image
   compile_opt idl2

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

   ; Use CASE to test which dimension is 3.
   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 6, "File Access") to read the file marsglobe.jpg , located in the IDLexamples/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
% Compiled module: CASE_EX.
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 error 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? 5? How about the (nonexistent) band 12?

```
IDL> switch_ex, band=7
```

```
Landsat band number 7 corresponds to band index 5
IDL> switch_ex, band=4
Landsat band number 4 corresponds to band index 3
IDL> switch_ex, band=12
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 the following table. 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.

Variable or expression type	Truth values
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 variables are true, null or undefined heap variables 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

   ; Generates 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** 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 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 iterate through the zoo one animal
   ; at a time.
   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 Sep 21 15:39:48 2012
C:\Users\bkamphaus\IDLWorkspace82\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.

Programming tips

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

- Main programs are good, but procedures and functions are better.

- Use COMPILE_OPT IDL2 in every new procedure and function. This implies the use of square brackets for subscripting arrays.
- Use a consistent coding style. This will make it easier for others (and, sometimes, yourself) to read your programs.
- Use descriptive variable and program names.
- 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 has collected programming tips at:

http://michaelgalloy.com/2006/07/14/12-tips-for-beginning-idl-programmers.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 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 and structured modular programming.

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 programs.

Chapter 6 File Access

This chapter describes how to read and write data files with IDL.

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, Notepad++, 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** later in this chapter).

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:

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 functions returns the 'basename' of a file path. The basename is the final rightmost segment of the file path, omitting directories.
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, or the actual file.
FILE_EXPAND_PATH	This will expand a file or partial directory name to its fully qualified name, generally by appending the current working directory to the beginning.
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 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 manipulation routines in IDL loosely grouped by function.

Routine	Purpose
FILE_LINES	Reports the number of lines of text in a file.
FILE_LINK	Creates UNIX file links, both regular (hard) and symbolic.
FILE_MKDIR	Creates a new directory on the file system.
FILE_MOVE	Moves files or directories to a new location.
FILE_READLINK	Returns the path pointed to by a UNIX symbolic link.
FILE_SAME	Determines whether two different file names refer to the same underlying file (i.e., file location, not a checksum).
FILE_SEARCH	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 ocmmand, 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 a platform-native graphical file selection dialog.
EOF	This functions test 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	Reads data in an open file and moves the file pointer. It will skip a known amount of data in a file without making it available to 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	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 to force synchronicity.
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

Here are some examples of using selected routines from the previous table.

Use FILE_WHICH to get the full path to the file containing the HANNING function:

```
IDL> file = file_which('hanning.pro')
```

```
% Compiled module: FILE_WHICH.
IDL> print, file
C:\Program Files\Exelis\IDL82\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\Exelis\IDL82\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('.')
C:\scratch
```

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)
% Compiled module: GET_COURSEFILES_DIR.
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\idl_wavy.png
C:\ IDL_coursefiles\data\wind_rose.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_internal.pro
ascii_template.pro
```

See the IDL Help system for more information on, and examples of, the routines listed in the previous table.

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 course files **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 bkamphaus@MKTBKAMPHAUS-LT, Wed Sep 19
12:19:56 2012.
% RESTORE: IDL version 8.2.1 (Win32, x86).
% RESTORE: Restored variable: TIME.
% RESTORE: Restored variable: TDRY.
```

This is much easier than reading these data with the netCDF file API every time we want to use them.

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.

Example IDL Standard File Type Access

Format	Extension	Query routine	Read routine	Write routine
ArcGIS binary 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	varies, **dat, img**		READ_BINARY, READU	WRITEU
Joint Photographic Expoerts Group	**jpg, jpeg**	QUERY_JPEG	READ_JPEG	WRITE_JPEG
JPEG2000	**jp2, jpx**	QUERY_JPEG2000 IDLffJPEG2000 class	READ_JPEG2000 "	WRITE_JPEG2000 "
NCAR Raster Interchange	**nrif**			WRITE_NRIF
Portable Network Graphics	**png**	QUERY_PNG	READ_PNG	WRITE_PNG
ESRI Shapefiles	**shp**	IDLffShape class	"	"
Multi-resolution Seamless Image Database	**sid**	QUERY_MRSID IDLffMrSID class	READ_MRSID "	
Tagged Image File Format (including GeotIFF)	**tif**, **tiff**	QUERY_TIFF	READ_TIFF	WRITE_TIFF
eXtensible Markup Language	**xml**	IDLffXMLSAX class, IDLffXMLDOM class		

In addition, IDL also supports the CDF/NETCDF and HDF families of scientific data formats. For more information on these, see the appropriate help topics:

Format	Extension	Help Topic in the IDL Help
Common Data Format	**CDF**	"CDF Routines"
Network Common Data Format (NetCDF)	**NC, CDF**	"NCDF Routines"
Hierarchical Data Format (v4)	**HDF**	"HDF Routines"
Hierarchical Data Format (v5)	**H5, HDF5**	"HDF5 Routines"
Hierarchical Data Format - Earth Observing System	**HDF**	"HDF-EOS Routines"

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
** Structure <bbf7000>, 18 tags, length=120, data length=116, refs=1:
   CHANNELS        LONG                 1
   DIMENSIONS      LONG      Array[2]
   HAS_PALETTE     INT                  0
   IMAGE_INDEX     LONG                 0
   NUM_IMAGES      LONG                 1
   PIXEL_TYPE      INT                  1
   TYPE            STRING    'TIFF'
   BITS_PER_SAMPLE LONG                 8
   ORIENTATION     LONG                 4
   PLANAR_CONFIG   LONG                 1
   PHOTOMETRIC     LONG                 1
   POSITION        FLOAT     Array[2]
   RESOLUTION      FLOAT     Array[2]
   UNITS           LONG                 2
   TILE_SIZE       LONG      Array[2]
   DESCRIPTION     STRING    'IDL TIFF file'
   DOCUMENT_NAME   STRING    'd:/rsi/debug/examples/data/image.tif'
   DATE_TIME       STRING    ''
```

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
% Loaded DLM: PNG.
```

Or use WRITE_JPEG to write the data to a JPEG file instead:

```
IDL> write_jpeg, 'image.jpg', nyny
% Loaded DLM: JPEG.
```

More information on the routines listed in the previous table 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 the previous table. 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 the prevoius table , 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 course files 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 course files 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 Libraryhttp://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

The following table 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.

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's directory structure in Chapter 3 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 <4454150>, 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      LONG      = 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]
```

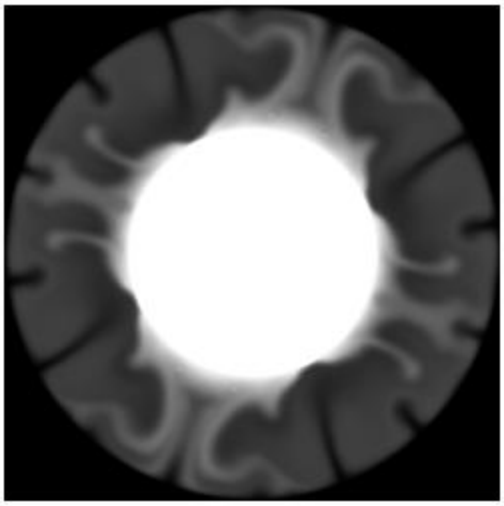
The file was read, but what to make of the data? We need more information on what this file contains. From the file index.txt in the examples/data subdirectory, we see that it holds one 248 x 248-element byte array representing a model of Earth's mantle convection. This supplemental information is needed 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. Define a data format for the file's contents by creating variables of appropriate type and dimensionality.
2. 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.Perform file operations on the file (querying, reading, writing, positioning).
4. 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:

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. Used when LUN is set manually.
FREE_LUN	Closes files and deallocates file units. Used when LUN is returned using GET_LUN

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\Exelis\IDL82\examples\data\ascii.txt
```

Close the file with the CLOSE procedure:

```
IDL> close, 1
```

Logcal 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's error 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 C:\Program Files\Exelis\IDL82\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 and 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.

Byte ordering for common platforms supported by IDL

Little endian	Windows (Intel or AMD), Linux (Intel or AMD), Mac (Intel)
Big endian	Mac (PowerPC), Sun Solaris (Sparc, Ultra)

In addition, there are several proprietary device formats for various sensors and instruments which may collect or aggregate binary data prior to it being delivered to any of the above platforms.

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 the first of the following tables. To change the endianness of numbers after reading them from a file, use the byte-swapping routines listed in the second table.

Keywords to OPEN routines for changing the endianness of data.

Keyword to OPEN routines	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.

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 overwrites the current variable in memory rather than storing the swapped data in a second variable.

Reading and Writing Text Files

The following table lists the low-level routines for reading and writing text files:

Routine	Purpose
READF	Reads data from a text file.
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.

- Input is performed on scalar variables. Array and structure variables are treated as collections of scalar variables.
- If the current input line is empty and there are variables left requiring input, read another line.

- If the current input line is not empty but there are no variables left requiring input, the remainder of the line is ignored.
- Input data must be separated by commas or white space (tabs, spaces, or new lines).
- When reading into a variable of type string, all the characters remaining in the current input line are placed into the string.
- 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.
- 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 = ''
```

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[0:7]
     399      692     1003     1333      811       90      100        4
```

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     =       23.3800
```

```
LAT1            FLOAT     =       38.0000
```

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" in Chapter 5) 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
% Compiled module: ASCII_READ_EX4.
```

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 geo-coordinates) 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

The following table lists the low-level 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 (see the earlier section from this chapter, "Byte ordering in binary files.") 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 fileconvec.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.”

Chapter 7 Band and Spectral Math

ENVI's Band Math and Spectral Math functions are particularly useful for quickly and flexibly manipulating image data. To use these tools, enter a mathematical expression and assign its variables to image bands or spectra. Band Math and Spectral Math operate with only three basic limitations:

- Your expression must return a 2D image array (for Band Math) or a 1D vector (for Spectral Math).
- The image bands (or spectra) assigned to each variable in the expression must have the same dimensions. In other words, all of the bands must have the same number of samples and lines, or all of the spectra must have the same number of bands.
- All variables must be named B1, B2, B3… or S1, S2, S3… and each of these variables must be assigned to one band or spectrum.

Because of their broad nature, these tools are often useful for quickly analyzing or combining different data sources. For example, Band Math can be useful for calculating a *difference image* for change detection, or for computing custom *indices*.

Exercise: Difference Image

1. From the ENVI main menu bar, select **File** → **Open…** Navigate to **envidata\extend** and open the file `avhrr.dat`.

2. Open the **Data Manager** by selecting its icon, or activating it from the **File** menu. Right click on the **SST Image** band and select **Load Grayscale**.

3. Create a second view by clicking on **Views**→ **Create New View** to split the display. Make sure the second view is active, then right click on **AVHRR Band 5 (12.0000)** in the Data Manager and click **Load Grayscale**.

4. Select **Display** → **Cursor Value** and move the cursor around an image and look at the data values for the two displays.

   This image is a NOAA AVHRR scene with 1.1 km pixel size, collected around 2:00 AM local time in mid-April 1997. The scene covers a section of the East Coast of the USA from approximately New York City to the North Carolina border. It extends out into the Atlantic Ocean about 1,000 km. You may recognize the Chesapeake Bay region along the left edge of the image.

   The first View contains sea surface temperature (SST) data in degrees Celsius, calculated from three thermal AVHRR bands. The second View contains SST data calibrated in Kelvin, calculated from one thermal band (at 12 µm). Note the spiral feature centered near image location (1030,1030). This is a warm eddy in the Gulf Stream current.

   These two images represent separate SST estimates in the scene. One algorithm is well-suited for SST estimates, and the other is more general. To examine the differences between these two

algorithms, you have to create a difference image and also account for the different temperature units in the two images. You can easily accomplish this with Band Math.

5. Select **Views→ One View**.

6. From the ENVI Toolbox, begin typing "Band Math" or select **Band Ratio→ Band Math**. A Band Math dialog appears.

7. In the **Enter an expression** field, type the following:

```
b1 - (b2 - 273.16)
```

8. Click **OK**. The Variables to Bands Pairings dialog appears.

9. The variable **B1** – [undefined] is selected by default. Map this variable to a specific band by selecting **SST Image** in the **Available Bands List** in the Variables to Bands Pairings dialog. The Variables used in expression window shows this assignment.

   Note: You can also apply a Band Math variable to an entire file instead of a single band by clicking Map Variable to Input File. This can be useful if, for example, you want to subtract one image band from all of the bands in a multi-band image.

10. Select **B2 – [undefined],** then click **AVHRR Band 5** to associate that band to the variable.

11. The bottom of the Variables to Bands Pairings dialog now provides options to output to file or memory. File is already selected by default. Enter an output filename of `sst_thermal_diff.dat`.

12. Click **OK**. The Band Math result appears in the Data Manager and it may also be displayed (if it is not displayed, right click on the Band Math result in the Data Manager and select **Load Grayscale**). Looking at the difference image, you should immediately see where the largest discrepancies occur in the two temperature estimates.

    Can you identify the features selected by the difference image? These are areas where the ocean surface is obstructed by clouds.

13. Right click on the **View** item in the Layer Manager and select **Remove all Layers**.

Any Band Math expressions you enter during an ENVI session are available throughout the session, and they appear in the Previous Band Math Expressions list of the Band Math dialog. You can save expressions that you use regularly to a file, so that you do not have to re-enter them in the Band Math dialog each time you need them. Give the current expression list a filename with an `.exp` extension and click Save in the Band Math dialog. Then, you can restore the expressions later when you need them.

Tips for Writing Expressions

In the first exercise, you produced useful results using a relatively simple mathematical expression. However, in some cases, you may require more complicated expressions. Depending on your familiarity with IDL, it may take some practice to find the best way to write a single expression to accomplish your goals. ENVI actually performs a quick test of Band Math expressions to see if they are valid, by using a 5 x 5 array of random data values ranging from 0 to 255. If you get an error when entering an expression and you are unsure of the problem, you may find it helpful to think of your expression in the following context:

```
IDL> result = expression
```

Note that data types are often a cause for concern. Two types of problems will be encountered regularly when performing band math; integer overflow and type specific division. Bring up the IDL workbench by clicking on its icon in the Windows start bar, then type in the following IDL expressions:

```
ENVI> print, 120B + 230B
   94
ENVI> print, 120 / 230
          0
```

In the first example, when adding 120 to 230 you might expect the results *should* give us 350. Instead, we get 94. Why is this the case? Because these two variables are of type byte (the B in the expression is a declaration of byte variable type), they are only allotted enough memory to store 256 unique values – 0 through 255. When an expression takes them beyond this range, the data type *overflows* and in so doing, wraps around to the lower end of its value. You can think about this as a car's odometer flipping over from 999999 to 000000 when driven one million miles. We get 94 from 350 – 256 (the number of unique values it overflows before flipping). So, 94 values after the data type has wrapped around.

In the second case, we can see that IDL defaults to floor division for integer types (floor division is just a fancy way of saying division that preserves whole numbers – the type of division with remainders you probably did in grade school). This ensures that an expression of integers always creates integers – an important feature of an optimized array-based mathematical language, but one that can throw new users off when they try to perform a normalized index on image data of integer values.

We can fix both of these problems by using conversion functions in IDL:

```
ENVI> print, long(120B) + 230B
         350
ENVI> print, float(120) / 230
         0.521739
```

In the first fix, we've promoted our byte data type to a long integer – making it so that our range of allowable values is much larger, allowing values from − 2,147,483,648 to 2,147,483,647! In the second fix, we've converted our data to a type that allows decimals – a floating point value. We also could have converted the first example to floating point to solve the overflow problem:

```
ENVI> print, float(120B) + 230B
          350.000
```

As such, if you 'd rather not worry about remembering all the details of which data types are which for band math expressions, using float() can be a good all-purpose fix. There are a few areas where this breaks down (e.g. converting really large integers with lots of degrees of precision, or adding very small and very large values together), but for band math calculations, it doesn't come up often.

Note: It is only necessary to convert one component of a simple expression. For more complex expressions, you will want to convert at least a single component of each sub expression. This has been done incorrectly in the first example below and correctly in the second example.

```
ENVI> print, float(120B) + 230B/120B
      121.000
ENVI> print, float(120B) + 230B/float(120B)
      121.917
```

If your expression works at the IDL command line as the expression portion of a statement, then it should generally work in Band Math or Spectral Math. In this context, you should see that you cannot use control statements like FOR loops or IF...THEN statements in a single Band Math expression. However, by taking advantage of IDL's powerful array operators, you can often accomplish the same tasks.

For example, if you wanted to assign all pixel values less than 30 to the smallest pixel value in an image, you do not need to loop through every pixel in the image and test its value, nor do you want to use IDL's WHERE function. Instead, you could use the following Band Math expression:

```
min(b1)*(b1 le 30) + (b1)*(b1 gt 30)
```

This expression works because the array operator statements (`b1 le 30`) and (`b1 gt 30`) are evaluated first as an array of the same size as `b1` filled with ones (for true) and zeros (for false). So, in the first part of the expression, the minimum value in `b1` is multiplied by an array of the same size as b1 that contains ones (where b1 is less than or equal to a pixel value of 30) and zeros everywhere else. In the next exercise, you will use this approach to make a composite image that represents temperature based on two different bands of the AVHRR file.

Exercise: Composite Image

1. In the **Data Manager**, right click on **SST image** and select **Load Grayscale**.
2. Click **Views** → **Two Horizontal Views** to open a second display. Select the second View, then in the **Data Manager**, right click on **AVHRR Band 5** and select **Load Grayscale**.
3. Link the two display groups by clicking **Views** → **Geo Link Views**. Click **Link All** then click **OK**.
4. Select **Cursor Value** from the menu then move your cursor over the large, dark feature near image location (1180, 1220) in file coordinates (displayed as **File:** in the **Cursor Value** dialog).

As you learned in Exercise 1, the estimated temperatures in the areas where the ocean is obscured by clouds differ markedly from those derived from the calibrated thermal band. The data values in the SST image are about -48 degrees Celsius. This is clearly not the correct ocean temperature.

The algorithm used to calculate SST does not accurately calculate cloud-top temperatures. For cloudy areas, a thermal band should better estimate SST. Using Band Math, you can selectively substitute the data values from one of the thermal bands into the SST band, producing a composite image of temperature. As an initial test, you will use a threshold value of 0 in the SST image for the substitution criterion.

5. From the ENVI Toolbox, select **Band Ratio** → **Band Math**.

6. Type the following in the **Enter an expression** field:

```
(b1)*(b1 gt 0.0) + (b2 - 273.16)*(b1 le 0.0)
```

7. Click **OK**. The Variables to Bands Pairings dialog appears.

8. The variable `B1 - [undefined]` is selected by default. Select **SST Image** in the **Available Bands List** of the Variables to Bands Pairings dialog.

9. Select `B2 - [undefined]`, then select **AVHRR Band 5** in the **Available Bands List** of the Variables to Bands Pairings dialog.

10. Enter an output filename of `sst_thermal_composite.dat`.

11. Click **OK**. The composite image appears in the **Data Manager** and will also load into whichever ever view is active (you can make a view active clicking on it in ENVI).

12. Move your cursor over the composite image. Note that data values in ocean areas unobscured by clouds are nearly identical to those from the SST image in Display #1. However, data values in cloudy regions are the same as those from the thermal band (AVHRR band 5). If your assumption is correct, the SST algorithm had significantly overestimated the cloud-top temperatures.

13. Click **Views→ One View**. Then right click on the View item in the Layer Manager and select **Remove**. Close the Cursor Value dialog.

Normalized Difference Indices

A common type of index used (especially for moderate resolution multispectral data) is the Normalized Difference Index. NDVI (Normalized Difference Vegetation Index) incorporates the Red and Near-Infrared bands, and is the most commonly known of these indices. As a result, it is available by default in many GIS and remote sensing software packages. Several others, such as the NDWI (Water Index), NBR (Normalized Burn Ratio) and NDBI (Built-up Index) have been proposed and used.

When Digital Globe's Worldview-2 was launched and began operations, the narrower bands allowed for new and alternative indices to those that were developed for Landsat MSS and TM sensors. In the next exercise, we will use the ability of band math to enter an expression once and re-use it to generate several Normalized Difference Images.

The formula for a generic Normalized Difference Image is as follows:

NDI = (Band1 – Band2) ÷ (Band1 + Band2)

For example, a normalized burn ratio index with Landsat would be:

NBR = (TM4 - TM7) ÷ (TM4 + TM7)

And the standard NDVI image could be expressed as:

NDVI = (NIR - RED) ÷ (NIR + RED)

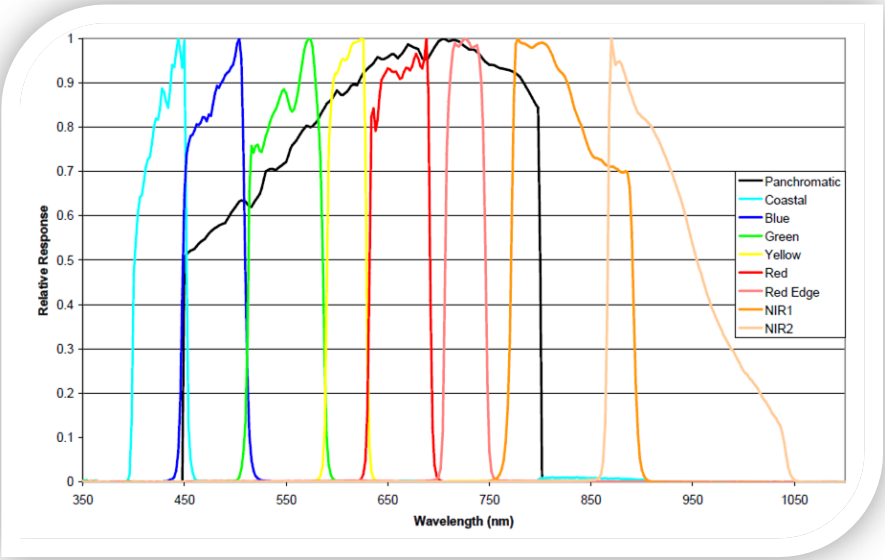
Therefore, we can represent this entire class of indices with the following band math expression:

```
(b1 - b2) / (b1 + b2)
```

Taking into account the data type concerns expressed in the **Tips for Writing Expressions** section, we can account for multiple data types by changing the expression to:

```
(float(b1)-b2)/(float(b1)+b2)
```

The spectra available in Worldview-2 (with Spectral Response) are (after Wolf 2010):



Wolf (2010) recommends the use of the following indices for WV-2 data:

——————— **= NDWI (Water)**

——————— **= NDVI (Vegetation)**

——————— **= NDSI (Soil)**

——————— **= NHFD (Non-homogenous Feature Index)**

Just a note in summary, the Bands typically available in WV-2 scenes and their corresponding wavelengths are:

Coastal	Band 1	427.000
Blue	Band 2	478.300
Green	Band 3	545.700
Yellow	Band 4	607.700
Red	Band 5	658.800
Red Edge	Band 6	724.100
NIR1	Band 7	832.900
NIR2	Band 8	949.300

Exercise: Generating a set of Normalized Difference Images from WV-2

1. In ENVI, select **File** → **Open** then navigate to `envidata\extend`, select `new_zeal_wv2_sub.dat` and click **Open**.
2. Click on the **Zoom To Full Extent** icon.
3. Go to the text box underneath the **Toolbox** dialog and begin typing "Band Math." Alternatively, navigate to **Band Ratio**→ **Band Math**. Double click the **Band Math** tool to start it.
4. Enter the formula for normalized difference indices described in the section previous.

```
(float(b1)-b2)/(float(b1)+b2)
```

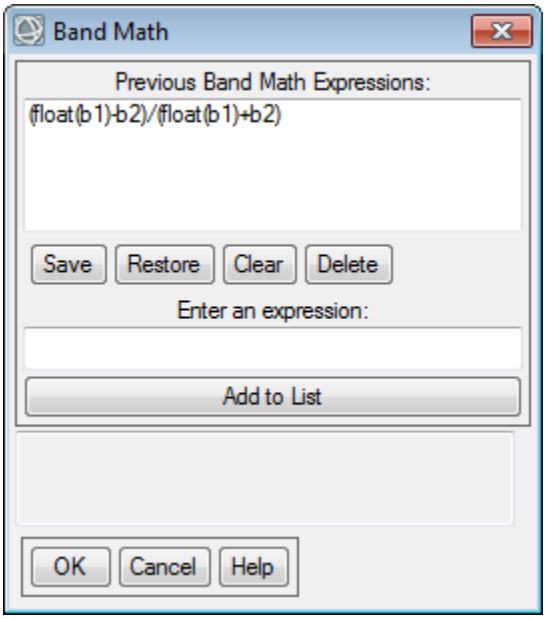
5. When the formula is entered, click **OK**.

6. The first index we'll calculate is Wolf's alternative NDWI. Map B1 to the Coastal Band by clicking **B1** and then clicking the first band (wavelength 427.00 micrometers). Then click **B2** followed by the NIR band 2 – the last band in our image.

7. For the output navigate to your **envidata\enviout** directory and name the file `new_zeal_wv2_ndwi.dat`, then click **OK**.

8. The result will be displayed. Adjust the transparency slider to investigate the relationship between the index and the original image. There are several false positives in shadowy areas and on asphalt, but as water is a low reflectance feature, these are expected from an index.

9. Double click on **Band Math** again. Notice that your previous expression is still stored:



Click on the expression. Save this expression by clicking **Save**. Save the file as norm_diff_ind.exp to your enviout directory then click **OK**. Now, you can still restore this same expression in a later session.

10. Click the expression, then click **OK**. We'll repeat step 4 from above for the three other indices by running band math three more times, mapping the variables in this way:

	B1	B2
NDVI (vegetation)	Band 8 (949.3)	Band 5 (658.8)
NSDI (soils)	Band 3 (545.0)	Band 4 (607.0)
NHFD (man-made features)	Band 6 (724.1)	Band 1 (427.0)

11. After finishing generating the bands, create four new views by clicking **Views** → **2x2 Views**.

12. Open the **Data Manager**.

13. Load each of the generated indices into one of the views by clicking on a **View** in the Layer Manager and then right clicking on an index in the Data Manager and selecting **Load Grayscale**. Then click the **Zoom To Full Extent** icon for each. If you want, load the `new_zeal_wv2_cal.dat` original image into each of the views and experiment with transparency in each of the images.

14. Close all but one of the Views and remove the images from the Layer Manager.

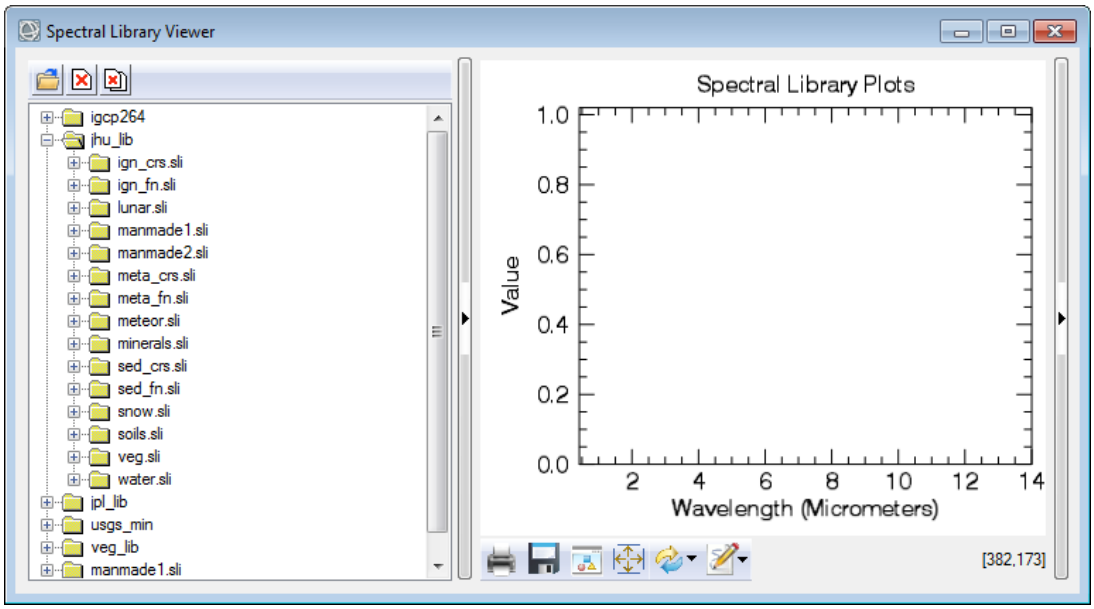
Spectral Math

Spectral math is used to customize spectra for use as endmembers in classification, spectral mixture analysis, and target detection. Spectra can be imported from either a pixel in a multi-band image (plotted in the Z-profile), a spectral library, an ASCII file, or an IDL variable. To perform Spectral Math, you will need to load each spectrum into a plot window prior to running the Spectral Math tool.

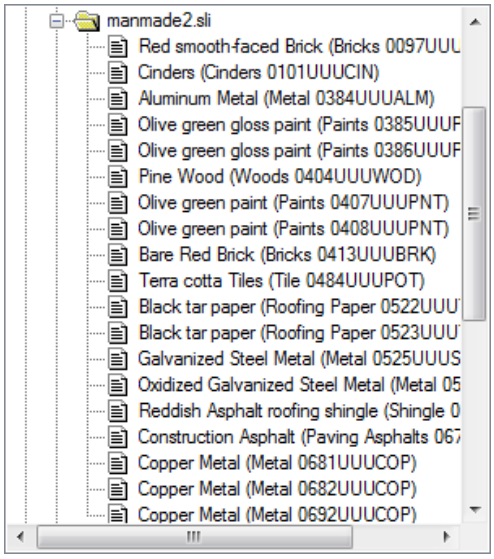
As with Band Math, you can save Spectral Math expressions to a file for future use, assign variables to files instead of individual spectra, and execute previously compiled IDL routines by calling them from the Spectral Math dialog. In this example, we will mix spectra by combining several known endmembers.

Exercise: Using Spectral Math to Create Mixed Spectra

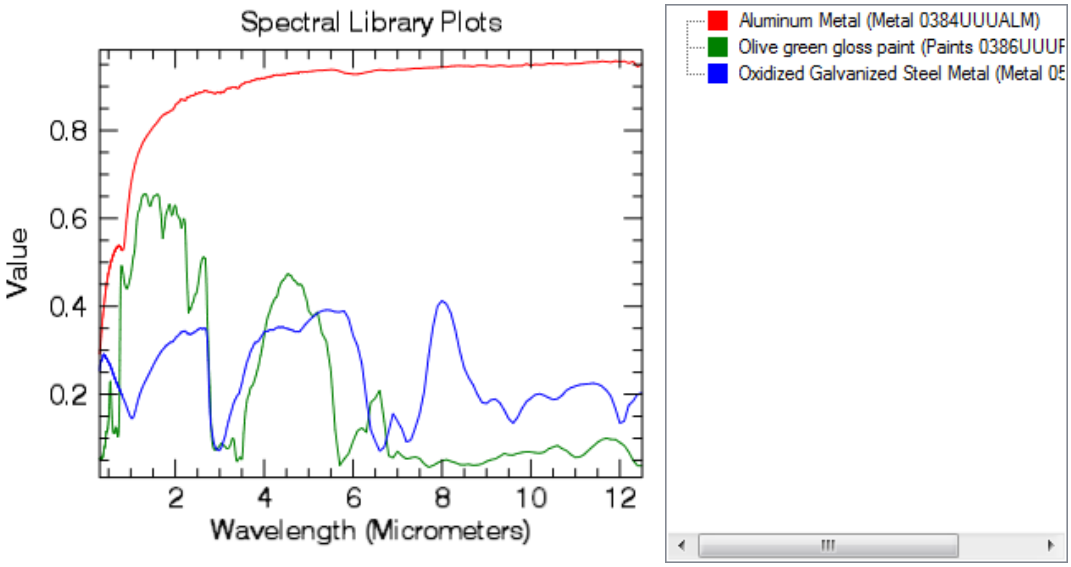
1. In ENVI, select **Display** → **Spectral Library Viewer**. Expand the `jhu_lib` folder. This folder contains spectral libraries from John Hopkins University.



2. Open the spectral library file `manmade2.sli` to see a list of spectra in the library.



3. Click on the library entries entitled **Aluminum Metal (0384UUUALM)**, **Olive Green Gloss Paint (Paints 0386UUUPNT)**, and **Oxidized Galvinized Steel Metal (Metal 0526UUUSTLa)**. These spectra will now be displayed in the Spectral Library Viewer. These plots are laboratory-measured reflectance spectra. The x-axis is wavelength and the y-axis is percent reflectance. Using spectral math, we will create a weighted average or mixed pixel spectrum comprised of these three materials.

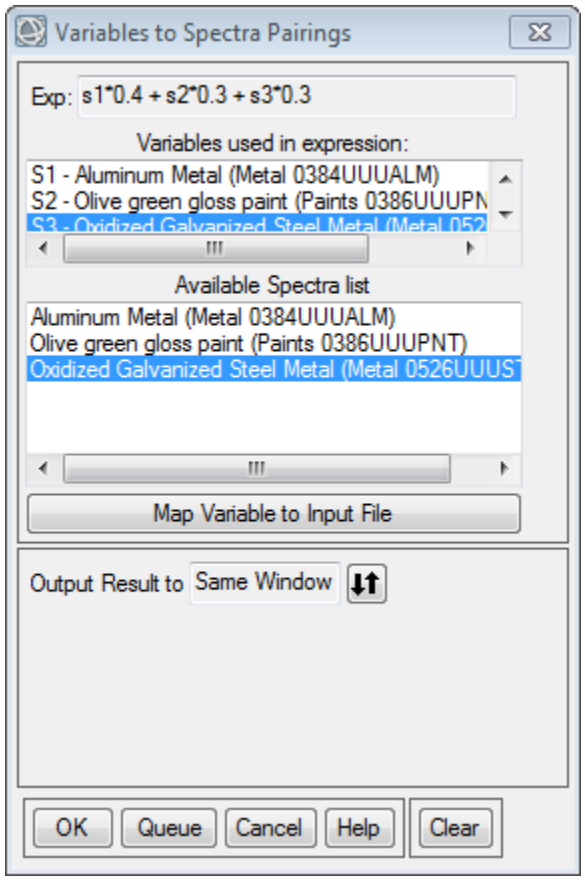


4. Locate the Spectral Math tool by typing **Spectral Math** into the Toolbox tool location dialog. Then, double click **Spectral Math** to launch the tool. In the **Enter an expression** text box, enter the expression:

