IDL

Using IDL

IDL Version 6.4

April 2007 Edition

0407IDL64USG

Acknowledgments

ENVI® and IDL® are registered trademarks of ITT Corporation, registered in the United States Patent and Trademark Office. ION™, ION Script™, ION Java™, and ENVI Zoom™ are trademarks of ITT Visual Information Solutions.

Numerical Recipes™ is a trademark of Numerical Recipes Software. Numerical Recipes routines are used by permission.

GRG2™ is a trademark of Windward Technologies, Inc. The GRG2 software for nonlinear optimization is used by permission.

NCSA Hierarchical Data Format (HDF) Software Library and Utilities. Copyright © 1988–2001, The Board of Trustees of the University of Illinois. All rights reserved.

NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities. Copyright © 1998–2002, by the Board of Trustees of the University of Illinois. All rights reserved.

CDF Library. Copyright © 2002, National Space Science Data Center, NASA/Goddard Space Flight Center.

NetCDF Library. Copyright © 1993–1999, University Corporation for Atmospheric Research/Unidata.

HDF EOS Library. Copyright © 1996, Hughes and Applied Research Corporation.

SMACC. Copyright © 2000-2004, Spectral Sciences, Inc. and ITT Visual Information Solutions. All rights reserved.

This software is based in part on the work of the Independent JPEG Group.

Portions of this software are copyrighted by DataDirect Technologies, © 1991–2003.

BandMax®. Copyright © 2003, The Galileo Group Inc.

Portions of this computer program are copyright © 1995–1999, LizardTech, Inc. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

Portions of this software were developed using Unisearch's Kakadu software, for which ITT has a commercial license. Kakadu Software. Copyright © 2001. The University of New South Wales, UNSW, Sydney NSW 2052, Australia, and Unisearch Ltd, Australia.

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

MODTRAN is licensed from the United States of America under U.S. Patent No. 5,315,513 and U.S. Patent No. 5,884,226.

FLAASH is licensed from Spectral Sciences, Inc. under a U.S. Patent Pending.

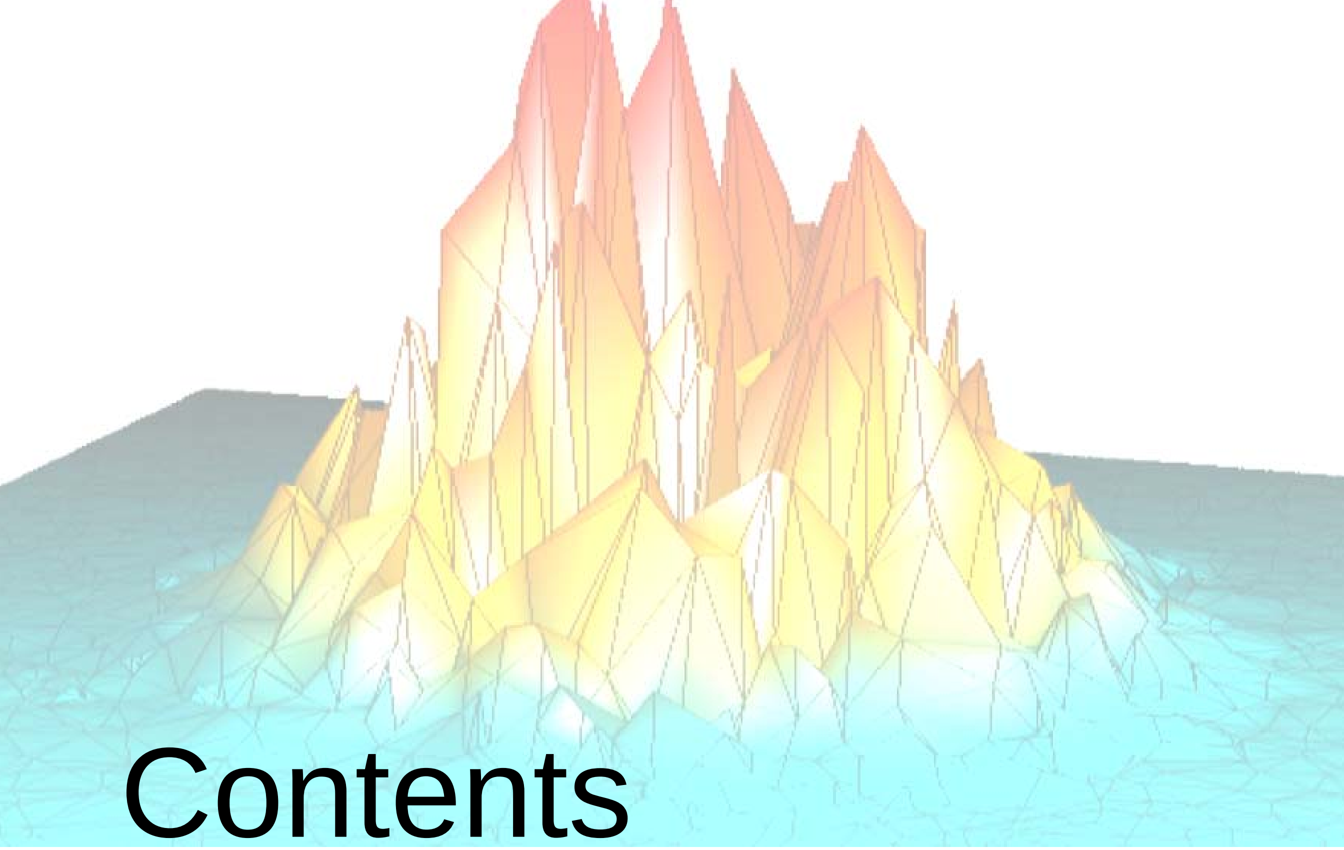
Portions of this software are copyrighted by Merge Technologies Incorporated.

Support Vector Machine (SVM) is based on the LIBSVM library written by Chih-Chung Chang and Chih-Jen Lin (http://www.csie.ntu.edu.tw/~cjlin/libsvm), adapted by ITT Visual Information Solutions for remote sensing image supervised classification purposes.

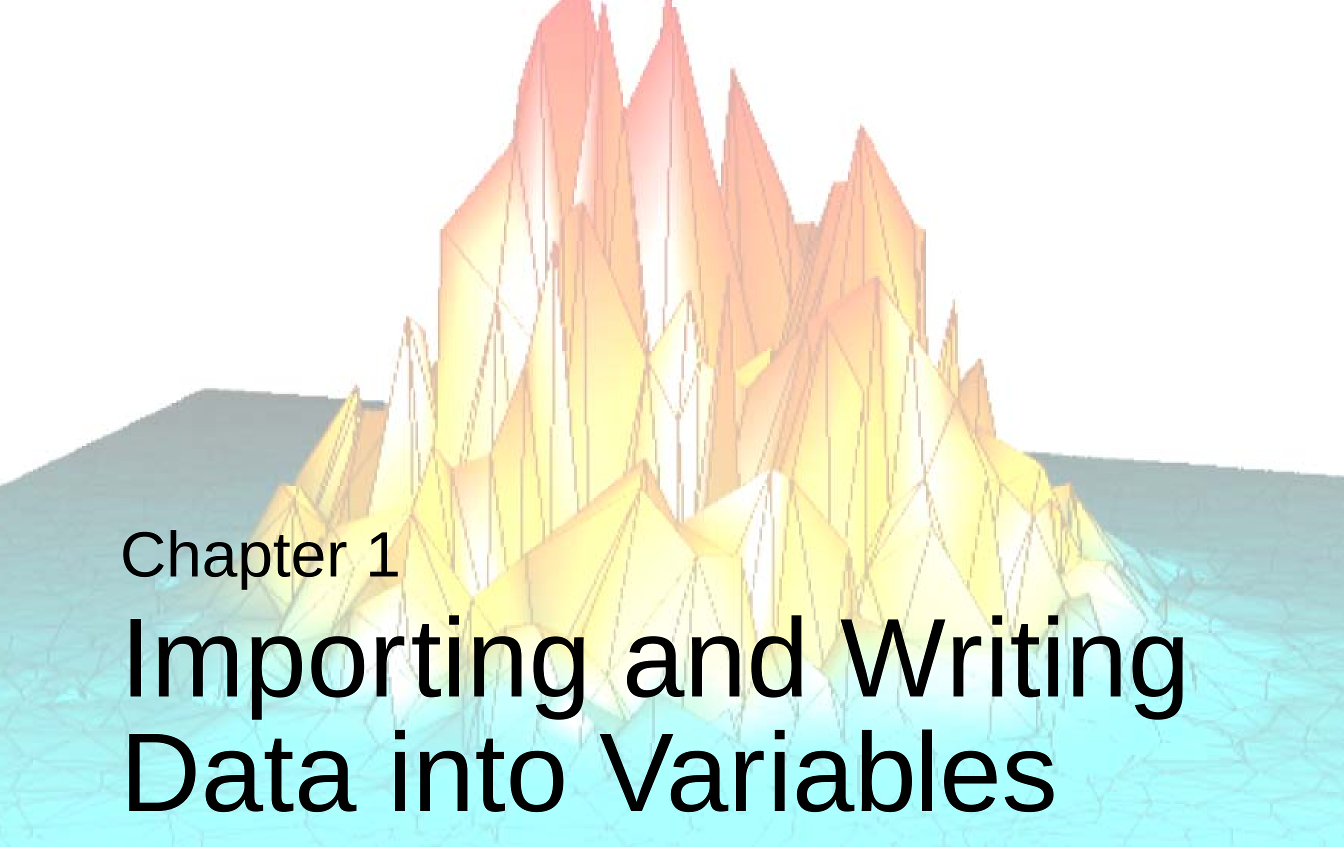
IDL Wavelet Toolkit Copyright © 2002, Christopher Torrence.

IMSL is a trademark of Visual Numerics, Inc. Copyright © 1970–2006 by Visual Numerics, Inc. All Rights Reserved.

Other trademarks and registered trademarks are the property of the respective trademark holders.



Contents



Chapter 1

Importing and Writing Data into Variables

This chapter provides an introduction to accessing, reading and writing data using the dialogs, and routines found in IDL.

Overview of Data Access in IDL

There are several ways to open files and access the data that they contain in IDL.You can open a file using interface elements, or using routines. In order of increasing complexity and flexibility, your options are:

- **Accessing data in iTools** — use **File** → **Open** from an iTool, and browse to select a file. This option automatically displays data (that is a supported type) in the iTool. See Chapter 2, "Importing and Exporting Data" (*iTool User's Guide*) for details.
- **Accessing files using dialogs** — launch an IDL dialog and browse to select or save a file. After accessing the file, use an IDL routine to access the data within the file. You can then preform additional data processing task or create a display. See "Accessing Files Using Dialogs" on page 9 for details.
- **Accessing files programmatically** — you can access data without requiring user interaction by using IDL statements in a program or at the command line. This give you the greatest control over the state of data at all times, but requires slightly more programming than the previous option. See "Accessing Files Programmatically" on page 14 for details.

There are advantages and disadvantages for each option. When you open a file using **File** → **Open** in the iTools, there is no opportunity to do pre-processing on the data. However, the display is created for you, and there are numerous interactive operations available.

You can combine the flexibility of accessing data using routines with the power of an iTool display by launching the iTool from the command line as described in "Parameter Data and the Command Line" (Chapter 2, *iTool User's Guide*). See "Accessing Image Data Programmatically" on page 16 and "Accessing Non-Image Data Programmatically" on page 20 for examples.

When you access data from the command line or in an IDL program, you have the greatest control over data modification. The iTools incorporate the functionality of many of the common data processing and manipulation routines. However, if you need greater control over data modification, want to create a custom display or object class, or need to use functionality that is not exposed through and iTool, you can import, export, and/or create your data programmatically.

Regardless of the method selected, it is important to note that only the options involving iTools will automatically display data for you. In other instances, you will need to configure a display yourself.

Accessing Files Using Dialogs

DIALOG_PICKFILE and DIALOG_READ_IMAGE are the two primary file access dialogs in IDL. Use DIALOG_PICKFILE to select any type of file. You can select multiple files, define the directory or define file filters using keywords. Use DIALOG_READ_IMAGE to access supported image formats (listed in "Image File Formats" (Chapter 2, *IDL Interface*)). This dialog offers preview capabilities and basic image information. The corollary DIALOG_WRITE_IMAGE allows you to write data to a select image file type.

See the following topics for more information:

- "Accessing Any File Type Using a Dialog" below
- "Importing an Image File Using a Dialog" on page 10
- "Saving an Image File Using a Dialog" on page 10

You can use other dialogs to access ASCII, binary and HDF data as described in:

- "Reading ASCII Data" on page 11
- "Reading Binary Data" on page 12

Also, several pre-defined IDL macros are provided to help you import data into the IDLDE. Each returns a structure, which you access programmatically in order to retrieve data. See "Using IDL Macros" on page 22 for details.

Note

Also see "CW_FILESEL" (*IDL Reference Guide*) for an example that configures a compound widget to open image files.

Accessing Any File Type Using a Dialog

The DIALOG_PICKFILE function lets you interactively pick a file using the platform's own native graphical file selection dialog. This function returns a string or an array of strings that contain the full path name of the selected file or files. The user can also enter the name of the file. The following statement opens the selection dialog and shows any `.pro` files in the current working directory. If you select a file and click **Open**, the `file` variable contains the full file path.

```
file = DIALOG_PICKFILE(/READ, FILTER = '*.pro')
```

Other keywords allow you to specify the initial directory, the dialog title, the filter list, and whether multiple file selection is permitted. See "DIALOG_PICKFILE" (*IDL Reference Guide*) for details.

After you select a file using DIALOG_PICKFILE, you can then use one of the file I/O routines to access the data within the file. See "Accessing Image Data Programmatically" on page 16 or "Accessing Non-Image Data Programmatically" on page 20 for more information.

Importing an Image File Using a Dialog

The DIALOG_READ_IMAGE function opens a graphical user interface which lets you read image files. This interface simplifies the use of IDL image file I/O. You can preview images with a quick and simple browsing mechanism which also reports important information about the image file. You can also control the preview mode.

The following statement opens the dialog so that you can select among `.gif`, `tiff`, `.dcm`, `.png` and `.jpg` files.

```
result = DIALOG_READ_IMAGE(FILE=selectedFile, IMAGE=image)
```

See "Using the Select Image File Dialog Interface" under "DIALOG_READ_IMAGE" (*IDL Reference Guide*) for additional information if desired. When you select a file and click **Open**, the file path is stored in `selectedFile` variable and the image data is stored in the `image` variable. Enter the following line to display image data in an iImage display.

```
IF result EQ 1 THEN iImage, image
```

Saving an Image File Using a Dialog

The DIALOG_WRITE_IMAGE function displays a graphical user interface that lets you write and save image files. This interface simplifies the use of IDL image file I/O. The following statements create and write a simple image to a `.tif` file name `myimage.tif`:

```
myimage = DIST(100)
result = DIALOG_WRITE_IMAGE(myimage, FILENAME='myimage.tif')
```

When you select **Save**, it creates a `.tif` file in your current working directory or the directory of your choice. See "DIALOG_WRITE_IMAGE" (*IDL Reference Guide*) for a complete list of keywords and a description of the dialog interface.

Reading ASCII Data

IDL recognizes two types of ASCII data files: free format files, and explicit format files. A free format file uses commas or tabs and spaces to distinguish each element in the file. An explicit format file distinguishes elements according to the commands specified in a format statement. Most ASCII files are free format files.

Note

If you prefer not to use an interactive dialog (described below), you can also use the READ/READF, or READS procedures to access ASCII data. The READ procedure reads free format data from standard input, READF reads free format data from a file, and READS reads free format data from a string variable.

Launching the ASCII Template Dialog

The ASCII_TEMPLATE function launches a dialog that you can use to configure the structure of data in an ASCII file. Access this feature in one of the following ways:

- From an iTool — select **File** → **Open** (or click the **Import File** button in the Data Manager or Insert Visualization dialog) and select a text file
- From the IDLDE — select **Macros** → **Import ASCII** and select a text file
- From the IDL command line — use the following syntax to call ASCII_TEMPLATE and select a text file:

  ```
  sTemplate = ASCII_TEMPLATE()
  ```

  Note

  If you specify a *Filename* argument to ASCII_TEMPLATE, the dialog allowing you to browse to select a file will not appear. See "ASCII_TEMPLATE" (*IDL Reference Guide*) if you want specify a file and other parameters programmatically.

See "Using the ASCII Template Dialog" under "ASCII_TEMPLATE" (*IDL Reference Guide*) for instructions on how to use the dialog to define the structure of your ASCII data.

Reading Binary Data

Data is sometimes stored in files as arrays of bytes instead of a known format like JPEG or TIFF. These files are referred to as binary files. Binary data or binary data files are more compact than ASCII data files and are frequently used for large data files. Binary data files are stored as one long stream of bytes in a file. You will need to define the structure of the fields in the file in order to correctly read in the binary data.

The BINARY_TEMPLATE and READ_BINARY functions are designed to define and access binary data. The READ_BINARY function, which reads binary data, is either invoked internally (when you open a binary file from the iTools or use the **Import Binary** macro), or is explicitly called from the command line. This function is intended to read raw binary data that requires no special processing (except possibly byte-order swapping). This function is not designed to read commercial spreadsheet or word processing files.

Note

If you prefer not to use an interactive **Binary Template** dialog (described below) to define the structure of the data in the binary file, you can use the READU procedure. To read binary data files, define the variables, open the file for reading, and read the bytes into those variables. Each variable reads as many bytes out of the file as required by the specified data type and organizational structure.

If you need to open a single binary file, it may be easier to use READ_BINARY to directly define data characteristics using keywords instead of creating a template using the **Binary Template** dialog (described below). See “READ_BINARY” (*IDL Reference Guide*) for an example.

Launching the Binary Template Dialog

The BINARY_TEMPLATE function launches a dialog that you can use to define the structure of data in an binary file. Access this feature in one of the following ways:

- From an iTool — select **File** → **Open** (or click the **Import File** button in the Data Manager or Insert Visualization dialog) and select a binary file
- From the IDLDE — select **Macros** → **Import Binary** and select a binary file
- From the IDL command line — use the following syntax to call BINARY_TEMPLATE and select a text file:

```
sTemplate = BINARY_TEMPLATE()
```

Note

If you specify a *Filename* argument to BINARY_TEMPLATE, the dialog allowing you to browse to select a file will not appear. See "BINARY_TEMPLATE" (*IDL Reference Guide*) if you want specify a file and other parameters programmatically.

See "Using the BINARY_TEMPLATE Interface" under "BINARY_TEMPLATE" (*IDL Reference Guide*) for instructions on how to use the dialog to define the structure of your binary file.

Accessing Files Programmatically

Regardless of the data type, there are several routines that are commonly used to access files and data. To read data into an IDL variable, you must identify the file containing the data, and then extract the data from the file. This section discusses file access. Following sections (discuss data access.

File Access

One of the most common file access routines is FILEPATH. Use this to select a named file in a specified directory. For example, to select a file in the `examples/data` directory of the existing working directory, use the statement:

```
file = FILEPATH('mr_brain.dcm', SUBDIRECTORY=['examples', 'data'])
```

To access a file outside the existing working directory, use the ROOT_DIR keyword. The following statement opens a file named `testImg.tif` in the `C:\tempImages` directory.

```
file = FILEPATH('testImg.tif', ROOT_DIR='C:', $
   SUBDIRECTORY='tempImages')
```

Cross-platform File Access

If your application requires a cross-platform path, one that is not specific to UNIX or Windows, consider using the DIALOG_PICKFILE routine with the GET_PATH keyword. This lets you choose a file and store the operating system native path to the file in a variable. In the following example, you choose an image file and the full directory path to the selected image is stored in `path`:

```
sFile = DIALOG_PICKFILE(/MUST_EXIST, $
   TITLE = 'Select an Image File', $
   FILTER = ['*.bmp', '*.jpg', '*.png', '*.ppm', '*.tif'], $
   GET_PATH=path)
```

When you need to access a file in the directory stored in `path`, you can use the PATH_SEP function to return the correct path separation character for the operating system. Suppose you have a file called `myTestFile.jpg` that you want to delete before a program ends. FILE_DELETE requires a string *File* argument that is in the native syntax for the current operating system. To delete this file, you can use the directory information stored in `path`, plus the PATH_SEP function, plus the name of the file to delete as follows (the + operator concatenates strings):

```
FILE_DELETE, path+PATH_SEP()+'myTestFile.jpg', /ALLOW_NONEXISTENT
```

IDL also provides an extensive number of other file manipulation routines. See "General File Access" under the functional category "Input/Output" (*IDL Quick Reference*) for a list.

FILEPATH is often used in conjunction with routines that access the data from a file, as shown in the following section.

Accessing Image Data Programmatically

You can access image data using routines designed for general image file access, designed specifically for an image file format, or using unformatted data access routines. Which option you choose depends on the file type and the level of control you want over reading and writing the file. See the following topics for details:

- "Importing Formatted Image Data Programmatically" below
- "Importing Unformatted Image Files" on page 17
- "Exporting Formatted Image Files Programmatically" on page 18
- "Exporting Unformatted Image Files" on page 19

Note

These sections describe how to load data into a variable and includes examples of passing variable data to an iTool programmatically. See "Importing Data from the IDL Session" (Chapter 2, *iTool User's Guide*) if you want information on how you can access variable data from the iTools Data Manager.

Importing Formatted Image Data Programmatically

The majority of IDL image data access routine require a file specification, indicating the file from which to access the data. The FILEPATH routine is often used within a data access routine as shown in the following example.

Note

To validate that an image file can be accessed using READ_* routines, you can query the image first. See "Returning Image File Information" on page 33 for details.

The following example opens a JPEG file from the `examples/data` directory, performs feature extraction, and displays both images using IIMAGE.

```
; Open a file and access the data.
file = FILEPATH('n_vasinfecta.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, image, /GRAYSCALE

; Mask out pixel values greater than 120
; and create a distance map.
binaryImg = image LT 120
distanceImg = MORPH_DISTANCE(binaryImg, NEIGHBOR_SAMPLING = 1)
```

```
; Launch iImage, creating a 2 column, 1 row layout.
; Display the original and distanceImg in the two views.
IIMAGE, image, VIEW_GRID=[2,1]
IIMAGE, distanceImg, /VIEW_NEXT, /OVERPLOT
```

In the previous example, you could use the READ_IMAGE function instead of the READ_JPEG function by replacing the following statement:

```
READ_JPEG, file, image, /GRAYSCALE
```

with

```
image = READ_IMAGE(file)
```

In this instance, you do not have control over the color table associated with the image. It is often more useful to use a specific READ_* routine or object designed for the image file format to precisely control characteristics of the imported image.

For a list of available image access, import and export routines and objects, see “Image Data Formats” under the functional category “Input/Output” (*IDL Quick Reference*).

Note

IDL can also import images stored in scientific data formats, such as HDF and netCDF. For more information on these formats, see the *Scientific Data Formats* manual.

Importing Unformatted Image Files

Images in unformatted binary files can be imported with the READ_BINARY function using the DATA_DIMS and DATA_TYPE keywords as follows:

- You must specify the size of the image within the file using the DATA_DIMS keyword. This is required because the READ_BINARY function assumes the data values are arranged in a single vector (a one-dimensional array). The DATA_DIMS keyword is used to specify the size of the two- or three-dimensional image array.
- You can set the DATA_TYPE keyword to the image’s data type using the associated IDL type code (see “IDL Type Codes and Names” under the SIZE function in the *IDL Reference Guide* for a complete list of type code). Most images in binary files are of the byte data type, which is the default setting for the DATA_TYPE keyword.

No standard exists by which image parameters are provided in an unformatted binary file. Often, these parameters are not provided at all. In this case, you should already be familiar with the size and type parameters of any images you need to access within binary files.

For example, the `worldelv.dat` file is a binary file that contains an image. You can only import this image by supplying the information that the data values of the image are byte and that the image has dimensions of 360 pixels by 360 pixels. Before using the READ_BINARY function to access this image, you must first determine the path to the file:

```
file = FILEPATH('worldelv.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

Define the size parameters of the image with a vector:

```
imageSize = [360, 360]
```

An image type parameter is not required because we know that the data values of image are byte, which is the default type for the READ_BINARY function.

The READ_BINARY function can now be used to import the image contained in the `worldelv.dat` file:

```
image = READ_BINARY(file, DATA_DIMS = imageSize)
IIMAGE, image
```

Exporting Formatted Image Files Programmatically

Images can be exported to common image file formats using the WRITE_IMAGE procedure. The WRITE_IMAGE procedure requires three inputs: the exported file's name, the image file type, and the image itself. You can also provide the red, green, and blue color components to an associated color table if these components exist.

For example, you can import the image from the `worldelv.dat` binary file:

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

You can export this image to an image file (a JPEG file) with the WRITE_IMAGE procedure:

```
WRITE_IMAGE, 'worldelv.dat', 'JPEG', image
```

IDL also provides format-specific WRITE_* routines that are similar to the WRITE_IMAGE procedure, but provide more flexibility when exporting a specific image file type. See "Image Data Formats" under the functional category "Input/Output" (*IDL Quick Reference*) for a list of available image access, import and export routines and objects.

Note

IDL can also export images stored in scientific data formats, such as HDF and netCDF. For more information on these formats, see the *Scientific Data Formats* manual.

Exporting Unformatted Image Files

Images can be exported to an unformatted binary file with the WRITEU procedure. Before using the WRITEU procedure, you must open a file to which the data will be written using the OPENW procedure. Any file you open must be specifically closed using either the FREE_LUN or CLOSE procedure when you are done exporting the image.

For example, you can import the image from the `rose.jpg` image file:

```
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_IMAGE(file)
```

You can export this image to a binary file by first opening a new file:

```
OPENW, unit, 'rose.dat', /GET_LUN
```

Then, use the WRITEU procedure to write the image to the open file:

```
WRITEU, unit, image
```

You must remember to close the file once the data has been written to it:

```
FREE_LUN, unit
```

Note

For complete details about reading, writing and formatting unformatted data, see Chapter 18, “Files and Input/Output” (*Application Programming*).

Accessing Non-Image Data Programmatically

There are a number of options available for reading non-image data into IDL. Depending upon the file type, consider using one of the following:

- **Formatted data** — use a data-type-specific routine (such as READ_ASCII or READ_BINARY). See "Reading Binary Data in a Volume" below for more information.
- **Unformatted data** — use a general data access routines (such as OPEN or WRITE). For complete details about reading, writing and formatting unformatted data, see Chapter 18, "Files and Input/Output" (*Application Programming*).
- **SAVE file data** — use the RESTORE procedure to access variable data in a SAVE file. See "Reading Contour Data from a SAVE File" on page 21 for an example.

Note

These sections describe how to load data into a variable and includes examples of passing variable data to an iTool programmatically. See "Importing Data from the IDL Session" (Chapter 2, *iTool User's Guide*) if you want information on how you can access variable data from the iTools Data Manager.

Reading Binary Data in a Volume

The following example uses READ_BINARY to access binary data (`head.dat`) consisting of a stack of 57 images slices of the human head. After reading the data, create a display using IVOLUME. Enter the following at the IDL command prompt:

```
file = FILEPATH('head.dat', $
   SUBDIRECTORY = ['examples', 'data'])
dataSize = [80,100,57]
volume= READ_BINARY(file, DATA_DIMS = dataSize)
iVolume, volume, /AUTO_RENDER
```

Note

You can also create a template for binary file access. See "Reading Binary Data" on page 12 for options.

Reading Contour Data from a SAVE File

You can also access information from a SAVE file. This example restores a SAVE file containing variable data (`marbells.dat`), configures the data, and displays it using ICONTOUR.

```
PRO maroonBellsContour_doc

; Restore Maroon Bells data into the IDL variable "elev".
RESTORE, FILEPATH('marbells.dat', SUBDIR=['examples','data'])

; Create x and y vectors giving the position of each
; column and row.
X = 326.850 + .030 * FINDGEN(72)
Y = 4318.500 + .030 * FINDGEN(92)

; Set missing data points to a large value. Reduce to a
; 72 x 92 matrix.
elev (WHERE (elev EQ 0)) = 1E6
new = REBIN(elev, 360/5, 460/5)

iContour, new, X, Y, C_VALUE = 2750 + FINDGEN(6) * 250.,$
 XSTYLE = 1, YSTYLE = 1, YMARGIN = 5, MAX_VALUE = 5000, $
   C_LINESTYLE = [1, 0], $
   C_THICK = [1, 1, 1, 1, 1, 3], $
   XTITLE = 'UTM Coordinates (KM)'

End
```

Note

See Chapter 4, “Creating SAVE Files of Programs and Data” (*Application Programming*) for complete details on creating and restoring SAVE files.

Using IDL Macros

When you are working in the IDLDE, you can use a pre-defined macro to help you import image, ASCII, binary or HDF data. These macros call internal functions and return structures containing data. From the IDL command line, you can access and display data elements contained in the structures. These macros are available through the **Macros** menu and also through IDL toolbar buttons.

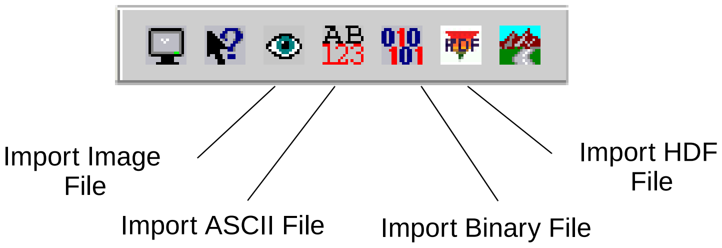


Figure 1-1: Macro Toolbar Buttons

See the follow sections for more information:

- "Using Macros to Import Image Files" on page 23
- "Using Macros to Import ASCII Files" on page 25
- "Using Macros to Import Binary Files" on page 27
- "Using Macros to Import HDF Files" on page 28

Using Macros to Import Image Files

To import an image file into IDL using a macro, complete the following steps:

1. Select the **Import Image** toolbar button. The **Select Image File** dialog is displayed.
2. Select a file to import. For example, select the `*IDL_DIR*/examples/data/muscle.jpg` file where *`IDL_DIR`* is the installation directory for IDL. See "Using the Select Image File Dialog Interface" under "DIALOG_READ_IMAGE" (*IDL Reference Guide*) for additional information if desired.
3. Click **Open**.

The `muscle.jpg` image data has been opened into a structure variable named MUSCLE_IMAGE. The **Import Image** macro opens and stores image data in a structure variable named *filename*_IMAGE where *filename* is the name of the file you opened without the extension.

Note

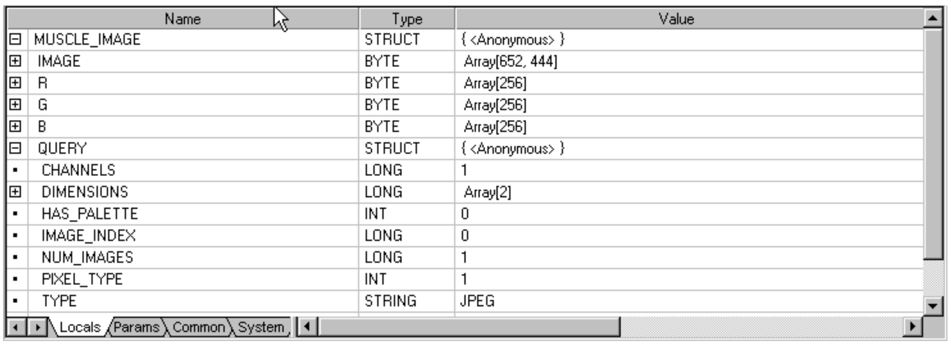
IDL variables must begin with a letter, and may contain only letters, digits, the underscore character, or the dollar sign. If the first character of *filename* is not a letter, the prefix "var" is added to the variable name. Any spaces within *filename* are converted to underscores. Any other illegal characters within *filename* are removed.

The MUSCLE_IMAGE structure contains the following fields:

- IMAGE — The actual image array.
- R — The red color table vectors.
- G — The green color table vectors.
- B — The blue color table vectors.
- QUERY — Contains information about the image.
  - CHANNELS — The number of channels in the image.
  - HAS_PALETTE — Specifies if the palette is present. 1 if the palette is present, else 0. If your image is *n*-by-*m* the palette is usually present and the R, G, and B color table vectors mentioned above will contain values. If your image is 3-by-*n*-by-*m*, the palette will not be present and the R,G, and B color table vectors will not contain any values.

- IMAGE_INDEX — The index of the image of the file. The default is 0, the first image in the file. If there are multiple images in the file that you read, this will be the number (or index) of the image.
- NUM_IMAGES — The number of images in the original file.
- PIXEL_TYPE — The IDL Type Code of the image pixel format. Valid types are described in "IDL Type Codes and Names" under "SIZE" (*IDL Reference Guide*).
- TYPE — The image format type.

The structure can be viewed in the Variable Watch Window.



Name	Type	Value
⊟ MUSCLE_IMAGE	STRUCT	{ <Anonymous> }
⊞ IMAGE	BYTE	Array[652, 444]
⊞ R	BYTE	Array[256]
⊞ G	BYTE	Array[256]
⊞ B	BYTE	Array[256]
⊟ QUERY	STRUCT	{ <Anonymous> }
▪ 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	JPEG

Figure 1-2: Variable Watch Window Showing MUSCLE_IMAGE Structure

You can specify which part of the structure variable you want to access by using the following syntax:

variable_name.element_name[*.element_name*]

For example, if you want to view the image, enter the following:

```
IIMAGE, MUSCLE_IMAGE.IMAGE
```

If you want to know the file type, enter the following:

```
PRINT, MUSCLE_IMAGE.QUERY.TYPE
```

IDL prints:

```
JPEG
```

Using Macros to Import ASCII Files

To import an ASCII file into IDL using a macro, complete the following steps:

1. Select the **Import ASCII** toolbar button. The **Select an ASCII file to read** dialog appears.
2. Select a file to import.
3. See "Using the ASCII Template Dialog" under "ASCII_TEMPLATE" (*IDL Reference Guide*) for instructions on how to use the dialog to define the structure of your ASCII data.

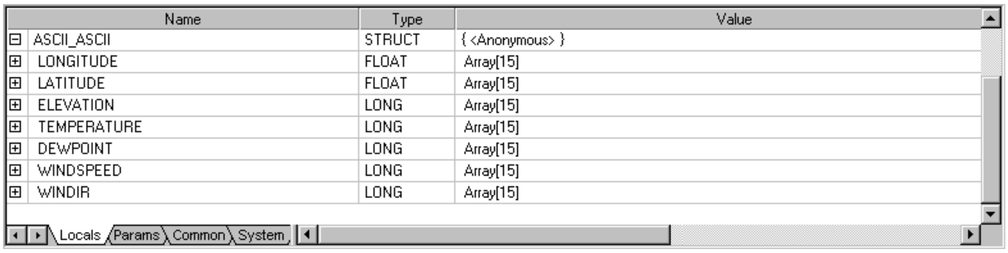
ASCII files opened with the **Import ASCII** macro are stored in structure variables which are named *filename*_ASCII where *filename* is the name of the file you opened without the extension.

Note

IDL variables must begin with a letter, and may contain only letters, digits, the underscore character, or the dollar sign. If the first character of *filename* is not a letter, the prefix "var" is added to the variable name. Any spaces within *filename* are converted to underscores. Any other illegal characters within *filename* are removed.

For example, if you opened `ascii.txt`, the data is now in the structure variable named ASCII_ASCII. Each field (named in the ASCII Template dialog) is an element of the structure.

The structure can be viewed in the Variable Watch Window.



Name	Type	Value
⊟ ASCII_ASCII	STRUCT	{ <Anonymous> }
⊞ LONGITUDE	FLOAT	Array[15]
⊞ LATITUDE	FLOAT	Array[15]
⊞ ELEVATION	LONG	Array[15]
⊞ TEMPERATURE	LONG	Array[15]
⊞ DEWPOINT	LONG	Array[15]
⊞ WINDSPEED	LONG	Array[15]
⊞ WINDIR	LONG	Array[15]

Figure 1-3: Variable Watch Window Showing ASCII_ASCII Structure

You can specify which part of the structure variable you want to access by using the following syntax:

variable_name.element_name

For example, if you want to view the Longitude field data, enter the following:

```
Print, ASCII_ASCII.LONGITUDE
```

If you want to plot the Temperature data, enter the following:

```
IPLOT, ASCII_ASCII.TEMPERATURE
```

The following figure results.

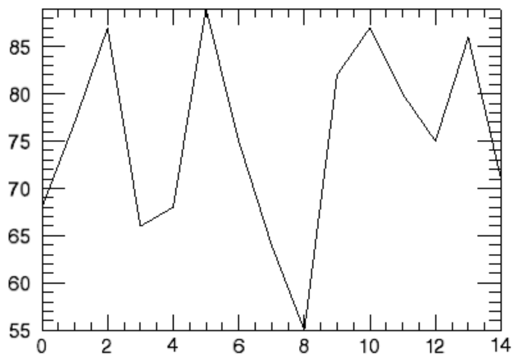


Figure 1-4: Plot of ASCII_ASCII.TEMPERATURE

Using Macros to Import Binary Files

To import a binary file into IDL using a macro, complete the following steps:

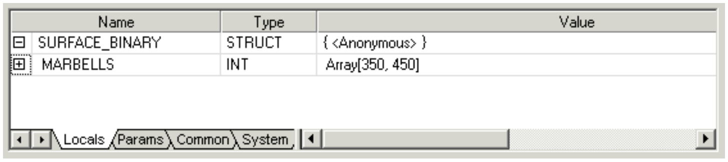
1. Select the **Import Binary** toolbar button. The **Select a binary file to read** dialog appears.
2. Select a file to import. For example, select the `surface.dat` from the `examples/data` directory in your IDL installation directory. Click **Open**.
3. See Using the BINARY_TEMPLATE Interface under "BINARY_TEMPLATE" (*IDL Reference Guide*) for instructions on how to use the dialog to define the structure of your binary data.

Binary files opened with the **Import Binary File** macro are stored in structure variables which are named *filename*_BINARY where *filename* is the name of the file you opened without the extension.

Note

IDL variables must begin with a letter, and may contain only letters, digits, the underscore character, or the dollar sign. If the first character of *filename* is not a letter, the prefix "var" is added to the variable name. Any spaces within *filename* are converted to underscores. Any other illegal characters within *filename* are removed.

So, the file we just opened (`surface.dat`) is now in the structure variable named SURFACE_BINARY. The variable is a structure, and contains elements that are the field names defined in the **Binary Template** dialog. In this case the single field is named `marbells`. The structure can be viewed in the Variable Watch Window.



Name	Type	Value
⊟ SURFACE_BINARY	STRUCT	{ <Anonymous> }
⊞ MARBELLS	INT	Array[350, 450]

Figure 1-5: Variable Watch Window Showing MARBELLS_BINARY Structure

Access data from the structure variable using the following syntax:

variable_name.element_name

For example, display the surface by entering:

```
ISURFACE, SURFACE_BINARY.marbells
```

Using Macros to Import HDF Files

To import a Hierarchical Data Format (HDF), HDF-EOS, or NETCDF file into IDL, complete the following steps:

1. Select the **Import HDF File** toolbar button. The **Select a valid HDF, NETCDF or HDF-EOS file** dialog is displayed.
2. Select a file to import. Click **Open**.
3. See “Using the HDF Browser Interface” under “HDF_BROWSER” for instructions on how to use the dialog.

After selecting to import data and clicking **OK**, HDF, NETCDF, or HDF-EOS files read with the **Import HDF** macro are stored in structure variables which are named *filename*_DF where *filename* is the name of the file you opened without the extension.

Note

IDL variables must begin with a letter, and may contain only letters, digits, the underscore character, or the dollar sign. If the first character of *filename* is not a letter, the prefix “var” is added to the variable name. Any spaces within *filename* are converted to underscores. Any illegal characters within *filename* are removed.

The variable is a structure with each data or metadata name being an element of the structure. You can specify which part of the structure variable you want to access by using the following syntax:

variable_name.data_name

For example, if you imported two data elements out of a file named `hydrogen.hdf` and you named the elements `IMAGE1` and `IMAGE2`, you could access each individual data element using the following:

```
HYDROGEN_DF.IMAGE1
HYDROGEN_DF.IMAGE2
```

If you wanted to view `IMAGE1`, you would enter:

```
IIMAGE, HYDTROGEN_DF.IMAGE1
```

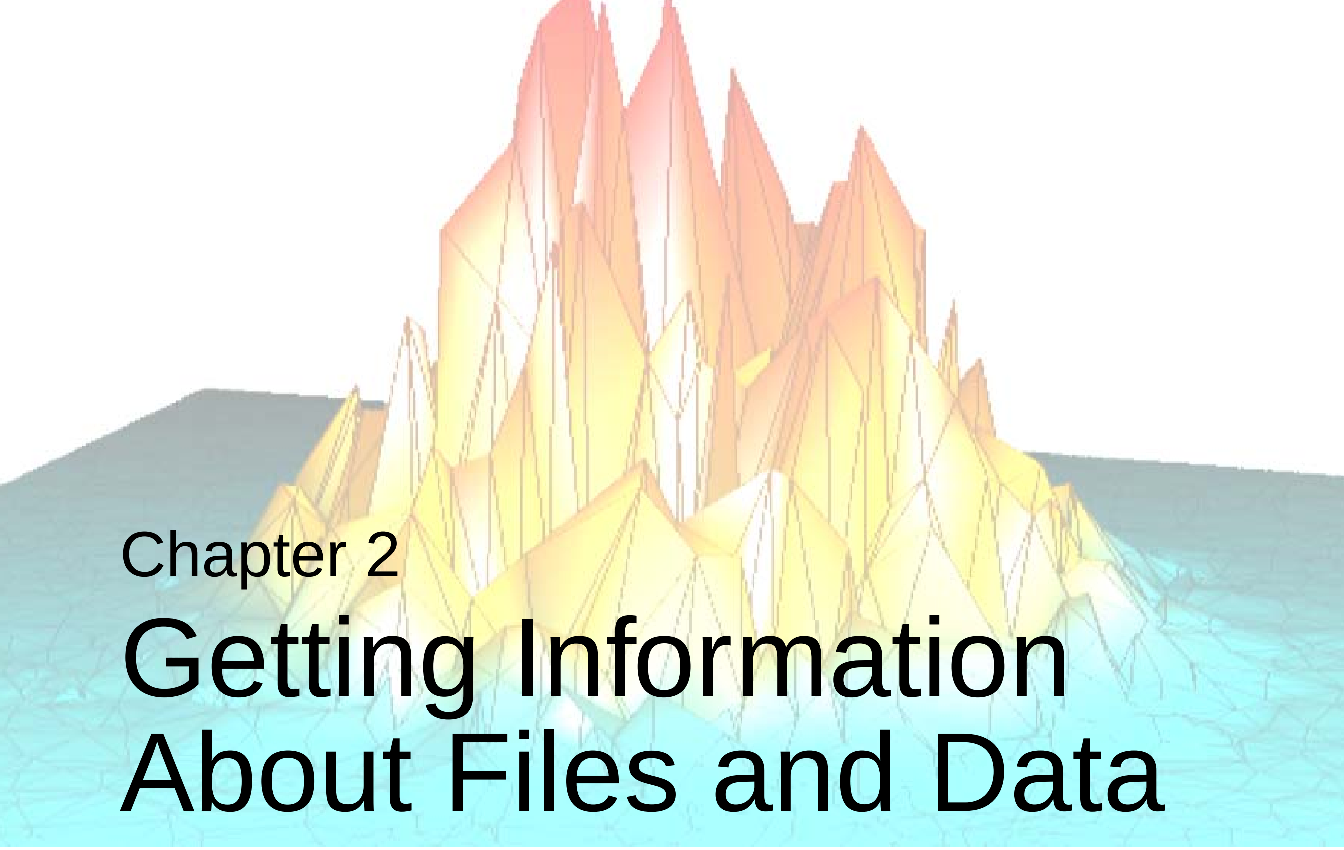
For more information on IDL support of HDF and other scientific data formats, see the *Scientific Data Formats* manual.

For information on importing HDF5 files using the **HDF5 Browser** dialog, see “H5_BROWSER” (*IDL Reference Guide*)

File Access Routines

See the following categories under "Input/Output" (*IDL Quick Reference*) for a list of available file and data access routines:

- "Image Data Formats" — includes read and write routines for supported image formats (such as JPEG, TIFF, DICOM, etc.), and routines that launch dialogs for image file access.
- "Scientific Data Formats" — includes CDF, EOS, NCDF, HDF, and HDF5 routines.
- "Other Data Formats" — includes routines that access ASCII, BINARY, XML, and other non-image data formats.
- "General Input/Output" — includes READ, WRITE and other routines commonly used when accessing unformatted data. Also see Chapter 18, "Files and Input/Output" for information on using these routines and formatting your data.



Chapter 2

Getting Information About Files and Data

The following topics are covered in this chapter:

Investigating Files and Data

There are a number of routines and functions in IDL that allow you to quickly access information about your data. While it is always a good idea to know your data before processing, the routines in this chapter can help you uncover details of arrays, expressions, SAVE files, objects, or specific images.

Accessing Information in iTools

When you are working in the iTools, there are a number of ways to get information about variable data, an object's properties, an image's statistical information, and the data hierarchy. For more information about these options, see the following topics:

- "About the Data Manager" (Chapter 2, *iTool User's Guide*) provides information on data associated with a visualization
- "The Visualization Browser" (Chapter 6, *iTool User's Guide*) provides information on the properties of a visualization
- "Additional Operations" (Chapter 7, *iTool User's Guide*) describes the Histogram and Statistics windows available in iTools

Returning Image File Information

When accessing formatted image data (not contained in a binary file), there are a number of ways to get information about the data characteristics. The most flexible is the QUERY_IMAGE routine, which returns a structure that includes the number of image channels, pixel data type and palette information. If you need specific information from a formatted image file, you can use the QUERY* routine specifically designed for images of that format.

Note

You can also use the SIZE function to quickly return the size of an image array. See "Using SIZE to Return Image Dimensions" on page 39 for details.

Using the QUERY_IMAGE Info Structure

Common image file formats contain standardized header information that can be queried. IDL provides the QUERY_IMAGE function to return valuable information about images stored in supported image file formats.

For example, using the QUERY_IMAGE function, you can return information about the `mineral.png` file in the `examples/data` directory. First, access the file. Then use the QUERY_IMAGE function to return information about the file:

```
file = FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])
queryStatus = QUERY_IMAGE(file, info)
```

To determine the success of the QUERY_IMAGE function, print the value of the *query* variable:

```
PRINT, 'Status = ', queryStatus
```

IDL prints

```
queryStatus =           1
```

If *queryStatus* is zero, the file cannot be accessed with IDL. If *queryStatus* is one, the file can be accessed. Because the query was successful, the *info* variable is now an IDL structure containing image parameters. The tags associated with this structure variable are standard across image files. You can view the tags of this structure by setting the STRUCTURE keyword to the HELP command with the *info* variable as its argument:

```
HELP, info, /STRUCTURE
```

IDL displays the following text in the Output Log:

```
** Structure <1407e70>, 7 tags, length=36, refs=1:
   CHANNELS        LONG                 1
   DIMENSIONS      LONG          Array[2]
   HAS_PALETTE     INT                  1
   IMAGE_INDEX     LONG                 0
   NUM_IMAGES      LONG                 1
   PIXEL_TYPE      INT                  1
   TYPE            STRING           'PNG'
```

The structure tags provide the following information:

Tag	Description

`CHANNELS`	Provides the number of dimensions within the image array:
	• 1 – two-dimensional array
	• 3 – three-dimensional array
	Print the number of dimensions using:
	`PRINT, 'Number of Channels: ', info.channels`
	For the `mineral.png` file, IDL prints:
	`Number of Channels:           1`

`DIMENSIONS`	Contains image array information including the width and height. Print the image dimensions using:
	`PRINT, 'Size: ', info.dimensions`
	For the `mineral.png` file, IDL prints:
	`Size:           288         216`

`HAS_PALETTE`	Describes the presence or absence of a color palette:
	• 1 (True) – the image has an associated palette
	• 0 (False) – the image does not have an associated palette
	Print whether a palette is present or not using:
	`PRINT, 'Is Palette Available?: ', info.has_palette`
	For the `mineral.png` file, IDL prints:
	`Is Palette Available?:           1`

Table 2-1: Image Structure Tag Information

Tag	Description
`IMAGE_INDEX`	Gives the zero-based index number of the current image. Print the index of the image using: `PRINT, 'Image Index: ', info.image_index` For the `mineral.png` file, IDL prints: `Image Index:           0`
`NUM_IMAGES`	Provides the number of images in the file. Print the number of images in the file using: `PRINT, 'Number of Images: ', info.num_images` For the `mineral.png` file, IDL prints: `Number of Images:           1`

Table 2-1: Image Structure Tag Information (Continued)

Tag	Description

`PIXEL_TYPE`	Provides the IDL type code for the image pixel data type:
	• 0 – Undefined
	• 1 – Byte
	• 2 – Integer
	• 3 – Longword integer
	• 4 – Floating point
	• 5 – Double-precision floating
	• 6 – Complex floating
	• 9 – Double-precision complex
	• 12 – Unsigned Integer
	• 13 – Unsigned Longword Integer
	• 14 – 64-bit Integer
	• 15 – Unsigned 64-bit Integer
	See “IDL Type Codes and Names” under the SIZE function in the *IDL Reference Guide* for a complete list of type codes.
	Print the data type of the pixels in the image using:
	`PRINT, 'Data Type: ', info.pixel_type`
	For the `mineral.png` file, IDL displays the following text in the Output Log:
	`Data Type:            1`

`TYPE`	Identifies the image file format. Print the format of the file containing the image using:
	`PRINT, 'File Type: ' + info.type`
	For the `mineral.png` file, IDL prints:
	`File Type: PNG`

Table 2-1: Image Structure Tag Information (Continued)

From the contents of the *info* variable, it can be determined that the single image within the `mineral.png` file is an indexed image because it has only one channel (is a two-dimensional array) and it has a color palette. The image also has byte pixel data.

Note

When working with RBG images (with a CHANNELS value of 3) it is important to determine the interleaving (the arrangement of the red, green, and blue channels of data) in order to properly display these image. See "RGB Image Interleaving" (Chapter 3, *Using IDL*) for an example that shows you how to determine the arrangement of these channels.

Using Specific QUERY_* Routines

All of the QUERY_* routines return a status, which determines if the file can be read using the corresponding READ_ routine. All of these routines also return the `Info` structure, (described in the previous section), which reports image dimensions, number of samples per pixel, pixel type, palette info, and the number of images in the file. However, some of the QUERY_* routines (such as QUERY_MRSID and QUERY_TIFF) return more detailed information particular to that specific image format. See "Query Routines" (*IDL Quick Reference*) for a complete list of the available QUERY_* routines.

Returning Type and Size Information

The SIZE function returns size and type information for a given expression. The returned vector is always of longword type.

- The first element is equal to the number of dimensions of the parameter and is zero if the parameter is a scalar.
- The next elements contain the size of each dimension.
- After the dimension sizes, the last two elements indicate the data type and the total number of elements, respectively.

See "IDL Type Codes and Names" under the SIZE function in the *IDL Reference Guide* for a complete list of type codes. See the following examples for more information on the SIZE function:

- "Determining if a Variable is a Scalar or an Array" below
- "Using SIZE to Return Image Dimensions" on page 39

In addition to the examples listed above, also see the following SIZE function examples in the *IDL Reference Guide*:

- "Example: Returning Array Dimension Information"
- "Example: Returning the IDL Type Code of an Expression"

Determining if a Variable is a Scalar or an Array

The SIZE function can be used to determine whether a variable holds a scalar value or an array. Setting the DIMENSIONS keyword causes the SIZE function to return a 0 if the variable is a scalar, or the dimensions if the variable is an array:

```
A = 1
B = [1]
C = [1,2,3]
D = [[1,2],[3,4]]

PRINT, SIZE(A, /DIMENSIONS)
PRINT, SIZE(B, /DIMENSIONS)
PRINT, SIZE(C, /DIMENSIONS)
PRINT, SIZE(D, /DIMENSIONS)
```

IDL Prints:

```
0
1
3
```

```
       2     2
```

Using SIZE to Return Image Dimensions

The following example reads an image array and uses the SIZE function DIMENSIONS keyword to access the number of rows and columns in the image file. In this simple example, the information is used to create a display window of the correct size.

```
PRO ex_displayImage

; Select and read the image file.
earth = READ_PNG (FILEPATH ('avhrr.png', $
   SUBDIRECTORY = ['examples', 'data']), R, G, B)

; Load the color table and designate white to occupy the
; final position in the red, green and blue bands.
TVLCT, R, G, B
maxColor = !D.TABLE_SIZE - 1
TVLCT, 255, 255, 255, maxColor

; Prepare the display device.
DEVICE, DECOMPOSED = 0, RETAIN = 2

; Get the size of the original image array.
earthSize = SIZE(earth, /DIMENSIONS)

; Prepare a window and display the new image.
WINDOW, 0, XSIZE = earthSize[0], YSIZE = earthSize[1]
TV, earth

END
```

Getting Information About SAVE Files

The IDL_Savefile object provides an object-oriented interface that allows you to query a SAVE file for information and restore one or more individual items from the file. Using IDL_Savefile, you can retrieve information about the user, machine, and system that created the SAVE file, as well as the number and size of the various items contained in the file (variables, common blocks, routines, *etc*). Individual items can be selectively restored from the SAVE file.

Use IDL_Savefile in preference to the RESTORE procedure when you need to obtain detailed information on the items contained within a SAVE file without first restoring it, or when you wish to restore only selected items. Use RESTORE when you want to restore everything from the SAVE file using a simple interface.

Note

The IDL_Savefile object does not provide methods that allow you to modify an existing SAVE file. The only way to modify an existing SAVE file is to restore its contents into a fresh IDL session, modify the contained routines or variables as necessary, and use the SAVE procedure to create a new version of the file.

To use the IDL_Savefile object to restore items from an existing SAVE file, do the following:

- Create a Savefile Object
- Query the Savefile Object
- Restore Items from the Savefile Object
- Destroy the Savefile Object

The following sections describe each of these steps. For complete information on the IDL_Savefile object and its methods, see "IDL_Savefile" (Chapter 11, *IDL Reference Guide*).

Create a Savefile Object

When an IDL_Savefile object is instantiated, it opens the actual SAVE file for reading and creates an in-memory representation of its contents — without actually restoring the file. The savefile object persists until it is explicitly destroyed (or until the IDL session ends); the SAVE file itself is held open for reading as long as the savefile object exists.

To create a savefile object from the `draw_arrow.sav` file created in "Example: A SAVE File of a Simple Routine" (Chapter 4, *Application Programming*), use the following command:

```
myRoutines = OBJ_NEW('IDL_Savefile', 'draw_arrow.sav')
```

Similarly, to create a savefile object from the saved image data, use the following command:

```
myImage = OBJ_NEW('IDL_Savefile', 'imagefile.sav')
```

Query the Savefile Object

Once you have created a savefile object, three methods allow you to retrieve information about its contents:

- The Contents method provides information about the SAVE file including the number and type of items contained therein.
- The Names method allows you to retrieve the names of routines and variables stored in the file.
- The Size method allows you to retrieve size and type information about the variables stored in the file.

Contents Method

The Contents method returns a structure variable that describes the SAVE file and its contents. The individual fields in the returned structure are described in detail in "IDL_Savefile::Contents" (Chapter 11, *IDL Reference Guide*).

In addition to providing information about the system that created the SAVE file, the Contents method allows you to determine the number of each type of saved item (variable, procedure, function, *etc*.) in the file. This information can be used to programmatically restore items from the SAVE file.

Assuming you have created the `myRoutines` savefile object, the data returned by the Contents method looks like this:

```
savefileInfo = myRoutines->Contents()
HELP, savefileInfo, /STRUCTURE
```

IDL Prints:

```
** Structure IDL_SAVEFILE_CONTENTS, 17 tags, length=176, data leng
th=172:
   FILENAME            STRING    '/itt/test/draw_arrow.sav'
   DESCRIPTION         STRING    ''
   FILETYPE            STRING    'Portable (XDR)'
```

```
      USER                STRING    'dquixote'
      HOST                STRING    'DULCINEA'
      DATE                STRING    'Thu May 08 12:04:46 2003'
      ARCH                STRING    'x86'
      OS                  STRING    'Win32'
      RELEASE             STRING    '6.4'
      N_COMMON            LONG64                          0
      N_VAR               LONG64                          0
      N_SYSVAR            LONG64                          0
      N_PROCEDURE         LONG64                          2
      N_FUNCTION          LONG64                          0
      N_OBJECT_HEAPVAR    LONG64                          0
      N_POINTER_HEAPVAR   LONG64                          0
      N_STRUCTDEF         LONG64                          0
```

From this you can determine the name of the SAVE file from which the information was extracted, the names of the user and computer who created the file, the creation date, and information about the IDL system that created the file. You can also see that the SAVE file contains definitions for two procedures and nothing else.

Names Method

The Names method returns a string array containing the names of the variables, procedures, functions, or other items contained in the SAVE file. By default, the method returns the names of variables; keywords allow you to specify that names of other items should be retrieved. The available keyword options are described in "IDL_Savefile::Names" (Chapter 11, *IDL Reference Guide*).

The names of items retrieved using the Names method can be supplied to the Size method to retrieve size and type information about the specific items, or to the Restore method to restore individual items.

For example, calling the Names method with the PROCEDURE keyword on the `myRoutines` savefile object yields the names of the two procedures saved in the file:

```
PRINT, myRoutines->Names(/PROCEDURE)
```

IDL Prints:

```
ARROW  DRAW_ARROW
```

Similarly, to retrieve the name of the variable saved in `imagefile.sav`, which is referred to by the `myImage` savefile object:

```
PRINT, myImage->Names()
```

IDL Prints:

```
IMAGE
```

Size Method

The Size method returns the same information about a variable stored in a SAVE file as the SIZE function does about a regular IDL variable. It accepts the same keywords as the SIZE function, and returns the same information using the same formats. The Size method differs only in that the argument is a string or integer identifier string (returned by the Names method) that specifies an item within a SAVE file, rather than an in-memory expression. See "IDL_Savefile::Size" (Chapter 11, *IDL Reference Guide*) for additional details.

For example, to determine the dimensions of the image stored in the `imagefile.sav` file, do the following:

```
imagesize = myImage->Size('image', /DIMENSIONS)
PRINT, 'Image X size:', imagesize[0]
PRINT, 'Image Y size:', imagesize[1]
```

IDL Prints:

```
Image X size:         256
Image Y size:         256
```

Restore Items from the Savefile Object

The Restore method allows you to selectively restore one or more items from the SAVE file associated with a savefile object. Items to be restored are specified using the item name strings returned by the Names method. In addition to functions, procedures, and variables, you can also restore COMMON block definitions, structure definitions, and heap variables. See "IDL_Savefile::Restore" (Chapter 11, *IDL Reference Guide*) for additional details.

For example, to restore the DRAW_ARROW procedure without restoring the ARROW procedure, do the following:

```
myRoutines->Restore, 'draw_arrow'
```

Note on Restoring Objects and Pointers

Object references and pointers rely on special IDL variables called *heap variables*. When you restore a regular IDL variable that contains an object reference or a pointer, the associated heap variable is restored automatically; there is no need to restore the heap variables separately. It is, however, possible to restore the heap variables independently of any regular IDL variables; see "Restoring Heap Variables Directly" (Chapter 11, *IDL Reference Guide*) for complete details.

Destroy the Savefile Object

To destroy a savefile object, use the OBJ_DESTROY procedure:

```
OBJ_DESTROY, myRoutines
OBJ_DESTROY, myImage
```

Destroying the savefile object will close the SAVE file with which the object is associated.

Returning Object Type and Validity

Three IDL routines allow you to obtain information about an existing object: OBJ_CLASS, OBJ_ISA, and OBJ_VALID.

OBJ_CLASS

Use the OBJ_CLASS function to obtain the class name of a specified object, or to obtain the names of a specified object's direct superclasses. For example, if we create the following class structures:

```
struct = {class1, data1:0.0 }
struct = {class2, data2a:0, data2b:0L, INHERITS class1 }
```

We can now create an object and use OBJ_CLASS to determine its class and superclass membership.

```
; Create an object.
A = OBJ_NEW('class2')

; Print A's class membership.
PRINT, OBJ_CLASS(A)
```

IDL prints:

```
CLASS2
```

Or you can print as superclasses:

```
; Print A's superclasses.
PRINT, OBJ_CLASS(A, /SUPERCLASS)
```

IDL prints:

```
CLASS1
```

See "OBJ_CLASS" (*IDL Reference Guide*) for further details.

OBJ_ISA

Use the OBJ_ISA function to determine whether a specified object is an instance or subclass of a specified object. For example, if we have defined the object A as above:

```
IF OBJ_ISA(A, 'class2') THEN $
    PRINT, 'A is an instance of class2.'
```

IDL prints:

```
A is an instance of class2.
```

See "OBJ_ISA" (*IDL Reference Guide*) for further details.

OBJ_VALID

Use the OBJ_VALID function to verify that one or more object references refer to valid and currently existing object heap variables. If supplied with a single object reference as its argument, OBJ_VALID returns TRUE (1) if the reference refers to a valid object heap variable, or FALSE (0) otherwise. If supplied with an array of object references, OBJ_VALID returns an array of TRUE and FALSE values corresponding to the input array. For example:

```
; Create a class structure.
struct = {cname, data:0.0}

; Create a new object.
A = OBJ_NEW('CNAME')

IF OBJ_VALID(A) PRINT, "A refers to a valid object." $
    ELSE PRINT, "A does not refer to a valid object."
```

IDL prints:

```
A refers to a valid object.
```

If we destroy the object:

```
; Destroy the object.
OBJ_DESTROY, A

IF OBJ_VALID(A) PRINT, "A refers to a valid object." $
    ELSE PRINT, "A does not refer to a valid object."
```

IDL prints:

```
A does not refer to a valid object.
```

See “OBJ_VALID” (*IDL Reference Guide*) for further details.

Returning Information About a File

You can use the FILE_INFO function to retrieve information about a file that is not currently open. To get information about an open file (for which there is an IDL Logical Unit Number), use the HELP procedure or the FSTAT function. See "Returning Information About a File Unit" (Chapter 18, *Application Programming*).

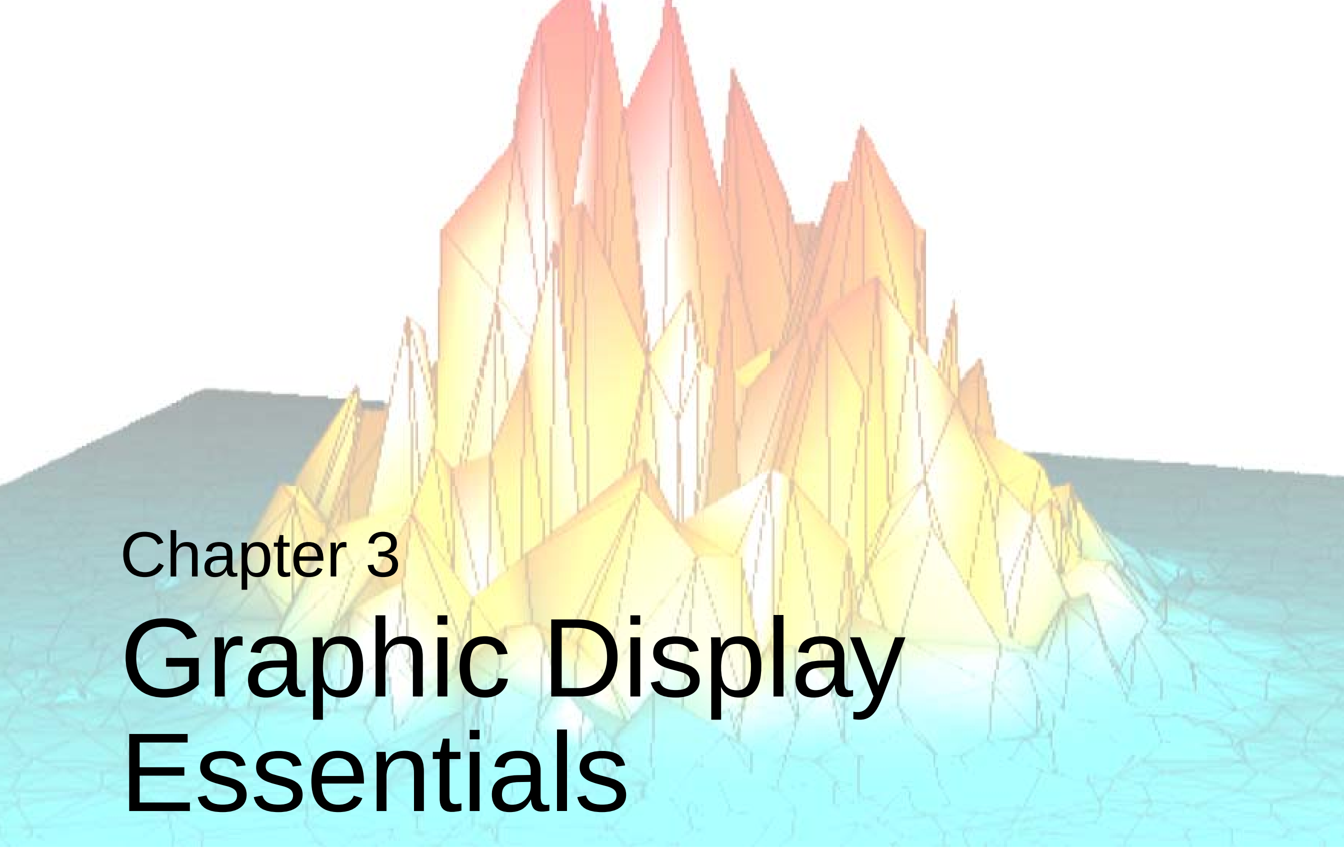
The FILE_INFO function returns a structure expression of type FILE_INFO containing information about the file. For example, get information on `dist.pro`:

```
HELP,/STRUCTURE, FILE_INFO(FILEPATH('dist.pro',
SUBDIRECTORY='lib'))
```

The above command will produce output similar to:

```
** Structure FILE_INFO, 21 tags, length=72:
     NAME            STRING   '/usr/local/itt/idl/lib/dist.pro'
     EXISTS          BYTE        1
     READ            BYTE        1
     WRITE           BYTE        0
     EXECUTE         BYTE        0
     REGULAR         BYTE        1
     DIRECTORY       BYTE        0
     BLOCK_SPECIAL   BYTE        0
     CHARACTER_SPECIAL
                     BYTE        0
     NAMED_PIPE      BYTE        0
     SETGID          BYTE        0
     SETUID          BYTE        0
     SOCKET          BYTE        0
     STICKY_BIT      BYTE        0
     SYMLINK         BYTE        0
     DANGLING_SYMLINK
                     BYTE        0
     MODE            LONG              420
     ATIME           LONG64                970241431
     CTIME           LONG64                970241595
     MTIME           LONG64                969980845
     SIZE            LONG64                     1717
```

The fields of the FILE_INFO structure provide various information about the file, such as the size of the file, and the dates of last access, creation, and last modification. For more information on the fields of the FILE_INFO structure, see "FILE_INFO" (*IDL Reference Guide*). See "FILE_LINES" (*IDL Reference Guide*) for information on how to retrieve the number of lines of text in a file.



Chapter 3

Graphic Display Essentials

The following topics are covered in this chapter:

IDL Visual Display Systems

When creating visualizations in IDL, you can choose to create a visualization in an IDL Intelligent Tool (iTool), in an Object Graphics display, or in a Direct Graphics display:

- **iTools** — introduced in IDL 6.0, the IDL Intelligent Tools (iTools) provide the power and flexibility of Object Graphics with a pre-built visualization system that offers a great deal of interactivity. This set of interactive utilities combine data analysis and visualization with the task of producing presentation quality graphics. See "iTools Visualizations" below for more information.
- **Object Graphics** — introduced in IDL 5.0, Object Graphics use an object-oriented programmers' interface to create graphic objects, which must then be drawn, explicitly, to a destination of the programmer's choosing. See "IDL Object Graphics" on page 51 for more information.
- **Direct Graphics** — the oldest visualization system of the three, Direct Graphics rely on the concept of a current graphics device to quickly create simple static visualizations using IDL commands like PLOT or SURFACE. See "IDL Direct Graphics" on page 52 for information.

This chapter introduces the IDL display systems and provides information on common topics shared by the systems. Topics include a discussion on coordinates, coordinate conversion, interpolation, color systems and color schemes, and fonts.

iTools Visualizations

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

The main enhancements the new iTools provide are more mouse interactivity, WYSIWYG (What-You-See-Is-What-You-Get) printing, built-in analysis, undo-redo capabilities, layout control, and better-looking plots. These robust, pre-built tools reduce the amount of programming IDL users must do to create interactive visualizations. At the same time, the iTools integrate in a seamless manner with the IDL Command Line, user interface controls, and custom algorithms. In this way, the iTools maintain and enhance the control and flexibility IDL users rely on for data exploration, algorithm design, and rapid application development. The following manuals provide more information:

- *iTool User's Guide* — describes how to create visualization using iTools
- *iTool Programming* — describes how to create and customize an iTool

IDL Object Graphics

The salient features of Object Graphics are:

- Object graphics are device independent. There is no concept of a current graphics device when using object-mode graphics; any graphics object can be displayed on any physical device for which a destination object can be created.
- Object graphics are object-oriented. Graphic objects are meant to be created and re-used; you may create a set of graphic objects, modify their attributes, draw them to a window on your computer screen, modify their attributes again, then draw them to a printer device without reissuing all of the IDL commands used to create the objects. Graphics objects also encapsulate functionality; this means that individual objects include method routines that provide functionality specific to the individual object.
- Object graphics are rendered in three dimensions. Rendering implies many operations not needed when drawing Direct Graphics, including calculation of normal vectors for lines and surfaces, lighting considerations, and general object overhead. As a result, the time needed to render a given object—a surface, say—will often be longer than the time taken to draw the analogous image in Direct Graphics.
- Object Graphics use a programmer's interface. Unlike Direct Graphics, which are well suited for both programming and interactive, ad hoc use, Object Graphics are designed to be used in programs that are compiled and run. While it is still possible to create and use graphics objects directly from the IDL command line, the syntax and naming conventions make it more convenient to build a program offline than to create graphics objects on the fly.
- Because Object Graphics persist in memory, there is a greater need for the programmer to be cognizant of memory issues and memory leakage. Efficient design—remembering to destroy unused object references and cleaning up—will avert most problems, but even the best designs can be memory-intensive if large numbers of graphic objects (or large datasets) are involved.

For more information on creating Object Graphic visualizations see:

- *Object Programming* — this manual introduces using IDL objects and also describes how to create custom objects in IDL.
- "Object Class and Method Reference" (*IDL Reference Guide*) — this section in the *IDL Reference Guide* provides complete reference material describing IDL's object classes
- *iTool User's Guide* and *iTool Programming* — these manuals provide complete details about using and creating object-based iTool displays

IDL Direct Graphics

IDL Direct Graphics is the original graphics rendering system introduced in IDL. Graphic displays creating using Direct Graphics are static — once created, no changes can be made without recreating the visualization being displayed. If you have used routines such as PLOT or SURFACE, you are already familiar with this graphics system. The salient features of Direct Graphics are:

- Direct Graphics use a graphics device (**X** for X-windows systems displays, **WIN** for Microsoft Windows displays, **PS** for PostScript files, etc.). You switch between graphics devices using the SET_PLOT command, and control the features of the current graphics device using the DEVICE command.
- IDL commands that existed in IDL 4.0 use Direct Graphics. Commands like PLOT, SURFACE, XYOUTS, MAP_SET, etc. all draw their output directly on the current graphics device.
- Once a direct-mode graphic is drawn to the graphics device, it cannot be altered or re-used. This means that if you wish to re-create the graphic on a different device, you must re-issue the IDL commands to create the graphic.
- When you add a new item to an existing direct-mode graphic (using a routine like OPLOT or XYOUTS), the new item is drawn in front of the existing items.

See "Direct Graphics" (*IDL Quick Reference*) for a list of available routines.

IDL Coordinate Systems

You can specify coordinates to IDL in one of the following coordinate systems:

DATA Coordinates

This coordinate system is established by the most recent PLOT, CONTOUR, or SURFACE procedure. This system usually spans the plot window, the area bounded by the plot axes, with a range identical to the range of the plotted data. The system can have two or three dimensions and can be linear, logarithmic, or semi-logarithmic. The mechanisms of converting from one coordinate system to another are described below.

DEVICE Coordinates

This coordinate system is the physical coordinate system of the selected plotting device. Device coordinates are integers, ranging from (0, 0) at the bottom-left corner to (V_x –1, V_y –1) at the upper-right corner. V_x and V_y are the number of columns and rows addressed by the device. These numbers are stored in the system variable !D as !D.X_SIZE and !D.Y_SIZE. In a widget base, device coordinates are measures from the upper-left corner

NORMAL Coordinates

The normalized coordinate system ranges from zero (0) to one (1) over each of the three axes.

Almost all of the IDL graphics procedures accept parameters in any of these coordinate systems. Most procedures use the data coordinate system by default. Routines beginning with the letters TV are notable exceptions. They use device coordinates by default. You can explicitly specify the coordinate system to be used by including one of the keyword parameters /DATA, /DEVICE, or /NORMAL in the call.

Understanding Windows and Related Device Coordinates

Images are displayed within a window (Direct Graphics) or within an instance of a window object (Object Graphics). In Direct Graphics, the WINDOW procedure is used to initialize the coordinates system for the image display. In Object Graphics, the IDLgrWindow, IDLgrView, and IDLgrModel objects are used to initialize the coordinate system for the image display.

A coordinate system determines how and where the image appears within the window. You can specify coordinates to IDL using one of the following coordinate systems:

- Data Coordinates — This system usually spans the window with a range identical to the range of the data. The system can have two or three dimensions and can be linear, logarithmic, or semi-logarithmic.
- Device Coordinates — This coordinate system is the physical coordinate system of the selected device. Device coordinates are integers, ranging from (0, 0) at the bottom-left corner to (V_x –1, V_y –1) at the upper-right corner of the display. V_x and V_y are the number of columns and rows of the device (a display window for example).

  Note

  For images, the data coordinates are the same as the device coordinates. The device coordinates of an image are directly related to the pixel locations within an image. Unless otherwise specified, IDL draws each image pixel per each device pixel.

- Normal Coordinates — The normalized coordinate system ranges from zero to one over columns and rows of the device.

Coordinates of 3-D Graphics

Points in *xyz* space are expressed by vectors of homogeneous coordinates. These vectors are translated, rotated, scaled, and projected onto the two-dimensional drawing surface by multiplying them by transformation matrices. The geometrical transformations used by IDL, and many other graphics packages, are taken from Chapters 7 and 8 of Foley and Van Dam (Foley, J.D., and A. Van Dam (1982), *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Co.). The reader is urged to consult this book for a detailed description of homogeneous coordinates and transformation matrices since this section presents only an overview. Three-dimensional graphics, coordinate systems, and transformations also are included in this chapter.

Homogeneous Coordinates

A point in homogeneous coordinates is represented as a four-element column vector of three coordinates and a scale factor w ¼¼≠ 0. For example:

$$P(wx, wy, wz, w) \equiv \mathrm{P}(x/w, y/w, z/w, 1) \equiv (x, y, z)$$

One advantage of this approach is that translation, which normally must be expressed as an addition, can be represented as a matrix multiplication. Another advantage is that homogeneous coordinate representations simplify perspective transformations. The notion of rows and columns used by IDL is opposite that of Foley and Van Dam (1982). In IDL, the column subscript is first, while in Foley and Van Dam (1982) the row subscript is first. This changes all row vectors to column vectors and transposes matrices.

Right-Handed Coordinate System

The coordinate system is right-handed so that when looking from a positive axis to the origin, a positive rotation is counterclockwise. As usual, the *x*-axis runs across the display, the *y*-axis is vertical, and the positive *z*-axis extends out from the display to the viewer. For example, a 90-degree positive rotation about the *z*-axis transforms the *x*-axis to the *y*-axis.

Transformation Matrices

Transformation matrices, which post-multiply a point vector to produce a new point vector, must be (4, 4). A series of transformation matrices can be concatenated into a single matrix by multiplication. If A1, A2, and A3 are transformation matrices to be applied in order, and the matrix A is the product of the three matrices, the following applies.

$$((P \bullet A_1) \bullet A_2) \bullet A_3 \equiv \mathrm{P} \bullet ((A_1 \bullet A_2) \bullet A_3) = P \bullet A$$

In Object Graphics, IDL the model object that contains the displayed object stores the transformation matrix. In Direct Graphics, IDL stores the concatenated transformation matrix in the system variable field !P.T.

Note

When displaying objects in a three-dimensional view, you can precisely configure the object position using transformation matrices. See "Translating, Rotating and Scaling Objects" (Chapter 3, *Object Programming*) for details.

Note

For most Direct Graphic applications, it is not necessary to create, manipulate, or to even understand transformation matrices. See the T3D procedure, which implements most of the common transformations.

Each of the operations of translation, scaling, rotation, and shearing can be represented by a transformation matrix.

Translation

The transformation matrix to translate a point by (D_x, D_y, D_z) is shown below.

$$\begin{bmatrix} 1 & 0 & 0 & D_x \\ 0 & 1 & 0 & D_y \\ 0 & 0 & 1 & D_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

Scaling by factors of S_x, S_y, and S_z about the *x*-, *y*-, and *z*-axes respectively, is represented by the matrix below.

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

Rotation about the *x*-, *y*-, and *z*-axes is represented respectively by the following three matrices:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x & 0 \\ 0 & \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 & 0 \\ \sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Coordinate Conversions

Depending upon the data and type of visualization, you may want to convert between normalized, data or device coordinates (described in "IDL Coordinate Systems" on page 53). This section details two-dimensional and three-dimensional coordinate system characteristics provides resources for various coordinate conversions. See the following for details:

- "Two-Dimensional Coordinate Conversion" on page 58
- "Three-Dimensional Coordinate Conversion" on page 59
- "Using Coordinate Conversions" on page 59

Two-Dimensional Coordinate Conversion

This section describes the formulae for conversions to and from each coordinate system. In the following discussion, D_x is a data coordinate, N_x is a normalized coordinate, and R_x is a raw device coordinate. Let V_x and V_y represent the size of the visible area of the currently selected display or drawing surface.

The field S is a two-element array of scaling factors used to convert X coordinates from data units to normalized units. S contains the parameters of the linear equation, converting data coordinates to normalized coordinates. S[0] is the intercept, and S[1] is the slope. Also, let D_x be the data coordinate, N_x the normalized coordinate, R_x the device coordinate, V_x the device X size (in device coordinates).

With the above variables defined, the linear two-dimensional coordinate conversions for the *x* coordinate can be written as follows:

Coordinate Conversion	Linear	Logarithmic
Data to normal	$N_x = S_0 + S_1 D_x$	$N_x = S_0 + S_1 \log D_x$
Data to device	$R_x = V_x(S + S_1 D_x)$	$R_x = V_x(S_0 + S_1 \log D_x)$
Normal to device	$R_x = N_x V_x$	$R_x = N_x V_x$
Normal to data	$D_x = (N_x - S_0)/S_1$	$D_x = 10^{(N_x - S_0)/S_1}$
Device to data	$D_x = (R_x/V_x - S_0)/S_1$	$D_x = 10^{(R_x/V_x - S_0)/S_1}$
Device to normal	$N_x = R_x/V_x$	$N_x = R_x/V_x$

Table 3-1: Equations for X-axis Coordinate Conversion

The *y*- and *z*-axis coordinates are converted in exactly the same manner, with the exception that there is no *z* device coordinate and that logarithmic *z*-axes are not permitted.

This coordinate conversion functionality is built into object graphics through the XCOORD_CONVERT and YCOORD_CONVERT properties or each type of visualization object. If you are working with a Direct Graphics display, you can use the CONVERT_COORD function.

Three-Dimensional Coordinate Conversion

To convert from a three-dimensional coordinate to a two-dimensional coordinate, IDL follows these steps:

- Data coordinates are converted to three-dimensional normalized coordinates. To convert the *x* coordinate from data to normalized coordinates, use the formula $N_x = X_0 + X_1 D_x$. The same process is used to convert the *y* and *z* coordinates using !Y.S and !Z.S.
- The three-dimensional normalized coordinate, P = (N_x, N_y, N_z), whose homogeneous representation is (N_x, N_y, N_z, 1), is multiplied by the concatenated transformation matrix !P.T:

  $$\mathrm{P}' = P \bullet \texttt{!P.T}$$

- The vector P′ is scaled by dividing by *w*, and the normalized two-dimensional coordinates are extracted:

  $$N'_x = P'_x / P'_w \text{ and } N'_y = P'_y / P'_w$$

- The normalized *xy* coordinate is converted to device coordinates as described in "Two-Dimensional Coordinate Conversion" on page 58.

Using Coordinate Conversions

How coordinate conversions are defined depend upon the display type as follows:

- **iTools** — in an iTool display, the interactive nature of the tool makes coordinate conversions transparent. There is no need to programmatically configure the transformation matrices of the objects. See Chapter 4, "Manipulating the Display" (*iTool User's Guide*) for information on zooming, scaling and translation.
- **Object Graphics** — converting an object's data coordinates into normalized coordinates for display is a common task. See "Positioning Visualizations in a View" (Chapter 3, *Object Programming*) for details on the elements involved

in defining an object's position. Chapter 3, "Positioning Objects in a View" (*Object Programming*) also includes information on how to use coordinate conversions (see "Converting Data to Normal Coordinates") and information on programmatically defining the object's placement in a view (see "Translating, Rotating and Scaling Objects").

- **Direct Graphics** — the IDL Direct Graphics system automatically positions and sizes static visualizations so there is no need to set up a transformation matrix. However, you can convert between the supported coordinate systems. See "CONVERT_COORD" (*IDL Reference Guide*) for information on this conversion in Direct Graphics.

Interpolation Methods

When a visualization undergoes a geometric transformation, the location of each transformed pixel may not map directly to a center of a pixel location in the output visualization as shown in the following figure.

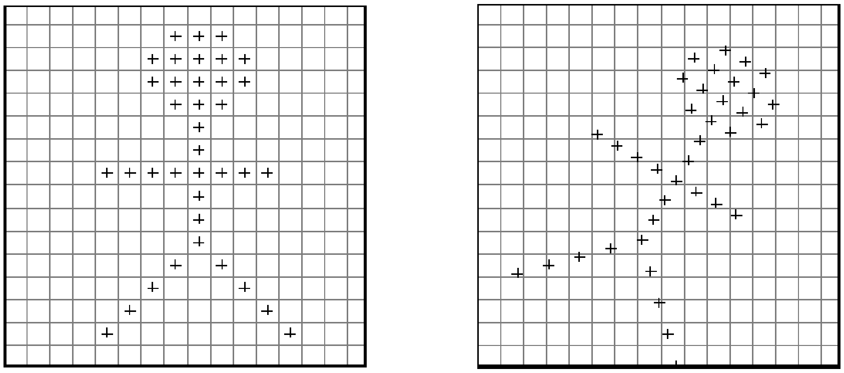


Figure 3-1: Original Pixel Center Locations (Left) and Rotated Pixel Center Locations (Right)

When the transformed pixel center does not directly coincide with a pixel in the output visualization, the pixel value must be determined using some form of interpolation. The appearance and quality of the output image is determined by the amount of error created by the chosen interpolation method. Note the differences in the line edges between the following two interpolated images.

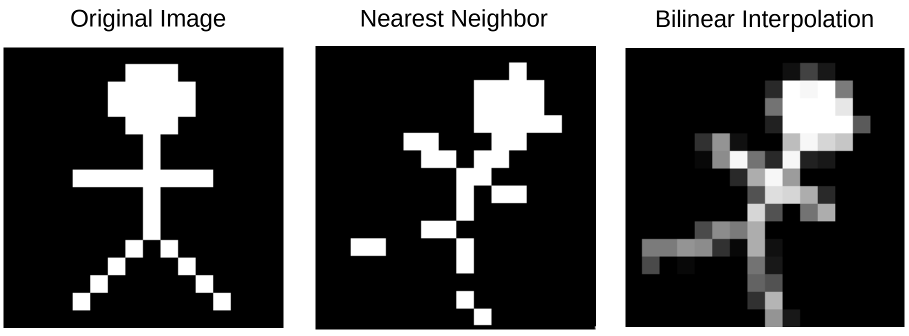


Figure 3-2: Simple Examples of Image Interpolation

There are a variety of possible interpolation methods available when using geometric transforms in IDL. Interpolation methods include:

Nearest-neighbor interpolation — Assigns the value of the nearest pixel to the pixel in the output visualization. This is the fastest interpolation method but the resulting image may contain jagged edges.

Linear interpolation — Surveys the 2 closest pixels, drawing a line between them and designating a value along that line as the output pixel value.

Bilinear interpolation — Surveys the 4 closest pixels, creates a weighted average based on the nearness and brightness of the surveyed pixels and assigns that value to the pixel in the output image.

Use cubic convolution if a higher degree of accuracy is needed. However, with still images, the difference between images interpolated with bilinear and cubic convolution methods is usually undetectable.

Trilinear interpolation — Surveys the 8 nearest pixels occurring along the x, y, and z dimensions, creates a weighted average based on the nearness and brightness of the surveyed pixels and assigns that value to the pixel in the output image.

Cubic Convolution interpolation — Approximates a sinc interpolation by using cubic polynomial waveforms instead of linear waveforms when resampling a pixel. With a one-dimension source, this method surveys 4 neighboring pixels. With a two-dimension source, the method surveys 16 pixels. Interpolation of three-dimension sources is not supported. This interpolation method results in the least amount of error, thus preserving the highest amount of fine detail in the output image. However, cubic interpolation requires more processing time.

Note

The *IDL Reference Guide* details the interpolation options available for each geometric transformation function.

Polygon Shading Method

The shading applied to each polygon, defined by its four surrounding elevations, can be either constant over the entire cell or interpolated. Constant shading takes less time because only one shading value needs to be computed for the entire polygon. Interpolated shading gives smoother results. The Gouraud method of interpolation is used: the shade values are computed at each elevation point, coinciding with each polygon vertex. The shading is then interpolated along each edge, finally, between edges along each vertical scan line.

Light-source shading is computed using a combination of depth cueing, ambient light, and diffuse reflection, adapted from Foley and Van Dam, Chapter 19 (Foley, J.D., and A. Van Dam (1982), *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Co.):

$$I = I_a + dI_p(\boldsymbol{L} \bullet \boldsymbol{N})$$

where

I_a	Term due to ambient light. All visible objects have at least this intensity, which is approximately 20 percent of the maximum intensity.
$I_p(\boldsymbol{L} \bullet \boldsymbol{N})$	Term due to diffuse reflection. The reflected light is proportional to the cosine of the angle between the surface normal vector ***N*** and the vector pointing to the light source, ***L***. I_p is approximately 0.9.
d	Term for depth cueing, causing surfaces further away from the observer to appear dimmer. The normalized depth is d=(z+2)/3, ranging from zero for the most distant point to one for the closest.

In Direct Graphics, the SET_SHADING method modifies the light source shading parameters. In Object Graphics similar OpenGL functionality is available through the SHADING property of objects such as IDLgrPolygon, IDLgrPolyline, IDLSurface and IDLgrContour.

Color Systems

Color can play a critical role in the display and perception of digital imagery. This section provides a basic overview of color systems, display devices, image types, and the interaction of these elements within IDL. The remainder of the chapter builds upon these fundamental concepts by describing how to load and modify color tables, convert between image types, utilize color tables to highlight features, and apply color annotations to images.

Color Schemes

Color can be encoded using a number of different schemes. Many of these schemes utilize a color triple to represent a location within a three-dimensional color space. Examples of these systems include RGB (red, green, and blue), HSV (hue, saturation, and value), HLS (hue, lightness, and saturation), and CMY (cyan, magenta, and yellow). Algorithms exist to convert colors from one system to another.

Computer display devices typically rely on the RGB color system. In IDL, the RGB color space is represented as a three-dimensional Cartesian coordinate system, with the axes corresponding to the red, green, and blue contributions, respectively. Each axis ranges in value from 0 (no contribution) to 255 (full contribution). By design, this range from 0 to 255 maps nicely to the full range of a byte data type.

An individual color is encoded as a coordinate within this RGB space. Thus, a color consists of three elements: a red value, a green value, and a blue value.

The following figure shows that each displayable color corresponds to a location within a three-dimensional color cube. The origin, (0, 0, 0), where each color coordinate is 0, is black. The point at (255, 255, 255) is white, representing an additive mixture of the full intensity of each of the three colors. Points along the main diagonal - where intensities of each of the three primary colors are equal - are shades of gray. The color yellow is represented by the coordinate (255, 255, 0), or a mixture of 100% red, plus 100% green, and no blue.

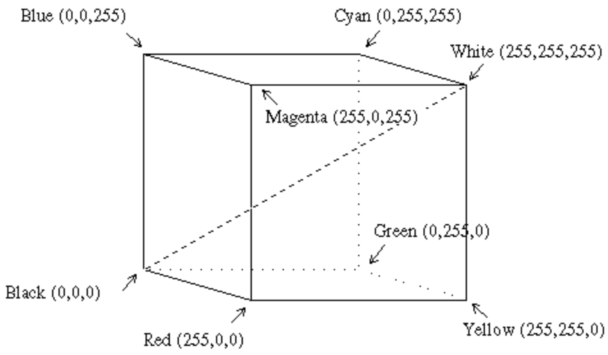


Figure 3-3: RGB Color Cube (Note: grays are on the main diagonal.)

Typically, digital display devices represent each component of an RGB color coordinate as an *n*-bit integer in the range of 0 to $2^n - 1$. Each displayable color is an RGB coordinate triple of *n*-bit numbers yielding a palette containing 2^{3n} total colors. Therefore, for 8-bit colors, each color coordinate can range from 0 to 255, and the total palette contains 2^{24} or 16,777,216 colors.

A display with an *m*-bit pixel can represent 2^m colors simultaneously, given enough pixels. In the case of 8-bit colors, 24-bit pixels are required to represent all colors. The more common case is a display with 8 bits per pixel which allows the display of $2^8 = 256$ colors selected from the much larger palette.

If there are not enough bits in a pixel to represent all colors, $m < 2^{3n}$, a color translation table is used to associate the value of a pixel with a color triple. This table is an array of color triples with an element for each possible pixel value. Given 8-bit pixels, a color table containing $2^8 = 256$ elements is required. The color table element with an index of *i* specifies the color for pixels with a value of *i*.

To summarize, given a display with an *n*-bit color representation and an *m*-bit pixel, the color translation table, *C*, is a 2^m long array of RGB triples:

$$C_i = \{r_i, g_i, b_i\}, \quad 0 \le i < 2^m$$

$$0 \le r_i, g_i, b_i < 2^n$$

Objects containing a value, or color index, of *i* are displayed with a color of C_i.

See “Color Table Manipulation” (*IDL Quick Reference*) for a list of color-related routines including those that covert RGB color triples to other color schemes.

Converting to Other Color Systems

IDL defaults to the RGB color system, but if you are more accustomed to other color systems, IDL is not restricted to working with only the RGB color system. You can also use either the HSV (hue, saturation, and value) system or the HLS (hue, lightness, and saturation) system. The HSV or HLS system can be specified by setting the appropriate keyword (for example /HSV or /HLS) when using IDL color routines.

IDL also contains routines to create color tables based on these color systems. The HSV routine creates a color table based on the Hue, Saturation, and Value (HSV) color system. The HLS routine creates a color table based on the Hue, Lightness, Saturation (HLS) color system. You can also convert values of a color from any of these systems to another with the COLOR_CONVERT routine. See `COLOR_QUAN` in the *IDL Reference Guide* for more information.

Display Device Color Schemes

Most modern computer monitors use one of two basic schemes for displaying color at each pixel:

- **Indexed -** A color is specified using an index into a hardware color lookup table (or *palette*). Each entry of the color lookup table corresponds to an individual color, and consists of a red value, a green value, and a blue value. The size of the lookup table depends upon the hardware.
- **RGB -** A color is specified using an RGB triple: [red, green, blue]. The number of bits used to represent each of the red, green, and blue components depends upon the hardware.

The description of how color is to be interpreted on a given display device is referred to as a *visual*. Each visual typically has a name that indicates how color is to be represented. Two very common visual names are PseudoColor (which uses an indexed color scheme) and TrueColor (which uses an RGB color scheme).

A visual also has a *depth* associated with it that describes how many bits are used to represent a given color. Common bit depths include 8-bit (for PseudoColor visuals) and 16- or 24-bit (for TrueColor visuals). An *n*-bit visual is capable of displaying 2^n total colors. Thus, an 8-bit PseudoColor visual can display 2^8 or 256 colors. A 24-bit TrueColor visual can display 2^{24} or 16,777,216 colors.

PseudoColor visuals rely heavily upon the display device’s hardware color table for image display. If the color table is modified, all images being displayed using that color table will automatically update to reflect the change.

TrueColor visuals do not typically use a color table. The red, green, and blue components are provided directly.

Note

You can display TrueColor images on pseudo-color displays by using the COLOR_QUAN function. This function creates a pseudo-color palette for displaying the TrueColor image and then maps the TrueColor image to the new palette. See COLOR_QUAN in the *IDL Reference Guide* for more information.

Setting a Visual on UNIX Platforms

On UNIX platforms, an application (such as IDL) may choose from among the set of X visuals that are supported for the current display. Each visual is either grayscale or color. Its corresponding color table may be either fixed (read-only), or it may be changeable from within IDL (read-write). The color interpretation scheme is either indexed or RGB. The following table shows the supported visuals for a given display, which may include any combination:

Visual	Description
StaticGray	grayscale, read-only, indexed
GrayScale	grayscale, read-write, indexed
StaticColor	color, read-only, indexed
PseudoColor	color, read-write, indexed
TrueColor	color, read-only, RGB
DirectColor	color, read-write, RGB

Table 3-2: Visuals Supported in IDL on UNIX Platforms

The most common of these is PseudoColor and TrueColor. Refer to the section "Colors and IDL Graphic Systems" on page 69 to learn more about how IDL selects a visual for image display.

To get the list of supported X visual classes on a given system, type the following command at the UNIX command line:

```
xdpyinfo
```

Setting a Visual on Windows Platforms

On Windows platforms, the visual is selected via the system Control Panel. To open the Control Panel, select the **Settings** → **Control Panel** item from the **Start** menu. Click on the **Display** and then select the **Settings** tab. Alter the **Color quality** setting to modify the visual before starting an IDL session. The following table shows three visuals are supported (for the particular display configuration used in this example):

Visual	Equivalence to UNIX Visuals
256 Colors	8-bit PseudoColor
High Color (16 bit)	16-bit TrueColor
True Color (32 bit)	32-bit TrueColor

Table 3-3: Visuals Supported in IDL on Windows Platforms

Colors and IDL Graphic Systems

IDL supports two graphics systems: Object Graphics and Direct Graphics. This section provides detailed descriptions of how color is represented and interpreted in the Direct Graphics system.

Using Color in Object Graphics

For complete details regarding color and Object Graphics, see "Color in Object Graphics" (Chapter 2, *Object Programming*).

Using Color in Direct Graphics

More information on the following topics is available in "X Windows Visuals" (Appendix A, *IDL Reference Guide*).

Visuals on UNIX Platforms

When IDL creates its first Direct Graphics window, it must select a visual to be associated with that window. By default, IDL selects an X Visual Class by requesting (in order) from the following table until a supported visual is found, but a specific visual can be explicitly requested at the beginning of an IDL session by setting the appropriate keyword to the DEVICE procedure:

Order	Visual	Depth	Related Keyword
First	TrueColor	24-bit (then 16-bit, then 15-bit)	TRUE_COLOR
Second	PseudoColor	8-bit, then 4-bit	PSEUDO_COLOR
Third	DirectColor	24-bit	DIRECT_COLOR
Fourth	StaticColor	8-bit, then 4-bit	STATIC_COLOR
Fifth	GrayScale	any depth	GRAY_SCALE
Sixth	StaticGray	any depth	STATIC_GRAY

Table 3-4: Order of Visuals and their Related DEVICE Keywords

To request an 8-bit PseudoColor visual, the syntax would be:

```
DEVICE, PSEUDO_COLOR=8
```

Another approach to setting the visual information is to include the `idl.gr_visual` and `idl.gr_depth` resources in your `.Xdefaults` file.

A visual is selected once per IDL session (when the first graphic window is created). Once selected, the same visual will be used for all Direct Graphics windows in that IDL session.

Private versus Shared Colormaps

On UNIX platforms, when a window manager is started, it creates a default colormap that can be shared among applications using the display. This is called the shared colormap.

A given application may request to use its own colormap that is not shared with other applications. This is called a private colormap.

IDL attempts, whenever possible, to get color table entries in the shared colormap. If enough colors are not available in the shared colormap, a private colormap is used. If an X Visual class and depth are specified and they do not match the default visual of the screen (see `xdpyinfo`), a private colormap is used.

If a private colormap is used, then colormap flashing may occur when an IDL window is made current (in which case, the colors of other applications on the desktop may no longer appear as you would expect), or when an application using the shared colormap is made current (in which case, the colors within the IDL graphics window may no longer appear as you would expect). This flashing behavior is to be expected. By design, the IDL graphics window has been assigned a dedicated color table so that the full range of requested colors can be utilized for image display.

Visuals on Windows Platforms

On Windows platforms, the visual that IDL uses is dependent upon the system setting. For more information, "Setting a Visual on Windows Platforms" on page 68.

IDL Color Table

IDL maintains a single current color table for Direct Graphics. Refer to the sections "Loading a Default Color Table" on page 78 and "Modifying and Converting Color Tables" on page 79. IDL provides 41 pre-defined color tables.

Foreground Color

In IDL Direct Graphics, colors used for drawing graphic primitives (such as lines, text annotations, etc.) are represented in one of two ways:

- Indexed - each color is an index into the current IDL color table

- RGB - each color is a long integer that contains the red value in the first eight bits, the green value in the next eight bits, and the blue value in the next eight bits. In other words, a color can be represented using the following equation:

  ```
  color = red + 256*green + (256^2)*blue
  ```

The RGB form is only supported on TrueColor display devices.

The DECOMPOSED keyword to the DEVICE procedure is used to notify IDL whether color is to be interpreted as an index or as a composite RGB value. IDL then maps any requested color to an encoding that is appropriate for the current display device.

The foreground color (used for drawing) can be set by assigning a color value to the !P.COLOR system variable field (or by setting the COLOR keyword on the individual graphic routine).

If a color value is to be interpreted as an index, then inform IDL by setting the DECOMPOSED keyword of the DEVICE routine to 0:

```
DEVICE, DECOMPOSED = 0
```

The foreground color can then be specified by setting !P.COLOR to an index into the IDL color table. For example, if the foreground color is to be set to the RGB value stored at entry 25 in the IDL color table, then use the following IDL command:

```
!P.COLOR = 25
```

If a color value is to be interpreted as a composite RGB value, then inform IDL by setting the DECOMPOSED keyword of the DEVICE routine to 1:

```
DEVICE, DECOMPOSED = 1
```

The foreground color can then be specified by setting !P.COLOR to a composite RGB value. For example, if the foreground color is to be set to the color yellow, [255,255,0], then use the following IDL command:

```
!P.COLOR = 255 + (256*255)
```

Image Colors

Color for image data is handled in a fashion similar to other graphic primitives, except that some special cases apply based upon the organization of the image data and the visual of the current display device.

If the image is organized as a:

- two-dimensional array -
  - If the display device is PseudoColor, then each pixel is interpreted as an index into the IDL color table

    - If the display device is TrueColor and if the DECOMPOSED keyword for the DEVICE procedure is set to 0, then each pixel value is interpreted as an index into the IDL color table (thereby emulating a PseudoColor display device).
    - If the display device is TrueColor and if the DECOMPOSED keyword for the DEVICE procedure is set to 1, then each pixel value is interpreted as the value to be copied to each of the red, green, and blue components of the RGB color.
- RGB array - (Supported only for TrueColor display devices)
    - Each pixel is interpreted as an RGB color composed of the three elements in the extra color dimension of the array.

To display an RGB image on a PseudoColor device, use the COLOR_QUAN routine to convert it to an indexed form. Refer to the section "Converting Between Image Types" on page 77.

The TV command can be used to display the image in IDL. For RGB images, the TRUE keyword can be used to indicate which form of interleaving is used.

Indexed and RGB Image Organization

IDL can display four types of images: binary, grayscale, indexed, and RGB. How an image is displayed depends upon its type. Binary images have only two values, zero and one. Grayscale images represent intensities and use a normal grayscale color table. Indexed images use an associated color table. RGB images contain their own color information in layers known as bands or channels. Any of these images can be displayed with iImage, Object Graphics, or Direct Graphics.

An image consists of a two-dimensional array of pixels. The value of each pixel represents the intensity and/or color of that position in the scene. Images of this form are known as sampled or raster images, because they consist of a discrete grid of samples. Such images come from many different sources and are a common form of representing scientific and medical data.

Numerous standards have been developed over the years to describe how an image can be stored within a file. However, once the image is loaded into memory, it typically takes one of two forms: indexed or RGB. An indexed image is a two-dimensional array, and is usually stored as byte data. A two-dimensional array of a different data type can be made into an indexed image by scaling it to the range from 0 to 255 using the BYTSCL function. See the BYTSCL description *in the IDL Reference Guide* for more information.

Image Orientation

The screen coordinate system for image displays puts the origin, (0, 0), at the lower-left corner of the device. The upper-right corner has the coordinate (*xsize*–1, *ysize*–1), where *xsize* and *ysize* are the dimensions of the visible area of the display. The descriptions of the image display routines that follow assume a display size of 512 x 512, although other sizes may be used.

The system variable !ORDER controls the order in which the image is written to the screen. Images are normally output with the first row at the bottom, i.e., in bottom-to-top order, unless !ORDER is 1, in which case images are written on the screen from top to bottom. The ORDER keyword also can be specified with TV and TVSCL. It works in the same manner as !ORDER except that its effect only lasts for the duration of the single call—the default reverts to that specified by !ORDER.

An image can be displayed with any of the eight possible combinations of axis reversal and transposition by combining the display procedures with the ROTATE function.

Indexed Images

An indexed image does not explicitly contain any color information. Its pixel values represent indices into a color Look-Up Table (LUT). Colors are applied by using these indices to look up the corresponding RGB triplet in the LUT. In some cases, the pixel values of an indexed image reflect the relative intensity of each pixel. In other cases, each pixel value is simply an index, in which case the image is usually intended to be associated with a specific LUT. In this case, the LUT is typically stored with the image when it is saved to a file. For information on the LUTs provided with IDL, see "Loading a Default Color Table" on page 78.

RGB Image Interleaving

An RGB (red, green, blue) image is a three-dimensional byte array that explicitly stores a color value for each pixel. RGB image arrays are made up of width, height, and three channels of color information. Scanned photographs are commonly stored as RGB images. The color information is stored in three sections of a third dimension of the image. These sections are known as color channels, color bands, or color layers. One channel represents the amount of red in the image (the red channel), one channel represents the amount of green in the image (the green channel), and one channel represents the amount of blue in the image (the blue channel).

Color interleaving is a term used to describe which of the dimensions of an RGB image contain the three color channel values. Three types of color interleaving are supported by IDL. In Object Graphics, an RGB image is contained within an image object where the INTERLEAVE property dictates the arrangement of the channels within the image file.

- Pixel interleaving (3, w, h) — the color information is contained in the first dimension, INTERLEAVE is set to 0.
- Line interleaving (w, 3, h) — the color information is contained in the second dimension, INTERLEAVE is set to 1.
- Planar interleaving (w, h, 3) — the color information is contained in the third dimension, INTERLEAVE is set to 2. This is also known as, image interleaving.

Note

In Direct Graphics, set the TRUE keyword of TV or TVSCL to match the interleaving of the image.

Determining RGB Image Interleaving

You can determine if an image file contains an RGB image by querying the file. The CHANNELS tag of the resulting query structure will equal 3 if the file's image is RGB. The query does not determine which interleaving is used in the image, but the array returned in DIMENSIONS tag of the query structure can be used to determine the type of interleaving.

The following example queries and imports a pixel-interleaved RGB image from the `rose.jpg` image file. This RGB image is a close-up photograph of a red rose. It is pixel interleaved. Complete the following steps for a detailed description of the process.

Example Code

See `displayrgbimage_object.pro` in the `examples/doc/image` subdirectory of the IDL installation directory for code that duplicates this example.

1. Determine the path to the `rose.jpg` file:

```
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Use QUERY_IMAGE to query the file to determine image parameters:

```
queryStatus = QUERY_IMAGE(file, imageInfo)
```

3. Output the results of the file query:

```
PRINT, 'Query Status = ', queryStatus
HELP, imageInfo, /STRUCTURE
```

The following text appears in the Output Log:

```
Query Status =            1
** Structure <14055f0>, 7 tags, length=36, refs=1:
   CHANNELS       LONG      3
   DIMENSIONS     LONG      Array[2]
   HAS_PALETTE    INT       0
   IMAGE_INDEX    LONG      0
   NUM_IMAGES     LONG      1
   PIXEL_TYPE     INT       1
   TYPE           STRING    'JPEG'
```

The CHANNELS tag has a value of 3. Thus, the image is an RGB image.

4. Set the image size parameter from the query information:

```
imageSize = imageInfo.dimensions
```

The type of interleaving can be determined from the image size parameter and actual size of each dimension of the image. To determine the size of each dimension, you must first import the image.

5. Use READ_IMAGE to import the image from the file:

```
image = READ_IMAGE(file)
```

6. Determine the size of each dimension within the image:

```
imageDims = SIZE(image, /DIMENSIONS)
```

7. Determine the type of interleaving by comparing the dimension sizes to the image size parameter from the file query:

```
interleaving = WHERE((imageDims NE imageSize[0]) AND $
   (imageDims NE imageSize[1]))
```

8. Output the results of the interleaving computation:

```
PRINT, 'Type of Interleaving = ', interleaving
```

The following text appears in the Output Log:

```
Type of Interleaving = 0
```

The image is pixel interleaved. If the resulting value was 1, the image would have been line interleaved. If the resulting value was 2, the image would have been planar interleaved.

9. Initialize the display objects:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'An RGB Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')
```

10. Initialize the image object:

```
oImage = OBJ_NEW('IDLgrImage', image, $
   INTERLEAVE = interleaving[0])
```

11. Add the image object to the model, which is added to the view, then display the view in the window:

```
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView
```

The following figure shows the resulting RGB image display.

Figure 3-4: RGB Image in Object Graphics

12. Clean up the object references. When working with objects always remember to clean up any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object.

    ```
    OBJ_DESTROY, oView
    ```

Converting Between Image Types

Sometimes an image type must be converted from indexed to RGB, RGB to grayscale, or RGB to indexed. For example, an image may be imported into IDL as an indexed image (from a PNG file for example) but it may need to be exported as an RGB image (to a JPEG file for example). The opposite may also need to be done. See "Foreground Color" on page 70 for more information on grayscale, indexed, and RGB images.

See the following routines s in the *IDL Reference Guide* for examples:

- **RGB to grayscale** — REFORM extracts the individual channels of data from an RGB image so that it can be displayed as a grayscale image
- **RGB to indexed** — COLOR_QUAN decomposes the millions of possible colors in an RGB image into the 256 used by an indexed image
- **Indexed to RGB** — TVLCT extracts the indexed image color table information, which is then assigned to an RGB image

Loading a Default Color Table

Although you can define your own color tables, IDL provides 41 pre-defined color lookup tables (LUTs). Each color table contained within this routine is specified through an index value ranging from 0 to 40, shown in the following table.

Tip

If you are running IDL on a TrueColor display, set `DEVICE, DECOMPOSED = 0` before your first color table related routine is used within an IDL session or program. See "Foreground Color" on page 70 for more information.

Number	Name	Number	Name
0	Black & White Linear	21	Hue Sat Value 1
1	Blue/White Linear	22	Hue Sat Value 2
2	Green-Red-Blue-White	23	Purple-Red + Stripes
3	Red Temperature	24	Beach
4	Blue-Green-Red-Yellow	25	Mac Style
5	Standard Gamma-II	26	Eos A
6	Prism	27	Eos B
7	Red-Purple	28	Hardcandy
8	Green/White Linear	29	Nature
9	Green/White Exponential	30	Ocean
10	Green-Pink	31	Peppermint
11	Blue-Red	32	Plasma
12	16 Level	33	Blue-Red 2
13	Rainbow	34	Rainbow 2
14	Steps	35	Blue Waves

Table 3-5: Pre-defined Color Tables

Number	Name	Number	Name
15	Stern Special	36	Volcano
16	Haze	37	Waves
17	Blue-Pastel-Red	38	Rainbow18
18	Pastels	39	Rainbow + white
19	Hue Sat Lightness 1	40	Rainbow + black
20	Hue Sat Lightness 2		

Table 3-5: Pre-defined Color Tables (Continued)

You can load a default color table in an iImage display, an Object Graphics Display or a Direct Graphics display as follows:

- iImage — select the **Edit Palette** button on the image panel. See "Using the Image Panel" (Chapter 10, *iTool User's Guide*) for details.
- Object Graphics — use the LoactCT method of an IDLgrPalette object to define the color table (see "IDLgrPalette::LoadCT" (*IDL Reference Guide*) for details). Associate the palette object with another object using the Palette property (for example, see the PALETTE property of the IDLgrImage object). Also see "Color in Object Graphics" (Chapter 2, *Object Programming*) for information on using color with indexed and RGB color models in Object Graphics.
- Direct Graphics — use the LOADCT routine or another color table related routine to set the color table. Also see "Using Color in Direct Graphics" on page 69.

Note

See "Color Table Manipulation" (*IDL Quick Reference*) for a list of related routines.

Modifying and Converting Color Tables

IDL contains two graphical user interface (GUI) utilities for modifying a color table, XLOADCT and XPALETTE (. The MODIFYCT routine lets you create or modify and store a new color table. See the following topics in the *IDL Reference Guide* for examples:

- XLOADCT — allows you to preview and select among pre-defined color tables
- XPALETTE — allows you to preview and adjust pre-defined color tables
- MODIFYCT — shows how to add modified color tables to IDL's list of pre-defined color tables.

These examples are based on the default RGB (red, green, and blue) color system. IDL also contains routines that allow you to use other color systems including hue, saturation, and value (HSV) and hue, lightness, and saturation (HLS). These routines and color systems are explained in "Converting to Other Color Systems" on page 66.

Highlighting Features with a Color Table

For indexed images, custom color tables can be derived to highlight specific features. Color tables are usually designed to vary within certain ranges to show dramatic changes within an image. Some color tables are designed to highlight features with drastic color change in adjacent ranges (for example setting 0 through 20 to black and setting 21 through 40 to white).

Note

Color tables are associated with indexed images. RGB images already contain their own color information. If you want to derive a color table for an RGB image, you should convert it to an indexed image with the COLOR_QUAN routine. You should also set COLOR_QUAN's CUBE keyword to 6 to insure the resulting indexed image is an intensity representation of the original RGB image. See `COLOR_QUAN` in the *IDL Reference Guide* for more information

See the following topics in the *IDL Reference Guide* for examples:

- IDLgrPalette provides an example that creates, defines and applies a palette object to an image
- TVLCT creates, defines and applies a color table in a Direct Graphics display
- H_EQ_CT applies histogram equalization to a color table to reveal previously indistinguishable feature

Multi-Monitor Configurations

IDL allows you to position windows on multiple monitors attached to the same computer. Such multi-monitor configurations may appear to the user (and to you as an IDL programmer) as a single extended desktop consisting of multiple physical monitors, or as a series of individual desktops appearing on multiple physical monitors.

IDL's support for multi-monitor configurations includes the following:

- The IDLsysMonitorInfo object, which allows you to query the system for the current monitor configuration and to determine the screen geometry of the various monitors.
- Keyword support for extended (or multiple) desktops within routines that draw a window on the monitor screen. For example, the XOFFSET, YOFFSET, and DISPLAY_NAME keywords to the WIDGET_BASE function and WIDGET_CONTROL procedure allow you to position widget applications anywhere on any available monitor. Similarly, the LOCATION and DISPLAY_NAME properties of the IDLgrWindow object afford you the same control for object graphics windows.

It is important to note that support for multi-monitor configurations is quite different on Windows and UNIX systems, and that as a result IDL's support varies by platform. By understanding how multi-monitor configurations are supported on each platform, you can create cross-platform IDL applications that will take advantage of multiple monitors when they are present. See the following sections for platform-specific details.

- "Windows Multi-Monitor Configurations" on page 83
- "UNIX Multi-Monitor Configurations" on page 87

See "Example: Multi-Monitor Window Positioning" on page 89 for example code that uses the IDL's multi-monitor support.

Multi-Monitor Terminology

In this discussion of IDL's multi-monitor support, the following terms are used with the meanings listed below.

Desktop — An onscreen user work area. Multiple desktops are generally managed either by the operating system itself or by a desktop management system and are dependant on the physical monitor configuration — that is, you can have multiple desktops on a single monitor.

Display — On UNIX systems, the word *Display* describes the connection between an X client and an X server. Do not confuse this with *monitor*.

Extended Desktop — A term for an onscreen user work area that may span multiple monitors. It is often used to describe the minimum bounding box that encloses the user work area defined by each monitor in the system. There may be “holes” in an extended desktop if two monitors with different display resolutions are used. Extended desktops are characterized by their ability to drag windows between monitors on the desktop.

Monitor — A physical display device such as a CRT or LCD.

Primary Monitor — In an extended desktop system, the primary monitor is the monitor that contains the origin (0,0). If the desktop is not extended, then the primary monitor is the one that is considered “default” by the graphics system.

Screen — On UNIX systems, the word *Screen* describes one of a display’s drawing surfaces. A single X server can control more than one *Screen*, but is generally operated or controlled by a single user with a single keyboard and pointing device.

Secondary Monitor — In an extended desktop system, a secondary monitor is any monitor that is not the primary monitor. If the desktop is not extended, then a secondary monitor is the one that is *not* considered “default” by the graphics system.

Virtual Desktop — A desktop configured so that it is larger than the monitor used to display it. The user can “pan” the desktop around to cause the desired parts of it to be visible on the monitor.

X Server — A program that runs on the machine to which the graphics adapter is attached. It owns the graphics adapter and is responsible for drawing on it.

X Client — A program that connects to an X server, sending commands to the X server to draw on the display device. The X client is typically the application and may or may not be executing on the same machine as the X server.

X Multi-Screen — The “core” method for an X server to handle more than one monitor. Each monitor is assigned a *Screen*; the user can move the pointing device from one monitor to another, but cannot drag windows between monitors. Each *Screen* is addressed by the final digit in the X Display name (e.g., the 1 in `ajax:0.1`).

XINERAMA — An X11 extension that allows a single X11 screen to be displayed across multiple monitors. This allows an application to open windows on any monitor using the same *Display*/*Screen* connection. This is an example of an extended desktop implementation for UNIX systems and is essentially a way to emulate the extended desktop that Windows presents to the user.

Windows Multi-Monitor Configurations

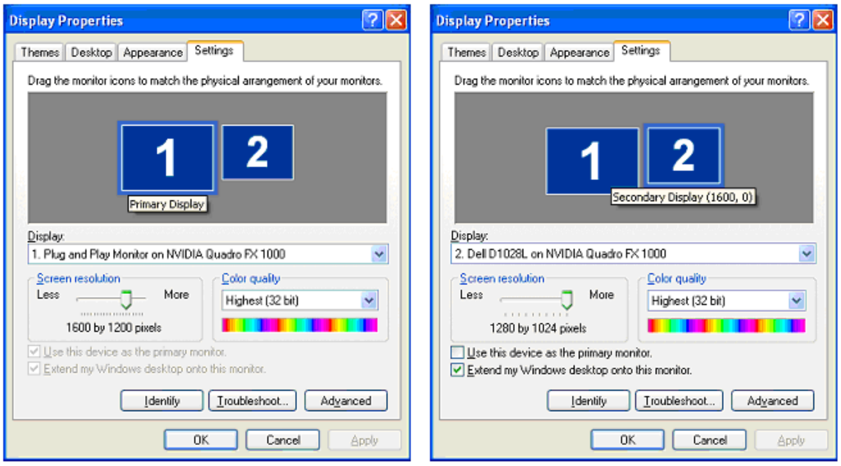
A multi-monitor configuration on a Windows system is always presented as an extended desktop, with the work area spanning the configured monitors. You can drag windows from one monitor to the other, or they can span monitors.

The extended desktop configuration works best when using a single graphics adapter with two video outputs. If you use multiple graphics adaptors, features such as 3D hardware video acceleration may only be available on one monitor.

To configure a multi-monitor configuration using the Windows Display applet; either:

- Right-click on the desktop and select **Properties**
- Select **Start** → **Settings** → **Control Panel** → **Display**

Figure 3-5 shows the **Display Properties** control panel for a common dual-monitor configuration. The left-hand image shows the primary display selected and identified as monitor 1. The right-hand image shows the secondary display selected and identified as monitor 2. The coordinates of the upper-left corner of the secondary display are shown in the tool-tip (“Secondary Display (1600, 0)”). Also, the **Extend my Windows desktop onto this monitor** checkbox is selected to extend the desktop onto the secondary monitor.



Drag the monitor icons to match the physical arrangement of your monitors.

Display:
1. Plug and Play Monitor on NVIDIA Quadro FX 1000

Screen resolution: 1600 by 1200 pixels

Color quality: Highest (32 bit)

☑ Use this device as the primary monitor.
☑ Extend my Windows desktop onto this monitor.

Drag the monitor icons to match the physical arrangement of your monitors.

Display:
2. Dell D1028L on NVIDIA Quadro FX 1000

Screen resolution: 1280 by 1024 pixels

Color quality: Highest (32 bit)

☐ Use this device as the primary monitor.
☑ Extend my Windows desktop onto this monitor.

Figure 3-5: Multi-monitor Configuration in Windows Display Properties

The extended desktop configured in Figure 3-5 appears as in Figure 3-6, with a dotted line showing where the two monitors meet in one desktop.

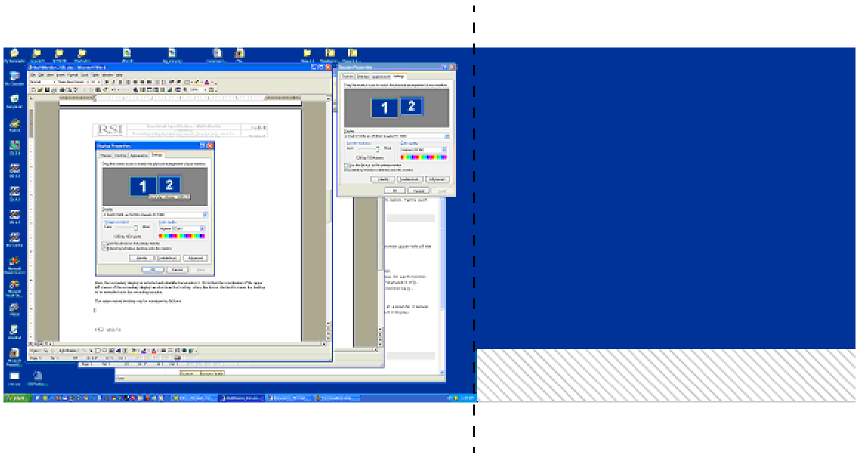
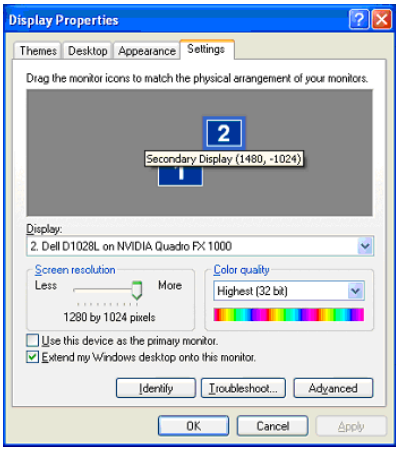


Figure 3-6: The Extended Desktop

In this example, there are no windows on the secondary monitor. The crosshatched area in the lower right exists because the monitor on the right has fewer pixel rows than the monitor on the left.

The **Display Properties** dialog allows you to change the location of the secondary monitor relative to the primary monitor. Note that pixel (0,0) is defined as being the upper left corner of the primary monitor. Figure 3-7 shows a configuration in which the secondary monitor is positioned “above” the primary monitor; the tooltip shows that the upper left corner of the secondary monitor is positioned 1480 pixels to the right of and -1024 pixels below pixel (0,0). Figure 3-8 shows the shape of the resulting extended desktop area.



Drag the monitor icons to match the physical arrangement of your monitors.

Display:
2. Dell D1028L on NVIDIA Quadro FX 1000

Screen resolution: 1280 by 1024 pixels

Color quality: Highest (32 bit)

☐ Use this device as the primary monitor.
☑ Extend my Windows desktop onto this monitor.

Figure 3-7: Moving the Location of the Second Monitor

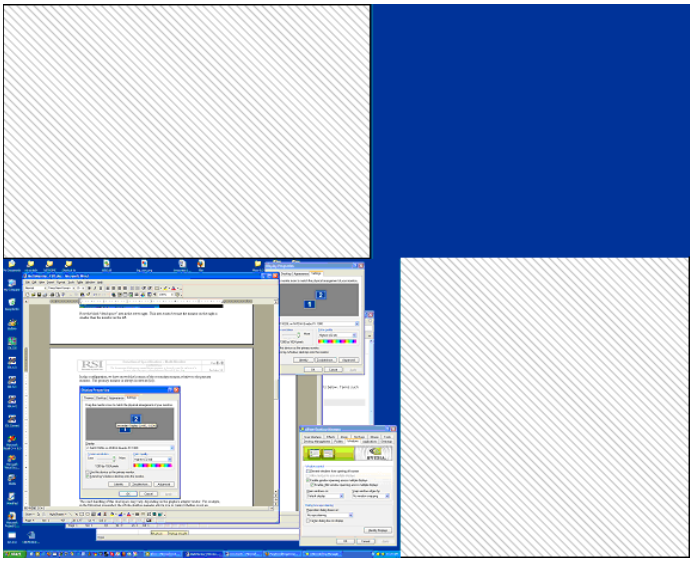


Figure 3-8: The Rearranged Desktop Configuration

There is now more “empty” space (represented by the crosshatched area). The handling of empty space depends on the graphics adapter vendor. For example, many desktop managers let you control whether or not an application can create a window in this empty space. (Remember that if you do create a window in empty space, there would be no way to drag the window back onto a visible portion of the desktop.) Many desktop managers also contain controls for opening windows and repositioning dialog boxes.

Warning

Third-party desktop managers may enforce their own positioning rules, overriding requests from other applications such as IDL. If you have trouble positioning windows on the screen using IDL, investigate whether your desktop manager’s control over other applications can be changed or relaxed.

UNIX Multi-Monitor Configurations

Because the UNIX platform encompasses multiple vendors, multi-monitor support can be more complex to configure. There are two primary multi-monitor solutions for UNIX platforms:

- Use the X Multi-Screen mechanism, wherein a distinct X11 *Screen* is displayed on each monitor to create multiple desktops. IDL supports this mechanism on all UNIX systems.
- Use the XINERAMA extension to create a single extended desktop. IDL 6.3 provides client support for the XINERAMA extension Macintosh OS X and several Linux distributions.

Note

Configure your UNIX multi-monitor systems using XINERAMA wherever possible. This gives you the most functionality and increases commonality with Windows.

Using X Multi-Screen

An X server running on a computer using multiple monitors can be configured so that a different *Screen* is assigned to each monitor. This is the traditional way for a UNIX system to support multiple monitors, and it is the only option available on IDL platforms for which there is no XINERAMA support.

In a multi-screen configuration, windows and dialogs cannot be dragged between windows interactively, and cannot span multiple monitors. Each monitor has a different display name and coordinate system with its own origin.

Using XINERAMA

The XINERAMA extension creates an extended desktop similar to that presented on Windows systems. Windows and dialogs can be dragged between windows interactively, and can span multiple monitors. All configured monitors share the same display name and have a common origin.

Stable XINERAMA support is only available on selected X Windows System releases. As of the IDL 6.3 release, IDL provides client support on Macintosh OS X and several Linux distributions. In addition, If the X server is running Macintosh OS X, Linux, or Solaris 10, IDL can treat multiple monitors as an extended desktop even though no information about individual monitor geometries is available.

UNIX systems that provide XINERAMA support are rarely configured to do so by default; consult your operating system documentation for configuration information. Some vendors supply configuration tools and desktop management controls to help use their systems. In addition, some X window managers are "XINERAMA-aware" and let you configure some multi-monitor-related behaviors.

Warning

Third-party desktop managers may enforce their own positioning rules, overriding requests from other applications such as IDL. If you have trouble positioning windows on the screen using IDL, investigate whether your desktop manager's control over other applications can be changed or relaxed.

XINERAMA Client/Server Interactions

When using networked UNIX systems, you are generally seated at an X workstation that is running an X server and some local programs such as command shells. You then log in remotely to another machine and execute X client programs (like IDL) with their DISPLAY environment variable pointing back to the X server you are using. The client program may be running on a machine that is of completely different architecture and capability than the machine running the X server. Table 3-6 shows the IDL X client's interactions with X servers on systems that do or do not support XINERAMA.

Client supports XINERAMA?	Server supports XINERAMA?	
	Yes	No
Yes — IDL running on Linux, OS X	IDL detects extended desktop with monitor information for each physical monitor.	IDL detects independent desktops with monitor information for each physical monitor.
No — IDL running on other UNIX platforms	IDL detects extended desktop with monitor information for single desktop spanning all monitors. Individual monitor information is not available.	IDL detects independent desktops with monitor information for each physical monitor

Table 3-6: Possible XINERAMA Client/Server Combinations

Example: Multi-Monitor Window Positioning

The IDL distribution contains example `.pro` code that illustrates how to use the IDLsysMonitorInfo object to position application windows on multiple monitors. With a little care, you can design the code to work on Windows, XINERAMA, and X Multi-Screen platforms and handle all monitor configurations.

The example code displays a simple splash screen in the middle of the primary monitor and opens a simple application GUI on the *n*th monitor in a system with *n* monitors.

Example Code

The application window positioning for multi-monitor example is included in the file `multimon_ex1.pro` in the `examples/doc/utilities` subdirectory of the IDL distribution.

Using Fonts in Graphic Displays

IDL uses three font systems for writing characters on the graphics device, whether that device be a display monitor or a printer: Hershey (vector) fonts, TrueType (outline) fonts, and device (hardware) fonts. Fonts are discussed in detail in Appendix H, "Fonts" (*IDL Reference Guide*).

Both TrueType and Vector fonts are displayed identically on all of the platforms that support IDL. This means that if your cross-platform application uses either the TrueType fonts supplied with IDL or the Vector fonts, there is no need for platform-dependent code.

In a widget application, specify a font using the FONT keyword. If you choose a device font, you may need to write platform-dependent code. See "Fonts Used in Widget Applications" (Chapter 9, *Application Programming*) for details.

To set the font in an Object Graphics display, create an IDLgrFont object and assign this object to a text object using the IDLgrText object FONT property. See "Font Objects" (Chapter 9, *Object Programming*) for more information.

Note

Within the IDLDE, you can specify what font is used in various areas (e.g., the Editor window or the Output Log window). See "Font Preferences" (Chapter 4, *IDL Interface*) for details.

Printing Graphics

Beginning with IDL version 5.0, IDL interacts with a system-level printer manager to allow printing of both IDL Direct Graphics and IDL Object Graphics. On Windows platforms, IDL uses the operating system's built-in printing facilities; on UNIX platforms, IDL uses the Xprinter print manager from Bristol Technology.

Use the DIALOG_PRINTERSETUP and DIALOG_PRINTJOB functions to configure your system printer and control individual print jobs from within IDL.

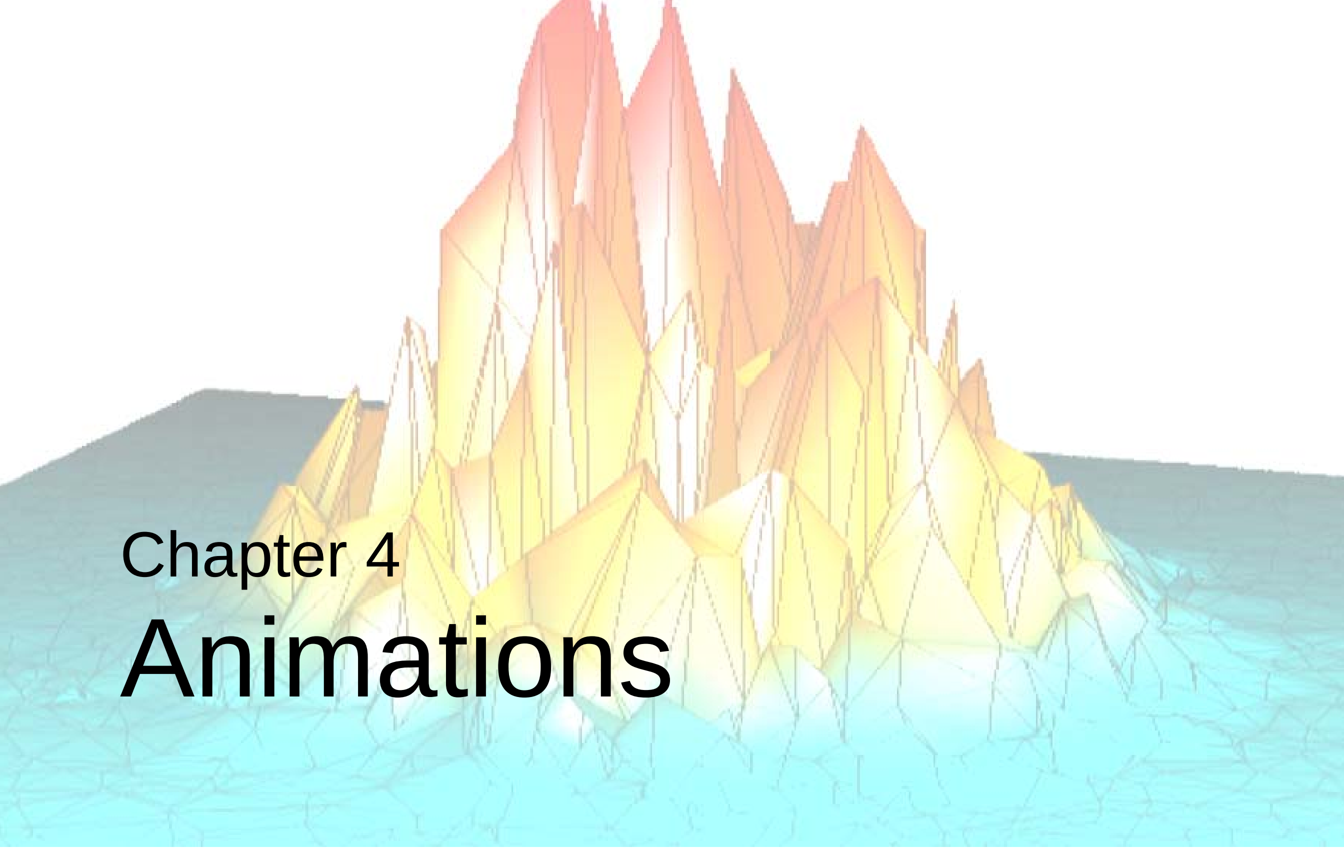
Printing IDL Direct Graphics

To print IDL Direct Graphics, you must first use the SET_PLOT procedure to make PRINTER your current device. Issue IDL commands as normal to create the graphics you wish to print, then use the CLOSE_DOCUMENT keyword to DEVICE to actually initiate the print job and print something from your printer. You can also create multiple pages before closing the document as well as being able to use tile graphics with the !P.MULTI system command.

See "Printing Graphics Output Files" (Appendix A, *IDL Reference Guide*) for details and examples.

Printing IDL Object Graphics

To print IDL Object Graphics, you must create a printer object to use as a destination for your Draw operations. You can also print multiple documents with the IDLgrPrinter object. See "Printer Objects" (Chapter 12, *Object Programming*)for information about printer objects and examples of their use. Also see "Bitmap and Vector Graphic Output" (Chapter 12, *Object Programming*) for information of when to output to bitmap or vector graphics based on picture content.



Chapter 4 Animations

This chapter describes how to create and play Motion JPEG2000 animations using the IDLffMJPEG2000 object. See the following topics for details:

Overview of Motion JPEG2000

Motion JPEG2000 is an extension of the still image JPEG2000 image format that is designed for storing animations. A Motion JPEG2000 file (MJ2) consists of a collection of frames. Each frame is an independent JPEG2000 image, and like JPEG2000 images, each frame may be made up of one or more components (bands or channels of data). The individual frame components may also be composed of tiles or contain regions.

The Motion JPEG2000 format offers several features that make it an excellent choice for data storage in scientific, security, and research arenas:

- Lossless compression option — the original image data can be retrieved from the file.
- Granular access — an animation can consist of individual components, tiles or regions in addition to entire frames.
- *Intra-frame* encoding — each frame is an independent entity and a true representation of the data at a single point in time. The older MPEG standard uses *inter-frame* encoding where interdependencies between the frames makes it impossible to extract a singular frame of data.

You can create and play Motion JPEG2000 (MJ2) files in IDL using the IDLffMJPEG2000 object. This chapter describes how to create and play your own MJ2 files. In brief, an IDLffMJPEG2000 object can open an MJ2 file (identified by a *Filename* argument) for playback or creation based on the value of the WRITE property. When you create (write) a file, you will use the IDLffMJPEG2000::SetData method to add frames, components or tiles of data to the file. When the animation is complete, call the IDLffMJPEG2000::Commit method to close the file. See "Creating a Motion JPEG2000 Animation" on page 96 for details.

Note

The same IDLffMJPEG2000 object cannot be used to both write and read an MJ2 file. You can write a file with one object (where WRITE=1), but you must create a separate object (where WRITE=0, the default) in order to read or play the new MJ2 file.

The IDLffMJPEG2000 object supports sequential and random playback. To create a sequential playback, you will use a group of methods to start the reading process, retrieve the frame, release the frame and stop the reading process. These methods are described in "Playing a Motion JPEG2000 Animation" on page 103. If you want to control the playback rate, you will need to include some sort of timer mechanism as described in "Controlling the Playback Rate" on page 106.

When creating and playing an MJ2 file, IDL uses an internal background processing thread to compress or decompress frames into a frame buffer. Depending upon the size and complexity of the frame, creation or playback may be delayed if frame compression or decompression takes longer than the associated method call. To avoid such a delay, modify the FRAME_BUFFER_LENGTH property as described in "High Speed MJ2 Reading and Writing" on page 108.

Sample Motion JPEG2000 Player and Writer

The IDL distribution includes a sample MJ2 player and an MJ2 writer as follows:

- The sample IDL Motion JPEG2000 Player can display RGB and monochrome MJ2 files. This example code, `mj2_player.pro`, and a sample image, `idl_mjpeg2000_example.mj2`, are located in the *`IDL_DIR`*`\examples\mjpeg2000` directory where *`IDL_DIR`* is the directory where you have installed IDL.
- The sample IDL Motion JPEG2000 Writer, `mj2_writer_rgb.pro`, creates an MJ2 animation. This example is located in the *`IDL_DIR`*`\examples\mjpeg2000` directory where *`IDL_DIR`* is the directory where you have installed IDL. Running the example creates a new MJ2 file, which is written to your application user directory, a subdirectory of your home directory.

Supported Platforms

The IDLffMJPEG2000 object is not supported on AIX or IRIX. See "Feature Support by Operating System" (Chapter 1, *Installation and Licensing Guide*) for details.

Creating a Motion JPEG2000 Animation

To create a Motion JPEG2000 file, create a new IDLffMJPEG2000 object and set the WRITE property equal to 1. During initialization, you must specify a filename, which is the path and location of the MJ2 file to be created.

Note

If you specify an existing MJ2 file as the *Filename* argument during initialization, and also set the WRITE keyword, the existing file will be overwritten without prompting and all existing data will be replaced with the new data. It is not possible to append data to an MJ2 file.

To create a file, you will need to use the IDLffMJPEG2000::SetData and IDLffMJPEG2000::Commit methods. The SetData method lets you add entire frames of data, or individual frame components or frame tiles to the MJ2 file. However, before the first call to SetData, there are several properties you may need to set.

Property	Brief Description
BIT_DEPTH	Specifies the bit depth of the data to be written to the file. If not set, the default value of 8 will specify byte data. **Note -** To write short or long integer data, you must set the BIT_DEPTH and SIGNED properties before calling SetData.
COMMENT	Specifies a descriptive comment for the file.
FRAME_BUFFER_LENGTH	Defaults to 3, the number of frame slots in the frame buffer. See "High Speed MJ2 Reading and Writing" on page 108 for information on how modifying this value can enable high-speed reading and writing of MJ2 files.
N_LAYERS	Defines the number of quality levels used to build the frame. If not set, the default value (1) is used.

Table 4-1: Properties that Must be Set Before Calling the SetData Method

Property	Brief Description
N_LEVELS	Defines the number of wavelet decompression levels. The default is 5 unless the PALETTE property is set, in which case the default is 0.
PALETTE	Set to a 3-by-*n* or an *n*-by-3 array of byte or integer values where *n* is the number of intensity values for the three (r, g, b) color channels.
REVERSIBLE	Set to 1 (lossless) to be able to retrieve the original data. The default is 0 (lossy) unless the PALETTE property has been set.
SIGNED	Set to 1 to write signed data. Otherwise, data will be written as unsigned (0, the default).

Table 4-1: Properties that Must be Set Before Calling the SetData Method

The following properties will be automatically set based on the first frame of data passed to SetData if not specified before the first call. If you are passing in a single frame component or tile component in each call to SetData, you need to set the related properties (N_COMPONENTS or TILE_DIMENSIONS) prior to the first call to SetData in order for the data to be written to the file correctly.

Property	Description
COLOR_SPACE	Defines the color space of the file. If the input data has 1 component, the default is monochrome; if it has 3 components, the default is RGB (unless the YCC property is set).
DIMENSIONS	Defaults to the *width*, *height* of the first frame of input data. The dimensions of each data array must match.
N_COMPONENTS	Defaults to the number of components in the first frame.
TILE_DIMENSIONS	Defaults to the DIMENSIONS of the frame if not set.

Table 4-2: Properties Set Based on SetData Input if Not Specified

Note

See "IDLffMJPEG2000 Properties" (*IDL Reference Guide*) for details.

Adding Data to MJ2 Animations

The source of the data for the MJ2 file can be existing data or incremental captures from data processing or data display. Regardless of the source of the data to be added to the MJ2 file, you will need to call the IDLffMJPEG2000::SetData method multiple times (minimally, once for each frame of the animation). Each SetData call adds the data to the frame buffer where it is compressed by a background processing thread. This processing thread is automatically started with the first SetData call. After all of the data has been added to the file, you must call the IDLffMJPEG2000::Commit method to stop the processing thread and close the file.

The first call to the IDLffMJPEG2000::SetData property is key. If you have not previously defined a number of object properties (noted in "Creating a Motion JPEG2000 Animation" on page 96), then the values are taken from the dimensions of the data that is passed in during the first SetData call. For example, if you pass in three arrays (*data1*, *data2* and *data3*) in the first SetData call, the COLOR_SPACE property will automatically be set to sRGB. If you are passing in three monochrome data arrays, this property would need to be set to sLUM prior to the first call to SetData to avoid unexpected results.

Note

It is possible to call SetData faster than the background processing thread can compress the data and write it to a file. If this is an issue, see "High Speed MJ2 Reading and Writing" on page 108 for additional file creation options.

When creating a new MJ2 file you can choose from the following options:

- "Animating Existing Data" on page 99 — add frames, components or tiles of data to the MJ2 file
- "Animating Screen Captures" on page 102 — add the contents of an object graphics animation to the MJ2 file
- "Animating Data Captures" on page 102—add newly created data to the MJ2 file

Note

The following examples use a simple WAIT statement mechanism for controlling the playback rate. In reality, you will likely use a more robust mechanism. See "Controlling the Playback Rate" on page 106 for options and information about a related example.

These examples, which are comparatively short and simple, use the GetData method instead of the group of methods described in "Sequential Motion JPEG2000 Playback" on page 103. Examples showing the use of the sequential playback methods are located in "Controlling the Playback Rate" on page 106 and the Examples section of "IDLffMJPEG2000::GetSequentialData" (*IDL Reference Guide*).

Animating Existing Data

The IDLffMJPEG2000 object stores entire frames of data as well as bands or channels of frame data (components) or frame tiles. The new MJ2 file can contain a series of images, components, or tiles as long as the dimensions and numbers of components are the same for each element. Examples of animating existing data include:

- "MJ2 Monochrome Frame Animation"
- "MJ2 Animation of an Image with a Palette" on page 100
- "MJ2 RGB Tile Animation" on page 101

The following examples write MJ2 files to your temporary directory. Use `PRINT, FILEPATH(' ', /TMP)` to display this location.

MJ2 Monochrome Frame Animation

The following simple example creates a short animation from a series of MRI frames of data contained in a binary file. An animation consisting of all available quality layers for a dozen frames is then displayed.

```
PRO mj2_frames_doc

; Read image data, which contains 57 frames.
nFrames = 57
head = READ_BINARY( FILEPATH('head.dat', $
  SUBDIRECTORY=['examples','data']), $
  DATA_DIMS=[80,100, 57])

; Create new MJ2 file in the temporary directory.
file = FILEPATH("mj2_frames_ex.mj2",/TMP)

; Create an IDLffMJPEG2000 object.
oMJ2write=OBJ_NEW('IDLffMJPEG2000', file, /WRITE, /REVERSIBLE, $
   N_LAYERS=10)

; Write the data of each frame into the MJ2 file.
FOR i=0, nFrames-1 DO BEGIN
```

```
      data = head[*,*,i]
      result = oMJ2write->SetData(data)
   ENDFOR

   ; Commit and close the IDLffMJPEG2000 object.
   return = oMJ2write->Commit(10000)
   OBJ_DESTROY, oMJ2write

   ; Create a new IDLffMJPEG2000 object to access MJ2 file.
   oMJ2read=OBJ_NEW("IDLffMJPEG2000", file)
   oMJ2read->GetProperty,N_FRAMES=nFrames, DIMENSIONS=dims

   ; Create a window and display simple animation.
   WINDOW, 0, XSIZE=2*dims[0], YSIZE=2*dims[1], TITLE="MJ2 Layers"

   ; Display all quality layers (j) of a dozen frames (i).
   FOR i=25, 36 DO BEGIN
      ; Return data and display magnified version. Pause
      ; between each frame for visibility. Unless a timer
      ; is used in conjunction with the FRAME_PERIOD and
      ; TIMESCALE properties, playback will occur as fast
      ; as the frames can be decompressed.
      FOR j=0, 10 DO BEGIN
         data = oMJ2read->GetData(i, MAX_LAYERS=j)
         TVSCL, CONGRID(data, 2*dims[0], 2*dims[1])
         WAIT, 0.1
      ENDFOR
   ENDFOR

   ; Cleanup.
   OBJ_DESTROY, oMJ2read

   End
```

This example is also available in the IDL distribution.

Example Code

This example, `mj2_frames_doc.pro`, is located in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `mj2_frames_doc` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT mj2_frames_doc.pro`.

MJ2 Animation of an Image with a Palette

The following example accesses the palette associated with a PNG file and assigns the values to the IDLffMJPEG2000 object PALETTE property. The image data is then modified in such a way that the resulting animation appears to be a shrinking view of the image. However, the shrunken image is padded to maintain the original image dimensions, which is a requirement of SetData. Each frame must have the same dimensions.

The following lines, abstracted from the entire example, show accessing the palette from the PNG file and assigning it to the new MJ2 file.

```
; Access image data and associated palette.
world = READ_PNG (FILEPATH ('avhrr.png', $
   SUBDIRECTORY = ['examples', 'data']), R, G, B)
;...
; Create an MJ2 file in the temporary directory. Assign the
; palette arrays to the PALETTE property.
file =FILEPATH("mj2_palette_ex.mj2", /TMP)
oMJ2write = OBJ_NEW('IDLffMJPEG2000', file, /WRITE, $
   PALETTE=[[R], [G], [B]])
```

See the following for the complete program.

Example Code

This example `mj2_palette_doc.pro`, is located in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `mj2_palette_doc` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT mj2_palette_doc.pro`.

MJ2 RGB Tile Animation

The following example creates a tiled, RGB JPEG2000 image from a 5,000 by 5,0000 pixel JPEG image. The JPEG2000 image tile data is then written to a Motion JPEG2000 image file. As shown in the following code, a smaller version of each tile is extracted from the MJ2 file and displayed sequentially in a window.

```
; Create object to read new MJ2 file. Set PERSISTENT to access
; tiled data. Set DISCARD_LEVELS to display smaller versions of
; the tiles.
oMJ2read = OBJ_NEW('IDLffMJPEG2000', file, /PERSISTENT)
oMJ2read->GetProperty, N_TILES=nTiles, TILE_DIMENSIONS=tileDims
WINDOW, 0, XSIZE=625, YSIZE=625
For j=0, nTiles-1 DO BEGIN
   data = oMJ2read->GetData(0, DISCARD_LEVELS=3, $
      TILE_INDEX=j, /RGB)
   TVSCL, data, j, TRUE=1
   WAIT, 0.3
ENDFOR
```

See the following for the complete program. A noticeable amount of time will be required the first time you run the example as several large files must be created.

Example Code

This example `mj2_tile_doc.pro`, is located in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `mj2_tile_doc` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT mj2_tile_doc.pro`.

Animating Screen Captures

You can capture the visible contents of an IDLgrWindow using the IDLgrWindow IMAGE_DATA property. The captured data can then be passed to the MJ2 file via the IDLffMJPEG2000::SetData method. This method of MJ2 creation is useful for recording an existing animation. For information on creating animations in an object graphics window see Chapter 10, "Animating Objects" (*Object Programming*). For an example that creates an MJ2 file using this method, see "Sample Motion JPEG2000 Player and Writer" on page 95, which describes the example, `mj2_writer_rgb.pro`, located in the `*IDL_DIR*\examples\mjpeg2000` directory.

A timer mechanism can be used to control the rate of the animation and the rate at which data is captured and written to an MJ2 file. See "Timer Mechanisms" on page 107 for more information.

Animating Data Captures

In addition to adding existing data to an MJ2 file, you can also add incremental data captures - snapshots of data at specified intervals. Data captured at any point during program execution can be added as long as each element passed to SetData has the same dimensions. The following example captures the incremental application of a thinning operator to an image, creating an animation that shows the changes to the original data.

Example Code

This example `mj2_morphthin_doc.pro`, is located in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `m2_morphthin_doc` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT m2_morphthin_doc.pro`.

Playing a Motion JPEG2000 Animation

You can use the IDLffMJPEG2000 object to access frames sequentially or randomly from a Motion JPEG2000 file (MJ2). Sequential access plays an animation, which can consist of entire frames, or can consist of frame components, tiles or regions, and uses a background processing thread. Random access plays selected frames, which can also consist of entire frames, or frame components, tiles or regions, without the use of a background processing thread. See the following sections for details:

- "Sequential Motion JPEG2000 Playback" on page 103
- "Random Motion JPEG2000 Playback" on page 104

Regardless of the type of playback, it is important to understand that unless you implement a timer mechanism to control playback, the default rate will be as fast as the frames can be decompressed. Options for timer mechanisms include widget timer and the more robust IDLitWindow timer mechanism.

Warning

Avoid using the WAIT procedure to control the sequential playback rate. On UNIX platforms there is an internal conflict between the background processing thread and the WAIT procedure. To avoid cross-platform compatibility issues, always use a widget timer or IDLitWindow timer mechanism to control the sequential playback rate.

The timer mechanism will typically use the FRAME_PERIOD and TIMESCALE properties to control the rate. See "Controlling the Playback Rate" on page 106 for more information.

Note

If you find the rate at which the frames can be decompressed is slower than the desired playback speed, see "High Speed Sequential Playback" on page 108 for an optional playback method.

Sequential Motion JPEG2000 Playback

To playback a large series of MJ2 frames, components, tiles or regions sequentially, your program will need to include the following methods and elements:

- IDLffMJPEG2000::StartSequentialReading—start the background decompression thread. You can indicate what data to display (the entire frame, or a component, tile, or region of the frame) as well as the resolution (level) of data. You can also specify the start and stop frames for the sequential playback.

- Timer—start a widget timer or IDLitWindow timer mechanism to play back frames at the desired rate. Within the timer event, call the following methods:
    - IDLffMJPEG2000::GetSequentialData—points at the data being retrieved from the frame buffer. This is *not* a copy of the data.
    - IDLffMJPEG2000::ReleaseSequentialData—releases the data from the frame buffer.

Note

You should always include a timer mechanism to control the playback rate. Without a timer, the playback rate will be equal to the rate at which the frames can be decompressed. See "Controlling the Playback Rate" on page 106 for details and an example.

- IDLffMJPEG2000::StopSequentialReading— releases the decompressed frames from the frame buffer memory and stops the background processing thread, (if it is still running). Call this method when the sequential playback is complete.

When playback ends, turn off the timer mechanism to stop the animation.

Examples showing the use of the sequential playback methods are located in "Controlling the Playback Rate" on page 106 and the Examples section of "IDLffMJPEG2000::GetSequentialData" (*IDL Reference Guide*).

Random Motion JPEG2000 Playback

To access a specified frame, use the IDLffMJPEG2000::GetData method. When using GetData, you can return an entire frame, or a component, tile, or region of a frame. You can also specify the resolution (level) of data to return.

The GetData method returns data when it has been decompressed. Unlike GetSequentialData, GetData does not use a background processing thread and there is no frame buffer involved. This means that the data returned by GetData can be accessed. (The data returned by GetSequentialData cannot be accessed as it returns only a pointer to the data on the frame buffer.) Since no background processing thread is involved, a simple WAIT statement can be used to control the playback rate when there is no need to implement a more robust timer mechanism.

Use GetData when you need to access a small number of distinct frames. Use GetSequentialData and the background processing thread when you want to playback a large number of frames at a specified rate as described in “Sequential Motion JPEG2000 Playback” on page 103.

Simple examples that use the GetData method are described in “Adding Data to MJ2 Animations” on page 98.

Controlling the Playback Rate

Sequential playback relies on the interaction of four IDLffMJPEG2000 methods, described in "Sequential Motion JPEG2000 Playback" on page 103. When you call StartSequentialReading, a background processing thread is started, and the selected data is decompressed and added to the frame buffer. Within a timer event, you must call the GetSequentialData and ReleaseSequentialData methods as a pair. These methods work cooperatively to access and then to release the frame data so that there is room for the decompression of the next frame.

Tip

If playback is delayed because there are not frame buffer slots available, you can modify the size of the frame buffer using the FRAME_BUFFER_LENGTH property. See "High Speed MJ2 Reading and Writing" on page 108 for details.

The timer mechanism can access the decompressed data from the frame buffer at intervals specified by a combination of the FRAME_PERIOD and TIMESCALE properties. The number of seconds allotted each frame is equal to the FRAME_PERIOD divided by the TIME_SCALE property (see the discussion under "FRAME_PERIOD" (*IDL Reference Guide*) for details). Access the required properties from an IDLffMJPEG2000 object (*oMJ2*) as follows:

```
oMJ2->GetProperty,N_FRAMES=nFrames, DIMENSIONS=dims, $
   FRAME_PERIOD=vFramePeriod, TIMESCALE=vTimeScale

