IDL

# Image Processing in IDL

RSI
Research Systems Inc.

IDL Version 6.1
July, 2004 Edition

0704IDL61IP

## Acknowledgments

IDL® is a registered trademark and ION™, ION Script™, ION Java™, are trademarks of Research Systems Inc., registered in the United States Patent and Trademark Office, for the computer program described herein.

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

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.

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

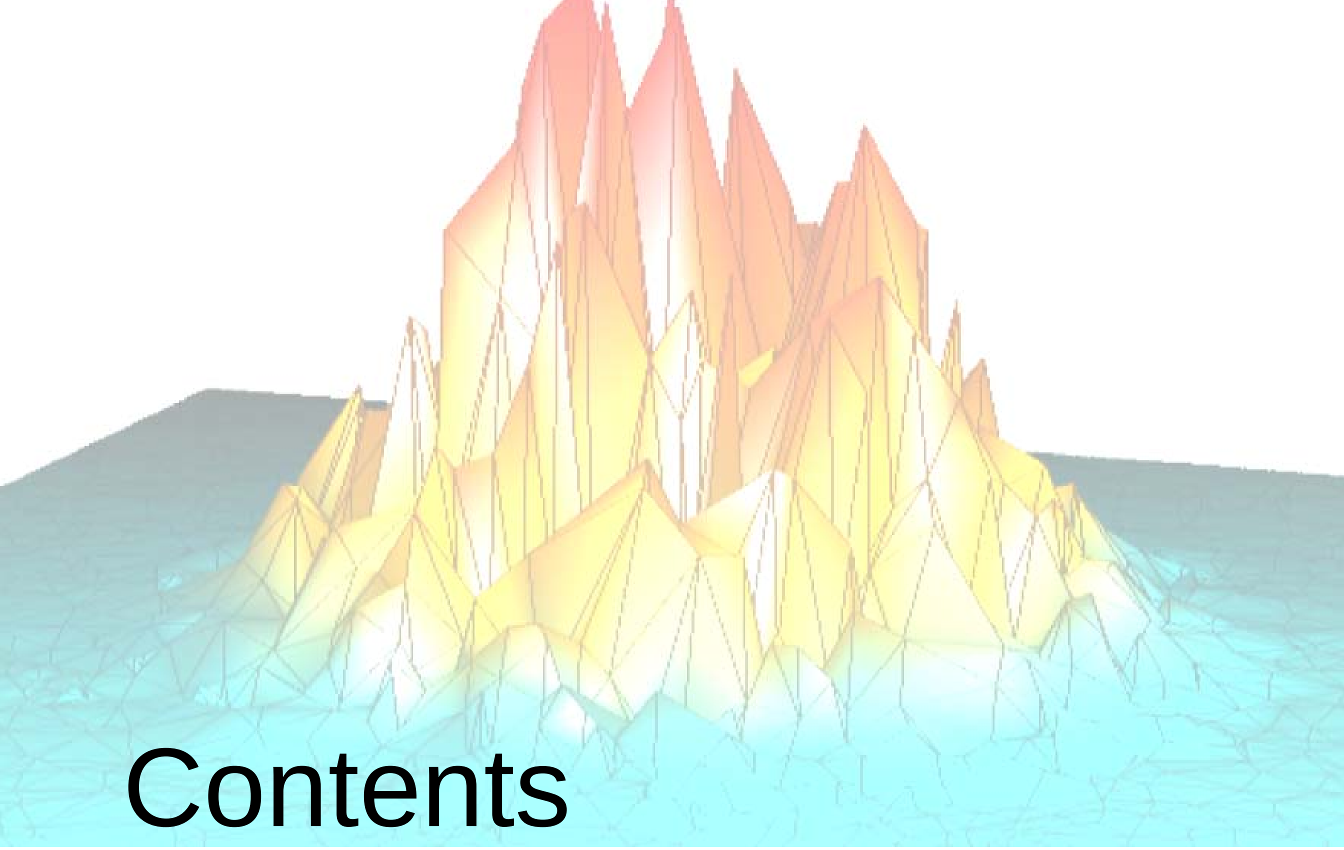
Portions of this computer program are copyright © 1995-1999 LizardTech, Inc. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

Portions of this software are copyrighted by Merge Technologies Incorporated.

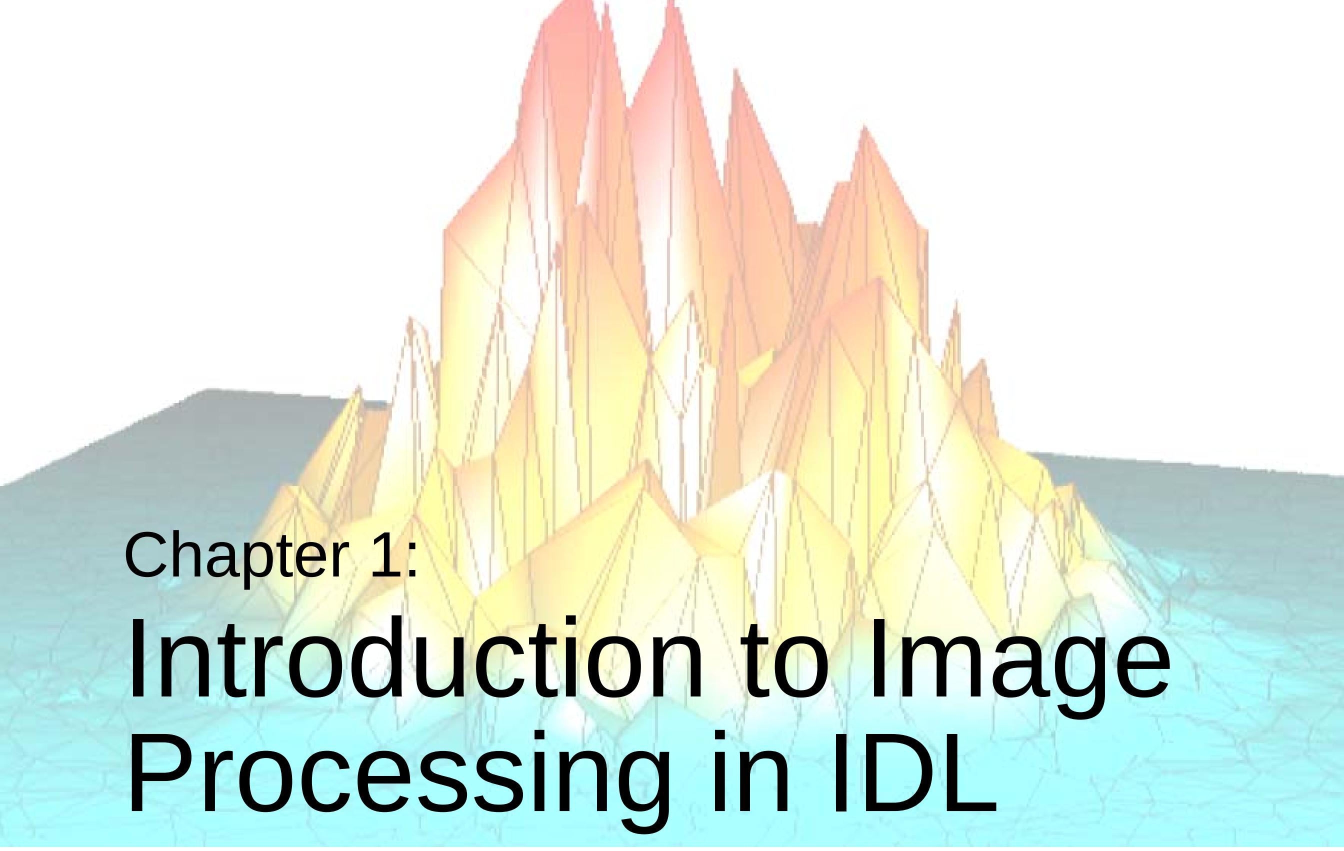
This product includes software developed by the Apache Software Foundation (http://www.apache.org/)

IDL Wavelet Toolkit Copyright © 2002 Christopher Torrence.

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



# Contents



# Chapter 1: Introduction to Image Processing in IDL

This chapter describes the following topics:

## Overview of Image Processing

Today, the medical industry, astronomy, physics, chemistry, forensics, remote sensing, manufacturing, and defense are just some of the many fields that rely upon images to store, display, and provide information about the world around us. The challenge to scientists, engineers and business people is to quickly extract valuable information from raw image data. This is the primary purpose of image processing – converting images to information.

This book explains how to process images using IDL (Interactive Data Language). IDL is a high-level programming language that contains an extensive library of image processing and analysis routines. With IDL, you can quickly access image data and begin investigating the best way to extract useful information.

Each chapter introduces image processing topics and includes information regarding when one method may be preferred over another to enhance specific image features. Numerous step-by-step examples illustrate IDL's image processing and analysis routines, allowing you to quickly understand how to get the desired results when working with your own image data. This book is not intended to be a complete source for image processing knowledge, an advanced image processing manual or an image processing reference guide. This book is designed to teach people how to use IDL to perform basic image processing, and does not assume that they are already experts in the field of image processing.

### Digital Images and Image Processing

A digital image is composed of a grid of pixels and stored as an array. A single pixel represents a value of either light intensity or color. Images are processed to obtain information beyond what is apparent given the image's initial pixel values. Image processing tasks can include any combination of the following:

**Accessing Image Data** — Image data must be displayed to initially determine what features are to be extracted or what problem needs to be solved. After processing, the image should be displayed to verify the results. Chapter 2, "Creating Image Displays" details how to create Direct and Object Graphic displays containing binary, indexed and RGB images.

**Enhancing Images Using Color** — Color can be a powerful tool for extracting previously unseen information from images. Chapter 3, "Working with Color" describes how to display images with inherent color information and alter the colors to highlight specific features. Color can also be used to display additional information, such as a legend describing the meaning of color values.

**Modifying the Image View** — Transforming, translating, rotating and resizing images are common tasks used to focus the viewer's attention on a specific area of the image. Chapter 4, "Transforming Image Geometry" provides information on how to precisely position images using IDL.

**Adding Dimensionality to Image Data** — Some images provide more information when they are placed on a polygon, surface, or geometric shape such as a sphere. Chapter 5, "Mapping an Image onto Geometry" shows how to display images over surfaces and geometric shapes.

**Working with Masks and Calculating Statistics** — Image processing uses some fundamental mathematical methods to alter image arrays. These include masking, clipping, locating, and statistics. Chapter 6, "Working with Masks and Image Statistics" introduces these operations and provides examples of masking and calculating image statistics.

**Warping Images** — Some data acquisition methods can introduce an unwanted curvature into an image. Image warping using control points can realign an image along a regular grid or align two images captured from different perspectives. See Chapter 7, "Warping Images" for more information.

**Specifying Regions of Interest (ROIs)** — When processing an image, you may want to concentrate on a specific region of interest (ROI). ROIs can be determined, displayed, and analyzed within IDL as described in Chapter 8, "Working with Regions of Interest (ROIs)".

**Manipulating Images in Various Domains** — One of the most useful tools in image processing is the ability to transform an image from one domain to another. Additional information can be derived from images displayed in frequency, time-frequency, Hough, and Radon domains. Moreover, some complex processing tasks are simpler within these domains. See Chapter 9, "Transforming Between Domains" for details.

**Enhancing Contrast and Filtering** — Contrasting and filtering provide the ability to smooth, sharpen, enhance edges and reduce noise within images. See Chapter 10, "Contrasting and Filtering" for details on manipulating contrast and applying filters to highlight and extract specific image features.

**Extracting and Analyzing Shapes** — Morphological operations provide a means of determining underlying image structures. Used in combination, these routines provide the ability to highlight, extract, and analyze features within an image. See Chapter 11, "Extracting and Analyzing Shapes" for details.

Before processing images, it is important to understand how images are defined, how image data is represented, and how images are accessed (imported and exported) within IDL. These topics are described within the following sections of this chapter:

- “Understanding Image Definitions in IDL” on page 15
- “Representing Image Data in IDL” on page 16
- “Accessing Images” on page 18

## Understanding Image Definitions in IDL

An understanding of basic image definitions is necessary before proceeding with image processing tasks. Some routines are specifically designed for certain types of images. Binary, grayscale, and indexed images are two-dimensional arrays, while RGB images are three-dimensional arrays. In which group an image belongs is determined by its contents and how it relates to its color information.

Within IDL, an image can be categorized as follows:

| Image Type | Descriptions |
|---|---|
| Binary Images | Binary images contain only two values (off or on). The off value is usually a zero and the on value is usually a one. This type of image is commonly used as a multiplier to mask regions within another image. |
| Grayscale Images | Grayscale images represent intensities. Pixels range from least intense (black) to most intense (white). Pixel values usually range from 0 to 255 or are scaled to this range when displayed. |
| Indexed Images | Instead of intensities, a pixel value within an indexed image relates to a color value within a color lookup table. Since indexed images reference color tables composed of up to 256 colors, the data values of these images are usually scaled to range between 0 and 255. |
| RGB Images | Within the three-dimensional array of an RGB image, two of the dimensions specify the location of a pixel within an image. The other dimension specifies the color of each pixel The color dimension always has a size of 3 and is composed of the red, green, and blue color bands (channels) of the image. |

*Table 1-1: Image Definitions*

**Note**

Grayscale and binary images can actually be treated as indexed images with an associated grayscale color table.

Color information can also be represented in other forms, which are described in "Converting to Other Color Systems" in Chapter 3.

## Representing Image Data in IDL

Pixel values in an image file can be stored in many different data types. IDL maintains 15 different data types. The original data type of an image is reflected in IDL when importing the image, but the type can be converted once the image is stored in an IDL variable. The following types are commonly used for images:

- Byte — An 8-bit unsigned integer ranging in value from 0 to 255. Pixels in images are commonly represented as byte data.
- Unsigned Integer — A 16-bit unsigned integer ranging from 0 to 65535.
- Signed Integer — A 16-bit signed integer ranging from -32,768 to +32,767.
- Unsigned Longword Integer — A 32-bit unsigned integer ranging in value from 0 to approximately four billion.
- Longword Integer — A 32-bit signed integer ranging in value from approximately minus two billion to plus two billion.
- Floating-point — A 32-bit, single-precision, floating-point number in the range from $-10^{38}$ to $10^{38}$, with approximately 6 or 7 decimal places of significance.
- Double-precision — A 64-bit, double-precision, floating-point number in the range from $-10^{308}$ to $10^{308}$ with approximately 14 decimal places of significance.

While pixel values are commonly stored in files as whole numbers, they are usually converted to floating-point or double-precision data types prior to performing numerical computations. See "Converting RGB Images to Grayscale Images" in Chapter 3 and "Calculating Image Statistics" in Chapter 6 for more information.

IDL provides predefined routines to convert data from one type to another. These routines are shown in the following table:

| Function | Description |
|---|---|
| BYTE | Convert to byte |
| BYTSCL | Scale data to range from 0 to 255 and then convert to byte |
| UINT | Convert to 16-bit unsigned integer |
| FIX | Convert to 16-bit integer, or optionally other type |

*Table 1-2: Some IDL Data Type Conversion Functions*

| Function | Description |
|---|---|
| ULONG | Convert to 32-bit unsigned integer |
| LONG | Convert to 32-bit integer |
| FLOAT | Convert to floating-point |
| DOUBLE | Convert to double-precision floating-point |

*Table 1-2: Some IDL Data Type Conversion Functions (Continued)*

## Accessing Images

How an image is imported into IDL depends upon whether it is stored in an unformatted binary file or a common image file format. IDL can query and import image data contained in the following common image file formats:

- BMP
- DICOM
- JPEG
- MrSID
- PICT
- PNG
- PPM
- SRF
- TIFF

**Note**

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

Accessing unformatted binary files requires you to provide information about the data within the file such as dimension sizes, data arrangement, and data type. See "Importing Unformatted Image Files" on page 24 for more information.

### Querying Images

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, the path to the file can be determined with the FILEPATH function:

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

Now, you can use the QUERY_IMAGE function to return information about the file:

```
query = QUERY_IMAGE(file, info)
```

To determine the results of the QUERY_IMAGE function, you can print the value of the *query* variable:

```
PRINT, 'query = ', query
```

If *query* is zero, the file cannot be accessed with IDL. If *query* is one, the file can be accessed. IDL displays the following text in the Output Log:

```
query =           1
```

Because the query was successful, the *info* variable is now an IDL structure containing important 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:<br>• 1 – two-dimensional array<br>• 3 – three-dimensional array<br>Print the number of dimensions using:<br>`PRINT, 'Number of Channels: ', info.channels`<br>For the `mineral.png` file, IDL displays the following text in the Output Log:<br>`Number of Channels:           1` |
| `DIMENSIONS` | Contains image array information including the width and height. Print the image dimensions using:<br>`PRINT, 'Size: ', info.dimensions`<br>For the `mineral.png` file, IDL displays the following text in the Output Log:<br>`Size:           288         216` |

*Table 1-3: Image Structure Tag Information*

| Tag | Description |
|---|---|
| `HAS_PALETTE` | Describes the presence or absence of a color palette:<br>• 1 (True) – the image has an associated palette<br>• 0 (False) – the image does not have an associated palette<br>Print whether a palette is present or not using:<br>`PRINT, 'Is Palette Available?: ', info.has_palette`<br>For the `mineral.png` file, IDL displays the following text in the Output Log:<br>`Is Palette Available?:           1` |
| `IMAGE_INDEX` | Gives the zero-based index number of the current image. Print the index of the image using:<br>`PRINT, 'Image Index: ', info.image_index`<br>For the `mineral.png` file, IDL displays the following text in the Output Log:<br>`Image Index:           0` |
| `NUM_IMAGES` | Provides the number of images in the file. Print the number of images in the file using:<br>`PRINT, 'Number of Images: ', info.num_images`<br>For the `mineral.png` file, IDL displays the following text in the Output Log:<br>`Number of Images:           1` |

*Table 1-3: Image Structure Tag Information  (Continued)*

| Tag | Description |
|---|---|
| `PIXEL_TYPE` | Provides the IDL type code for the image data type. IDL type codes represent the following data types:<br>• 0 – Undefined<br>• 1 – Byte<br>• 2 – Integer<br>• 3 – Longword integer<br>• 4 – Floating point<br>• 5 – Double-precision floating<br>• 6 – Complex floating<br>• 7 – String (*does not apply for images*)<br>• 8 – Structure (*does not apply for images*)<br>• 9 – Double-precision complex<br>• 10 – Pointer (*does not apply for images*)<br>• 11 – Object reference (*does not apply for images*)<br>• 12 – Unsigned Integer<br>• 13 – Unsigned Longword Integer<br>• 14 – 64-bit Integer<br>• 15 – Unsigned 64-bit Integer<br>Print the data type of the pixels in the image using:<br>`PRINT, 'Data Type: ', info.pixel_type`<br>For the `mineral.png` file, IDL displays the following text in the Output Log:<br>`Data Type:           1` |
| `TYPE` | Identifies the image file format. Print the format of the file containing the image using:<br>`PRINT, 'File Type: ' + info.type`<br>For the `mineral.png` file, IDL displays the following text in the Output Log:<br>`File Type: PNG` |

*Table 1-3: 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 values.

In addition to the generic QUERY_IMAGE routine, IDL provides query functions for each of the following individual image file types:

- QUERY_BMP
- QUERY_DICOM
- QUERY_JPEG
- QUERY_MrSID
- QUERY_PICT
- QUERY_PNG
- QUERY_PPM
- QUERY_SRF
- QUERY_TIFF

These functions have the same syntax and usage as the QUERY_IMAGE function.

## Importing Formatted Image Files

Images stored in common image file formats (shown in the introduction to this section, "Accessing Images" on page 18) can be imported into IDL with the READ_IMAGE function. This function provides output arguments for red, green, and blue color table components, if available.

**Note**

You can use the QUERY_IMAGE function to determine the parameters of an image as described in "Querying Images" on page 18.

For example, the `rose.jpg` file is a JPEG image file that contains an RGB image. You can import this image using the READ_IMAGE function. First, you must determine the path to this file:

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

Now you can use the READ_IMAGE function to import the image:

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

IDL also provides individual READ_* routines for the following image file types:

- READ_BMP
- READ_DICOM
- READ_INTERFILE
- READ_JPEG
- READ_MrSID
- READ_PICT
- READ_PNG
- READ_PPM
- READ_SRF
- READ_TIFF

These routines are similar to the READ_IMAGE function, but provide more details for importing a specific image file if required.

**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.

## Exporting Formatted Image Files

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 individual WRITE_* routines for the following image file types:

- WRITE_BMP
- WRITE_INTERFILE
- WRITE_JPEG
- WRITE_PICT
- WRITE_PNG
- WRITE_PPM
- WRITE_SRF
- WRITE_TIFF

These routines are similar to the WRITE_IMAGE procedure, but provide more flexibility when exporting a specific image file type.