```
s1 * 0.4 + s2*0.3 + s3*0.3
```

Note: Regardless of which spectra you will select, you always have to represent the spectra with the variables s1, s2, etc. in spectral math's expression dialog.

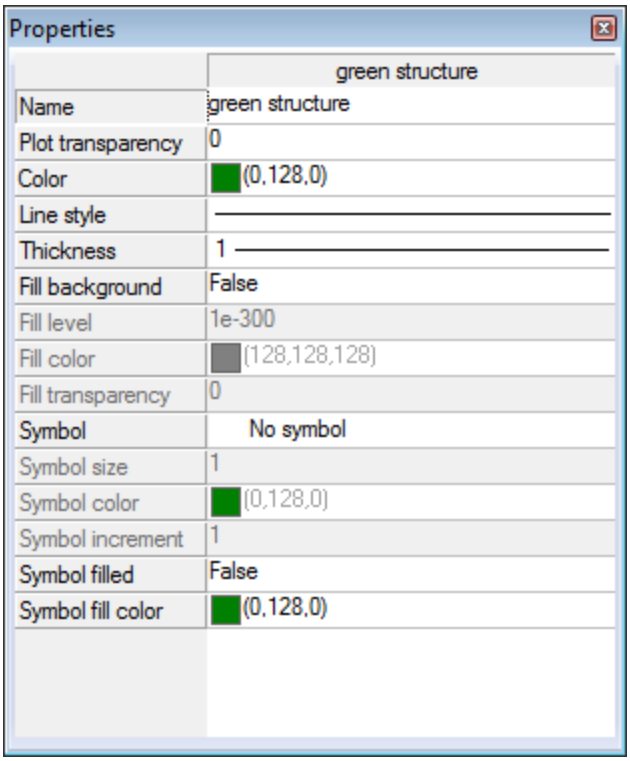
5. Click **OK**. You will be prompted to pair variables to spectra.



6. Match **S1** to **Aluminum Metal**, **S2** to **Olive Green Gloss Paint**, and **S3** to **Oxidized Galvanized Steel Metal**. To do this, first click the Variable to be used from the Variables used in expression: sub-dialog, then click the matching spectrum from the Available Spectra list.

7. You can toggle the Output Result to **New Window** or leave it to output to the **Same Window**. Same window will allow you to compare the resulting spectrum to its component spectra. Select **New Window**, then click **OK**.

8. The resulting spectrum will be loaded into a new window. Its name is currently the same as the expression we used to create it. Click the expand arrow in the dialog to view the legend. Double click on the title of the new plot, and click in the **Name** field, delete the current name, and enter g**reen structure.** You may also change the line style, thickness, or color of the plot if you so choose. Click the **X** to close the dialog when you are finished.



9. The finished spectrum can now be added to a spectral library or used as input for a spectral mapping routine. Close the Spectral Library Viewer and the spectral plot.

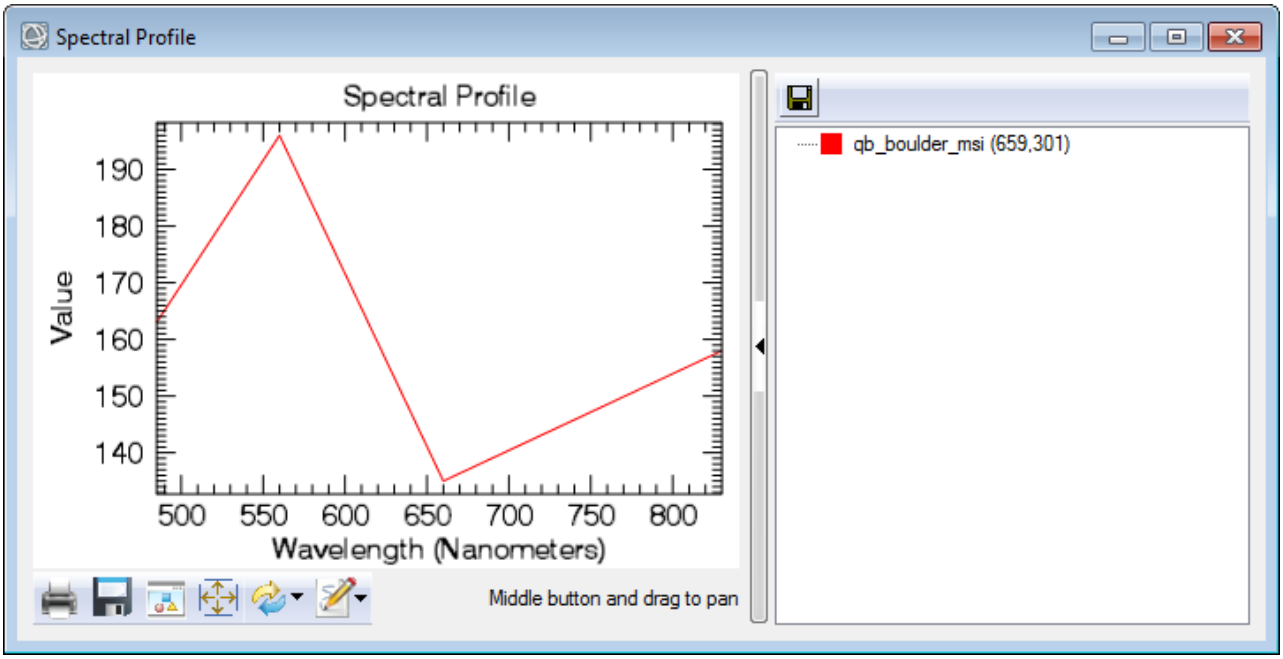
Exercise: Perform Simple Linear Subtraction to Remove Background

1. Open `qb_boulder_msi` from the ENVI installation location (**C:\Program Files\Exelis\ENVI50\data**).

2. Open a Spectral Profile by selecting **Display** → **Spectral Profile** from the menu.

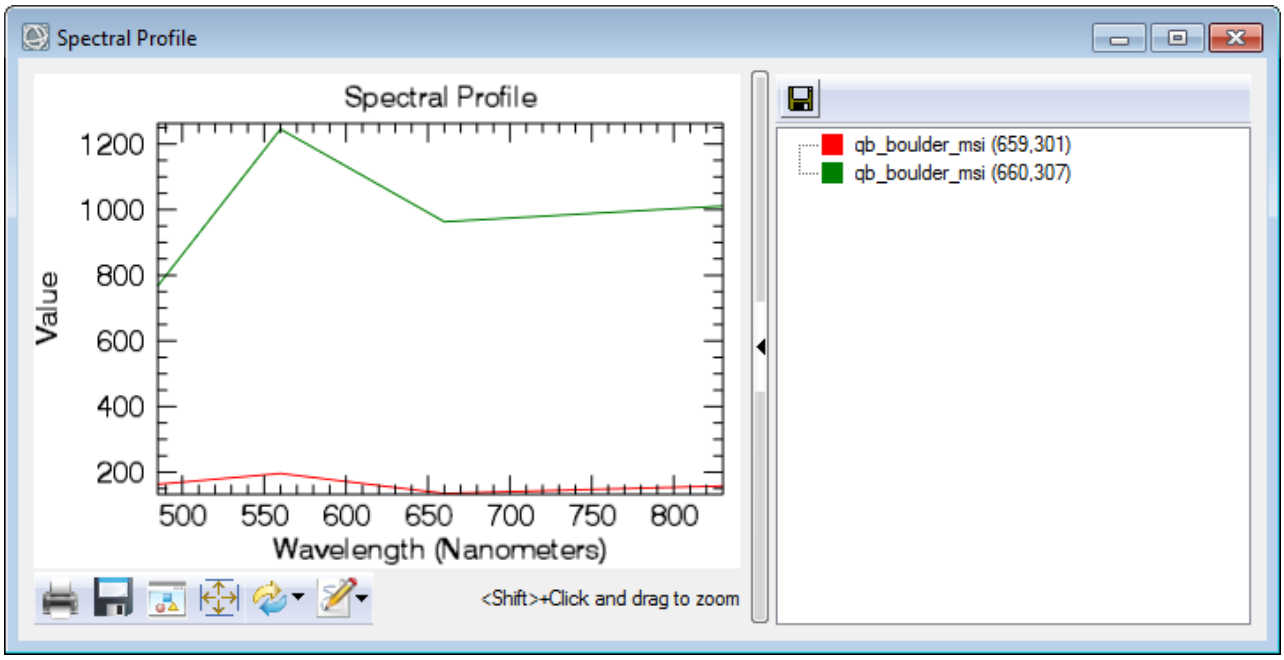
3. Navigate to the center of the image, near the large reservoir. Zoom into the bright building and click on a dark pixel in the shadow of the building (be very precise with the mouse click).

4. The spectrum will be loaded into the Spectral Profile dialog. Click on the right side of the dialog to open the legend. The pixel coordinates of the selected spectrum is its current title.

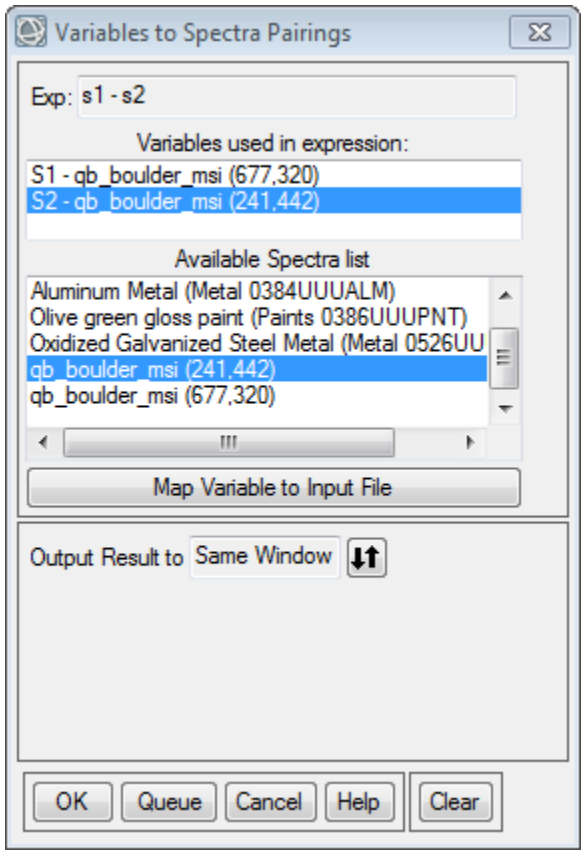


5. Now, SHIFT click and select a pixel from the bright building.

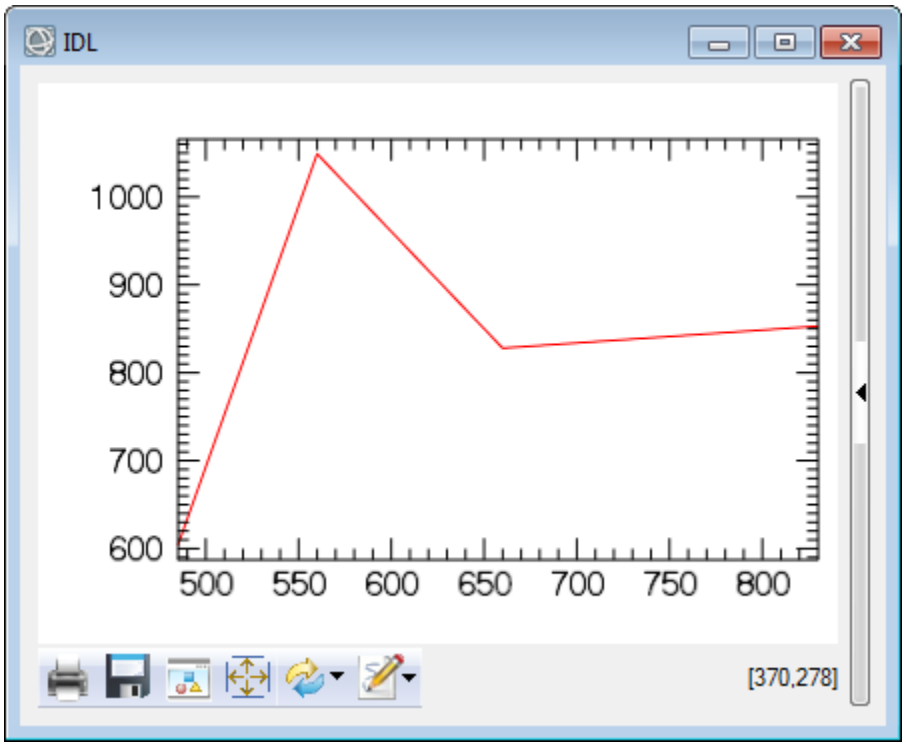
6. The spectral profile for the selected area will be shown in the Spectral Profile dialog as well, again containing pixel coordinates in its title.



7. While making sure to keep the Spectral Profile dialog open, run the **Spectral Math** tool. You will enter in a new expression: `s1 - s2`. Click **OK**.



8. Now map the variables **s1** and **s2** to the two spectra you selected from the image. The bright impervious surface should be mapped to **s1** and the dark shadow should be mapped to **s2**. When these spectra have been matched to their variables, click **OK**.



9. This resulting spectrum is a basic linear approximation of the signal from the chosen bright impervious pixel that is not accounted for by the basic background signal in the image.

   Note: This is similar to how the scattered-light correction method **dark subtraction** functions, except that it applies this subtraction over the entire scene.

10. To more easily compare the spectra, expand the legend of the new plot and then drag the label for the spectral math result into the legend of the first plot.

11. Close all plots and remove all images from the display.

Chapter 8 ENVI Library Routines

Like IDL, ENVI has a library of built-in routines. There are over 200 documented routines in the ENVI library. They encompass almost all of the functionality provided in the interactive ENVI program. Each library routine is written in the form of an IDL procedure or function, and is called as such. A complete index of these routines—as well as a full reference page for each—is located in the ENVI Help under **Programming > Routines**.

The ENVI API is currently split between the older Procedural API and the newer Object API. This chapter covers the basics of using the procedural API (and also, therefore, the version of the API most older code will be written in). Chapter 9 delves into the basics of the Object API.

Keywords common to ENVI routines

There are a few keywords that are used in nearly every ENVI library routine. These keywords refer to standard types of information about files and the data contained within them.

Recurring keywords in the ENVI library

Keyword	Description
FID	A file identifier, or FID, is a long integer value used by ENVI to identify a file. All access and processing of a file is accomplished by referring to the file's FID. The FID is provided to the ENVI programmer as a named variable by one of several ENVI routines for opening or selecting a file. Note that ENVI, unlike IDL, does not require a logical unit number (LUN) to access a file. A FID is not equivalent to a LUN.
R_FID	ENVI processing routines that return new images have an R_FID , or "returned FID" keyword. If results are saved to memory, setting the R_FID keyword is the only method for accessing the data.
M_FID	Processing routines that allow masking will have a M_FID , or "mask FID" keyword for specifying the file to use as the mask band.
DIMS	The DIMS (dimensions) keyword uses a five-element long integer array that defines the spatial subset of a file or an array to be used for processing. When an FID is specified, DIMS will almost always be needed. The elements of the dims vector are an id for an open region of interest, otherwise set to -1the start sample number (zero-based)the end sample number the start line number the end line number
POS	The POS (position) keyword defines the band positions to be used for processing. It uses a long integer array of variable length. Bands are specified starting at zero (band 1 = 0, band 2 = 1, etc.). For example, to process only bands 3 and 4 of a multiband file, set POS = [2, 3].

Working with files

There are several ENVI routines for selecting and opening files, requiring varying degrees of user interaction.

ENVI library routines for working with files

Routine	Description
ENVI_PICK_FILE	This routine produces a dialog (the same as selecting File > Open Image File from the ENVI menu) that prompts the user to choose a file on disk.This routine doesn't actually open the file; it returns the fully qualified pathname of the file as a string.
ENVI_SELECT	This routine presents a dialog that prompts the user to select a file from among those that have already been opened. ENVI_SELECT produces ENVI's standard file selection dialog, including working buttons for spatial and spectral subsetting, as well as for choosing a mask band, if desired. This routine incorporates ENVI_PICKFILE through a button to open ENVI format files from disk. ENVI_SELECT returns FID, DIMS and POS for the selected file.
ENVI_OPEN_FILE	This routine returns a FID without user interaction. It is the simplest way to open ENVI-format files. It requires no user interaction.
ENVI_OPEN_DATA_FILE	This routine has a keyword for each type of external file format that ENVI can read. It returns a FID . It requires no user interaction.
ENVI_GET_FILE_IDS	This routine returns all currently open and valid FIDs.
ENVI_FILE_MNG	This routine allows opened files to be closed (i.e., removed from memory) and/or deleted from the disk. No user interaction is required.The DELETE keyword to this routine should be used with extreme caution—files deleted in this manner can not be recovered.

Here's an example of opening an ENVI-format file using ENVI library routines.

Call ENVI_PICKFILE from the ENVI> command line:

```
ENVI> filename = envi_pickfile(title='Open Boulder Data')
```

This dialog window is the same one you see when you open a file in a normal ENVI session. All of the ENVI library routines that include a GUI use the same interfaces as interactive ENVI. Navigate to the extending directory and select the file boulderTM.sub. Click OK to return to the ENVI command line. Display the full path to the boulderTM.sub file with

```
ENVI> help, filename
FILENAME STRING = 'C:\IDL_coursefiles\extending \data\boulderTM.sub'
```

Use ENVI_OPEN_FILE to open this file. At the ENVI command line,

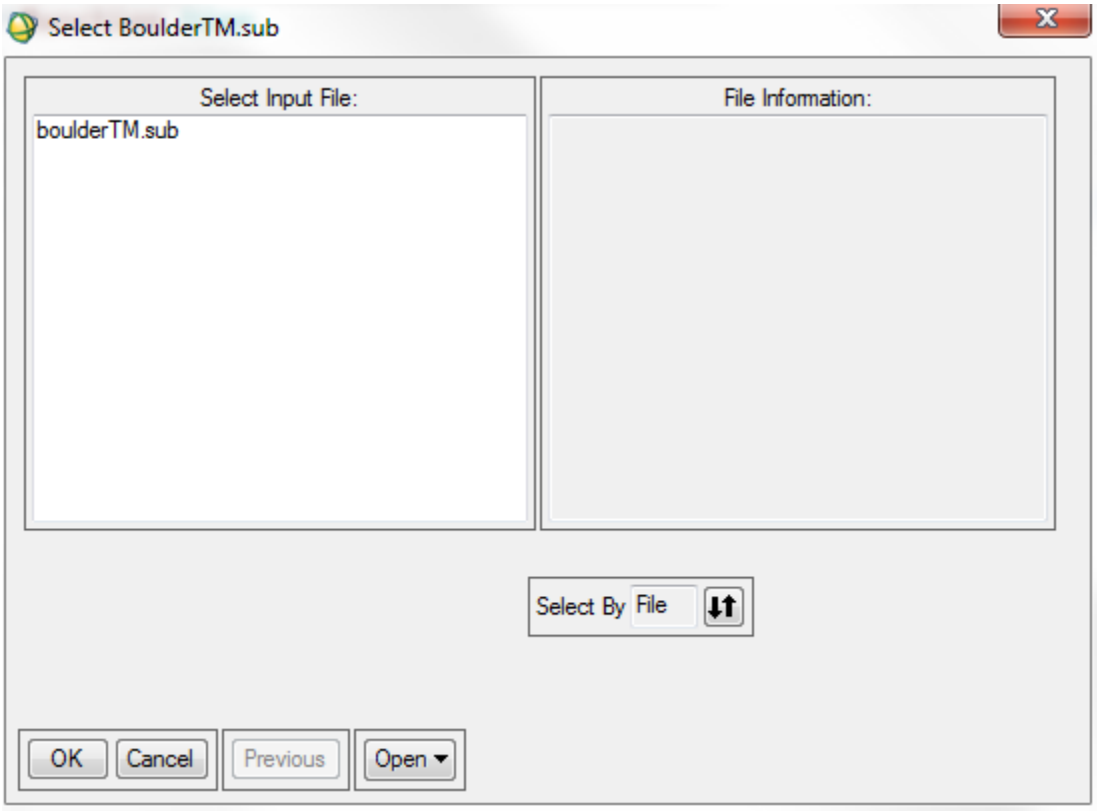
```
ENVI> envi_open_file, filename, r_fid=bldr_fid
```

The target for the R_FID keyword, the variable bldr_fid , holds the FID for the opened file. Use HELP to check that the bldr_fid variable is set to a positive long integer scalar. If there was a problem opening or accessing the file, the FID is set to a value of -1. FIDs with a value of -1 are invalid and can't be processed. Like logical unit numbers (LUNs) in IDL, the exact value of a valid FID is not important.

However, because files are referred to using the name of the variable that contains the FID (in this case bldr_fid), be sure to give a descriptive name to the FID variable to make it easy to identify the file.

Here's another technique for opening the same file. Use ENVI_SELECT to return the FID for the image into a variable called bldr_fid2 . Also use the DIMS and POS keywords to collect the dimensions and band positions of the image.

```
ENVI> envi_select, dims=bldr_dims, fid=bldr_fid2, $
> pos=bldr_pos, title='Select BoulderTM.sub'
```



When the ENVI_SELECT dialog window is on screen and you have selected the BoulderTM image, make note of its dimensions displayed in the right panel of the window. All of the information that the user could input into this file selection dialog window (e.g., the FID of a mask file) can be returned by the ENVI_SELECT procedure. See the Online Help reference page for a full list of available keywords.

Select OK to return to the ENVI command line. Two variables now hold FIDs that refer to the same image file. Do the FIDs have the same value? Check them with the HELP procedure:

```
ENVI> help, bldr_fid, bldr_fid2
BLDR_FID        LONG      =            2
BLDR_FID2       LONG      =            2
```

The two FID s (bldr_fid and bldr_fid2) are identical because they refer to the same file. Once a file has been opened in a given ENVI session, it will always have the same FID regardless of the routine that is used to open or select the file.

Recall that DIMS and POS are used to specify the spatial and spectral subsets for processing. Use the PRINT procedure to see the values in the arrays:

```
ENVI> print, 'dims=', bldr_dims
dims=        -1         0       499         0         599
ENVI> print, 'pos=', bldr_pos
pos=         0         1         2         3         4           5
```

Can you relate the values of the DIMS and POS variables to the image that was selected?

The DIMS variable will always be a 5-element long integer vector where the first element indicates if an ROI was used to define the spatial subsetting, and the remainder of the elements represent the IDL array subscripts for the first sample, the last sample, and the first line and the last line respectively. The POS variable represents the band positions and is also a long array. It will contain as many elements as bands that were selected. The band positions are identified with zero-based (i.e., IDL) subscripts. Because we did not spectrally subset the file when it was selected, the POS variable has 6 elements with values ranging from zero to five. If the file was spatially or spectrally subsetted when selected, the returned DIMS and POS variables would reflect these actions.

Exercise

Use ENVI_SELECT again to select the Boulder TM image, but this time spatially and spectrally subset the image. Print out the DIMS and POS values and note how they've changed to reflect the subsetting.

In most cases it's not necessary to manually close a file once it has been opened because the opened file is not actually consuming any memory. However, when you want to remove a file's FID from an ENVI session, use the ENVI_FILE_MNG routine:

```
ENVI> envi_file_mng, id=bldr_fid, /remove
```

To see if the last command worked, use ENVI_GET_FILE_IDS to see if there are any open FIDs:

```
ENVI> open_ids = envi_get_file_ids()
ENVI> print, open_ids
      -1
```

The returned value of-1 confirms that there are no currently open files.

Note that the two variables which held the FIDs (bldr_fid and bldr_fid2) still exist and retain their original values, but using them to refer to a file will cause an error.

Reading the ENVI header

In order to work with an image, you'll usually need to know some basic information about it, including the number of samples, lines and bands, the data type, file type, and any associated map information. In an ENVI-format file, all of this information is stored in the ENVI header file.

The ENVI_FILE_QUERY routine automatically parses information out of the ENVI header file and returns selected information through keywords into IDL variables. For supported external file formats, ENVI_FILE_QUERY parses the information from the file as appropriate.

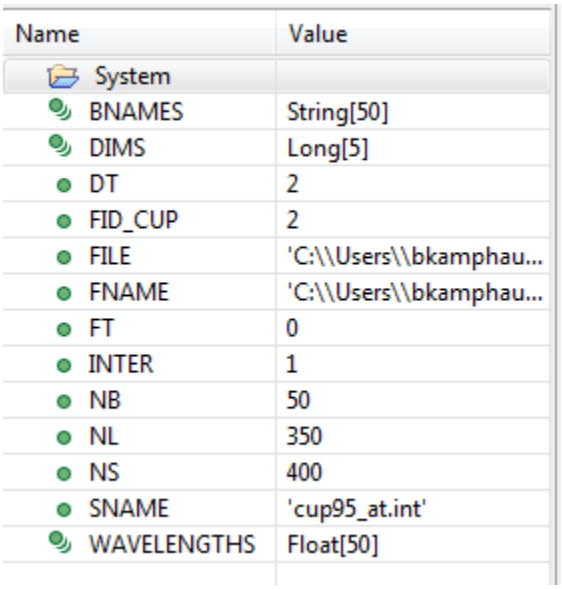
For reference, locate the entry for ENVI_FILE_QUERY in the Online Help. Note that in this routine the FID is passed as a positional parameter, not a keyword. From the command line, use ENVI_PICKFILE and ENVI_OPEN_FILE to open the file CUP95_AT.INT in the extending directory of the course files. This is an AVIRIS hyperspectral scene of Cuprite, NV. Name the FID variable fid_cup :

```
ENVI> file = envi_pickfile()
ENVI> envi_open_file, file, r_fid=fid_cup, /no_realize
ENVI> help, fid_cup
FID_CUP         LONG      =            3
```

Remember, if fid_cup is-1, then there was an error opening the file. The NO_REALIZE keyword prevents the Available Bands List from appearing. Use ENVI_FILE_QUERY to return some basic information about the image from its ENVI header file:

```
ENVI> envi_file_query, fid_cup, ns=ns, nl=nl, nb=nb, data_type=dt,$
> interleave=inter, fname=fname, sname=sname, bnames=bnames, $
> wl=wavelengths, file_type=ft, dims=dims
```

Use the IDL Workbench Variables view to examine the values that were returned.



The dt (data type), inter (interleave), and ft (file type) variables return indices corresponding to a lookup table for these parameters. A data type of 2 is short integer, an interleave of 1 means BIL, and a file type of 0 means ENVI Standard file. The Online Help page for ENVI_FILE_QUERY has a complete list of these lookup table values. The variables returned by ENVI_FILE_QUERY can be used like any other IDL variable. For example, wavelengths is a floating point array of 50 elements, where each element is the wavelength (in microns) of the corresponding band of the Cuprite AVIRIS image.

The variable bnames also has 50 elements, where each element is the name of the corresponding image band. Use IDL's WHERE function to define a selected set of AVIRIS bands in the shortwave infrared part of the spectrum:

```
ENVI> i_swir_bands = where((wavelengths ge 2.20) and $
> (wavelengths lt 2.25), n_swir_bands)
```

Print the number of shortwave infrared bands and their names:

```
ENVI> print, 'Number of shortwave infrared bands:', n_swir_bands
Number of shortwave infrared bands:           5
ENVI> print, bnames[i_swir_bands]
Band 193 Band 194 Band 195 Band 196 Band 197
```

To process the entire Cuprite scene (that is, with no spatial subsetting), but spectrally limit the processing to the shortwave infrared bands defined above, set the POS variable appropriately:

```
ENVI> swir_pos = i_swir_bands
```

To run the entire process above in line, we would have simply entered:

```
ENVI> swir_pos = where((wavelengths ge 2.20) and (wavelengths lt 2.25))
```

We'll use the band positions in swir_pos to extract data from the Cuprite scene in the next section.

Accessing image data

When image files are large—as is often the case with remote sensing data—it is not advisable (and often not even possible) to read an entire file into memory. Instead, ENVI provides two routines for reading image data in a logical organization—either by band or by spectral slice (for routines that read the entire image into memory, see the Object API in the next chapter).

Routines for reading image data from a file.

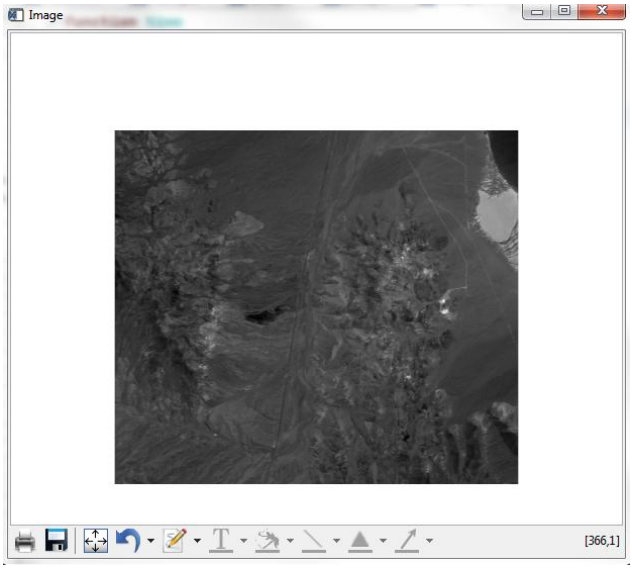
Routine	Description
ENVI_GET_DATA	This function is designed to retrieve spatial image data, one band at a time. If spatial data from more than one band are required, then this routine must be called multiple times.The FID, DIMS and POS keywords are required when calling this routine.
ENVI_GET_SLICE	This function is designed to retrieve spectral image data. It returns data from all of the image bands for one specified line, for any number of samples in that line. The data can be returned in either BIP or BIL storage order.

Let's read spatial data for the selected shortwave infrared bands in the Cuprite AVIRIS scene into IDL variables. We'll need the file identifier fid_cup , the spatial dimensions of the scene in the dims vector and the band positions in swir_pos . Read the data for the first band specified in swir_pos (Band 193) by entering these statements at the IDL Workbench command line:

```
ENVI> band193 = envi_get_data(fid=fid_cup, dims=dims, $
> pos=swir_pos[0])
ENVI> help, band193
BAND193         INT       = Array[400, 350]
```

The ENVI processing routines for reading data automatically account for the data type and interleave, producing a variable of the correct size and type. In your ENVI session, use File > Open Image File to open the Cuprite image (CUP95_AT.INT in the extending directory) and display band 193 in a new window. For comparison, display the band193 variable we just read in a new IDL window. At the Workbench command line, enter:

```
ENVI> img = image(band193)
```



Put the main image window from the ENVI display next to the IDL window. How do the two image displays differ? If the data are the same, why don't they look the same?

ENVI and IDL have different display defaults for images. ENVI uses the remote sensing convention of drawing images line-by-line from the top down, starting at the upper left. IDL uses the Cartesian coordinates convention of drawing from the bottom up, starting at the lower left. We can make IDL draw images like ENVI by setting the ORDER keyword to TVSCL. To enhance contrast, a two percent linear stretch is automatically applied to images displayed in ENVI. IDL's IMAGE function automatically bytescales data, which is equivalent to a zero percent linear stretch. To make the IDL display look like ENVI's, we could use the ENVI STRETCH_DOIT routine to apply a two percent linear stretch to the data, then display those results instead of the original image data. With ENVI_GET_DATA, you can read and process bands from any location within a file without having to manually position the file pointer. For example, to read a different shortwave IR band and compute its data range, execute the following statements:

```
ENVI> band = envi_get_data(fid=fid_cup, dims=dims, $
> pos=swir_pos[3])
ENVI> help, band
BAND            INT       = Array[400, 350]
ENVI> print, bnames[swir_pos[3]], max(band), min(band)
Band 196     644     129
```

How would you access each of the near-IR bands sequentially and compute their data ranges? You could repeat these statements for each band, but a better way exists. This topic is addressed in two chapters under the topic, "Batch Mode Programming."

A spectral slice can be extracted from a multiband image horizontally (all of the bands for a single line of the image), vertically (all of the bands for a single pixel column of the image) or arbitrarily (you define the direction by selecting an ROI polyline).

Use ENVI_GET_SLICE to read a complete spectral slice for the first line of the AVIRIS data and return the slice in BIL format. Note that you can generate a POS vector on demand with the IDL LINDGEN function:

```
ENVI> all_bands = lindgen(nb)
ENVI> slice1_bil = envi_get_slice(fid=fid_cup, pos=all_bands, $
> line=0, /bil)
```

What are the dimensions of slice1_bil (remember, the image dimensions are 400 samples, 350 lines and 50 bands)? Use HELP to find the answer:

```
ENVI> help, slice1_bil
SLICE1_BIL      INT       = Array[400, 50]
```

A BIL slice is dimensioned ns x nb . A BIP slice is the transpose of the BIL slice, or nb x ns . Try returning the same slice with a BIP interleave:

```
ENVI> slice1_bip = envi_get_slice(fid=fid_cup, pos=all_bands, $
> line=0, /bip)
ENVI> help, slice1_bip
SLICE1_BIP      INT       = Array[50, 400]
```

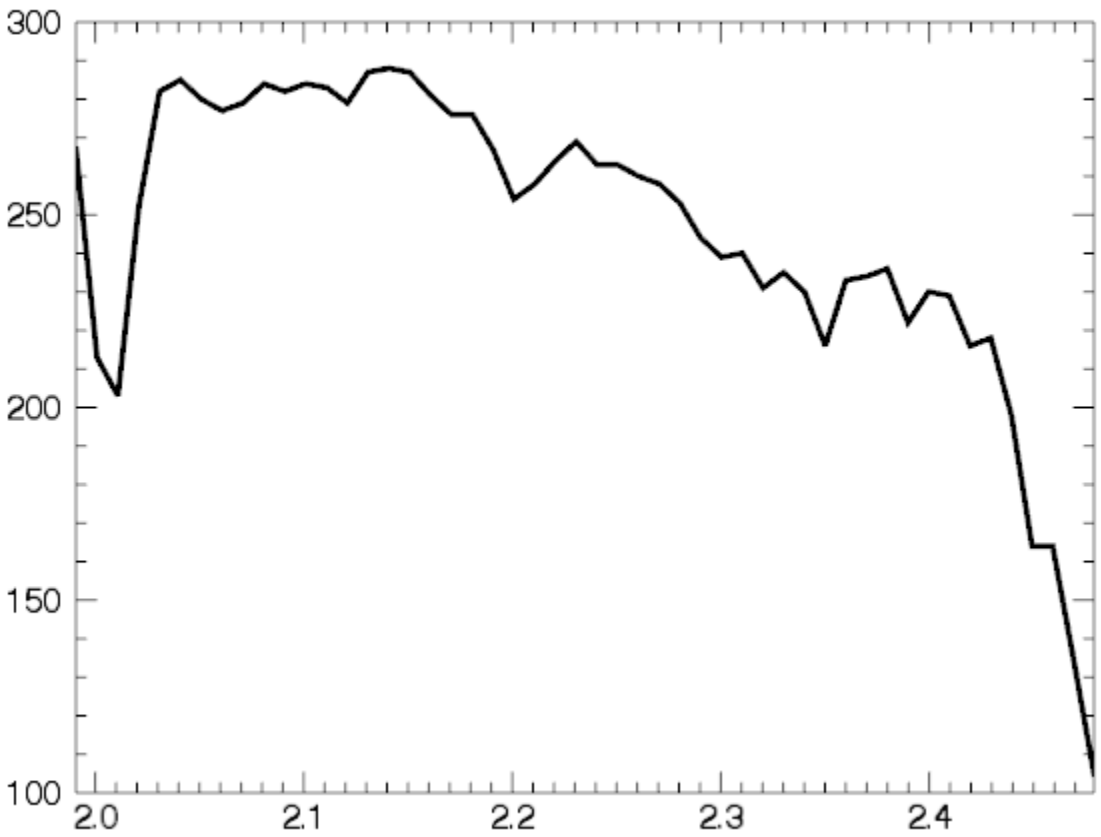
Note that we could also use the IDL TRANSPOSE function to get the same result. Verify the spectral slices are the same using IDL's ARRAY_EQUAL function:

```
ENVI> print, array_equal(slice1_bil, transpose(slice1_bip))
   1
```

Plot the spectrum for the first pixel in the image:

```
ENVI> p = plot(wavelengths, slice1_bip[*,0], thick=3, /xstyle)
```

Recall that IDL array subscripts start at index zero.



Regions of interest (ROI)

Regions of interest (ROI) are a powerful and flexible way to collect and process data. Their use is integral to many functions within ENVI. Before learning how to work with ROIs using the ENVI library routines, it is important to understand exactly what an ROI file contains. ROI files do not contain image data, they simply contain the addresses of the pixels that are part of the ROI. These pixel addresses are subscripts into the IDL array that contains the image. The ROI pixel addresses are stored using the single-index method, not the more intuitive dimensional (row, column) or (sample, line) subscripting method. To convert the single-index subscripts to (sample, line) locations, the image dimensions must be known. This means an ROI file is inherently associated with images of a certain spatial dimension—the dimensions of the image from which the ROI was defined. This property makes ROI files quite versatile as they can be applied to any image whose spatial dimensions are the same as the file from which the ROI was defined.

Working with ROIs is a multi-step process. The ROI files must first be restored from disk. Then, because any given ROI file can contain multiple ROIs, you must obtain a unique ROI ID for the ROI you wish to use. The ROI ID is similar in usage to the FID. Once you have the ROI ID, the ROI data can be collected from an open image file, or the ROI's pixel addresses can be returned into a variable.

The ENVI library routines for creating and working with ROIs are listed below:

ENVI library routines for working with regions of interest (ROI).

Routine	Purpose
ENVI_CREATE_ROI	Used to create a new ROI in memory and define its attributes, such as the color, name, and spatial dimensions. This routine returns the ROI ID for the new ROI. Pixels are added to the new ROI using the ENVI_DEFINE_ROI routine. The newly created ROI exists only in memory unless the ROI is saved.
ENVI_DEFINE_ROI	Used to add objects to an ROI. Points, polygons, and polylines can be added. The additions are made only in memory unless they are subsequently saved with ENVI_SAVE_ROIS .
ENVI_SAVE_ROIS	Saves ROIs that have been updated.
ENVI_DELETE_ROIS	Deletes ROIs from memory, removing the ROI ID from the current session.
ENVI_RESTORE_ROIS	ROIS Restores previously saved ROIs into memory.
ENVI_GET_ROI_IDS	Retrieves the IDs of all the currently restored ROIs. This function is used in a manner similar to ENVI_GET_FILE_IDS
ENVI_GET_ROI	Returns the array addresses of pixels in a specified ROI as single-index subscripts. Recall that ROIs themselves do not contain any data, they are just a list of pixel addresses that reference an image file.
ENVI_GET_ROI_DATA	Returns the image data associated with the specified ROI.
ENVI_GET_ROI_DIMS_PTR	Used to obtain a pointer to a specified ROI that is used as the first element of the DIMS variable. When the ROI pointer is used in the DIMS definition, ENVI spatially subsets the image file using the ROI instead of a range of samples and lines.

Make an ROI file in ENVI

Let's start by opening a second ENVI session, displaying an image file and defining a few ROIs.

1. In the new ENVI session, open the image file boulderTM.sub , located in the extending directory.
2. Display a false-color RGB composite using bands (4, 3, 2) as red, green and blue respectively.
3. There are four features that are relatively easy to identify in this color infrared composite:
    - Cultivated vegetation is bright red.
    - Urban areas are cyan.
    - Clouds are white,.
    - Water (such as lakes and reservoirs) are nearly black.
4. Using the ROI Tool (accessible from Display Menu > Overlay > Region of Interest...) define four ROIs representing these four cover types.
   Some hints:
    - There's a sizeable reservoir at about (450,130); name this ROI water.
    - Just above or below the reservoir is vegetation, colored bright red. Name this ROI veg .
    - A large cloud is at about (40,350). Name this ROI clouds .
    - The largest concentration of urban area is in the lower right of the image, at about (385,370). Name this ROI urban .
    - You can use any combination of ROI types; e.g., polygons, points, etc.
5. Save the ROIs in the Default project directory as `boulder.roi` .
6. Close this ENVI session.

Opening ROI files programmatically

To collect ROI data from an image file, the image file must be open in ENVI.

Using FILE_WHICH and ENVI_OPEN_FILE , open the boulderTM.sub image that's in the extending directory:

```
ENVI> bldr_file = file_which('boulderTM.sub')
ENVI> help, bldr_file
BLDR_FILE STRING = 'C:\IDL_coursefiles \extending\data\boulderTM.sub'
ENVI> envi_open_file, bldr_file, r_fid=bldr_fid
```

Remember, if bldr_fid has a value of -1 then there was an error in opening the file!

Next, restore the ROI file from disk.

Using ENVI_PICKFILE or FILE_WHICH, get the path to the ROI file we just created (boulder.roi). Then, use ENVI_RESTORE_ROIS to restore the four ROIs from file.