; Compute seconds per frame.
vFrameRate = FLOAT(vFramePeriod)/vTimeScale
```

In the previous line, the FLOAT function ensures the return of a floating point frame rate value and avoids errors caused by attempting to divide by zero. This frame rate value can then be passed to the timer mechanism to control playback rate. For an MJ2 file that has frames with varied FRAME_PERIOD property values, computing the frame rate for each frame and passing it to the timer mechanism will alter the playback speed. The following example creates an MJ2 file with varied frame period values and then uses these values to compute a value to be passed to a widget timer event, which alters the playback rate to reflect the frame period of each frame.

Example Code

This example `mj2_timer_doc.pro`, is located in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `mj2_timer_doc` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT mj2_timer_doc.pro`.

Timer Mechanisms

There are two primary options for timer mechanisms that can be used to control the playback rate of an MJ2 animation in an IDL application:

Option	Description
IDLitWindow	A number of IDLitWindow methods work in concert to control what happens during a timer event: • IDLitWindow::SetEventMask — use this method to turn timer events on and off • IDLitWindow::SetTimerInterval — set this equal to the desired frame rate (seconds/frame) • IDLitWindow::OnTimer — write code in this procedure to get and release frame data at the rate specified in SetTimerInterval The sample MJ2 player, `mj2_player.pro`, located in the *`IDL_DIR`*`\examples\mjpeg2000` directory uses an IDLitWindow timer mechanism. See "Sample Motion JPEG2000 Player and Writer" on page 95 for more information.
Widget Timer	A timer event can be associated with a number of widgets although it is typically associated with one that has no events of its own such as a base or label. The WIDGET_CONTROL procedure associates a timer with a widget and sets the rate. The `mj2_timer_doc.pro` example, located in the `examples/doc/objects` subdirectory of the IDL distribution, shows how to control playback rate with a widget timer. See "Timer Events" (Chapter 4, *Widget Application Programming*) for more information on these events.

Table 4-3: Timer Mechanisms Options for MJ2 Playback

Of the two options listed above, the IDLitWindow timer will more accurately reflect true frame rates. The widget timer will show rate changes, but may not have the same degree of accuracy as the IDLitWindow timer mechanism.

High Speed MJ2 Reading and Writing

Animation playback or creation can be delayed due to the time required to decompress or compress frame data. The following sections describe ways to avoid such delays during file reading or writing.

High Speed Sequential Playback

If the desired playback speed exceeds the rate at which frames can be decompressed (as described in "Sequential Motion JPEG2000 Playback" on page 103), you can decompress all of the frames before starting the playback. To do so, you need to set the FRAME_BUFFER_LENGTH property to the total number of frames to be played back before calling IDLffMJPEG2000::StartSequentialReading.

When you call StartSequentialReading, the background processing thread will begin decompressing the frames and storing them in the frame buffer. Before calling the GetSequentialData/ReleaseSequentialData pair of methods, make sure that all frames have been read into the frame buffer. You can check this using one of the following:

- Check the STATE property—if frames are still being decompressed by the processing thread, the property returns 1 (running). When all frames have been decompressed, the background processing thread shuts down and the STATE property returns to 0 (idle).
- Check the FRAMES_IN_BUFFER property—if the number of frames in the buffer equals the FRAME_BUFFER_LENGTH property you set prior to starting the decompression, then all of the desired frames have been decompressed.

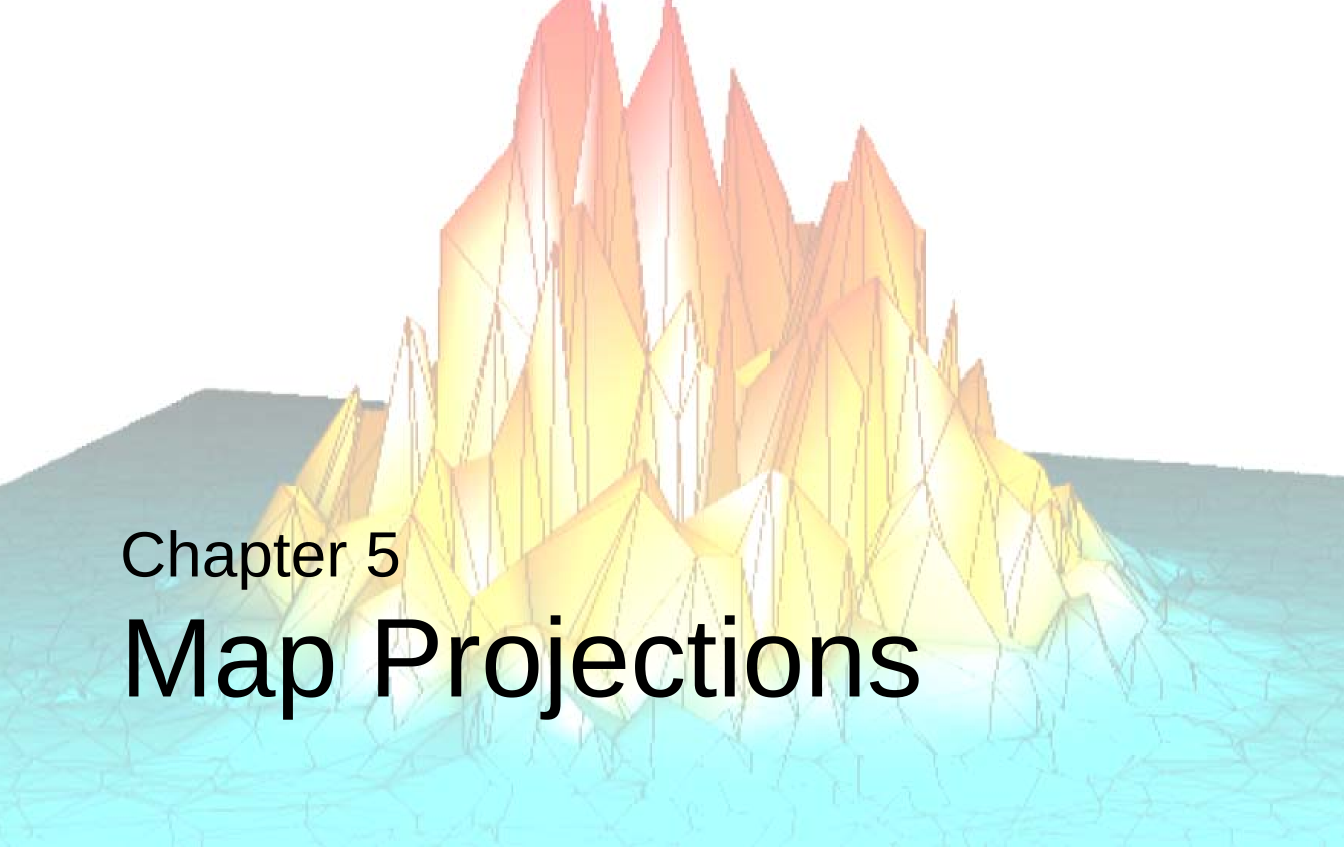
Note

This technique, decompressing all the desired frames prior to playback, can consume large amounts of memory depending on the number and size of the frames. Also, remember that the decompressed frames will remain in the frame buffer until you call the StopSequentialReading method.

High Speed MJ2 File Writing

In some situations, the desired write speed may exceed the rate at which frames can be compressed. When you call SetData, the data is added to the frame buffer where it is compressed by a background processing thread. If compression cannot keep up with the SetData calls, the frame buffer fills up and SetData must wait for an available frame buffer slot before it can return.

To avoid such a delay, you can make sure there is always a slot available for the SetData call by increasing the FRAME_BUFFER_LENGTH property value. This technique ensures there is no delay caused by file compression, but can consume large amounts of memory depending on the number and size of the frames.



Chapter 5

Map Projections

The following topics are covered in this chapter:

Overview of Mapping

This section introduces graphic map display considerations as well as information about common map projections. This section does not describe how to create a map display. See the following topic for these resources.

Creating a Map Display

IDL provides interactive and static map display functionality. You can use the iMap iTool to interactively configure a map display. If you prefer a static display, you can use map routines. See the following for details:

- Interactive iMap display — see Chapter 15, “Working with Maps” (*iTool User’s Guide*)
- Map-related routines — see “Mapping” (*IDL Quick Reference*)

Examples of Creating Map Displays

See the following resources in the *IDL Reference Guide* for examples:

- IMAP — provides examples of displaying images and contours over a map projection.
- MAP_PROJ_FORWARD — creates a latitude and longitude grid with labels for a Goodes Homolosine map projection in an Object Graphics display. Typically MAP_PROJ_INIT is used with MAP_PROJ_FORWARD and MAP_PROJ_INVERSE.
- MAP_SET — establishes the coordinate conversion mechanism for mapping points on a globe’s surface to points on a plane, according to the selected projections type. You can then use MAP_GRID and MAP_CONTINENTS to add grid lines and continents to the map display. See MAP_IMAGE for an example of warping an image to a projection.

Graphics Techniques for Mapping

Standard graphics techniques are insufficient when projecting areas on a sphere to a two-dimensional surface for two reasons. First, two points on a sphere are connected by two different lines. Second, areas may wrap around the edges of cylindrical and pseudo-cylindrical projections.

Graphical entities on the surface of a sphere can be properly represented on any map by using a combination of the following four stages: splitting, 3D clipping, projection, and rectangular clipping. The IMAP and MAP_SET procedures automatically sets up the proper mapping technique to best fit the projection selected by the user.

Warning

For proper rendering, splitting, and clipping, polygons must be traversed in counter-clockwise order when observed from outside the sphere. If this requirement is not met, the exterior, instead of the interior, of the polygons may be filled. Also, vectors connecting the points spanning the singular line for cylindrical projections will be drawn in the wrong direction if polygons are not traversed in the correct order.

Splitting

The splitting stage is used for cylindrical and pseudo-cylindrical projections. The singular line, one half of a great circle line, is located opposite the center of the projection; points on this line appear on both edges of the map. The singular line is the intersection of the surface of the sphere with a plane passing through the center of projection, one of the poles of projections, and the center of the sphere.

3D Clipping

Map graphics are clipped to one side of an arbitrary clipping plane in one or more clipping stages. For example, to draw a hemisphere centered on a given point, the clipping plane passes through the center of the sphere and has a normal vector that coincides with the given point.

Projection

In the projection stage, a point expressed in latitude and longitude is transformed to a point on the mapping plane.

Rectangular Clipping

After the map graphics have been projected onto the mapping plane, a conventional rectangular clipping stage ensures that the graphics are properly bounded and closed in the rectangular display area.

Map Projection Types

In the following sections, the available IDL projections are discussed in detail. The projections are grouped within three categories:

- "Azimuthal Projections" on page 116
- "Cylindrical Projections" on page 125
- "Pseudocylindrical Projections" on page 130

Note

The General Cartographic Transformation Package (GCTP) map projections are not described here. Documentation for the GCTP package is available from the US Geologic Survey at `http://mapping.usgs.gov`.

Note

In this text, the plane of the projection is referred to as the *UV* plane with horizontal axis *u* and vertical axis *v*.

Azimuthal Projections

With azimuthal projections, the *UV* plane is tangent to the globe. The point of tangency is projected onto the center of the plane and its latitude and longitude are the points at the center of the map projection, respectively. Rotation is the angle between North and the *v*-axis.

Important characteristics of azimuthal maps include the fact that directions or azimuths are correct from the center of the projection to any other point, and great circles through the center are projected to straight lines on the plane.

The IDL mapping package includes the following azimuthal projections:

- "Orthographic Projection" on page 117
- "Stereographic Projection" on page 117
- "Gnomonic Projection" on page 118
- "Azimuthal Equidistant Projection" on page 119
- "Aitoff Projection" on page 120
- "Lambert's Equal Area Projection" on page 121
- "Hammer-Aitoff Projection" on page 122
- "Satellite Projection" on page 123

Orthographic Projection

The orthographic projection was known by the Egyptians and Greeks 2000 years ago. This projection looks like a globe because it is a perspective projection from infinite distance. As such, it maps one hemisphere of the globe into the *UV* plane. Distortions are greatest along the rim of the hemisphere where distances and land masses are compressed.

The following figure shows an orthographic projection centered over Eastern Spain at a scale of 70 million to 1.

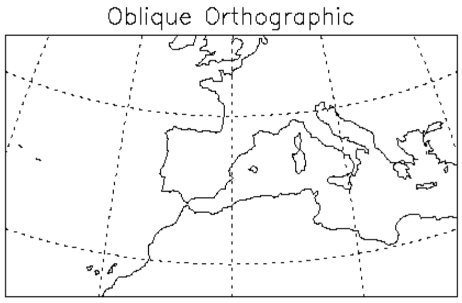


Figure 5-1: Orthographic Projection

Stereographic Projection

The stereographic projection is a true perspective projection with the globe being projected onto the *UV* plane from the point *P* on the globe diametrically opposite to the point of tangency. The whole globe except *P* is mapped onto the *UV* plane. There is great distortion for regions close to *P*, since *P* maps to infinity.

The stereographic projection is the only known perspective projection that is also conformal. It is frequently used for polar maps. For example, a stereographic view of the north pole has the south pole as its point of perspective.

The following figure shows an equatorial stereographic projection with the hemisphere centered on the equator at longitude –105 degrees.

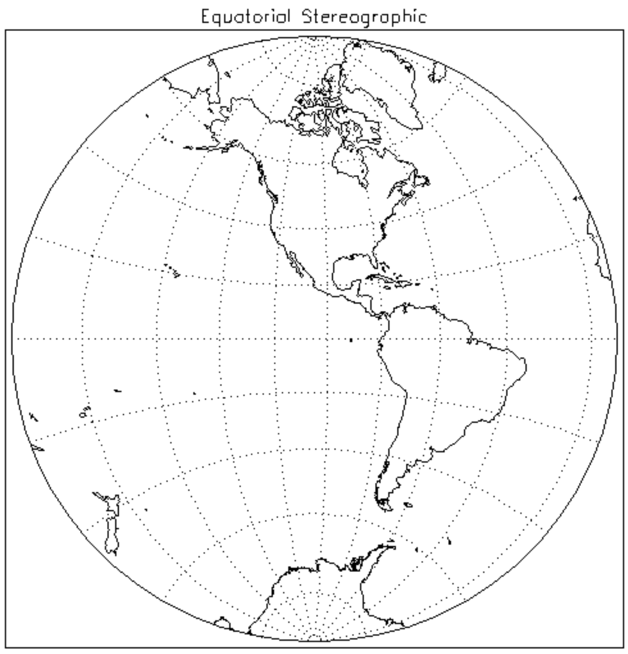


Figure 5-2: An Azimuthal Projection

Gnomonic Projection

The gnomonic projection (also called Central or Gnomic) projects all great circles to straight lines. The gnomonic projection is the perspective, azimuthal projection with point of perspective at the center of the globe. Hence, with the gnomonic projection, the interior of a hemispherical region of the globe is projected to the *UV* plane with the rim of the hemisphere going to infinity. Except at the center, there is great distortion of shape, area, and scale. The default clipping region for the gnomonic projection is a circle with a radius of 60 degrees at the center of projection.

The projection in the following figure is centered around the point at latitude 40 degrees and longitude –105 degrees. The region on the globe that is mapped lies between 20 degrees and 70 degrees of latitude and –130 degrees and –70 degrees of longitude.

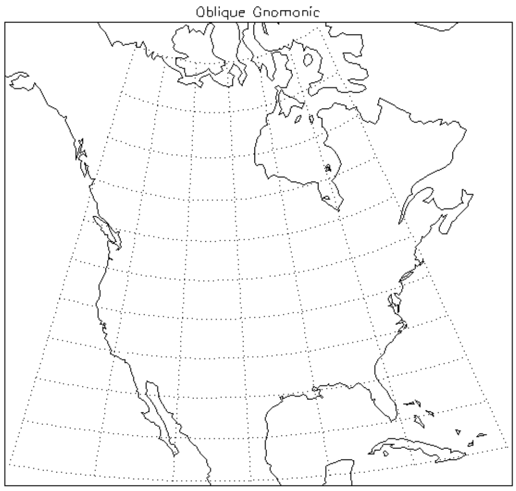


Figure 5-3: A Gnomonic Projection

Azimuthal Equidistant Projection

The azimuthal equidistant projection is also not a true perspective projection, because it preserves correctly the distances between the tangent point and all other points on the globe. Any line drawn through the tangent point reports distance correctly. Therefore, this projection type is useful for determining flight distances. The point *P* opposite the tangent point is mapped to a circle on the *UV* plane, and hence, the whole globe is mapped to the plane. There is infinite distortion close to the outer rim of the map, which is the circular image of *P*.

The following Azimuthal projection is centered at the South Pole and shows the entire globe.

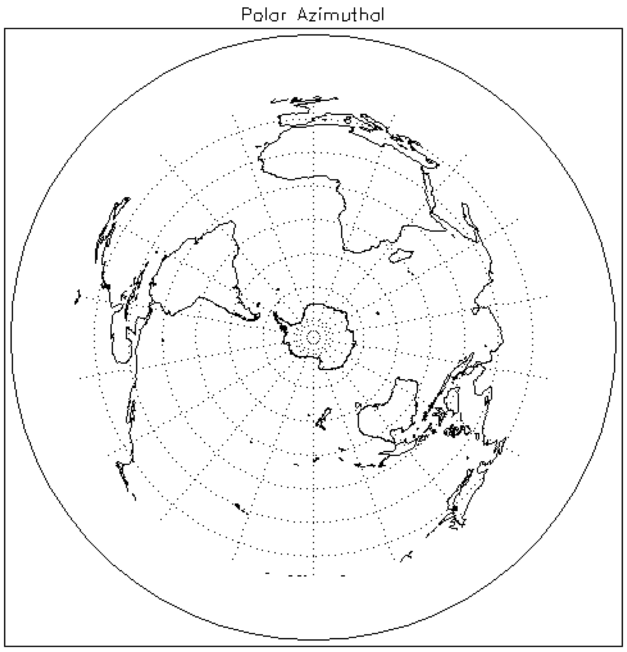


Figure 5-4: An Azimuthal Equidistant Projection

Aitoff Projection

The Aitoff projection modifies the equatorial aspect of one hemisphere of the azimuthal equidistant projection, described above. Lines parallel to the equator are stretched horizontally and meridian values are doubled, thereby displaying the world as an ellipse with axes in a 2:1 ratio. Both the equator and the central meridian are represented at true scale; however, distances measured between the point of tangency and any other point on the map are no longer true to scale.

An Aitoff projection centered on the international dateline is shown in the following figure.

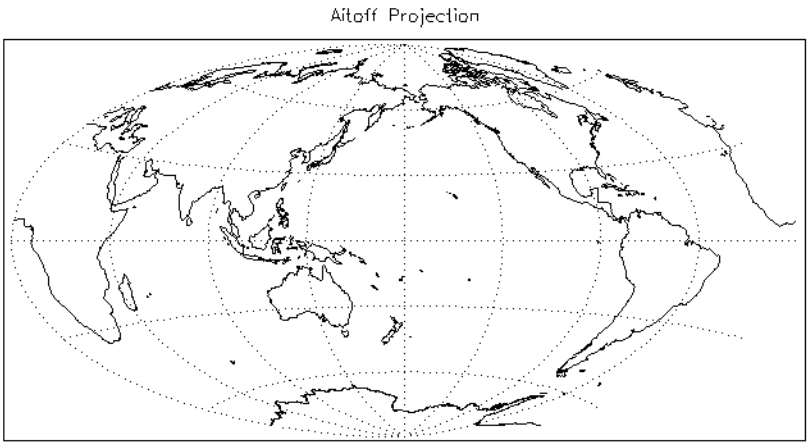


Figure 5-5: An Aitoff Projection

Lambert's Equal Area Projection

Lambert's equal area projection adjusts projected distances in order to preserve area. Hence, it is not a true perspective projection. Like the stereographic projection, it maps to infinity the point *P* diametrically opposite the point of tangency. Note also that to preserve area, distances between points become more contracted as the points become closer to *P*. Lambert's equal area projection has less overall scale variation than the other azimuthal projections.

The following figure shows the Northern Hemisphere rotated counterclockwise 105 degrees, and filled continents.

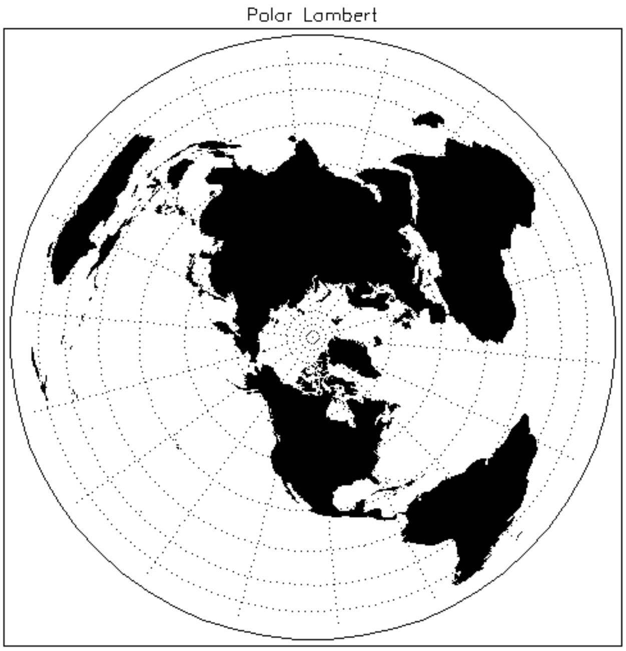


Figure 5-6: A Lambert's Equal Area Projection

Hammer-Aitoff Projection

Although the Hammer-Aitoff projection is not truly azimuthal, it is included in this section because it is derived from the equatorial aspect of Lambert's equal area projection limited to a hemisphere (in the same way Aitoff's projection is derived from the equatorial aspect of the azimuthal equidistant projection). In this derivation, the hemisphere is represented inside an ellipse with the rest of the world in the lunes of the ellipse.

Because the Hammer-Aitoff projection produces an equal area map of the entire globe, it is useful for visual representations of geographically related statistical data and distributions. Astronomers use this projection to show the entire celestial sphere on one map in a way that accurately depicts the relative distribution of the stars in different regions of the sky.

A Hammer-Aitoff projection centered on the international dateline is shown in the following figure:

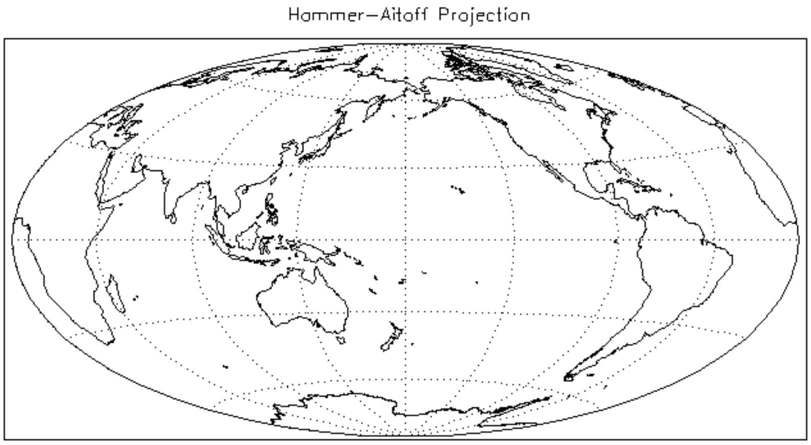


Figure 5-7: The Hammer-Aitoff Projection

Satellite Projection

The satellite projection, also called the General Perspective projection, simulates a view of the globe as seen from a camera in space. If the camera faces the center of the globe, the projection is called a Vertical Perspective projection (note that the orthographic, stereographic, and gnomonic projections are special cases of this projection), otherwise the projection is called a Tilted Perspective projection.

The globe is viewed from a point in space, with the viewing plane touching the surface of the globe at the point directly beneath the satellite (the sub-satellite point). If the projection plane is perpendicular to the line connecting the point of projection and the center of the globe, a Vertical Perspective projection results. Otherwise, the projection plane is horizontally turned Γ degrees clockwise from the north, then tilted ω degrees downward from horizontal.

The map in the accompanying figure shows the eastern seaboard of the United States from an altitude of about 160km, above Newburgh, NY.

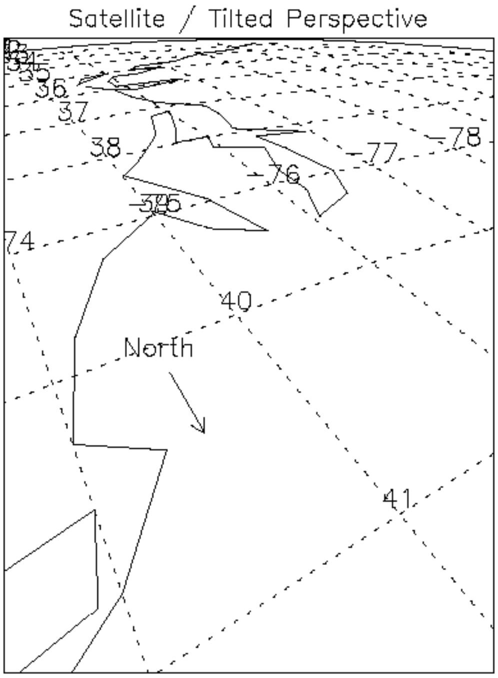


Figure 5-8: Satellite Projection

Cylindrical Projections

A cylindrical projection maps the globe to a cylinder which is formed by wrapping the *UV* plane around the globe with the *u*-axis coinciding with a great circle. The parameters P_{0lat}, P_{0lon}, and *Rot* determine the great circle that passes through the point $C=(P_{0lat}, P_{0lon})$. In the discussions below, this great circle is sometimes referred to as EQ. *Rot* is the angle between North at the map's center and the *v*-axis (which is perpendicular to the great circle). The cylinder is cut along the line parallel to the *v*-axis and passing through the point diametrically opposite to C. It is then rolled out to form a plane.

The cylindrical projections in IDL include: Mercator, Transverse Mercator, cylindrical equidistant, Miller, Lambert's conformal conic, and Alber's equal-area conic.

Mercator Projection

Mercator's projection is partially developed by projecting the globe onto the cylinder from the center of the globe. This is a partial explanation of the projection because vertical distances are subjected to additional transformations to achieve conformity—that is, local preservation of shape. Therefore, uses include navigation maps and equatorial maps. To properly use the projection, the user should be aware that the two points on the globe 90 degrees from the central great circle (e.g., the North and South Poles in the case that the selected great circle is the equator) are mapped to infinite distances. Limits are typically specified because of the great distortions around the poles when the equator is selected.

A simple mercator projection with latitude ranges from –80 degrees to 80 degrees is shown in the following figure.

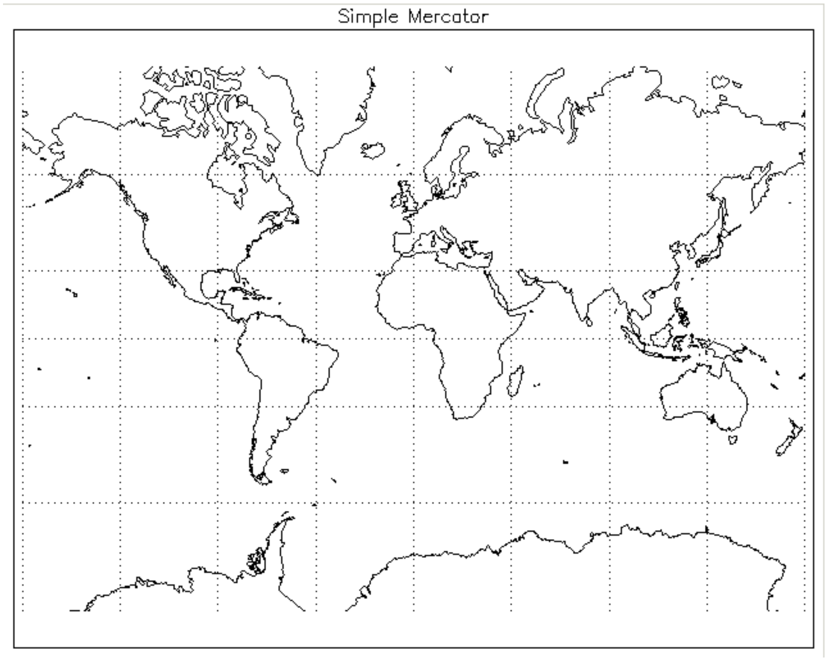


Figure 5-9: Simple Mercator Projection

Transverse Mercator Projection

The Transverse Mercator (also called the *UTM*, and *Gauss-Krueger* in Europe) projection rotates the equator of the Mercator projection 90 degrees so that it follows a specified central meridian. In other words, the Transverse Mercator involves projecting the Earth onto a cylinder which is always in contact with a meridian instead of with the Equator.

The central meridian intersects two meridians and the Equator at right angles; these four lines are straight. All other meridians and parallels are complex curves which are concave toward the central meridian. Shape is true only within small areas and the areas increase in size as they move away from the central meridian. Most other IDL projections are scaled in the range of +/– 1 to +/– 2 Pi; the UV plane of the Transverse Mercator projection is scaled in meters. The conformal nature of this projection and its use of the meridian makes it useful for north-south regions. The Clarke 1866 ellipsoid is used for the default.

The following Transverse Mercator map shows North and South America, with a central meridian of –90 degrees West and centered on the Equator.

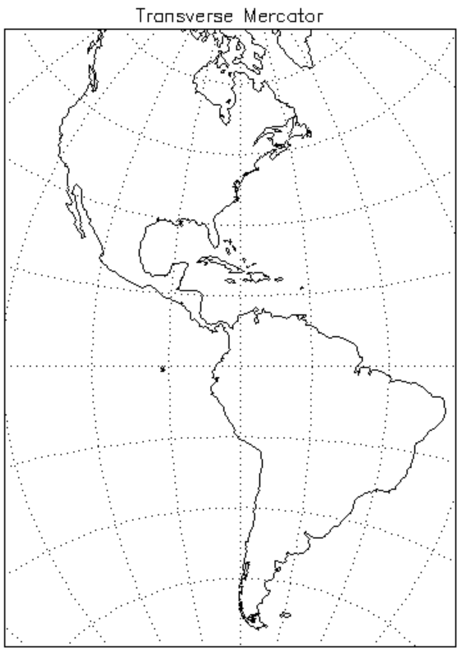


Figure 5-10: Transverse Mercator Projection

Cylindrical Equidistant Projection

The cylindrical equidistant projection is one of the simplest projections to construct. If EQ is the equator, this projection simply lays out horizontal and vertical distances on the cylinder to coincide numerically with their measurements in latitudes and longitudes on the sphere. Hence, the equidistant cylindrical projection maps the entire globe to a rectangular region bounded by

$-180 \leq u \leq 180$

and

$-90 \leq v \leq 90$

If EQ is the equator, meridians and parallels will be equally spaced parallel lines.

The following figure shows a simple cylindrical equidistant projection and an oblique cylindrical equidistant projection rotated by 45°.

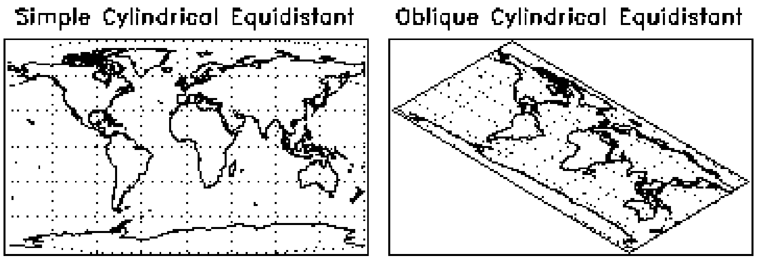

Simple Cylindrical Equidistant
Oblique Cylindrical Equidistant

Figure 5-11: Cylindrical Projections

Miller Cylindrical Projection

The Miller projection is a simple mathematical modification of the Mercator projection, incorporating some aspects of cylindrical projections. It is not equal-area, conformal or equidistant along the meridians. Meridians are equidistant from each other, but latitude parallels are spaced farther apart as they move away from the Equator, thereby keeping shape and area distortion to a minimum. The meridians and parallels intersect each other at right angles, with the poles shown as straight lines. The Equator is the only line shown true to scale and free of distortion.

Conic Projection

The Lambert’s conformal conic with two standard parallels is constructed by projecting the globe onto a cone passing through two parallels. Additional scaling achieves conformity. The pole under the cone’s apex is transformed to a point, and the other pole is mapped to infinity. The scale is correct along the two standard parallels. Parallels can be specified and are projected onto circles and meridians onto equally spaced straight lines. The following figure shows the map shown in the accompanying figure, which features North America with standard parallels at 20 degrees and 60 degrees.

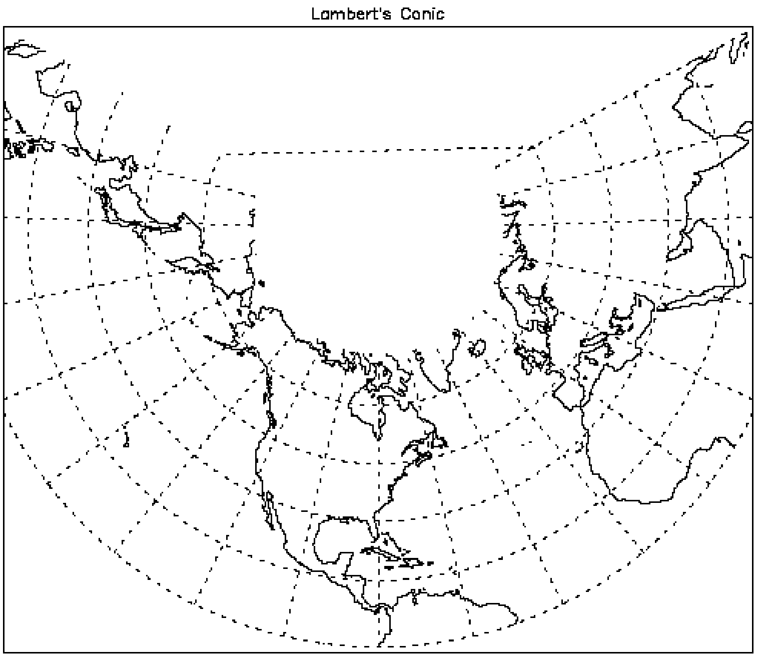


Figure 5-12: Lambert's Conformal Conic with Standard Parallels at 20° and 60°

Albers Equal-Area Conic Projection

The Albers Equal-Area Conic is like most other conics in that meridians are equally spaced radii, parallels are concentric arcs of circles and scale is constant along any parallel. To maintain equal area, the scale factor along meridians is the reciprocal of the scale factor along parallels, with the scale along the parallels between the two standard parallels too small, and the scale beyond the standard parallels too large. Standard parallels are correct in scale along the parallel, as well as in every direction.

The Albers projection is particularly useful for predominantly east-west regions. Any keywords for the Lambert conformal conic also apply to the Albers conic.

Pseudocylindrical Projections

Pseudocylindrical projections are distinguished by the fact that in their simplest form, lines of latitude are parallel straight lines and meridians are curved lines.

Robinson Cylindrical

This pseudocylindrical projection was designed by Arthur Robinson in 1963 for Rand McNally. It is suitable for World maps and is a compromise to best fulfill a number of conflicting requirements, including an uninterrupted format, minimal shearing, minimal apparent area-scale distortion for major continents, and simplicity. It was designed to make the world look right. Since its introduction, it has been adopted by the National Geographic Society for many of their world maps.

Each individual parallel is equally divided by the meridians. The poles are represented by lines rather than points to avoid compressing the northern land masses. The central meridian should always be 0 degrees longitude to retain the correct balance of shapes, sizes, and relative positions.

The following figure shows a Robinson projection.

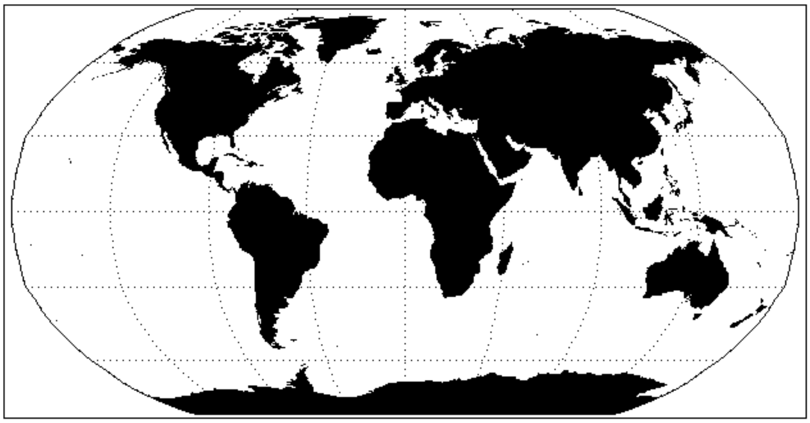


Figure 5-13: Robinson Projection

Sinusoidal Projection

With the sinusoidal projection, the central meridian is a straight line and all other meridians are equally spaced sinusoidal curves. The scaling is true along the central meridian as well as along all parallels.

The sinusoidal projection is one of the easiest projections to construct. The formulas below from Snyder (1987) give the relationship between the latitude ϕ and longitude λ of a point on the globe and its image on the *UV* plane.

$$u = \lambda \cos\phi$$

$$v = \phi$$

The following shows the sinusoidal map of the whole globe centered at longitude 0 degrees and latitude 0 degrees.

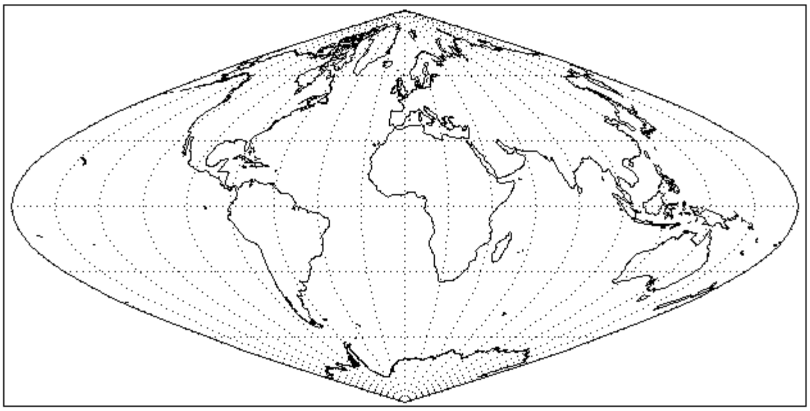


Figure 5-14: Sinusoidal Projection

Mollweide Projection

With the Mollweide projection, the central meridian is a straight line, the meridians 90 degrees from the central meridian are circular arcs and all other meridians are elliptical arcs. The Mollweide projection maps the entire globe onto an ellipse in the *UV* plane. The circular arcs encompass a hemisphere and the rest of the globe is contained in the lunes on either side.

The following figure shows a Mollweide projection in oblique form.

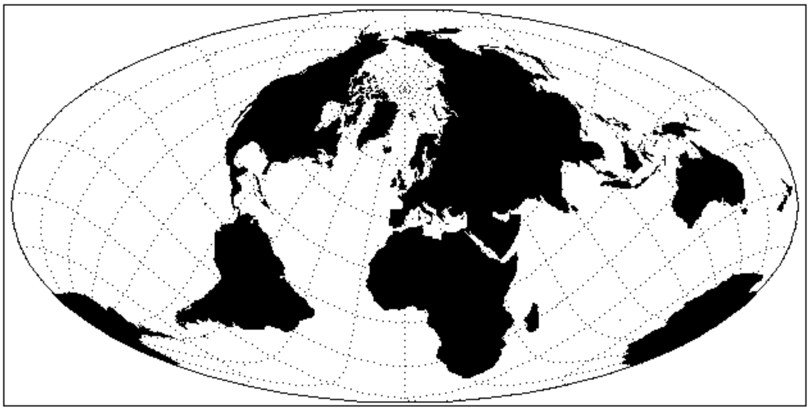


Figure 5-15: Mollweide Projection

Since the center of the projection is not on the equator, parallels of latitude are not straight lines, just as they are not straight lines with an oblique Mercator or cylindrical equidistant projection.

Goode's Homolosine Projection

The Goode interrupted Homolosine projection, developed by J. Paul Goode, in 1923, is designed for World maps to show the continents with minimal scale and shape distortion. This is accomplished by interrupting the projection and choosing several central meridians to coincide with large land masses. This projection is a fusion of the Sinusoidal projection between the latitudes of 44.7 degrees North and South, and the Mollweide projection between these parallels and the poles.

The following figure shows an example of Goode’s Homolosine projection.

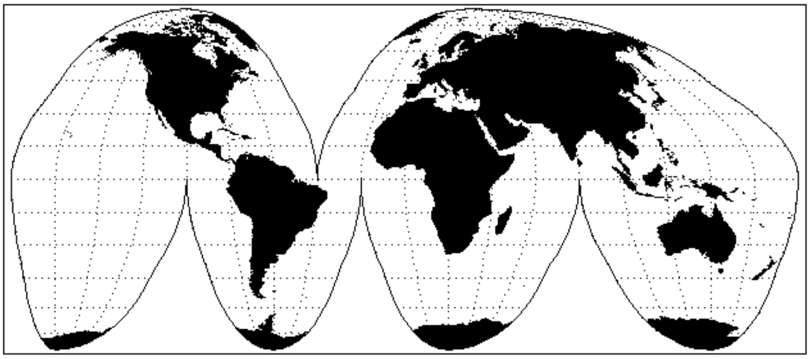


Figure 5-16: Goode’s Homolosine Projection

High-Resolution Continent Outlines

IDL supports two different datasets that contain continent outlines and other geographical and political boundaries. The default data set is a low-resolution continental outline database that is automatically installed when you install IDL. The high-resolution database was adapted from the 1993 CIA World Map database by Thomas Oetli of the Swiss Meteorological Institute. The high-resolution outlines are found in an optional data set that may not have been installed when your copy of IDL was first installed.

To access the high-resolution data set, simply set the HIRES keyword when calling MAP_CONTINENTS with the COASTS, COUNTRIES, FILL_CONTINENTS, or RIVERS keywords. You can also get high-resolution continent boundaries by calling MAP_SET with the HIRES and CONTINENTS keywords set. See MAP_CONTINENTS in the *IDL Reference Guide* for an example of using the high-resolution outlines.

Resolution of Map Databases

Data points in the CIA World Map database are approximately one kilometer apart. Note, however, that in the case of the coast and river databases, actual distances between the data points may be much smaller because of convolutions in the coastline or riverbed.

Data points in the low-resolution map database are either a subset of the high-resolution database (rivers and country boundaries) or are based on the continental map database used in previous versions of IDL (the file `supmap.dat` in the `resource/maps` subdirectory of the IDL distribution). Data points in the low-resolution database are approximately 10 kilometers apart.

Neither of the map databases is intended for high-precision work.

The following table compares the low-resolution and high-resolution map databases:

Feature	Low-Resolution	High-Resolution
Coastlines, islands, and lakes (including continental outlines)	Data in file `supmap.dat`.	Entire CIA World Map
Continental polygons	Data extracted from `supmap.dat`.	Every 20th point of CIA World Map.
Rivers	Every 250th point of the CIA World Map.	Entire CIA World Map.
National boundaries	Every 100th point of CIA World Map.	Entire CIA World Map.

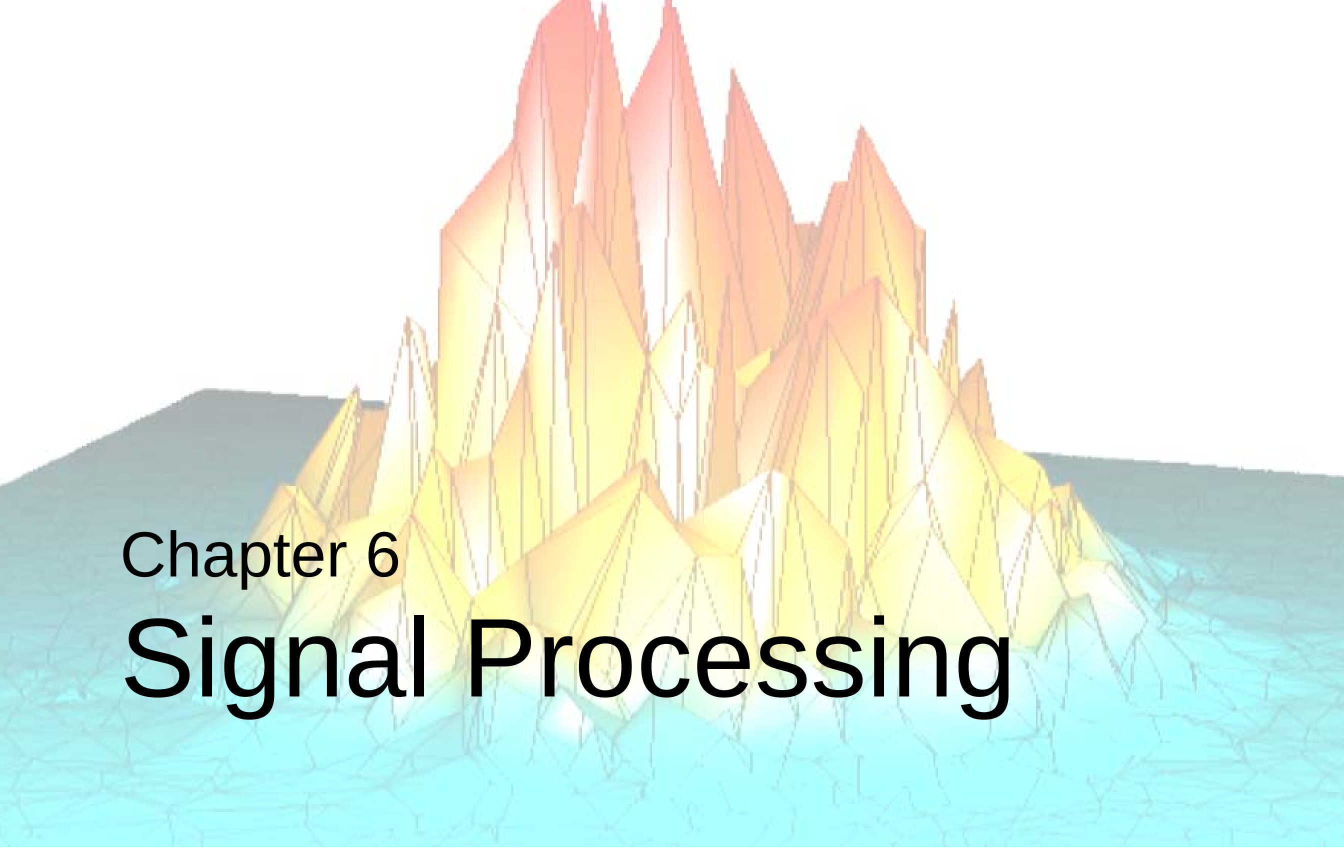
Table 5-1: Comparison of Low- and High-resolution Map Databases

References

Greenwood, David (1964), *Mapping*, University of Chicago Press, Chicago.

Pearson, Frederick II (1990), *Map Projections: Theory and Applications*, CRC Press, Inc., Boca Raton.

Snyder, John P. (1987), *Map Projections—A Working Manual*, U.S. Geological Survey Professional Paper 1395, U.S.Government Printing Office, Washington, D.C.



Chapter 6

Signal Processing

The following topics are covered in this chapter:

Overview of Signal Processing

A signal, by definition, contains information. Any signal obtained from a physical process also contains noise. It is often difficult or impossible to make sense of the information contained in a digital signal by looking at it in its raw form—that is, as a sequence of real values at discrete points in time. Signal analysis transforms offer natural, meaningful, alternate representations of the information contained in a signal.

This chapter describes IDL's digital signal processing tools. Most of the procedures and functions mentioned here work in two or more dimensions. For simplicity, only one dimensional signals are used in the examples.

Routines for Signal Processing

For a list of IDL signal processing routines, see the functional category of "Signal Processing" (*IDL Quick Reference*). There you will find a brief introduction to the routines. More detailed information is available in the *IDL Reference Guide*.

Running the Example Code

The examples in this chapter are written to take advantage of iTools. The example code is part of the IDL distribution. All of the files mentioned are located in the `examples/doc/signal` subdirectory of the IDL distribution. By default, this directory is part of IDL's path; if you have not changed your path, you will be able to run the examples as described here. See "!PATH" (Appendix D, *IDL Reference Guide*) for information on IDL's path.

Digital Signals

A one-dimensional digital signal is a sequence of data, represented as a vector in an array-oriented language like IDL. The term digital actually describes two different properties:

1. The signal is defined only at discrete points in time as a result of sampling, or because the instrument which measured the signal is inherently discrete-time in nature. Usually, the time interval between measurements is constant.
2. The signal can take on only discrete values.

In this discussion, we assume that the signal is sampled at a time interval. The concepts and techniques presented here apply equally well to any type of signal—the independent variable may represent time, space, or any abstract quantity.

The following IDL commands create a simulated digital signal $u(k)$, sampled at an interval `delt`. This simulated signal will be used in examples throughout this chapter. The simulated signal contains 1024 time samples, with a sampling interval of 0.02 seconds. The signal contains a DC component and components at 2.8, 6.5, and 11.0 cycles per second.

Enter the following commands at the IDL prompt to create the simulated signal:

```
N = 1024 ; number of samples
delt = 0.02 ; sampling interval

; Simulated signal.
u = -0.3 $
   + 1.0 * SIN(2 * !PI * 2.8 * delt * FINDGEN(N)) $
   + 1.0 * SIN(2 * !PI * 6.25 * delt * FINDGEN(N)) $
   + 1.0 * SIN(2 * !PI * 11.0 * delt * FINDGEN(N))
```

Example Code

Alternately, type `@sigprc01` at the IDL prompt to run the `sigprc01`batch file that creates the signal. See “Running the Example Code” on page 138 if IDL does not find the batch file.

Because the signal is digital, the conventional way to display it is with a histogram (or step) plot. To create a histogram plot, set the PSYM keyword to the PLOT routine equal to 10. A section of the example signal *u*(*k*) is plotted in the figure below.

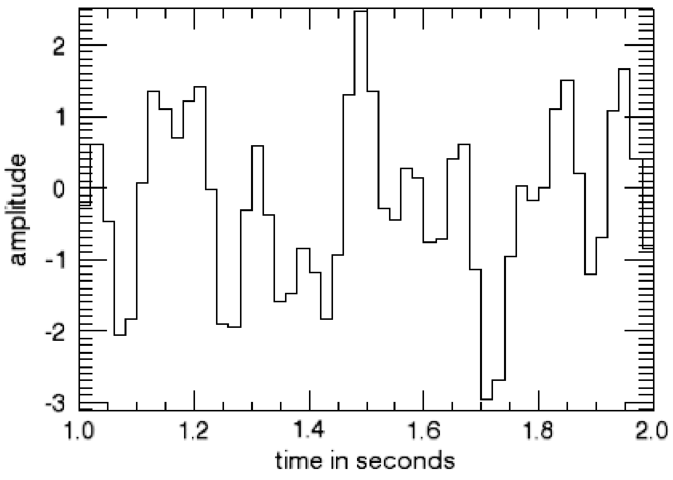


Figure 6-1: Histogram Plot of Sample Signal u(k)

Note

When the number of sampled data points is large, the steps in the histogram plot are too small to see. In such cases you should not plot in histogram mode.

Example Code

Type `@sigprc02` at the IDL prompt to run the batch file that creates this display. The source code is located in `sigprc02`, in the `examples/doc/signal` directory. See "Running the Example Code" on page 138 if IDL does not find the batch file.

Signal Analysis Transforms

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

The Fourier Transform

The Discrete Fourier Transform (DFT) is the most widely used method for determining the frequency spectra of digital signals. This is due to the development of an efficient algorithm for computing DFTs known as the Fast Fourier Transform (FFT).

The discrete Fourier transform, $v(m)$, of an N-element, one-dimensional function, $u(k)$, is defined as:

$$v(m) = \frac{1}{N}\sum_{k=0}^{N-1} u(k)\exp[-j2\pi mk/N]$$

The inverse transform is defined as:

$$u(k) = \sum_{m=0}^{N-1} v(m)\exp[j2\pi mk/N]$$

IDL implements the Fast Fourier Transform in the FFT function. You can find details on using IDL's FFT function in the following sections and in "FFT" (*IDL Reference Guide*).

Interpreting FFT Results

Just as the sampled time data represents the value of a signal at discrete points in time, the result of a (forward) Fast Fourier Transform represents the spectrum of the signal at discrete frequencies. These discrete frequencies are a function of the frequency index (*m*), the number of samples collected (*N*), and the sampling interval (δ):

$$f(m) = \frac{m}{N\delta}$$

The frequencies for which the FFT of a sampled signal are defined are sometimes called frequency bins, which refers to the histogram-like nature of a discrete spectrum. The width of each frequency bin is 1/(*N* * δ).

Due to the complex exponential in the definition of the DFT, the spectrum has a cyclic dependence on the frequency index *m*. That is:

$$v(m + pN) = v(m)$$

for *p* = any integer.

The frequency spectrum computed by IDL's FFT function for a one-dimensional time sequence is stored in a vector with indices running from 0 to *N*–1, which is also a valid range for the frequency index *m*. However, the frequencies associated with frequency indices greater than *N*/2 are above the Nyquist frequency and are not physically meaningful for sampled signals. Many textbooks choose to define the range of the frequency index *m* to be from – (*N*/2 – 1) to *N*/2 so that it is (nearly) centered around zero. From the cyclic relation above with *p* = –1:

$v(-(N/2 - 1)) = v(N/2 + 1 - N) = v(N/2 + 1)$

$v(-(N/2 - 2)) = v(N/2 + 2 - N) = v(N/2 + 2)$

...

$v(-2) = v(N - 2 - N) = v(N - 2)$

$v(-1) = v(N - 1 - N) = v(N - 1)$

This index shift is easily accomplished in IDL with the SHIFT function. See "Real and Imaginary Components" on page 144 for an example.

Displaying FFT Results

Depending on the application, there are many ways to display spectral data, the result of the (forward) FFT function.

Real and Imaginary Components

The most direct way is to plot the real and imaginary parts of the spectrum as a function of frequency index or as a function of the corresponding frequencies. The following figure displays the real and imaginary parts of the spectrum *v*(*m*) of the sampled signal *u*(*k*) for frequencies from $-(N/2 - 1)/(N * \delta)$ to $(N/2)/(N * \delta)$ cycles per second.

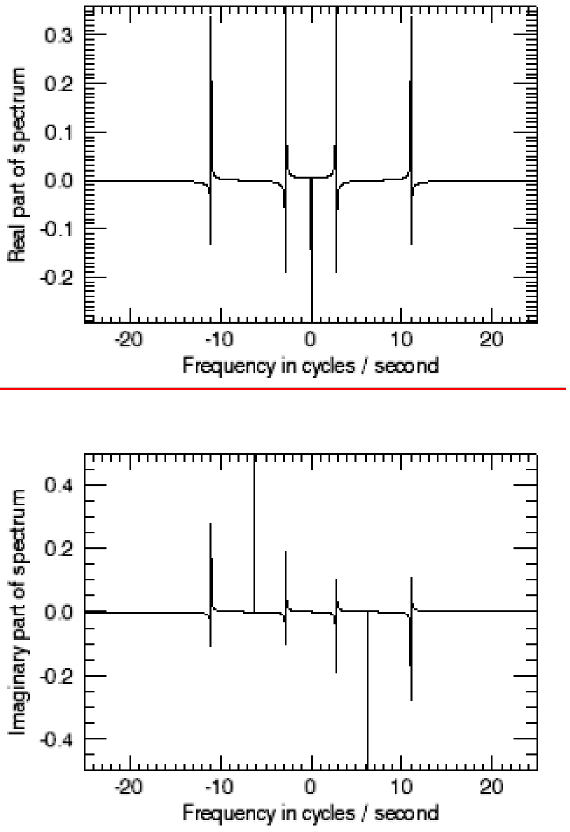


Figure 6-2: Real and Imaginary Parts of the Sample Signal

Example Code

Type `@sigprc03` at the IDL prompt to run the batch file that creates this display. The source code is located in `sigprc03`, in the `examples/doc/signal` directory. See "Running the Example Code" on page 138 if IDL does not find the batch file.

IDL's FFT function always returns a single- or double-precision complex array with the same dimensions as the input argument. In the case of a forward FFT performed on a one-dimensional vector of *N* real values, the result is an *N*-element vector of complex quantities, which takes 2*N* real values to represent. It would seem that there is twice as much information in the spectral data as there is in the time sequence data. This is not the case. For a real valued time sequence, half of the information in the frequency sequence is redundant. Specifically:

```
; 1 redundant value:
IMAGINARY(v(0)) = 0.0
; 1 redundant value:
IMAGINARY(v(N/2)) = 0.0
```

and

```
; for m=1 to N/2-1, N-2 redundant values:
v(N-m) = CONJ(v(m))
```

so that exactly *N* of the single- or double-precision values used to represent the frequency spectrum are redundant. This redundancy is evident in the previous figure. Notice that the real part of the spectrum is an even function (symmetric about zero), and the imaginary part of the spectrum is an odd function (anti-symmetric about zero). This is always the case for the spectra of real-valued time sequences.

Because of the redundancy in such spectra, it is common to display only half of the spectrum of a real time sequence. That is, only the spectral values with frequency indices from 0 to *N*/2, which correspond to frequencies from 0 to 1/(2 * δ), the Nyquist frequency. This vector of positive frequencies is generated in IDL with the following command:

```
; f = [0.0, 1.0/(N*delt), ... , 1.0/(2.0*delt)]
F = FINDGEN(N/2+1)/(N*delt)
```

Magnitude and Phase

It is also common to display the magnitude and phase of the spectrum, which have physical significance, as opposed to the real and imaginary parts of the spectrum, which do not have physical significance. Since there is a one-to-one correspondence between a complex number and its magnitude and phase, no information is lost in the transformation from a complex spectrum to its magnitude and phase. In IDL, the magnitude is easily determined with the absolute value (ABS) function, and the phase with the arc-tangent (ATAN) function. By one widely used convention, the magnitude of the spectrum is plotted in decibels (dB) and the phase is plotted in degrees, against frequency on a logarithmic scale. The magnitude and phase of our sample signal are plotted in the same data space, shown in the figure below.

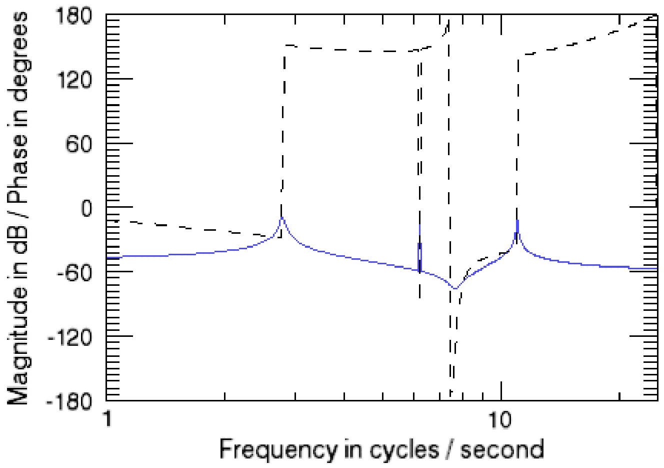


Figure 6-3: Magnitude (Solid LIne) and Phase (Dashed Line) of the Sample Signal

Example Code

Type `@sigprc04` at the IDL prompt to run the batch file that creates this display. The source code is located in `sigprc04`, in the `examples/doc/signal` directory. See "Running the Example Code" on page 138 if IDL does not find the batch file.

Using a logarithmic scale for the frequency axis has the advantage of spreading out the lower frequencies, while higher frequencies are crowded together. Note that the spectrum at zero frequency (DC) is lost completely on a semi-logarithmic plot.

The previous figure shows the strong frequency components at 2.8, 6.25, and 11.0 cycles/second as peaks in the magnitude plot, and as discontinuities in the phase plot. The magnitude peak at 6.25 cycles/second is a narrow spike, as would be expected from the pure sine wave component at that frequency in the time data sequence. The peaks at 2.8 and 11.0 cycles/second are more spread out, due to an effect known as smearing or leakage. This effect is a direct result of the definition of the DFT and is not due to any inaccuracy in the FFT. Smearing is reduced by increasing the length of the time sequence, or by choosing a sample size which includes an integral number of cycles of the frequency component of interest. There are an integral number of cycles of the 6.25 cycles/second component in the time sequence used for this example, which is why the peak at that frequency is sharper.

The apparent discontinuity in the phase plot at around 7.45 cycles/second is an anomaly known as phase wrapping. It is a result of resolving the phase from the real and imaginary parts of the spectrum with the arctangent function (ATAN), which returns principal values between –180 and +180 degrees.

Power Spectrum

Finally, for many applications, the phase information is not useful. For these, it is often customary to plot the power spectrum, which is the square of the magnitude of the complex spectrum. The resulting plot is shown in the figure below.

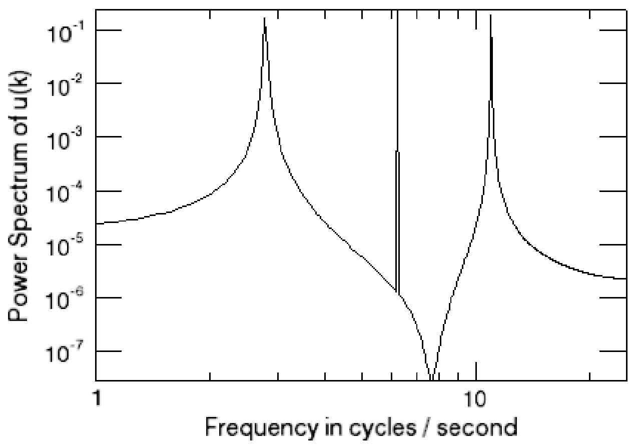


Figure 6-4: Power Spectrum of the Sample Signal

Example Code

Type `@sigprc05` at the IDL prompt to run the batch file that creates this display. The source code is located in `sigprc05`, in the `examples/doc/signal` directory. See "Running the Example Code" on page 138 if IDL does not find the batch file.

Using Windows

The smearing or leakage effect mentioned previously is a direct consequence of the definition of the Discrete Fourier Transform and of the fact that a finite time sample of a signal often does not include an integral number of some of the frequency components in the signal. The effect of this truncation can be reduced by increasing the length of the time sequence or by employing a windowing algorithm. IDL's HANNING function computes two windows which are widely used in signal processing: the Hanning window and the Hamming window.

Hanning Window

The Hanning window is defined as:

$$w(k) = \frac{1}{2}\left(1 - \cos\left(\frac{2\pi k}{N}\right)\right)$$

The resulting vector is multiplied element-by-element with the sampled signal vector before applying the FFT. For example, the following IDL command computes the Hanning window and then applies the FFT function:

```
v_n = FFT(HANNING(N)*U)
```

The power spectrum of the Hanning windowed signal shows the mitigation of the truncation effect (see the figure below).

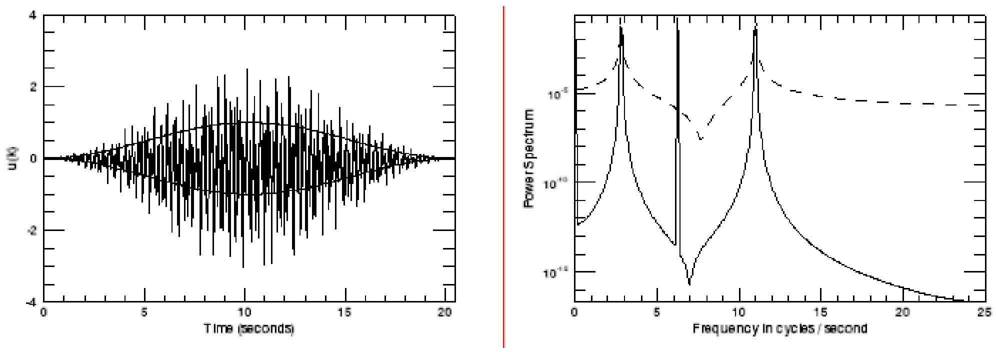


Figure 6-5: Time Series Multiplied by Hanning Window (Left) and Power Spectrum (Right) with Hanning Window (Solid) and without (Dashed)

Example Code

Type `@sigprc06` at the IDL prompt to run the batch file that creates this display. The source code is located in `sigprc06`, in the `examples/doc/signal` directory. See "Running the Example Code" on page 138 if IDL does not find the batch file.

Hamming Window

The Hamming window is defined as:

$$w(k) = 0.54 - 0.46\cos\left(\frac{2\pi k}{N}\right)$$

The resulting vector is multiplied element-by-element with the sampled signal vector before applying the FFT. For example, the following IDL command computes the Hamming window and then applies the FFT function:

```
v_m = FFT(HANNING(N, ALPHA=0.56)*U)
```

The power spectrum of the Hamming windowed signal shows the mitigation of the truncation effect (see the figure below).

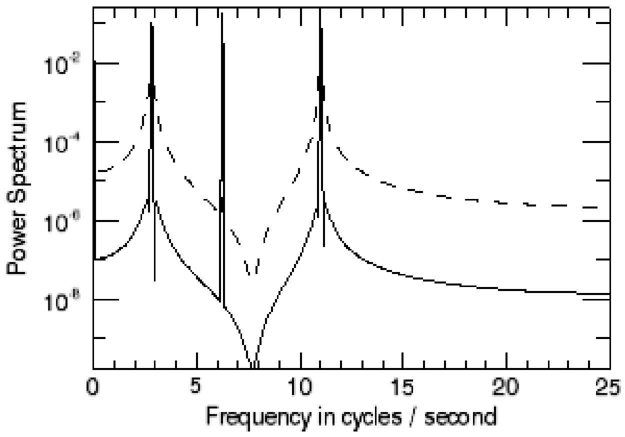


Figure 6-6: Power Spectrum with Hamming Window (Solid) and without (Dashed)

Example Code

Type `@sigprc07` at the IDL prompt to run the batch file that creates this display. The source code is located in `sigprc07`, in the `examples/doc/signal` directory. See "Running the Example Code" on page 138 if IDL does not find the batch file.

Aliasing

Aliasing is a well known phenomenon in sampled data analysis. It occurs when the signal being sampled has components at frequencies higher than the Nyquist frequency, which is equal to half the sampling frequency. Aliasing is a consequence of the fact that after sampling, every periodic signal at a frequency greater than the Nyquist frequency looks exactly like some other periodic signal at a frequency less than the Nyquist frequency. For example, suppose we add a 30 cycle per second periodic component to our sampled data sequence *u*(*t*). The power spectrum of the augmented signal appears below.

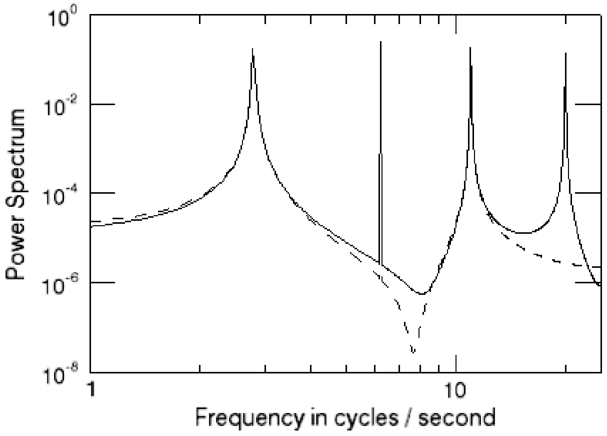


Figure 6-7: Power Spectrum of the Sample Signal After Adding a 30 Cycles per Second Component

Because the frequency of the new component is above the Nyquist frequency of 25 cycles per second (25 = 1/(2*delt)), the power spectrum shows the contribution of the new component as an alias at 20 cycles per second. To prevent aliasing, frequency components of a signal above the Nyquist frequency must be removed before sampling.

Example Code

Type `@sigprc08` at the IDL prompt to run the batch file that creates this display. The source code is located in `sigprc08`, in the `examples/doc/signal` directory. See "Running the Example Code" on page 138 if IDL does not find the batch file.

FFT Algorithm Details

IDL's implementation of the fast Fourier transform is based on the Cooley-Tukey algorithm. The algorithm takes advantage of the fact that the discrete Fourier transform (DFT) of a discrete time series with an even number of points is equal to the sum of two DFTs, each half the length of the original. For data lengths that are a power of 2, this algorithm is used recursively, each iteration subdividing the data into smaller sets to be transformed. In the IDL FFT, this method is also extended to powers of 3 and 5. If the number of points in the original time series does not contain powers of 2, 3, or 5, the original data are still subdivided into data sets with lengths equal to the prime factors of N. The resulting subdivisions with lengths equal to prime numbers other than 2, 3, or 5 must be transformed using a slow DFT. The slow DFT is mathematically equivalent to the FFT, but requires N^2 operations instead of $N\log2(N)$.

This implementation means that the FFT function is fastest when the number of points is rich in powers of 2, 3, or 5. The slowest case is when the number of samples is a large prime number. In this case, a significant improvement in efficiency can be gained by padding the data set with zeros to increase the number of data points to a power of 2, 3, or 5.

For real input data of even lengths, the FFT algorithm also takes advantage of the fact that the real array can be packed into a complex array of half the length, and unpacked at the end, thus cutting the running time in half.

The Hilbert Transform

The Hilbert transform is a time-domain to time-domain transformation which shifts the phase of a signal by 90 degrees. Positive frequency components are shifted by +90 degrees, and negative frequency components are shifted by – 90 degrees. Applying a Hilbert transform to a signal twice in succession shifts the phases of all of the components by 180 degrees, and so produces the negative of the original signal. IDL’s HILBERT function accepts both real and complex valued signals as inputs; the imaginary part of the result is zero for real inputs.

In optics and signal analysis, the Hilbert transform of the time signal *r*(*t*) is known as the quadrature function of *r*(*t*), which is used to form a complex function known as the analytic signal. The analytic signal is defined as:

$$\hat{r}(t) = r(t) - jH(r(t))$$

where *j* is the square root of –1 and *H* is the Hilbert function.

The projection of the analytic signal onto the plane defined by the real axis and the time axis is the original signal. The projection onto the plane defined by the imaginary axis and the time axis is the Hilbert transform of the original signal.

The following example plots the complex analytic signal of a periodic time signal with a slowly varying amplitude.

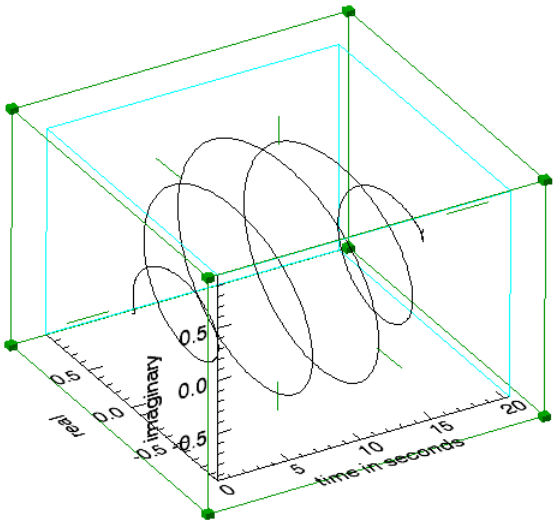


Figure 6-8: Analytic Signal for r(t)

Example Code

Type `@sigprc09` at the IDL prompt to run the batch file that creates this display. The source code is located in `sigprc09`, in the `examples/doc/signal` directory. See “Running the Example Code” on page 138 if IDL does not find the batch file.

The Wavelet Transform

Like the discrete Fourier transform, the discrete wavelet transform (DWT) is a linear operation that defines a forward and inverse relationship between the time-domain and the frequency-domain, also called the wavelet domain. This relationship is expressed through the use of basis functions. In the case of the DFT, trigonometric sines and cosines of varying angles are used. In the case of the DWT, the basis functions are more complicated and usually called mother functions or wavelets. Also like the DFT, the DWT is orthogonal, making many operations computationally efficient. For example, the inverse wavelet transform, when viewed as a matrix operator, is simply the transpose of the forward transform.

Most of the usefulness of wavelets relies on the fact that wavelet transforms can usefully be severely truncated—that is, they can be effectively turned into sparse expressions. This property is a result of the simultaneous compact representation of the wavelet basis functions in the time and frequency domains. See "WTN" (*IDL Reference Guide*) for an example using the wavelet transform. Also see "Wavelet Toolkit" (*IDL Quick Reference*) for a brief description of the available wavelet routines.

Convolution

Discrete convolution in digital signal processing is used (among other things) to smooth sampled signals using a weighted moving average. It also has many applications outside of signal processing.

IDL has two functions for doing discrete convolution: BLK_CON and CONVOL. BLK_CON takes advantage of the fact that the convolution of two signals is the Inverse Fourier transform of the product of the Fourier transforms of the two signals. BLK_CON is faster than CONVOL, but not as flexible. Among the many applications for discrete convolution is the implementation of digital filters. See the example in the "Finite Impulse Response (FIR) Filters" on page 159.

Correlation and Covariance

Correlation and covariance (which is correlation with any non-zero mean values of the signals removed beforehand) are closely related to convolution. They are useful in analyzing signals with random components. Autocorrelation and autocovariance of signals are computed with the A_CORRELATE function, and crosscorrelation and crosscovariance are computed with the C_CORRELATE function. See "Time-Series Analysis" on page 204 for details.

Digital Filtering

Digital filters can be implemented on a computer to remove unwanted frequency components (noise) from a sampled signal. Two broad classes of filters are Finite Impulse Response (FIR) or Moving Average (MA) filters, and Infinite Impulse Response (IIR) or AutoRegressive Moving Average (ARMA) filters. Both of these classes of filters are described in the following sections:

- "Finite Impulse Response (FIR) Filters" on page 159
- "Infinite Impulse Response (IIR) Filters" on page 163

Note

IDL's IR_FILTER function filters data with an infinite impulse response (IIR) or finite impulse response (FIR) filter. See "IR_FILTER" (*IDL Reference Guide*) for more information.

Finite Impulse Response (FIR) Filters

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

IDL's DIGITAL_FILTER function computes the impulse response of an FIR filter based on Kaiser's window, which in turn is based on the modified Bessel function. The Kaiser filter is "nearly optimum in the sense of having the largest energy in the mainlobe for a given peak sidelobe level" [Jackson, Leland B., *Digital Filters and Signal Processing*]. The DIGITAL_FILTER function constructs lowpass, highpass, bandpass, or bandstop filters. The figure below plots a bandstop filter which suppresses frequencies between 7 cycles per second and 15 cycles per second for data sampled every 0.02 seconds.

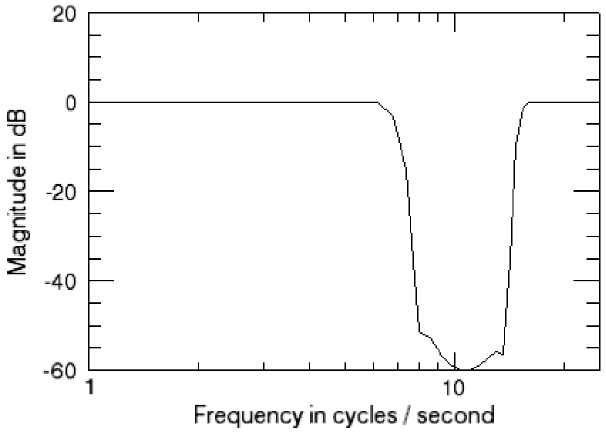


Figure 6-9: Bandstop FIR Filter

Example Code

Type `@sigprc10` at the IDL prompt to run the batch file that creates this display. The source code is located in `sigprc10`, in the `examples/doc/signal` directory. See "Running the Example Code" on page 138 if IDL does not find the batch file.

Other FIR filters can be designed based on the Hanning and Hamming windows (see "Using Windows" on page 148), or any other user-defined window function. The design procedure is simple:

1. Compute the impulse response of an ideal filter using the inverse FFT.
2. Apply a window to the impulse response. The modified impulse response defines the FIR filter.

The figure below shows the plot using the same sampling period and frequency cutoffs as above, and the corresponding ideal filter is constructed in the frequency domain using the Hanning window.

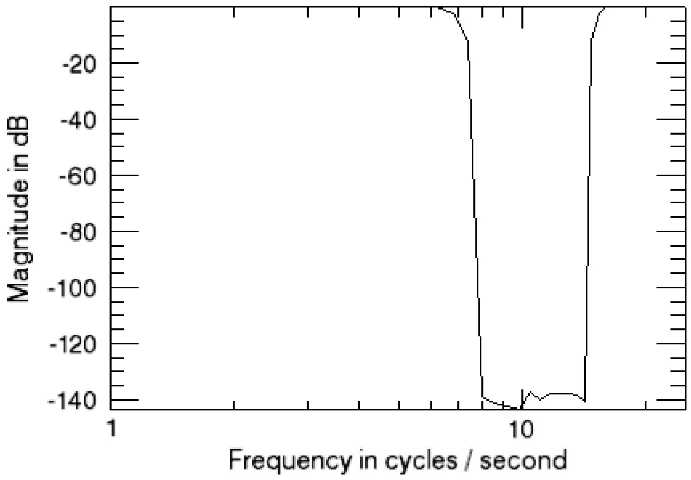


Figure 6-10: Bandstop Filter Using Hanning Window

Example Code

Type `@sigprc11` at the IDL prompt to run the batch file that creates this display. The source code is located in `sigprc11`, in the `examples/doc/signal` directory. See "Running the Example Code" on page 138 if IDL does not find the batch file.

FIR Filter Implementation

The simplest FIR (Finite Impulse Response) filter to apply to a signal is the rectangular or boxcar filter, which is implemented with IDL's SMOOTH function, or the closely related MEDIAN function.

Applying other FIR filters to signals is straightforward since the filter is non-recursive. The filtered signal is simply the convolution of the impulse response of the filter with the original signal. The impulse response of the filter is computed with the DIGITAL_FILTER function or by the procedure in the previous section.

IDL's BLK_CON function provides a simple and efficient way to convolve a filter with a signal. Using *u*(*k*) from the previous example and the bandstop filter created above creates the plot shown in the figure below.

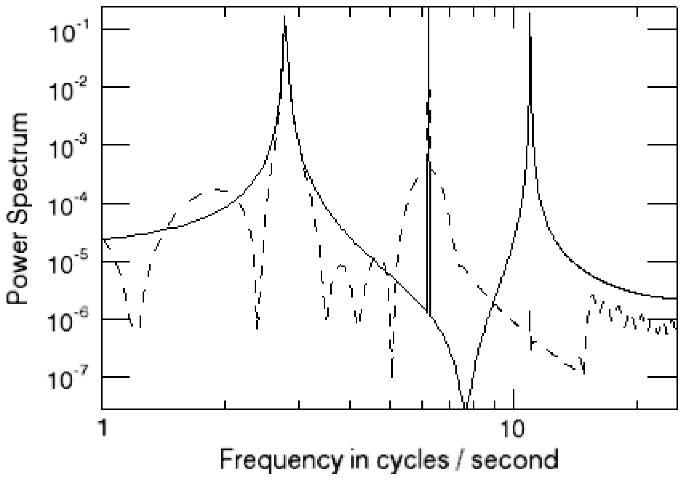


Figure 6-11: Digital Signal Before and After Filtering

Example Code

Type `@sigprc12` at the IDL prompt to run the batch file that creates this display. The source code is located in `sigprc12`, in the `examples/doc/signal` directory. See "Running the Example Code" on page 138 if IDL does not find the batch file.

The frequency response of the filtered signal shows that the frequency component at 11.0 cycles / second has been filtered out, while the frequency components at 2.8 and 6.25 cycles / second, as well as the DC component, have been passed by the filter.

Infinite Impulse Response (IIR) Filters

Digital filters which must be implemented recursively are called Infinite Impulse Response (IIR) filters because, theoretically, the response of these filters to an impulse never settles to zero. In practice, the impulse response of many IIR filters approaches zero asymptotically, and may actually reach zero in a finite number of samples due to the finite word length of digital computers.

One method of designing digital filters starts with the Laplace transform representation of an analog filter with the required frequency response. For example, the Laplace transform representation (or continuous transfer function) of a second order notch filter with the notch at f_0 cycles per second is:

$$\frac{y(s)}{u(s)} = \frac{\left(\frac{f_0}{2\pi} + s^2\right)}{\left(1 + 2s\left(\frac{f_0}{2\pi}\right) + s^2\right)}$$

where *s* is the Laplace transform variable. Then the continuous transfer function is converted to the equivalent discrete transfer function using one of several techniques. One of these is the bilinear (Tustin) transform, where

```
(2/δ)*(z-1)/(z+1)
```

is substituted for the Laplace transform variable *s*. In this expression, *z* is the unit delay operator.

For the notch filter above, the bilinear transformation yields the following discrete transfer function:

$$\frac{y(z)}{u(z)} = \frac{\left(\frac{1+c^2}{2} - 2cz + \frac{1+c^2}{2}z^2\right)}{(c^2 - 2cz + z^2)}$$

where $c = (1 - \pi * f_0 * \delta) / (1 + \pi * f_0 * \delta)$.

Enter the following IDL statements to compute the coefficients of the discrete transfer function:

```
delt = 0.02
; Notch frequency in cycles per second:
f0 = 6.5
c = (1.0-!PI*F0*delt) / (1.0+!PI*F0*delt)
```