**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.

## 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 shown in the PIXEL_TYPE description in the previous table. 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, the owner of the file should already be familiar with the size and type parameters of any images they 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'])
```

You can 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)
```

## Exporting Unformatted Image Files

Images in unformatted binary files can be exported 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
```

## References

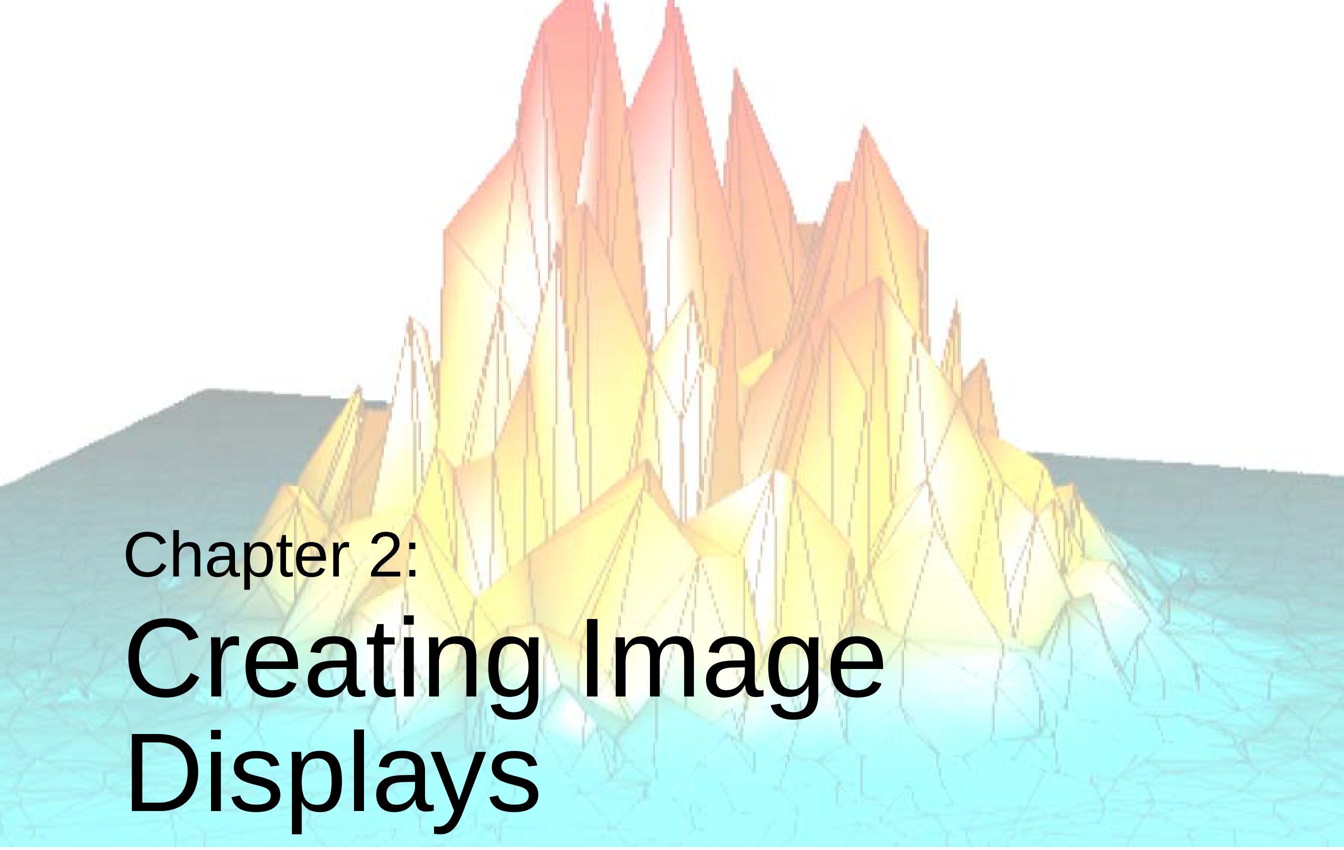
The following image processing sources were used in writing this book:

Baxes, Gregory A. *Digital Image Processing: Principles and Applications*. John Wiley & Sons. 1994. ISBN 0-471-00949-0

Lee, Jong-Sen. “Speckle Suppression and Analysis for Synthetic Aperture Radar Images”, *Optical Engineering*. vol. 25, no. 5, pp. 636 - 643. May 1986.

Russ, John C. *The Image Processing Handbook, Third Edition*. CRC Press LLC. 1999. ISBN 0-8493-2532-3

Weeks, Jr., Arthur R. *Fundamentals of Electronic Image Processing*. The Society of Photo-Optical Instrumentation Engineers. 1996. ISBN 0-8194-2149-9



Chapter 2:

# Creating Image Displays

This chapter describes the following topics:

## Overview of Creating Image Displays

To understand how to display an image, you must understand IDL's graphics systems, window coordinate systems, and the types of images you can display. IDL has two graphics systems, Direct Graphics and Object Graphics. Direct Graphics draws directly to a current device. Object Graphics renders graphical elements objects with instances of window objects. For more information, see "Differentiating Between Graphics Systems" on page 30.

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 Direct Graphics (see "Creating Direct Graphics Image Displays" on page 33) or with Object Graphics (see "Creating Object Graphics Image Displays" on page 46).

For information on how to display multiple images in the same window, see "Displaying Multiple Images in a Window" on page 62.

You can magnify (zoom in on) a specific area of an image by changing the display to show just that region. See "Zooming in on an Image" on page 73 for more information.

When you zoom in on a feature within an image, you may want to move along the feature at that magnification. The movement is known as panning. For more information on panning, see "Panning Within an Image" on page 80.

The following list introduces image display tasks and associated IDL image processing routines covered in this chapter.

| Task | Routine(s)/Object(s) | Description |
|---|---|---|
| "Creating Direct Graphics Image Displays" on page 33. | TV<br>TVSCL | Display binary, grayscale, indexed, and RGB images using the Direct Graphics system. |

*Table 2-1: Image Display Tasks and Related Image Display Routines.*

| Task | Routine(s)/Object(s) | Description |
|---|---|---|
| "Creating Object Graphics Image Displays" on page 46 | IDLgrImage<br>IDLgrPalette | Display binary, grayscale, indexed, and RGB images using the Object Graphics system. |
| "Displaying Multiple Images in a Window" on page 62. | TV<br>TVSCL<br>IDLgrImage | Display multiple images in a single Direct Graphics and Object Graphics window. |
| "Zooming in on an Image" on page 73. | ZOOM<br>ZOOM_24<br>IDLgrImage<br>IDLgrView | Magnify specific areas of an image using Direct and Object Graphics. |
| "Panning Within an Image" on page 80. | SLIDE_IMAGE<br>IDLgrImage<br>IDLgrView | Zoom in on specific areas of an image and then move to another area within the image using Direct and Object Graphics. |

*Table 2-1: Image Display Tasks and Related Image Display Routines.*

**Note**

This chapter uses data files from the `IDL examples/data` directory. Two files, `data.txt` and `index.txt`, contain descriptions of the files, including array sizes.

## Differentiating Between Graphics Systems

IDL supports two distinct graphics modes: Direct Graphics and Object Graphics. Direct Graphics relies on the concept of a current graphics device; IDL commands like TV or PLOT create displays directly on the current graphics device. Object Graphics uses an object-oriented programming interface to create graphic objects, which must then be explicitly drawn to a destination of the programmer’s choosing.

### Direct Graphics

The important aspects of Direct Graphics are:

- Direct Graphics uses 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.
- Commands like TV, PLOT, 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 recreate the graphic on a different device, you must reissue 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.

### Object Graphics

The important aspects of Object Graphics are:

- Object Graphics is 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 is object-oriented. Graphics objects are meant to be created and reused; 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 displays are rendered in three dimensions. 3D Rendering implies many operations not needed when drawing Direct Graphics displays, 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 uses a programmer's interface. Unlike Direct Graphics, which are well suited for both programming and interactive, ad hoc use, Object Graphics is 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 objects 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.

## 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.

## Creating Direct Graphics Image Displays

The procedure used to display an image in Direct Graphics depends upon the type of image to be displayed. Binary, grayscale, and indexed images are displayed with the TV or TVSCL procedures in Direct Graphics. The TV procedure displays the image in its original form. The TVSCL procedure displays the image scaled to range from 0 up to 255 depending on the colors available to IDL. RGB images are displayed with the TV procedure.

Examples of creating such displays are shown in the following sections:

- "Displaying Binary Images with Direct Graphics".
- "Displaying Grayscale Images with Direct Graphics" on page 35.
- "Displaying Indexed Images with Direct Graphics" on page 38.
- "Displaying RGB Images with Direct Graphics" on page 42.

### Displaying Binary Images with Direct Graphics

Binary images are composed of pixels having one of two values, usually zero or one. With most color tables, pixels having values of zero or one are displayed with almost the same color, such as with a the default grayscale color table. Thus, a binary image is usually scaled to display the zeros as black and the ones as white.

The following example imports a binary image of the world from the `continent_mask.dat` binary file. In this image, the oceans are zeros (black) and the continents are ones (white). This type of image can be used to mask out data over the oceans. The image contains byte data values and is 360 pixels by 360 pixels.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Binary Images with Direct Graphics" on page 35 or complete the following steps for a detailed description of the process.

1. Determine the path to the `continent_mask.dat` file:

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

2. Initialize the image size parameter:

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

3. Use READ_BINARY to import the image from the file:

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

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. This command implies a color table will be used. See "Foreground Color" in Chapter 3 for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

5. Load a grayscale color table:

   ```
   LOADCT, 0
   ```

6. Create a window and display the original image with the TV procedure:

   ```
   WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'A Binary Image, Not Scaled'
   TV, image
   ```

   The resulting window should be all black (blank). The binary image contains zeros and ones, which are almost the same color (black). Binary images should be displayed with the TVSCL procedure in order to scale the ones to white.

7. Create another window and display the scaled binary image:

   ```
   WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'A Binary Image, Scaled'
   TVSCL, image
   ```

   The following figure shows the results of scaling this display.

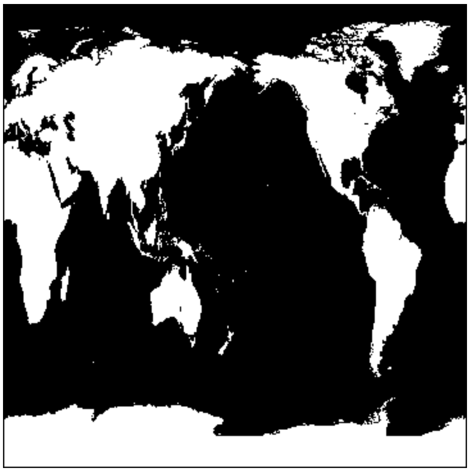


*Figure 2-1: Binary Image in Direct Graphics*

### Example Code: Displaying Binary Images with Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayBinaryImage_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayBinaryImage_Direct

; Determine the path to the file:
file = FILEPATH('continent_mask.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [360, 360]

; Import in the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize the display,
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the original image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'A Binary Image, Not Scaled'
TV, image

; Create another window and display the image scaled
; to range from 0 up to 255.
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'A Binary Image, Scaled'
TVSCL, image

END
```

## Displaying Grayscale Images with Direct Graphics

Features within grayscale images are created by pixels that have varying intensities. Pixel values range from least intense (black) to the most instance (white). Since a grayscale image is composed of pixels of varying intensities, it is best displayed with a color table that progresses linearly from black to white. Although IDL has several such predefined color tables, the grayscale color table (B-W LINEAR), is the most fitting choice when displaying grayscale images. IDL's B-W LINEAR color table is represented by an index value of 0. See "Loading Pre-defined Color Tables" in Chapter 3 for more information on IDL's predefined color tables.

The following example imports a grayscale image from the `convec.dat` binary file. This grayscale image shows the convection of the Earth's mantle. The image contains byte data values and is 248 pixels by 248 pixels. Since the data type is byte, this image does not need to be scaled before display. If the data was of any type other than byte and the data values were not within the range from 0 to 255, the image would need to be scaled prior to being displayed. See the TVSCL description *in the IDL Reference Guide* for more information.

For code that you can copy and paste into an Editor window, see "The following figure shows the resulting grayscale image display.Example Code: Displaying Grayscale Images with Direct Graphics" on page 37 or complete the following steps for a detailed description of the process.

1. Determine the path to the `convec.dat` file:

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

2. Initialize the image size parameter:

```
imageSize = [248, 248]
```

3. Using READ_BINARY, import the image from the file:

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

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "Foreground Color" in Chapter 3 for more information.

```
DEVICE, DECOMPOSED = 0
```

5. Load a grayscale color table:

```
LOADCT, 0
```

6. Create a window and display the original image with the TV procedure:

```
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'A Grayscale Image'
TV, image
```

The following figure shows the resulting grayscale image display.**Example Code:**

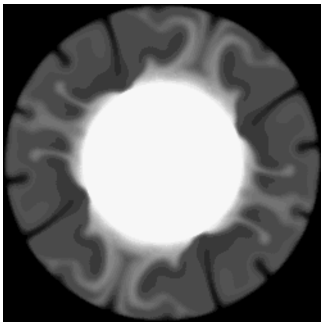


*Figure 2-2: Grayscale Image in Direct Graphics*

## Displaying Grayscale Images with Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayGrayscaleImage_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayGrayscaleImage_Direct

; Determine the path to the file.
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [248, 248]

; Import in the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'A Grayscale Image'
TV, image

END
```

## Displaying Indexed Images with Direct Graphics

An indexed image contains up to 256 colors, typically defined by a color table associated with the image. The value of each pixel relates to a color within the associated color table. Combinations of the primary colors (red, green, and blue) make up the colors within the color table. Most indexed images are stored as byte and therefore do not require scaling prior to display.

The following example imports an indexed image from the `avhrr.png` image file. This indexed image is a satellite photograph of the world.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Indexed Images with Direct Graphics" on page 41 or complete the following steps for a detailed description of the process.

1. Determine the path to the `avhrr.png` file:

```
file = FILEPATH('avhrr.png', $
   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 <141d0b0>, 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'
```

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

```
imageSize = imageInfo.dimensions
```

   The HAS_PALETTE tag has a value of 1. Thus, the image has a palette (color table), which is also contained within the file. The color table is made up of its three primary components (the red component, the green component, and the blue component).

5. Use READ_IMAGE to import the image and its associated color table from the file:

```
image = READ_IMAGE(file, red, green, blue)
```

6. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See “Foreground Color” in Chapter 3 for more information.

```
DEVICE, DECOMPOSED = 0
```

7. Load the red, green, and blue components of the image’s associated color table:

```
TVLCT, red, green, blue
```

8. Create a window and display the original image with the TV procedure:

```
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'An Indexed Image'
TV, image
```

9. Use the XLOADCT utility to display the associated color table:

```
XLOADCT
```

Click on the **Done** button of XLOADCT to exit out of the utility.

The following figure shows the resulting indexed image and its color table.

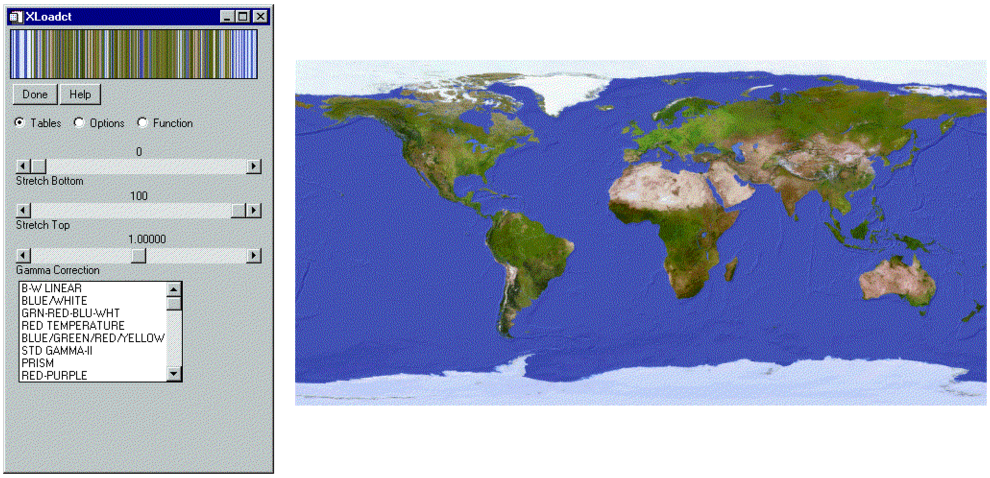


*Figure 2-3: Indexed Image and Associated Color Table in Direct Graphics*

The data values within the image are indexed to specific colors within the table. You can change the color table associated with this image to show how an indexed image is dependent upon its related color table.

10. Change the current color table to the EOS B pre-defined color table:

```
LOADCT, 27
```

11. Redisplay the image to show the color table change:

```
TV, image
```

**Note**

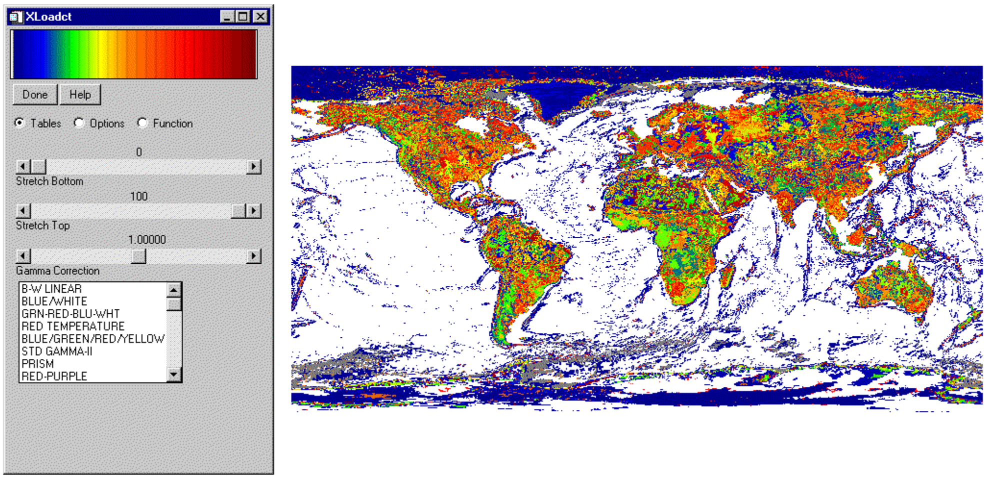
This step is not always necessary to redisplay the image. On PseudoColor (8-bit) or DirectColor systems, the display will update automatically when the current color table is changed.

12. Use the XLOADCT utility to display the current color table:

```
XLOADCT
```

Click on the **Done** button of XLOADCT to exit out of the utility.

The following figure shows the indexed image with the EOS B color table.



[✓ Tables] Options Function

0
Stretch Bottom

100
Stretch Top

1.00000
Gamma Correction

B-W LINEAR
BLUE/WHITE
GRN-RED-BLU-WHT
RED TEMPERATURE
BLUE/GREEN/RED/YELLOW
STD GAMMA-II
PRISM
RED-PURPLE

*Figure 2-4: Indexed Image and EOS B Color Table in Direct Graphics*

## Example Code: Displaying Indexed Images with Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayIndexedImage_Direct.pro`, compile and run the program to reproduce the previous example. The BLOCK keyword is set when using the XLOADCT utility to force the example routine to wait until the **Done** button is pressed to continue.

```
PRO DisplayIndexedImage_Direct

; Determine the path to the file.
file = FILEPATH('avhrr.png', $
   SUBDIRECTORY = ['examples', 'data'])

; Query the file to determine image parameters.
queryStatus = QUERY_IMAGE(file, imageInfo)

; Output the results of the file query.
PRINT, 'Query Status = ', queryStatus
HELP, imageInfo, /STRUCTURE

; Set image size parameter.
imageSize = imageInfo.dimensions

; Import in the image and its associated color table
; from the file.
image = READ_IMAGE(file, red, green, blue)

; Initialize the display.
DEVICE, DECOMPOSED = 0
TVLCT, red, green, blue

; Create a window and display the image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'An Indexed Image'
TV, image

; Use the XLOADCT utility to display the color table.
XLOADCT, /BLOCK

; Change the color table to the EOS B pre-defined color
; table.
LOADCT, 27

; Redisplay the image with the EOS B color table.
TV, image

; Use the XLOADCT utility to display the current color
; table.
XLOADCT, /BLOCK

END
```

## Displaying RGB Images with Direct Graphics

RGB images are three-dimensional arrays made up of width, height, and three channels of color information. In Direct Graphics, these images are displayed with the TV procedure. The TRUE keyword to TV is set according to the interleaving of the RGB image. With RGB images, the interleaving, or arrangement of the channels within the image file, dictates the setting of the TRUE keyword. If the image is:

- pixel interleaved (3, w, h), TRUE is set to 1.
- line interleaved (w, 3, h), TRUE is set to 2.
- planar interleaved (w, h, 3), TRUE is set to 3.

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.

If you are using a PseudoColor display, your RGB images must be converted to indexed images to be displayed within IDL. See "Foreground Color" in Chapter 3 for more information on RGB images and PseudoColor displays.

The following example queries and imports a pixel-interleaved RGB image from the `rose.jpg` image file. This pixel interleaved RGB image is a close-up photograph of a red rose.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying RGB Images with Direct Graphics" on page 44 or complete the following steps for a detailed description of the process.

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])) + 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 = 1
   ```

   The image is pixel interleaved. If the resulting value was 2, the image would have been line interleaved. If the resulting value was 3, the image would have been planar interleaved.

9. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to one before your first RGB image is displayed within an IDL session or program. See "Foreground Color" in Chapter 3 for more information:

```
DEVICE, DECOMPOSED = 1
```

10. Create a window and display the image with the TV procedure:

```
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'An RGB Image'
TV, image, TRUE = interleaving[0]
```

The following figure shows the resulting RGB image display.

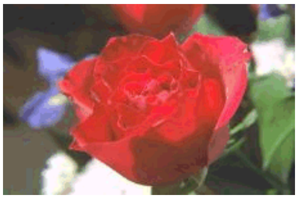


*Figure 2-5: RGB Image in Direct Graphics*

## Example Code: Displaying RGB Images with Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayRGBImage_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayRGBImage_Direct

; Determine the path to the file.
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Query the file to determine image parameters.
queryStatus = QUERY_IMAGE(file, imageInfo)

; Output the results of the file query.
PRINT, 'Query Status = ', queryStatus
HELP, imageInfo, /STRUCTURE

; Set the image size parameter from the query
; information.
imageSize = imageInfo.dimensions
```

```
; Import the image.
image = READ_IMAGE(file)

; Determine the size of each dimension within the image.
imageDims = SIZE(image, /DIMENSIONS)

; Determine the type of interleaving by comparing the
; dimension sizes with the image size parameter from the
; file query.
interleaving = WHERE((imageDims NE imageSize[0]) AND $
   (imageDims NE imageSize[1])) + 1

; Output the results of the interleaving computation.
PRINT, 'Type of Interleaving = ', interleaving

; Initialize display.
DEVICE, DECOMPOSED = 1

; Create a window and display the image with the TV
; procedure and its TRUE keyword.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'An RGB Image'
TV, image, TRUE = interleaving[0]

END
```

# Creating Object Graphics Image Displays

In Object Graphics, binary, grayscale, indexed, and RGB images are contained in image objects. For display, the image object is contained within an object hierarchy, which includes a model object and a view object. The view object is then drawn to a window object. Some types of images must be scaled with the BYTSCL function prior to display.

This section includes the following examples:

- "Displaying Binary Images with Object Graphics".
- "Displaying Grayscale Images with Object Graphics" on page 49.
- "Displaying Indexed Images with Object Graphics" on page 52.
- "Displaying RGB images with Object Graphics" on page 57.

## Displaying Binary Images with Object Graphics

Binary images are composed of pixels having one of two values, usually zero or one. With most color tables, pixels having values of zero and one are displayed with almost the same color, such as with the default grayscale color table. Thus, a binary image is usually scaled to display the zeros as black and the ones as white.

The following example imports a binary image of the world from the `continent_mask.dat` binary file. In this image, the oceans are zeros (black) and the continents are ones (white). This type of image can be used to mask out (omit) data over the oceans. The image contains byte data values and is 360 pixels by 360 pixels.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Binary Images with Object Graphics" on page 48 or complete the following steps for a detailed description of the process.

1. Determine the path to the `continent_mask.dat` file:

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

2. Initialize the image size parameter:

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

3. Use READ_BINARY to import the image from the file:

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

4. Initialize the display objects:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Binary Image, Not Scaled')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')
```

5. Initialize the image object:

```
oImage = OBJ_NEW('IDLgrImage', image)
```

6. 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 resulting window should be all black (blank). The binary image contains zeros and ones, which are almost the same color (black). A binary image should be scaled prior to displaying in order to show the ones as white.