```
ENVI> roi_file = envi_pickfile();Use for GUI
ENVI> roi_file = file_which('boulder.roi') ;Use for no interaction
ENVI> envi_restore_rois, roi_file
```

Do not repeatedly restore ROI files using the ENVI_RESTORE_ROIS routine! You will get multiple IDs for the same ROI. While this doesn't cause any harm, it does create needless confusion.

Next, we need the ROI IDs for the ROIs in the file. The routine ENVI_GET_ROI_IDS is used to do this; however, there are several different ways to use this routine depending on the number of ROIs that have been restored or created in the current session. If ENVI_GET_ROI_IDS is used without specifying any parameters, all ROIs that are currently opened in memory will be returned, regardless of the spatial dimensions on which they are based. Thus, it is common to specify the spatial dimensions when getting the IDs so that only those ROIs that can be applied to an image of that size will be returned. Because we are interested in the ROIs for the Boulder TM scene, we can specify this image's size when getting the ROI IDs:

Use ENVI_FILE_QUERY to get the number of samples and lines for the Boulder TM scene:

```
ENVI> envi_file_query, bldr_fid, ns=ns, nl=nl
```

Next, get the ROI IDs:

```
ENVI> roi_ids = envi_get_roi_ids(ns=ns, nl=nl, roi_names=roi_names, $
> /short_name)
```

What did we get for the ROI IDs? Use PRINT to see their values:

```
ENVI> print, roi_ids
           4           5           6           7
```

The value of the ROI ID depends on the number of ROIs that have already been opened in the current session. Like FIDs, their exact numeric value is not important. Each ROI ID refers to a different ROI (even if the ROIs are stored in the same file). To figure out which ROI ID is associated with which ROI, use the names of the ROIs that were returned when we got the IDs:

```
ENVI> print, roi_names
water veg clouds urban
```

The order of the ROI names is the same as the order of the ROI IDs, so the first element in roi_ids corresponds to the ROI whose name is the first element in roi_names . In the example above, roi_ids[0] is the ROI ID for the water ROI.

Get the ROI pixel addresses

To get the addresses of the pixels that are in the veg ROI, use ENVI_GET_ROI with the ROI ID for veg. In this example, since we have only four ROIs open, we can simply look at the list of the ROI names to see that the ROI ID for the veg ROI is the second element in roi_ids. However, if there were many ROIs open, or the ROI ID for a specific ROI needed to be determined at run time, you could take a more sophisticated approach, using the IDL WHERE function.

Enter the following at the ENVI command line:

```
ENVI> i_veg_id = where(strlowcase(roi_names) eq 'veg', n_veg)
ENVI> veg_id = roi_ids[i_veg_id]
```

The STRLOWCASE function is used to force the ROI names to be lower case. This is important because whenever strings are compared it is actually the ASCII codes for the strings that are compared, and the ASCII codes for lowercase are different than those for uppercase characters. Get the ROI pixel addresses for the veg ROI:

```
ENVI> i_veg = envi_get_roi(veg_id)
```

From the Variables view in the Workbench you can see that i_veg is a vector. It has one element for each pixel in the veg ROI. Each element is a single-index subscript referencing a location in the boulderTM.sub image. Print the subscript for the first pixel in the veg ROI:

```
ENVI> print, i_veg[0]
      108910
```

To convert this single-index address into a (column, row) address, you need to know the number of samples for the image from which the ROI was defined. Convert the single-index subscripts into dimensional subscripts using the IDL ARRAY_INDICES function:

```
ENVI> i_veg0 = array_indices([ns,nl], i_veg[0], /dimensions)
% Compiled module: ARRAY_INDICES.
ENVI> print, i_veg0
         410         217
```

Go to your ENVI session and display the boulderTM image. Bring up the the ROI Tool; this should display the ROIs in color in the Display Window. To confirm that this pixel address is indeed in the veg ROI, use the Pixel Locator (under the Display's Tools menu) place the cursor at the address of the first pixel in the veg ROI. Did it work? Remember, the ROI's pixel addresses are IDL subscripts in columns and rows. All IDL subscripts start at zero (they are 'zero-based' numbers). However, ENVI's interactive image coordinates are 'one-based.' For example, the upper left corner of an image in ENVI has image coordinates [1,1] even though this corresponds to element [0,0] in an IDL array that contains the image data. So, to convert the ROI addresses into the equivalent interactive cursor location in ENVI, you have to add 1 to the computed (column, row) address. Add one to each element of i_veg0 to convert it to ENVI image coordinates:

```
ENVI> print, i_veg0 + 1
         411         218
```

Using the Pixel Locator in the ENVI Display menu, confirm that this pixel is part of the veg ROI.

Get ROI data

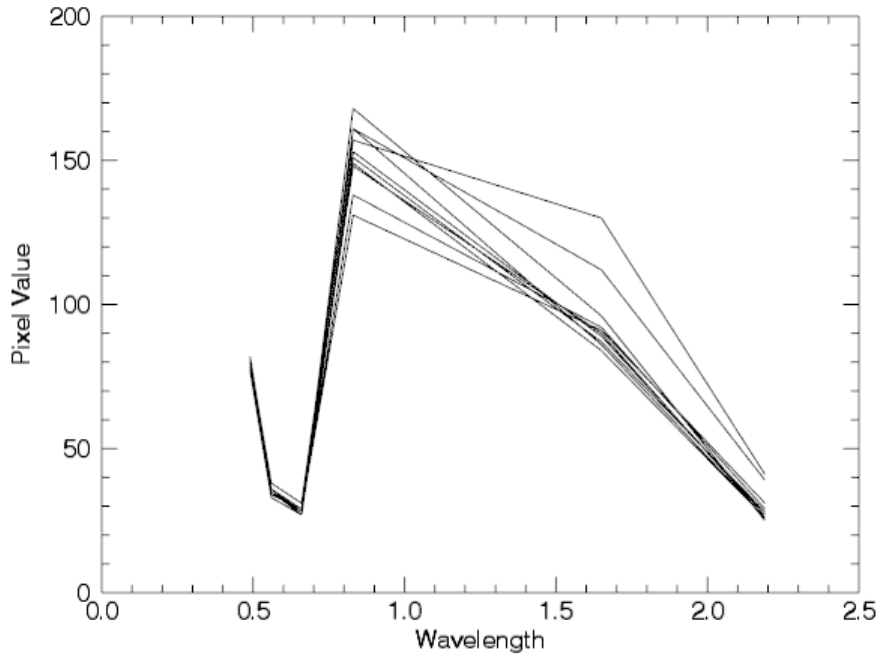
Next, we can use the routine ENVI_GET_ROI_DATA to collect the data associated with the veg ROI. This routine requires that we specify the FID of the image file from which data are to be extracted, as well as which bands to use in the file.

Use ENVI_FILE_QUERY to get the number of bands in the Boulder TM image. Define a POS variable so that all of its bands are used. Then, use ENVI_GET_ROI_DATA to collect ROI data from the Boulder TM scene:

```
ENVI> envi_file_query, bldr_fid, nb=nb, wl=wl
ENVI> pos = lindgen(nb)
ENVI> veg_data = envi_get_roi_data(veg_id, fid=bldr_fid, pos=pos)
```

What did you get? The ROI data is returned as a 2D array with the bands in the columns and the ROI pixel values in the rows. Open a new IDL graphics window and plot the Z-profile of the first 10 pixels in the veg ROI:

```
ENVI> p = plot(wl, veg_data[*,0], xtitle='Wavelength', ytitle='Pixel Value')
ENVI> for i=1,9 do !null = plot(wl, veg_data[*,i], /overplot)
```



Spatial subsetting using ROIs

Many ENVI routines allow the option to spatially subset an image with an ROI. In these routines, the first element of the input to the DIMS is called the ROI pointer. It is used to tell ENVI that the spatial subset is based on an ROI. In all the examples thus far, the ROI pointer has been set to a value of -1L, which means that an ROI is not being used.

In this example, we compute the mean spectra for the water and clouds ROIs from the Boulder TM image. The statistics are computed with ENVI_STATS_DOIT, which uses DIMS to define the image subset to process.

Get the ROI pointers for the water and clouds ROIs. Use the procedure from step 11 to obtain the ROI IDs and ENVI_GET_ROI_DIMS_PTR to get the ROI pointers.

```
ENVI> i_water_id = where(strlowcase(roi_names) eq 'water')
ENVI> i_clouds_id = where(strlowcase(roi_names) eq 'clouds')
ENVI> water_ptr = envi_get_roi_dims_ptr(roi_ids[i_water_id])
ENVI> clouds_ptr = envi_get_roi_dims_ptr(roi_ids[i_clouds_id])
```

Set up DIMS variables for the ENVI_STATS_DOIT routine:

```
ENVI> water_dims = [water_ptr,0,0,0,0]
ENVI> clouds_dims = [clouds_ptr,0,0,0,0]
```

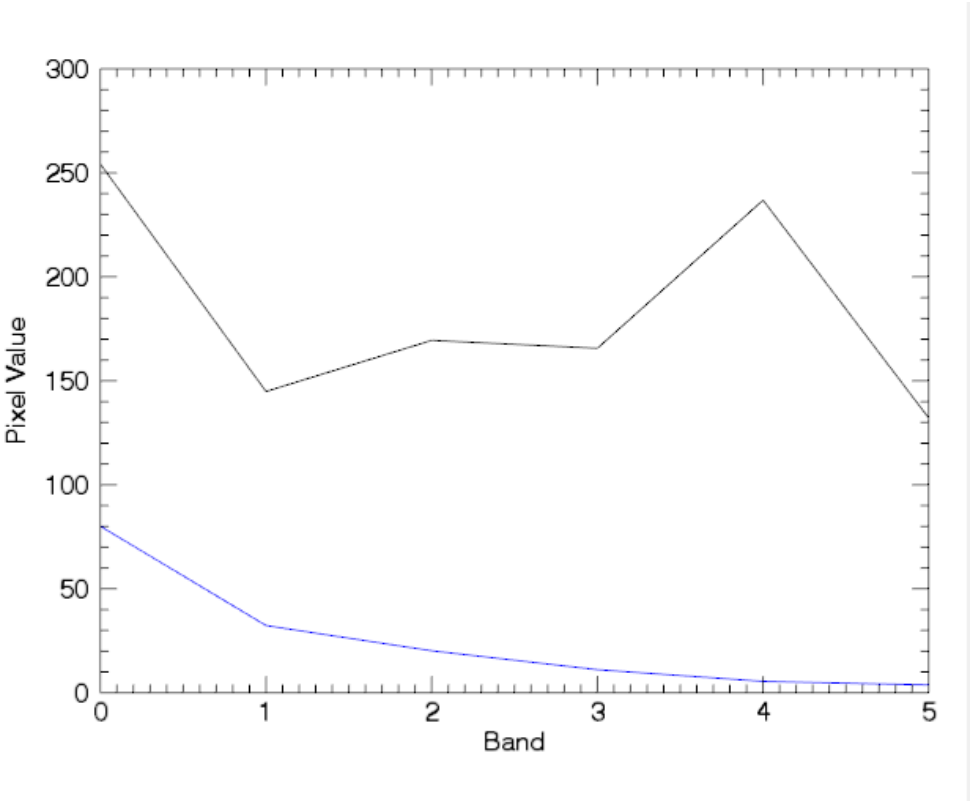
Compute the statistics:

```
ENVI> envi_doit, 'envi_stats_doit', fid=bldr_fid, pos=pos, $
> dims=water_dims, mean=water_mean, comp_flag=1
ENVI> envi_doit, 'envi_stats_doit', fid=bldr_fid, pos=pos, $
> dims=clouds_dims, mean=clouds_mean, comp_flag=1
```

The ENVI_DOIT wrapper is discussed two chapters from now in “Batch Mode Programming.”

Compare the results by plotting the mean spectrum for each spatial subset:

```
ENVI> p = plot(clouds_mean, xtitle='Band', ytitle='Pixel Value')
ENVI> p = plot(water_mean, xtitle='Band', ytitle='Pixel Value', $
> color='blue', /overplot)
```



Writing ENVI-format files

ENVI's image format is perhaps the simplest possible: a 'flat binary' file in which the raster image data are stored as a stream of bytes in BSQ, BIL, or BIP storage order. The file contains only image data—header information is not embedded in the file. The IDL WRITEU procedure automatically uses this format when arrays of image data are written to disk. An ENVI image file can have any name; there is no required file extension.

Accompanying each image file is an ASCII header file that specifies the image's basic characteristics (such as its dimensions and data type) as well as any additional information about the file (such as its georeferencing data and sensor type). In order for ENVI to recognize the header file, it must have the same root name as the image file with the .hdr extension. The format of the header file is detailed in the ENVI help system under Explore Imagery > Open and Browse Data > ENVI Header Files.

A list of ENVI routines for writing ENVI-format files is given in the following table.

ENVI library routines for writing ENVI-format files.

Routine	Purpose
ENVI_ENTER_DATA	Use this routine to enter image data from an IDL array into the Available Bands List. This routine automatically sets up a temporary ENVI header for the image and returns a FID. Once the image appears in the Available Bands List it can be used

	as any other ENVI image, and can even be saved to disk using File > Save File As. An important limitation of this routine is that the image data must be BSQ interleaved, so you will need to change the image interleave in IDL first if it is not already BSQ.
ENVI_WRITE_ENVI_FILE	This routine writes an image contained in an IDL array to an ENVI-format file. This routine automatically creates the ENVI header file for the image and you can use keywords to set up some of the initial header attribute values.
ENVI_SETUP_HEAD	Use this routine to create the ENVI header file for an image that is already saved to disk. This routine also allows the image file to be opened into the Available Bands List by setting the OPEN keyword. If ENVI_SETUP_HEAD is called without setting the OPEN or WRITE keyword, the ENVI header is created in memory only (which allows ENVI_FILE_QUERY to be used on the file's header information throughout the rest of the user function code). ENVI_SETUP_HEAD optionally returns a FID for the image file on disk.
CF_DOIT	Use this library routine to create a new ENVI format file from existing ENVI files. The images that are incorporated into the new file can be any combination of opened ENVI files on disk or in memory, and the resulting new file can also be saved to either file or memory. CF_DOIT is equivalent to using File > Save File As > ENVI Standard in the main ENVI menu.
ENVI_ASSIGN_HEADER_VALUE	Use this routine to assign a new user-defined header attribute setting to a file. This attribute will be saved to memory only and must be persisted with ENVI_WRITE_FILE_HEADER
ENVI_GET_HEADER_VALUE	Use this routine to retrieve a user-defined header attribute value from a file open in ENVI.
ENVI_WRITE_FILE_HEADER	This routine will persist any changes made in memory to an ENVI header to disk.
ENVI_CHANGE_HEAD	This undocumented (!) routine changes header file settings in memory without overwriting the current file. It can be helpful for troubleshooting files that have incorrect header data, but as it is undocumented use it at your own risk! Any changes made with ENVI_CHANGE_HEAD must be saved with ENVI_WRITE_FILE_HEADER if you want them saved to disc.

Saving images to memory

When a user function results in image data contained in an IDL array, this data can readily be entered into ENVI as a memory-only item using ENVI_ENTER_DATA.

Here's an example of using ENVI_ENTER_DATA to load the contents of a JPEG file from IDL into ENVI.

Locate and read a JPEG file into your ENVI+IDL session.

```
ENVI> read_jpeg, file_which('rose.jpg'), rose, /order
ENVI> help, rose
ROSE            BYTE      = Array[3, 227, 149]
```

Set the ORDER keyword to READ_JPEG to use the remote sensing convention for drawing images (used by ENVI) instead of the Cartesian convention (used by IDL).The variable rose has BIP interleaving. Convert it to BSQ for use with ENVI_ENTER_DATA :

```
ENVI> rose_bsq = transpose(rose, [1,2,0])
ENVI> help, rose_bsq
ROSE_BSQ        BYTE      = Array[227, 149, 3]
```

Now, pass the data to ENVI:

```
ENVI> bnames = ['R band','G band','B band']
ENVI> descrip = 'An RGB image of a rose.'
ENVI> envi_enter_data, rose_bsq, bnames=bnames, descrip=descrip
```

After executing this statement, the Available Bands List appears, displaying the bands of the image, output to memory. Display the data as an RGB image in ENVI. Note that the variable rose_bsq is undefined in the IDL session after the call to ENVI_ENTER_DATA . Variables are erased from memory once they are written to an ENVI session with ENVI_ENTER_DATA, but you can duplicate an array and write it by adding a copy step to the process above:

```
ENVI> bnames = ['R band','G band','B band']
ENVI> descrip = 'An RGB image of a rose.'
ENVI> rose_bsq_copy = rose_bsq
ENVI> envi_enter_data, rose_bsq, bnames=bnames, descrip=descrip
```

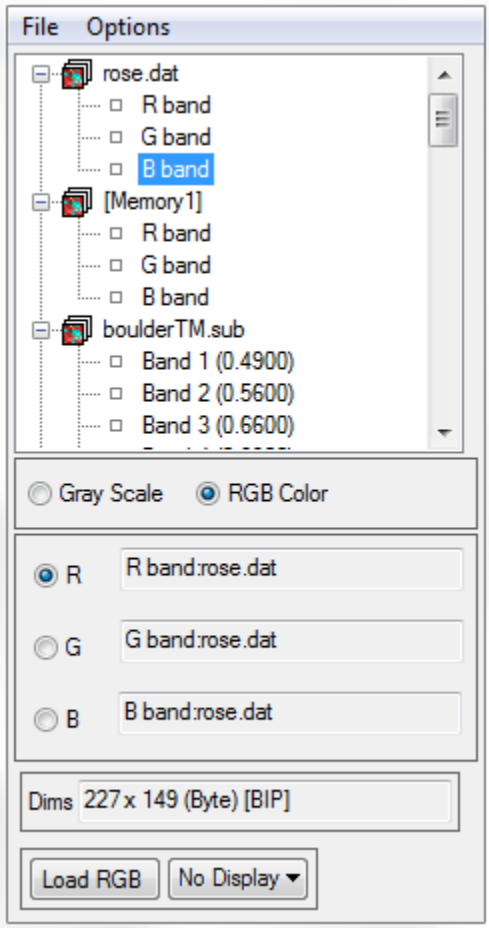
Saving images to disk

If the image data are held in an IDL array, ENVI has library routines to write an ENVI image file and its header.

Instead of writing the rose variable to memory, write it to the file system with ENVI_WRITE_ENVI_FILE :

```
ENVI> envi_write_envi_file, rose, out_name='rose.dat', $
> r_fid=rose_fid, data_type=1, interleave=2, $
> ns=227, nl=149, nb=3, offset=0, bnames=bnames, $
> descrip=descrip
```

After executing this statement, the Available Bands List appears, displaying the bands of the image, output to the file rose.dat . Display the data as an RGB image in ENVI. Note that unlike ENVI_ENTER_DATA , the BIP interleaving was retained in writing the file with ENVI_WRITE_ENVI_FILE .



IDL's WRITEU procedure can also be used to produce ENVI-format image files. WRITEU is often more convenient to use to make the image file because the image can be written as it is produced (i.e., within a processing loop without the need to store the entire result in a separate array). If you use such an approach, then the ENVI header must be created separately using ENVI_SETUP_HEAD .

Write the rose variable to another ENVI-format file using the IDL OPENW, WRITEU and FREE_LUN procedures:

```
ENVI> fname = 'another_rose.dat'
ENVI> openw, lun, fname, /get_lun
ENVI> writeu, lun, transpose(rose, [1, 2, 0])
ENVI> free_lun, lun
```

The file is written in BSQ format with native byte ordering. Use ENVI_SETUP_HEAD create the header:

```
ENVI> envi_setup_head, fname=fname, data_type=1, /open, /write, $
> r_fid=another_rose_fid, ns=227, nl=149, nb=3, offset=0, $
> bnames=bnames, descrip=descrip, interleave=0
```

The OPEN keyword writes the header to the Available Bands List, which appears after the call to ENVI_SETUP_HEAD . The WRITE keyword writes the header. Note that the WRITE keyword is optional—it is possible to create the header & not write it to a file.

Working with Vectors in ENVI

As ENVI vector routines are a more advanced topic, they are only given a very basic treatment here. Note that the following routines refer to the ENVI Vector File (EVF) format and routines for accessing that format. To get EVF output as a shape file (a more common GIS format), use ENVI_EVF_TO_SHAPE

Vector Routines in ENVI

Routine	Description
ENVI_EVF_OPEN	This function opens an existing EVF and returns an EVF ID. EVF IDs are used by routines like ENVI_EVF_INFO and ENVI_EVF_READ_RECORD.
ENVI_EVF_CLOSE	This routine closes an open EVF file.
ENVI_EVF_INFO	Returns general information about an ENVI Vector File such as projection information or the number of records.
ENVI_EVF_DEFINE_INIT	This function initializes a new EVF file and returns a pointer to it. This pointer differs from a standard EVF ID and is only valid for an EVF at creation time, and until that EVF is finalized.
ENVI_EVF_DEFINE_ADD_RECORD	Using an EVF file pointer returned from ENVI_EVF_DEFINE_INIT, ENVI_EVF_DEFINE_ADD_RECORD adds records to an EVF file. Each record consists of a point or set of points in X, Y coordinates as well as information on the type of record (point, line, polygon, etc.) Attributes are stored in a linked DBF file with the separate routine ENVI_WRITE_DBF_FILE
ENVI_EVF_DEFINE_CLOSE	Closes and finalizes an EVF file. You call this function when you are finished adding all records to the EVF file.
ENVI_READ_RECORD	This returns an X, Y coordinate vector (or scalar in the case of a point vector) for any given record number in an open EVF file. ENVI_READ_RECORD requires the EVF ID returned by ENVI_EVF_OPEN.
ENVI_WRITE_DBF_FILE	Use this format to define a table of attributes for an ENVI vector format file. There must be a one to one correspondence to records in the DBF file and records in the EVF file.

Most of the above routines are used in conjunction in the course file CREATE_EVF.PRO which has been adapted from examples in the ENVI Classic Help:

```
pro create_evf
  compile_opt idl2

  ; Create a Geographic projection
  ; with the default datum and units.
  proj = envi_proj_create(/geographic)


  ; Define a polyline vector in Lat/Lon coordinates
  polyline = [-106.904, 41.5887, $
              -106.821, 42.2302, $
              -106.013, 42.2183, $
              -105.206, 41.3749, $
              -105.657, 40.5078, $
              -105.835, 39.5574, $
              -105.170, 38.8447, $
              -104.125, 39.4862, $
              -103.269, 40.0563, $
              -103.269, 40.0682, $
              -102.913, 39.0585, $
              -102.901, 39.0585, $
              -102.901, 39.0348, $
              -103.210, 38.4289, $
              -103.804, 38.3695]

  polyline = reform(polyline, 2, 15)

  ; Define a polygon in Lat/Lon coordinates
  ; (note first and last coords are identical
  ; in order to close polygon)
  polygon = [-104.113, 41.6956, $
             -103.994, 42.1589, $
             -103.934, 41.6838, $
             -103.471, 41.8738, $
             -103.887, 41.5531, $
             -103.863, 41.0185, $
             -103.851, 41.0185, $
             -104.041, 41.4818, $
             -104.041, 41.4937, $
             -104.552, 41.2680, $
             -104.220, 41.6006, $
             -104.422, 42.0995, $
             -104.113, 41.6956]
  polygon = reform(polygon, 2, 13)
```

```
; Define a set of discrete points in Lat/Lon coordinates
points = [-106.572, 39.6643, $
          -106.643, 39.5218, $
          -106.453, 39.4386, $
          -106.417, 39.6168, $
          -106.595, 39.4386, $
          -106.774, 39.2011, $
          -106.215, 39.4030, $
          -106.393, 39.2961, $
          -106.560, 39.1892, $
          -106.536, 38.9872, $
          -106.227, 39.1417, $
          -106.192, 39.1179, $
          -106.263, 38.9635, $
          -106.073, 39.1298, $
          -106.073, 39.2367]
points = reform(points, 2, 15)

; Initialize the new EVF file
evf_ptr = envi_evf_define_init('sample.evf', $
     projection=proj, data_type=4, $
     layer_name='Sample EVF File')
if (ptr_valid(evf_ptr) eq 0) then return

; Add the discrete points as individual records
for i=0,14 do $
   envi_evf_define_add_record, evf_ptr, points[*,i]

; Add the polyline record
envi_evf_define_add_record, evf_ptr, polyline

; Add the polygon record
envi_evf_define_add_record, evf_ptr, polygon

; Finish defining the new EVF file and
; then close the EVF file
evf_id = envi_evf_define_close(evf_ptr, /return_id)
envi_evf_close, evf_id

; Create an attribute file for this new EVF file:
; Records 1-15 are individual points
; Record 16 is a polyline
; Record 17 is a polygon
attributes = replicate( {name:'', id:0L}, 17)
for i=0,14 do begin
```

```
    attributes[i].name = 'Sample Point ' + strtrim(i+1,2)
    attributes[i].id = i+1
  endfor

  attributes[15].name = 'Sample Polyline'
  attributes[15].id = 16
  attributes[16].name = 'Sample Polygon'
  attributes[16].id = 17

  ; Write the attributes to the DBF
  envi_write_dbf_file, 'sample.dbf', attributes
end
```

A few things to note:

- Vector geometry must be defined as an ordered set of points represented as X and Y vectors. Polygons must reconnect to the first point by repeating it in order to close.
- Polylines, points, and polygons are entities which are added to the EVF file using ENVI_EVF_DEFINE_ADD_RECORD.
- DBF attribute records and the vector geometry are managed separately, but must still be made to correspond to one another. DBF attribute records are defined as an array of structures and written to a DBF file with ENVI_WRITE_DBF_FILE.

Working with GIS Datasets

Once you've run processing in ENVI, it's frequently necessary to make results available to users of GIS applications, and custom developed workflows frequently require GIS output. This is not an exhaustive list of GIS compatibility routines, but a quick few to help get started:

GIS Compatibility Routines in ENVI

Routine	Description
ENVI_OPEN_GDB	This reads a raster dataset form an ArcGIS geodatabase using a catalog path to the location of the raster dataset. It returns a FID which can be used to access the raster.
ENVI_EVF_TO_SHAPEFILE	This converts an ENVI vector format file to a SHP file.
IDLffShape	IDLffShape is the object-based file reader for the ESRI shape file format. All access to and from IDL and ENVI both is done through this interface. See the help topic IDLffShape in the IDL documentation for more information.
ENVI_OUTPUT_TO_EXTERNAL_FORMAT	This routine outputs raster/image data to non-ENVI formats, including ESRI grid, JP2, and NITF (if NITF license is available).

Several of the above routines are used in conjunction in the course file ENVI_GIS_INTEROP_EX.PRO:

```
pro envi_gis_interop_ex
  compile_opt idl2

  ; *** Convert from an EVF to a Shapefile

  ; Get an initial EVF to convert by locating or
  ; creating it.
  if ~file_test('sample.evf') then create_evf

  ; Open the vector file and get its ID.
  evf_id = envi_evf_open('sample.evf')

  ; Convert the file
  envi_evf_to_shapefile, evf_id, 'output.shp'

  ; *** Read an image from a GDB file
  envi_open_gdb, file_which('Iowa_agriculture_gdb.gdb') + path_sep() $
           + '1986_sub', $
           r_fid=iowa_fid

  ; *** Output to a GRID file
  ; Read a file from the ENVI distribution and read metadata.
  envi_open_file, file_which('boulder-ETM.dat'), r_fid=bldr_fid
  envi_file_query, bldr_fid, nb=nb, dims=dims
  pos = lindgen(nb)

  ; Output the results to a JPEG2000 file.
  envi_output_to_external_format, fid=bldr_fid,
      out_name='qb_boulder_jp2.jp2', /jp2, $
      dims=dims, pos=pos
end
```

As you can see, most of these routines are relatively simple to use. The one exception to this is IDLffShape - to use this routine, consult the help topic on IDLffShape. IDLffShape involves the use of file records, cursor/iterators for record entries, and shape geometry and is very similar to the EVF format, albeit packaged in an object API.

Chapter 9 ENVI's Object API

An object-based API was introduced in ENVI 5.0 to let you control the new ENVI user interface, manage data, and work with layers and views. The traditional API for customizing ENVI Classic is *procedural*, meaning that procedures and functions are the basis for extending functionality. The new API lets you customize ENVI functionality by writing simple object-oriented commands. You can still use IDL's traditional procedures and functions as well as objects to create your own object modules. However, applications built from objects are generally easier to maintain and extend than their traditional counterparts.

See "Object-Oriented Programming Concepts" in IDL Help, or consult another reference on object-oriented programming if you are new to the subject. Here are some basic concepts:

Think of objects as building blocks that can be created, re-used, or destroyed as you need them.

The ENVI API consists of the following objects (among others):

ENVI Application Object Basics

Class	Purpose
ENVI	Object used to start, close, access, and control the main instance of the ENVI application.
ENVIView	Object used to create, close, and control the contents of any of the ENVI application's views.
ENVIRasterLayer	Object used to control the visual properties of any Raster that is visualized in a view and exists in the Layer Manager's layer hierarchy.
ENVIRaster	An object representation of an image or other gridded dataset (such as a DEM).
ENVIPortal	An image portal exists within an instance of a view, and is used to 'look through' the highest layer in the layer hierarchy to the layer loaded in the portal.

Let's look at an example. The following procedural code starts the ENVI Classic interface, then closes it:

```
IDL> envi
IDL> envi_exit
```

The next set of code starts the ENVI interface by creating an ENVI application object ("e"):

```
IDL> e = envi()
ENVI> e.close
```

While this may look more complicated than the first example, this object reference is actually convenient to have. Now you can work with the object's data (called *properties*) or perform actions on the object (called *methods*).

Properties

Some objects have properties associated with them - things like size, data type, metadata, and so on. You can *set* some properties when you create (or *initialize*) the object. Other properties are only viewable after you create the object.

The ENVI object ("e") has several properties, one of which is LAYOUT. Set this property to a two-element array specifying the number of views to open when you start ENVI. Run the following code:

```
IDL> e = envi(layout=[2,2])
```

ENVI starts with four empty views.

You can see the list of available properties by putting your cursor at the end of "ENVI" in the IDL command line and clicking **Ctrl**-**Space**.

You can also set properties for ENVI once the object has already been created:

```
ENVI> e.layout = [2,1]
```

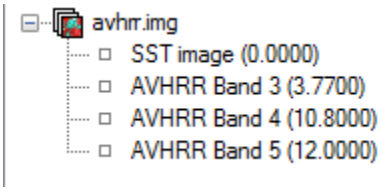
Methods

You can control an object with methods. IDL has both **procedure** methods and **function** methods. The use of `e.Close` in the example above shows the use of a procedure method. To see an example of a function method, type the following into the IDL command line:

```
ENVI> avhrr = e.openRaster(file_which('avhrr.img'))
```

OpenRaster is a function method that acts on the ENVI object. It is a function, so it returns a value--in this case, a reference to the ENVIRaster object that the ENVI application just opened. This reference is not contained in the variable `avhrr`.

Also, because we are controlling the ENVI application, elements of the user interface change when we invoke methods using our object reference to ENVI. Open the Data Manager in the ENVI application by clicking on the  icon. In the data manager, you will note that the avhrr.img file has been opened and its bands are available in ENVI, although it has not yet been added to any views:



Now that you understand the basic role of objects, properties, and methods, let's take a closer look at some of the ways you can use them in ENVI programming.

Manipulating the Main ENVI Application

In terms of basics, at the top level we have the ENVI application and the object which we can use to control it. We get access to the instance of this object by use of its constructor. There are three main ways that this constructor is called. The first is from an IDL session, in order to begin running ENVI:

```
IDL> e = envi()
% Restored file: ENVI.
ENVI>
```

The prompt will change to `ENVI>` and the user will be able to begin accessing ENVI functionality, including using the variable e, which now contains an object reference to the application, to control the application's behavior.

If you start an IDL/ENVI session *while an ENVI session is already active*, or you need access to the main ENVI application in a subroutine in an extension you're developing, you can explicitly get access to the current session of ENVI through the use of the /CURRENT keyword. An example of the usage is below:

```
ENVI> e = envi(/current)
```

This is a useful way to explicitly get access to a current session. If there is no current ENVI session available, the constructor will return a value of !NULL instead of an object reference and the prompt will remain IDL>:

```
IDL> e = envi(/current)
% Restored file: ENVI.
IDL> help, e
E               OBJREF    = <NullObject>
```

We can use this behavior to check to see whether or not ENVI is open :

```
e = envi(/current)
if e eq !null then begin
  ; Do something
endif
```

NOTE: Without using the IDL_IDLBridge object, it is impossible to instantiate multiple ENVI applications. Each IDL session is only capable of supporting one ENVI session, and multiple calls to the constructor will only create duplicate references to the current ENVI application.