```
b = [(1+c^2)/2, -2*c, (1+c^2)/2]
a = [ c^2, -2*c, 1]
```

Example Code

Alternately, type `@sigprc13` at the IDL prompt to run the `sigprc13` batch file and create the plot variables. See "Running the Example Code" on page 138 if IDL does not find the batch file.

IIR Filter Implementation

Since an Infinite Impulse Response filter contains feedback loops, its output at every time step depends on previous outputs, and the filter must be implemented recursively with difference equations. The discrete transfer function

$$y(z) = \left(\frac{b_0 + b_1 z + \ldots + b_{nb} z^{nb}}{a_0 + a_1 z + \ldots + a_{na} z^{nb}}\right) u(z)$$

is implemented with the difference equation

$$y(k) = \frac{(b_0 u(k-nb) + b_1 u(k-nb+1) + \ldots + b_{nb} u(k) - a_0 y(k-na) - a_1 y(k-na+1) - \ldots - a_{na-1} y(k-1))}{a_{na}}$$

An IIR filter is stable if the absolute values of the roots of the denominator of the discrete transfer function *a*(*z*) are all less than one. The impulse response of a stable IIR filter approaches zero as the time index *k* approaches infinity. The frequency response function of a stable IIR filter is the Discrete Fourier Transform of the filter's impulse response.

The figure below plots the impulse and frequency response functions of the notch filter defined above using recursive difference equations.

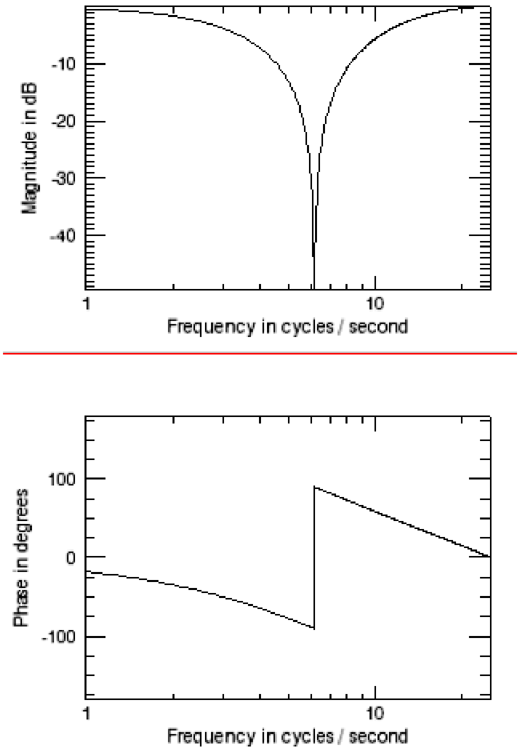


Figure 6-12: Impulse and Frequency Response of a Notch Filter

Example Code

Type `@sigprc14` at the IDL prompt to run the batch file that creates this display. The source code is located in `sigprc14`, in the `examples/doc/signal` directory. See "Running the Example Code" on page 138 if IDL does not find the batch file.

Note

Because the impulse response approaches zero, IDL may warn of floating-point underflow errors. This is an expected consequence of the digital implementation of an Infinite Impulse Response filter.

The same code could be used to filter any input sequence $u(k)$.

References

Bracewell, Ronald N., *The Fourier Transform and Its Applications*, New York: McGraw-Hill, 1978. ISBN 0-07-007013-X

Chen, Chi-Tsong, *One-Dimensional Digital Signal Processing*, New York: Marcel Dekker, Inc., 1979. ISBN 0-8247-6877-9

Jackson, Leland B., *Digital Filters and Signal Processing*, Boston: Kluwer Academic Publishers, 1986. ISBN 0-89838-174-6

Mayeda, Wataru, *Digital Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1993. ISBN 0-13-211301-5

Morgera, Salvatore D. and Krishna, Hari, *Digital Signal Processing: Applications to Communications and Algebraic Coding Theories*, Boston: Academic Press, 1989. ISBN 0-12-506995-2

Oppenheim, Alan V. and Schafer, Ronald W., *Discrete-time signal processing*, Englewood Cliffs, NJ: Prentice-Hall, 1989. ISBN 0-13-216292-X

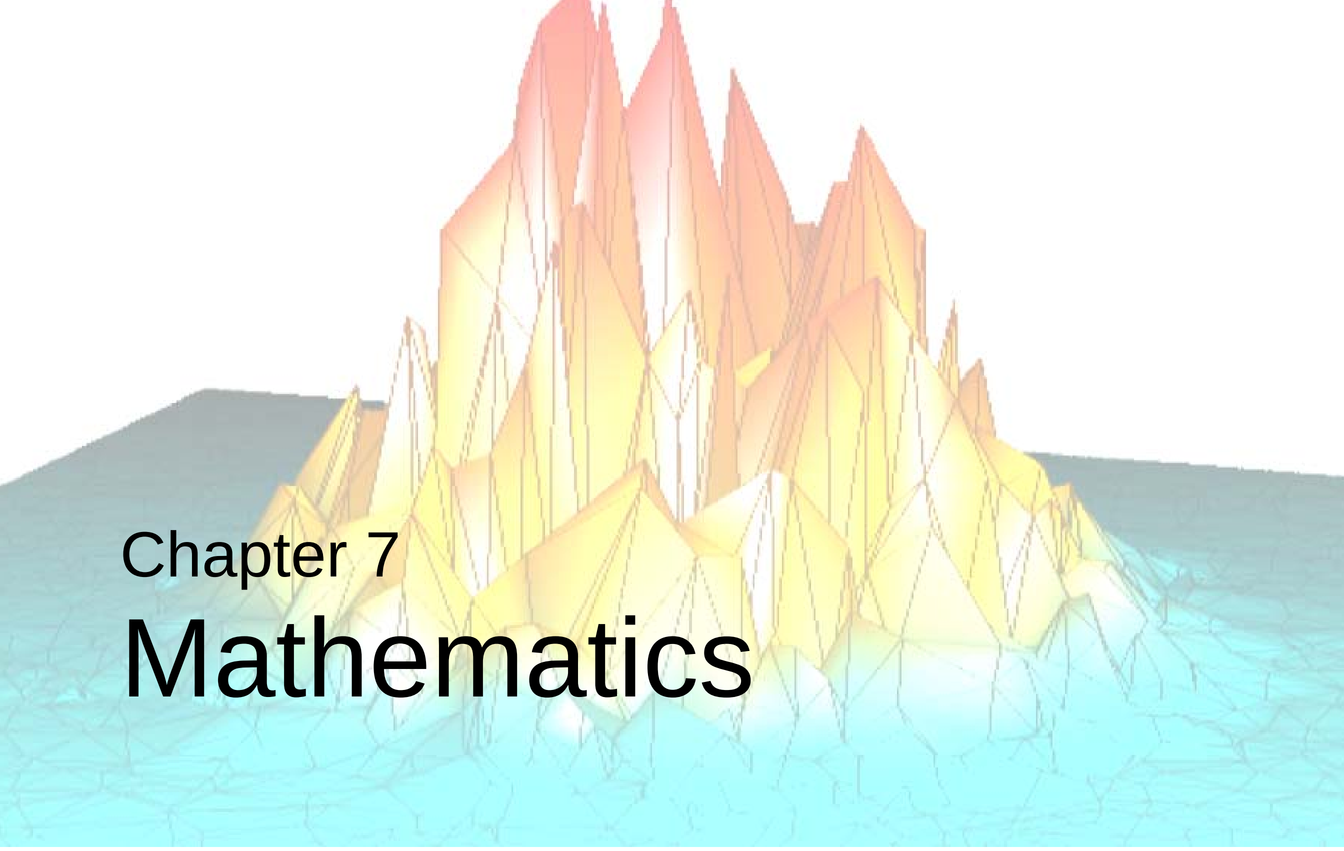
Peled, Abraham and Liu, Bede, *Digital Signal Processing*, New York: John Wiley & Sons, Inc., 1976. ISBN 0-471-01941-0

Press, William H. et al. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Proakis, John G. and Manolakis, Dimitris G., *Digital Signal Processing: Principles, Algorithms, and Applications*, New York: Macmillan Publishing Company, 1992. ISBN 0-02-396815-X

Rabiner, Lawrence R. and Gold, Bernard, *Theory and application of digital signal processing*, Englewood Cliffs, NJ: Prentice-Hall, 1975. ISBN 0-139-14101-4

Strang, Gilbert and Nguyen, Truong, *Wavelets and Filter Banks*, Wellesley, MA: Wellesley-Cambridge Press, 1996. ISBN 0-961-40887-1



Chapter 7

Mathematics

The following topics are covered in this chapter:

Overview of Mathematics in IDL

This chapter documentsIDL's mathematics and statistics procedures and functions. These include Numerical Recipes™ algorithms published in *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition). For a list of IDL mathematical routines, see the functional category of "Mathematics" (*IDL Quick Reference*). There you will find a brief introduction to the routines. Detailed information is available in the *IDL Reference Guide*. This chapter also includes introductory discussions of the following topics and an overview of the way IDL handles the particular problems involved:

- "Correlation Analysis" on page 170
- "Curve and Surface Fitting" on page 174
- "Eigenvalues and Eigenvectors" on page 176
- "Gridding and Interpolation" on page 182
- "Hypothesis Testing" on page 183
- "Integration" on page 185
- "Linear Systems" on page 190
- "Nonlinear Equations" on page 197
- "Optimization" on page 199
- "Sparse Arrays" on page 201
- "Time-Series Analysis" on page 204
- "Multivariate Analysis" on page 207

References are provided at the end of each section for a more detailed description and understanding of the topic.

ITT Visual Information Solutions is extremely interested in the accuracy of its algorithms. Bug reports, documentation errors and suggestions for mathematics and statistics enhancements can be sent to ITT Visual Information Solutions via:

Internet: `support@ittvis.com`

Fax: (303) 786-9909

Note

Floating-point numbers are inherently inaccurate. See "Accuracy and Floating Point Operations" on page 274 for details on roundoff and truncation errors.

IDL's Numerical Recipes Functions

IDL includes a number of routines based on algorithms published in *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition). Routines derived from Numerical Recipes are noted as such in the *IDL Reference Guide* and in the IDL Online Help.

In IDL versions up to and including IDL version 3.6, mathematics functions based on Numerical Recipes algorithms required that input be in column-major format. This is no longer the case. Routines based on Numerical Recipes algorithms have been reworked and renamed, so that all IDL functions now expect input arrays to be in row-major format (composed of row vectors).

Note

To maintain compatibility with IDL programs based on earlier versions, the old routines (using the older input convention) are still available. No alterations need be made to existing code as a result of this change in IDL. We recommend that all new IDL programs take advantage of the new names and input convention.

Correlation Analysis

Given two *n*-element sample populations, *X* and *Y*, it is possible to quantify the degree of fit to a linear model using the correlation coefficient. The correlation coefficient, *r*, is a scalar quantity in the interval [-1.0, 1.0], and is defined as the ratio of the covariance of the sample populations to the product of their standard deviations.

$$r = \frac{\text{covariance of X and Y}}{(\text{standard deviation of X})(\text{standard deviation of Y})}$$

or

$$r = \frac{\dfrac{1}{N-1}\displaystyle\sum_{i=0}^{N-1}\left(x_i - \left[\sum_{k=0}^{N-1}\frac{x_k}{N}\right]\right)\left(y_i - \left[\sum_{k=0}^{N-1}\frac{y_k}{N}\right]\right)}{\sqrt{\dfrac{1}{N-1}\displaystyle\sum_{i=0}^{N-1}\left(x_i - \left[\sum_{k=0}^{N-1}\frac{x_k}{N}\right]\right)^2}\sqrt{\dfrac{1}{N-1}\displaystyle\sum_{i=0}^{N-1}\left(y_i - \left[\sum_{k=0}^{N-1}\frac{y_k}{N}\right]\right)^2}}$$

The correlation coefficient is a direct measure of how well two sample populations vary jointly. A value of $r = +1$ or $r = –1$ indicates a perfect fit to a positive or negative linear model, respectively. A value of *r* close to +1 or –1 indicates a high degree of correlation and a good fit to a linear model. A value of *r* close to 0 indicates a poor fit to a linear model.

Correlation Example

The following sample populations represent a perfect positive linear correlation.

```
X = [-8.1,  1.0, -14.3, 4.2, -10.1, 4.3, 6.3, 5.0, 15.1, -2.2]
Y = [-9.8, -0.7, -16.0, 2.5, -11.8, 2.6, 4.6, 3.3, 13.4, -3.9]
;Compute the correlation coefficient of X and Y.
PRINT, CORRELATE(X, Y)
```

IDL prints:

```
1.00000
```

The following sample populations represent a high negative linear correlation.

```
X = [ 1.8, -2.7,  0.7, -0.5, -1.3, -0.9,  0.6, -1.5,  2.5,  3.0]
Y = [-4.7,  9.8, -3.7,  2.8,  5.1,  3.9, -3.6,  5.8, -7.3, -7.4]
;Compute the correlation coefficient of X and Y:
PRINT, CORRELATE(X, Y)
```

IDL prints:

```
-0.979907
```

The following sample populations represent a poor linear correlation.

```
X = [-1.8,  0.1, -0.1, 1.9, 0.5,  1.1, 1.9,  0.3, -0.2, -1.0]
Y = [ 1.5, -1.0, -0.6, 1.1, 0.7, -0.7, 1.1, -0.1,  0.6, -0.1]
;Compute the correlation coefficient of X and Y:
PRINT, CORRELATE(X, Y)
```

IDL prints:

```
0.0322859
```

Notes on Interpreting the Correlation Coefficient

When interpreting the value of the correlation coefficient, it is important to remember the following two caveats:

1. Although a high degree of correlation (a value close to +1 or –1) indicates a good mathematical fit to a linear model, its applied interpretation may be completely nonsensical. For example, there may be a high degree of correlation between the number of scientists using IDL to study atmospheric phenomena and the consumption of alcohol in Russia, but the two events are clearly unrelated.
2. Although a correlation coefficient close to 0 indicates a poor fit to a linear model, it does not mean that there is no correlation between the two sample populations. It is possible that the relationship between *X* and *Y* is accurately described by a nonlinear model. See "Curve and Surface Fitting" on page 174 for further details on fitting data to linear and nonlinear models.

Multiple Linear Models

The fundamental principles of correlation that apply to the linear model of two sample populations may be extended to the multiple-linear model. The degree of relationship between three or more sample populations may be quantified using the multiple correlation coefficient. The degree of relationship between two sample populations when the effects of all other sample populations are removed may be quantified using the partial correlation coefficient. Both of these coefficients are scalar quantities in the interval [0.0, 1.0]. A value of +1 indicates a perfect linear relationship between populations. A value close to +1 indicates a high degree of linear relationship between populations; whereas a value close to 0 indicates a poor linear relationship between populations. (Although a value of 0 indicates no linear relationship between populations, remember that there may be a nonlinear relationship.)

Partial Correlation Example

Define the independent (*X*) and dependent (*Y*) data.

```
X = [[0.477121, 2.0, 13.0], $
   [0.477121, 5.0, 6.0], $
   [0.301030, 5.0, 9.0], $
   [0.000000, 7.0, 5.5], $
   [0.602060, 3.0, 7.0], $
   [0.698970, 2.0, 9.5], $
   [0.301030, 2.0, 17.0], $
   [0.477121, 5.0, 12.5], $
   [0.698970, 2.0, 13.5], $
   [0.000000, 3.0, 12.5], $
   [0.602060, 4.0, 13.0], $
   [0.301030, 6.0, 7.5], $
   [0.301030, 2.0, 7.5], $
   [0.698970, 3.0, 12.0], $
   [0.000000, 4.0, 14.0], $
   [0.698970, 6.0, 11.5], $
   [0.301030, 2.0, 15.0], $
   [0.602060, 6.0, 8.5], $
   [0.477121, 7.0, 14.5], $
   [0.000000, 5.0, 9.5]]
Y = [97.682, 98.424, 101.435, 102.266, 97.067, 97.397, $
   99.481, 99.613, 96.901, 100.152, 98.797, 100.796, $
   98.750, 97.991, 100.007, 98.615, 100.225, 98.388, $
   98.937, 100.617]
```

Compute the multiple correlation of *Y* on the first column of *X*. The result should be 0.798816.

```
PRINT, M_CORRELATE(X[0,*], Y)
```

IDL prints:

```
0.798816
```

Compute the multiple correlation of *Y* on the first two columns of *X*. The result should be 0.875872.

```
PRINT, M_CORRELATE(X[0:1,*], Y)
```

IDL prints:

```
0.875872
```

Compute the multiple correlation of *Y* on all columns of *X*. The result should be 0.877197.

```
PRINT, M_CORRELATE(X, Y)
```

IDL prints:

```
0.877197
;Define the five sample populations.
X0 = [30, 26, 28, 33, 35, 29]
X1 = [0.29, 0.33, 0.34, 0.30, 0.30, 0.35]
X2 = [65, 60, 65, 70, 70, 60]
X3 = [2700, 2850, 2800, 3100, 2750, 3050]
Y  = [37, 33, 32, 37, 36, 33]
```

Compute the partial correlation of *X*1 and *Y* with the effects of *X*0, *X*2 and *X*3 removed.

```
PRINT, P_CORRELATE(X1, Y, REFORM([X0,X2,X3], 3, N_ELEMENTS(X1)))
```

IDL prints:

```
0.996017
```

Routines for Computing Correlations

See "Correlation Analysis" (in the functional category "Mathematics" (*IDL Quick Reference*)) for a brief description of IDL routines for computing correlations. Detailed information is available in the *IDL Reference Guide*.

Curve and Surface Fitting

The problem of curve fitting may be stated as follows:

Given a tabulated set of data values $\{x_i, y_i\}$ and the general form of a mathematical model (a function $f(x)$ with unspecified parameters), determine the parameters of the model that minimize an error criterion. The problem of surface fitting involves tabulated data of the form $\{x_i, y_i, z_i\}$ and a function $f(x, y)$ of two spatial dimensions.

For example, we can use the CURVEFIT routine to determine the parameters A and B of a user-supplied function $f(x)$, such that the sums of the squares of the residuals between the tabulated data $\{x_i, y_i\}$ and function are minimized. We will use the following function and data:

$f(x) = a(1 - e^{-bx})$

$x_i = [0.25, 0.75, 1.25, 1.75, 2.25]$

$y_i = [0.28, 0.57, 0.68, 0.74, 0.79]$

First we must provide a procedure written in IDL to evaluate the function, f, and its partial derivatives with respect to the parameters a_0 and a_1:

```
PRO funct, X, A, F, PDER
   F = A[0] * (1.0 - EXP(-A[1] * X))
   ; If the function is called with four parameters,
   ; calculate the partial derivatives:
   IF N_PARAMS() GE 4 THEN BEGIN
      ; PDER’s column dimension is equal to the number of
      ; elements in xi and its row dimension is equal to
      ; the number of parameters in the function F:
      pder = FLTARR(N_ELEMENTS(X), 2)
      ; Compute the partial derivatives with respect to
      ; a0 and place in the first row of PDER:
      pder[*, 0] = 1.0 - EXP(-A[1] * X)
      ; Compute the partial derivatives with respect to
      ; a1 and place in the second row of PDER:
      pder[*, 1] = A[0] * x * EXP(-A[1] * X)
  ENDIF
END
```

Note

The function will not calculate the partial derivatives unless it is called with four parameters. This allows the calling routine (in this case CURVEFIT) to avoid the extra computation in cases when the partial derivatives are not needed.

Next, we can use the following IDL commands to find the function’s parameters:

```
;Define the vectors of tabulated:
X = [0.25, 0.75, 1.25, 1.75, 2.25]
;data values:
Y = [0.28, 0.57, 0.68, 0.74, 0.79]
;Define a vector of weights:
W = 1.0 / Y
;Provide an initial guess of the function’s parameters:
A = [1.0, 1.0]
;Compute the parameters a0 and a1:
yfit = CURVEFIT(X, Y, W, A, SIGMA_A, FUNCTION_NAME = 'funct')
;Print the parameters, which are returned in A:
PRINT, A
```

IDL prints:

```
  0.787386  1.71602
```

Thus the nonlinear function that best fits the data is:

$$f(x) = 0.787386 (1 - e^{-1.71602x})$$

Routines for Curve and Surface Fitting

See “Curve and Surface Fitting” (in the functional category “Mathematics” (*IDL Quick Reference*)) for a brief description of IDL routines for curve and surface fitting. Detailed information is available in the *IDL Reference Guide*.

Eigenvalues and Eigenvectors

Consider a system of equations that satisfies the array-vector relationship $Ax = \lambda x$, where A is an n-by-n array, x is an n-element vector, and λ is a scalar. A scalar λ and nonzero vector x that simultaneously satisfy this relationship are referred to as an eigenvalue and an eigenvector of the array A, respectively. The set of all eigenvectors of the array A is then referred to as the eigenspace of A. Ideally, the eigenspace will consist of n linearly-independent eigenvectors, although this is not always the case.

IDL computes the eigenvalues and eigenvectors of a real symmetric n-by-n array using Householder transformations and the QL algorithm with implicit shifts. The eigenvalues of a real, n-by-n nonsymmetric array are computed from the upper Hessenberg form of the array using the QR algorithm. Eigenvectors are computed using inverse subspace iteration.

Although it is not practical for numerical computation, the problem of computing eigenvalues and eigenvectors can also be defined in terms of the determinant function. The eigenvalues of an n-by-n array A are the roots of the polynomial defined by $\det(A - \lambda \mathrm{I})$, where I is the identity matrix (an array with 1s on the main diagonal and 0s elsewhere) with the same dimensions as A. By expressing eigenvalues as the roots of a polynomial, we see that they can be either real or complex. If an eigenvalue is complex, its corresponding eigenvectors are also complex.

The following examples demonstrate how to use IDL to compute the eigenvalues and eigenvectors of real, symmetric and nonsymmetric n-by-n arrays. Note that it is possible to check the accuracy of the computed eigenvalues and eigenvectors by algebraically manipulating the definition given above to read $Ax - \lambda x = 0$; in this case 0 denotes an n-element vector, all elements of which are zero.

Symmetric Array with *n* Distinct Real Eigenvalues

To compute eigenvalues and eigenvectors of a real, symmetric, n-by-n array, begin with a symmetric array A.

Note

The eigenvalues and eigenvectors of a real, symmetric n-by-n array are real numbers.

```
A = [[ 3.0,  1.0, -4.0], $
     [ 1.0,  3.0, -4.0], $
     [-4.0, -4.0,  8.0]]
```

```
; Compute the tridiagonal form of A:
TRIRED, A, D, E
; Compute the eigenvalues (returned in vector D) and
; the eigenvectors (returned in the rows of the array A):
TRIQL, D, E, A
; Print eigenvalues:
PRINT, D
```

IDL prints:

```
  2.00000  4.76837e-07  12.0000
```

The exact values are: [2.0, 0.0, 12.0].

```
;Print the eigenvectors, which are returned as row vectors in A:
PRINT, A
```

IDL prints:

```
  0.707107  -0.707107   0.00000
 -0.577350  -0.577350  -0.577350
 -0.408248  -0.408248   0.816497
```

The exact eigenvectors are:

$$\begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ -1/\sqrt{3} & -1/\sqrt{3} & -1/\sqrt{3} \\ -1/\sqrt{6} & -1/\sqrt{6} & 2/\sqrt{6} \end{bmatrix}$$

Nonsymmetric Array with *n* Distinct Real and Complex Eigenvalues

To compute the eigenvalues and eigenvectors of a real, nonsymmetric *n*-by-*n* array, begin with an array *A*. In this example, there are *n* distinct eigenvalues and *n* linearly-independent eigenvectors.

```
A = [[ 1.0, 0.0,  2.0], $
     [ 0.0, 1.0, -1.0], $
     [-1.0, 1.0,  1.0]]
; Reduce to upper Hessenberg format:
hes = ELMHES(A)
; Compute the eigenvalues:
evals = HQR(hes)
; Print the eigenvalues:
PRINT, evals
```

IDL prints:

```
(  1.00000, -1.73205)(  1.00000,  1.73205)
(  1.00000,  0.00000)
```

Note

The three eigenvalues are distinct, and that two are complex. Note also that complex eigenvalues of an *n*-by-*n* real, nonsymmetric array always occur in complex conjugate pairs.

```
; Initialize a variable to contain the residual:
residual = 1
; Compute the eigenvectors and the residual for each
; eigenvalue/eigenvector pair, using double-precision arithmetic:
evecs = EIGENVEC(A, evals, /DOUBLE, RESIDUAL=residual)
; Print the eigenvectors, which are returned as
; row vectors in evecs:
PRINT, evecs[*,0]
```

IDL prints:

```
(  0.68168704,  0.18789033)( -0.34084352, -0.093945164)
(  0.16271780, -0.59035830)
PRINT, evecs[*,1]
```

IDL prints:

```
(  0.18789033,  0.68168704)( -0.093945164, -0.34084352)
( -0.59035830,  0.16271780)
PRINT, evecs[*,2]
```

IDL prints:

```
(  0.70710678,  0.0000000)(  0.70710678,  0.0000000)
( -2.3570226e-21, 0.0000000)
```

We can check the accuracy of these results using the relation $Ax - \lambda x = 0$. The array contained in the variable specified by the RESIDUAL keyword contains the result of this computation.

```
PRINT, residual
```

IDL prints:

```
( -1.2021898e-07,  1.1893681e-07)(  6.0109490e-08, -5.9468404e-08)
(  1.0300230e-07,  1.0411269e-07)
(  1.1893681e-07, -1.2021898e-07)( -5.9468404e-08,  6.0109490e-08)
(  1.0411269e-07,  1.0300230e-07)
(      0.0000000,      0.0000000)(      0.0000000,      0.0000000)
```

The results are all zero to within machine precision.

Repeated Eigenvalues

To compute the eigenvalues and eigenvectors of a real, nonsymmetric *n*-by-*n* array, begin with an array *A*. In this example, there are fewer than *n* distinct eigenvalues, but *n* independent eigenvectors are available.

```
A = [[8.0, 0.0, 3.0], $
     [2.0, 2.0, 1.0], $
     [2.0, 0.0, 3.0]]
; Reduce A to upper Hessenberg form and compute the eigenvalues.
; Note that both operations can be combined into a single command.
evals = HQR(ELMHES(A))
; Print the eigenvalues:
PRINT, evals
```

IDL prints:

```
(  9.00000,  0.00000) (  2.00000,  0.00000)
(  2.00000,  0.00000)
```

Note

The three eigenvalues are real, but only two are distinct.

```
; Initialize a variable to contain the residual:
residual = 1
; Compute the eigenvectors and residual, using
; double-precision arithmetic:
evecs = EIGENVEC(A, evals, /DOUBLE, RESIDUAL=residual)
; Print the eigenvectors:
PRINT, evecs[*,0]
```

IDL prints:

```
(  0.90453403,  0.0000000)(  0.30151134,  0.0000000)
(  0.30151134,  0.0000000)
PRINT, evecs[*,1]
```

IDL prints:

```
( -0.27907279,  0.0000000)( -0.78140380,  0.0000000)
(  0.55814557,  0.0000000)
PRINT, evecs[*,2]
```

IDL prints:

```
( -0.27907279,  0.0000000)( -0.78140380,  0.0000000)
(  0.55814557,  0.0000000)
```

We can compute an independent eigenvector for the repeated eigenvalue (2.0) by perturbing it slightly, allowing the algorithm `EIGENVEC` to recognize the eigenvalue as distinct and to compute a linearly-independent eigenvector.

```
newresidual = 1
evecs[*,2] = EIGENVEC(A, evals[2]+1.0e-6, /DOUBLE, $
   RESIDUAL = newresidual)
PRINT, evecs[*,2]
```

IDL prints:

```
( -0.33333333,  0.0000000)(  0.66666667,  0.0000000)
(  0.66666667,  0.0000000)
```

Once again, we can check the accuracy of these results by checking that each element in the residuals —for both the original eigenvectors and the perturbed eigenvector—is zero to within machine precision.

The So-called Defective Case

In the so-called defective case, there are fewer than *n* distinct eigenvalues and fewer than *n* linearly-independent eigenvectors. Begin with an array *A*:

```
A = [[2.0, -1.0], $
     [1.0,  0.0]]
; Reduce A to upper Hessenberg form and compute the eigenvalues.
; Note that both operations can be combined into a single command.
evals = HQR(ELMHES(A))
; Print the eigenvalues:
PRINT, evals
```

IDL prints:

```
(  1.00000,  0.00000)(  1.00000,  0.00000)
```

Note

The two eigenvalues are real, but not distinct.

```
; Compute the eigenvectors, using double-precision arithmetic:
evecs = EIGENVEC(A, evals, /DOUBLE)
; Print the eigenvectors:
PRINT, evecs[*,0]
```

IDL prints:

```
(  0.70710678,  0.0000000)(  0.70710678,  0.0000000)
PRINT, evecs[*,1]
```

IDL prints:

```
(  0.70710678,  0.0000000)(  0.70710678,  0.0000000)
```

We attempt to compute an independent eigenvector using the method described in the previous example:

```
evecs[*,1] = EIGENVEC(A, evals[1]+1.0e-6, /DOUBLE)
PRINT, evecs[1,*]
```

IDL prints:

```
(  0.70710678,  0.0000000)(  0.70710678,  0.0000000)
```

In this example, *n* independent eigenvectors do not exist. This situation is termed the defective case and cannot be resolved analytically or numerically.

Routines for Computing Eigenvalues and Eigenvectors

See “Eigenvalues and Eigenvectors” (in the functional category “Mathematics” (*IDL Quick Reference*)) for a brief description of IDL routines for computing eigenvalues and eigenvectors. Detailed information is available in the *IDL Reference Guide*.

Gridding and Interpolation

Given a set of tabulated data in *n*-dimensions with each dimension being described as follows:

1. $\{x_i, y_i = f(x_i)\}$,
2. $\{x_i, y_i, z_i = f(x_i, y_i)\}$, or
3. $\{x_i, y_i, z_i, w_i = f(x_i, y_i, z_i)\}$

it is possible to calculate intermediate values of the function *f* using interpolation. IDL includes a variety of routines to solve this type of problem.

The determination of intermediate values is based upon an interpolating function that establishes a relationship between the tabulated data points. Different algorithms employ different types of interpolating functions suitable for different types of data trends.

Unlike curve-fitting algorithms, interpolation requires that the interpolating function be an exact fit at each of the tabulated data points. Interpolation does not use any type of error analysis and its accuracy depends upon the behavior of the interpolating function between successive data points. Polynomial, spline, and nearest-neighbor are among the interpolation methods used in IDL. Kriging is another interpolation method, one which does not require an exact fit at each tabulated data point. Kriging applies a weighting to each of the tabulated data points based on spatial variance and trends among the points. Weights are computed by combining calculations of spatial continuity and anistropy within either an exponential or spherical semivariogram model.

Gridding, a topic closely related to interpolation, is the problem of creating uniformly-spaced planar data from irregularly-spaced data. IDL handles this type of problem by constructing a Delaunay triangulation. This method is highly accurate and has great utility since many of IDL's graphics routines require uniformly-gridded data. Extrapolation, the estimation of values outside the range of tabulated data, is also possible using this method.

Routines for Gridding and Interpolation

See "Gridding and Interpolation" (in the functional category "Mathematics" (*IDL Quick Reference*)) for a brief description of IDL routines for gridding and interpolation. Detailed information is available in the *IDL Reference Guide*.

Hypothesis Testing

Hypothesis testing tests one or more sample populations for a statistical characteristic or interaction. The results of the testing process are generally used to formulate conclusions about the probability distributions of the sample populations.

Hypothesis testing involves four steps:

- The formulation of a hypothesis.
- The selection and collection of sample population data.
- The application of an appropriate test.
- The interpretation of the test results.

For example, suppose the FDA wishes to establish the effectiveness of a new drug in the treatment of a certain ailment. Researchers test the assumption that the drug is effective by administering it to a sample population and collecting data on the patients' health. Once the data are collected, an appropriate statistical test is selected and the results analyzed. If the interpretation of the test results suggests a statistically significant improvement in the patients' condition, the researchers conclude that the drug will be effective in general.

It is important to remember that a valid or successful test does not prove the proposed hypothesis. Only by disproving competing or opposing hypotheses can a given assumption's validity be statistically established.

One- and Two-sided Tests

In the above example, only the hypothesis that the drug would significantly improve the condition of the patients receiving it was tested. This type of test is called one-sided or one-tailed, because it is concerned with deviation in one direction from the norm (in this case, improvement of the patients' condition). A hypothesis designed to test the improvement or ill-effect of the trial drug on the patient group would be called two-sided or two-tailed.

Parametric and Nonparametric Tests

Tests of hypothesis are usually classified into parametric and nonparametric methods. Parametric methods make assumptions about the underlying distribution from which sample populations are selected. Nonparametric methods make no assumptions about a sample population's distribution and are often based upon magnitude-based ranking, rather than actual measurement data. In many cases it is possible to replace a parametric test with a corresponding nonparametric test without significantly affecting the conclusion.

The following example demonstrates this by replacing the parametric T-means test with the nonparametric Wilcoxon Rank-Sum test to test the hypothesis that two sample populations have significantly different means of distribution.

Define two sample populations.

```
X = [257, 208, 296, 324, 240, 246, 267, 311, 324, 323, 263, $
     305, 270, 260, 251, 275, 288, 242, 304, 267]
Y = [201,  56, 185, 221, 165, 161, 182, 239, 278, 243, 197, $
     271, 214, 216, 175, 192, 208, 150, 281, 196]
```

Compute the T-statistic and its significance, using IDL's TM_TEST function, assuming that *X* and *Y* belong to Normal populations with the same variance.

```
PRINT, TM_TEST(X, Y)
```

IDL prints:

```
5.52839  2.52455e-06
```

The small value of the significance (2.52455e-06) indicates that *X* and *Y* have significantly different means.

Compute the Wilcoxon Rank-Sum Test, using IDL's RS_TEST function, to test the hypothesis that *X* and *Y* have the same mean of distribution.

```
PRINT, RS_TEST(X, Y)
```

IDL prints:

```
-4.26039  1.01924e-05
```

The small value of the computed probability (1.01924e-05) requires the rejection of the proposed hypothesis and the conclusion that *X* and *Y* have significantly different means of distribution.

Each of IDL's 11 parametric and nonparametric hypothesis testing functions is based upon a well-known and widely-accepted statistical test. Each of these functions returns a two-element vector containing the statistic on which the test is based and its significance. Examples are provided and demonstrate how the result is interpreted.

Routines for Hypothesis Testing

See "Hypothesis Testing" (in the functional category "Mathematics" (*IDL Quick Reference*)) for a brief description of IDL routines for hypothesis testing. More detailed information is available in the *IDL Reference Guide*.

Integration

Numerical methods of approximating integrals are important in many areas of pure and applied science. For a function of a single variable, $f(x)$, it is often the case that the antiderivative $F = \int f(x)\,dx$ is unavailable using standard techniques such as trigonometric substitutions and integration-by-parts formulas. These standard techniques become increasingly unusable when integrating multivariate functions, $f(x, y)$ and $f(x, y, z)$. Numerically approximating the integral operator provides the only method of solution when the antiderivative is not explicitly available. IDL offers the following numerical methods for the integration of uni-, bi-, and trivariate functions:

- Integration of a univariate function over an open or closed interval is possible using one of several routines based on well known methods developed by Romberg and Simpson.

$$I = \int_{x=a}^{x=b} f(x)dx$$

- The problem of integrating over a tabulated set of data $\{ x_i, y_i = f(x_i) \}$ can be solved using a highly accurate 5-point Newton-Cotes formula. This method is more accurate and efficient than using interpolation or curve-fitting to find an approximate function and then integrating.
- Integration of a bivariate function over a regular or irregular region in the *x*-*y* plane is possible using an iterated Gaussian Quadrature routine.

$$I = \int_{x=a}^{x=b}\int_{y=p(x)}^{y=q(x)} f(x, y)dydx$$

- Integration of a trivariate function over a regular or irregular region in *x*-*y*-*z* space is possible using an iterated Gaussian Quadrature routine.

$$I = \int_{x=a}^{x=b}\int_{y=p(x)}^{y=q(x)}\int_{z=u(x,y)}^{z=v(x,y)} f(x, y, z)dzdydx$$

Note

IDL's iterated Gaussian Quadrature routines, INT_2D and INT_3D, follow the *dy-dx* and *dz-dy-dx* order of evaluation, respectively. Problems not conforming to this standard must be changed as described in the following example.

A Bivariate Function

Suppose that we wish to evaluate

$$\int_{y=0}^{y=4}\int_{x=\sqrt{y}}^{x=2} y \cdot \cos(x^5)\,dx\,dy$$

The order of integration is initially described as a *dx*-*dy* region in the *x*-*y* plane. Using the diagram below, you can easily change the integration order to *dy*-*dx*.

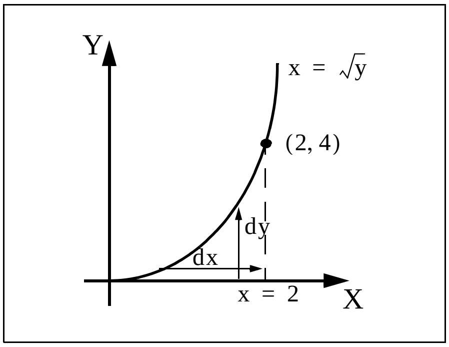


Figure 7-1: The Bivariate Function

The integral is now of the form

$$\int_{x=0}^{x=2}\int_{y=0}^{y=x^2} y \cdot \cos(x^5)\,dy\,dx$$

The new expression can be evaluated using the INT_2D function.

To use INT_2D, we must specify the function to be integrated and expressions for the upper and lower limits of integration. First, we write an IDL function for the integrand, the function *f* (*x*, *y*):

```
FUNCTION fxy, X, Y
  RETURN, Y * COS(X^5)
END
```

Next, we write a function for the limits of integration of the inner integral. Note that the limits of the outer integral are specified numerically, in vector form, while the limits of the inner integral must be specified as an IDL function even if they are constants. In this case, the function is:

```
FUNCTION pq_limits, X
  RETURN, [0.0, X^2]
END
```

Now we can use the following IDL commands to print the value of the integral expressed above. First, we define a variable AB_LIMITS containing the vector of lower and upper limits of the outer integral. Next, we call INT_2D. The first argument is the name of the IDL function that represents the integrand (FXY, in this case). The second argument is the name of the variable containing the vector of limits for the outer integral (AB_LIMITS, in this case). The third argument is the name of the IDL function defining the lower and upper limits of the inside integral (PQ_LIMITS, in this case). The fourth argument (48) refers to the number of transformation points used in the computation. As a general rule, the number of transformation points used with iterated Gaussian Quadrature should increase as the integrand becomes more oscillatory or the region of integration becomes more irregular.