7. Initialize another window:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Binary Image, Scaled')
```

8. Update the image object with a scaled version of the image:

```
oImage -> SetProperty, DATA = BYTSCL(image)
```

9. Display the view in the window:

```
oWindow -> Draw, oView
```

The following figure shows the results of scaling this display.

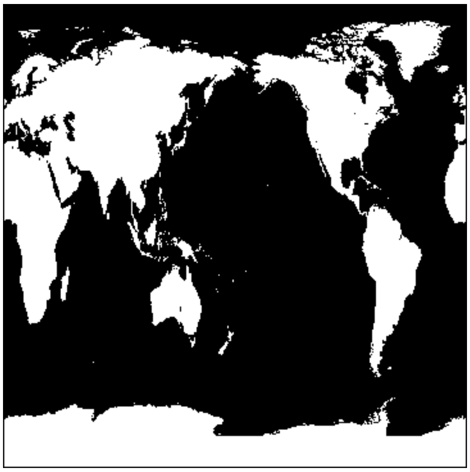


*Figure 2-6: Binary Image in Object Graphics*

10. 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
```

## Example Code: Displaying Binary Images with Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayBinaryImage_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayBinaryImage_Object

; Determine the path to the file.
file = FILEPATH('continent_mask.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [360, 360]

; Import the image.
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

```
; Initialize display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Binary Image, Not Scaled')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize image object.
oImage = OBJ_NEW('IDLgrImage', image)

; Add the image to the model, which is added to the
; view, and then display the view in the window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Initialize another window.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Binary Image, Scaled')

; Update the image object with a scaled version of the
; image.
oImage -> SetProperty, DATA = BYTSCL(image)

; Display the view in the window.
oWindow -> Draw, oView

; Clean up object references.
OBJ_DESTROY, oView

END
```

## Displaying Grayscale Images with Object Graphics

Since grayscale images are composed of pixels of varying intensities, they are best displayed with color tables that progress linearly from black to white. IDL provides several such pre-defined color tables, but the default grayscale color table is generally suitable.

The following example imports a grayscale image from the `convec.dat` binary file. This grayscale image shows the convection of the Earth's mantle. The image contains byte data values and is 248 pixels by 248 pixels. Since the data type is byte, this image does not need to be scaled before display. If the data was of any type other than byte and the data values were not within the range of 0 up to 255, the display would need to scale the image in order to show its intensities.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Grayscale Images with Object Graphics" on page 51 or complete the following steps for a detailed description of the process.

1. Determine the path to the `convec.dat` file:

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

2. Initialize the image size parameter:

```
imageSize = [248, 248]
```

3. Using READ_BINARY, import the image from the file:

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

4. Initialize the display objects:

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

5. Initialize the image object:

```
oImage = OBJ_NEW('IDLgrImage', image, /GREYSCALE)
```

6. 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 grayscale image display

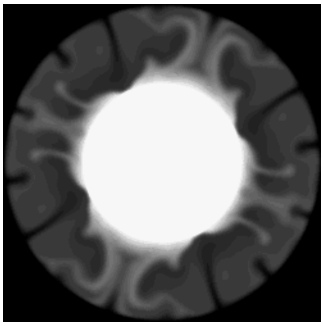


*Figure 2-7: Grayscale Image in Object Graphics*

7. 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
```

## Example Code: Displaying Grayscale Images with Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayGrayscaleImage_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayGrayscaleImage_Object

; Determine the path to the file.
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameters.
imageSize = [248, 248]

; Import the image.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Grayscale Image')
```

```
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize image object.
oImage = OBJ_NEW('IDLgrImage', image, $
   /GREYSCALE)

; 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

; Clean up object references.
OBJ_DESTROY, oView

END
```

## Displaying Indexed Images with Object Graphics

An indexed image contains up to 256 colors, typically defined by a color table associated with the image. The value of each pixel relates to a color within the associated color table. Combinations of the primary colors (red, green, and blue) make up the colors within the color table. Most indexed images are stored as byte and therefore do not require scaling prior to display.

The following example imports an indexed image from the `avhrr.png` image file. This indexed image is a satellite photograph of the world.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Indexed Images with Object Graphics" on page 56 or complete the following steps for a detailed description of the process.

1. Determine the path to the `avhrr.png` file:

   ```
   file = FILEPATH('avhrr.png', $
      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 <141d0b0>, 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'
```

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

```
imageSize = imageInfo.dimensions
```

The HAS_PALETTE tag has a value of 1. Thus, the image has a palette (color table), which is also contained within the file. The color table is made up of its three primary components (the red component, the green component, and the blue component).

5. Use READ_IMAGE to import the image and its associated color table from the file:

```
image = READ_IMAGE(file, red, green, blue)
```

6. Initialize the display objects:

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

7. Initialize the image's palette object:

```
oPalette = OBJ_NEW('IDLgrPalette', red, green, blue)
```

8. Initialize the image object with the resulting palette object:

```
oImage = OBJ_NEW('IDLgrImage', image, $
   PALETTE = oPalette)
```

9. 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
```

10. Use the colorbar object to display the associated color table in another window:

```
oCbWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [256, 48], $
   TITLE = 'Original Color Table')
oCbView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., 256., 48.])
oCbModel = OBJ_NEW('IDLgrModel')
oColorbar = OBJ_NEW('IDLgrColorbar', PALETTE = oPalette, $
   DIMENSIONS = [256, 16], SHOW_AXIS = 1)
oCbModel -> Add, oColorbar
oCbView -> Add, oCbModel
oCbWindow -> Draw, oCbView
```

The following figure shows the resulting indexed image and its color table.

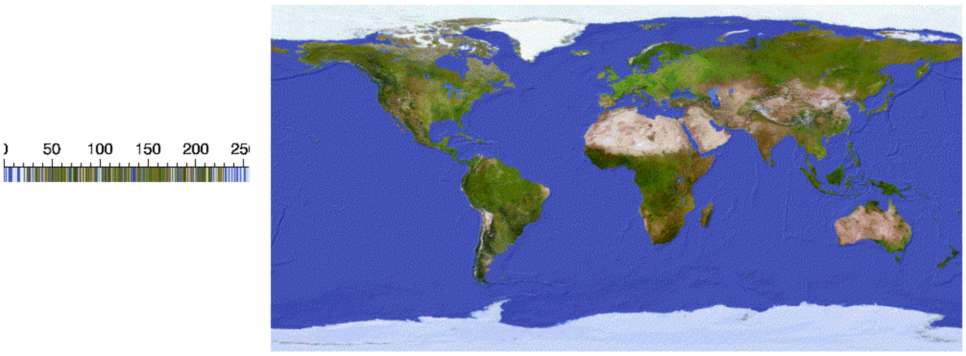


*Figure 2-8: Indexed Image and Associated Color Table in Object Graphics*

The data values within the image are indexed to specific colors within the table. You can change the color table associated with this image to show how an indexed image is dependent upon its related color tables.

11. Change the palette (color table) to the EOS B pre-defined color table:

```
oPalette -> LoadCT, 27
```

12. Redisplay the image in another window to show the palette change:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'An Indexed Image')
oWindow -> Draw, oView
```

13. Redisplay the colorbar in another window to show the palette change:

```
oCbWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [256, 48], $
   TITLE = 'EOS B Color Table')
oCbWindow -> Draw, oCbView
```

The following figure shows the indexed image with the EOS B color table.

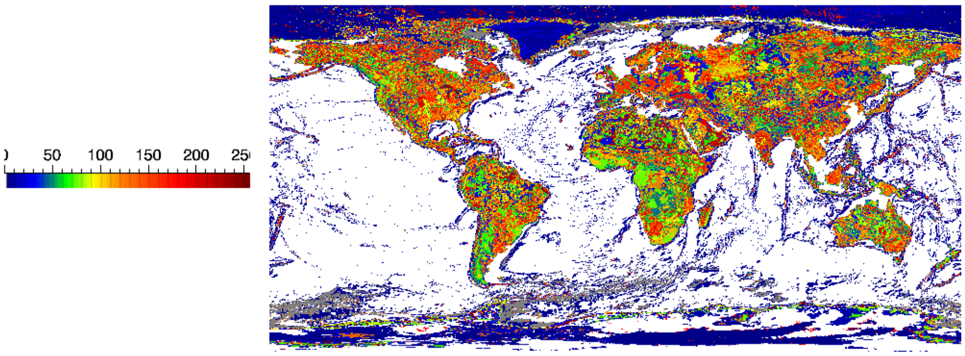


*Figure 2-9: Indexed Image and EOS B Color Table in Object Graphics*

14. 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, oCbVeiw, oPalette]
```

## Example Code: Displaying Indexed Images with Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayIndexedImage_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayIndexedImage_Object

; Determine the path to the file.
file = FILEPATH('avhrr.png', $
   SUBDIRECTORY = ['examples', 'data'])

; Query the file to determine image parameters.
queryStatus = QUERY_IMAGE(file, imageInfo)

; Output the results of the query.
PRINT, 'Query Status = ', queryStatus
HELP, imageInfo, /STRUCTURE

; Set the image size parameter.
imageSize = imageInfo.dimensions

; Import in the image.
image = READ_IMAGE(file, red, green, blue)

; Initialize the display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'An Indexed Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize the image's palette object.
oPalette = OBJ_NEW('IDLgrPalette', red, green, blue)

; Initialize the image object with the resulting
; palette object.
oImage = OBJ_NEW('IDLgrImage', image, $
   PALETTE = oPalette)

; 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

; Use the colorbar object to display the associated
; color table in another window.
oCbWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [256, 48], $
   TITLE = 'Original Color Table')
oCbView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., 256., 48.])
oCbModel = OBJ_NEW('IDLgrModel')
oColorbar = OBJ_NEW('IDLgrColorbar', PALETTE = oPalette, $
   DIMENSIONS = [256, 16], SHOW_AXIS = 1)
oCbModel -> Add, oColorbar
oCbView -> Add, oCbModel
oCbWindow -> Draw, oCbView

; Change the palette (color table) to the EOS B
; pre-defined color table.
oPalette -> LoadCT, 27

; Redisplay the image with the other color table in
; another window.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'An Indexed Image')
oWindow -> Draw, oView

; Redisplay the colorbar with the other color table
; in another window.
oCbWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [256, 48], $
   TITLE = 'EOS B Color Table')
oCbWindow -> Draw, oCbView

; Clean up object references.
OBJ_DESTROY, [oView, oCbView, oPalette]

END
```

## Displaying RGB images with Object Graphics

RGB images are three-dimensional arrays made up of width, height, and three channels of color information. In Object Graphics, an RGB image is contained within an image object. The interleaving, or arrangement of the channels within the image file, dictates the setting of the INTERLEAVE property of the image object. If the image is:

- pixel interleaved (3, w, h), INTERLEAVE is set to 0.
- line interleaved (w, 3, h), INTERLEAVE is set to 1.
- planar interleaved (w, h, 3), INTERLEAVE is set to 2.

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.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying RGB Images with Object Graphics" on page 60 or complete the following steps for a detailed description of the process.

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.

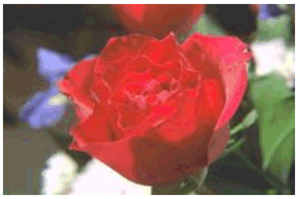


*Figure 2-10: RGB Image in Object Graphics*

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
```

## Example Code: Displaying RGB Images with Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayRGBImage_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayRGBImage_Object

; Determine the path to the file.
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Query the file to determine image parameters.
queryStatus = QUERY_IMAGE(file, imageInfo)

; Output the results of the query.
PRINT, 'Query Status = ', queryStatus
HELP, imageInfo, /STRUCTURE

; Set the image size parameter from the query
; information.
imageSize = imageInfo.dimensions

; Import in the image.
image = READ_IMAGE(file)

; Determine the size of each dimension within the image.
```

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

; Determine the type of interleaving by comparing
; dimension size and the size of the image.
interleaving = WHERE((imageDims NE imageSize[0]) AND $
   (imageDims NE imageSize[1]))

; Output the results of the interleaving computation.
PRINT, 'Type of Interleaving = ', interleaving

; 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')

; Initialize the image object.
oImage = OBJ_NEW('IDLgrImage', image, $
   INTERLEAVE = interleaving[0])

; 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

; Clean up object references.
OBJ_DESTROY, oView

END
```

## Displaying Multiple Images in a Window

How multiple images are displayed in a single window depends upon which graphics system is being used to display the images. Direct Graphics uses location input arguments for the TV procedure to position images in a window. See "Displaying Multiple Images in Direct Graphics" for more information. Object Graphics uses the LOCATION keyword to the Init method of the image object to position images in a window. See "Displaying Multiple Images in Object Graphics" on page 66 for more information.

### Displaying Multiple Images in Direct Graphics

The following example imports an RGB image from the `rose.jpg` image file. This RGB image is a close-up photograph of a red rose and is pixel interleaved. This example extracts the three color channels of this image, and displays them as grayscale images in various locations within the same window.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Multiple Images in Direct Graphics" on page 65 or complete the following steps for a detailed description of the process.

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. Set the image size parameter from the query information:

```
imageSize = imageInfo.dimensions
```

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

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

5. Extract the channels (as images) from the pixel interleaved RGB image:

```
redChannel = REFORM(image[0, *, *])
greenChannel = REFORM(image[1, *, *])
blueChannel = REFORM(image[2, *, *])
```

6. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See “Foreground Color” in Chapter 3 for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

7. Since the channels are grayscale images, load a grayscale color table:

   ```
   LOADCT, 0
   ```

   The TV procedure can be used to display the channels (grayscale images). The TV procedure has two different location input arguments. One argument is *position*. This argument arranges the image in a calculated location based on the size of the display and the dimension sizes of the image. See *TV in the IDL Reference Guide* for more information.

8. Create a window and horizontally display the three channels with the *position* argument:

   ```
   WINDOW, 0, XSIZE = 3*imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'The Channels of an RGB Image'
   TV, redChannel, 0
   TV, greenChannel, 1
   TV, blueChannel, 2
   ```

   The following figure shows the resulting grayscale images.

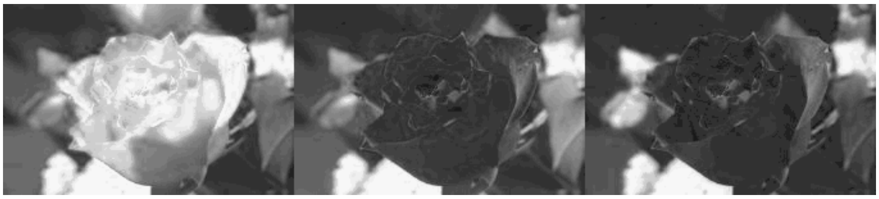


*Figure 2-11: Horizontal Display of RGB Channels in Direct Graphics*

The TV procedure can also be used with its *x* and *y* input arguments. These arguments define the location of the lower left corner of the image. The values of these arguments are in device coordinates by default. However, you can provide data or normalized coordinates when the DATA or NORMAL keyword is set. See *TV in the IDL Reference Guide* for more information.

9. Create a window and vertically display the three channels with the *x* and *y* arguments:

   ```
   WINDOW, 0, XSIZE = imageSize[0], YSIZE = 3*imageSize[1], $
      TITLE = 'The Channels of an RGB Image'
   TV, redChannel, 0, 0
   TV, greenChannel, 0, imageSize[1]
   TV, blueChannel, 0, 2*imageSize[1]
   ```

   The following figure shows the resulting grayscale images.

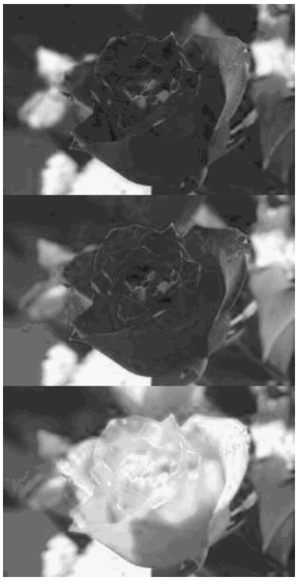


*Figure 2-12: Vertical Display of RGB Channels in Direct Graphics*

The *x* and *y* arguments can also be used to create a display of overlapping images. When overlapping images in Direct Graphics, you must remember the last image placed in the window will be in front of the previous images. So if you want to bring a display from the back of the window to the front, you must redisplay it after all the other displays.

10. Create another window:

    ```
    WINDOW, 2, XSIZE = 2*imageSize[0], YSIZE = 2*imageSize[1], $
       TITLE = 'The Channels of an RGB Image'
    ```

11. Make a white background to distinguish the edges of the images:

```
ERASE, !P.COLOR
```

12. Diagonally display the three channels with the *x* and *y* arguments:

```
TV, redChannel, 0, 0
TV, greenChannel, imageSize[0]/2, imageSize[1]/2
TV, blueChannel, imageSize[0], imageSize[1]
```

The following figure shows the resulting grayscale images.

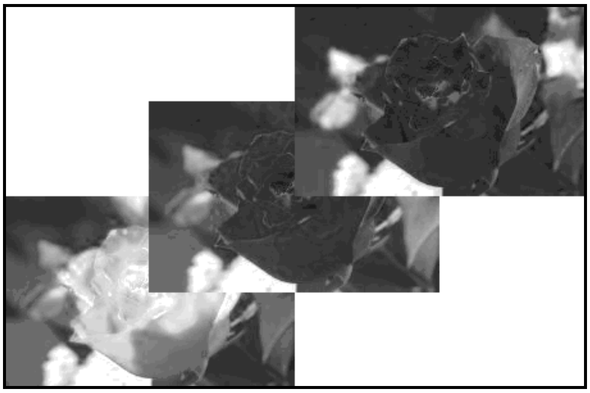


*Figure 2-13: Diagonal Display of RGB Channels in Direct Graphics*

## Example Code: Displaying Multiple Images in Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayMultiples_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayMultiples_Direct

; Determine the path to the file.
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Query the file to determine image parameters.
queryStatus = QUERY_IMAGE(file, imageInfo)

; Set the image size parameter from the query
; information.
imageSize = imageInfo.dimensions
```

```
; Import the image.
image = READ_IMAGE(file)

; Extract the channels (as images) from the RGB image.
redChannel = REFORM(image[0, *, *])
greenChannel = REFORM(image[1, *, *])
blueChannel = REFORM(image[2, *, *])

; Initialize displays.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and horizontally display the channels.
WINDOW, 0, XSIZE = 3*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'The Channels of an RGB Image'
TV, redChannel, 0
TV, greenChannel, 1
TV, blueChannel, 2

; Create another window and vertically display the
; channels.
WINDOW, 1, XSIZE = imageSize[0], YSIZE = 3*imageSize[1], $
   TITLE = 'The Channels of an RGB Image'
TV, redChannel, 0, 0
TV, greenChannel, 0, imageSize[1]
TV, blueChannel, 0, 2*imageSize[1]

; Create another window.
WINDOW, 2, XSIZE = 2*imageSize[0], YSIZE = 2*imageSize[1], $
   TITLE = 'The Channels of an RGB Image'

; Make a white background.
ERASE, !P.COLOR

; Diagonally display the channels.
TV, redChannel, 0, 0
TV, greenChannel, imageSize[0]/2, imageSize[1]/2
TV, blueChannel, imageSize[0], imageSize[1]

END
```

## Displaying Multiple Images in Object Graphics

The following example imports an RGB image from the `rose.jpg` image file. This RGB image is a close-up photograph of a red rose and is pixel interleaved. This example extracts the three color channels of this image, and displays them as grayscale images in various locations within the same window.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Multiple Images in Object Graphics" on page 70 or complete the following steps for a detailed description of the process.

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. Set the image size parameter from the query information:

```
imageSize = imageInfo.dimensions
```

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

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

5. Extract the channels (as images) from the pixel interleaved RGB image:

```
redChannel = REFORM(image[0, *, *])
greenChannel = REFORM(image[1, *, *])
blueChannel = REFORM(image[2, *, *])
```

   The LOCATION keyword to the Init method of the image object can be used to position an image within a window. The LOCATION keyword uses data coordinates, which are the same as device coordinates for images. Before initializing the image objects, you should initialize the display objects. The following steps display multiple images horizontally, vertically, and diagonally.