```
IDL> e = envi()
ENVI> help, e
E               ENVI <152019>
```

```
ENVI> e = envi()
ENVI> help, e
E               ENVI <152019>
```

The other main way in which the ENVI constructor is called is headless mode. Headless mode can be used to batch processes without user interface elements of the application present. Headless mode will be covered in more detail in the chapter 10 on batch processing. The syntax for calling ENVI in headless mode is:

```
IDL> e = envi(/headless)
```

NOTE: The same rule about having only one ENVI session per IDL instance applies. If you first instantiate a headless instance of ENVI, you will first have to close the headless instance in order to open a fully interactive ENVI session. You can do this programmatically with a method invocation:

```
ENVI> e.close
IDL> e = envi()
```

Otherwise, the ENVI constructor will return a redundant reference to the current headless session.

Layers, Views, and Portals

Layers, views, and portals are the basic visual components of ENVI's visualization environment. In order for any dataset to be displayed in ENVI, it has to be first added to a layer, as the ENVIRasterLayer object contains information like the current stretch and band combination settings that ENVI requires to display a raster. Second, this layer itself, which describes how data from a raster is to be drawn, must be added to a View. The ENVIVIEW object contains one or multiple ENVIRasterLayer objects and draws them to its display.

To see this in operation, we'll work interactively at the command line. If ENVI is currently not active in your IDL workbench session, start it as follows:

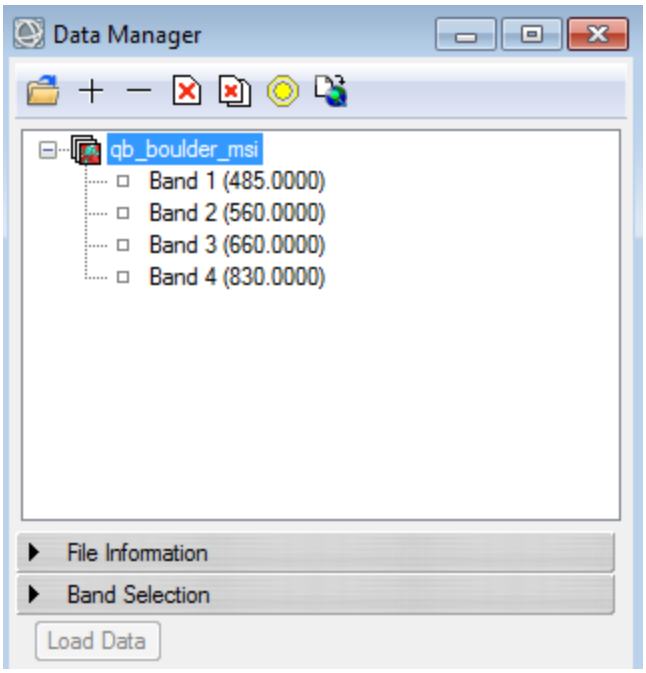
```
IDL> e = envi()
% Restored file: ENVI.
ENVI>
```

With the ENVI prompt available, type the following command to load a Quickbird multispectral image of Boulder:

```
ENVI> msi = e.openRaster(file_which('qb_boulder_msi'))
```

Note: If the course files are not available in the workbench, you will need to locate this file under the DATA directory of the ENVI50 installation directory.

At this point, the file will be open in ENVI, but not currently loaded for visualization. If you click on the data manager icon () and launch the data manager, you will see that the file has been loaded:



To visualize this image, we will need to create an ENVIRasterLayer to describe how the image is to be visualized, then add it to our active view (or to a new view we create). Because layers can be created through a method invocation on an ENVIView object, we will first get access to the currently active view (which happens to be the only available view for our new ENVI session):

```
ENVI> view = e.getView()
```

We now have a reference to the active view in a variable called VIEW. To create a layer from our existing ENVIRaster (stored in the variable MSI) we use the following method invocation:

```
ENVI> msi_lyr = view.createLayer(msi)
```

This is only one very simple way to create a view and populate it with a layer. The CreateLayer method of ENVIView objects has several optional keywords. If you look at the current ENVI application view you will see that the default visualization for the layer we loaded was a true color image (RGB composite with bands closest to red, green, and blue mapped to their respective color channels). Here we will create a second layer visualized as an infrared composite image:

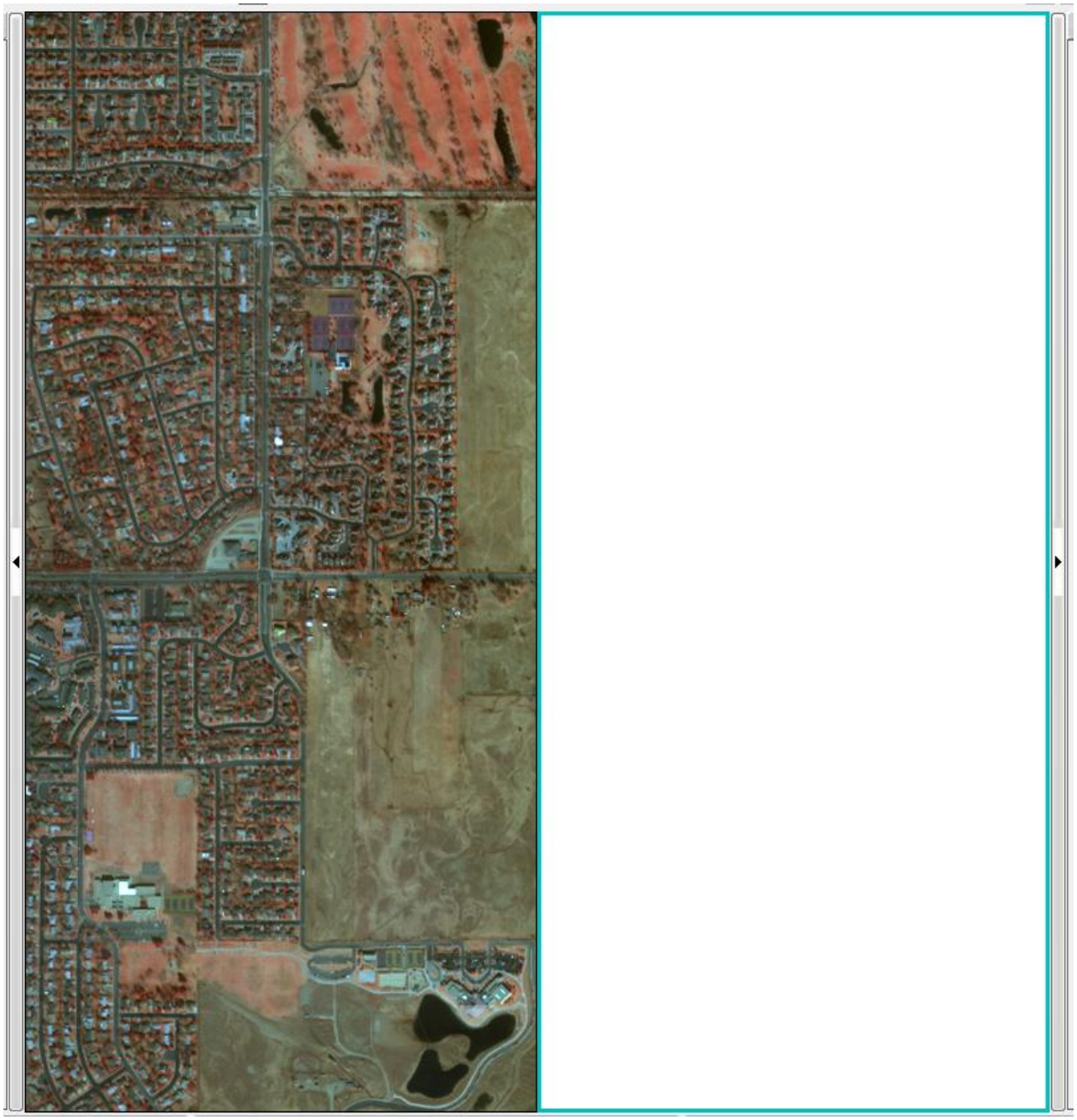
```
ENVI> cir_lyr = view.createLayer(msi, /cir)
```

Executing this command causes a new layer to be loaded which contains a composite infrared image derived from the same base raster through a new band combination. This layer has been loaded and added to the layer hierarchy under the name "`[1] qb_boulder_msi`"

We'll explore some other options with layer creation, but first we will create a new view to visualize them in. To do so, execute the following command:

```
ENVI> new_view = e.createView()
```

This will create a new view with a reference assigned to the variable NEW_VIEW. If you look at your ENVI application window you can see that the a new and currently unpopulated view has been added to the display.



We will populate this view with a custom band combination layer as follows:

```
ENVI> veg_lyr = new_view.createLayer(msi, bands=[2,3,0])
```

We have now added a new visualization where red and blue correspond to their closest real world color channel but the near infrared band is displayed in green (which more closely matches our perceived color of vegetation). However, it doesn't contrast much with the information we have displayed in our original view! To return to our true color composite, we'll use the following command:

```
ENVI> msi_lyr.moveToTop
```

This moves our original true color composite image to the top of the layer hierarchy. MoveUp, MoveDown, and MoveToBottom are also available layer hierarchy commands. We can also change the stretch on our original layer if we choose to:

```
ENVI> msi_lyr.quick_stretch = 'gaussian'
```

The quick stretches can all be assigned using a string that corresponds to the stretch name (identical to the one in the drop down menu in the application in most cases), so commands like:

```
ENVI> msi_lyr.quick_stretch = 'square root'
```

Are also possibilities. Most other layer properties, including contrast, brightness, etc. can be set this way. To change our transparency to look through the true color image into the underlying composite infrared image, we can set the transparency to 50% like so:

```
ENVI> msi_lyr.transparency = 50
```

Right now our two separate views are isolated from each other, despite the fact that they contain visualizations of the same imagery. If we pan or zoom around one image, for example, changes are not reflected in the other view. We can link these two views together very easily though:

```
ENVI> view.geoLink, new_view
```

Now if we pan through one view, the other view follows it in geographic extent. Since only two views are present, we also could have used this command:

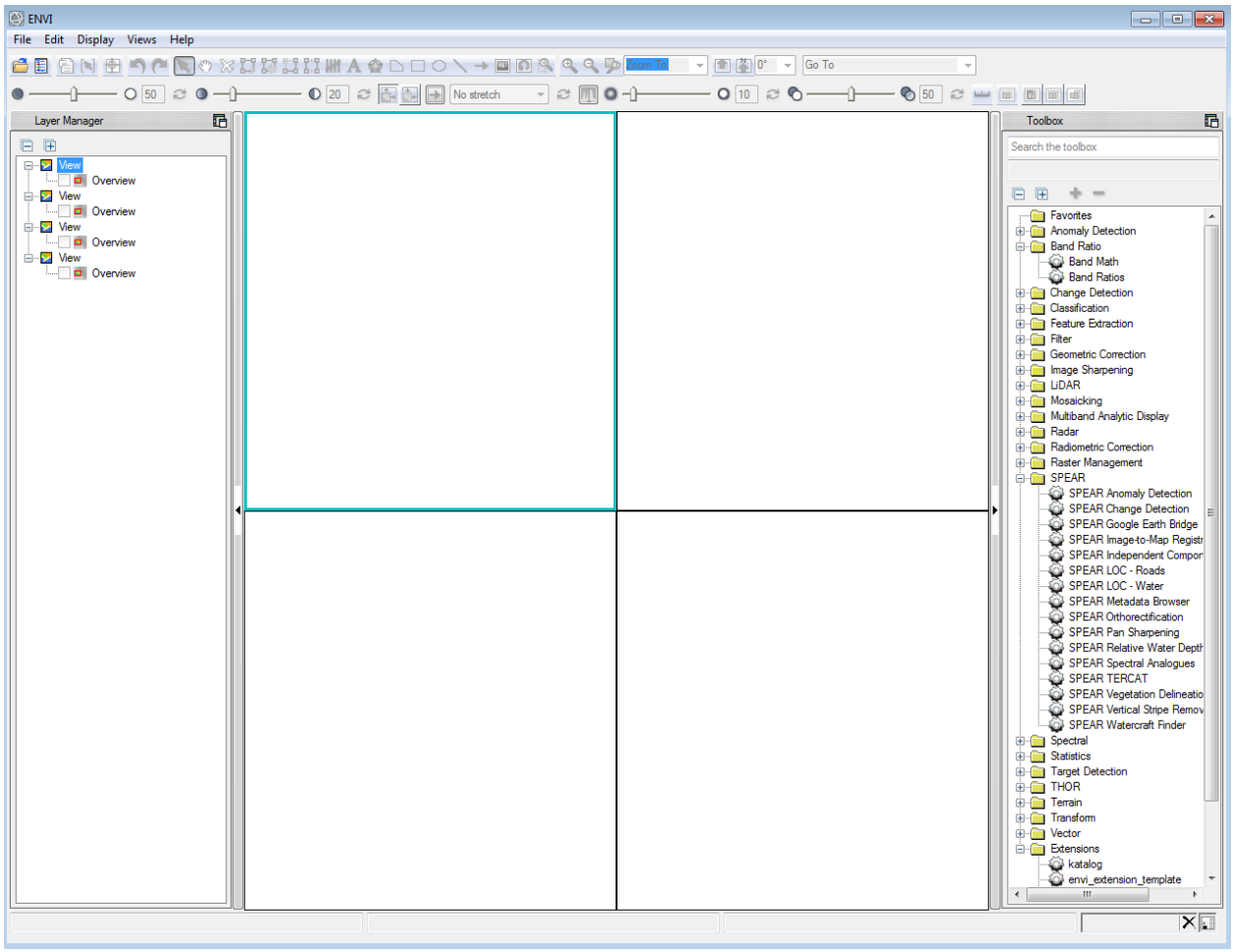
```
 ENVI> view.geoLink, /link_all
```

So far we have created and manipulated views using method invocations starting from a blank ENVI session, but we can actually initialize all of these views when we first initialize the ENVI session. To reset our current IDL session and clear ENVI from memory, type:

```
ENVI> .reset
IDL>
```

Now, to initialize ENVI with 4 views open, type in the following:

```
IDL> e = envi(layout=[2,2])
```



This will initialize ENVI with 4 views laid out in a 2 x 2 pattern (as shown above). Now, we'll get a reference to our multispectral image again:

```
ENVI> msi = e.openRaster(file_which('qb_boulder_msi'))
```

And we can get access to all of our ENVI views with a keyword argument to GetView:

```
ENVI> views = e.getView(/all)
```

This will return an array of object references, each corresponding to an ENVI view. We can use this array to quickly populate all the views in a FOREACH loop:

```
ENVI> layers = list()
ENVI> foreach view, views do layers.add, view.createLayer(msi)
```

Now we'll close all views but the first to look at how to visualize multiple datasets using a different technique:

```
ENVI> foreach view, views[1:*] do view.close
ENVI> view = views[0]
```

Sometimes it makes more sense to look “through” an image, rather than to compare two images side by side. For these cases, it makes more sense to use portals than to construct multiple views. First we’ll open the panchromatic image that matches the multispectral image:

```
ENVI> pan = e.openRaster(file_which('qb_boulder_pan'))
```

Now we’ll visualize it in a portal:

```
ENVI> pan_lyr = view.createLayer(pan)
ENVI> pan_lyr.moveToBottom
ENVI> portal = view.createPortal(layer=pan_lyr)
```

NOTE: a portal must be populated with an active layer. This is why we first load the pan image in a new layer and then move that layer to the base of the layer hierarchy for the view.

Portals and their background displays are inherently linked, so if we navigate to a new location (here done programmatically using the GoToLocation method of our ENVIView object) both the View’s layer and the portal’s layer change to the new location.

```
ENVI> view.goToLocation, 482234.9000D, 4427500.7000D, /map
```

Advanced Application Control

If you are an experienced IDL widget programmer, The ENVI object gives you access to widget_id of the top level ENVI widget through the WIDGET_ID property:

```
ENVI> tlb = e.WIDGET_ID
ENVI> help, tlb
TLB             LONG      =          909
```

You can use this WIDGET_ID in very basic dialog commands to assign widget grouping, such as with the following call to DIALOG_PICKFILE:

```
ENVI> file = dialog_pickfile(dialog_parent = e.widget_id)
```

In addition, you can use this information to "hack" the ENVI interface, adding or removing widgets, etc. Just remember the basic caveat that if the information you gain through this process is undocumented, it is subject to change in future versions of ENVI!

Exercises

1. Initialize ENVI with a 3 x 3 layout of views and populate each with different band combinations of the multispectral image of your choice.
2. Write a short program that uses DIALOG_PICKFILE to have the user select a multispectral image, opens that file as a raster, then constructs a 2 x 1 set of views with a true color and infrared composite of the same image with both views geo-linked.

Manipulating Data with the Object API

The previous section deals with controlling the ENVI application using the Object API. The Object API can also be used to get access to data in rasters, change or retrieve raster properties (analogous to header information in the procedural API), and get data from ENVIRaster objects, either in the form of entire bands and full image cubes, or through ENVI's tiling system through tile iterators. In addition, the Object API streamlines the process of having rasters inherit properties such as cell size and projection and coordinate system information from other rasters – of special importance when writing custom algorithms that create new raster products!

Getting Access to Image Data

Getting access to image data is really a very simple process in the new API. It can be done easily with a single line when a raster image is available:

If ENVI is not currently open, open it with:

```
ENVI> e = envi()
```

With ENVI open, type the following to get access to the MSI band we used above:

```
ENVI> file = filepath('qb_boulder_msi', root_dir=e.root_dir,
subdirectory='data')
ENVI> raster = e.openRaster(file)
```

To read data from an ENVIRaster object, we use the GetData method from that object. This can be a very simple method call in which the entire file is read as an IDL array:

```
ENVI> data = raster.getData()
ENVI> help, data
DATA            UINT      = Array[1024, 1024, 4]
```

Or it can be a much more targeted method call in which a specific subset is extracted:

```
ENVI> sub = raster.getData(bands=[1,2,3], interleave='bip',
sub_rect=[0,0,50,50])
ENVI> help, sub
SUB             UINT      = Array[3, 51, 51]
```

Either way, the data is immediately available for analysis and/or visualization in IDL, such as with this quick NDVI example:

```
ENVI> ndvi = (float(data[*,*,3]) - data[*,*,2]) / (float(data[*,*,3])
+ data[*,*,2])
ENVI> ndvi_img = image(ndvi, /order)
```

Writing Image Data

ENVI's object API also allows us to easily write image data. Here we use the function method GetTemporaryFilename from the ENVI application object (specifying an extension of .dat) and write the our NDVI result to an output file.

```
ENVI> ndvifile1 = e.getTemporaryFilename('.dat')
ENVI> print, ndvifile1
C:\Users\bkamphaus\AppData\Local\Temp\envitempfileWedJul251936572012694_1.dat
ENVI> ndvi_output1 = e.createRaster(ndvifile1, ndvi)
ENVI> ndvi_output1.save
```

We've now generated a new ENVI file containing our ENVI result. This file currently has no meaningful metadata and no coordinate projection system information, however. The Object API allows this information to be inherited from another raster at the time that we create the file. To use raster inheritance we can redo the previous sequence in the following way:

```
ENVI> ndvi_output2 = e.createRaster(e.GetTemporaryFilename('.dat'), $
ndvi, inherits_from=raster)
```

Using raster inheritance insures that any custom algorithm development we do in IDL will generate a product that preserves ENVI metadata – its use is recommended in all cases where your result is a raster of the same size and spatial extent as the original raster.

Raster Metadata

Every raster has a set of metadata stored in an ENVIRasterMetadata object. The properties in the ENVIRasterMetadata correspond to much of the same information accessed through ENVI_FILE_QUERY in the procedural API. ENVIRasterMetadata stores this information as tags, or keys in a HASH object. It is overloaded to print all defined properties when a call to print is made. Here we'll look at the metadata for our MSI ENVIRaster object.

```
ENVI> print, msi.metadata
ENVIRASTERMETADATA <182872>
  BAND NAMES                = 'Band 1', 'Band 2', 'Band 3', 'Band 4'
  DESCRIPTION               = 'Demo QuickBird 2 data courtesy
     DigitalGlobe', 'Inc.  Not for commercial use.'
  SENSOR TYPE               = 'QuickBird'
  WAVELENGTH                = 485.000, 560.000, 660.000, 830.000
  WAVELENGTH UNITS          = 'Nanometers'
ENVI> msi_metadata = msi.metadata
ENVI> help, msi_metadata
MSI_METADATA    ENVIRASTERMETADATA <182872>
ENVI> print, msi_metadata['band names']
Band 1 Band 2 Band 3 Band 4
```

Because the metadata object is wrapping a HASH, to get access to any of the fields in it we have to use key based subscripting in brackets, e.g. MSI_METADATA['band names']. We can use the AddItem method to add custom tags, which can then be later retrieved:

```
ENVI> msi_metadata.addItem, 'cloud cover percent', 0.0
ENVI> print, msi_metadata['cloud cover percent']
0.000000
```

We can also update items that the ENVI application uses:

```
ENVI> msi_metadata.updateItem, 'band names', ['Blue', 'Green', 'Red',$
> 'NIR']
ENVI> print, msi_metadata['band names']
Blue Green Red NIR
```

This will alter any metadata in memory. Let's quickly visit the case of our NDVI_OUTPUT2 raster which inherited metadata from its 'parent' raster, MSI:

```
ENVI> ndvi_info = ndvi_output2.metadata
ENVI> print, ndvi_info
ENVIRASTERMETADATA <182871>
  BAND NAMES                = 'Band 1', 'Band 2', 'Band 3', 'Band 4'
  DESCRIPTION               = 'Demo QuickBird 2 data courtesy
     DigitalGlobe', 'Inc.  Not for commercial use.'
  SENSOR TYPE               = 'QuickBird'
  WAVELENGTH                = 485.000, 560.000, 660.000, 830.000
  WAVELENGTH UNITS          = 'Nanometers'
```

Uh oh! It looks like our NDVI file inherited some metadata that while correct in its original context, no longer applies to the derivative raster! Luckily, we can overwrite this data:

```
ENVI> ndvi_info['band names'] = 'NDVI'
ENVI> ndvi_info['description'] = $
> 'NDVI image derived from Quickbird scene over Colorado'
ENVI> ndvi_info.RemoveItem, 'wavelength'
ENVI> ndvi_info.RemoveItem, 'wavelength units'
```

Let's check our final result:

```
print, ndvi_info
ENVIRASTERMETADATA <182871>
  BAND NAMES                = 'NDVI'
  DESCRIPTION               = 'NDVI image derived from Quickbird scene
     over Colorado'
  SENSOR TYPE               = 'QuickBird'
```

Now that we've cleared erroneous metadata and replaced it with accurate metadata, we can make our changes permanent:

```
ENVI> ndvi_output2.WriteMetadata
```

This will write the changes we've made to our metadata file to disc and therefore persist them to future ENVI sessions.

Advanced Data Access – Raster Tiles and Iteration

To process extremely large files using ENVI's internal tiling system, we must make use of an ENVIRasterIterator object. The syntax for creating a tile iterator is relatively simple:

```
ENVI> msi_i = msi.CreateTileIterator()
```

We can visualize each of the resulting tiles:

```
ENVI> foreach tile, msi_i do !null = image(tile)
```

The ENVIRasterIterator object is defined in such a way that we can declare a FOREACH loop to process per tile. The results may vary based on your workstation, but with sufficient hardware you're likely to get each band served as a full tile. This will result in four images being made, and a query to the number of tiles will also return the value '4':

```
ENVI> print, msi_i.ntiles
           4
```

We also have the option to iterate through a while loop if we choose, or, demonstrated next, to manually iterate through tiles using the NEXT method of ENVIRasterIterator. Now, we can't do that just yet because our FOREACH loop exhausted the tiles in our image, as the output from the CURRENT_BAND and CURRENT_SUBRECT properties indicates:

```
ENVI> print, msi_i.current_band
          -1
ENVI> print, msi_i.current_subrect
          -1          -1          -1          -1
```

So, we'll need to reset our tile iterator to the beginning and then have it deliver our first tile using the NEXT function method:

```
ENVI> msi_i.Reset
ENVI> tile = msi_i.Next()
ENVI> !null = image(tile)
ENVI> print, msi_i.current_band, msi_i.current_subrect
           0           0           0        1023        1023
```

This indicates that our first tile is of the Band at index 0, and defining the spatial bounds (in pixel coordinates) of [0, 0] (the origin) through [1023, 1023] (in this case the termination of the band).

If we wanted to run a custom edge detection algorithm against the red band that used a 15 by 15 consecutive moving window, we could define our own tiles to match by manually setting the tile size with the TILE_SIZE keyword if we choose to and by setting specific bands with the BANDS keyword.

```
ENVI> msi_alt_i = msi.CreateTileIterator(bands=[2], tile_size=[15,15])
ENVI> first_tile = msi_alt_i.Next()
ENVI> help, first_tile
FIRST_TILE      UINT      = Array[15, 15]
ENVI> print, first_tile[0:3, 0:3]
     266     254     284     270
     232     211     217     188
     230     201     189     168
     181     178     175     214
```

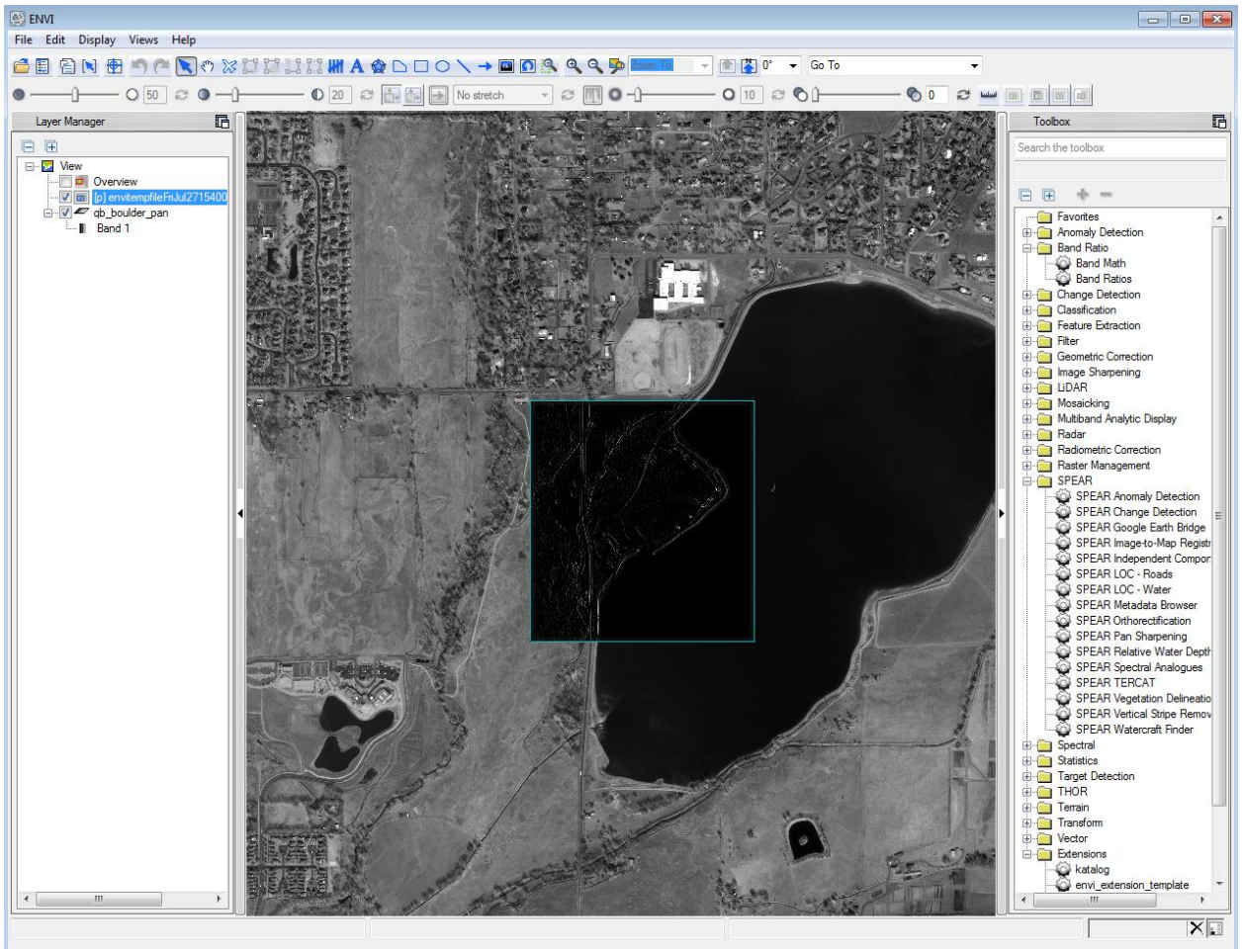
We can also write data using a tile iterator by using the ENVIRaster method SETTILE. Open the course file emboss_by_tile.pro to see an example of this:

```
pro emboss_by_tile
  compile_opt idl2

  ; Start ENVI if it's not open
  e = envi(/current)
  ife e eq !null then e = envi()

  ; Create an ENVIRaster
  bldr_pan_file = filepath('qb_boulder_pan', $
                           root_dir=e.root_dir, $
                           subdirectory = ['data'])
  bldr_pan_raster = e.OpenRaster(bldr_pan_file)

  ; Create an output raster
  new_file = e.GetTemporaryFilename()
  output_raster = e.CreateRaster(new_file, $
    inherits_from=bldr_pan_raster)

  ; Iterate through the tiles of the original data set
  bldr_tiles = bldr_pan_raster.CreateTileIterator()
  foreach tile, bldr_tiles do begin
    data = emboss(tile, /EDGE_WRAP)
    output_raster.SetTile, data, bldr_tiles
  endforeach

  ; Save the data
```

```
  output_raster.Save

  ; Visualize original file and results.
  view = e.GetView()
  lyr2 = view.CreateLayer(output_raster)
  lyr1 = view.CreateLayer(bldr_pan_raster)
  port = view.CreatePortal(layer=lyr2)
end
```

Here, because our new raster inherits from our Quickbird panchromatic image of Boulder, we can use one ENVIRasterIterator to iterate through both images at the same time, reading and processing one set of tiles `{ data = emboss(tile, /EDGE_WRAP)}` while outputting our results to the other tiles in the same loop: `output_raster.SetTile, data, bldr_tiles`

When we run this code, it generates this output:



Chapter 9 Batch Processing in ENVI

One of the most common tasks performed in ENVI programming is the automtaion of processes. This is referred to as **batching**. Batch processes in ENVI can be run whether the GUI is present or not.

- When the ENVI GUI is present, this is termed **ENVI + IDL** mode.
- When the ENVI GUI is not present, then ENVI is said to be running in **headless** mode.

Batch routines use ENVI_DOIT library routines to run different processes. To see the full list, consult the help document: **Alphabetic List of ENVI Classic Library Routines**. The library routines that end in DOIT are processes that can be batched with ENVI_DOIT. Some examples are CLASS_RULE_DOIT, DARK_SUB_DOIT, ENVI_FILTER_DOIT, and MNF_DOIT.

Batch Processing in ENVI + IDL Mode

When ENVI and IDL are opened using **ENVI 5.0 > ENVI + IDL** from the start menu, you do not have to initiate batch/headless mode in the concurrent IDL session to run ENVI_DOIT routines. You can use the linked IDL session directly to automate tasks in ENVI or perform unrelated IDL operations. To get access to your current ENVI session as an object, type the following into the console:

```
ENVI> e = envi(/current)
```

If an IDL procedure you are running at the command line produces a new image band, you can directly enter this new data by opening it, creating a layer from it, and adding it to the view (more on how this is done is included later). However, if the IDL procedure crashes, you will also have crashed the concurrent ENVI session. Also, ENVI may change the scope of your IDL session so that you do not have access to all the routines normally available at the main program level. When this occurs, you will receive several unexplained errors until you type `retall` into the command line to return to the main level of the program. There are also a few routines that operate differently when ENVI's GUI is present as opposed to when ENVI is being run in headless mode.

For large volume batch processing, we recommend using headless mode. This will help you avoid any GUI based errors or unexpected behavior while ENVI is batching processes.

Running ENVI in Headless Mode

Running ENVI headless is no different than working in an ordinary IDL session, except that ENVI library routines may be used. To get access to ENVI in headless mode, change the focus to your IDL command line and type:

```
IDL> e = envi(/headless)
```

This will restore ENVI library routines and allow you to access various methods off of the envi object. You can run routines that make use of ENVI's library without its GUI from any IDL program or from the IDL command line.

Example Batch Code

Below is a simple batch example that makes use of ENVI 5.0 library routines. This example can be used as a template for other batch processes:

```
pro batch_envi_veg_ind
  compile_opt idl2

  ; Start ENVI in headless mode.
  e = envi(/headless)

  in_file = $
      'C:\IDL_Coursefiles\extend\JasperRidge98av_flaash_refl.img'
  out_file = $
      'C:\IDL_Coursefiles\envidata50\enviout\jasper_ridge_veg_ind2.dat'

  ; Open the file, get back a Raster object.
  jasper_ridge = e.OpenRaster(in_file)

  ; Calculate several vegetation indices.
  envi_doit, 'envi_veg_index_doit', $
             fid=ENVIRasterToFID(jasper_ridge), $
             out_name=out_file