```
ab_limits = [0.0, 2.0]
PRINT, INT_2D('fxy', ab_limits, 'pq_limits', 48)
```

IDL prints:

```
0.055142668
```

This is the exact solution to 9 decimal accuracy.

A Trivariate Function

Suppose that we wish to evaluate

$$\int_{x=-2}^{x=2}\int_{y=-\sqrt{4-x^2}}^{y=\sqrt{4-x^2}}\int_{z=0}^{z=\sqrt{4-x^2-y^2}} z(x^2+y^2+z^2)^{3/2}\,dz\,dy\,dx$$

This integral can be evaluated using the INT_3D function. As with INT_2D, we must specify the function to be integrated and expressions for the upper and lower limits of integration. Note that in this case IDL functions must be provided for the upper and lower integration limits of both inside integrals.

For the above integral, the required functions are the integrand $f(x, y, z)$:

```
FUNCTION fxyz, X, Y, Z
  RETURN, Z * (X^2 + Y^2 + Z^2)^1.5
END
```

The limits of integration of the first inside integral:

```
FUNCTION pq_limits, X
  RETURN, [-SQRT(4.0 - X^2), SQRT(4.0 -X^2)]
END
```

The limits of integration of the second inside integral:

```
FUNCTION uv_limits, X, Y
  RETURN, [0.0, SQRT(4.0 - X^2 - Y^2)]
END
```

We can use the following IDL commands to determine the value of the above integral using 6, 10, 20 and 48 transformation points.

For 6 transformation points:

```
PRINT, INT_3D('fxyz', [-2.0, 2.0], $
   'pq_limits', 'uv_limits', 6)
```

IDL prints:

```
57.417720
```

For 10 transformation points:

```
PRINT, INT_3D('fxyz', [-2.0, 2.0], $
   'pq_limits', 'uv_limits', 10)
```

IDL prints:

```
57.444248
```

20 transformation points:

```
PRINT, INT_3D('fxyz', [-2.0, 2.0], $
   'pq_limits', 'uv_limits', 20)
```

IDL prints:

```
57.446201
```

48 transformation points:

```
PRINT, INT_3D('fxyz', [-2.0, 2.0], $
   'pq_limits', 'uv_limits', 48)
```

IDL prints:

```
57.446265
```

The exact solution to 6-decimal accuracy is 57.446267.

Routines for Differentiation and Integration

See “Differentiation and Integration” (in the functional category “Mathematics” (*IDL Quick Reference*)) for a brief description of IDL routines for differentiation and integration. Detailed information is available in the *IDL Reference Guide*.

Linear Systems

IDL offers a variety of methods for the solution of simultaneous linear equations. In order to use these routines successfully, the user should consider both existence and uniqueness criteria and the potential difficulties in finding the solution numerically.

The solution vector *x* of an *n*-by-*n* linear system A*x* = b is guaranteed to exist and to be unique if the coefficient array A is invertible. Using a simple algebraic manipulation, it is possible to formulate the solution vector *x* in terms of the inverse of the coefficient array A and the right-side vector b: $x = A^{-1}b$. Although this relationship provides a concise mathematical representation of the solution, it is never used in practice. Array inversion is computationally expensive (requiring a large number of floating-point operations) and prone to severe round-off errors.

An alternate way of describing the existence of a solution is to say that the system A*x* = b is solvable if and only if the vector b may be expressed as a linear combination of the columns of A. This definition is important when considering the solutions of non-square (over- and under-determined) linear systems.

While the invertabiltiy of the coefficient array A may ensure that a solution exists, it does not help in determining the solution. Some systems can be solved accurately using numerical methods whereas others cannot. In order to better understand the accuracy of a numerical solution, we can classify the condition of the system it solves.

The scalar quantity known as the condition number of a linear system is a measure of a solution’s sensitivity to the effects of finite-precision arithmetic. The condition number of an *n*-by-*n* linear system A*x* = b is computed explicitly as $|A||A^{-1}|$ (where | | denotes a Euclidean norm). A linear system whose condition number is small is considered well-conditioned and well suited to numerical computation. A linear system whose condition number is large is considered ill-conditioned and prone to computational errors. To some extent, the solution of an ill-conditioned system may be improved using an extended-precision data type (such as double-precision float). Other situations require an approximate solution to the system using its Singular Value Decomposition.

The following two examples show how the singular value decomposition may be used to find solutions when a linear system is over- or underdetermined.

Overdetermined Systems

In the case of the overdetermined system (when there are more linear equations than unknowns), the vector b cannot be expressed as a linear combination of the columns of array *A*. (In other words, b lies outside of the subspace spanned by the columns of *A*.) Using IDL's SVDC procedure, it is possible to determine a projected solution of the overdetermined system (b is projected onto the subspace spanned by the columns of A and then the system is solved). This type of solution has the property of minimizing the residual error $E = b - Ax$ in a least-squares sense.

Suppose that we wish to solve the following linear system:

$$\begin{bmatrix} 1.0 & 2.0 \\ 1.0 & 3.0 \\ 0.0 & 0.0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 4.0 \\ 5.0 \\ 6.0 \end{bmatrix}$$

The vector *b* does not lie in the two-dimensional subspace spanned by the columns of *A* (there is no linear combination of the columns of *A* that yield *b*), and therefore an exact solution is not possible.

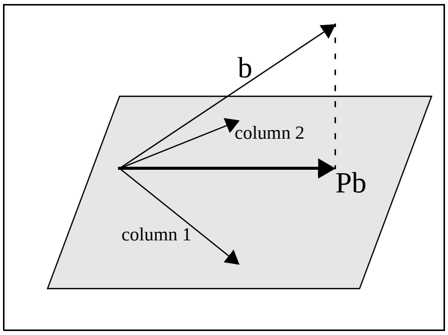


Figure 7-2: Overdetermined System Diagram

It is possible, however, to find a solution to this system that minimizes the residual error by orthogonally projecting the vector b onto the two-dimensional subspace spanned by the columns of the array *A*. The projected vector is then used as the right-hand side of the system. The orthogonal projection of *b* onto the column space of *A* may be expressed with the array-vector product $A(A^TA)^{-1}A^Tb$, where $A(A^TA)^{-1}A^T$ is known as the projection matrix, P.

In this example, the array-vector product Pb yields:

$$\begin{bmatrix} 4.0 \\ 5.0 \\ 0.0 \end{bmatrix}$$

and we wish to solve the linear system

$$\begin{bmatrix} 1.0 & 2.0 \\ 1.0 & 3.0 \\ 0.0 & 0.0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 4.0 \\ 5.0 \\ 0.0 \end{bmatrix} \quad \text{where} \quad \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.0 \end{bmatrix}$$

In many cases, the explicit calculation of the projected solution is numerically unstable, resulting in large accumulated round-off errors. For this reason it is best to use singular value decomposition to effect the orthogonal projection of the vector b onto the subspace spanned by the columns of the array A.

The following IDL commands use singular value decomposition to solve the system in a numerically stable manner. Begin with the array A:

```
A = [[1.0, 2.0], $
     [1.0, 3.0], $
     [0.0, 0.0]]
; Define the right-hand side vector B:
B = [4.0, 5.0, 6.0]
; Compute the singular value decomposition of A:
SVDC, A, W, U, V
```

Create a diagonal array WP of reciprocal singular values from the output vector W. To avoid overflow errors when the reciprocal values are calculated, only elements with absolute values greater than or equal to 1.0×10^{-5} are reciprocated.

```
N = N_ELEMENTS(W)
WP = FLTARR(N, N)
FOR K = 0, N-1 DO $
   IF ABS(W(K)) GE 1.0e-5 THEN WP(K, K) = 1.0/W(K)
```

We can now express the solution to the linear system as a array-vector product. (See Section 2.6 of *Numerical Recipes* for a derivation of this formula.)

```
X = V ## WP ## TRANSPOSE(U) ## B
; Print the solution:
PRINT, X
```

IDL Prints:

```
2.00000
1.00000
```

Underdetermined Systems

In the case of the underdetermined system (when there are fewer linear equations than unknowns), a unique solution is not possible. Using IDL’s SVDC procedure it is possible to determine the minimal norm solution. Given a vector norm, this type of solution has the property of having the minimal length of all possible solutions with respect to that norm.

Suppose that we wish to solve the following linear system.

$$\begin{bmatrix} 1.0 & 3.0 & 3.0 & 2.0 \\ 2.0 & 6.0 & 9.0 & 5.0 \\ -1.0 & -3.0 & 3.0 & 0.0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 5.0 \\ 5.0 \end{bmatrix}$$

Using elementary row operations it is possible to reduce the system to

$$\begin{bmatrix} 1.0 & 3.0 & 3.0 & 2.0 \\ 0.0 & 0.0 & 3.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 3.0 \\ 0.0 \end{bmatrix}$$

It is now possible to express the solution x in terms of x_1 and x_3:

$$x = \begin{bmatrix} -2 - 3x_1 - x_3 \\ x_1 \\ 1 - x_3/3 \\ x_3 \end{bmatrix}$$

The values of x_1 and x_3 are completely arbitrary. Setting $x_1 = 0$ and $x_3 = 0$ results in one possible solution of this system:

Another possible solution is obtained using singular value decomposition and results in the minimal norm condition. The minimal norm solution for this system is:

$$x = \begin{bmatrix} -2.0 \\ 0.0 \\ 1.0 \\ 0.0 \end{bmatrix}$$

$$x = \begin{bmatrix} -0.211009 \\ -0.633027 \\ 0.963303 \\ 0.110092 \end{bmatrix}$$

Note that this vector also satisfies the solution x as it is expressed in terms of x_1 and x_3.

The following IDL commands use singular value decomposition to find the minimal norm solution. Begin with the array *A*:

```
A = [[ 1.0, 3.0, 3.0, 2.0], $
     [ 2.0, 6.0, 9.0, 5.0], $
     [-1.0, -3.0, 3.0, 0.0]]
; Define the right-hand side vector B:
B = [1.0, 5.0, 5.0]
; Compute the decomposition of A:
SVDC, A, W, U, V
```

Create a diagonal array WP of reciprocal singular values from the output vector W. To avoid overflow errors when the reciprocal values are calculated, only elements with absolute values greater than or equal to 1.0×10^{-5} are reciprocated.

```
N = N_ELEMENTS(W)
WP = FLTARR(N, N)
FOR K = 0, N-1 DO $
   IF ABS(W(K)) GE 1.0e-5 THEN WP(K, K) = 1.0/W(K)
```

We can now express the solution to the linear system as a array-vector product. (See Section 2.6 of *Numerical Recipes* for a derivation of this formula.) The solution is expressed in terms of x_1 and x_3 with minimal norm.

```
X = V ## WP ## TRANSPOSE(U) ## B
;Print the solution:
PRINT, X
```

IDL Prints:

```
-0.211009
-0.633027
 0.963303
```

```
    0.110092
```

Complex Linear Systems

We can use IDL's LU_COMPLEX function to compute the solution to a linear system with real and complex coefficients. Suppose we wish to solve the following linear system:

$$\begin{bmatrix} -1+0i & 1-3i & 2+0i & 3+3i \\ -2+0i & -1+3i & 0+1i & 3+1i \\ 3+0i & 0+4i & 0-1i & 0-3i \\ 2+0i & 1+1i & 2+1i & 2+1i \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 15-2i \\ -2-1i \\ -20+11i \\ -10+10i \end{bmatrix}$$

```
;First we define the real part of the complex coefficient array:
re = [[-1, 1, 2, 3], $
      [-2, -1, 0, 3], $
      [3, 0, 0, 0], $
      [2, 1, 2, 2]]
;Next, we define the imaginary part of the coefficient array:
im = [[0, -3, 0, 3], $
      [0, 3, 1, 1], $

      [0, 4, -1, -3], $
      [0, 1, 1, 1]]
; Combine the real and imaginary parts to form
; a single complex coefficient array:
A = COMPLEX(re, im)
; Define the right-hand side vector B:
B = [COMPLEX(15,-2), COMPLEX(-2,-1), COMPLEX(-20,11), $
   COMPLEX(-10,10)
; Compute the solution using double-precision complex arithmetic:
Z = LU_COMPLEX(A, B, /DOUBLE)
PRINT, TRANSPOSE(Z), FORMAT = '(f5.2, ",", f5.2, "i")'
```

IDL prints:

```
-4.00, 1.00i
 2.00, 2.00i
 0.00, 3.00i
-0.00,-1.00i
```

We can check the accuracy of the computed solution by computing the residual, A*z*–b:

```
PRINT, A##Z-B
```

IDL prints:

```
(      0.00000,      0.00000)
(      0.00000,      0.00000)
(      0.00000,      0.00000)
(      0.00000,      0.00000)
```

Routines for Solving Simultaneous Linear Equations

See “Linear Systems” (in the functional category “Mathematics” (*IDL Quick Reference*)) for a brief description of IDL routines for solving simultaneous linear equations. Detailed information is available in the *IDL Reference Guide*.

Nonlinear Equations

The problem of finding a solution to a system of *n* nonlinear equations, F(*x*) = 0, may be stated as follows:

given F: $R^n \rightarrow R^n$, find x_* (an element of R^n) such that $F(x_*) = 0$

For example:

$$F(x) = \begin{bmatrix} x_0 + x_1 - 3 \\ x_0^2 + x_1^2 - 9 \end{bmatrix}$$

$x_* = [0, 3]$ or $x_* = [3, 0]$

Note

A solution to a system of nonlinear equations is not necessarily unique.

The most powerful and successful numerical methods for solving systems of nonlinear equations are loosely based upon a simple two-step iterative method frequently referred to as Newton's method. This method begins with an initial guess and constructs a solution by iteratively approximating the *n*-dimensional nonlinear system of equations with an *n*-by-*n* linear system of equations.

The first step formulates an *n*-by-*n* linear system of equations (J*s* = – F) where the coefficient array J is the Jacobian (the array of first partial derivatives of F), *s* is a solution vector, and – F is the negative of the nonlinear system of equations. Both J and – F are evaluated at the current value of the *n*-element vector *x*.

$J(x_k)\, s_k = -F(x_k)$

The second step uses the solution s_k of the linear system as a directional update to the current approximate solution x_k of the nonlinear system of equations. The next approximate solution x_{k+1} is a linear combination of the current approximate solution x_k and the directional update s_k.

$x_{k+1} = x_k + s_k$

The success of Newton's method relies primarily on providing an initial guess close to a solution of the nonlinear system of equations. In practice this proves to be quite difficult and severely limits the application of this simple two-step method.

IDL provides two algorithms that are designed to overcome the restriction that the initial guess be close to a solution. These algorithms implement a line search which checks, and if necessary modifies, the course of the algorithm at each step ensuring progress toward a solution of the nonlinear system of equations. IDL's NEWTON and BROYDEN functions are among a class of algorithms known as quasi-Newton methods.

The solution of an *n*-dimensional system of nonlinear equations, F(*x*) = 0, is often considered a root of that system. As a one-dimensional counterpart to NEWTON and BROYDEN, IDL provides the FX_ROOT and FZ_ROOTS functions.

Routines for Solving Nonlinear Equations

See "Nonlinear Equations" (in the functional category "Mathematics" (*IDL Quick Reference*)) for a brief description of IDL routines for solving systems of nonlinear equations. Detailed information is available in the *IDL Reference Guide*.

Optimization

The problem of finding an unconstrained minimizer of an *n*-dimensional function, *f*, may be stated as follows:

given *f*: $R^n \rightarrow R$, find x_* (an element of R^n) such that $f(x_*)$ is a minimum of *f*.

For example:

$f(x) = (x_0 - 3)^4 + (x_1 - 2)^2$

$x_* = [3, 2]$

In minimizing an *n*-dimensional function *f*, it is a necessary condition that the gradient at the minimizer x_*, $\nabla f(x_*)$, be the zero vector. Mathematically expressing this condition defines the following system of nonlinear equations.

$$\begin{bmatrix} \dfrac{\partial f(x)}{\partial x_0} \\ \dfrac{\partial f(x)}{\partial x_1} \\ \ldots \\ \dfrac{\partial f(x)}{\partial x_{n-1}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \ldots \\ 0 \end{bmatrix}$$

This relation might suggest that finding a minimizer is equivalent to solving a system of linear equations based on the gradient. In most cases, however, this is not true. It is just as likely that a solution, x_*, of $\nabla f(x)=0$ be a maximizer or a local minimizer of *f*. Thus the gradient alone does not provide sufficient information in determining the role of x_*.

IDL provides two algorithms that do sufficiently determine the global minimizer of an n-dimensional function. IDL's DFPMIN routine is among a class of algorithms known as variable metric methods and requires a user-supplied analytic gradient of the function to be minimized. IDL's POWELL routine implements a direction-set method that does not require a user-supplied analytic gradient. The utility of the POWELL routine is evident as the function to be minimized becomes more complicated and partial derivatives become more difficult to calculate.

Routines for Optimization

See “Optimization” (in the functional category “Mathematics” (*IDL Quick Reference*)) for a brief description of IDL routines for optimization. Detailed information is available in the *IDL Reference Guide*.

Sparse Arrays

The occurrence of zero elements in a large array is both a computational and storage inconvenience. An array in which a large percentage of elements are zeros is referred to as being *sparse*.

Because standard linear algebra techniques are highly inefficient when dealing with sparse arrays, IDL incorporates a collection of routines designed to handle them effectively. These routines use the row-indexed sparse storage method, which stores the array in structure form, as a vector of data and a vector of indices. The length of each vector is equal to 1 plus the number of diagonal elements of the array plus the number of off-diagonal elements with an absolute magnitude greater than or equal to a specified threshold value. Diagonal elements of the array are always retained even if their absolute magnitude is less than the specified threshold. Sparse array routines that handle array-vector and array-array multiplication, file input/output, and the solution of systems of simultaneous linear equations are included.

Note

For more information on IDL's sparse array storage method, see section 2.7, "Sparse Linear Systems," in *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press.

When considering using IDL's sparse array routines, remember that the computational savings gained by working in sparse storage format is at least partially offset by the need to first convert the arrays to that format. Although an absolute determination of when to use sparse format is not possible, the example below demonstrates the time savings when solving a 500 by 500 linear system in which approximately 50% of the coefficient array's elements as zeros.

Diagonally-Dominant Array

Create a 500-by-500 element pseudo-random diagonally-dominant floating-point array in which approximately 50% of the elements as zeros. (In a diagonally-dominant array, the diagonal element in a given row is greater than the sum of the absolute values of the non-diagonal elements in that row.)

```
N = 500L
A = RANDOMN(SEED, N, N)*10
; Set elements with absolute magnitude greater than or
; equal to eight to zero:
I = WHERE(ABS(A) GE 8)
A[I] = 0.0
; Set each diagonal element to the absolute sum of
; its row elements plus 1.0:
diag = TOTAL(ABS(A), 1)
A(INDGEN(N) * (N+1)) = diag + 1.0
; Create a right-hand side vector, b, in which 40% of
; the elements are ones and 60% are twos.
B = [REPLICATE(1.0, 0.4*N), REPLICATE(2.0, 0.6*N)]
```

We now calculate a solution to this system using two different methods, measuring the time elapsed. First, we compute the solution using the iterative biconjugate gradient method and a sparse array storage format. Note that we include everything between the start and stop timer commands as a single operation, so that only computation time (as opposed to typing time) is recorded.

```
; Begin with an initial guess:
X = REPLICATE(1.0, N_ELEMENTS(B))
; Start the timer:
start = SYSTIME(1) & $
; Solve the system:
result1 = LINBCG(SPRSIN(A), B, X) & $
; Stop the timer.
stop = SYSTIME(1)
; Print the time taken, in seconds:
PRINT, 'Time for Iterative Biconjugate Gradient:', stop-start
```

IDL prints:

```
Time for Iterative Biconjugate Gradient       1.1259040
```

Remember that your result will depend on your hardware configuration.

Next, we compute the solution using LU decomposition.

```
; Start the timer:
start = SYSTIME(1) & $
; Compute the LU decomposition of A:
LUDC, A, index & $
; Compute the solution:
result2 = LUSOL(A, index, B) & $
; Stop the timer:
stop = SYSTIME(1)
; Print the time taken, in seconds:
PRINT, 'Time for LU Decomposition:', stop-start
```

IDL prints:

```
Time for LU decomposition       14.871168
```

Finally, we can compare the absolute error between result1 and result2. The following commands will print the indices of any elements of the two results that differ by more than 1.0×10^{-5}, or a –1 if the two results are identical to within five decimal places.

```
error = ABS(result1-result2)
PRINT, WHERE(error GT 1.0e-5)
```

IDL prints:

```
-1
```

See the documentation for the WTN function for an example using IDL's sparse array functions with image data.

Note

The times shown here were recorded on a DEC 3000 Alpha workstation running OSF/1; they are shown as examples only. Your times will depend on your specific computing platform.

Routines for Handling Sparse Arrays

See "Sparse Arrays" (in the functional category "Mathematics" (*IDL Quick Reference*)) for a brief description of IDL routines for handling sparse arrays. More detailed information is available in the *IDL Reference Guide*.

Time-Series Analysis

A time-series is a sequential collection of data observations indexed over time. In most cases, the observed data is continuous and is recorded at a discrete and finite set of equally-spaced points. An *n*-element time-series is denoted as $x = (x_0, x_1, x_2, \ldots, x_{n-1})$, where the time-indexed distance between any two successive observations is referred to as the sampling interval.

A widely held theory assumes that a time-series is comprised of four components:

- A trend or long term movement.
- A cyclical fluctuation about the trend.
- A pronounced seasonal effect.
- A residual, irregular, or random effect.

Collectively, these components make the analysis of a time-series a far more challenging task than just fitting a linear or nonlinear regression model. Adjacent observations are unlikely to be independent of one another. Clusters of observations are frequently correlated with increasing strength as the time intervals between them become shorter. Often the analysis is a multi-step process involving graphical and numerical methods.

The first step in the analysis of a time-series is the transformation to stationary series. A stationary series exhibits statistical properties that are unchanged as the period of observation is moved forward or backward in time. Specifically, the mean and variance of a stationary time-series remain fixed in time. The sample autocorrelation function is a commonly used tool in determining the stationarity of a time-series. The autocorrelation of a time-series measures the dependence between observations as a function of their time differences or lag. A plot of the sample autocorrelation coefficients against corresponding lags can be very helpful in determining the stationarity of a time-series.

For example, suppose the IDL variable *X* contains time-series data:

```
X = [5.44, 6.38, 5.43, 5.22, 5.28, $
     5.21, 5.23, 4.33, 5.58, 6.18, $
     6.16, 6.07, 6.56, 5.93, 5.70, $
     5.36, 5.17, 5.35, 5.61, 5.83, $
     5.29, 5.58, 4.77, 5.17, 5.33]
```

The following IDL commands plot both the time-series data and the sample autocorrelation versus the lags.

```
; Set the plotting window to hold two plots and plot the data:
IPLOT, X, VIEW_GRID=[1,2]
```

Compute the sample autocorrelation function for time lagged values 0 – 20 and plot.

```
lag = INDGEN(21)
result = A_CORRELATE(X, lag)
IPLOT, lag, result, /VIEW_NEXT
; Add a reference line at zero:
IPLOT, [0,20], [0,0], /OVERPLOT
```

The following figure shows the resulting graphs.

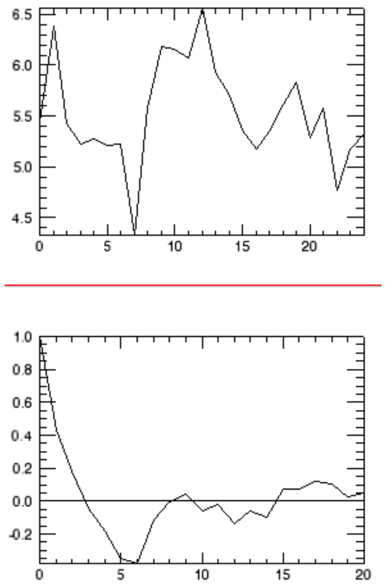


Figure 7-3: Time-series data (Top) and Autocorrelation of that Data Versus the Lag (Bottom)

The top graph plots time-series data. The bottom graph plots the autocorrelation of that data versus the lag. Because the time-series has a significant autocorrelation up to a lag of seven, it must be considered non-stationary.

Nonstationary components of a time-series may be eliminated in a variety of ways. Two frequently used methods are known as moving averages and forward differencing. The method of moving averages dampens fluctuations in a time-series by taking successive averages of groups of observations. Each successive overlapping sequence of k observations in the series is replaced by the mean of that sequence. The method of forward differencing replaces each time-series observation with the difference of the current observation and its adjacent observation one step forward in time. Differencing may be computed recursively to eliminate more complex nonstationary components.

Once a time-series has been transformed to stationarity, it may be modeled using an autoregressive process. An autoregressive process expresses the current observation, x_t, as a combination of past time-series values and residual white noise. The simplest case is known as a first order autoregressive model and is expressed as

$$x_t = \phi x_{t-1} + \omega_t$$

The coefficient ϕ is estimated using the time-series data. The general autoregressive model of order *p* is expressed as

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \ldots + \phi_p x_{t-p} + \omega_t$$

Modeling a stationary time-series as a *p*-th order autoregressive process allows the extrapolation of data for future values of time. This process is know as forecasting.

Routines for Time-Series Analysis

See "Time-Series Analysis" (in the functional category "Mathematics" (*IDL Quick Reference*)) for a brief description of IDL routines for time-series analysis. Detailed information is available in the *IDL Reference Guide*.

Multivariate Analysis

IDL provides a number of tools for analyzing multivariate data. These tools are broadly grouped into two categories: Cluster Analysis and Principal Components Analysis.

Cluster Analysis

Cluster Analysis attempts to construct a sensible and informative classification of an initially unclassified sample population using a set of common variables for each individual. The clusters are constructed so as to group samples with the similar features, based upon a set of variables. The samples (contained in the rows of an input array) are each assigned a cluster number based upon the values of their corresponding variables (contained in the columns of an input array).

In computing a cluster analysis, a predetermined number of cluster centers are formed and then each sample is assigned to the unique cluster which minimizes a distance criterion based upon the variables of the data. Given an m-column, n-row array, IDL's CLUST_WTS and CLUSTER functions compute n cluster centers and n clusters, respectively. Conceivably, some clusters will contain multiple samples while other clusters will contain none. The choice of clusters is arbitrary; in general, however, the user will want to specify a number less than the default (the number of rows in the input array). The cluster number (the number that identifies the cluster group) assigned to a particular sample or group of samples is not necessarily unique.

It is possible that not all variables play an equal role in the classification process. In this situation, greater or lesser importance may be given to each variable using the VARIABLE_WTS keyword to the CLUST_WTS function. The default behavior is to assume all variables contained in the data array are of equal importance.

Under certain circumstances, a classification of variables may be desired. The CLUST_WTS and CLUSTER functions provide this functionality by first transposing the *m*-column, *n*-row input array using the TRANSPOSE function and then interchanging the roles of variables and samples.

Example of Cluster Analysis

Define an array with 5 variables (columns) and 9 samples (rows):

```
array = [[ 99,  79,  63,  87, 249 ], $
         [ 67,  41,  36,  51, 114 ], $
         [ 67,  41,  36,  51, 114 ], $
         [ 94, 191, 160, 173, 124 ], $
         [ 42, 108,  37,  51,  41 ], $
```

```
            [ 67,  41,  36,  51, 114 ], $
            [ 94, 191, 160, 173, 124 ], $
            [ 99,  79,  63,  87, 249 ], $
            [ 67,  41,  36,  51, 114 ]]
   ; Compute the cluster weights with four cluster centers:
   weights = CLUST_WTS(array, N_CLUSTERS = 4)
   ; Compute the cluster assignments, for each sample,
   ; into one of four clusters:
   result  = CLUSTER(array, weights, N_CLUSTERS = 4)
   ; Display the cluster assignment and corresponding sample (row):
   FOR k = 0, 8 DO $
      PRINT, result[k], array[*, k]
```

IDL prints:

```
   1      99      79      63      87     249
   3      67      41      36      51     114
   3      67      41      36      51     114
   0      94     191     160     173     124
   2      42     108      37      51      41
   3      67      41      36      51     114
   0      94     191     160     173     124
   1      99      79      63      87     249
   3      67      41      36      51     114
```

Samples 0 and 7 contain identical data and are assigned to cluster #1. Samples 1, 2, 5, and 8 contain identical data and are assigned to cluster #3. Samples 3 and 6 contain identical data and are assigned to cluster #0. Sample 4 is unique and is assigned to cluster #2.

If this example is run several times, each time computing new cluster weights, it is possible that the cluster number assigned to each grouping of samples may change.

Principal Components Analysis

Principal components analysis is a mathematical technique which describes a multivariate set of data using derived variables. The derived variables are formulated using specific linear combinations of the original variables. The derived variables are uncorrelated and are computed in decreasing order of importance; the first variable accounts for as much as possible of the variation in the original data, the second variable accounts for the second largest portion of the variation in the original data, and so on. Principal components analysis attempts to construct a small set of derived variables which summarize the original data, thereby reducing the dimensionality of the original data.

The principal components of a multivariate set of data are computed from the eigenvalues and eigenvectors of either the sample correlation or sample covariance matrix. If the variables of the multivariate data are measured in widely differing units (large variations in magnitude), it is usually best to use the sample correlation matrix in computing the principal components; this is the default method used in IDL's PCOMP function.

Another alternative is to standardize the variables of the multivariate data prior to computing principal components. Standardizing the variables essentially makes them all equally important by creating new variables that each have a mean of zero and a variance of one. Proceeding in this way allows the principal components to be computed from the sample covariance matrix. IDL's PCOMP function includes COVARIANCE and STANDARDIZE keywords to provide this functionality.

For example, suppose that we wish to restate the following data using its principal components. There are three variables, each consisting of five samples.

	Var 1	**Var 2**	**Var 3**
Sample 1	2.0	1.0	3.0
Sample 2	4.0	2.0	3.0
Sample 3	4.0	1.0	0.0
Sample 4	2.0	3.0	3.0
Sample 5	5.0	1.0	9.0

Table 7-1: Data for Principal Component Analysis

We compute the principal components (the coefficients of the derived variables) to 2 decimal accuracy and store them by row in the following array.

$$\begin{bmatrix} 0.87 & -0.70 & 0.69 \\ 0.01 & -0.64 & -0.66 \\ 0.49 & 0.32 & -0.30 \end{bmatrix}$$

The derived variables $\{z_1, z_2, z_3\}$ are then computed as follows:

$$z1 = (0.87)\begin{bmatrix}2.0\\4.0\\4.0\\2.0\\5.0\end{bmatrix} + (-0.70)\begin{bmatrix}1.0\\2.0\\1.0\\3.0\\1.0\end{bmatrix} + (\ \ 0.69)\begin{bmatrix}3.0\\3.0\\0.0\\3.0\\9.0\end{bmatrix}$$

$$z2 = (0.01)\begin{bmatrix}2.0\\4.0\\4.0\\2.0\\5.0\end{bmatrix} + (-0.64)\begin{bmatrix}1.0\\2.0\\1.0\\3.0\\1.0\end{bmatrix} + (-0.66)\begin{bmatrix}3.0\\3.0\\0.0\\3.0\\9.0\end{bmatrix}$$

$$z3 = (0.49)\begin{bmatrix}2.0\\4.0\\4.0\\2.0\\5.0\end{bmatrix} + (\ \ 0.32)\begin{bmatrix}1.0\\2.0\\1.0\\3.0\\1.0\end{bmatrix} + (-0.30)\begin{bmatrix}3.0\\3.0\\0.0\\3.0\\9.0\end{bmatrix}$$

In this example, analysis shows that the derived variable z_1 accounts for 57.3% of the total variance of the original data, the derived variable z_2 accounts for 28.2% of the total variance of the original data, and the derived variable z_3 accounts for 14.5% of the total variance of the original data.

Example of Derived Variables from Principal Components

The following example constructs an appropriate set of derived variables, based upon the principal components of the original data, which may be used to reduce the dimensionality of the data. The data consist of four variables, each containing of twenty samples.

```
; Define an array with 4 variables and 20 samples:
data = [[19.5, 43.1, 29.1, 11.9], $
        [24.7, 49.8, 28.2, 22.8], $
        [30.7, 51.9, 37.0, 18.7], $
        [29.8, 54.3, 31.1, 20.1], $
        [19.1, 42.2, 30.9, 12.9], $
```

```
           [25.6, 53.9, 23.7, 21.7], $
           [31.4, 58.5, 27.6, 27.1], $
           [27.9, 52.1, 30.6, 25.4], $
           [22.1, 49.9, 23.2, 21.3], $
           [25.5, 53.5, 24.8, 19.3], $
           [31.1, 56.6, 30.0, 25.4], $
           [30.4, 56.7, 28.3, 27.2], $
           [18.7, 46.5, 23.0, 11.7], $
           [19.7, 44.2, 28.6, 17.8], $
           [14.6, 42.7, 21.3, 12.8], $
           [29.5, 54.4, 30.1, 23.9], $
           [27.7, 55.3, 25.7, 22.6], $
           [30.2, 58.6, 24.6, 25.4], $
           [22.7, 48.2, 27.1, 14.8], $
           [25.2, 51.0, 27.5, 21.1]]
```

The variables that will contain the values returned by the COEFFICIENTS, EIGENVALUES, and VARIANCES keywords to the PCOMP routine must be initialized as nonzero values prior to calling PCOMP.

```
coef = 1 & eval = 1 & var = 1
; Compute the derived variables based upon
; the principal components.
result = PCOMP(data, COEFFICIENTS = coef, $
   EIGENVALUES = eval, VARIANCES = var)
; Display the array of derived variables:
PRINT, result, FORMAT = '(4(f5.1, 2x))'
```

IDL prints:

```
 81.4   15.5   -5.5    0.5
102.7   11.1   -4.1    0.6
109.9   20.3   -6.2    0.5
110.5   13.8   -6.3    0.6
 81.8   17.1   -4.9    0.6
104.8    6.2   -5.4    0.6
121.3    8.1   -5.2    0.6
111.3   12.6   -4.0    0.6
 97.0    6.4   -4.4    0.6
102.5    7.8   -6.1    0.6
118.5   11.2   -5.3    0.6
118.9    9.1   -4.7    0.6
 81.5    8.8   -6.3    0.6
 88.0   13.4   -3.9    0.6
 74.3    7.5   -4.8    0.6
113.4   12.0   -5.1    0.6
109.7    7.7   -5.6    0.6
117.5    5.5   -5.7    0.6
 91.4   12.0   -6.1    0.6
102.5   10.6   -4.9    0.6
```

Display the percentage of total variance for each derived variable:

```
PRINT, var
```

IDL prints:

```
0.712422
0.250319
0.0370950
0.000164269
```

Display the percentage of variance for the first two derived variables; the first two columns of the resulting array above.

```
PRINT, TOTAL(var[0:1])
```

IDL prints:

```
0.962741
```

This indicates that the first two derived variables (the first two columns of the resulting array) account for 96.3% of the total variance of the original data, and thus could be used to summarize the original data.

Routines for Multivariate Analysis

See "Multivariate Analysis" (in the functional category "Mathematics" (*IDL Quick Reference*)) for a brief description of IDL routines for multivariate analysis. Detailed information is available in the *IDL Reference Guide*.

References

Correlation Analysis

Harnet, Donald L. *Introduction to Statistical Methods*. Reading, Massachusetts: Addison-Wesley, 1975. ISBN 0-201-02752-6

Neter, John., William Wasserman, and G.A. Whitmore. *Applied Statistics*. Newton, Massachusetts: Allyn and Bacon, 1988. ISBN 0-205-10328-6

Press, William H. *et al*. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Curve and Surface Fitting

Bevington, Philip R. *Data Reduction and Error Analysis for the Physical Sciences*. New York: McGraw-Hill, 1969.

Lancaster, Peter and Kestutis Salkauskas. *Curve and Surface Fitting* (*An Introduction*). San Diego: Academic Press, 1986. ISBN 0-124-36060-0

Eigenvalues and Eigenvectors

Press, William H. *et al*. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Strang, Gilbert. *Linear Algebra and Its Applications*. San Diego: Harcourt Brace Jovanovich, 1988. ISBN 0-155-551005-3

Gridding and Interpolation

Lancaster, Peter and Kestutis Salkauskas. *Curve and Surface Fitting (An Introduction)*. San Diego: Academic Press, 1986. ISBN 0-124-36060-0

Press, William H. *et al*. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Hypothesis Testing

Harnett, Donald H. *Introduction to Statistical Methods*. Reading, Massachusetts: Addison-Wesley, 1975. ISBN 0-201-02752-6

Kraft, Charles H. and Constance Van Eeden. *A Nonparametric Introduction to Statistics*. New York: Macmillan, 1968.

Sprent, Peter. *Applied Nonparametric Statistical Methods*. London: Chapman and Hall, 1989. ISBN 0-412-30600-X

Integration

Chapra, Steven C. and Raymond P. Canale. *Numerical Methods for Engineers*. New York: McGraw-Hill, 1988. ISBN 0-070-79984-9

Press, William H. *et al*. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Linear Systems

Golub, Gene H. and Van Loan, Charles F. *Matrix Computations*. Baltimore: Johns Hopkins University Press, 1989. ISBN 0-8018-3772-3

Kreyszig, Erwin. *Advanced Engineering Mathematics*. New York: Wiley & Sons, Inc., 1993. ISBN 0-471-55380-8

Press, William H. *et al*. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Strang, Gilbert. *Linear Algebra and Its Applications*. San Diego: Harcourt Brace Jovanovich, 1988. ISBN 0-155-551005-3

Nonlinear Equations

Dennis, J.E. Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, NJ: Prentice-Hall, 1983. ISBN 0-136-27216-9

Press, William H. *et al*. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Optimization

Dennis, J.E. Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, NJ: Prentice-Hall, 1983. ISBN 0-136-27216-9

Press, William H. *et al*. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Sparse Arrays

Press, William H. *et al*. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Time-Series Analysis

Chatfield, C. *The Analysis of Time Series*. London: Chapman and Hall, 1975. ISBN 0-412-31820-2

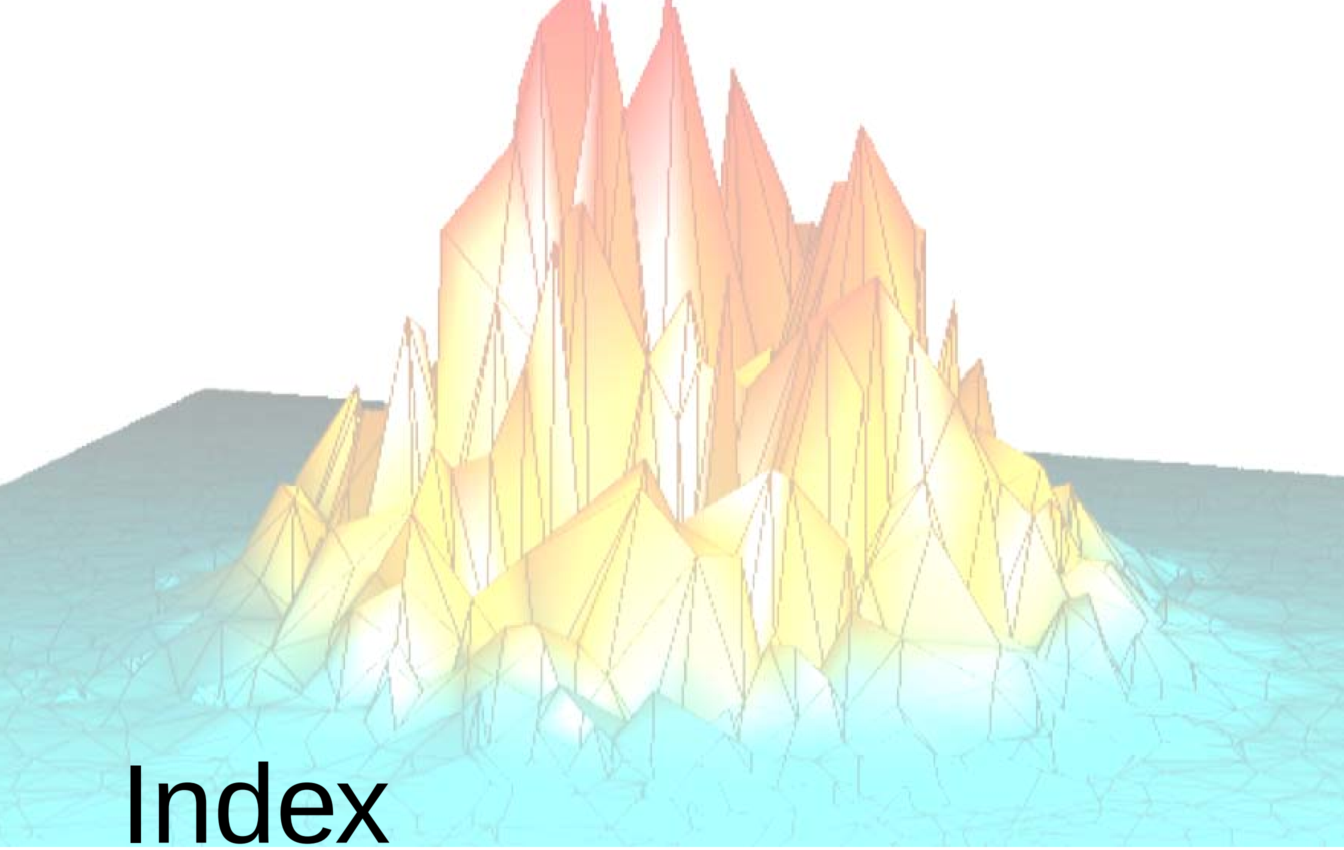
Neter, John., William Wasserman, and G.A. Whitmore. *Applied Statistics*. Newton, Massachusetts: Allyn and Bacon, 1988. ISBN 0-205-10328-6

Multivariate Analysis

Jackson, Barbara Bund. *Multivariate Data Analysis*. Homewood, Illinois: R.D. Irwin, 1983. ISBN 0-256-02848-6

Everitt, Brian S. *Cluster Analysis*. New York: Halsted Press, 1993. ISBN 0-470-22043-0

Kachigan, Sam Kash. *Multivariate Statistical Analysis*. New York: Radius Press, 1991. ISBN 0-942154-91-6



Index