6. Initialize the display objects:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[3, 1], $
   TITLE = 'The Channels of an RGB Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 3, 1])
oModel = OBJ_NEW('IDLgrModel')
```

7. Now initialize the image objects and arrange them with the LOCATION keyword, see IDLgrImage for more information:

```
oRedChannel = OBJ_NEW('IDLgrImage', redChannel)
oGreenChannel = OBJ_NEW('IDLgrImage', greenChannel, $
   LOCATION = [imageSize[0], 0])
oBlueChannel = OBJ_NEW('IDLgrImage', blueChannel, $
   LOCATION = [2*imageSize[0], 0])
```

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

```
oModel -> Add, oRedChannel
oModel -> Add, oGreenChannel
oModel -> Add, oBlueChannel
oView -> Add, oModel
oWindow -> Draw, oView
```

   The following figure shows the resulting grayscale images.

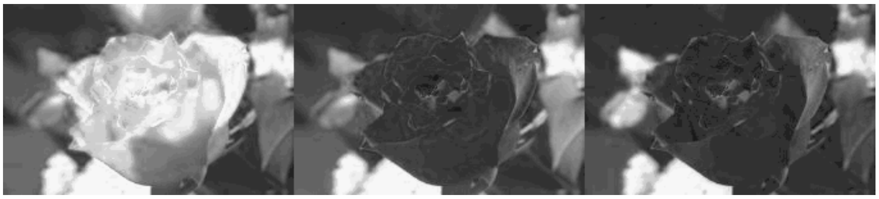


*Figure 2-14: Horizontal Display of RGB Channels in Object Graphics*

   These images can be displayed vertically in another window by first initializing another window and then updating the view and images with different location information.

9. Initialize another window object:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[1, 3], $
   TITLE = 'The Channels of an RGB Image')
```

10. Change the view from horizontal to vertical:

```
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 1, 3]
```

11. Change the locations of the channels:

```
oGreenChannel -> SetProperty, LOCATION = [0, imageSize[1]]
oBlueChannel -> SetProperty, LOCATION = [0, 2*imageSize[1]]
```

12. Display the updated view within the new window:

```
oWindow -> Draw, oView
```

The following figure shows the resulting grayscale images.

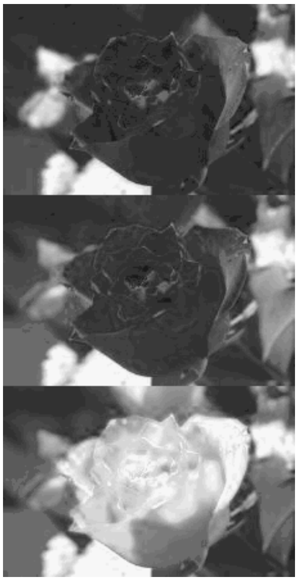


*Figure 2-15: Vertical Display of RGB Channels in Object Graphics*

These images can also be displayed diagonally in another window by first initializing the other window and then updating the view and images with different location information.The LOCATION can also be used to create a display overlapping images. When overlapping images in Object Graphics, you must remember the last image added to the model will be in front of the previous images.

13. Initialize another window object:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[2, 2], $
   TITLE = 'The Channels of an RGB Image')
```

14. Change the view to prepare for a diagonal display:

```
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 2, 2]
```

15. Change the locations of the channels:

```
oGreenChannel -> SetProperty, $
   LOCATION = [imageSize[0]/2, imageSize[1]/2]
oBlueChannel -> SetProperty, $
   LOCATION = [imageSize[0], imageSize[1]]
```

16. Display the updated view within the new window:

```
oWindow -> Draw, oView
```

The following figure shows the resulting grayscale images.

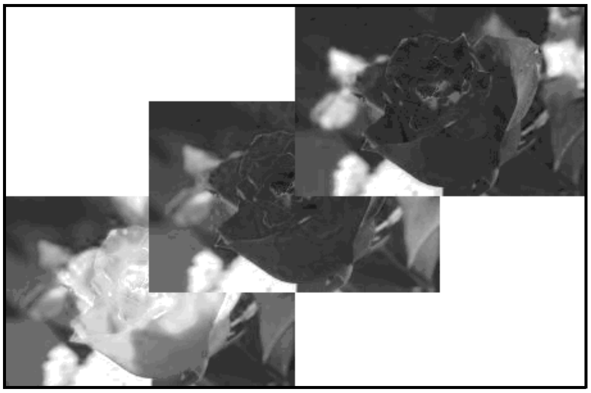


*Figure 2-16: Diagonal Display of RGB Channels in Object Graphics*

17. 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
```

## Example Code: Displaying Multiple Images in Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayMultiples_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayMultiples_Object

; Determine the path to the file.
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Query the file to determine image parameters.
queryStatus = QUERY_IMAGE(file, imageInfo)

; Set the image size parameter from the query
; information.
imageSize = imageInfo.dimensions

; Import the image.
image = READ_IMAGE(file)

; Extract the channels (as images) from the RGB image.
redChannel = REFORM(image[0, *, *])
greenChannel = REFORM(image[1, *, *])
blueChannel = REFORM(image[2, *, *])

; Horizontally display the channels.

; Initialize the display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[3, 1], $
   TITLE = 'The Channels of an RGB Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 3, 1])
oModel = OBJ_NEW('IDLgrModel')

; Initialize the image objects.
oRedChannel = OBJ_NEW('IDLgrImage', redChannel)
oGreenChannel = OBJ_NEW('IDLgrImage', greenChannel, $
   LOCATION = [imageSize[0], 0])
oBlueChannel = OBJ_NEW('IDLgrImage', blueChannel, $
   LOCATION = [2*imageSize[0], 0])

; Add the image objects to the model, which is added to
; the view, then display the view in the window.
oModel -> Add, oRedChannel
oModel -> Add, oGreenChannel
oModel -> Add, oBlueChannel
oView -> Add, oModel
oWindow -> Draw, oView

; Vertically display the channels.

; Initialize another window object.
```

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[1, 3], $
   TITLE = 'The Channels of an RGB Image')

; Change the view from horizontal to vertical.
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 1, 3]

; Change the locations of the channels.
oGreenChannel -> SetProperty, $
   LOCATION = [0, imageSize[1]]
oBlueChannel -> SetProperty, $
   LOCATION = [0, 2*imageSize[1]]

; Display the updated view in the new window.
oWindow -> Draw, oView

; Diagonally display the channels.

; Initialize another window object.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[2, 2], $
   TITLE = 'The Channels of an RGB Image')

; Change the view from vertical to diagonal.
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 2, 2]

; Change the locations of the channels.
oGreenChannel -> SetProperty, $
   LOCATION = [imageSize[0]/2, imageSize[1]/2]
oBlueChannel -> SetProperty, $
   LOCATION = [imageSize[0], imageSize[1]]

; Display the updated view in the new window.
oWindow -> Draw, oView

; Clean up object references.
OBJ_DESTROY, oView

END
```

# Zooming in on an Image

Enlarging a specific section of an image is known as zooming. How zooming is performed within IDL depends on the graphics system. In Direct Graphics, you can use the ZOOM procedure to zoom in on a specific section of an image. See "Zooming in on a Direct Graphics Image Display" for more information. If you are working with RGB images, you can use the ZOOM_24 procedure.

In Object Graphics, the VIEWPLANE_RECT keyword is used to change the view object. Using this method, the entire image is still contained within the image object, while the view is changed to only show specific areas of the image object. See "Zooming in on an Object Graphics Image Display" on page 76 for more information.

## Zooming in on a Direct Graphics Image Display

The following example imports a grayscale image from the `convec.dat` binary file. This grayscale image shows the convection of the Earth's mantle. The image contains byte data values and is 248 pixels by 248 pixels. The ZOOM procedure, which is a Direct Graphics routine, is used to zoom in on the lower left corner of the image.

For code that you can copy and paste into an Editor window, see "Example Code: Zooming in Direct Graphics" on page 75 or complete the following steps for a detailed description of the process.

1. Determine the path to the `convec.dat` file:

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

2. Initialize the image size parameter:

```
imageSize = [248, 248]
```

3. Import the image from the file:

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

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "Foreground Color" in Chapter 3 for more information.

```
DEVICE, DECOMPOSED = 0
```

5. Load a grayscale color table:

```
LOADCT, 0
```

6. Create a window and display the original image with the TV procedure:

```
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'A Grayscale Image'
TV, image
```

The following figure shows the resulting grayscale image display.

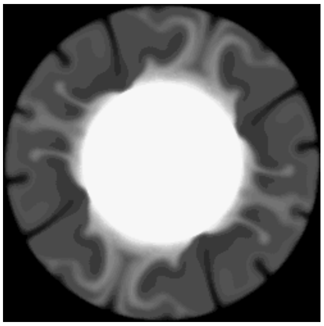


*Figure 2-17: A Grayscale Image in Direct Graphics*

7. Use the ZOOM to enlarge the lower left quarter of the image:

```
ZOOM, /NEW_WINDOW, FACT = 2, $
   XSIZE = imageSize[0], YSIZE = imageSize[1]
```

Click in the lower left corner of the original image window.

The following figure shows the resulting zoomed image.

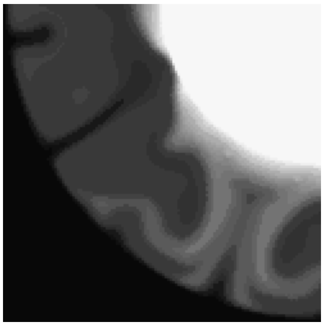


*Figure 2-18: Enlarged Image Area in Direct Graphics*

8. Right-click in the original image window to quit out of the ZOOM procedure.

## Example Code: Zooming in Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `Zooming_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO Zooming_Direct

; Determine the path to the file.
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [248, 248]

; Import in the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the image.
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'A Grayscale Image'
TV, image

; Zoom into the lower left quarter of the image.
```

```
ZOOM, /NEW_WINDOW, FACT = 2, $
   XSIZE = imageSize[0], YSIZE = imageSize[1]

END
```

## Zooming in on an Object Graphics Image Display

The following example imports a grayscale image from the `convec.dat` binary file. This grayscale image shows the convection of the Earth's mantle. The image contains byte data values and is 248 pixels by 248 pixels. The VIEWPLANE_RECT keyword to the view object is updated to zoom in on the lower left corner of the image.

For code that you can copy and paste into an Editor window, see "Example Code: Zooming in Object Graphics" on page 78 or complete the following steps for a detailed description of the process.

1. Determine the path to the `convec.dat` file:

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

2. Initialize the image size parameter:

   ```
   imageSize = [248, 248]
   ```

3. Import the image from the file:

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

4. Initialize the display objects:

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

5. Initialize the image object:

   ```
   oImage = OBJ_NEW('IDLgrImage', image, /GREYSCALE)
   ```

6. 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 grayscale image display.

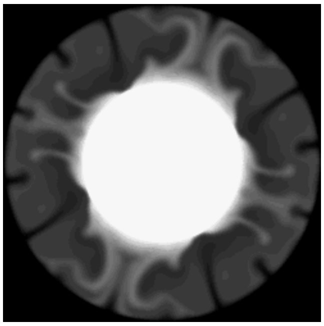


*Figure 2-19: A Grayscale Image in Object Graphics*

7. Initialize another window:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'Zoomed Image')
```

8. Change the view to enlarge the lower left quarter of the image:

```
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize/2]
```

   The view object still contains the entire image object, but the region displayed by the view (the viewplane rectangle) is reduced in size by half in both directions. Since the window object remains the same size, the view region is enlarged to fit it to the window.

9. Display the updated view in the new window:

```
oWindow -> Draw, oView
```

The following figure shows the resulting zoomed image.

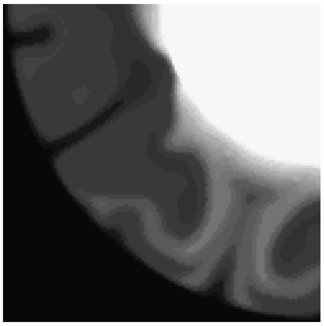


*Figure 2-20: Enlarged Image Area in Object Graphics*

10. 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
```

## Example Code: Zooming in Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `Zooming_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO Zooming_Object

; Determine the path to the file.
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [248, 248]

; Import in the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Grayscale Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize image object.
oImage = OBJ_NEW('IDLgrImage', image, /GREYSCALE)

; 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

; Initialize another window.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'Enlarged Area')

; Change view to zoom into the lower left quarter of
; the image.
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize/2]

; Display updated view in new window.
oWindow -> Draw, oView

; Clean up object references.
OBJ_DESTROY, oView

END
```

## Panning Within an Image

Panning involves moving an area of focus from one section of an image to other sections. How panning is performed within IDL depends on the graphics system. In Direct Graphics, you can use the SLIDE_IMAGE procedure to pan with sliders in an application that contains the image. See "Panning in Direct Graphics" for more information.

In Object Graphics, the VIEWPLANE_RECT keyword is used to change the view object. The entire image is still contained within the image object, but the view is changed to pan over specific areas of the image object. See "Panning in Object Graphics" on page 82 for more information.

### Panning in Direct Graphics

The following example imports a grayscale image from the `nyny.dat` binary file. This grayscale image is an aerial view of New York City. The image contains byte data values and is 768 pixels by 512 pixels. You can use the SLIDE_IMAGE procedure to zoom in on the image and pan over it.

For code that you can copy and paste into an Editor window, see "Example Code: Panning in Direct Graphics" on page 81 or complete the following steps for a detailed description of the process.

1. Determine the path to the `nyny.dat` file:

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

2. Initialize the image size parameter:

```
imageSize = [768, 512]
```

3. Import the image from the file:

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

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "Foreground Color" in Chapter 3 for more information.

```
DEVICE, DECOMPOSED = 0
```

5. Load a grayscale color table:

```
LOADCT, 0
```

6. Display the image with the SLIDE_IMAGE procedure:

   ```
   SLIDE_IMAGE, image
   ```

   Use the sliders in the display on the right side to pan over the image.

   The following figure shows a possible display within the SLIDE_IMAGE application.

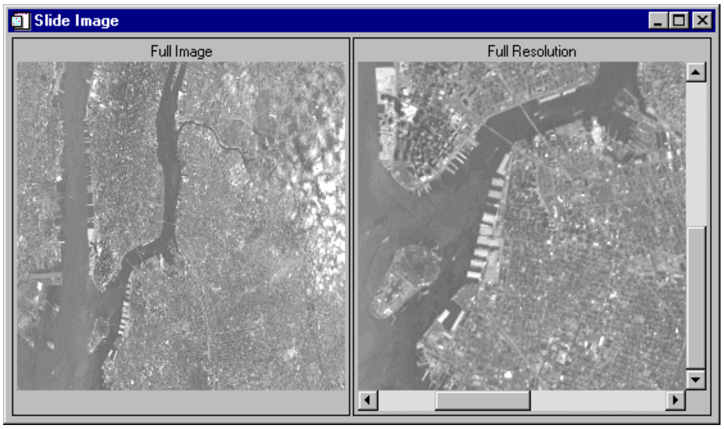


*Figure 2-21: The SLIDE_IMAGE Application Displaying an Image of New York*

## Example Code: Panning in Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `Panning_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO Panning_Direct

; Determine the path to the file.
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [768, 512]

; Import in the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize the display.
DEVICE, DECOMPOSED = 0
```

```
LOADCT, 0

; Display the image with the SLIDE_IMAGE procedure.
SLIDE_IMAGE, image

END
```

## Panning in Object Graphics

The following example imports a grayscale image from the `nyny.dat` binary file. This grayscale image is an aerial view of New York City. The image contains byte data values and is 768 pixels by 512 pixels. The VIEWPLANE_RECT keyword to the view object is updated to zoom in on the lower left corner of the image. Then the VIEWPLANE_RECT keyword is used to pan over the bottom edge of the image.

For code that you can copy and paste into an Editor window, see "Example Code: Panning in Object Graphics" on page 85 or complete the following steps for a detailed description of the process.

1. Determine the path to the `nyny.dat` file:

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

2. Initialize the image size parameter:

   ```
   imageSize = [768, 512]
   ```

3. Import the image from the file:

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

4. Resize this large image to entirely display it on the screen:

   ```
   imageSize = [256, 256]
   image = CONGRID(image, imageSize[0], imageSize[1])
   ```

5. Initialize the display objects:

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

6. Initialize the image object:

   ```
   oImage = OBJ_NEW('IDLgrImage', image, /GREYSCALE)
   ```

7. 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 grayscale image display.

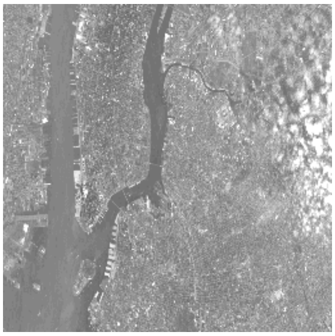


*Figure 2-22: A Grayscale Image Of New York in Object Graphics*

8. Initialize another window:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'Panning Enlarged Image')
```

9. Change the view to zoom into the lower left quarter of the image:

```
viewplane = [0., 0., imageSize/2]
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize/2]
```

The view object still contains the entire image object, but the region displayed by the view (the viewplane rectangle) is reduced in size by half in both directions. Since the window object remains the same size, the view region is enlarged to fit it to the window.

10. Display the updated view in the new window:

```
oWindow -> Draw, oView
```

The following figure shows the resulting enlarged image area.

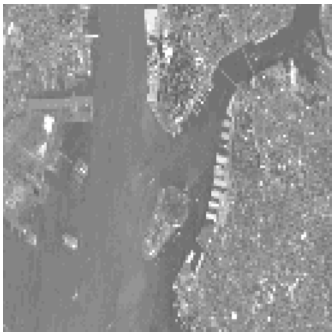


*Figure 2-23: Enlarged Image Area of New York in Object Graphics*

11. Pan the view from the left side of the image to the right side of the image:

```
FOR i = 0, ((imageSize[0]/2) - 1) DO BEGIN & $
   viewplane = viewplane + [1., 0., 0., 0.] & $
   oView -> SetProperty, VIEWPLANE_RECT = viewplane & $
   oWindow -> Draw, oView & $
ENDFOR
```

**Note**

The & after BEGIN and the `$` allow you to use the FOR/DO loop at the IDL command line. These & and `$` symbols are not required when the FOR/DO loop in placed in an IDL program as shown in "Example Code: Panning in Object Graphics" on page 85.

The following figure shows the resulting enlarged image area panned to the right side.

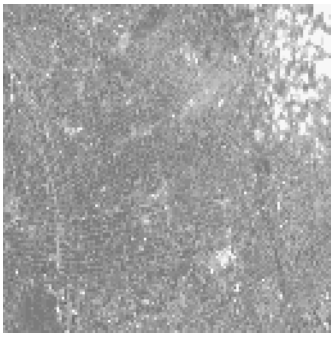


*Figure 2-24: Enlarged New York Image Area Panned to the Right in Object Graphics*

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.

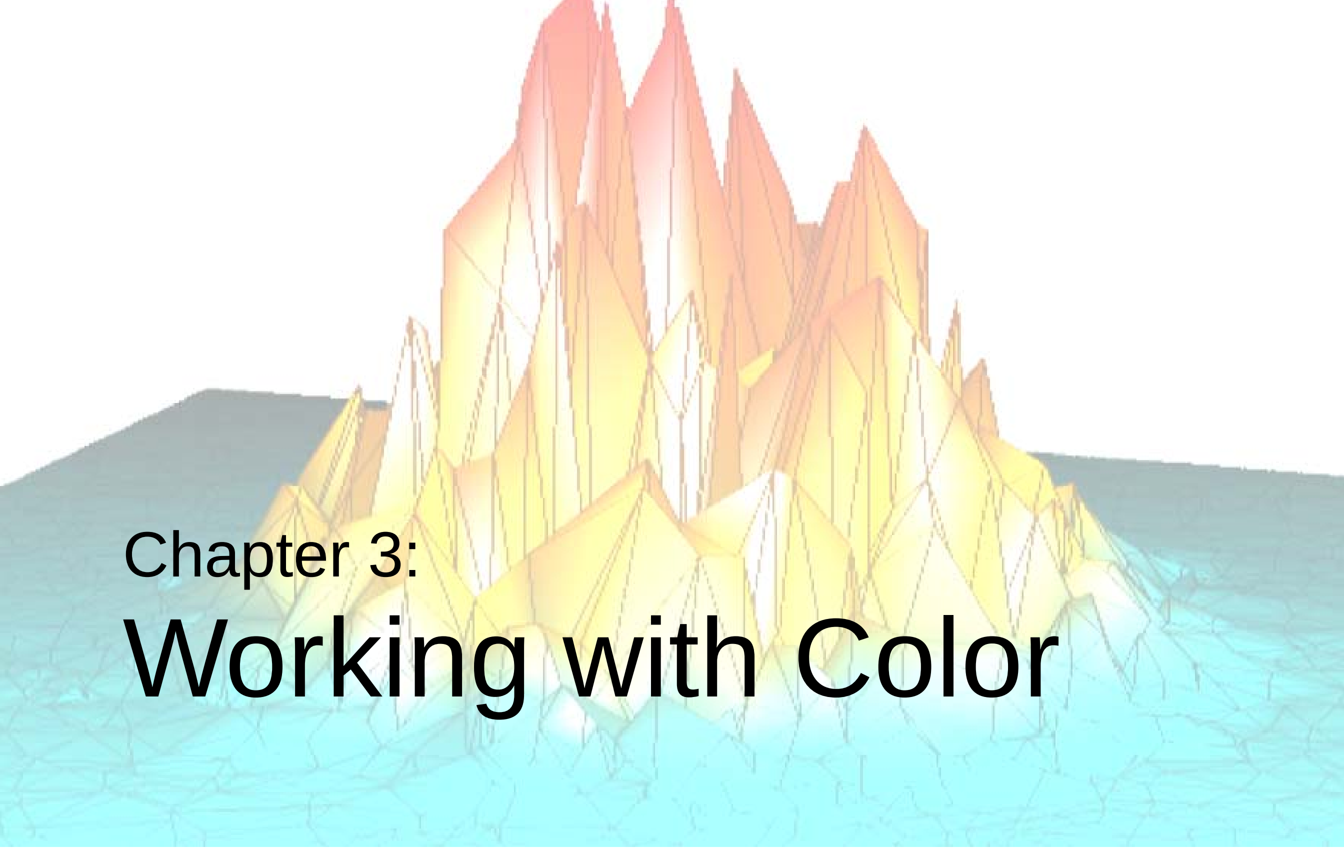
    ```
    OBJ_DESTROY, oView
    ```

## Example Code: Panning in Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `Panning_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO Panning_Object

; Determine the path to the file.
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [768, 512]

; Import in the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Resize the image.
imageSize = [256, 256]
```

```
image = CONGRID(image, imageSize[0], imageSize[1])

; Initialize display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Grayscale Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize image object.
oImage = OBJ_NEW('IDLgrImage', image, /GREYSCALE)

; 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

; Initialize another window.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'Panning Enlarged Image')

; Change view to zoom into the lower left quarter of
; the image.
viewplane = [0., 0., imageSize/2]
oView -> SetProperty, VIEWPLANE_RECT = viewplane