  ; Exit batch mode and end the program.
  e.close
end
```

Note that there are only a few important steps that must be followed. First, ENVI must be started in headless mode (`e = envi(/headless)`). Second, input and output file paths must be provided, and the input file must be opened (here using `e.OpenRaster(in_file)`). Third, the ENVI_DOIT routine has to be called, making use of properties that can be gotten from the raster object. Finally, ENVI is closed if this is the last process that will be batched (`e.close`).

Next you will run this program from the IDL workbench. The steps to run a batch program for the exercises in this section are:

1. Open the file from the IDL workbench.
2. Edit any of the parameters you wish to change, including hard-coded directory paths if present.
3. With the file active in the editor, click Run.

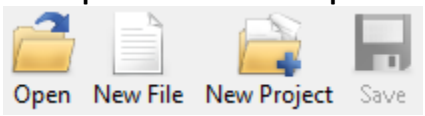
Run a simple batch program

Normally you will run batch programs on multiple files. This example has just one file.

1. Typically when running in batch mode you will start a new session of IDL. You can open and run a batch program in the current ENVI+IDL session. However, if something causes ENVI to crash, you will also exit IDL. If you start a new IDL session with the current session of ENVI+IDL running

you will need to specify a different directory for the additional IDL Workspace. Start a new session of IDL by clicking the Windows Start button then browse and select it from All Programs. If you have ENVI+IDL still running you will be prompted to open a different directory for the new IDL Workspace.

2. From the IDL menu select **File** → **Open** or click on the **Open** folder icon from the menu bar.



3. Navigate to the **extending\src** directory and open **batch_envi_veg_ind.pro**. The program will be loaded into the IDL workbench.

4. Note that the two variables `in_file` and `out_file` are hard coded to point to two files on the file system. Verify that the input file exists on your operating system at the location specified by in_file and that you have write permissions to write to the location specified by out_file. You may need to change the file path for both of these files to point to your `extend` directory.

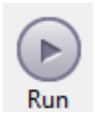
```
in_file = 'C:\...\JasperRidge98av_flaash_refl.img'
out_file = 'C:\...\jasper_ridge_veg_ind2.dat'
```

Note: The above location will only match some systems. You can test to see if the input file exists by using the IDL command line in the workbench, e.g.:

```
ENVI> print, file_test('C:\envidata\extend\JasperRidge98av_flaash_refl.dat')
```

This will return a **1** if the path is correct and the file exists and a **0** if the path is not correct or the file does not exist.

5. Verifying that **batch_envi_veg_ind.pro** is the active file in the editor, click the **Run** icon. This will both compile and run the program.



6. Restart ENVI by typing the following in the **IDL Console**:

```
IDL> e = envi(/current)
```

7. When ENVI starts, click **File** → **Open** then navigate to the location specified by the `out_file` variable and select `jasper_ridge_veg_ind2.dat`. One of the products from the processing will be displayed.

8. Open up the Data Manager and view other vegetation indices. After a product is displayed you may want to right click on its listing in the Layer Manager and select **Change Color Table** then selecting a specific color table to apply. You can also use Raster Color Slices by right clicking on a product in the Data Manager and selecting **Load Raster Color Slices**.

9. When you are finished inspecting the results click the **Close All Files** button in the Data Manager.

ENVI_DOIT Routines

NOTE: As of version ENVI 5.0, batch processes still use the ENVI_DOIT procedural wrapper. You will see some notes on converting back and forth from the procedural and object API further in the chapter.

ENVI_DOIT is a wrapper that is used to call ENVI processing from both current and headless session of ENVI. The name of the procedure called is always the first argument to ENVI_DOIT and is passed as a string. The remaining parameters vary based on which specific DOIT routine is called. Let's examine our call to ENVI_DOIT in the previous example:

```
envi_doit, 'envi_veg_index_doit', $
           fid=ENVIRasterToFID(jasper_ridge), $
           out_name=out_file
```

In this example, we called the vegetation index routine with the string 'envi_veg_index_doit.' The vegetation index generation routine is actually one of the easiest ENVI_DOIT routines to call (the reason it is used in the first example) and only has two *required* input keyword parameters. These are FID - the file ID of the input file, and OUT_NAME, the name of the output file the process will generate.

Understanding ENVI Documentation on ENVI_DOIT Routines

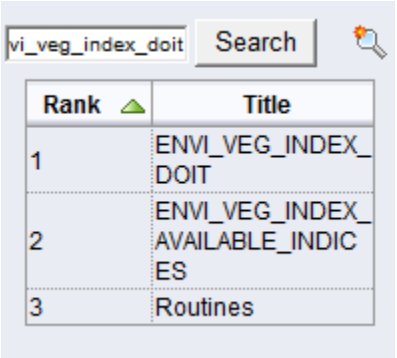
Because there are so many processes in ENVI that can be called through ENVI_DOIT, it is very likely you will have to learn a few specific ENVI_DOIT routines very well, and that these routines will differ for different users of ENVI working in different fields. It is therefore inevitable that you will encounter ENVI_DOIT routines that will not have encountered in course work or training and have to use them based on the code documentation in the ENVI help library.

Here we will look at how we 'knew' to call ENVI_DOIT with the FID and the OUT_NAME keywords to generate the correct result by looking at the documentation for ENVI_VEG_INDEX_DOIT.

1. Open ENVI help. You may do this by selecting **ENVI > Help > Contents** from an active ENVI session, or **ENVI 5.0 > Help > ENVI Help** from the Start Menu.

2. Select **Search** from the menu in the bottom left:

3. Type 'envi_veg_ind_doit' in the search dialog and click the **Search** button. ENVI will locate the best matching results. In this case, the first hit is the topic we're looking for:



4. The topic will now be loaded in the right side of the help menu. Here we will look at the documentation in some detail:

ENVI_VEG_INDEX_DOIT

Use this procedure to calculate vegetation indices from a spectral input image. The input spectral data must be atmospherically corrected and calibrated to reflectance.

Syntax

ENVI_DOIT, **'ENVI_VEG_INDEX_DOIT'**, [, /CROSS_CHECK] [, DIMS=*array*], FID=*file ID* [, /IN_MEMORY] [, M_FID=*file ID*] [, M_POS=*value*] [, OUT_NAME=*string*] [, R_FID=*file ID*] [, VI_LIST=*string*]

As shown above, the first line of any ENVI_DOIT routine's help topic will be the name of the DOIT routine, in this case ENVI_VEG_INDEX_DOIT, as well as a short description of what the routine will accomplish and potentially some recommendations for using the routine properly (such as the recommendation here that the input spectral data be atmospherically corrected and calibrated to reflectance).

The single most important section of the help topic for figuring out how to correctly call an ENVI_DOIT routine follows immediately under the **Syntax** subtitle! Here we see the syntax for properly calling a routine, as well as a list of all documented keywords. Every keyword appears colored in light blue, and

clicking on any keyword will take you to a subtopic describing the use of that keyword. Some keywords, such as FID, occur without brackets, whereas others, like VI_LIST occur surrounded by brackets […]. You will also note that a few keywords, CROSS_CHECK and IN_MEMORY for example, occur with a '/' preceding them. The reasons for these differences are as follows:

- Bracketed keywords are optional, keywords not appearing in brackets are required. Therefore every call to ENVI_VEG_INDEX_DOIT *must* include a FID for the input file (otherwise we have nothing to process!) but not every call to ENVI_VEG_INDEX_DOIT is required tocontain a list of which vegetation indices should be processed (VI_LIST), as ENVI has a default setting it can use (all available indices) when this keyword is not set.
- Keywords marked with a '/' are binary switches that use IDL's '/' syntax to be turned on and off. In almost every instance any keyword set this way is off by default and must be switched on. So if we want to perform biophysical cross-checking to filter out invalid results from our vegetation index calculations, we must use /CROSS_CHECK as a keyword argument when we call ENVI_VEG_INDEX_DOIT. If we don't set this keyword on, this component of the process won't be run.

There is one caveat with regards to required keywords, and in this caveat lies all the complexity of ENVI batch programming – ***some keywords, though specified as optional, may in fact be required depending on how other keywords are set.*** This comes up very frequently, and is probably the #1 cause of ENVI_DOIT routines not working correctly, so we will repeat this again, as clearly as possible:

SOME KEYWORDS, THOUGH SPECIFIED AS <u>OPTIONAL</u>, MAY IN FACT BE <u>REQUIRED</u> DEPENDING ON HOW OTHER KEYWORDS ARE SET.

We have one example of this in our first example batch program. Again, stressing that our example's declaration of ENVI_VEG_INDEX_DOIT uses the minimal set of keywords necessary to make a call:

```
envi_doit, 'envi_veg_index_doit', fid=ENVIRasterToFID(jasper_ridge),
out_name=out_file
```

However, according to our **Syntax** sub-topic for ENVI_VEG_INDEX_DOIT, only **FID** is not listed in brackets and is therefore required by every ENVI_VEG_INDEX_DOIT call. What's going on?

This is a case where we have to do one other thing with a keyword declaration, but we can use different keywords to meet the requirement of the routine. For ENVI_VEG_INDEX_DOIT, this is a very common scenario. We *have* to tell the DOIT routine what to do with the output, but we could do this using two different (and exclusive keywords). We either tell ENVI_VEG_INDEX_DOIT the name of the output file it will generate, OR, we tell it perform the task and generate the result in memory using the IN_MEMORY keyword. So returning to our example program, there is another way we can 'minimally' call this DOIT routine:

```
envi_doit, 'envi_veg_index_doit', fid=ENVIRasterToFID(jasper_ridge),
/in_memory
```

This choice between in memory output or file output is one scenario in which you will often encounter keywords listed as optional when one from a set is required. More complex DOIT procedures, like CLASS_DOIT for example, will have more complex scenarios where one set of keywords will be required for one classification method whereas another set of keywords will be required for another classification method. When this occurs, the documentation is usually broken up by topic, though, for these issues to be more clearly delineated. Here is the example of how the **Syntax** section looks for CLASS_DOIT

ENVI_DOIT, **'CLASS_DOIT'**, CLASS_NAMES=*string array*, DIMS=*array*, FID=*file ID*, /IN_MEMORY, LOOKUP=*array* [, M_FID=*file ID*] [, M_POS=*value*], METHOD={0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8} [, OUT_BNAME=*string*], OUT_NAME=*string*, POS=*array* [, R_FID=*variable*]
Keywords for any supervised classification method:
MEAN=*array* [, RULE_FID=*file ID*] [, /RULE_IN_MEMORY] [, RULE_OUT_BNAME=*string array*] [, RULE_OUT_NAME=*string*]
Keywords for parallelepiped classification:
STDV=*array* [, STD_MULT=*value*]
Keywords for minimum distance classification:
[, DATA_SCALE=*floating point*], STDV=*array* [, STD_MULT=*value*] [, THRESH=*value*]
Keywords for maximum likelihood classification:
COV=*array*, DATA_SCALE=*floating point*, STDV=*array* [, THRESH=*value*]
[…]

Converting Between the Object API and the Procedural API

Here we also see our first example of converting between the procedural and the object API, in the keyword assignment to FID: `fid=ENVIRasterToFID(jasper_ridge)`. This keyword assignment expects a FID, but at the time we call ENVI_DOIT we have opened our file with ENVI's OpenRaster and don't know what its FID is. ENVIRasterToFID will return a FID when we pass in a raster object. This operation is performed in line in this case, but we could also declare it earlier and call both procedural and object API ENVI calls throughout the program using the FID when appropriate and the ENVIRaster object when appropriate.

NOTE: Later in this chapter you will see ENVIRasterQuery , an optional extension to the ENVI API that allows access to raster metadata that matches the results of ENVI_FILE_QUERY and may be more straight forward to work within the context of ENVI_DOIT routines than the built in ENVIRasterMetadata (which is structured to support Object API operations, rather than procedural API operations).

Common Keywords for ENVI_DOIT

As stated before, this first example uses ENVI_VEG_INDEX_DOIT – a DOIT which does not require too many keywords to be set. Most ENVI_DOIT routines, however, require that other parameters be set.

Below is a guideline to several common keywords which must be set by DOIT routines. Several of these are review items from the procedural API chapter.

FID

FID tells ENVI which open file it needs to process. FID stands for File ID and is a long-integer scalar (single value) that is greater than 0. An invalid FID has a value of -1. The FID is provided as a named variable by one of several ENVI routines used to open or select a file. You can get a FID back from ENVI_OPEN_FILE, or by using the ENVIRasterToFID() function on an ENVIRaster object.

DIMS

DIMS tells ENVI_DOIT which part of the file to process. The "dimensions" keyword is a five-element array of long integers that defines the spatial subset (of a file or array) to use for processing. Almost every time you specify the keyword FID, you must also specify the spatial subset of the file to use (even if the entire file, with no spatial subsetting, is to be processed).

POS

POS tells ENVI which bands to use. Use this keyword to specify an array of band positions, indicating the band numbers on which to perform the operation. For example, to use the second and third band if a 6 band image, POS would look like [1, 2]. To use every band in this 6 band image, set pos to LINDGEN(6).

IN_MEMORY

IN_MEMORY tells ENVI whether to create the output in memory or on the file system. This keyword is set on: (/in_memory) to specify that output should be stored in memory only (e.g. no file will be created). If you do not set IN_MEMORY, usually you need to set OUT_NAME.

OUT_NAME

OUT_NAME tells ENVI the name of the file to create on the disk to store the results of the process. Use this keyword to specify a string with the output filename for the resulting data. If IN_MEMORY is set, you do not need to set OUT_NAME.

OUT_BNAME

Use this keyword to specify a string array of output band names. For example, if using MATH_DOIT to calculate a custom index, you could specify the result as ["NDBR"].

R_FID

Use R_FID when you need to get the file ID of the image created by an ENVI process to perform more processing on the image. The resulting value of the variable set to R_FID can be used to run another ENVI Routine or to return results with ENVI_FILE_QUERY, just as any other FID.

Open and run several complex batch programs.

1. If you still have the previous IDL and ENVI session open, go to the IDL workbench and after the ENVI prompt type:

```
ENVI> e = envi(/current)
ENVI> e.Close
```

   This will close ENVI. Because you started this session as just IDL, closing ENVI will not close IDL.

2. From the IDL workbench, select **File** → **Open** or click on the **Open** icon from the file manipulation menu bar:

3. Navigate to your **extend\src** directory and examine the available files. In addition to **batch_envi_veg_ind.pro** from the previous exercise there are several other batch examples. These files contain several types of ENVI_DOIT routines, so looking through them should give you some basic familiarity with how different keywords are called and used. In addition, a few of them contain ways to search for and run operations on multiple files or to accept input from the user prior to initiating the batch. Select the file **batch_envi_enh_lee.pro** and click **Open**.

4. The batch code for running the enhanced lee filter in `batch_envi_enh_lee.pro` is similar to our simple example from the previous exercise. There are, however, a few areas of code worth pointing out:

```
            envi_file_query, ENVIRasterToFID(img), dims=dims
            ; Run Enhanced Lee Filter in batch mode.
            envi_doit, 'adapt_filt_doit', $
            pos=lindgen(img.nbands), $
            dims=dims, $
            fid=ENVIRasterToFID(img), $
            method=6, $ ; Enhanced Lee
            kx=5,$
            cu=0.52, $
            cmax=1.73, $
            damp=0.0, $
            out_name=out_file
```

   In this case, we are running an ENVI process that requires several more parameters, including the standard keywords POS and DIMS. The METHOD, KX, CU, CMAX, and DAMP keywords are all explained in the help for ADAPT_FILT_DOIT. These correspond to parameters you would see when running the tool from ENVI's Toolbox (Filter → Enhanced Filter).

   To set dims, we use the older routine, ENVI_FILE_QUERY. ENVI_FILE_QUERY accepts a FID, rather than a raster object, so we use ENVIRasterToFID to get a FID from our ENVIRaster object

"IMG." Similarly, we use the NBANDS property of our ENVIRaster to generate our POS array. The result of this operation for an 8-band image will be [0,1,2,3,4,5,6,7] which will tell ENVI to use every band in the image.

Optional: If you are not sure what a tool or batch program is doing, check to see which DOIT routine is being used, then search for it in the Toolbox. You may need to check the help for that particular DOIT routine. Open and run the tool and try lining up parameters in the tool versus the ones in the DOIT. There will not always be a one-to-one correspondence (sometimes extra things are exposed in the GUI, sometimes in the API), but you will be able to get a good feel for how to set the parameters programmatically by exploring the GUI tools.

5. Change the hard coded input and output file paths as necessary. Then click the Run button to compile and run **batch_envi_enh_lee_rq.pro**. This batch program is similar to the previous example, except that it makes use of a convenience routine not included with the ENVI default. This program is ENVIRasterQuery, an object interface to get keywords and parameter settings after querying a file. By default, all the settings of ENVIRasterQuery match those of the input file. Therefore, it should be used when you intend to run the process on the entire file, rather than a subset.
6. In the following lines, we can see how ENVIRasterQuery is used:

```
img_info = ENVIRasterQuery(img)

envi_doit, 'adapt_filt_doit', $
          pos=img_info.pos, $ ; Standard Keyword POS
          dims=img_info.dims, $ ; Standard Keyword DIMS
          fid=img_info.fid, $ ; Standard Keyword FID
          method=6, $ ; Enhanced Lee
          kx=5,$ ; Kernel size
          cu=0.52, $ ; Homogenous Area Cutoff
          cmax=1.73, $ ; Heterogenous Area Cutoff
          damp=1.0, $ ; Dampening coefficient
          out_name=out_file ; Output file
```

Because we used the raster query object, now instead of deriving all of the standard keywords from an input parameter, we can assign them by calling each needed keyword by name from our query object. So for POS, we use `img_info.pos`, and for dims we use `img_info.dims` and so on. ENVIRasterQuery greatly simplifies the necessary IDL knowledge required to run a batch process, so if lines like `pos=lindgen(nb)` still seem a bit esoteric to you, feel free to use it in your own batch programs. You will need to keep a copy of it in the same directory you call batch programs from, or elsewhere in your IDL path.

Note: You can see which directories are in your IDL path by typing the following at the command line:

```
ENVI> print, !path
```

7. There is one other section of code to highlight from batch_envi_enh_lee_rq.pro, namely the block in which ENVI is started:

```
; Start ENVI in headless mode unless ENVI is open.
 e = envi(/current)
 if e eq !null then begin
   e = envi(/headless)
   must_exit = 1B
 endif else must_exit = 0B
```

This code checks to see if ENVI is open, and if it is not open, starts a new process. Otherwise, we use the current ENVI process. A code block like this allows us to write a batch program that will still run whether or not ENVI was open. You'll notice that the must_exit variable is assigned a value of 1 in the case that ENVI was not open. We check on this and exit ENVI at the end of the program if necessary with the following line:

```
; Exit ENVI unless ENVI was already open
if must_exit then e.close
```

8. Next, open and look at `batch_ica.pro`. Note the following lines of code:

```
pro batch_ica, in_file, out_file99
  compile_opt idl2, logical_predicate

[…]

; If called from command line use command line arguments.
  args = command_line_args(count=args_count)
  if args_count then begin
    in_file = args[0]
    out_file = args[1]
  endif
```

The code above enables batch_ica to be run either from the workbench or IDL command line, or from an external process such as your operating system command line. In either case, the input file and output file names must be passed as parameters. The way to call this from IDL is shown in the next step.

9. To run **batch_ica.pro** hit the 

button and then go to the command line. Type:

```
ENVI> batch_ica, file_which("boulder-etm.dat"), $
"C:\ ENVI_coursefiles\envidata50\extend\boulder_ica.dat"
```

10. Note that your output file path will probably differ. Also, if `boulder-etm.dat` is not in your file path, you may need to hard code the input file path (it is located in your `extend` directory).
11. Now you can run this program multiple times from the command line and supply a different input and output file each time, without worrying about what the settings are. Constructing programs like this is referred to as modularizing them; breaking them into the necessary components so those components can be re-used or called as needed. If you have a little bit of IDL programming knowledge, it's possible to do things like construct a string array of file names from a file search in order to run your batch process on every file in a particular directory.
12. Also note that BATCH_ICA is written to be compatible with earlier versions of ENVI (`compile_opt idl2, logical_predicate`). For this reason, it uses the procedural interface to ENVI, and makes calls to ENVI_BATCH_INIT and ENVI_BATCH_EXIT rather than `e = envi(/headless)` and `e.close()`. Note that if you want to share code with colleagues using a version of ENVI prior to 5.0, your batch routines will need to be written in this way. You can consult the help topic on these routines for more information.

Optional: Note that there are other batch and programming examples included in your **extending\src** directory in the course data. These include a few batch examples not discussed directly in the manual. Each of these components is commented in the code at a detailed level to help a new user learn how to use similar coding practices in their own batch programs.

Note on code use/license: Feel free to copy and paste from and/or use these batch programs as templates for your own work. This code is not protected and is meant for instructional purposes. This code has no warranty and represents no obligation from Exelis VIS, and therefore the following legal terms apply:

Basic String and File Routines for Handling Multiple Files

The above examples deal with cases where we have single files to be processed. However, it is much more common to batch multiple files. Here we will look at the example of using BATCH_ICA above, so that we can separate the process of figuring out which files to process and what to call the results from

the process of running an ENVI_DOIT routine. Also, this is actually a good programming practice! Let's review our syntax for the course program BATCH_ICA:

```
ENVI> batch_ica, file_which("boulder-etm.dat"), $
> "C:\ ENVI_coursefiles\envidata50\extend\boulder_ica.dat"
```

This program only requires an input file name and an output file name. We'll look at how to do both quickly and conveniently.

Generating a List of Input Files

There are many ways this can be done, but these are the top two:

```
ENVI> file_list = dialog_pickfile(/multiple_files)
```

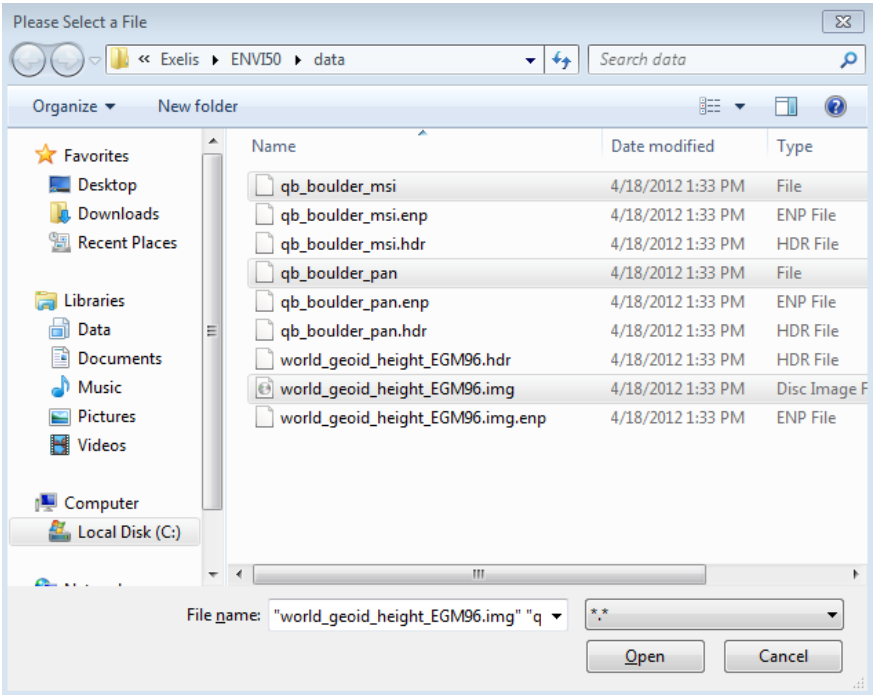
And:

```
ENVI> file_list = file_search('c:\extending\data\*.dat')
```

The first one, DIALOG_PICKFILE, prompts us to navigate to a directory and then choose a file or files to run processes on. It will return a string array when the /multiple_files keyword is set. Here's a more fleshed out example:

```
ENVI> file_list = dialog_pickfile(/multiple_files, $
> path=(e.root_dir + 'data'))
```

This results in a dialog box where we can select three different input files:



When the user clicks open, our variable FILE_LIST now contains the following values:

```
ENVI> foreach file, file_list do print, file
C:\Program Files\Exelis\ENVI50\data\qb_boulder_msi
C:\Program Files\Exelis\ENVI50\data\qb_boulder_pan
C:\Program Files\Exelis\ENVI50\data\world_geoid_height_EGM96.img
```

Here we're displaying the results one line at a time within a FOREACH loop because in practice, this will be a great way to process multiple files.

The second method uses string matching to locate all files that meet a condition. In this case, every file in the course data directory ending with '.dat' will be selected and added to a string array of file names. Here is another quick example where we get all of the header files from ENVI's data directory:

```
ENVI> headers = file_search(e.root_dir + 'data' + path_sep() +
'*.hdr')
ENVI> foreach head, headers do print, head
C:\Program Files\Exelis\ENVI50\data\qb_boulder_msi.hdr
C:\Program Files\Exelis\ENVI50\data\qb_boulder_pan.hdr
C:\Program Files\Exelis\ENVI50\data\world_geoid_height_EGM96.hdr
```

Here we're building a search string in line. If we did this separately, the results would look like so:

```
ENVI> print, e.root_dir + 'data' + path_sep() + '*.hdr'
C:\Program Files\Exelis\envi50\data\*.hdr
```

This is a search string passed to the operating system. Here we use a couple of tools to make this search portable. The first, using E.ROOT_DIR ensures that we look inside the base installation directory of ENVI no matter where our software is currently deployed directory wise. The second, using the PATH_SEP function, will ensure that if we call this code from a UNIX based system or WINDOWS based system that correct path delimiter will be used (e.g. / vs. \).

The most important character in this search string is the wildcard, or '*' – and this works just like it does in system searches in most operating systems. Anything that matches the path information and file name information prior to the * and also matches the file name information after the * will be determined a match and added to our string array result.

Deriving Output File Names

There are many options for string processing, certainly far too many to cover in this topic, but two suggested tools for deriving output file names are STRMID and STRPOS, and here we will briefly cover their use. Refer to the topic STRING PROCESSING in the IDL help for more information and options like regular expressions, etc.

Here we'll use string processing on the first file from our headers file search as an example:

```
ENVI> hdr= headers[0]
```

```
ENVI> print, hdr
C:\Program Files\Exelis\ENVI50\data\qb_boulder_msi.hdr
ENVI> base = strmid(hdr, 0, strpos(hdr, '.'), /reverse_search)
ENVI> print, base
C:\Program Files\Exelis\ENVI50\data\qb_boulder_msi
```

In this example we've used STRMID and STRPOS in conjunction to derive a prefix for our file name that lacks the extension. We can use string concatenation to add a suffix and extension to the file:

```
ENVI> output = base + '_ica.dat'
ENVI> help, output
OUTPUT          STRING    = 'C:\Program
Files\Exelis\ENVI50\data\qb_boulder_msi_ica.dat'
```

What if we need to write to a different location than where we're inputting files from? No problem! Here we'll select an arbitrary output location:

```
ENVI> output_dir = 'c:\scratch'
```

Then we'll derive just the basename without path information from our HDR file:

```
ENVI> output_sname = file_basename(hdr)
ENVI> print, output_sname
qb_boulder_msi.hdr
```

We'll use the same combination of STRMID and STRPOS we used before to get the location of the extension and return an extension-less file prefix:

```
ENVI> base = strmid(output_sname, 0, strpos(output_sname, '.'))
ENVI> print, base
qb_boulder_msi
```

Now we'll concatenate using portable coding practices:

```
ENVI> output = output_dir + path_sep() + base + '_ica.dat'
ENVI> print, output
c:\scratch\qb_boulder_msi_ica.dat
```

Multiple Batching Exercise

It's time to put everything we've looked at together in one exercise. This is going to be the most complex exercise we've tackled in the manual so far, and we're going to break it up incrementally.

- Write a batch program that runs ICA on every ENVI file in one directory and writes the results to an output directory.

To do this, follow these steps:

1. Make a new empty directory on your computer (somewhere where you have write permissions). Copy 3 multispectral files (Landsat, etc.) from the course files and/or from ENVI's installation data directories. **Note the location of this directory! Put a string in the text file if you have to.**
2. Make a second empty directory titled something like 'scratch' or 'output' – again somewhere on the computer where you have write permissions. **Note the location of this directory! Put a string in the text file if you have to.**
3. Begin writing a program. Make it a procedure that can be called by name or by clicking **Run**. It will be in a text block like:

```
pro batch_multiple_files
  compile_opt idl2
end
```

4. Use either the FILE_SEARCH or the DIALOG_PICKFILE method to get a string array containing the name of the files in the directory.

   NOTE: Feel free to change them to have the same extension, like .DAT for example, so that this is easier. ENVI doesn't care what the file's extension is so long as there's a matching header file, so you can change a file like "qb_boulder_msi" to "qb_boulder_msi.dat" just by using Windows Explorer or something similar.
5. Determine the output name and process the files using the course program BATCH_ICA in a FOREACH loop, using a template similar to:

```
foreach file, filelist do begin
  output_file = ;string and file processing
  batch_ica, file, output_file
endforeach
```

6. Run your completed program from the command line and inspect the results! For one possible solution look at BATCH_ICA_MULTIPLE.PRO in the coursefiles.

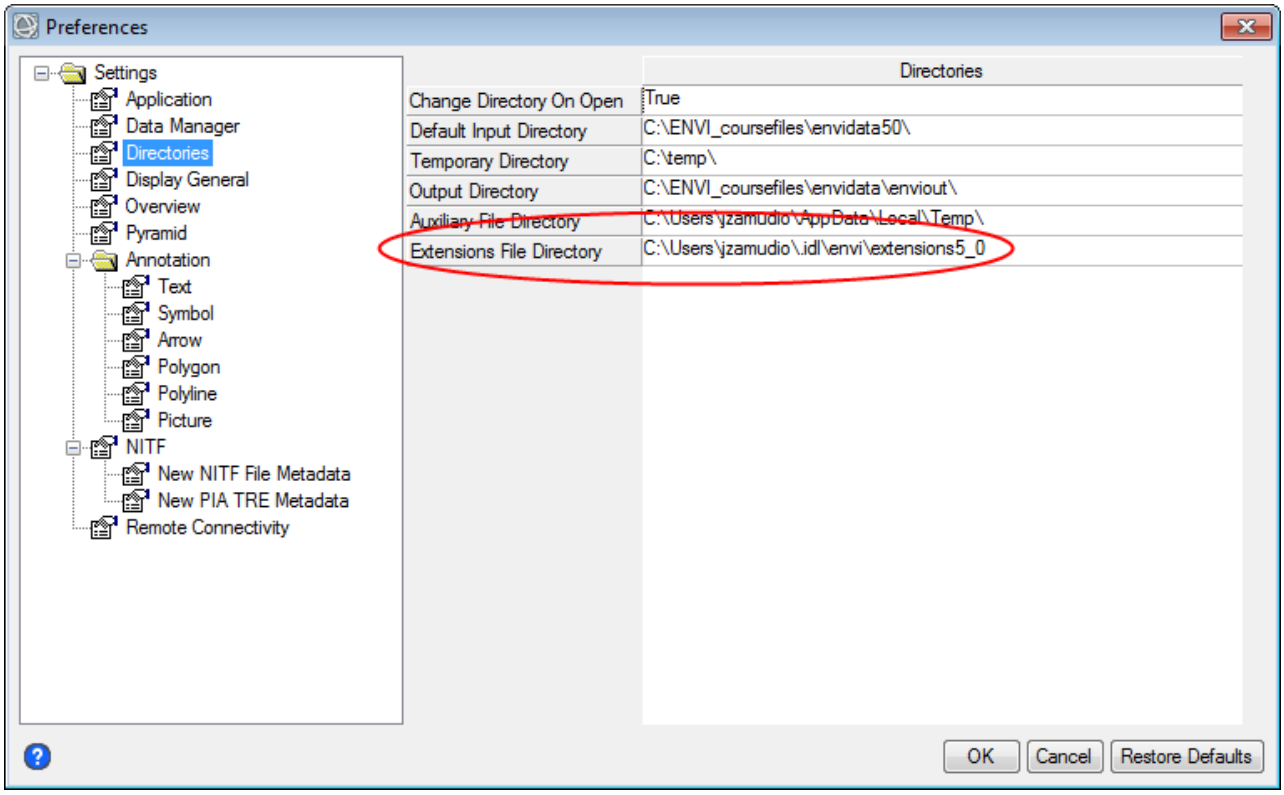
OPTIONAL (for the expert programmer): If you finish early try to call an ENVI_DOIT routine instead of BATCH_ICA inside the FOREACH loop.

Chapter 10 ENVI Extensions

An ENVI extension, in essence, is code that a developer writes to create or expose new functionality to the user through the graphic user interface. From the user's perspective, it's code that they download and add to their extensions folder that adds additional tools to ENVI's toolbox. From the developer's perspective, it's code with a few GUI elements and basic error handling that can be called for different files and with different parameter settings based on different use case scenarios.

Basic Mechanics

You may have noted the presence of the Extensions folder in the ENVI Toolbox. Programs that are placed in the Extensions folder, and that meet the necessary IDL criteria to be recognized by ENVI as extensions are automatically accessible. These routines must end in .pro (text code) or .sav (compiled code), and contain an IDL procedure that will execute when the tool is clicked on in the menu.



The Directories topic under the **File** → **Preferences** contains the file path to the current Extensions File Directory. For your system configuration, it may be necessary to make sure this exists somewhere in your user space so you can modify its contents (on Windows, somewhere under your username folder, and on Unix-based systems like Linux and Mac, somewhere in your home folder).

Note: If you do not have write permissions to the directory, you will need to change the Extensions File Directory path to a location where you have write permissions prior to attempting the final two exercises of this chapter.

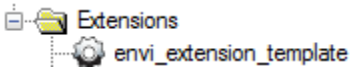
A Template for ENVI Extensions

As with the batch examples, a quick template is provided here so that an experienced IDL programmer can immediately begin converting existing ENVI and IDL programs to ENVI extensions.

```
pro envi_extension_template
  compile_opt idl2

  catch, envi_error
  if envi_error ne 0 then begin
    catch, /cancel
    if obj_valid(e) then $
      e.ReportError, 'ERROR: ', + !error_state.msg
    message, /reset
    return
  endif

  ; Get ENVI session
  e = envi(/current)

  ;Prompt for the input file
  inFile=dialog_pickfile(dialog_parent=e.widget_id, $
                         title='Please select input file', $
                         /must_exist)

  ;Prompt for the output file
  outFile=dialog_pickfile(dialog_parent=e.WIDGET_ID, $
                          title='Please select output file')

  e.ReportError, 'Output file would have been: ' + outFile
end
```

The initial error block is a general purpose ENVI error block that can be used with most ENVI extensions. Note that the method `e.ReportError` is called from the ENVI object declared just below it in the code block (`e = envi(/current)`) and that both uses of GUI elements refer to `e.widget_id` as their parent widget. In ENVI 5.0, many ENVI library routines are handled as methods of the ENVI process, created with `e = envi()` for a new ENVI process or `e = envi(/current)` for an existing ENVI process.

When the above .pro file is copied into the current extensions directory and ENVI is restarted, it will appear in the Extensions folder of the Toolbox:



The new tool name matches the name of the procedure declared as the last (and in this case only) procedure in the file. When we double click it, it will run the code above, prompting the user for an

input file and an output file, then reporting what the output file would have been as an error (as the code currently does nothing else).

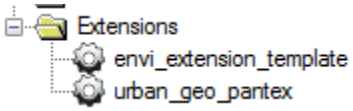
Note: The following code uses some more advanced examples which we will only touch on briefly in this chapter. You will see more on widget programming in the next chapter (Widgets and GUI elements). Individual routines and general topics can be examined in the help as well.

Exercise: Installing an ENVI Extension

1. Using your Operating System, navigate to the `enviout` directory and create a folder called **active_envi_extensions**. You will use this folder as your Extending ENVI directory for the purposes of this course. Normally you would create this extensions folder in your home directory.
2. Using your Operating System browse to **extend\src** select **urban_geo_pantex.pro** and copy it to the **active_envi_extensions** directory you just created.
3. You currently should only have IDL running. To start ENVI go to the command line and type:
   ```
   IDL> e = envi()
   ```
4. In the ENVI interface click **File** → **Preferences** then click on **Directories**.
5. Click to select **Extensions File Directory –** you will see the button  appear on the right side of the current active directory. Click the  to bring up a directory browser, then navigate to and select the **active_envi_extensions** folder you created.
6. Restart ENVI. You can do this from ENVI directly by selecting **File**→ **Exit** or, if you'd like, you can use the IDL command line to do the same:

```
ENVI> .reset
IDL> e = envi()
```

7. Urban_geo_pantex should now be a tool available from the Extensions folder in the Toolbox:



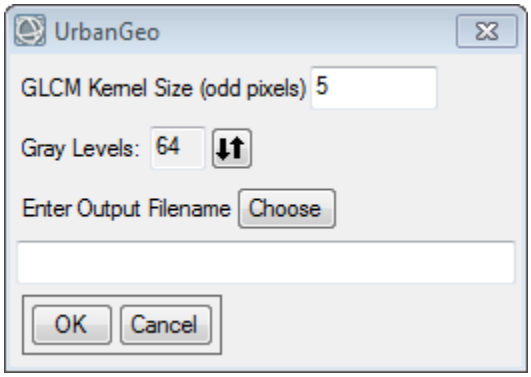
8. Double click on **urban_geo_pantex**. You will be prompted to select an input file. Navigate to `envidata\extend` and select `urban_geo_sub.dat` to run the process on this file. You see this dialog to select the file because of the following code:

```
inFile=dialog_pickfile(dialog_parent=e.widget_id, $
                       title='Please select input file', $
                     /must_exist)
```

9. The file you selected will be loaded into the display. This loading is accomplished by the following code:

```
imageRaster = e.OpenRaster(inFile)
view = e.GetView()
layer = view.CreateLayer(imageRaster)
```

10. As the image is loaded into the display, you will see the UrbanGeo dialog box open. You are prompted to enter three different values, the **GLCM Kernel Size**, the number of **Gray Levels**, and **Enter Output Filename**. Each of these components is put together using auto-managed widgets in ENVI. Auto- managed widgets are not as fully featured as standard IDL widgets, but can be used to quickly build ENVI interfaces when only a few different types of input are required.



The code that creates this dialog box (with explanatory comments) and captures the input from the user is below. This will be discussed more in the widget programming chapter, it is included here only for reference.