; Display updated view in new window.
oWindow -> Draw, oView

; Pan the view from the left side of the image to the
; right side of the image.
FOR i = 0, ((imageSize[0]/2) - 1) DO BEGIN
   viewplane = viewplane + [1., 0., 0., 0.]
   oView -> SetProperty, VIEWPLANE_RECT = viewplane
   oWindow -> Draw, oView
ENDFOR

; Clean up object references.
OBJ_DESTROY, oView

END
```



# Chapter 3: Working with Color

This chapter describes the following topics:

## Overview of Working with Color

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 Systems

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).

Computer display devices typically rely on the RGB color system. In IDL, the RGB color space is represented as a three-dimensional 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.

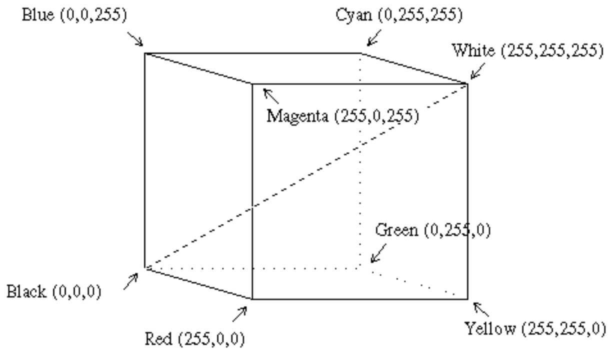


*Figure 3-1: RGB Color Cube (Note: grays are on the main diagonal.)*

## 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 2n 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.

## 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 3-1: Visuals Supported in IDL on Unix Platforms*

The most common of these is PseudoColor and TrueColor.

Refer to the section "Understanding Colors within IDL Graphic Systems" on page 94 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** control to open the Display Properties window. Within this window, select the **Settings** tab. The **Colors** menu lists the supported visuals. The following table shows that 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 3-2: Visuals Supported in IDL on Windows Platforms*

You can use this dialog to change between visuals before starting an IDL session.

## Image Data Organization

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.

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.

An RGB (red, green, blue) image is a three-dimensional byte array that explicitly stores a color value for each pixel. 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:

- Pixel interleaving — the color information is contained in the first dimension, (3, n, m).
- Line interleaving — the color information is contained in the second dimension, (n, 3, m).
- Planar interleaving — the color information is contained in the third dimension, (n, m, 3).

## Chapter Overview

The following list describes the color image display tasks and associated IDL image color display routines covered in this chapter.

| Tasks | Routine(s)/Object(s) | Description |
|---|---|---|
| "Understanding Colors within IDL Graphic Systems" on page 94 | DEVICE | Learn the differences of working with color in Direct and Object Graphics on platforms supported in IDL. |
| "Loading Pre-defined Color Tables" on page 100. | LOADCT<br>XLOADCT | Load and view one of IDL's pre-defined color tables. |
| "Modifying and Converting Color Tables" on page 103. | XLOADCT<br>XPALETTE<br>TVLCT<br>MODIFYCT<br>HLS<br>HSV<br>COLOR_CONVERT | Use the XLOADCT and XPALETTE utilities to modify a color table and apply it to an image. Save this new color table as one of IDL's pre-defined tables. |

*Table 3-3: Color Image Display Tasks and Related Color Display Routines*

| Tasks | Routine(s)/Object(s) | Description |
|---|---|---|
| "Converting Between Image Types" on page 121. | TVLCT<br>COLOR_QUAN | Change an indexed image with an associated color table to an RGB image, and vice versa. |
| "Highlighting Features with a Color Table" on page 134. | TVLCT<br>IDLgrPalette<br>IDLgrImage | Create an entire color table to highlight features within an image. |
| "Showing Variations in Uniform Areas" on page 145. | H_EQ_CT<br>H_EQ_INT<br>TVLCT<br>IDLgrPalette | Modify a color table with histogram equalization to display minor variations in nearly uniform areas of an image. |
| "Applying Color Annotations to Images" on page 153. | TVLCT<br>IDLgrPalette | Apply specific colors to annotations on indexed or RGB images to highlight certain features within these images. |

*Table 3-3: Color Image Display Tasks and Related Color Display Routines*

**Note**

This chapter uses data files from the IDL `examples/data` directory. Two files, `data.txt` and `index.txt`, contain descriptions of these files, including array sizes.

## Understanding Colors within IDL Graphic Systems

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

### Direct Graphics

#### 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 | DirectColor | 24-bit | DIRECT_COLOR |
| Second | TrueColor | 24-bit (16-bit on Linux) | TRUE_COLOR |
| Third | PseudoColor | 8-bit, then 4-bit | PSEUDO_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 90.

## IDL Color Table

IDL maintains a single current color table for Direct Graphics. Refer to the sections "Loading Pre-defined Color Tables" on page 100 and "Modifying and Converting Color Tables" on page 103. 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 121.

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.

## Object Graphics

In Object Graphics, an underlying understanding of display device visuals and corresponding color interpretation is not required. The color model has been simplified (relative to Direct Graphics) to make the process of color display more straightforward.

### Palettes

The IDLgrPalette object class is used to represent color lookup tables. Any number of palette objects may be instantiated at a given time. The following section will describe how and when these palettes are utilized.

### Color Models

Object Graphics supports two color models for its destination objects (such as an IDLgrWindow): the Indexed Color Model, and the RGB Color Model.

If the Indexed Color Model is used, a color value (or individual image pixel) is expected to be an index into the palette associated with the destination object. To load a particular color table, create a palette object, then set it as a property of the destination object in which the graphics are to be drawn (using the PALETTE keyword in the SetProperty method of the destination object). If a palette is not explicitly provided for a given destination object, a gray scale ramp is loaded by default.

For the Indexed Color Model, a color may also be specified as an RGB triple (although this is much less common). In this case, the nearest match within the destination object's palette will be sought and used to represent that color.

If the RGB Color model is used, a color (or individual image pixel) is expected to be either an index into a palette or an explicit RGB triple. When a color is specified as an index, the index is used to look up a color in the nearest palette (if the graphic includes a palette, that is used first; if the destination has a palette, that will be used next; if no palette is available, a grayscale palette is assumed). If the RGB color model is used, the palette associated with a destination object does not necessarily have a one-to-one mapping to the hardware color lookup table for the device. For instance, the destination object may have a grayscale ramp loaded as a palette, but the hardware color lookup table for the device may be loaded with an even sampling of colors from the RGB color cube. When a user requests that a graphical object be rendered in a particular color, that object will appear in the nearest approximation to that color that the device can supply.

The color model can be explicitly specified using the COLOR_MODEL keyword of the Init method of a destination object. For example, to create a window using the Indexed Color Model:

```
oWindow = OBJ_NEW('IDLgrWindow', COLOR_MODEL = 1)
```

The RGB color model is the default.

## Atomic Graphic Object Colors

In IDL Object Graphics, colors used for drawing atomic graphic objects (such as an IDLgrText object) are typically represented in one of two ways:

- Indexed **-** a color is an index into a palette
- RGB **-** a color is a three-element vector, [red, green, blue].

Color is set using the COLOR keyword of the Init or SetProperty method of the graphic object. For example:

```
oPolyline -> SetProperty, COLOR = 128
```

or

```
oText -> SetProperty, COLOR = [255,0,255]
```

The interpretation of this color depends upon the color model associated with the destination object. See the description of color models (above) for details.

## Image Colors

The IDLgrImage object is used to represent images in Object Graphics. This object stores image data using the byte data type, and can take any of the following forms:

- An array with dimensions [n, m]. Each pixel is interpreted as an index into a palette, or as an explicit gray scale value (if the GREYSCALE keyword is set).
- An array with dimensions [2, n, m] or [n, 2, m] or [n, m, 2]. Each pixel consists of a gray scale value and an associated alpha channel value (alpha is used for transparency effects).
- An array with dimensions [3, n, m] or [n, 3, m] or [n, m, 3]. Each pixel consists of an RGB triple.
- An array with dimensions [4, n, m] or [n, 4, m] or [n, m, 4]. Each pixel consists of an RGB triple and an associated alpha channel value.

The index or RGB triple for each pixel is interpreted according to the color model set for the destination object in which it is to be drawn.

## Loading Pre-defined Color Tables

Although you can define your own color tables, IDL provides 41 pre-defined color tables. You can access these tables through the LOADCT routine. Each color table contained within this routine is specified through an index value ranging from 0 to 40.

**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 95 for more information.

1. View a list of IDL's tables and their related indices by calling LOADCT without an argument:

```
LOADCT
```

The following list is displayed in the Output Log:

```
% Compiled module: LOADCT.
% Compiled module: FILEPATH.
0-B-W LIMEAR            14-STEPS                 28-Hardcandy
1-BLUE/WHITE            15-STERN SPECIAL         29-Nature
2-GRN-RED-BLU-WHT       16-Haze                  30-Ocean
3-RED TEMPERATURE       17-Blue-Pastel-Red       31-
Peppermint
4-BLU/GRN/RED/YEL       18-Pastels               32-Plasma
5-STD GAMMA-II          19-Hue Sat Lightness 1   33-Blue-Red
6-PRISM                 20-Hue Sat Lightness 2   34-Rainbow
7-RED-PURPLE            21-Hue Sat Value 1      35-Blue Waves
8-GREEN/WHITE LINEAR    22-Hue Sat Value 2       36-Volcano
9-GRN/WHT EXPOMENTIAL   23-Purple-Red+Stripes    37-Waves
10-GREEN-PINK           24-Beach                 38-Rainbow18
11-BLUE_RED             25-Mac Style             39-
Rainbow+white
12-16 LEVEL             26-Eos A                 40-
Rainbow+black
13-RAINBOW              27-Eos B
```

When running LOADCT without an argument, it will prompt you to enter the number of one of the above color tables at the IDL command line.

2. Enter in the number 5 at the `Enter table number:` prompt:

   ```
   Enter table number: 5
   ```

   The following text is displayed in the Output Log:

   ```
   % LOADCT: Loading table STD GAMMA-II
   ```

   If you already know the number of the pre-defined color table you want, you can load a color table by providing that number as the first input argument to LOADCT.

3. Load in color table number 13 (RAINBOW):

   ```
   LOADCT, 13
   ```

   The following text is displayed in the Output Log:

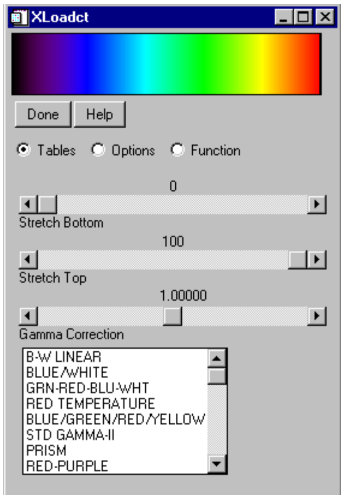
   ```
   % LOADCT: Loading table RAINBOW
   ```

   You can view the current color table with the XLOADCT utility.

4. View color table with XLOADCT utility:

   ```
   XLOADCT
   ```

   The following figure shows the resulting XLOADCT display.



Tables [✓] Options Function

0
Stretch Bottom
100
Stretch Top
1.00000
Gamma Correction

B-W LINEAR
BLUE/WHITE
GRN-RED-BLU-WHT
RED TEMPERATURE
BLUE/GREEN/RED/YELLOW
STD GAMMA-II
PRISM
RED-PURPLE

*Figure 3-2: The XLOADCT Utility*

This utility is designed to individually display each pre-defined color table. When the **Done** button is pressed, the selected color table automatically becomes IDL's current color table. IDL maintains a color table on PseudoColor displays or when the DECOMPOSED keyword to the DEVICE command is set to zero (`DEVICE, DECOMPOSED = 0`) on TrueColor displays. XLOADCT also allows you to make adjustments to the current color table. Among other options, you can stretch the bottom, stretch the top, or apply a gamma correction factor. See the next section, "Modifying and Converting Color Tables" on page 103, for more information.

## Modifying and Converting Color Tables

IDL contains two graphical user interface (GUI) utilities for modifying a color table, XLOADCT and XPALETTE. "Using the XLOADCT Utility" (below) and "Using the XPALETTE Utility" on page 113 describe how to use these utilities to modify color tables. See "Highlighting Features with a Color Table" on page 134 for more information on how to programmatically modify and design a color table. Then the "Using the MODIFYCT Routine" on page 119 section shows how to add the changed color table from XLOADCT and XPALETTE to IDL's list of pre-defined color tables.

The following 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 120.

### Using the XLOADCT Utility

The XLOADCT utility allows you to load one of IDL's 41 pre-defined color tables and change that color table if necessary. The following example shows how to use XLOADCT to load a color table and then change that table to highlight specific features of an image. The indexed image used in this example is a computed tomography (CT) scan of a human thoracic cavity and is contained (without a default color table) within the `ctscan.dat` file in IDL's `examples/data` directory.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Using the XLOADCT Utility" on page 111 or complete the following steps for a detailed description of the process.

1. Determine the path to the `ctscan.dat` binary file:

```
ctscanFile = FILEPATH('ctscan.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Initialize the image size parameter:

```
ctscanSize = [256, 256]
```

3. Import the image from the file:

```
ctscanImage = READ_BINARY(ctscanFile, $
   DATA_DIMS = ctscanSize)
```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword of the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See “Foreground Color” on page 95 for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

   Since the imported image does not have an associated color table, you need to apply a pre-defined color table to display the image.

5. Initialize the display by applying the B-W LINEAR color table (index number 0):

   ```
   LOADCT, 0
   WINDOW, 0, TITLE = 'ctscan.dat', $
      XSIZE = ctscanSize[0], YSIZE = ctscanSize[1]
   ```

6. Display the image using this color table:

   ```
   TV, ctscanImage
   ```

   As the following figure shows, the B-W LINEAR color table does not highlight all of the aspects of this image. The XLOADCT utility can be used to change the color table to highlight more features.

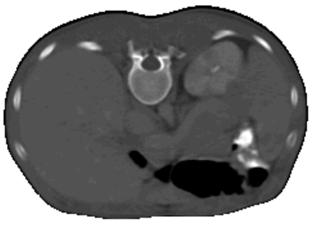


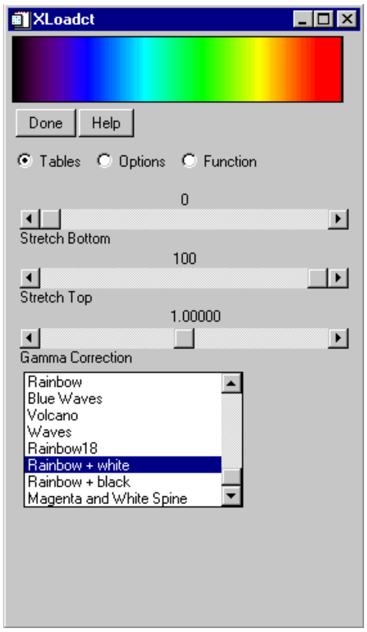
*Figure 3-3: CT Scan Image with Grayscale Color Table*

7. Open the XLOADCT utility:

```
XLOADCT
```

Select Rainbow + white and click **Done** to apply the color table.

The following figure shows the resulting XLOADCT display



[✓ Tables] Options Function

0
Stretch Bottom
100
Stretch Top
1.00000
Gamma Correction

Rainbow
Blue Waves
Volcano
Waves
Rainbow18
[✓ Rainbow + white]
Rainbow + black
Magenta and White Spine

*Figure 3-4: Selecting Rainbow + white Color Table in XLOADCT Utility*

After applying the new color table, you can now see the spine, liver, and kidney within the image, as shown in the following figure. However, the separations between the skin, the organs, and the cartilage and bone within the spine are hard to distinguish.

8. Now re-display the image to show it on the Rainbow + white color table:

```
TV, ctscanImage
```

**Note**

You do not have to perform the previous step on a PseudoColor display. Changes to the current color table automatically show in the current image window within a PseudoColor display.

The following figure shows the CT scan image with the Ranbow+white color table.

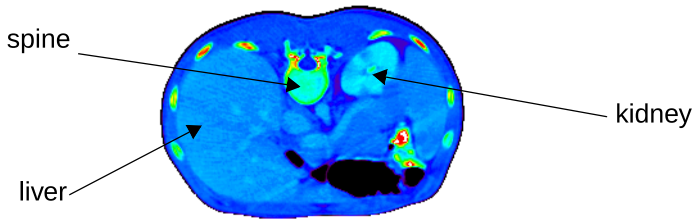


*Figure 3-5: CT Scan Image with the Rainbow + white Color Table*

9. Redisplay the color table with the XLOADCT utility:

```
XLOADCT
```

Comparing the image to the color table, you can see that most image pixels are not within the black to purple range. Therefore the black to purple pixels in the image can be replaced by black. The black range can be stretched to move the purple range to help highlight more features.

The **Stretch Bottom** slider in the XLOADCT utility increases the range of the lowest color index. For example, if black was the color of the lowest index and you increased the bottom stretch by 50 percent, the lower half of the color table would become all black. The remaining part of the color table will contain a scaled version of all the previous color ranges.

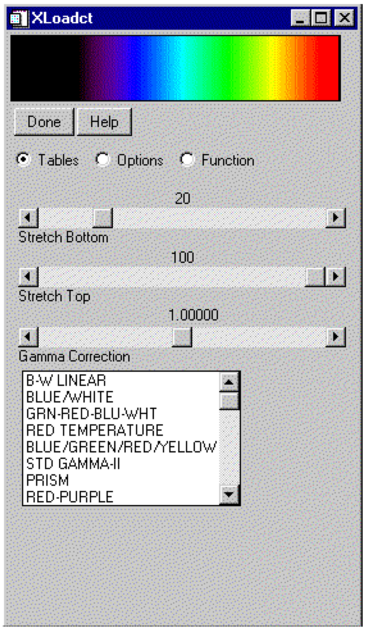
10. Within XLOADCT, stretch the bottom part of the color table by 20 percent by moving the slider as shown in the following figure:

```
TV, ctscanImage
```

**Note**

Remember to click on the **Done** button after changing the **Stretch Bottom** slider, then use TV to re-display the image to include the last changing made in the XLOADCT utility.

In the following figure, you can now see the difference between skin and organs. You can also see where cartilage and bone is located within the spine, but now organs are hard to see. Most of the values in the top (the yellow to red to white ranges) of the color table show just the bones. You can use less of these ranges to show bones by stretching the top of the color table.



[● Tables] Options Function

Stretch Bottom: 20

Stretch Top: 100

Gamma Correction: 1.00000

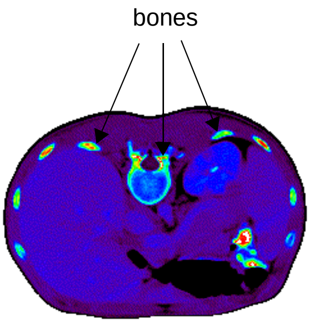


*Figure 3-6: CT Scan Image with Bottom Stretched by 20%*

The **Stretch Top** slider in the XLOADCT utility allows you increase the range of the highest color index. For example, if white was the color of the highest index and you increased the top stretch by 50 percent, the higher half of the color table would become all white. The remaining part of the color table will contain a scaled version of all the previous color ranges.

11. Open XLOADCT:

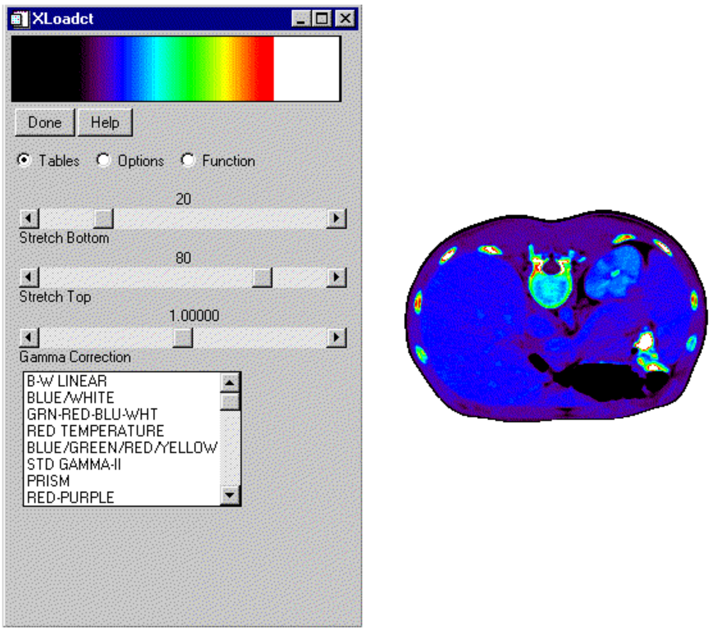
    ```
    XLOADCT
    ```

    Stretch the bottom part of the color table by 20 percent and stretch the top part of the color table by 20 percent (changing it from 100 to 80 percent).

    Click **Done** and redisplay the image:

    ```
    TV, ctscanImage
    ```

    The following figure shows that the organs are more distinctive, but now the liver and kidneys are not clearly distinguished. These features occur in the blue range. You can shift the green range more toward the values of these organs with a gamma correction.



(• Tables) Options Function

Stretch Bottom: 20

Stretch Top: 80

Gamma Correction: 1.00000

*Figure 3-7: CT Scan Image with Bottom and Top Stretched by 20%*

With the **Gamma Correction** slider in the XLOADCT utility you can change the contrast within the color table. A value of 1.0 indicates a linear ramp (no gamma correction). Values other than 1.0 indicate a logarithmic ramp. Higher values of gamma give more contrast. Values less than 1.0 yield lower contrast.

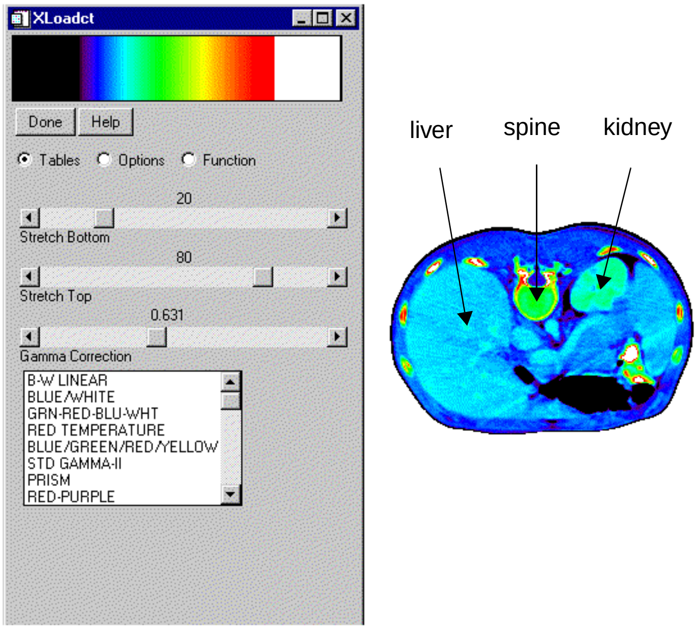
12. Within XLOADCT, stretch the bottom part of the color table by 20 percent, stretch the top part of the color table by 20 percent (change it from 100 percent to 80 percent), and decrease the Gamma Correction factor to 0.631:

```
XLOADCT
```

Redisplay the image:

```
TV, ctscanImage
```

All the features are now highlighted within the image as shown in the following figure:



20
Stretch Bottom
80
Stretch Top
0.631
Gamma Correction

B-W LINEAR
BLUE/WHITE
GRN-RED-BLU-WHT
RED TEMPERATURE
BLUE/GREEN/RED/YELLOW
STD GAMMA-II
PRISM
RED-PURPLE

*Figure 3-8: CT Scan Image with Bottom and Top Stretched by 20% and Gamma Correction at 0.631*

The previous steps showed how to use the **Tables** section of the XLOADCT utility. XLOADCT also contains two other sections: **Options** and **Function**.

The **Options** section allows you to change what the sliders represent and how they are used. When the **Gang** option is selected, the sliders become dependent upon each other. When either the **Stretch Bottom** or **Stretch Top** sliders are moved, the other ones reset to their default values (0 or 100, respectively). With the **Chop** option, you can chop off the top of the color table (the range of the **Stretch Top** is now black instead of the color at the original highest index). With the **Intensity** option, you can change the slider to control the intensity instead of the index location. The **Stretch Bottom** slider will darken the color table and the **Stretch Top** slider will brighten the color table.

The **Function** section allows you to place control points which you can use to change the color table with respect to the other colors in that table. The color table function is shown as a straight line increasing from the lowest index (0) to the highest index (255). The x-axis ranges from 0 to 255 and the y-axis ranges from 0 to 255. Moving a control point in the x-direction has the same effects as the previous sliders. Moving a control point in the y-direction changes the color of that index to another color within the color table. For example, if a control point is red at an index of 128 and the color table is green at an index of 92, when the control point is moved in the y-direction to an index of 92, the color at that x-location will become green. To understand how the **Function** section work, you can use it to highlight just the bones with the CT scan image.

13. Open XLOADCT:

    ```
    XLOADCT
    ```

    Select the Rainbow + white color table.

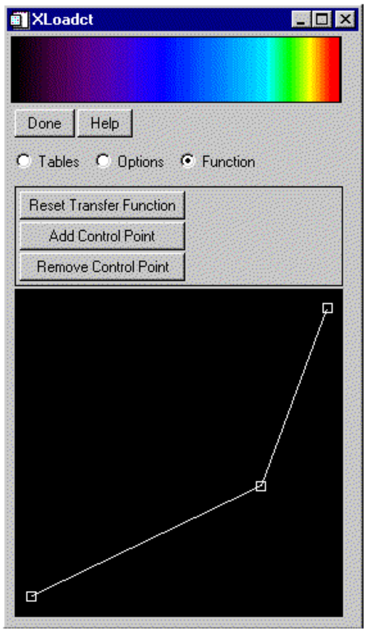
    Switch to the **Function** section by selecting that option.

    Select the **Add Control Point** button, and drag this new center control point one half of the way to the right and one quarter of the way down as shown in the following figure.

    Click **Done** and redisplay the image:

    ```
    TV, ctscanImage
    ```

The bones in the image are now highlighted.



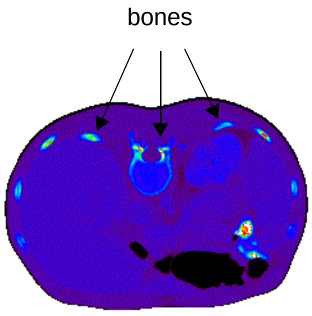


*Figure 3-9: CT Scan Image with Central Control Point Moved One Half to the Right and One Quarter Down*

## Example Code: Using the XLOADCT Utility

Copy and paste the following text into the IDL Editor window. After saving the file as `UsingXLOADCT.pro`, compile and run the program to reproduce the previous example. The BLOCK keyword is set when using XLOADCT to force the example routine to wait until the **Done** button is pressed to continue. If the BLOCK keyword was not set, the example routine would produce all of the displays at once and then end.

```
PRO UsingXLOADCT

; Determine the path to the file.
ctscanFile = FILEPATH('ctscan.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize image size parameter.
ctscanSize = [256, 256]
```

```
; Import the image from the file.
ctscanImage = READ_BINARY(ctscanFile, $
   DATA_DIMS = ctscanSize)

; Initialize display.
DEVICE, DECOMPOSED = 0
LOADCT, 0
WINDOW, 0, TITLE = 'ctscan.dat', $
   XSIZE = ctscanSize[0], YSIZE = ctscanSize[1]

; Display image.
TV, ctscanImage

; Select and display the "Rainbow + white" color
; table
XLOADCT, /BLOCK
TV, ctscanImage

; Increase "Stretch Bottom" by 20%.
XLOADCT, /BLOCK
TV, ctscanImage

; Increase "Stretch Bottom" by 20% and decrease
; "Stretch Top" by 20% (to 80%).
XLOADCT, /BLOCK
TV, ctscanImage

; Increase "Stretch Bottom" by 20%, decrease "Stretch
; Top" by 20% (to 80%), and decrease "Gamma Correction"
; to 0.631.
XLOADCT, /BLOCK
TV, ctscanImage

; Switch to "Function" section, select "Add Control
; Point" and drag this center control point one quarter
; of the way up and one quarter of the way left.
XLOADCT, /BLOCK
TV, ctscanImage

END
```

## Using the XPALETTE Utility

Another utility, XPALETTE, can be used to change a specific color table entry or range of entries. This example uses a single color (orange) to highlight pixels within the spine of the CT scan image. Then, starting with the entry that was changed to orange, a range of entries is selected and replaced with a ramp from orange to white to highlight the bones within this image.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Using the XPALETTE Utility" on page 117 or complete the following steps for a detailed description of the process.

1. Determine the path to the `ctscan.dat` binary file:

   ```
   ctscanFile = FILEPATH('ctscan.dat', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Initialize the image size parameter:

   ```
   ctscanSize = [256, 256]
   ```

3. Import the image from the file:

   ```
   ctscanImage = READ_BINARY(ctscanFile, $
      DATA_DIMS = ctscanSize)
   ```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "Foreground Color" on page 95 for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

5. Display the image from the `ctscan.dat` file with the B-W LINEAR color table:

   ```
   LOADCT, 0
   WINDOW, 0, TITLE = 'ctscan.dat', $
      XSIZE = ctscanSize[0], YSIZE = ctscanSize[1]
   TV, ctscanImage
   ```

As shown in the following figure, the B-W LINEAR color table does not distinguish all of the aspects of this image. The XPALETTE utility can be used to change the color table.

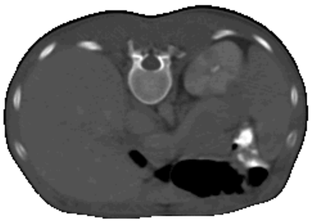


*Figure 3-10: CT Scan Image with Grayscale Color Table*

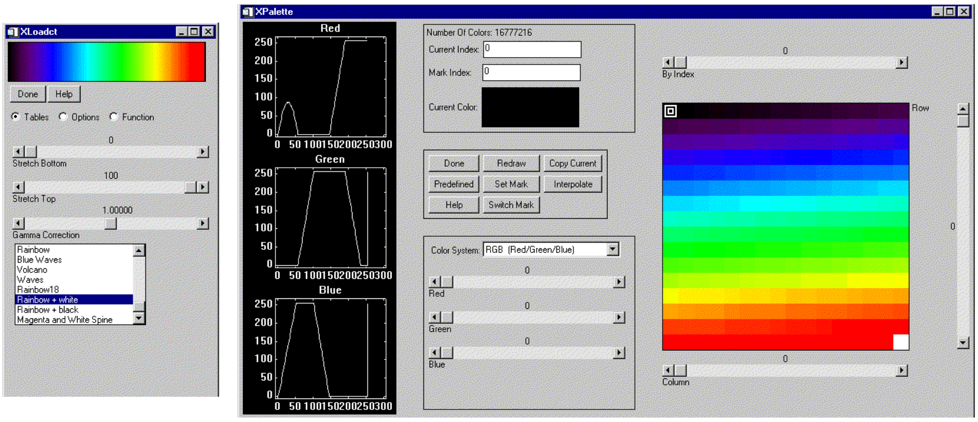
6. Open the XPALETTE utility:

```
XPALETTE
```

Select the **Predefined** button in the XPALETTE utility to change the color table to Rainbow + white.

Click on the **Done** button after you select the Rainbow + white color table in XLOADCT and then click on the **Done** button in XPALETTE.

The following figure shows the resulting XPALETTE and XLOADCT displays.



Stretch Bottom: 0
Stretch Top: 100
Gamma Correction: 1.00000
Tables [✓ Rainbow + white]

Number Of Colors: 16777216
Current Index: 0
Mark Index: 0
Current Color:
Color System: RGB (Red/Green/Blue)
Red: 0
Green: 0
Blue: 0

*Figure 3-11: Selecting Rainbow + white Color Table in XPALETTE Utility*

7. Now redisplay the image to show it with the Rainbow + white color table:

```
TV, ctscanImage
```

Your display should be similar to the following figure.

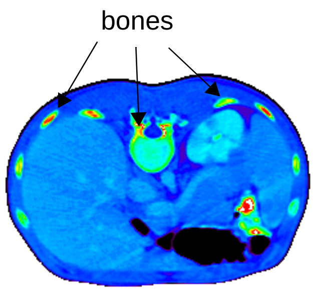


*Figure 3-12: CT Scan Image with the Rainbow + white Color Table*

You can use XPALETTE to change a single color within the current color table. For example, you can change the color at index number 115 to orange.

8. Open XPALETTE and click on the 115th index (in column 3 and row 7):

```
XPALETTE
```

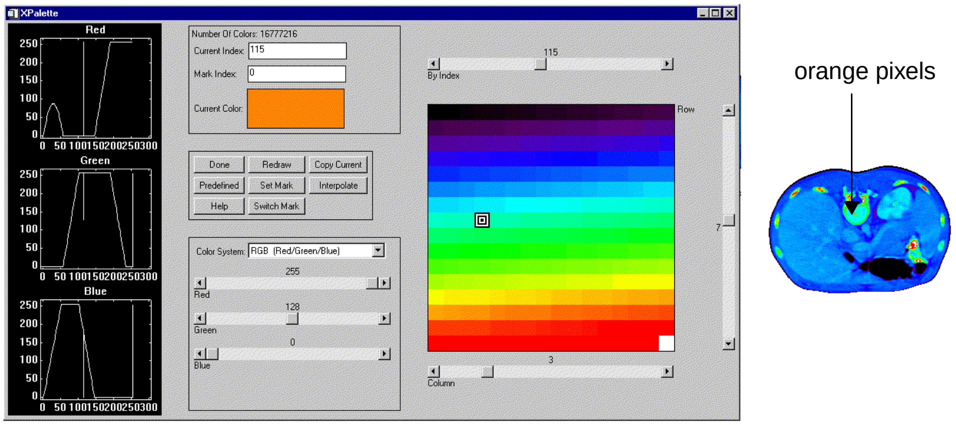
Change its color to orange by moving the RGB (red, green, and blue) sliders (Orange is made up of 255 red, 128 green, and 0 blue)

Click on the **Done** button after changing the **Red**, **Green**, and **Blue** sliders.

Use TV to redisplay the image to include the last changes made in the XPALETTE utility:

```
TV, ctscanImage
```

The orange values now highlight some areas of the spine, kidney, and bones as shown in the following figure.



Number Of Colors: 16777216
Current Index: 115
Mark Index: 0
Current Color:
Color System: RGB (Red/Green/Blue)
Red: 255
Green: 128
Blue: 0
By Index: 115
Row: 7
Column: 3

*Figure 3-13: CT Scan Image with Orange Added to the Color Table*

You can highlight the bones even further by interpolating a new range in between the orange and white indices.

9. Open XPALETTE:

   Click on the 115th index and select the **Set Mark** button.

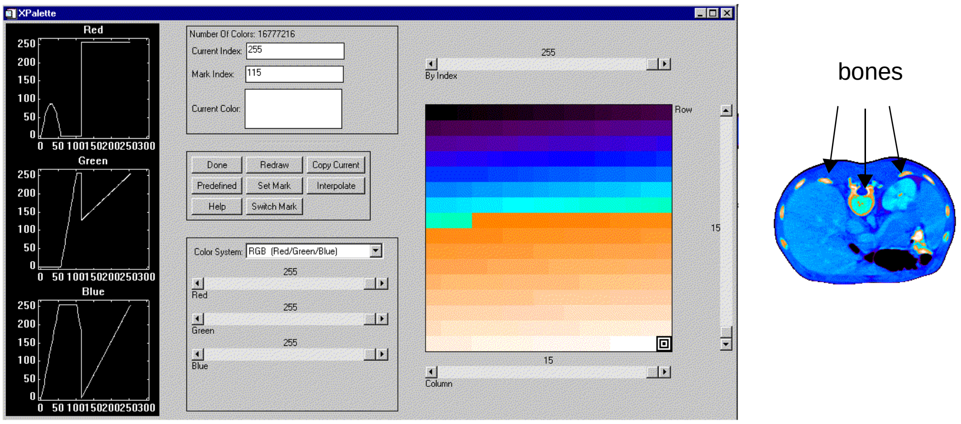
   Click on the highest index (which is usually 255 but it could be less) and then select the **Interpolate** button.

   To see the result of this interpolation within XPALETTE, click on the **Redraw** button.

   Click **Done** and redisplay the image:

   ```
   TV, ctscanImage
   ```

   The following figure displays the image using the modified color table.


Number Of Colors: 16777216

Current Index: 255

Mark Index: 115

Color System: RGB (Red/Green/Blue)

*Figure 3-14: CT Scan Image with Orange to White Range Added*

## Example Code: Using the XPALETTE Utility

Copy and paste the following text into the IDL Editor window. After saving the file as `UsingXPALETTE.pro`, compile and run the program to reproduce the previous example. The BLOCK keyword is set when using XPALETTE to force the example routine to wait until the **Done** button is pressed to continue. If the BLOCK keyword was not set, the example routine would produce all of the displays at once and then end.

```
PRO UsingXPALETTE

; Determine the path to the file.
ctscanFile = FILEPATH('ctscan.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize image size parameter.
ctscanSize = [256, 256]

; Import the image from the file.
ctscanImage = READ_BINARY(ctscanFile, $
   DATA_DIMS = ctscanSize)

; Initialize display.
DEVICE, DECOMPOSED = 0
LOADCT, 0
WINDOW, 0, TITLE = 'ctscan.dat', $
   XSIZE = ctscanSize[0], YSIZE = ctscanSize[1]

; Display image.
TV, ctscanImage

; Click on the "Predefined" button and select the
; "Rainbow + white" color table.
XPALETTE, /BLOCK
TV, ctscanImage

; Click on the 115th index, which is in column 3 and row
; 7, and then change its color to orange with the RGB
; (red, green, and blue) sliders.  Orange is made up of
; 255 red, 128 green, and 0 blue.
XPALETTE, /BLOCK
TV, ctscanImage

; Click on the 115th index, click on the "Set Mark"
; button, click on the 255th index, and click on the
; "Interpolate" button.  The colors within the 115 to
; 255 range are now changed to go between orange and
; white.  To see this change within the XPALETTE
; utility, click on the "Redraw" button.
XPALETTE, /BLOCK
TV, ctscanImage

; Obtain the red, green, and blue vectors of this
; current color table.
TVLCT, red, green, blue, /GET

; Add this modified color table to IDL's list of
; pre-defined color tables and display results.
```

```
MODIFYCT, 41, 'Orange to White Bones', $
   red, green, blue
XLOADCT, /BLOCK
TV, ctscanImage

END
```

## Using the MODIFYCT Routine

The previously derived color table created in "Using the XPALETTE Utility" on page 113 can be added to IDL's list of pre-defined color tables with the TVLCT and MODIFYCT routines. For code that you can copy and paste into a text editor (for example the IDL Editor), see "Example Code: Using the XPALETTE Utility" on page 117.

By default, TVLCT allows you to load in red, green, and blue vectors (either derived by you or imported from an image file) to load a different current color table. TVLCT also has a GET keyword. When the GET keyword is set, TVLCT returns the red, green, and blue vectors of the current color table back to you. Using this you can obtain the red, green, and blue vectors of the previously derived color table.

1. Obtain the red, green, and blue vectors of the current color table after performing the steps in "Using the XPALETTE Utility" on page 113:

   ```
   TVLCT, red, green, blue, /GET
   ```

   The MODIFYCT routine uses these vectors as arguments. Now you can use MODIFYCT to add this new color table to IDL's list of pre-defined color tables.

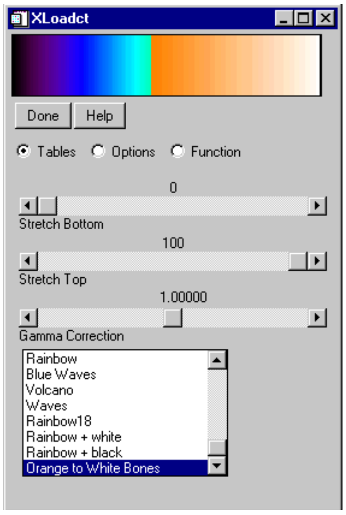
2. Add this modified color table to IDL's list of pre-defined color tables and display results:

   ```
   MODIFYCT, 41, 'Orange to White Bones', $
      red, green, blue
   ```

3. Display the results with XLOADCT:

   ```
   XLOADCT
   ```

The modified color table has been added to IDL's list of pre-defined color tables as shown in the following figure.



0
Stretch Bottom
100
Stretch Top
1.00000
Gamma Correction

Rainbow
Blue Waves
Volcano
Waves
Rainbow18
Rainbow + white
Rainbow + black
Orange to White Bones

*Figure 3-15: XLOADCT Showing Results of MODIFYCT*

The MODIFYCT routine can also be used to save changes to one of the existing pre-defined color tables. See MODIFYCT in the *IDL Reference Guide* for more information.

## 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_CONVERT` in the *IDL Reference Guide* for more information.

## 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 95 for more information on grayscale, indexed, and RGB images.

### Converting Indexed Images to RGB Images

The `convec.dat` file is a binary file that contains an indexed image (a two-dimensional image and its associated color table) of the convection of the earth's mantle. This file does not contain a related color table. The following example applies a color table to this image and then converts the image and table to an RGB image (which contains its own color information).

For code that you can copy and paste into an IDL Editor window, see "Example Code: Converting Indexed Images to RGB Images" on page 123 or complete the following steps for a detailed description of the process.

1. Determine the path to the `convec.dat` binary file:

   ```
   convecFile = FILEPATH('convec.dat', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Initialize the image size parameter:

   ```
   convecSize = [248, 248]
   ```

3. Import the image from the file:

   ```
   convecImage = READ_BINARY(convecFile, $
      DATA_DIMS = convecSize)
   ```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "Foreground Color" on page 95 for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

   The EOS B color table is applied to highlight the features of this image.

5. Load the EOS B color table (index number 27) to highlight the image's features and initialize the display:

   ```
   LOADCT, 27
   WINDOW, 0, TITLE = 'convec.dat', $
      XSIZE = convecSize[0], YSIZE = convecSize[1]
   ```

6. Now display the image with this color table:

   ```
   TV, convecImage
   ```

   The following figure shows the original image with an applied color table.

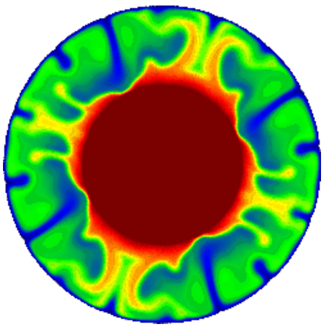


*Figure 3-16: Example of an Indexed Image With Associated Color Table*

A color table is formed from three vectors (the red vector, the green vector, and the blue vector). The same element of each vector together form an RGB triplet to create a color. For example, the *i-th* element of the red vector may be 255, the *ith* element of the green vector may be 255, and the *ith* element of the blue vector maybe 0. The RGB triplet of the *ith* element would then be (255, 255, 0), which is the color yellow. Since a color table contains 256 indices, its three vectors have 256 elements each. You can access these vectors with the TVLCT routine using the GET keyword.

**Note**

On some PseudoColor displays, fewer than 256 entries will be available.

7. Access the values of the color table by setting the GET keyword to the TVLCT routine.

   ```
   TVLCT, red, green, blue, /GET
   ```

   This color table (color information) can be stored within the image by converting it to an RGB image. For this example, the RGB image will be pixel interleaved in order to be exported to a JPEG file.

**Tip**

If the original indexed image contains values of a data type other than byte, you should byte-scale the image (with the BYTSCL routine) before using the following method.

---

   Before converting the indexed image into an RGB image, the resulting three-dimensional array must be initialized.

8. Initialize the data type and the dimensions of the resulting RGB image:

   ```
   imageRGB = BYTARR(3, convecSize[0], convecSize[1])
   ```

   Each channel of the resulting RGB image can be derived from the red, green, and blue vectors of the color table and the original indexed image.

9. Use the red, green, and blue vectors of the color table and the original indexed image to form a single image composed of these channels:

   ```
   imageRGB[0, *, *] = red[convecImage]
   imageRGB[1, *, *] = green[convecImage]
   imageRGB[2, *, *] = blue[convecImage]
   ```

10. Export the resulting RGB image to a JPEG file:

    ```
    WRITE_JPEG, 'convecImage.jpg', imageRGB, TRUE = 1, $
       QUALITY = 100.
    ```

    The TRUE keyword is set to 1 because the resulting RGB image is pixel interleaved. See WRITE_JPEG for more information.