```
; Set up gray level list for selection.
gl_list = ['8', '16', '32', '64', '128', '256']

; Set up window as top level base widget.
tlb = widget_auto_base(title='UrbanGeo')

; Make a row for our moving window / kernel size.
row1 = widget_base(tlb, /row)

; Set moving window / kernel size.
k_param = widget_param(row1, $
                       /auto_manage, $
                       default=5B, $
                       dt=1, $
                       floor=3B, $
                       ceil=21B, $
                       prompt='GLCM Kernel Size (odd pixels)', $
                       uvalue='k')
```

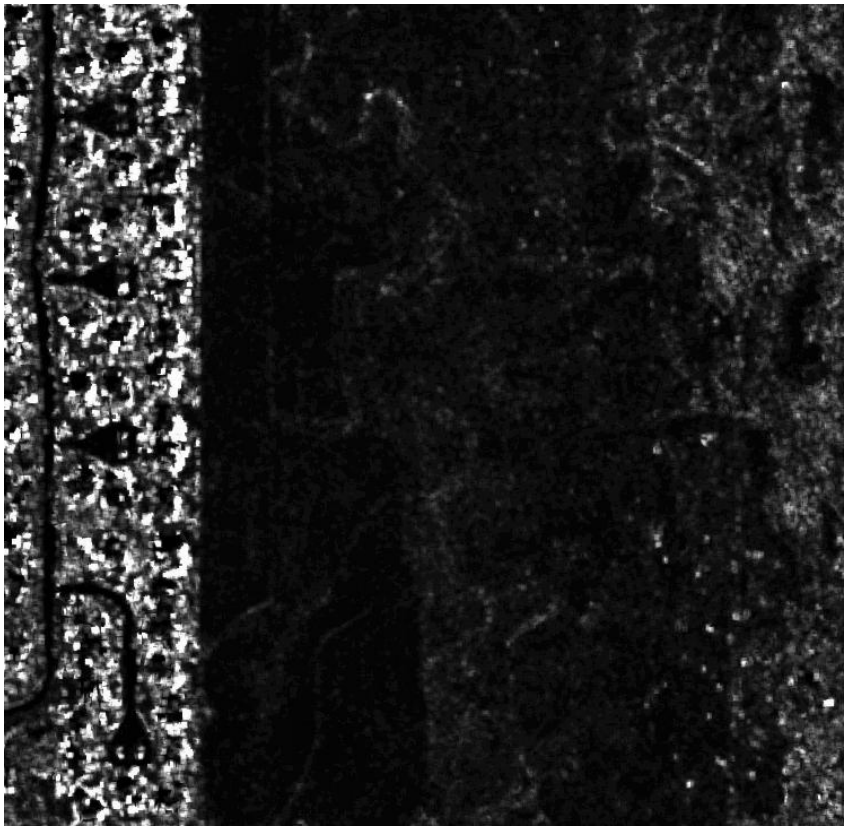
```
; Make a row for the toggle widget.
row2 = widget_base(tlb, /row)

; Toggle Gray Levels for GLCM
gl_toggle = widget_toggle(row2, $
                          /auto_manage, $
                          default=3, $
                          list=gl_list, $
                          prompt='Gray Levels: ', $
                          uvalue='gl_select')

; Prompt user to select output file.
out_f = widget_outf(tlb, $
                    /auto_manage, $
                     uvalue='out_file')

; Call the auto-managed widget manager.
result = auto_wid_mng(tlb)
if result.accept eq 0 then return
if (result.k mod 2B) ne 1B then message, 'Kernel size must be odd.'
```

11. Click **Choose**, browse to your output directory, and type `pantex_test.dat` for the file name. Then click **Open**. Click **OK** to run the program.
12. You will see several different processes run with progress bars. PanTex actually performs several GLCM (Gray-Level Co-occurrence Matrix) processes then calculates a minimum result through band math. Because this code is modularized, the actual processing takes place in a different procedure. This procedure is called by the following line:

```
    ug_pantex, ENVIRasterToFID(imageRaster), $
        result.out_file, $
        kernel=result.k, $
        g_levels = gl_list[result.gl_select]
```

Each of our widgets stored the user input in the structure labeled `result`. In calling the PanTex algorithm programmatically, all we have to do to retrieve each of these values is dereference them from the structure, as with `result.out_file`. If the user inputs `'c:\scratch\test.dat'` as their output file, then `result.out_file` resolves to `'c:\scratch\test.dat'` and this value is passed to the UG_PANTEX procedure where it is used to determine the location to write the output.

13. When the process is finished, you will see the result added to the View:

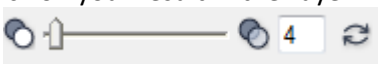


This result is added by code similar to that in step 9:

```
; Add result to display
resultRaster = e.OpenRaster(result.out_file)
layer = view.CreateLayer(resultRaster)
```

Note: view was declared previously in the code (Step 9) with:

```
view = e.GetView()
```

14. The PanTex algorithm is meant to highlight built-up areas. Click on your result in the Layer Manager to make it active, then use the transparency slider:  to look at the relationship between the originating data and its output.
15. When finished, right click on **View** and select **Remove All Layers**.
16. Minimize the IDL Workbench.

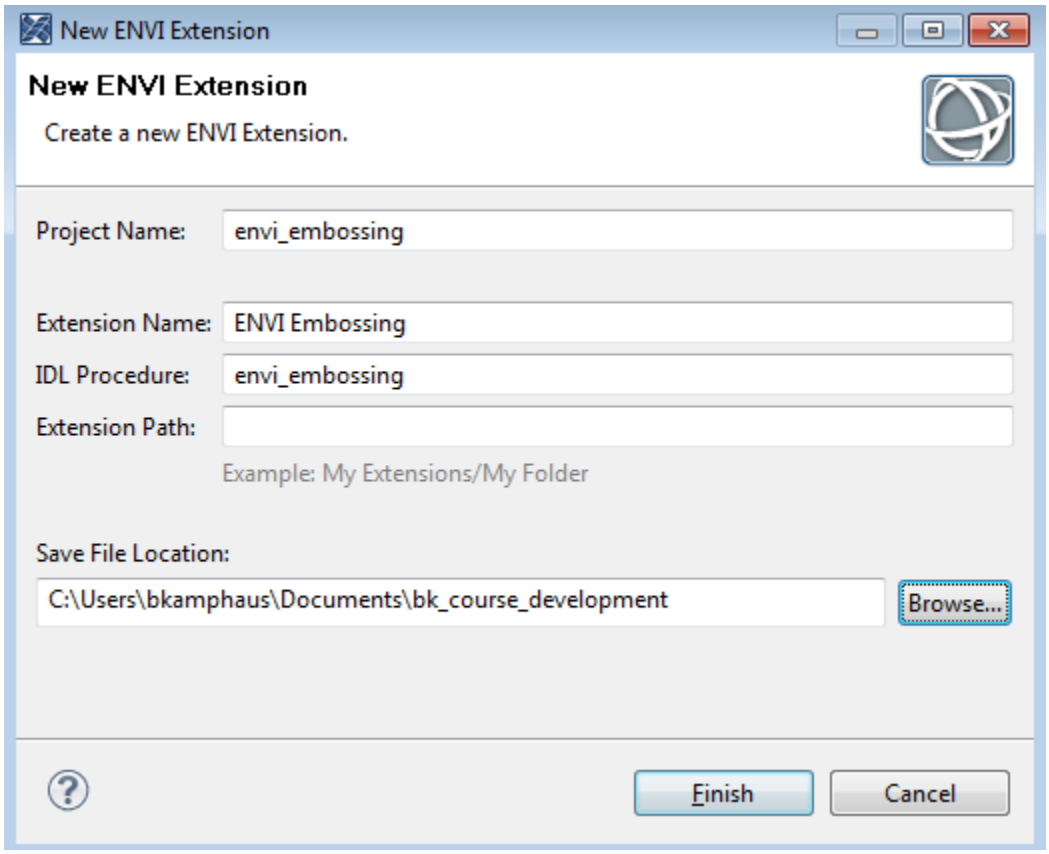
Example 1: Embossing

Now that we've seen the process of installing and using an ENVI extension, we'll look at the steps it takes to author one ourselves. For our first example, we're going to take our tile-based processing example from the Object API chapter and create an extension using copy and paste development, the ENVI Extensions Wizard, and some basic IDL programming practices. Once we've seen the basic process,

we'll produce an ENVI extension from scratch in our second example, which will create a brightness normalized image from a multispectral image.

Using the ENVI Extensions Wizard

If your copy of IDL has ENVI installed, you will see the option **File > New ENVI Extension**. Click this menu option now, and the following dialog will appear:



Fill out the fields as in the above example, except change the **Save File Location** to a location on your own disk where you have write permissions. When done, click **Finish**. A program titled envi_embossing.pro will be created and it will contain the following text:

```
; Add the extension to the toolbox. Called automatically on ENVI
startup.
pro envi_embossing_extensions_init

  ; Set compile options
  compile_opt IDL2

  ; Get ENVI session
  e = envi(/current)

  ; Add the extension to a subfolder
```

```
  e.AddExtension, 'ENVI Embossing', 'envi_embossing', path=''
end

; ENVI Extension code. Called when the toolbox item is chosen.
pro envi_embossing

  ; Set compile options
  compile_opt idl2

  ; General error handler
  catch, err
  if (err ne 0) then begin
    catch, /cancel
    if obj_valid(e) then $
      e.reportError, 'ERROR: ' + !error_state.msg
    message, /reset
    return
  endif

  ;Get ENVI session
  e = envi(/current)

;****************************************
; Insert your ENVI Extension code here...
;****************************************
; Create an ENVIRaster
  my_ui = e.ui
  my_raster = my_ui.SelectInputData(/raster)

  ; Create an output raster
  new_file = e.GetTemporaryFilename()
  output_raster = e.CreateRaster(new_file, inherits_from=my_raster)

  ; Iterate through the tiles of the original data set
  tiles = my_raster.CreateTileIterator()
  foreach tile, tiles DO BEGIN
    data = emboss(tile, /EDGE_WRAP)
    output_raster.SetTile, data, tiles
  endforeach

  ; Save the data
  output_raster.Save

  ; Visualize original file and results.
  view = e.GetView()
```

```
  lyr2 = view.CreateLayer(output_raster)
  lyr1 = view.CreateLayer(my_raster)
  port = view.CreatePortal(layer=lyr2)
end
```

As you can see, the ENVI extension wizard will generate boiler plate code that does a few things. First, it adds an extension to the folder specified in the **Extension Path** directory, so in our case:

```
  e.AddExtension, 'ENVI Embossing', 'envi_embossing', PATH=''
```

Because no path was specified. If we had specified a path, such as "Course Extensions" in the wizard, we would have seen code like:

```
  e.AddExtension, 'ENVI Embossing', 'envi_embossing', $
  PATH='Course Extensions'
```

Now, the second string here, 'envi_embossing' stores the name of the entry point procedure which will be called when the user clicks the button that has been added to their ENVI Toolbox, and for that reason, the entry point procedure in our file (And the file itself) are named 'envi_embossing' as well:

```
pro envi_embossing
```

Only two sections in the code have been auto-completed with boiler plate:

- The generic error handling block:

```
  ; General error handler
  catch, err
  if (err ne 0) then begin
    catch, /CANCEL
    if obj_valid(e) then $
      e.reportError, 'ERROR: ' + !error_state.msg
    message, /RESET
    return
  endif
```

There is a more detailed section on error checking in the second extension we'll write. For this first extension, we're relying on a 'general error handler.' This means that for any given error, ENVI will do the same thing – cancel the process, report the error message (recorded in the system state variable !ERROR_STATE.MSG) and then return to the calling level of the program – in this case, our main ENVI program loop.

- The generation of a reference to our current ENVI session:

```
;Get ENVI session
e = ENVI(/CURRENT)
```

You've seen this a few times already – this just gives us the reference to the main application which can be to control it. Now navigate to the course program EMBOSS_BY_TILE.PRO that you saw in the object API chapter and open it in an editor window.

The first section, COMPILE_OPT IDL2, and the generation of an ENVI reference are already included in the boiler plate code, so we'll select the following code:

```
; Create an ENVIRaster
bldr_pan_file = filepath('qb_boulder_pan', root_dir=e.root_dir, $
                subdirectory = ['data'])
bldr_pan_raster = e.OpenRaster(bldr_pan_file)


; Create an output raster
new_file = e.GetTemporaryFilename()
output_raster = e.CreateRaster(new_file, $
                  inherits_from=bldr_pan_raster)

; Iterate through the tiles of the original data set
bldr_tiles = bldr_pan_raster.CreateTileIterator()
foreach tile, bldr_tiles DO BEGIN
  data = emboss(tile, /EDGE_WRAP)
  output_raster.SetTile, data, bldr_tiles
endforeach

; Save the data
output_raster.Save

; Visualize original file and results.
view = e.GetView()
lyr2 = view.CreateLayer(output_raster)
lyr1 = view.CreateLayer(bldr_pan_raster)
port = view.CreatePortal(layer=lyr2)
```

and paste it into our new extension under the section labeled:

```
;*****************************************
; Insert your ENVI Extension code here...
;*****************************************
```

```
; PASTE HERE
end
```

Getting User Input

If a user ran our extension right now, it would always run using the panchromatic Quickbird scene of Boulder, CO included with the ENVI distribution. This isn't very useful! What we want the user to be able to do is to select the file of their choice. To facilitate this, we're going to make a couple of edits to our original code using ENVI's UI object (part of the main application).

First, let's work with the UI object interactively:

```
IDL> e = envi()
ENVI> my_ui = e.ui
```

After typing these lines, MY_UI will contain a reference to some of ENVI's user interface functionality contained within the UI object. Now we will use the SelectInputData method of the user interface object:

```
ENVI> my_raster = my_ui.SelectInputData(/raster)
```

This will return a raster object (indicated by the /RASTER argument). Navigate to the location of the Boulder panchromatic image and select it. To double check your result, use the print command and pass in the raster object (which has been overloaded to supply a print behavior):

```
ENVI> print, my_raster
ENVIRASTER <152124>
  AUXILIARY_SPATIALREF      = !NULL
  AUXILIARY_URI             = <Array[2]>
  DATA_TYPE                 = 'uint'
  INTERLEAVE                = 'bsq'
  METADATA                  = <ObjHeapVar152140(ENVIRASTERMETADATA)>
  NBANDS                    = 1
  NCOLUMNS                  = 4096
  NROWS                     = 4096
  PYRAMID_EXISTS            = 1
  READ_ONLY                 = 1
  SPATIALREF                = ...
  URI                       = 'C:\Program…
                             …Files\Exelis\ENVI50\data\qb_boulder_pan'
```

If we need access to the file's fully qualified path name, we can get access to it by typing:

```
ENVI> print,my_raster.uri
C:\Program Files\Exelis\ENVI50\data\qb_boulder_pan
```

Let's return to our editor window and make the modifications necessary to include this in our code. Locate the section that currently looks like this:

```
; Create an ENVIRaster
bldr_pan_file = filepath('qb_boulder_pan', root_dir=e.root_dir, $
                  subdirectory = ['data'])
bldr_pan_raster = e.OpenRaster(bldr_pan_file)
```

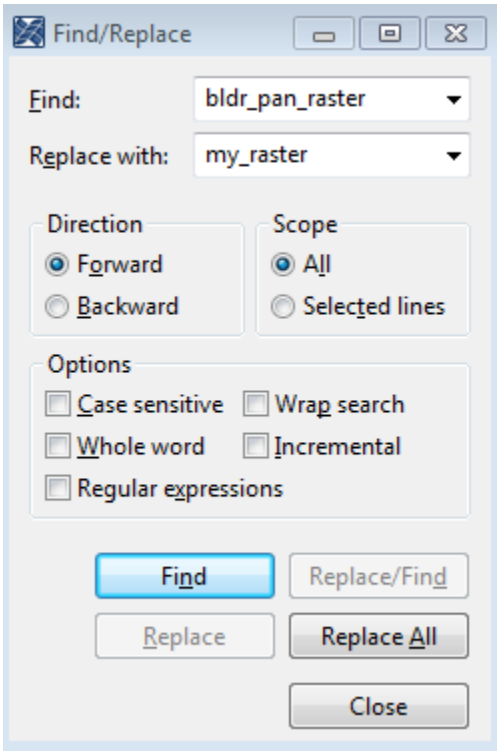
and comment it out by pressing CTRL + ; with the section selected. We don't want this to be hard-coded! Use either your command line history or the console, and drag the lines from above into the editor window:

```
ENVI> my_ui = e.ui
ENVI> my_raster = my_ui.SelectInputData(/raster)
```

Change the preceding comment to be more appropriate. These changes should give you something like this:

```
  ; Get the input raster from the user.
  my_ui = e.ui
  my_raster = my_ui.SelectInputData(/raster)
```

But wait, we have references to bldr_pan_raster elsewhere! We don't want to break our code, so let's refactor using the Replace All tool. Key in CTRL + F to bring up the **Find/Replace** dialog:



Enter the settings above, setting **Find** to **BLDR_PAN_RASTER** and **Replace With** to **my_raster** and then click **Replace All**. After the changes have gone through, click **Close**.

We do have one area that also includes BLDR in its name, BLDR_TILES in our FOREACH loop:

```
  ; Iterate through the tiles of the original data set
  bldr_tiles = my_raster.CreateTileIterator()
  foreach tile, bldr_tiles DO BEGIN
    data = emboss(tile, /EDGE_WRAP)
    output_raster.SetTile, data, bldr_tiles
  endforeach
```

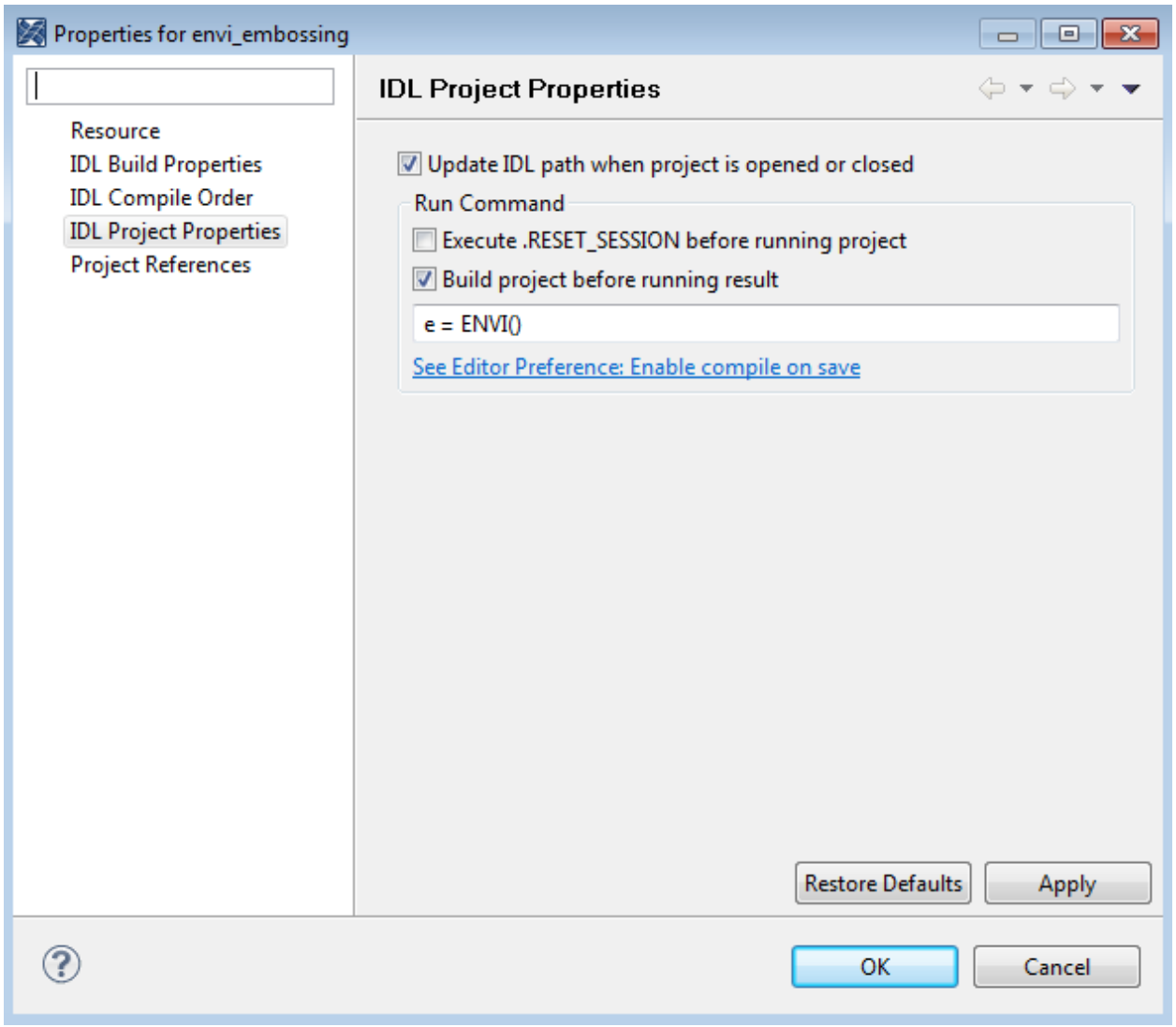
Now, it's not necessary to change this to make the code portable – despite our specifically named variable, it will appropriately store a tile iterator for any file, whether over Boulder or not. It is a little 'tacky' to leave this the same, though, and it also may damage code readability if anyone else in our organization has to go through and make edits. So let's just change it to the very non-specific TILES:

```
  ; Iterate through the tiles of the original data set
  tiles = my_raster.CreateTileIterator()
  foreach tile, tiles DO BEGIN
    data = emboss(tile, /EDGE_WRAP)
    output_raster.SetTile, data, tiles
  endforeach
```

To make sure the code is relatively error free, click **Compile**. It should compile successfully with no error messages if everything has been changed correctly.

Our code is now generally portable and ready to install. To deploy it, first we're going to build it using the IDL Workbench's automatic tool for building SAV files, then we're going to locate our extensions directory and copy it there (as we did in the previous exercise).

Right click on the envi_embossing project and navigate to IDL Project Properties. Here we can see that some ENVI specific settings have already been added to the project. This was done when we used the ENVI extension wizard to create the project.



Right click on the envi_embossing project in the **Project**… tab in the workbench and click **Build Project**. You can also click Ctrl + B with the project selected. You should see console output like the following:

```
*** Executing .RESET_SESSION
*** Compiling files...
*** .COMPILE
'C:\Users\bkamphaus\IDLWorkspace82\envi_embossing\envi_embossing.pro'
% Compiled module: ENVI_EMBOSSING_EXTENSIONS_INIT.
% Compiled module: ENVI_EMBOSSING.
*** Compile complete: Time = 0.02s
*** Running build command: RESOLVE_ALL, /CONTINUE_ON_ERROR,
SKIP_ROUTINES='envi'
% Compiled module: RESOLVE_ALL.
% Compiled module: EMBOSS.
*** Creating save file...
% SAVE: Portable (XDR) SAVE/RESTORE file.
```

```
% SAVE: Saved procedure: ENVI_EMBOSSING.
% SAVE: Saved procedure: ENVI_EMBOSSING_EXTENSIONS_INIT.
% SAVE: Saved procedure: RESOLVE_ALL.
% SAVE: Saved procedure: RESOLVE_ALL_CLASS.
% SAVE: Saved function: EMBOSS.
% SAVE: Saved function: RESOLVE_ALL_BODY.
% SAVE: Saved function: UNIQ.
*** Save File  At:
C:\Users\bkamphaus\Documents\bk_course_development\envi_embossing.sav
*** Build complete: Time = 0.49s
```

Note the last line prior to the build complete message, in this case:

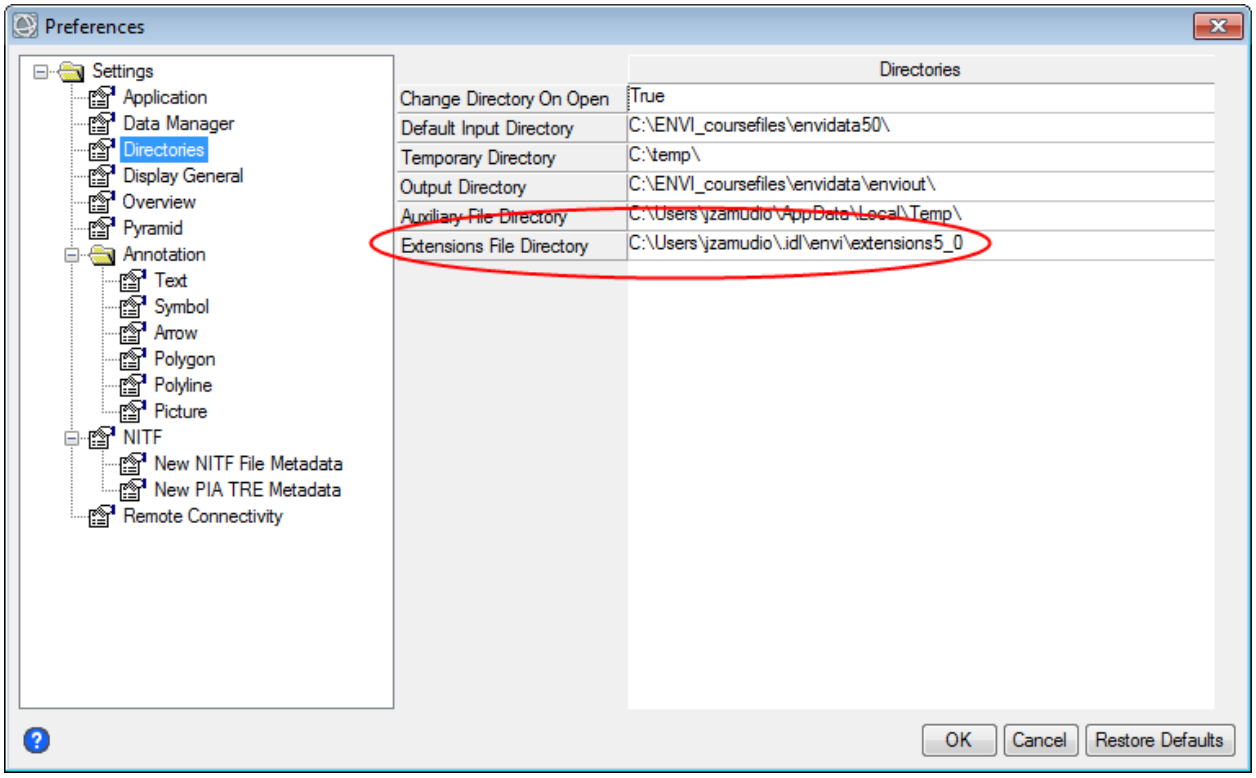
```
C:\Users\bkamphaus\Documents\bk_course_development\envi_embossing.sav
```

This line will contain the fully qualified path to the location of your output file and will be different from the location above. Bring up a windows explorer dialog box, or whichever file management system you prefer to use. Locate the first file and keep the explorer window open.

Double check the location of your ENVI extensions directory in your preferences. You may have to restart your envi session with:

```
IDL> e = envi()
```

Go to **File > Preferences** in the menu, which will bring up the following dialog:

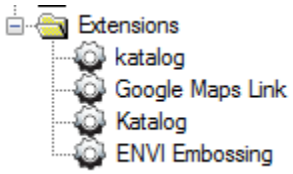


Copy the file you selected in the previous step to the location you just now noted.

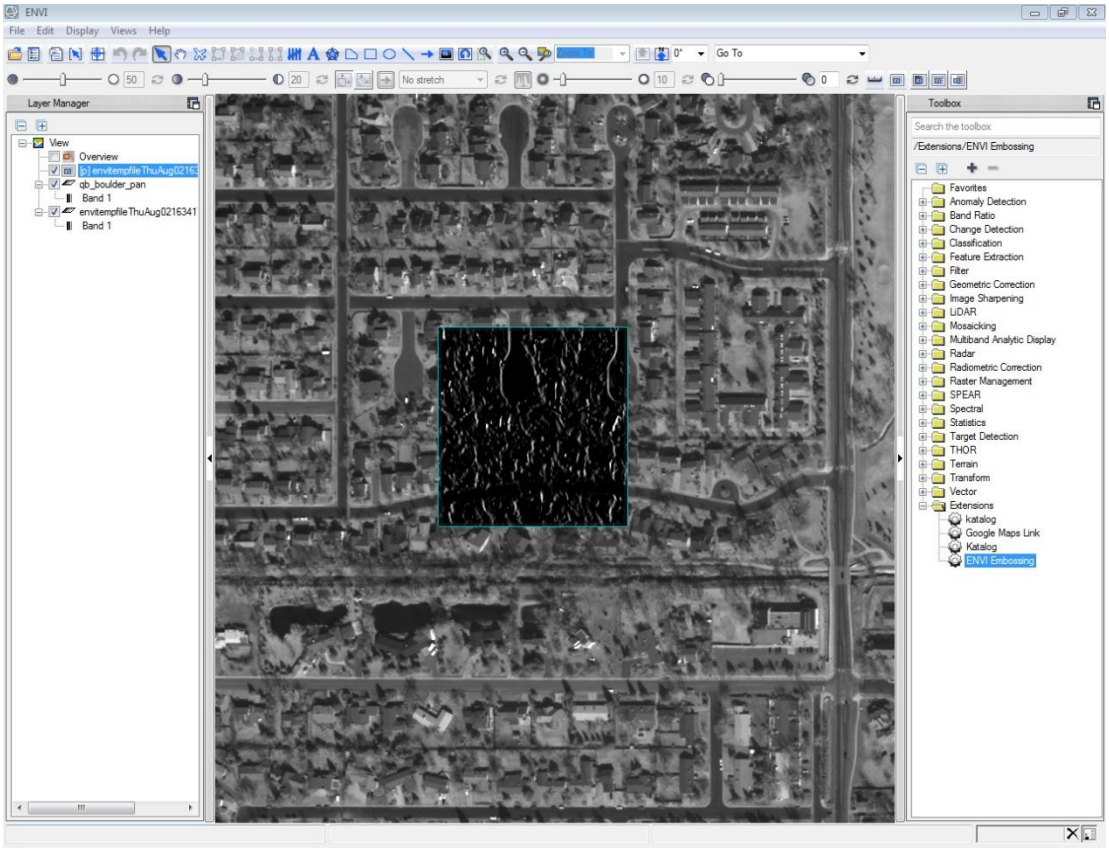
When the file has been copied to the new location, type the following code:

```
ENVI> .reset
IDL> e = envi()
```

Look under the **Toolbox** view in the new ENVI session and expand the **Extensions** directory. You should see “ENVI Embossing” added to your list of options:



Double click on this tool. Select the qb_boulder_pan image in C:\Program Files\Exelis\ENVI50\data and click **Open**, then click **OK**. It will run the tool and load the result to the display as defined in the code:



Exercises

1. Because we used the previous code, ENVI is currently writing the output as a temp file. Using DIALOG_PICKFILE, alter the code to prompt the user to select the name of an output file and write the output file to this location instead. You may need to reference the sections on file

access, ENVI's procedural and object API's, and the earlier example. Once this change has been made, redeploy the code as an extension.

Example 2: Brightness Normalization

In the previous example, we used the ENVI Extension wizard to aid us in generating a boiler plate and copied and pasted previously generated functionality to create our extension. Here we'll be coding an extension from scratch to build a general purpose tool.

The name of this extension will be 'Brightness Normalization' and our entry point procedure will be BRIGHTNESS_NORMALIZE. This algorithm will be modeled after Changshan Wu's 2004 method for monitoring urban composition (see reference information for Wu's article in the "References" section of this chapter). Wu's approach requires that each pixel be normalized for its total albedo or brightness prior to being analyzed. To do this, each pixel's value per band is divided by its mean value for all bands.

Let's take the case of a Landsat TM image. Any given pixel has six values (excluding the thermal bands). Assuming our image has already been calibrated to at-ground or exo-atmospheric reflectance, these values would be R_{485}, R_{565}, R_{660}, R_{825}, R_{1650}, and R_{2200}. To find for NR_{485} and so on, we divide R_{485} by the mean of R from wavelengths 485:2200, or, $NR_{485} = R_{485} / ((R_{485} + R_{565} + R_{660} + R_{825} + R_{1650} + R_{2200}) / 6)$. We then repeat this for each pixel at each band value until the entire image has been normalized.

In terms of IDL code, if we simulate one spectral slice for a given pixel, we'd be looking at something like this:

```
ENVI> pixel = [420, 476, 380, 792, 688, 588]
ENVI> norm_pixel = pixel / mean(pixel)
ENVI> print, norm_pixel
     0.753589     0.854067     0.681818      1.42105      1.23445
     1.05502
```

For efficient processing we're going to need to accommodate ENVI's spectral tiling system. ENVI's RasterIterator can be set for 'spectral' mode, and if this is set, it returns two dimensional tiles in the form:

```
[range(columns dimension), range(bands dimension)]
```

Let's look at one of these spectral tiles. Launch ENVI from the IDL command line if it's not already open using the standard:

```
IDL> e = envi()
```

Now, let's open the course file boulder-ETM.dat

```
ENVI> bldr_etm = e.OpenRaster(file_which('boulder-ETM.dat'))
```

Let's use HELP to make sure BLDR_ETM is a successfully opened ENVIRaster object:

```
ENVI> help, bldr_etm
```

```
BLDR_ETM        ENVIRASTER <157088>
```

Assuming it looks correct, let's create a tile iterator:

```
ENVI> etm_tiles = bldr_etm.CreateTileIterator(mode='spectral')
ENVI> help, etm_tiles
ETM_TILES       ENVIRASTERITERATOR <157089>
```

Then do a test tile iteration:

```
ENVI> tile1 = etm_tiles.Next()
ENVI> help, tile1
TILE1           BYTE      = Array[575, 6]
```

Ok, there's our [COLUMNS, BANDS] tile, as expected. Now, let's create a vector of mean values per pixel:

```
ENVI> tile1_means = mean(tile1, dimension=2)
ENVI> help, tile1_means
TILE1_MEANS     FLOAT     = Array[575]
ENVI> print, tile1_means[0:3]
      67.8333      64.0000      66.5000      73.8333
```

Now we'll compute the normalization for each pixel.