## Example Code: Converting Indexed Images to RGB Images

Copy and paste the following text into the IDL Editor window. After saving the file as `IndexedToRGB.pro`, compile and run the program to reproduce the previous example.

```
PRO IndexedToRGB

; Determine the path to the file.
convecFile = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

```
; Initialize the image size parameter.
convecSize = [248, 248]

; Import the image from the file.
convecImage = READ_BINARY(convecFile, $
   DATA_DIMS = convecSize)

; Initialize display.
DEVICE, DECOMPOSED = 0
LOADCT, 27
WINDOW, 0, TITLE = 'convec.dat', $
   XSIZE = convecSize[0], YSIZE = convecSize[1]

; Display image.
TV, convecImage

; Obtain the red, green, and blue vectors that form the
; current color table.
TVLCT, red, green, blue, /GET

; Initialize the resulting RGB image.
imageRGB = BYTARR(3, convecSize[0], convecSize[1])

; Derive each color image from the vectors of the
; current color table.
imageRGB[0, *, *] = red[convecImage]
imageRGB[1, *, *] = green[convecImage]
imageRGB[2, *, *] = blue[convecImage]

; Write the resulting RGB image out to a JPEG file.
WRITE_JPEG, 'convec.jpg', imageRGB, TRUE = 1, $
   QUALITY = 100.

END
```

## Converting RGB Images to Grayscale Images

The following example extracts the three channels of an RGB image contained in the `glowing_gas.jpg` file, which is in the `examples/data` directory. This file is provided by the Hubble Heritage Team, which is made of AURA, STScI, and NASA.

The channels are extracted as grayscale (intensity) images. These images are converted to floating-point data and then added together to form a single image, which is a grayscale version of the original RGB image.

For code that you can copy and paste into an Editor window, see "Example Code: Converting RGB Images into Grayscale Images" on page 128 or complete the following steps for a detailed description of the process.

1. Determine the path to the file:

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

2. Query the file to determine the image parameters:

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

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

```
imageSize = imageInfo.dimensions
```

4. Import the image from the file:

```
READ_JPEG, file, image
```

5. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to one before your first RGB image is displayed within an IDL session or program. See "Foreground Color" on page 95 for more information:

```
DEVICE, DECOMPOSED = 1
```

6. Create a window and display the image:

```
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Glowing Gas RGB Image'
TV, image, TRUE = 1
```

The following figure shows the original RGB image.

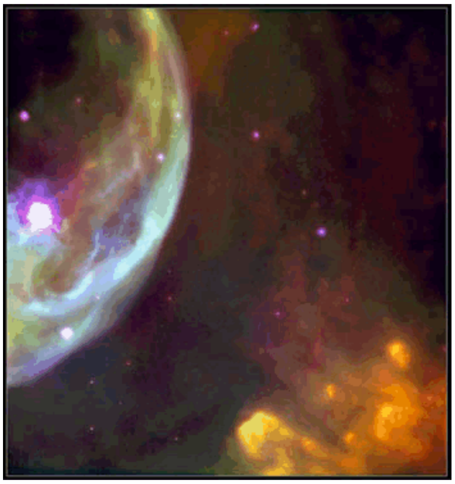


*Figure 3-17: The Glowing Gas RGB Image*

7. Extract the channels (as images) from the RGB image:

```
redChannel = REFORM(image[0, *, *])
greenChannel = REFORM(image[1, *, *])
blueChannel = REFORM(image[2, *, *])
```

8. Initialize the grayscale display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

9. Create another window and display each channel of the RGB image:

```
WINDOW, 1, XSIZE = 3*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Red (left), Green (middle), ' + $
   'and Blue (right) Channels of the RGB Image'
TV, redChannel, 0
TV, greenChannel, 1
TV, blueChannel, 2
```

The following figure shows the RGB channels. The red channel is on the left, the green channel is in the middle, and the blue channel is on the right.

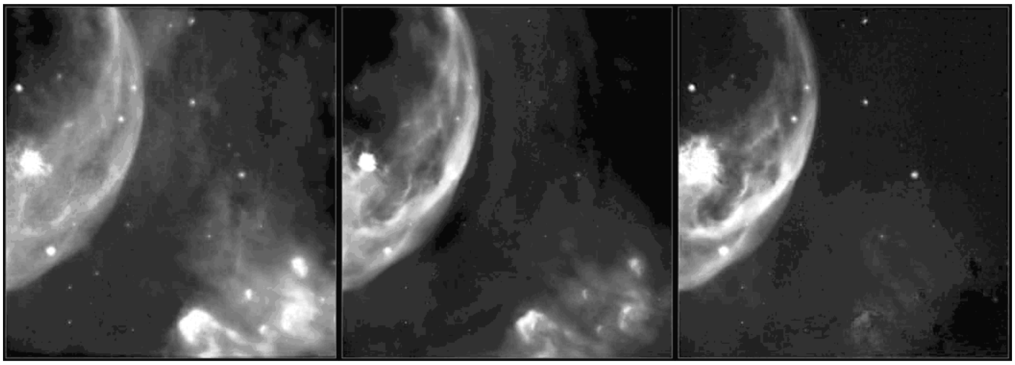


*Figure 3-18: The Channels of the Glowing Gas RGB Image*

10. Convert the channels into a single grayscale image.

```
grayscaleImage = BYTE(0.299*FLOAT(redChannel) + $
   0.587*FLOAT(redChannel) + 0.114*FLOAT(blueChannel))
```

The pixel values of the channels are converted from byte values to floating-point values because byte values cannot exceed 255. The adjustment factors (0.299, 0.587, and 0.114) are used to enhance visual perception and to scale the results to a range from 0 to 255. The BYTE function is used to restore the pixel values back to their original data type.

11. Create another window and display the grayscale image:

```
WINDOW, 2, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Resulting Grayscale Image' + $
TV, grayscaleImage
```

The following figure shows the result of creating a grayscale image from the individual channels of an RGB image.

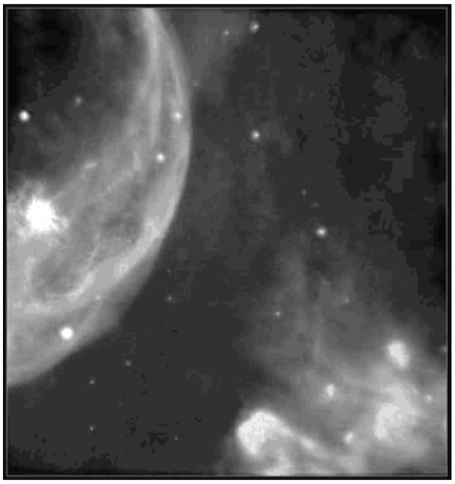


*Figure 3-19: Resulting Grayscale Image*

## Example Code: Converting RGB Images into Grayscale Images

Copy and paste the following text into the IDL Editor window. After saving the file as `RGBToGrayscale.pro`, compile and run the program to reproduce the previous example.

```
PRO RGBToGrayscale

; Determine the path to the file.
file = FILEPATH('glowing_gas.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Query the file to determine image parameters.
queryStatus = QUERY_JPEG(file, imageInfo)

; Set the image size parameter from the query
; information.
imageSize = imageInfo.dimensions

; Import the image from the file.
READ_JPEG, file, image

; Initialize the RGB display.
```

```
DEVICE, DECOMPOSED = 1

; Create a window and display the image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Glowing Gas RGB Image'
TV, image, TRUE = 1

; Extract the channels (as images) from the RGB image.
redChannel = REFORM(image[0, *, *])
greenChannel = REFORM(image[1, *, *])
blueChannel = REFORM(image[2, *, *])

; Initialize the grayscale display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create another window and display each channel of the
; RGB image.
WINDOW, 1, XSIZE = 3*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Red (left), Green (middle), ' + $
   'and Blue (right) Channels of the RGB Image'
TV, redChannel, 0
TV, greenChannel, 1
TV, blueChannel, 2

; Convert the channels into a grayscale image.
grayscaleImage = BYTE(0.299*FLOAT(redChannel) + $
   0.587*FLOAT(redChannel) + 0.114*FLOAT(blueChannel))

; Create another window and display the resulting
; grayscale image.
WINDOW, 2, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Resulting Grayscale Image'
TV, grayscaleImage

END
```

## Converting RGB Images to Indexed Images

Although it is a relatively simple process to convert an RGB image to a grayscale image, the process needed to convert an RGB image to an indexed image is more complex. This process is more complex because the millions of possible colors provided by an RGB image must be decomposed into the 256 colors used by an indexed image. IDL's COLOR_QUAN function may be used to perform this process.

The following example shows how to use the COLOR_QUAN function to convert an RGB image to an indexed image. The `elev_t.jpg` file contains a pixel interleaved RGB image, which has its own color information. This example converts the image to an indexed image with an associated color table.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Converting RGB Images to Indexed Images" on page 132 or complete the following steps for a detailed description of the process.

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

   ```
   elev_tFile = FILEPATH('elev_t.jpg', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Import the image from the `elev_t.jpg` file into IDL:

   ```
   READ_JPEG, elev_tFile, elev_tImage
   ```

3. Determine the size of the imported image:

   ```
   elev_tSize = SIZE(elev_tImage, /DIMENSIONS)
   ```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to one before your first RGB image is displayed within an IDL session or program. See "Foreground Color" on page 95 for more information:

   ```
   DEVICE, DECOMPOSED = 1
   ```

5. Initialize the display:

   ```
   WINDOW, 0, TITLE = 'elev_t.jpg', $
      XSIZE = elev_tSize[1], YSIZE = elev_tSize[2]
   ```

6. Display the imported image:

   ```
   TV, elev_tImage, TRUE = 1
   ```

The following figure shows the original RGB image.

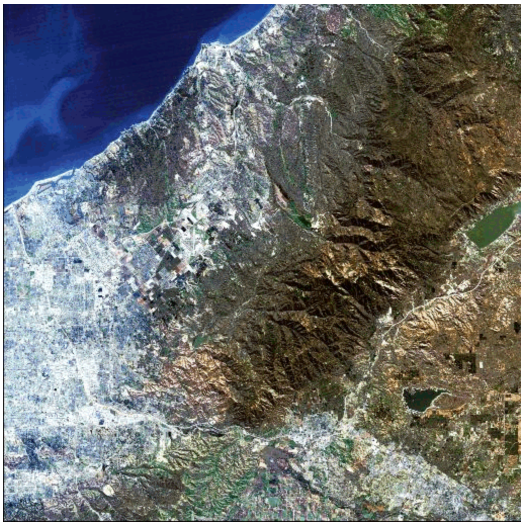


*Figure 3-20: Example of an RGB Image*

**Note**

If you are running IDL on a PseudoColor display, the RGB image will not be displayed correctly. A PseudoColor display only allows the display of indexed images. You can change the RGB image to an indexed image with the COLOR_QUAN routine. An example of this method is shown in this section.

The RGB image is converted to an indexed image with the COLOR_QUAN routine, but the DECOMPOSED keyword to the DEVICE command must be set to zero (for TrueColor displays) before using COLOR_QUAN because it is a color table related routine. See COLOR_QUAN in the *IDL Reference Guide* for more information.

**Note**

COLOR_QUAN may result in some loss of color information since it quantizes the image to a fixed number of colors (stored in the color table).

7. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "Foreground Color" on page 95 for more information.

```
DEVICE, DECOMPOSED = 0
```

8. Convert the RGB image to an indexed image with an associated color table:

```
imageIndexed = COLOR_QUAN(elev_tImage, 1, red, green, $
   blue)
```

9. Export the resulting indexed image and its associated color table to a PNG file:

```
WRITE_PNG, 'elev_t.png', imageIndexed, red, green, blue
```

## Example Code: Converting RGB Images to Indexed Images

Copy and paste the following text into the IDL Editor window. After saving the file as `RGBToIndexed.pro`, compile and run the program to reproduce the previous example.

```
PRO RGBToIndexed

; Determine path to the "elev_t.jpg" file.
elev_tFile = FILEPATH('elev_t.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Import image from file into IDL.
READ_JPEG, elev_tFile, elev_tImage

; Determine the size of the imported image.
elev_tSize = SIZE(elev_tImage, /DIMENSIONS)

; Initialize display.
DEVICE, DECOMPOSED = 1
WINDOW, 0, TITLE = 'elev_t.jpg', $
   XSIZE = elev_tSize[1], YSIZE = elev_tSize[2]

; Display image.
TV, elev_tImage, TRUE = 1

; Convert RGB image to indexed image with associated
; color table.
DEVICE, DECOMPOSED = 0
```

```
imageIndexed = COLOR_QUAN(elev_tImage, 1, red, green, $
   blue)

; Write resulting image and its color table to a PNG
; file.
WRITE_PNG, 'elev_t.png', imageIndexed, red, green, blue

END
```

## 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

### Highlighting Features with Color in Direct Graphics

The data in the `mineral.png` file in the `examples/data` directory comes with its own color table. The following example will apply this related color table, then a pre-defined color table, and finally derive a new color table to highlight specific features.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Highlighting Features with Color in Direct Graphics" on page 137 or complete the following steps for a detailed description of the process.

1. Determine the path to the `mineral.png` file:

```
mineralFile = FILEPATH(‘mineral.png', $
   SUBDIRECTORY = [‘examples', ‘data'])
```

2. Import the image from the `mineral.png` file into IDL:

```
mineralImage = READ_PNG(mineralFile, red, green, blue)
```

   The image's associated color table is contained within the resulting red, green, and blue vectors.

3. Determine the size of the imported image:

```
mineralSize = SIZE(mineralImage, /DIMENSIONS)
```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "Foreground Color" on page 95 for more information.

```
DEVICE, DECOMPOSED = 0
```

5. Load the image's associated color table with the TVLCT routine:

```
TVLCT, red, green, blue
```

6. Initialize the display:

```
WINDOW, 0, XSIZE = mineralSize[0], YSIZE = mineralSize[1], $
   TITLE = 'mineral.png'
```

7. Display the imported image:

```
TV, mineralImage
```

   This scanning electron microscope image shows mineral deposits in a sample of polished granite and gneiss. The associated color table is a reverse grayscale.

   The following figure shows that the associated color table highlights the gneiss very well, but the other features are not very clear. The other features can be defined with IDL's pre-defined color table, RAINBOW.

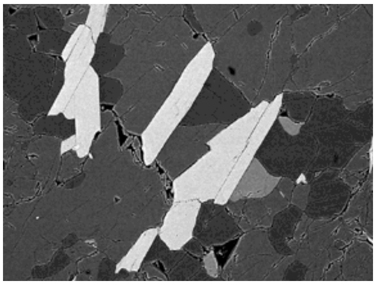


*Figure 3-21: Mineral Image and Default Color Table (Direct Graphics)*

8. Load the RAINBOW color table and redisplay the image in another window:

```
LOADCT, 13
WINDOW, 1, XSIZE = mineralSize[0], YSIZE = mineralSize[1], $
   TITLE = ‘RAINBOW Color'
TV, mineralImage
```

The following figure shows that the yellow, cyan, and red sections are now apparent, but the cracks are no longer visible. Details within the yellow areas and the green background are also difficult to distinguish. These features can be highlighted by designing your own color table.

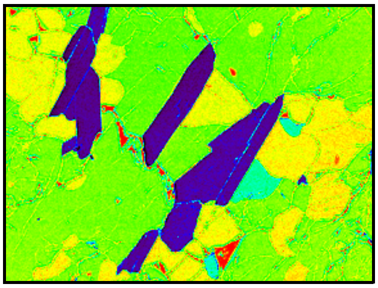


*Figure 3-22: Mineral Image and RAINBOW Color Table (Direct Graphics)*

The features within the image are at specific ranges in between 0 and 255. Instead of a progressive color table, specific colors can be defined to be constant over these ranges. Any contrasting colors can be used, but it is easiest to derive the additive and subtractive primary colors used in the previous section.

9. Define the colors for a new color table:

```
colorLevel = [[0, 0, 0], $ ; black
   [255, 0, 0], $ ; red
   [255, 255, 0], $ ; yellow
   [0, 255, 0], $ ; green
   [0, 255, 255], $ ; cyan
   [0, 0, 255], $ ; blue
   [255, 0, 255], $ ; magenta
   [255, 255, 255]] ; white
```

10. Create a new color table that contains eight levels, including the highest end boundary by first deriving levels for each color in the new color table:

```
numberOfLevels = CEIL(!D.TABLE_SIZE/8.)
level = INDGEN(!D.TABLE_SIZE)/numberOfLevels
```

11. Place each color level into its appropriate range.

```
newRed = colorLevel[0, level]
newGreen = colorLevel[1, level]
newBlue = colorLevel[2, level]
```

12. Include the last color in the last level:

```
newRed[!D.TABLE_SIZE - 1] = 255
newGreen[!D.TABLE_SIZE - 1] = 255
newBlue[!D.TABLE_SIZE - 1] = 255
```

13. Make the new color table current:

```
TVLCT, newRed, newGreen, newBlue
```

14. Display the image with this new color table in another window:

```
WINDOW, 2, XSIZE = mineralSize[0], $
   YSIZE = mineralSize[1], TITLE = 'Cube Corner Colors'
TV, mineralImage
```

The following figure shows that each feature is now highlighted including the cracks. The color table also highlights at least three different types of cracks.

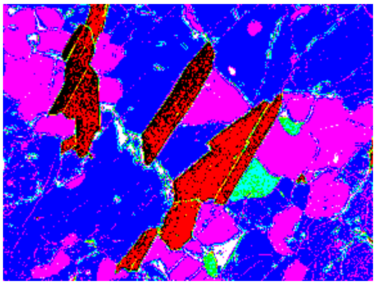


*Figure 3-23: Mineral Image and Derived Color Table (Direct Graphics)*

## Example Code: Highlighting Features with Color in Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `HighlightFeatures_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO HighlightFeatures_Direct

; Determine path to "mineral.png" file.
mineralFile = FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])

; Import image from file into IDL.
mineralImage = READ_PNG(mineralFile, $
   red, green, blue)
```

```
; Determine size of imported image.
mineralSize = SIZE(mineralImage, /DIMENSIONS)

; Apply imported color vectors to current color table.
DEVICE, DECOMPOSED = 0
TVLCT, red, green, blue

; Initialize display.
WINDOW, 0, XSIZE = mineralSize[0], YSIZE = mineralSize[1], $
   TITLE = 'mineral.png'

; Display image.
TV, mineralImage

; Load "RAINBOW" color table and display image in
; another window.
LOADCT, 13
WINDOW, 1, XSIZE = mineralSize[0], YSIZE = mineralSize[1], $
   TITLE = 'RAINBOW Color'
TV, mineralImage

; Define colors for a new color table.
colorLevel = [[0, 0, 0], $ ; black
   [255, 0, 0], $ ; red
   [255, 255, 0], $ ; yellow
   [0, 255, 0], $ ; green
   [0, 255, 255], $ ; cyan
   [0, 0, 255], $ ; blue
   [255, 0, 255], $ ; magenta
   [255, 255, 255]] ; white

; Derive levels for each color in the new color table.
; NOTE: some displays may have less than 256 colors.
numberOfLevels = CEIL(!D.TABLE_SIZE/8.)
level = INDGEN(!D.TABLE_SIZE)/numberOfLevels

; Place each color level into its appropriate range.
newRed = colorLevel[0, level]
newGreen = colorLevel[1, level]
newBlue = colorLevel[2, level]

; Include the last color in the last level.
newRed[!D.TABLE_SIZE - 1] = 255
newGreen[!D.TABLE_SIZE - 1] = 255
newBlue[!D.TABLE_SIZE - 1] = 255

; Make the new color table current.
TVLCT, newRed, newGreen, newBlue
```

```
; Display image in another window.
WINDOW, 2, XSIZE = mineralSize[0], $
   YSIZE = mineralSize[1], TITLE = 'Cube Corner Colors'
TV, mineralImage

END
```

## Highlighting Features with Color in Object Graphics

The previous example could have been done with Object Graphics. The color table is derived in the same matter. This example shows how to create a color table to highlight image features using Object Graphics.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Highlighting Features with Color in Object Graphics" on page 142 or complete the following steps for a detailed description of the process.

1. Determine the path to the `mineral.png` file:

   ```
   mineralFile = FILEPATH('mineral.png', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Import the image and its associated color table into IDL:

   ```
   mineralImage = READ_PNG(mineralFile, red, green, blue)
   ```

3. Determine the size of the imported image:

   ```
   mineralSize = SIZE(mineralImage, /DIMENSIONS)
   ```

4. Initialize objects necessary for a graphics display:

   ```
   oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
      DIMENSIONS = [mineralSize[0], mineralSize[1]], $
      TITLE = 'mineral.png')
   oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT = [0., 0., $
      mineralSize[0], mineralSize[1]])
   oModel = OBJ_NEW('IDLgrModel')
   ```

5. Initialize a palette object containing the image's associated color table and apply the palette to the image objects:

   ```
   oPalette = OBJ_NEW('IDLgrPalette', red, green, blue)
   oImage = OBJ_NEW('IDLgrImage', mineralImage, $
      PALETTE = oPalette)
   ```

   The objects are then added to the view, which is displayed in the window.

6. Add the image to the model, then add the model to the view:

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

Draw the view in the window:

```
oWindow -> Draw, oView
```

This scanning electron microscope image shows mineral deposits in a sample of polished granite and gneiss. The associated color table is a reverse grayscale.

The following figure shows that the associated color table highlights the gneiss very well, but the other features are not very clear. The other features can be defined with IDL's pre-defined color table, RAINBOW.

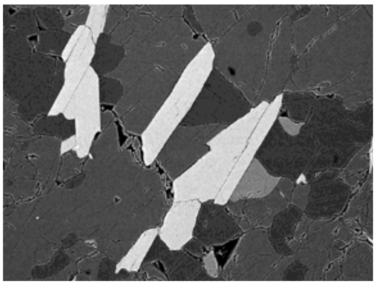


*Figure 3-24: Mineral Image and Default Color Table (Object Graphics)*

The palette can easily be modified to show the RAINBOW pre-defined color table in another instance of the window object.

7. Update palette with RAINBOW color table and then display the image with this color table in another instance window of the window object:

```
oPalette -> LoadCT, 13
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [mineralSize[0], mineralSize[1]], $
   TITLE = 'RAINBOW Color')
oWindow -> Draw, oView
```

The following figure shows that the yellow, cyan, and red sections are now apparent, but the cracks are no longer visible. Details within the yellow areas and the green background are also difficult to distinguish. These features can be highlighted by designing your own color table.