```
ENVI> dims = size(tile1)
ENVI> norm_tile = fltarr(dims[1], dims[2], /nozero)
ENVI> for i=0, dims[2]-1 do norm_tile[*,i] = tile1[*,i] / tile1_means
ENVI> help, norm_tile
```

We can print out one pixel from each of these arrays to make sure the process is running as designed:

```
ENVI> print, tile1[0,*], tile1_means[0], norm_tile[0,*]
  75, 64, 60, 70, 83, 55
  67.8333
  1.10565, 0.943489, 0.884521, 1.03194, 1.22359, 0.810811
```

This looks like our expected set of normalized values. Let's open an editor window and create our first routine inside of it. We'll design it as a procedure that takes an input ENVIRaster and output ENVIRaster, performs spectral tiling, and computes brightness normalization for each pixel value of each band from the input ENVIRaster, writing it to the output ENVIRaster. You can use the code from ENVI_EMBOSSING as a guide to create the tile iterator if need be. The results should be similar to the below code:

```
pro brightness_normalize_by_tile, input_raster, output_raster
  compile_opt idl2

  ; Iterate through the tiles of the original data set and normalize
```

```
  ; for each pixel value at each band.
  tiles = input_raster.CreateTileIterator(mode='spectral')
  foreach tile, tiles do begin

    ; Calculate the mean for all band values for each pixel
    tile_mean = mean(tile, dimension=2)

    ; Get the dimensions of the tile
    tile_dims = size(tile)

    ; Create an empty container array to store normalized values.
    norm_tile = fltarr(tile_dims[1], tile_dims[2], /nozero)

    ; Divide each pixel for each band by the pixel's mean,
    ; one column at a time.
    for i=0, tile_dims[2]-1 do norm_tile[*,i] = tile[*,i] / tile_mean

    ; Store the results in the output raster.
    output_raster.SetTile, norm_tile, tiles
  endforeach

  ; Save the output raster.
  output_raster.Save
end
```

We'll save this as BRIGHTNESS_NORMALIZE_BY_TILE.PRO, but we'll be putting everything in a new file eventually. For now, though, we want to test running this process on the entire image. We should still have our Boulder Landsat ENVIRaster available:

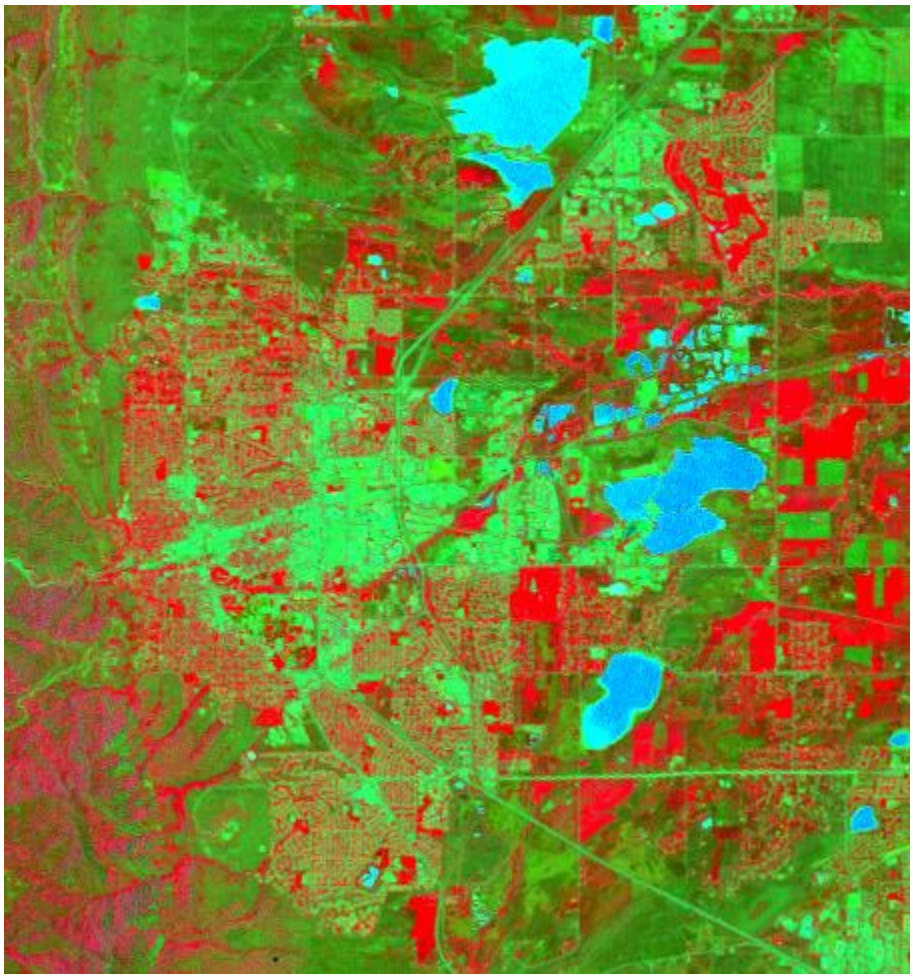
```
ENVI> help, bldr_etm
BLDR_ETM        ENVIRASTER <157088>
```

So now we'll use it to test our function:

```
ENVI> new_file = e.GetTemporaryFilename()
ENVI> output = e.CreateRaster(temp, inherits_from=bldr_etm,
data_type=4)
ENVI> brightness_normalize_by_tile, bldr_etm, output
```

You can then use the **Data Manager** in ENVI to load and display the results. Here I've loaded a CIR composite of the imagery, which enhanced contrast between vegetation, water, impervious surface, and soil. Feel free to experiment with other band combinations.

NOTE: One downside you might notice at the moment is that there is no progress bar or dialog box telling us when processing is completed or how long it will take. We'll see approaches to this in the next chapter on GUI Programming in ENVI.



Ok, so now we have implemented a custom algorithm from a research journal in ENVI, and we can call it from the command line, but what we *really* want to do is make this available to an end user with no programming expertise. To do this, we're going to add two other procedures to our brightness_normalize_by_tile.pro code, and then save it under a different name. The first we will put after our current procedure (this one needs to be the last defined procedure in the file):

```
pro brightness_normalize
  compile_opt idl2
  ; more here later
End
```

The second we'll put above BRIGHTNESS_NORMALIZE_BY_TILE

```
pro brightness_normalize_extensions_init
  compile_opt idl2
  ; more here later
end
```

The names of both of these are important. Now that we've added our procedures, click **File > Save As** – you will be prompted to select brightness_normalize.pro as the new name of your file. This is because ENVI sees that it is now the final procedure in the file and therefore *should* be the entry point procedure (assuming good programming practices are followed).

Now, the first procedure we'll adjust is the second one we added, BRIGHTNESS_NORMALIZE_EXTENSIONS_INIT. This name must match the file name – so the formula for determining this name is EXTENSIONS_FILENAME_EXTENSIONS_INIT. Remember this, as we'll need to keep track of this to build our save file later!

If you remember the code the ENVI extension wizard generated for us before during the ENVI_EMBOSSING exercise, we'll be doing something similar. Let's review that code now:

```
  ; Get ENVI session
  e = envi(/CURRENT)

  ; Add the extension to a subfolder
  e.AddExtension, 'ENVI Embossing', 'envi_embossing', PATH=''
```

So, we just need to get a reference to the current ENVI session, and then add an extension, providing as string arguments the name of the extension and the name of the entry point procedure:

```
pro brightness_normalize_extensions_init
  compile_opt idl2
  ; Initialize extensions
  e = envi(/current)
  e.AddExtension, 'Brightness Normalization','brightness_normalize', $
    path='Custom Algorithms'
end
```

Just a few simple adjustments and our extension is set to initialize correctly. Now let's take a look at our BRIGHTNESS_NORMALIZE procedure, as this is what will be called when the user clicks the button. First, we should include a general error handler (more on this shortly).

```
  ; General error handler
  catch, err
  if (err ne 0) then begin
    catch, /CANCEL
    if obj_valid(e) then $
```

```
      e.ReportError, 'ERROR: ' + !error_state.msg
    message, /RESET
    return
  endif
```

As in the previous exercise, any error that the user hits trying to run our process from ENVI will result in the program reporting an error and returning to the main level. Next, we need the user to select an input file, which we can do in a similar way to our previous exercise:

```
; Get the input raster from the user.
e = envi(/current)
my_ui = e.ui
my_raster = my_ui.SelectInputData(/raster)
```

Now, we will prompt the user to select the name of an output raster. Previously we used:

```
new_file = e.GetTemporaryFilename()
```

But here we don't want just any temporary filename, we want the user to specify the location and file name of their choice. We'll use DIALOG_PICKFILE() to meet this requirement:

```
; Create an output raster based on user input
new_file = dialog_pickfile(dialog_parent=e.widget_id, $
                           title='Output Normalized Image')
output_raster = e.CreateRaster(new_file, $
       inherits_from=my_raster, data_type=4)
```

This will prompt the user to select a file. Highlight the lines that include dialog_pickfile, right click, and select **Run Selected Text** or use the keyboard shortcut **Shift + F8** in the workbench. You will be prompted to enter a name. After doing so, you can use print to see that you'll be getting a fully qualified path name:

```
ENVI> print, new_file
C:\Users\envi_training\blah.dat
```

The second part of this code block creates a new raster object from the user's selection. Although we are inheriting most properties from the image the user selects, we're overwriting the DATA_TYPE keyword and setting it to 4. This is because our program assumes the output will be floating point, but the user could select an input raster built with any variety of data types. If we try to write floating point values from 0 to 1 to a byte raster, for example, our results won't be correct.

Now we just need to call our brightness_normalize_by_tile procedure:

```
; Call our custom brightness normalization algorithm
```

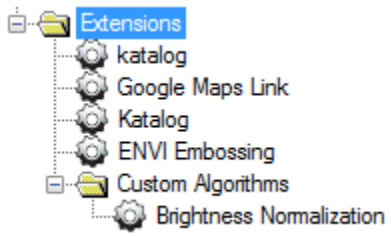
```
  brightness_normalize_by_tile, my_raster,  $
      output_raster
```

Just one line! This is the value of writing modular code. Give the new .PRO with all three completed procedures a test compile. You should see:

```
ENVI> .compile -v 'C:\Users\bkamphaus\brightness_normalize.pro'
% Compiled module: BRIGHTNESS_NORMALIZE_EXTENSIONS_INIT.
% Compiled module: BRIGHTNESS_NORMALIZE_BY_TILE.
% Compiled module: BRIGHTNESS_NORMALIZE.
```

If for any reason you see errors here check what you have against the course file BRIGHTNESS_NORMALIZE_V1.PRO in the course files.

For testing purposes, when ENVI is open, we can actually call our initialization procedure from the command line:



As long as BRIGHTNESS_NORMALIZE is compiled in memory and our initialization procedure is correctly written, we can now use our extension. Test the program now.

NOTE: we didn't write code to add results to the display, so you'll have to check the **Data Manager** in ENVI to find them after the process finishes running. We did do this with our previous example, however – making the edit would be rather simple and is one of the exercises at the end of the chapter.

Now that we've got a functioning extension, let's play the role of the misbehaving user and try to see if we can break it. Double click on **Custom Algorithms > Brightness Normalization** again and navigate to C:\Program Files\Exelis\ENVI50\data and select qb_boulder_pan as the input file. Name the output file whatever you wish and click **OK**.

Nothing seems to happen! Your IDL workbench may also go unresponsive, although you can still click buttons in ENVI. Click **File > Exit** in ENVI. Restart ENVI and add your extension:

```
IDL> e = envi()
ENVI> .compile -v 'C:\Users\bkamphaus\brightness_normalize.pro'
% Compiled module: BRIGHTNESS_NORMALIZE_EXTENSIONS_INIT.
% Compiled module: BRIGHTNESS_NORMALIZE_BY_TILE.
% Compiled module: BRIGHTNESS_NORMALIZE.
ENVI> brightness_normalize_extensions_init
```

Try again with the same input file. Wait a while, and eventually you will get a completely black image (you will have to load it using the **Data Manager**). Any stretch you will apply will look the same. Bring up **Display > Cursor Value** and navigate through the image. The entire image has been written with the value 1.0.

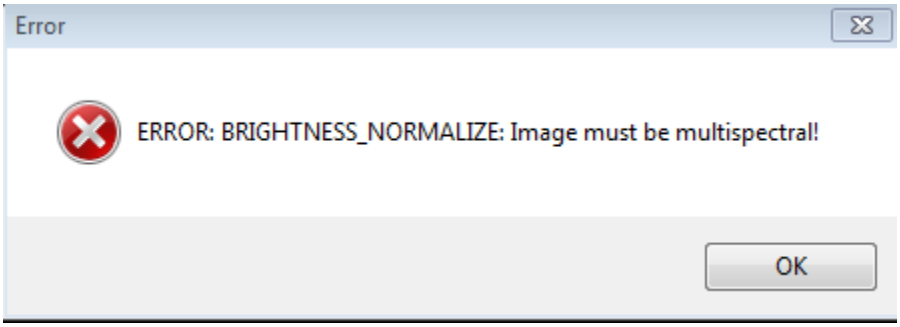
So it's possible that at this point you may be screaming that you know the answer to the manual and yes, in fact, the problem is that we chose a panchromatic image to run a routine which divides each pixel value per band by the average pixel value across bands. With one band, we're just dividing every pixel value by itself, so we produce an image with a value of 1.0 across the board. That's not useful!

Rudimentary Error Handling

We'll make one basic edit that will work in conjunction with our previous general error handler. Navigate to the location where the user selects an input file and add the following logic after the selection is made:

```
; Get the input raster from the user.
e = envi(/current)
my_ui = e.ui
my_raster = my_ui.SelectInputData(/raster)

; Make sure the input raster is acceptable
if my_raster.nbands lt 2 then $
  message, "Image must be multispectral!"
```

Now, save and compile the new image. Double click **Custom Algorithms > Brightness Normalization** in the **Toolbox** again, and select a panchromatic image again. You should immediately be prompted with an error dialog:



We have now controlled what actions the user can take with our software and given them a message they can use to correct mistakes they make when using our tool, all without crashing their software.

Here is a quick rundown of how this process works. Whenever MESSAGE is called, its default behavior is to trigger an error state and to set the error state's message to whatever string is passed as the first argument. If we caught this prior to the catch block being evoked, we could see the error state in the !error_state.msg system variable field.

```
ENVI> print,!error_state.msg
```

Because we have a catch block enabled, however, we will return to the catch block (starting with CATCH, ERR) when we first trigger this error state. Let's review this code block again:

```
  ; General error handler
  catch, err
  if (err ne 0) then begin
    catch, /CANCEL
    if obj_valid(e) then $
      e.ReportError, 'ERROR: ' + !error_state.msg
    message, /RESET
    return
  endif
```

So, we enter this code block with an error present, and start working our way into the IF block. We cancel the current error state with CATCH, /CANCEL (otherwise we might get stuck in an infinite loop of errors), check to make sure ENVI hasn't completely crashed on us (we have a valid ENVI object we can use to do things), and if that additional condition is met, we then use the ReportError method on the ENVI object to report the error state back to the user. I can include any text here when I report an error:

```
ENVI> e.ReportError, 'Ok, now the user is just generating errors on
purpose.'
```

And it will use an error dialog box to report a message to the user. I can also trigger an error anywhere I want in my program with MESSAGE and let e.ReportError just echo back the error state:

```
ENVI> message, "You broke my pretty extension!"
% $MAIN$: You broke my pretty extension!
% Execution halted at: $MAIN$
ENVI> e.ReportError, !error_state.msg
```

This also will report error states defined by the system, although these don't always make sense to the end user:

```
ENVI> print, blah
% PRINT: Variable is undefined: BLAH.
% Execution halted at: $MAIN$
ENVI> e.ReportError, !error_state.msg
```

Distributing Extensions with SAV files

In order to distribute extensions to other users, it is best to use the .SAV format. The .SAV format can store multiple routines and does so in a way that hides the source code from the end user, unlike a .PRO. It can also be distributed to a user without an ENVI + IDL license, whereas .PRO code requires an ENVI + IDL license to run.

This only takes a few steps to accomplish:

- Reset to a clean IDL session.
- Compile all the code required by the extension.
- Run RESOLVE_ALL to compile all dependencies.
- Run SAVE to save the routine.

At the command line here run through these steps:

```
IDL> .reset
IDL> .compile brightness_normalize
% Compiled module: BRIGHTNESS_NORMALIZE_EXTENSIONS_INIT.
% Compiled module: BRIGHTNESS_NORMALIZE_BY_TILE.
% Compiled module: BRIGHTNESS_NORMALIZE.
IDL> resolve_all, /continue_on_error, skip_routines='envi'
% Compiled module: RESOLVE_ALL.
IDL> save, /routines, filename='brightness_normalize.sav'
```

Some quick notes:

- We stored all the code for our extension in one file, so we only have to call .COMPILE once. If we had multiple files, we'd have to compile each of them.
- CONTINUE_ON_ERROR and SKIP_ROUTINES='ENVI" keywords are set to RESOLVE_ALL to make sure we're not including any of the routines that are built into ENVI and therefore redundant in our code. Some will include the procedure ENVI and need to be skipped, others may cause compile errors we'll want to continue past.
- The ROUTINES keyword for SAVE sets it so that routines rather than variables are being saved to the SAV file. We can also store data in a SAV file as well if need be (covered in the File Access chapter).

Use your operating system's file explorer to navigate to the location where your SAV file was stored (should be in the **Current Directory** box in IDL). Locate the file and copy and paste it to your ENVI extensions directory. Now, close any projects in the workbench that contain BRIGHTNESS_NORMALIZE.PRO, just in case any conflicts arise. Relaunch ENVI with:

```
IDL> .reset
IDL> e = envi()
```

Custom Algorithms > Brightness Normalization should now be in the Extensions directory. Run it to give a final test. Brightness_normalize.sav can now be distributed to other users and they can use it in their own ENVI installation.

Exercises

1. In the embossing example, we added the results of the process to the display at the end of processing. Using this as a guideline (and referring to the Object API chapter as appropriate), add code to the Brightness Normalization extension to add results to the display.
2. Add another error check to the Brightness Normalization extension that will generate an error if the user calls the program from IDL when ENVI is not present or available.
3. Document/comment the Brightness Normalization .PRO code using IDLDoc documentation standards for IDL.
4. With changes made, re-deploy the extension.

References

Pesaresi, M., Gerhardinger, A. and Kayitakire, F. 2008. A Robust Built-Up Area Presence Index by Anisotropic Rotation-Invariant Textural Measure. IEEE Journal of Remote Sensing Selected Topics in Applied Earth Observations and REmote Sensing 1(3):180-192.

This article describes how the PanTex algorithm used in this chapter functions.

Wu, Changshan. 2004. Normalized spectral mixture analysis for monitorung urban composition using ETM+ imagery. Remote Sensing of Environment 93(4):480-492.

This article describes the brightness normalization algorithm used in Example 2.

Chapter 11 ENVI GUI Programming

In order to facilitate user interaction – parameter settings or specification of input and output files and directories for example – we have to allow the user to make these selections from the GUI. We have seen a few examples, such as ENVI_SELECT in the procedural library, or ENVI's UI.SelectData in the Object API, that generate graphic user elements and collect user input as part of higher level programming. If we want to build our own wizard-like workflow or collect numeric parameter settings, however, we must build our own graphic user interfaces through the use of *widgets*. Widgets are graphic user interface elements, and there are two ways to approach widget programming in ENVI:

- By using ENVI's streamlined and auto-managed widget library
- By using IDL's lower level widget routines

The first optional will allow you to construct GUI programs more quickly, but has less flexibility built into it. IDL's full widget library is more powerful and flexible than the ENVI convenience routines (themselves built on top of IDL's widget library), but have a steeper learning curve. Their use is covered in the *Application Development with IDL* manual.

ENVI Widgets

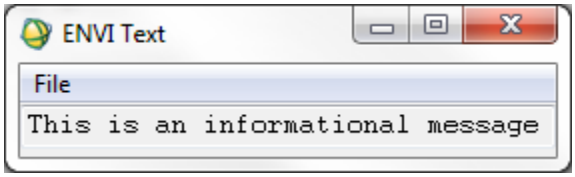
ENVI provides more than twenty compound widgets that can be incorporated into user functions. Most of these routines begin with ENVI_ or WIDGET_ and are listed in the Online Help. The most common ENVI widgets are illustrated here below:

An illustrated glossary of ENVI Widgets

Below follows a glossary of ENVI widgets as well as screenshots of their appearance:

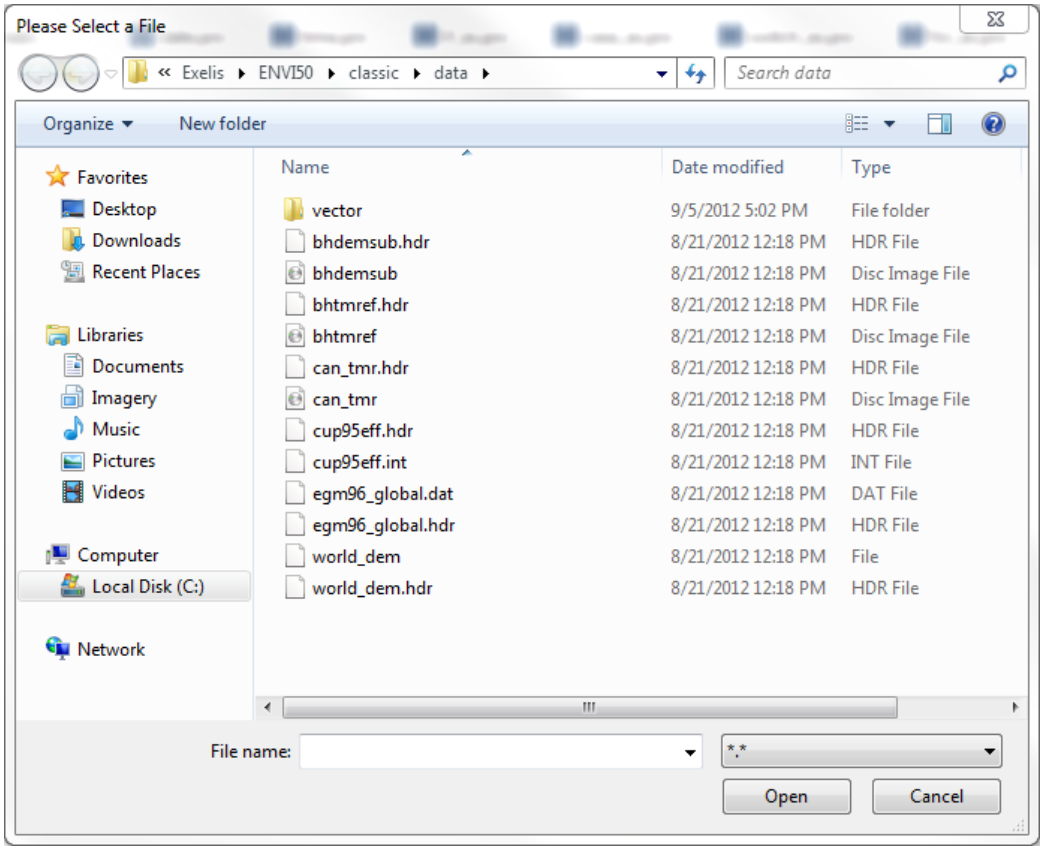
ENVI_INFO_WID

Used to present information in the form of a string to the user. Based on settings, the user can save the output to an ASCII file.

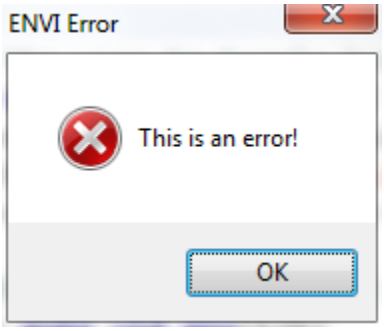


ENVI_PICKFILE

Used to select a file on the disk. It returns the fully qualified filepath to the file. ENVI_PICKFILE is an ENVI based wrapper for DIALOG_PICKFILE. It has a few less options, but by default will point the user to the input directory that matches their ENVI preference (DIALOG_PICKFILE will start in the current working IDL directory).



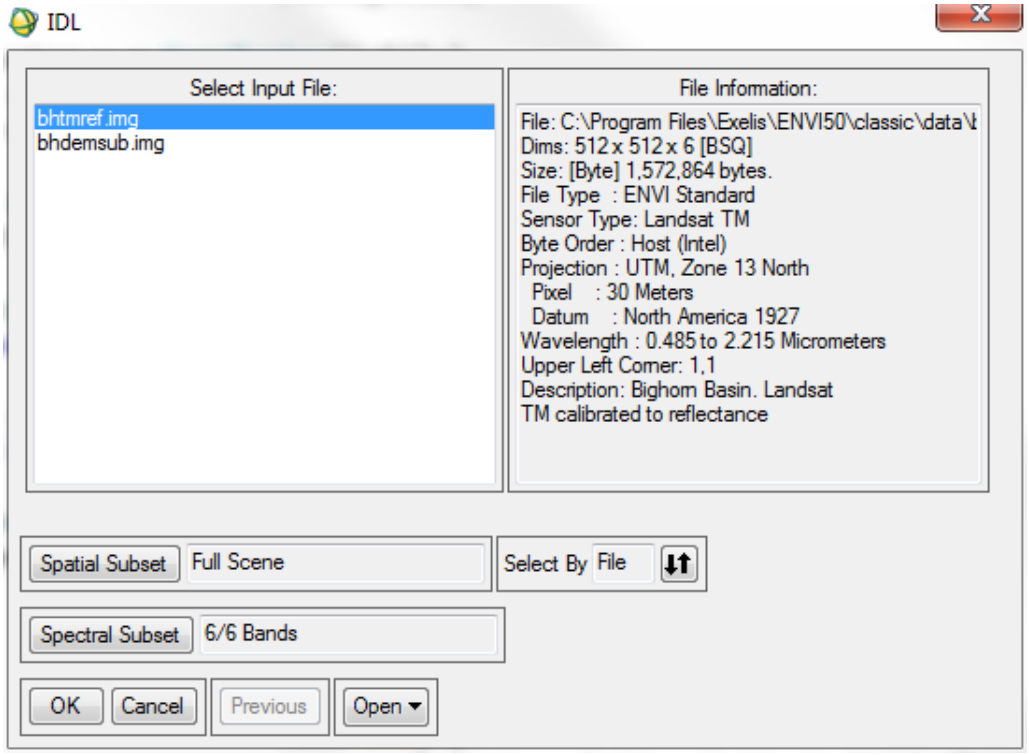
ENVI_REPORT_ERROR



This is ENVI's way of letting the user know an error has occurred. It also triggers an error state that must be managed with a CATCH block.

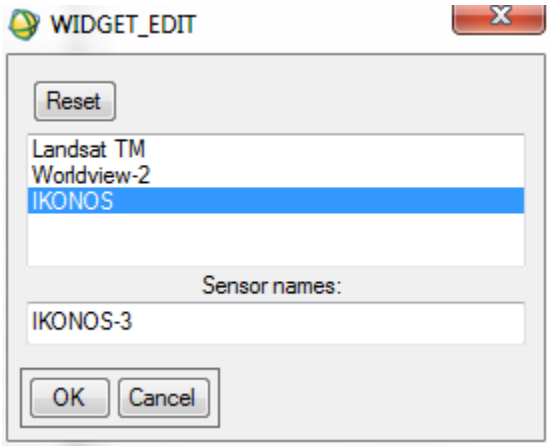
ENVI_SELECT

This is ENVI Classic's standard file selection dialog. It allows selection of an opened file, spatial and spectral subsetting, and selection of a mask band. This compound widget also includes a button that allows the user to open a new file from the disk.



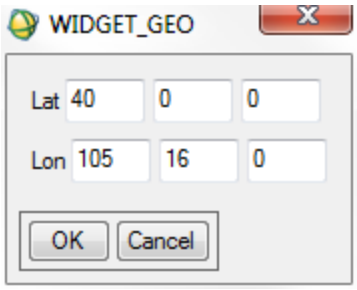
WIDGET_EDIT

Provides an interface for editing items in a list.



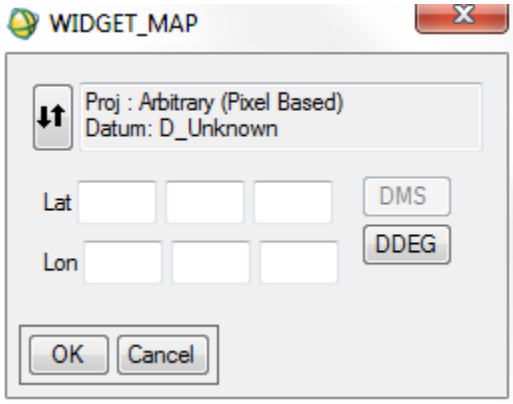
WIDGET_GEO

A compound widget used to enter latitude and longitude values in DMS or decimal degrees (the default).



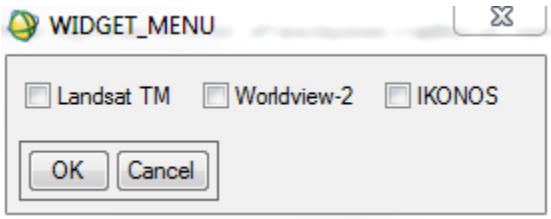
WIDGET_MAP

A compound widget for editing map coordinates and projections.



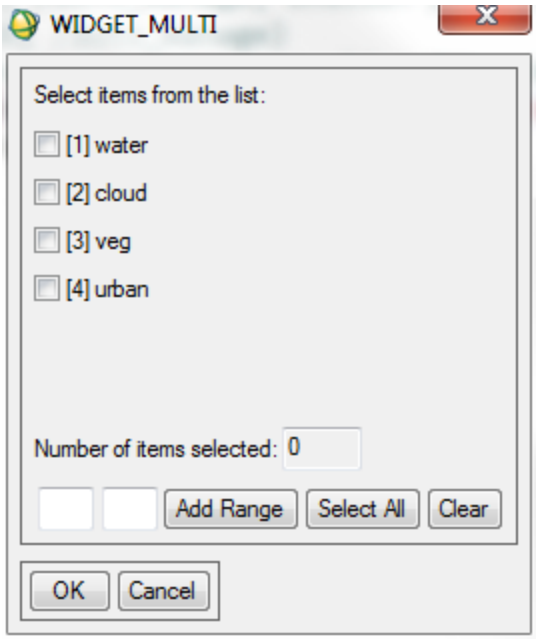
WIDGET_MENU

A compound widget that represents a list of exclusive (radio) or non-exclusive (check box) buttons.

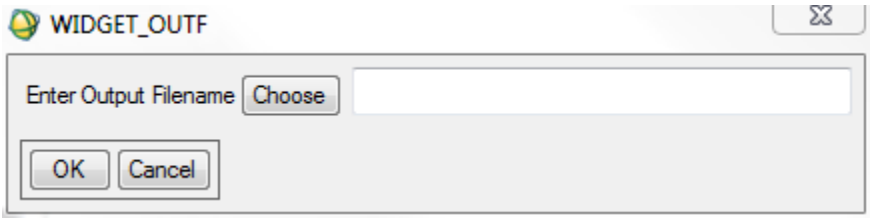


WIDGET_MULTI

A compound widget that allows selection of multiple items from a list. This widget is typically used to select ROIs for a classification.



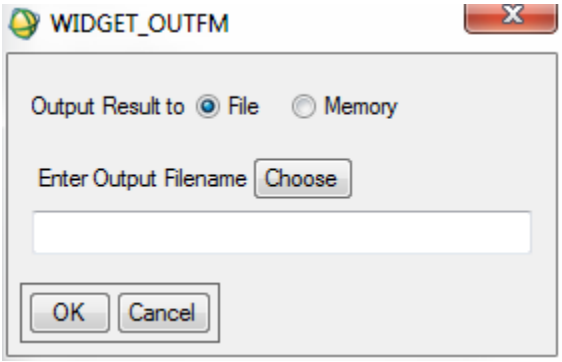
WIDGET_OUTF



A compound widget for entering or selecting an output filename.

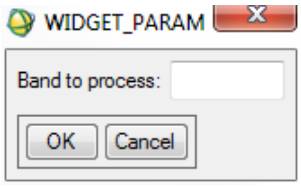
WIDGET_OUTFM

Like WIDGET_OUTF, this compound widget allows the user to enter/select an output filename, but also allows the option for a result to be stored in memory only.



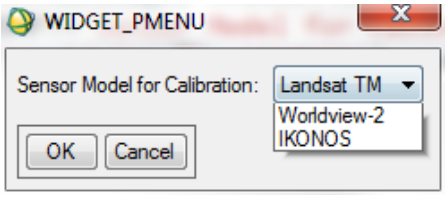
WIDGET_PARAM

A compound widget used for entering numbers.



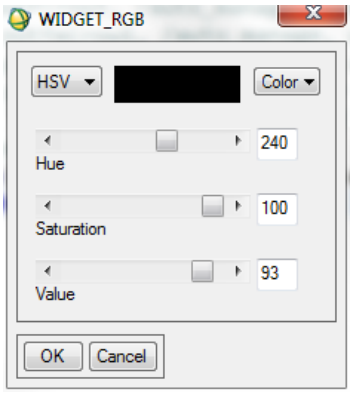
WIDGET_PMENU

A pulldown menu. This compound widget is an alternative to using WIDGET_MENU with exclusive buttons.



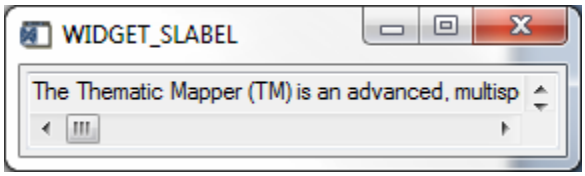
WIDGET_RGB

A compound widget for modifying an RGB color value with the option of using other color space models.



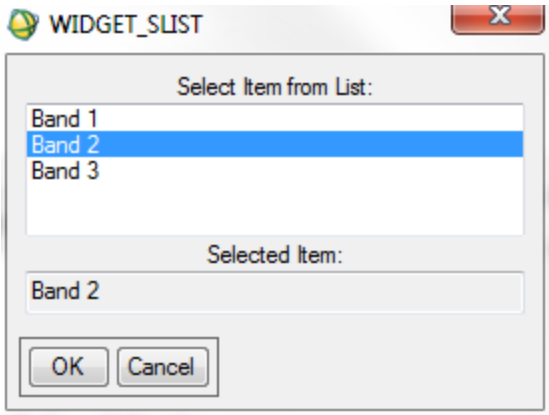
WIDGET_SLABEL

A compound widget used to display a text message with scroll bars.



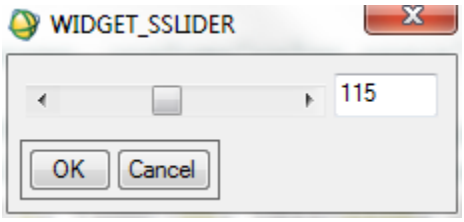
WIDGET_SLIST

A compound widget for creating and displaying lists. When an item is selected, it's listed in a separate text box.



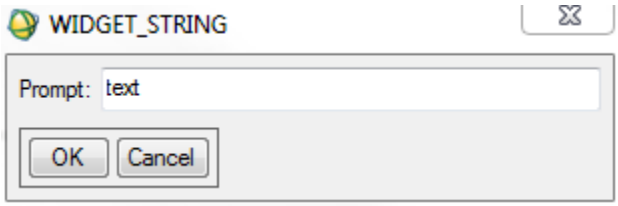
WIDGET_SSLIDER

A compound widget for setting a numeric value using a slider.



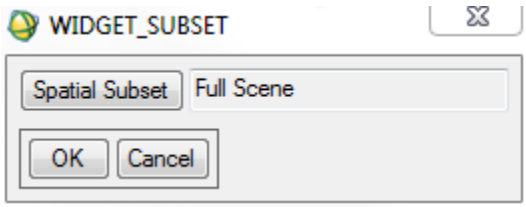
WIDGET_STRING

A compound widget used for entering text.



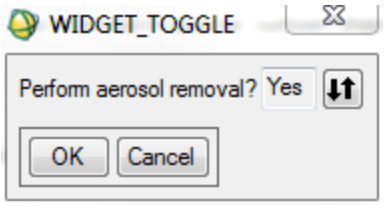
WIDGET_SUBSET

This is ENVI's standard widget for specifying spatial subsets. This widget has a button labeled **Spatial Subset** that brings up the standard spatial subset dialog.



WIDGET_TOGGLE

A compound widget for creating a toggle (on/off or yes/no) button with ENVI's arrow icon.



Auto-managing Widgets

In IDL, widget programs must have even handlers to process user interaction. Event handlers can be confusing to code, especially for someone new to IDL. To make the task of building ENVI GUIs easier, ENVI automatically manages events that originate from ENVI widgets using WIDGET_AUTO_BASE and AUTO_WID_MNG.

WIDGET_AUTO_BASE

WIDGET_AUTO_BASE creates a base widget for an ENVI extension. A base widget is a container in which other widgets are placed. For example, the ENVI application itself has a base widget whose reference we can gain access to by access to the WIDGET_ID propery of the ENVI application object. WIDGET_AUTO_BASE is distinguished by other base widgets in that it is auto-managed.

The following are properties of WIDGET_AUTO_BASE that should be considered:

- A base created with WIDGET_AUTO_BASE is column-based, centered, and modal (i.e. it blocks other user actions when being displayed/interacted with).
- WIDGET_AUTO_BASE accepts only four keywords: GROUP, TITLE, XBIG, YBIG – these keywords allow the GUI to be tied to an existing widget, to be given a title, and to be offset if unusually wide or tall. This differs from IDL's WIDGET_BASE which accepts dozens of keywords for controlling the base widget's appearance settings.
- ENVI widgets on an automanaged base must have their AUTO_MANAGE keyword set and have a user value set (with the UVALUE keyword) in order to be auto-managed.

AUTO_WID_MNG

In an IDL widget program, once the interface is defined, the XMANAGER procedure is called to register the program and begin listening for events (all of which are managed by event handling procedures which have been programmed by the developer for each specific event). In ENVI auto-managed user functions, by contrast, the AUTO_WID_MNG function is called instead. It does the following:

- AUTO_WID_MNG accepts one parameter, a widget base which has been properly created by WIDGET_AUTO_BASE.
- AUTO_WID_MNG draws the widget interface under the auto-managed base to the screen, including the **OK** and **Cancel** buttons at the bottom. It then waits for user input (since all WIDGET_AUTO_BASE widgets are blocking, the user will be blocked from doing anything else during this process).
- When the **OK** or **Cancel** button is pushed, AUTO_WID_MNG exits, returing a structure variable to the user function. This structure is referred to as the *result* structure.

- If **OK** was pressed, the *result* structure contains information that was entered into the widget interface. The name of each field in the *result* structure is defined by the user value (UVALUE) assigned to each ENVI compound widget when the interface was defined. The value of the structure fields are those values returned from the ENVI widgets.
- Pressing **Cancel** also returns a result structure, but it contains null or empty values.
- The *result* structure always contains a field called ACCEPT which is set to 1 if **OK** was selected, 0 if **Cancel** was selected.

ENVI user functions that employ auto-managed widgets therefore take a common form:

```
pro widget_template
  compile_opt idl2

  ; Set up containers
  tlb = widget_auto_base(title='window_tite') ; Defines window
  row1 = widget_base(tlb, /row) ; First container

  ; All items added follow
  _edit = widget_edit(row1, select_prompt='Sensor names: ', $
      list=['Landsat TM','Worldview-2','IKONOS'], uvalue='variable', $
      /auto_manage)
  _outfm = widget_outfm(row1, /auto_manage, uvalue='file')

  ; Auto-mangae widget and get results
  result = auto_wid_mng(tlb) ;Takes widget_auto_base as argument

  ; Do something with result structure
  if result.variable eq 0 then print, 'Something happens.'
end
```

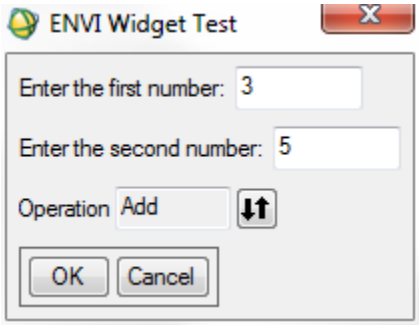
A Simple ENVI Widget Example

Here we'll create a simple ENVI widget program that will be called from the command line and prompt the user to input two numbers, then select **Add** or **Multiply**, then output the result:

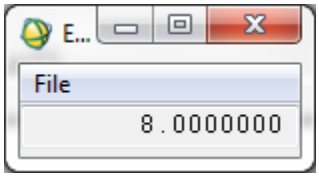
```
pro test_widgets, event
      compile_opt idl2

      ; Make an automanaged top-level base.
      tlb = widget_auto_base(title='ENVI Widget Test')

      ; Make two parameter widgets for user to input two numbers.
      row1 = widget_base(tlb, /row)
      p1 = widget_param(row1, /auto_manage, dt=4, field=2, $
```

```
            prompt='Enter the first number: ', uvalue='p1')
      row2 = widget_base(tlb, /row)
      p2 = widget_param(row2, /auto_manage, dt=4, field=2, $
            prompt='Enter the second number: ', uvalue='p2')

      ; Allow user to toggle between "Add" and "Multiply" operations.
      row3 = widget_base(tlb, /row)
      operation = widget_toggle(row3, /auto_manage, $
            default=0, list=['Add', 'Multiply'], $
            prompt='Operation', uvalue='operation')

      ; Call the automanaged widget manager.
      result = auto_wid_mng(tlb)
      if (result.accept eq 0) then return

      ; Display the result in a text box.
      if (result.operation eq 0) then $
            envi_info_wid, string(result.p1 + result.p2) $
      else $
            envi_info_wid, string(result.p1 * result.p2)
end
```

The above code in the course file TEST_WIDGETS.PRO creates the following GUI:



When the user enters two numbers, selects the operation, and then clicks **OK**, they will see:



Exercise

1. Modify the above program by adding **Divide** and **Subtract** to the list of operations.
2. (Advanced) Add a toggle to have the user select which operand to use as the 'first' and 'second' operands. Note that this will change the output results for **Divide** and **Subtract** only.

The Widget Interface for Pan-Tex

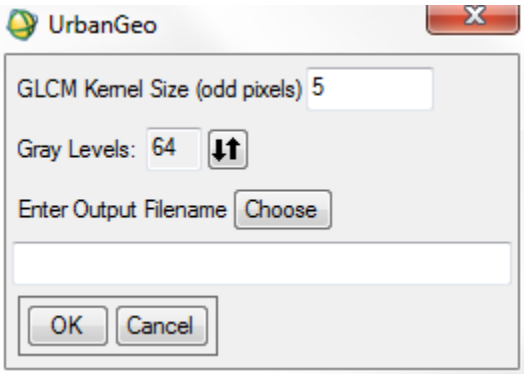
```
; Get ENVI session
e = envi(/current)

;Prompt for the input file
inFile=dialog_pickfile(dialog_parent=e.widget_id, $
                       title='Please select input file', $
                       /must_exist)

imageRaster = e.OpenRaster(inFile)
view = e.GetView()
layer = view.CreateLayer(imageRaster)

; Set up gray level list for selection.
gl_list = ['8', '16', '32', '64', '128', '256']

; Set up window as top level base widget.
tlb = widget_auto_base(title='UrbanGeo')

; Make a row for our moving window / kernel size.
row1 = widget_base(tlb, /row)

; Set moving window / kernel size.
k_param = widget_param(row1, $
                       /auto_manage, $
                       default=5B, $
                       dt=1, $
                       floor=3B, $
                       ceil=21B, $
                       prompt='GLCM Kernel Size (odd pixels)', $
                       uvalue='k')

; Make a row for the toggle widget.
row2 = widget_base(tlb, /row)

; Toggle Gray Levels for GLCM
gl_toggle = widget_toggle(row2, $
                          /auto_manage, $
                          default=3, $
                          list=gl_list, $
```

```
                              prompt='Gray Levels: ', $
                              uvalue='gl_select')

  ; Prompt user to select output file.
  out_f = widget_outf(tlb, $
                      /auto_manage, $
                       uvalue='out_file')

  ; Call the auto-managed widget manager.
  result = auto_wid_mng(tlb)
  if result.accept eq 0 then return
  if (result.k mod 2B) ne 1B then message, 'Kernel size must be odd.'
```

The above code is in the file URBAN_GEO_PANTEX.PRO and is an example of what an ENVI widget program looks like in practice. A basic description of how the widgets are used is given below:

- WIDGET_AUTO_BASE - the title setting 'UrbanGeo' provides the name for the window group.
- WIDGET_BASE(... /ROW) Orients the parameter widget following it by row.
- WIDGET_PARAM - default values are assigned, as is the data type and the floor and ceiling values. This means the user will have to enter a value between 3 and 21 and it will have to be of type byte.
- WIDGET_BASE(... /ROW_ orients the following toggle widget to be organized by row as well.
- WIDGET_TOGGLE gives us a toggle between all the values in the gray list levels array defined as GL_LIST. The PROMPT keyword defines the text that appears before the toggle, and the UVALUE 'GL_SELECT' will be the variable that contains the index of the user's current selection.
- WIDGET_OUTF will prompt the user to select the name and path of the output file, giving them both a text box to type into as well as a an option to browse and select a file.
- Finally, AUTO_WID_MNG is called to automatically manage the widgets.

The resulting interface looks as follows:



The user will use the WIDGET_PARAM interface to enter the kernel size, the WIDGET_TOGGLE labeled 'Gray Levels' to pick from the list of appropriate gray level reduction settings, and can use either **Choose**

or the empty text box created by the WIDGET_OUTF interface element to select an output file and its location.

Getting Values from Auto-managed Widgets

The next block in URBAN_GEO_PANTEX.PRO contains logic for handling user input through the auto-managed widgets:

```
; Call the auto-managed widget manager.
  result = auto_wid_mng(tlb)
  if result.accept eq 0 then return
  if (result.k mod 2B) ne 1B then message, 'Kernel size must be odd.'

  ; Run PanTex algorithm using user's settings.
  ug_pantex, ENVIRasterToFID(imageRaster), $   ;toFID()
             result.out_file, $
             kernel=result.k, $
             g_levels = gl_list[result.gl_select]
```

The call to AUTO_WID_MNG will block further IDL progress until the user makes a select. That select will be returned as a structure stored in the variable RESULT. The label RESULT will contain settings for each of the widgets that accept input as tags in the structure with names defined by the keyword argument UVALUE. It will also contain a field labeled ACCEPT (accessed as RESULT.ACCEPT) that contains 1 if the user clicked 'OK' and 0 if the user clicked 'Cancel.'

When we defined our earlier WIDGET_PARAM call, we set its UVALUE to 'K.' That is why we can check its setting with the structure dereference RESULT.K. The value stored in RESULT.K lets us see which option the user selected, and the code above contains error handling logic to enforce that the value is odd (because of requirements of the kernel size).

One other thing you will note about the logic in URBAN_GEO_PANTEX is that the user interface constructed builds a call to the procedure UG_PANTEX. This process of disentangling the code which handles the user interface and the code which handles the processing is recommended and enables a number of features:

- Easy maintainability: if a variable is handled differently inside UG_PANTEX, the code will only have to be modified there and the extension entry point will stay the same.
- Headless capability and automation: the entry point for the processing, UG_PANTEX, can be called from either the user interface using the widget logic, or a separate call to it can be built from headless mode, allowing it to be automated easily with no modification to the code.
- Encapsulation: the programmer building the interface or making calls to UG_PANTEX does not have to know how UG_PANTEX functions or note when it changes, except when those changes require that parameters be passed differently.

These are general principles of both structured modular programming and object design that are covered in more detail in the Application Development with IDL course.

IDL's Native Widgets

While the built in ENVI_ family of widgets are auto-managed, there is nothing preventing you from using IDL's built in standard widgets, and in fact, most advanced programmers build their applications by using IDL's standard GUI tools.

Widget Type	Definition Function	Description
ActiveX control	WIDGET_ACTIVEX	Embeds an ActiveX control in an IDL widget program (Windows only)
Base	WIDGET_BASE	A base widgets is a platform or container for other widgets, including bases. Works identically with auto-managed and standard widgets.
Button	WIDGET_BUTTON	Button widgets come in four varieties: push buttons, toggle buttons (nonexclusive), radio buttons (exclusive), and menu buttons.
Combobox	WIDGET_COMBOBOX	An editable droplist.
Draw	WIDGET_DRAW	A draw widget is used to display IDL graphics; it has built-in interactivity with the mouse ; it may have scroll bars.
Droplist	WIDGET_DROPLIST	Droplist widgets present a list of items that cascade from a button.
Label	WIDGET_LABEL	A label widget contains text that cannot be edited.
List	WIDGET_LIST	A list widget provides a list of items from which a user can select one item ,or several, using the mouse.
Property Sheet	WIDGET_PROPERTYSHEET	Property sheet widgets enable the user to view and edit the properties of an object subclassed from the IDLitComponent class.
Slider	WIDGET_SLIDER	Slider widgets have a marker that can be moved between two endpoint values. The value of the slider is determined by the marker's position.
Tab	WIDGET_TAB	Tab widgets create a "tabbed" interface that allows the user to select one of a set of bases displayed in a single space.
Table	WIDGET_TABLE	Table widgets store and display data in tabular form. Individual cells in the table can be made editable.
Text	WIDGET_TEXT	Text widgets are used to display lines of text, or to accept input from a user.
Tree	WIDGET_TREE	Tree widgets are used to display a hierarchical view of a data structure.
Window	WIDGET_WINDOW	Creates a Graphics window that can be embedded in a widgets interface.

Managing IDL's Native Widgets

IDL's native widgets are managed with two tools:

- WIDGET_CONTROL both draws and destroys widgets as well as retrieves values and sets values for various widget components.
- XMANAGER listens for events in either blocking or non-blocking mode and calls the appropriate event procedure when an event occurs.

For WIDGET_CONTROL and XMANAGER to function correctly, UVALUE and EVENT_PRO keyword settings must be defined for individual widgets. These are defined as follows:

- UVALUE is typically a pointer to a state variable which contains information we need to pass back and forth between programs.
- EVENT_PRO contains a string reference to the name of the procedure which handles the events for the individual widget. For example, a WIDGET_BUTTON EVENT_PRO would handle what to do in the event that a button was clicked.
- The EVENT structure, which is the first parameter for any EVENT_PRO, can be used to obtain information about the calling widget structure.
- From here, subsequent calls to WIDGET_CONTROL can be made to obtain program state information and manage events.

A full treatment of IDL's widget library is outside of this scope, but here we will present a limited application in which IDL's basic widgets are used to construct a progress bar for our EMBOSS_BY_TILE procedure. The changes are highlighted:

Adding a Progress Bar to EMBOSS_BY_TILE

```
pro emboss_by_tile_w_progress
  compile_opt idl2

  ; Start ENVI if it's not open
  e = envi(/current)
  if e eq !null then e = envi()

  ; Create an ENVIRaster
  bldr_pan_file = filepath('qb_boulder_pan', root_dir=e.root_dir, $
                  subdirectory = ['data'])
  bldr_pan_raster = e.OpenRaster(bldr_pan_file)


  ; Create an output raster
  new_file = e.GetTemporaryFilename()
  output_raster = e.CreateRaster(new_file, $
      inherits_from=bldr_pan_raster)

  ; Create an empty progress bar
  base = widget_base(title='Progress of emboss process')
  lbl = widget_label(base, value='Current progress: 0%', $
     /align_center, xsize=350, ysize=40)
```

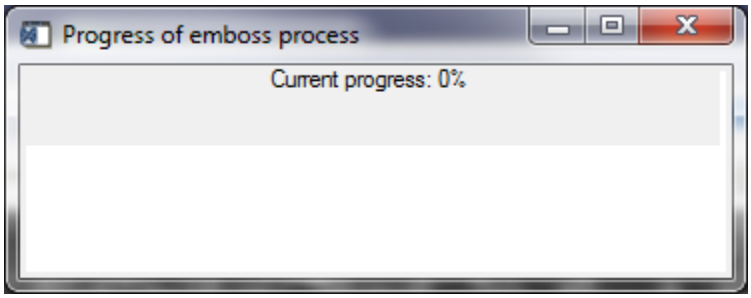
```
  base2 = widget_base(base, /row)
  draw = widget_window(base2, xsize=350, ysize=100)

  ; Draw progress bar widget.
  widget_control, base, /realize
  widget_control, draw, get_value=win
  win.select

  ; Iterate through the tiles of the original data set
  bldr_tiles = bldr_pan_raster.CreateTileIterator()
  increment = 1.0/bldr_tiles.ntiles
  progress=0
  p = polygon([0, progress, progress, 0],[0.4, 0.4, 0.6, 0.6], $
              fill_color='teal', /fill_background)

  foreach tile, bldr_tiles DO BEGIN
    data = emboss(tile, /EDGE_WRAP)
    output_raster.setTile, data, bldr_tiles

    ; Update progress bar and label after tile completes:
    progress = progress+increment
    p.setData, [0, progress, progress, 0],[0.4, 0.4, 0.6, 0.6]
    widget_control, lbl, set_value='Current progress: ' + $
            string(fix(progress * 100), format='(i3)') + '%'

  endforeach

  ; Save the data
  output_raster.Save

  ; Visualize original file and results.
  view = e.GetView()
  lyr2 = view.CreateLayer(output_raster)
  lyr1 = view.CreateLayer(bldr_pan_raster)
  port = view.CreatePortal(layer=lyr2)

  ; Close the progress bar.
  widget_control, base, /destroy
end
```

The following lines of code from these modifications:

```
  ; Create an empty progress bar
  base = widget_base(title='Progress of emboss process', /column)
```

```
  lbl = widget_label(base, value='Current progress: 0%', $
     /align_center, xsize=350, ysize=40)
  draw = widget_window(base2, xsize=350, ysize=100)
```

create a basic widget interface that looks as follows:



- The base widget is the window or container for other widgets. It is called with WIDGET_BASE
- The label is created with WIDGET_LABEL. The VALUE field is the label's text.
- WIDGET_WINDOW is a canvas that we can use to draw new graphics. As a note, WIDGET_DRAW is used more typically, and is a container which holds IDL object graphics items. Here we are using WIDGET_WINDOW for simplicity's sake.
- The keyword /COLUMN to WIDGET_BASE is what makes all elements added line up vertically.

Note that this application does not appear until it is drawn with:

```
widget_control, base, /realize
```

This application does not use XMANAGER to accept inputs from the user. It does, however, use WIDGET_CONTROL to periodically update values. In particular, it updates the extent of the POLYGON by iterating through a variable, PROGRESS, through a regular interval, INCREMENT, that is set based on the number of tiles ENVI pulls from the image:

```
  increment = 1.0/bldr_tiles.ntiles
  progress=0
  p = polygon([0, progress, progress, 0],[0.4, 0.4, 0.6, 0.6], $
              fill_color='teal', /fill_background)
```

After these initial states are set, they are updated for each tile processed with the following code:

```
    progress = progress+increment
    p.setData, [0, progress, progress, 0],[0.4, 0.4, 0.6, 0.6]
    widget_control, lbl, set_value='Current progress: ' + $
            string(fix(progress * 100), format='(i3)') + '%'
```

- The PROGRESS variable initially starts at 0 and updates by an amount equivalent to INCREMENT, or [1.0 / (the number of tiles)]. Therefore, if there are 4 tiles, it till start at 0 and update by 0.25 each time the process finishes.
- The call to the SETDATA method of Polygon changes the extent of the polygon in normalized coordinates to draw from 0, to 25%, and so on up to 100% through the space of the canvas's X dimension (thereby drawing a full progress bar).
- WIDGET_CONTROL is called to change the VALUE (that is the text) of the WIDGET_LABEL assigned to variable LBL to the text 'Current progress: ' plus a short math conversion process that translates 0.25 to the value 25%. FIX also truncates any decimals as we assume the user here will not be interested in the difference between 70% progress and 70.3% progress (this may not be the case, in which case a different numeric type and format code can be used).

For more on widget programming, refer to the Application Development with IDL course manual or code examples (`IDL_coursefiles\appdev`)

Exercises

1. Add code to include a progress bar using IDL style widgets for the routine BRIGHTNESS_NORMALIZE_BY_TILE from the Extensions chapter.

'Hacking' the ENVI Application

A few of the newer ENVI extensions work by 'hacking' the ENVI application - taking advantage of undocumented components of the ENVI GUI. Most of these function by accessing the top level widget ID, which is retrievable through the following code:

```
ENVI> e = envi(/current)
ENVI> tlb = e.widget_id
```

From there, if you are well versed in IDL widget programming you can begin to get information back using WIDGET_CONTROL:

```
ENVI> widget_control, tlb, get_uvalue=uvalue
ENVI> print, *uvalue
{<ObjHeapVar78721(IDLCFUICONTAINER)>        1228         870
      58.0000
      855.000      100.000
      23.0000        1202        1354}
```

Or you can add widgets or modify the widget interface layout:

```
ENVI> lbl = widget_label(tlb, value='Something')
```

```
ENVI> btn = widget_button(tlb, value='My Button',
event_pro='my_event')
```

-and so on. If your requirements stipulate that you modify ENVI's interface, these tools provide a route for doing so, but do be aware that you do so at your own risk and without official support from Exelis VIS.

Learning More

This is the end of the Extending ENVI manual. You should have a basic familiarity with IDL as well as the ENVI library routines available in both the procedural and object APIs. In addition, you should have a basic understanding of how ENVI extensions are constructed, as well as how to build your own.

To construct more sophisticated ENVI programs, consider these classes: **Application Development with IDL** is a logical next step for developers. **Scientific Programming with IDL** is a good fit for scientists performing algorithm development for subjects such as classification and spectral mixture analysis. For more on ENVI and its capabilities, Exelis VIS offers **Exploring ENVI** and **Spectral Analysis with ENVI**.

Chapter 12 Glossary

24-bit color	A system for displaying color that can employ the entire 2563 colors in the RGB color system. Most modern GPUs support 24-bit color.
8-bit color	A system for displaying color that can only use 256 unique colors simultaneously. Older GPUs only support 8-bit color.
Absorption Feature	A dip in the spectrum at a specific wavelength region where a material absorbs incoming radiation or light.
Abundance	Amount of a material present in a pixel.
ACE (Adaptive Coherence Estimator)	A tool used in ENVI that helps with target detection analysis. It will cross-reference the spectrum of a user-chosen pixel with the spectra in a user-chosen spectral library. Based on confidence intervals and probability intervals the user can determine if the pixel they have chosen can be matched to a material in a spectral library. Derived from the Generalized Likelihood Ratio (GLR) approach, ACE is invariant to relative scaling of input spectra and has a Constant False Alarm Rate (CFAR) with respect to such scaling.
Active Sensor	An active sensor generates and sends out electromagnetic energy and then records the signal that returns. Examples include radar and LiDAR sensors
actual parameter	The actual variable or expression that gets passed to a subprogram. Cf. **formal parameter**.
Aerosol	Particulate matter (liquid or solid) suspended in the atmosphere. Atmospheric aerosol size distribution ranges from nanometers up to hundreds of micrometers.
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.
Aspect	The compass direction that a topographic slope faces, usually measured in degrees from north.
Associated variable	A variable linked to an open file that maps a repeating array structure onto the file's contents.

Atmospheric Absorption	Atmospheric absorption is defined as a process in which solar radiation at a particular wavelength is absorbed by atmospheric gases.
Atmospheric Correction	A critical pre-processing step to help get data to reflectance. This includes the removal of atmospheric gas absorption features and atmospheric scattering of light. The conversion to reflectance also includes the removal of the solar irradiance curve. Hyperspectral images in particular provide enough spectral information to measure atmospheric absorption bands. Atmospheric properties are used to constrain highly accurate models of radiation transfer to produce an estimate of the true surface reflectance.
Atmospheric Scattering	An atmospheric process where small particles and gas molecules scatter part of the incoming solar radiation. The amount of scattering that takes place is dependent on the wavelength of the incoming radiation and the size of the scattering particle or gas molecule. In the Earth's atmosphere, the presence of a large number of particles with a size of about 0.5 Âµm results in shorter visible wavelengths being preferentially scattered. This factor causes our sky to look blue because this wavelength gets scattered the most.
Atmospheric Windows	Wavelength region where the atmospheric transmission of solar radiation is high.
Band	A particular wavelength range detected by a sensor channel.
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.
Blend Tool	An ENVI tool that transitions between images by gradually adjusting the transparency of the top image.
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 28 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 Composite Image	Displaying three bands of data as red, green, and blue to produce a color image.
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.
Composite Image	A composite image is made up of a set of overlapping images. For each pixel in the composite image, the best or most appropriate pixel is chosen from the available overlapping images.
Concatenation	The act of joining two variables into one. Used often with arrays and strings.
Conditional	A branching control statement.
Contrast	The difference in emitted or reflected energy by different objects in an image.
Contrast Enhancement	A technique used to improve the contrast between objects in an image.
Contrast Stretch	A process in which the range of brightness levels of an image is expanded to use the full brightness range of the display device. Equalization applies a histogram equalization stretch. Gaussian applies a Gaussian stretch. Square Root performs a square root gray scale transformation, then applies a linear stretch. Logarithmic logarithmically stretches the input image. It is a non-linear technique where the low-range brightness is enhanced. Optimized Linear applies an optimized linear stretch, also known as a dynamic range adjustment. By default ENVI uses an optimized linear stretcdh on 16-bit, unsigned integer data. Custom Stretch in ENVI allows you to adjust dark and light ranges using a histogram.
Control statement	A statement used to control the execution of other statements in a program.
Data Type	See **variable type**.
Decimal Degrees (DDEG)	Latitude and longitude geographic coordinates expressed as decimal fractions of degrees. It is an alternative to using degrees, minutes, and seconds (DMS).
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.

Degrees, Minutes, Seconds (DMS)	Latitude and longitude or Geographic Coordinates expressed as degrees, minutes, and seconds.
Device-copy technique	A method of copying data from the graphics card to an IDL Direct Graphics window.
Digital Surface Model (DSM)	Digital image that contains elevation data for the Earth's surface and also for objects such as buildings and trees.
Digital Terrain Model (DTM)	Digital image that contains elevation data for only the Earth's surface excluding buildings and trees.
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.
Earth Resource Satellite	Spaceborne sensor designed to map renewable and non-renewable resources. Examples are Landsat, Hyperion, JERS, ERS, IRS, ASTER, and MODIS.
Eclipse	An open-source software development environment. The IDL workbench is built on Eclipse.
Endianness	See **byte ordering**.
Enhance	The ENVI Enhance menu provides access to filters and contrast stretches that can be applied to the displayed images.
ENVI Configuration	The set up of various directories and other default preferences in ENVI. On start-up, ENVI searches for a file called envi.cfg. This file contains preferences on default directories, the configureation of ENVI display groups, which contrast stretch, etc. This file is editable.
ENVI Help	Documentation on ENVI's tools and algorithms. Printable documentation and tutorials are available on the ENVI web site (http://www.exelisvis.com). ENVI Help includes Contents, Index, Search, and Bookmarks tabs.
ENVI Main Menu Bar	This is the primary control panel for working in ENVI, allowing you to do such things as open files and apply processing functions.
Environmental Satellite	Spaceborne sensor designed to monitor meteorological phenomena, oceanographic systems, and land masses. Examples are AVHRR, GOES, CZCS
Error	A code execution or syntax problem that prevents further execution. Cf. **warning**.
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.
Feature	A feature is a unique characteristic of a spectral signature. It can be an absorption or emission spike for example
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.

Filter	Filters are used on images to remove certain spatial frequencies. Sharpening filters use a high-pass filter on an image. A certain amount of the original image is added back to the filtered product. Sharpening filters in ENVI with higher numbers in the brackets have a larger amount of the original data added back to the filtered image.
Flicker Tool	An ENVI tool that allows you to compare images by turning the transparency of the top image off and on.
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.
GCP	Ground Control Points. Geographic features of known locations that are recognizable on images and can be used to make geometric corrections. Control points are used in georeferencing both raster and vector data.
Global Services Group (GSG)	Exelis Visual Information Solutions team of consultants who provide custom software development, and consulting services to commercial, academic, and government markets.
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.
HSI	Hyperspectral Imaging or Imager
Hyperspectral	Hyperspectral sensors or imaging spectrometers measure reflected radiation in many narrow, contiguous spectral bands so that a complete spectral signature can be obtained from each pixel in the image.
IDE	Integrated Development Environment.
IDL	Interactive Data Language.
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.
IDL Workbench	The IDL Workbench interface is the control panel for the IDL session that is running ENVI, and is typically minimized at the bottom of the computer display This interface is used for advanced techniques involving customizing and extending ENVI.
IDLDE	IDL Development Environment. See **IDL workbench**.
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.)
Interactive Data Language (IDL)	A high level, structured programming language that offers integrated image processing. It can be used to extend ENVI beyond the usual installed tools.
Interactive Histogram	A tool used to manually adjust the contrast of an image by working with histograms.
Interpreter	The part of IDL that converts source code to byte code.
iTool	An interactive IDL application used to analyze and visualize data, then produce publication-quality output.
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.
Linear Mixing	In the case of a mixed pixel, linear mixing means that the spectral signature of that pixel is determined by how much area each material in the pixel covers.
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.
Metafile Format	A file format used by applications in the Windows operating system.
Method	A program built into an object. Used often with Graphics; e.g., the save method

	of a plot.
Mixed Pixel	A pixel containing more them one material.
Mosaic	A image created by combining two or more images that cover adjacent areas.
MSI	MultiSpectral Imaging or Imager
Multispectral	Multispectral sensors measure radiation over a few broad wavelength regions that may or may not be contiguous. This allows for the discrimination of materials but not identification.
Named Structure	A structure that has a name. Its definition is fixed within an IDL session.
Noise	Noise is random signal that is undesirable and not part of the signal from sensed materials. Noise can come from the electronics within a sensor.
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.
Orthorectification	The process of correcting the geometry of an image so that each pixel has accurate coordinates. Orthorectification uses elevation data to correct for terrain distortions.
Panchromatic	A panchromatic sensor measures radiation in only one wavelength band. However, the band is broad enough (covers a wide wavelength region) to allow for higher spatial resolution than for other sensor types.
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.
Passive Sensor	A passive sensor detects naturally occurring reflected or emitted energy.
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 rearranging views.
Pixel	Short for "picture element", the fundamental unit of an image.
Portal	An ENVI tool that allows you to compare overlapping images by exposing the lower image in a window. The portal can cover the whole image display or just a portion of it.
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.
Quick Atmospheric Correction (QUAC)	QUAC is an atmospheric correction method for multispectral and hyperspectral imagery that works with visible, near-infrared, and short wave infrared (VNIR-SWIR) imagery. It also removes the shape of the solar irradiance curve so that the result approximates reflectance. QUAC is an "in-scene" correction meaning no additional inputs such as atmospheric and solar conditions are required. QUAC works best if there are many different materials in the scene as well as dark areas.
Radiance	The measure of radiation power or light that passes through or comes from the surface of a material and falls within a solid (3D) angle in a specific direction. It varies with wavelength and is affected by the properties of the illumination source (for example the sun), the atmosphere, and the material itself.
Radiometric Calibration	Radiometric correction is the process of converting raw DN values to radiance. This is done by multiplying raw data by factors derived from calibration experiments.
Rare Target	A rare target is a target that occurs infrequently in a image.
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.
Reflectance	Reflectance is the percentage of incoming radiation or light that is reflected by a material. It varies with wavelength.
Region of Interest (ROI)	This is typically an area in an image that you define. It contains a material or materials you wish to analyze.
Regular Expression	A pattern that describes a string or a set of strings.
Remote Sensing	The science, technology, and art of obtaining information about objects or phenomena without being in physical contact with them. In a practical sense, this involves collecting reflected or emiited radiation from objects.
RGB Color System	A system for specifying colors based on combining, additively, intensities of red, green and blue light.
RGB Color-Composite Image	RGB color-composite images are made by displaying three different wavelength bands as red, green, and blue. The color of each pixel is dependent on the relative brightness of the materials in that pixel in each of the three bands.

RGB Triple	A three-element array that gives the red, green and blue components of a color.
ROI	See Region of Interest.
Routine	A synonym for **program**.
RPC	Remote Procedure Call (RPC) is a communication mechanism which allows one UNIX process to communicate with another UNIX process. These communicating processes can be on different computers over a network.
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.
Solar Irradiance	Solar irradiance is the energy emited from the sun. It varies with wavelength.
Solar Spectrum	The solar spectrum shows the distribution of the sun's energy as a function of wavelength. Because the sun is similar to a blackbody, the spectrum resembles a Planck function.
Source Code	Code written by a programmer. In IDL, like other languages, source code is plain text.
Spatial Domain	This represents an area within an image (in sample/line space),
Spatial Subset Button	ENVI's spatial subset button allows you to define spatial subsets in several different ways: • Sample and line ranges can be explicitly defined using the fields provided. • Clicking the Subset Using Image button allows you to drag and size a box on a thumbnail picture of the image to define a subset. • If the image is geo-referenced, clicking the Subset Using Map button allows the subset to be defined by entering map coordinates. • You can subset your image using the subset area of another sub-setted image. • You can subset your image based on the area encompassed by selected regions of interest (ROIs) or ENVI Vector Files (EVFs).
Spectra	Plural for spectrum.
Spectral Domain	This represents the wavelength dimension. The spectrum from one pixel shows the variation in signal with respect to wavelength.
Spectral Library	A spectral library is a collection of reflectance spectra for different materials. It is used to help detect and identify targets
Spectral Processing Exploitation and Analysis Resource (SPEAR) Tools	The Spectral Processing Exploitation and Analysis Resource (SPEAR) tools in ENVI provide a series of workflows that walk you through the steps required to process imagery. The ENVI SPEAR tools were designed specifically for some of the most common tasks military analysts need to perform. Basic instructions are included in each dialog. Detailed information on any step in the process can be

	accessed using the Help button at the bottom of each dialog.
Spectrum	A spectrum is plot of reflectance or some other measure of brightness that varies with wavelength.
Statement	A command or instruction in a programming language.
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.
Subset	A subset is part of an image and can be spatial, spectral, or both. A spatial subset is typically defined by a rectangular region in image pixels. A spectral subset is defined by a list of bands (channels) or wavelengths.
Swipe Tool	An ENVI tool that transitions between images by passing a vertical divider back and forth to alternate between image layers.
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	Vectors are representation of features you might see in a map or image. Features can be represented by points, polylines, or polygons.
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.
Wavelength Calibration	The process by which wavelengths are assigned to each band in an image. Wavelength information is typically contained in the image header file and is used in many spectral algorithms.
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.