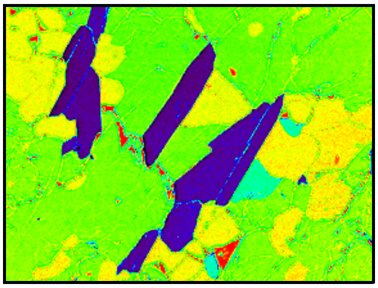


*Figure 3-25: Mineral Image and RAINBOW Color Table (Object Graphics)*

The features within the image are at specific ranges in between 0 and 255. Instead of a progressive color table, specific colors can be defined to be constant over these ranges. Any contrasting colors can be used, but the easiest to derive are the additive and subtractive primary colors used in the previous section.

8. Define colors for a new color table:

```
colorLevel = [[0, 0, 0], $ ; black
   [255, 0, 0], $ ; red
   [255, 255, 0], $ ; yellow
   [0, 255, 0], $ ; green
   [0, 255, 255], $ ; cyan
   [0, 0, 255], $ ; blue
   [255, 0, 255], $ ; magenta
   [255, 255, 255]] ; white
```

9. Create a new color table that contains eight levels, including the highest end boundary by first deriving levels for each color in the new color table:

```
numberOfLevels = CEIL(!D.TABLE_SIZE/8.)
level = INDGEN(!D.TABLE_SIZE)/numberOfLevels
```

10. Place each color level into its appropriate range.

```
newRed = colorLevel[0, level]
newGreen = colorLevel[1, level]
newBlue = colorLevel[2, level]
```

11. Include the last color in the last level:

```
newRed[!D.TABLE_SIZE - 1] = 255
newGreen[!D.TABLE_SIZE - 1] = 255
newBlue[!D.TABLE_SIZE - 1] = 255
```

Apply the new color table to the palette object:

12. Display the image with this color table in another window:

```
oPalette -> SetProperty, RED_VALUES = newRed, $
   GREEN_VALUES = newGreen, BLUE_VALUES = newBlue
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [mineralSize[0], mineralSize[1]], $
   TITLE = 'Cube Corner Colors')
oWindow -> Draw, oView
```

The following figure shows that each image feature is readily distinguishable.

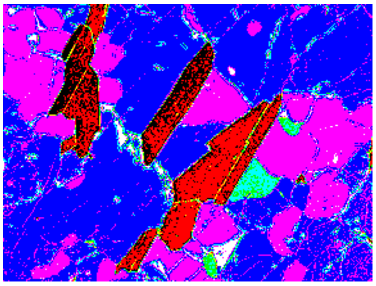


*Figure 3-26: Mineral Image and Derived Color Table (Object Graphics)*

13. Clean up 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 and the palette object:

```
OBJ_DESTROY, [oView, oPalette]
```

## Example Code: Highlighting Features with Color in Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `HighlightFeatures_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO HighlightFeatures_Object

; Determine path to "mineral.png" file.
mineralFile = FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])

; Import image from file into IDL.
mineralImage = READ_PNG(mineralFile, $
   red, green, blue)

; Determine size of imported image.
mineralSize = SIZE(mineralImage, /DIMENSIONS)

; Initialize objects.
; Initialize display.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [mineralSize[0], mineralSize[1]], $
   TITLE = 'mineral.png')
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT = [0., 0., $
   mineralSize[0], mineralSize[1]])
oModel = OBJ_NEW('IDLgrModel')
; Initialize palette and image.
oPalette = OBJ_NEW('IDLgrPalette', red, green, blue)
oImage = OBJ_NEW('IDLgrImage', mineralImage, $
   PALETTE = oPalette)

; Add image to model, then model to view, and draw final
; view in window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Update palette with RAINBOW color table and then
; display image in another instance of the window object.
oPalette -> LoadCT, 13
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [mineralSize[0], mineralSize[1]], $
   TITLE = 'RAINBOW Color')
oWindow -> Draw, oView

; Define colors for a new color table.
colorLevel = [[0, 0, 0], $ ; black
   [255, 0, 0], $ ; red
   [255, 255, 0], $ ; yellow
   [0, 255, 0], $ ; green
   [0, 255, 255], $ ; cyan
   [0, 0, 255], $ ; blue
   [255, 0, 255], $ ; magenta
   [255, 255, 255]] ; white
```

```
; Derive levels for each color in the new color table.
; NOTE: some displays may have less than 256 colors.
numberOfLevels = CEIL(!D.TABLE_SIZE/8.)
level = INDGEN(!D.TABLE_SIZE)/numberOfLevels

; Place each color level into its appropriate range.
newRed = colorLevel[0, level]
newGreen = colorLevel[1, level]
newBlue = colorLevel[2, level]

; Include the last color in the last level.
newRed[!D.TABLE_SIZE - 1] = 255
newGreen[!D.TABLE_SIZE - 1] = 255
newBlue[!D.TABLE_SIZE - 1] = 255

; Update palette with new color table and then
; display image in another instance of the window object.
oPalette -> SetProperty, RED_VALUES = newRed, $
   GREEN_VALUES = newGreen, BLUE_VALUES = newBlue
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [mineralSize[0], mineralSize[1]], $
   TITLE = 'Cube Corner Colors')
oWindow -> Draw, oView

; Clean-up object references.
OBJ_DESTROY, [oView, oPalette]

END
```

## Showing Variations in Uniform Areas

Histogram equalization is used to change either an image or its associated color table to display minor variations within nearly uniform areas of the image. The histogram of the image is used to determine where the image or color table should be equalized to highlight these minor variations. Since this chapter pertains to color and color tables, this section only discusses histogram equalization of color tables. See "Working with Histograms" on page 417 for more information on how histogram equalization effects images.

The histogram of an image shows the number of pixels for each color value within the range of the image. If the minimum value of the image is 0 and the maximum value of the image is 255, the histogram of the image shows the number of pixels for each value ranging between and including 0 and 255. Peaks in the histogram represent more common values within the image which usually consist of nearly uniform regions. Valleys in the histogram represent less common values. Empty regions within the histogram indicate that no pixels within the image contain those values.

The following figure shows an example of a histogram and its related image. The most common value in this image is 180, which appears to be the background of the image. Although the background appears nearly uniform, it contains many subtle variations (cracks).

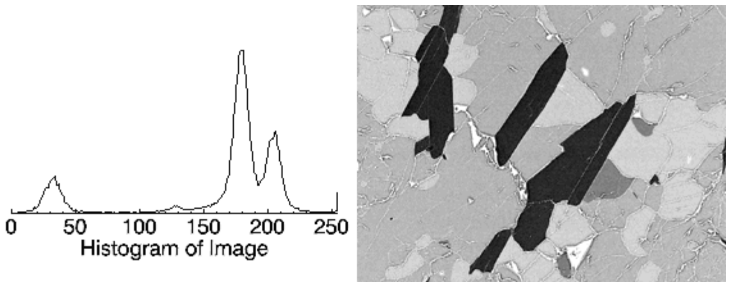


*Figure 3-27: Example of a Histogram (left) and Its Related Image (right)*

During histogram equalization, the color table values associated with the empty regions of the histogram are redistributed equally among the peaks and valleys. This process creates intensity gradients within the peaks and valleys (replacing nearly uniform values), thus highlighting minor variations.

The following section provides a histogram equalization example in Direct Graphics, which uses routines that directly work with the current color table. Since the concept of a current color table does not apply to Object Graphics, you must use histogram equalization routines that directly effect the image. See “Working with Histograms” on page 417 for more information on histogram equalization with Object Graphics.

## Showing Variations

The following example will apply histogram equalization to a color table associated with an image of mineral deposits to reveal previously indistinguishable features.

For code that you can copy and paste into an IDL Editor window, see “Example Code: Showing Variations with Direct Graphics” on page 150 or complete the following steps for a detailed description of the process.

1. Determine the path to the `mineral.png` file:

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

2. Import the image from the `mineral.png` file into IDL:

   ```
   image = READ_PNG(file)
   ```

3. Determine the size of the imported image:

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

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program:

   ```
   DEVICE, DECOMPOSED = 0
   ```

5. Initialize the image display:

   ```
   LOADCT, 0
   WINDOW, 0, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'Histogram/Image'
   ```

6. Compute and display the histogram of the image. This step is not required to perform histogram equalization on a color table within IDL. It is done here to show how the histogram equalization affects the color table:

   ```
   brightnessHistogram = BYTSCL(HISTOGRAM(image))
   PLOT, brightnessHistogram, XSTYLE = 9, YSTYLE = 5, $
      POSITION = [0.05, 0.2, 0.45, 0.9], $
      XTITLE = 'Histogram of Image'
   ```

7. Display the image within the same window.

   ```
   TV, image, 1
   ```

   The following figure shows the resulting histogram and its related image.

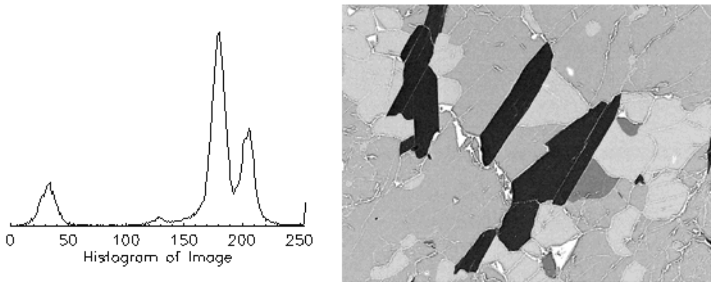


*Figure 3-28: Histogram (left) of the Mineral Image (right) in Direct Graphics*

8. Use the H_EQ_CT procedure to perform histogram equalization on the current color table:

   ```
   H_EQ_CT, image
   ```

9. Display the original image in another window with the updated color table:

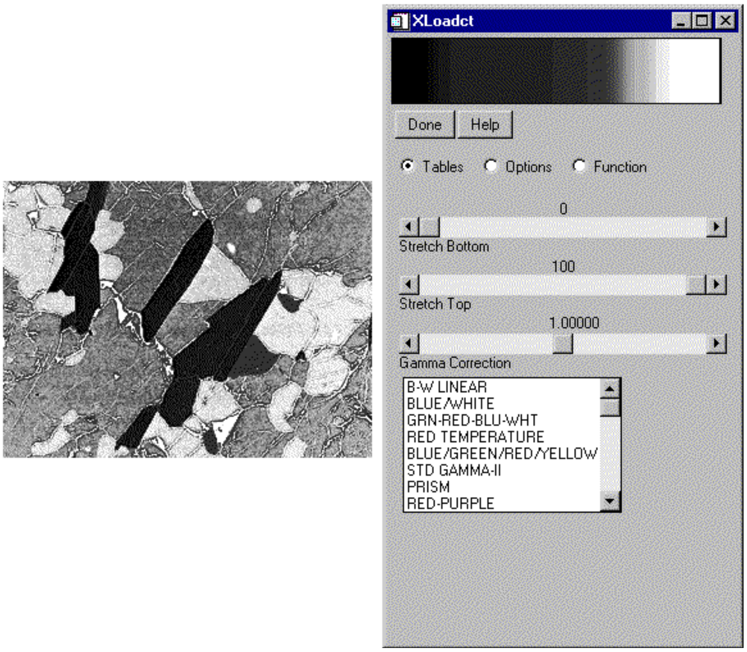
   ```
   WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'Histogram Equalized Color Table'
   TV, image
   ```

   Display the updated color table with the XLOADCT utility:

   ```
   XLOADCT
   ```

   Click on the **Done** button close the XLOADCT utility.

   The following figure contains the results of the equalization on the image and its color table. After introducing intensity gradients within previously uniform regions of the image, the cracks are now more visible. However, some of the original features are not as clear. These regions can be clarified by interactively applying the amount of equalization to the color table.



[✓ Tables] Options Function

Stretch Bottom: 0

Stretch Top: 100

Gamma Correction: 1.00000

*Figure 3-29: Resulting Image (left) and Color Table (right) of the Histogram Equalization in Direct Graphics*

The histogram equalizing process can also be interactively applied to a color table with the H_EQ_INT procedure. The H_EQ_INT procedure provides an interactive display, allowing you to use the cursor to control the amount of equalization. The equalization applied to the color table is scaled by a fraction, which is controlled by the movement of the cursor in the x-direction. If the cursor is all the way to the left side of the interactive display, the fraction equalized is close to zero, and the equalization has little effect on the color table. If the cursor is all the way to the right side of the interactive display, the fraction equalized is close to one, and the equalization is fully applied to the color table (which is similar to the results from the H_EQ_CT procedure). You can click on the right mouse button to set the amount of equalization and exit out of the interactive display.

10. Use the H_EQ_INT procedure to interactively perform histogram equalization on the current color table:

    ```
    H_EQ_INT, image
    ```

    Place the cursor at about 130 in the x-direction, which is about 0.5 equalized (about 50% of the equalization applied by the H_EQ_CT procedure). You do not have to be exact for this example. The y-direction location is arbitrary.

    Click on the right mouse button.

    The interactive display is similar to the following figure.

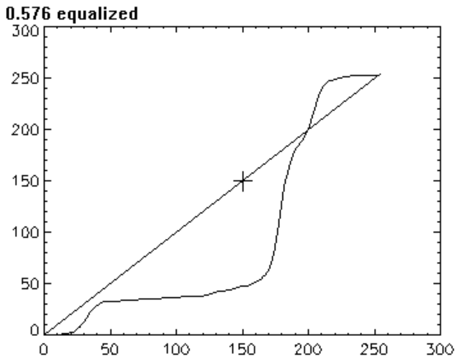


*Figure 3-30: Interactive Display for Histogram Equalization*

11. Display the image using the updated color table in another window:

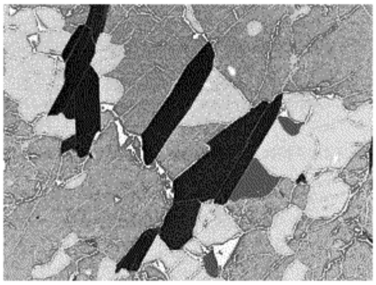
    ```
    WINDOW, 2, XSIZE = imageSize[0], YSIZE = imageSize[1], $
       TITLE = 'Interactively Equalized Color Table'
    TV, image
    ```

    Display the updated color table with the XLOADCT utility:

    ```
    XLOADCT
    ```

    Click on the **Done** button close the XLOADCT utility.

The following figure contains the results of the equalization on the image and its color table. The original details have returned and the cracks are still visible.



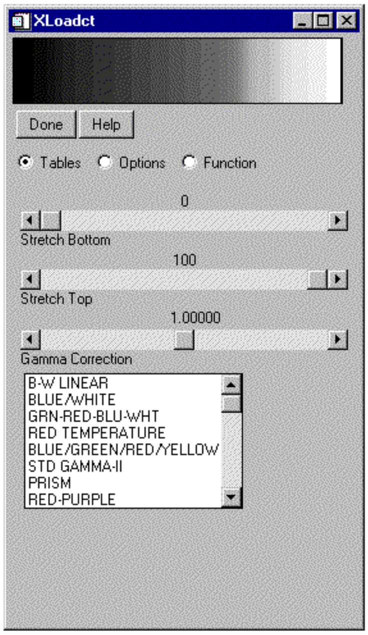


*Figure 3-31: Resulting Image (left) and Color Table (right) of the Interactive Histogram Equalization in Direct Graphics*

## Example Code: Showing Variations with Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `HistogramEqualizing_Direct.pro`, compile and run the program to reproduce the previous example. The BLOCK keyword is set when using XLOADCT to force the example routine to wait until the **Done** button is pressed to continue. If the BLOCK keyword was not set, the example routine would produce all of the displays at once and then end.

```
PRO HistogramEqualizing_Direct

; Determine path to file.
file = FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])

; Import image from file into IDL.
```

```
image = READ_PNG(file)

; Determine size of imported image.
imageSize = SIZE(image, /DIMENSIONS)

;Initialize IDL on a TrueColor display to use
; color-related routines.
DEVICE, DECOMPOSED = 0

; Initialize the image display.
LOADCT, 0
WINDOW, 0, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Histogram/Image'

; Compute and scale histogram of image.
brightnessHistogram = BYTSCL(HISTOGRAM(image))

; Display histogram plot.
PLOT, brightnessHistogram, XSTYLE = 9, YSTYLE = 5, $
   POSITION = [0.05, 0.2, 0.45, 0.9], $
   XTITLE = 'Histogram of Image'

; Display image.
TV, image, 1

; Histogram equalize the color table.
H_EQ_CT, image

; Display image and updated color table in another
; window.
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Histogram-Equalized Color Table'
TV, image

; Display the updated color table with the XLOADCT
; utility.
XLOADCT, /BLOCK

; Interactively histogram equalize the color table. The
; H_EQ_INT routine provides an interactive display to
; allow you to select the amount of equalization. Place
; the cursor at about 130 in the x-direction, which is
; about 0.5 equalized. The y-direction is arbitrary.
; Click on the right mouse button.
; NOTE: you do not have to be exact for this example.
H_EQ_INT, image

; Display image and updated color table in another
; window.
```

```
WINDOW, 2, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Interactively Equalized Color Table'
TV, image

; Display the updated color table with the XLOADCT
; utility.
XLOADCT, /BLOCK

END
```

## Applying Color Annotations to Images

Many images are annotated to explain certain features or highlight specific details. Color annotations are more noticeable than plain black or white annotations. In Direct Graphics, how color annotations are applied depends on the type of image (indexed or RGB) displayed. With indexed images, annotation colors are derived from the image's associated color table. With RGB images, annotation colors are independent of the RGB image in Direct Graphics. Annotation colors and images are separated within Object Graphics regardless of the image type.

### Applying Color Annotations to Indexed Images in Direct Graphics

Indexed images are usually associated with color tables. With Direct Graphics, these related color tables are used for all the colors shown within a display. Color tables are made up of up to 256 color triplets (red, green, and blue values of each color within the table). If you want to apply a specific color to data or to an annotation, you must change the red, green, and blue values at a specific index within the color table.

Color annotations are usually applied to label each color level within the image or to allow color comparisons. This section shows how to label each color level on an indexed image in Direct Graphics. As an example, an image of average world temperature is imported from the `worldtmp.png` file. This file does not contain a color table associated with this image, so a pre-defined color table will be applied. This table provides the colors for the polygons and text used to make a colorbar for this image. Each polygon uses the color of each level in the table. The text represents the average temperature (in Celsius) of each level.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Applying Color Annotations to Indexed Images in Direct Graphics" on page 156 or complete the following steps for a detailed description of the process.

1. Determine the path to the `worldtmp.png` file:

   ```
   worldtmpFile = FILEPATH('worldtmp.png', $
      SUBDIRECTORY = ['examples', 'demo', 'demodata'])
   ```

2. Import the image from the `worldtmp.png` file into IDL:

   ```
   worldtmpImage = READ_PNG(worldtmpFile)
   ```

3. Determine the size of the imported image:

   ```
   worldtmpSize = SIZE(worldtmpImage, /DIMENSIONS)
   ```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See “Foreground Color” on page 95 for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

   Since the imported image does not have an associated color table, the Rainbow18 color table (index number 38) is applied to the display.

5. Initialize display:

   ```
   LOADCT, 38
   WINDOW, 0, XSIZE = worldtmpSize[0], YSIZE = worldtmpSize[1],
   $
      TITLE = 'Average World Temperature (in Celsius)'
   ```

6. Now display the image with this color table:

   ```
   TV, worldtmpImage
   ```

   The following figure is displayed.

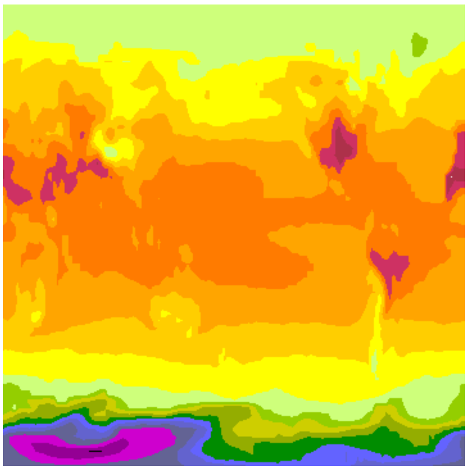


*Figure 3-32: Temperature Image and Rainbow18 Color Table (Direct Graphics)*

Before applying the color polygons and text of each level, you must first initialize their color values and their locations. The Rainbow18 color table has only 18 different color levels, but still has 256 elements. You can use the INDGEN routine to make an array of 18 elements ranging from 0 to 17 in value, where each element contains the index of that element. Then you can use the BYTSCL routine to scale these values to range from 0 to 255. The resulting array contains the initial color value (from 0 to 255) of the associated range (from 0 to 17, equalling 18 elements).

7. Initialize the color level parameter:

```
fillColor = BYTSCL(INDGEN(18))
```

8. Initialize the average temperature of each level, which directly depends on the initial color value of each range. Temperature is linearly scaled to range from -60 to 40 Celsius. You can convert the resulting temperature value to a string variable to be used as text:

```
temperature = STRTRIM(FIX(((20.*fillColor)/51.) - 60), 2)
```

**Note**

When the *fillColor* variable in the previous statement is multiplied by the floating-point value of 20 (denoted by the decimal after the number), the elements of the array are converted from byte values to floating-point values. These elements are then converted to integer values with the FIX routine so the decimal part will not be displayed. The STRTRIM routine converts the integer values to string values to be displayed as text. The second argument to STRTRIM is set to 2 to note the leading and trailing blank characters should be trimmed away when the integer values are converted to string values.

With the polygon color and text now defined, you can determine their locations. You can use the POLYFILL routine to draw each polygon and the XYOUTS routine to display each element of text. The process is repetitive from level to level, so a FOR/DO loop is used to display the entire colorbar. Since each polygon and text is drawn individually within the loop, you only need to determine the location of a single polygon and an array of offsets for each step in the loop. The following two steps describe this process.

9. Initialize the polygon and the text location parameters. Each polygon is 35 pixels in width and 18 pixels in height. The offset will move the y-location 18 pixels every time a new polygon is displayed:

```
x = [5., 40., 40., 5., 5.]
y = [5., 5., 23., 23., 5.] + 5.
offset = 18.*FINDGEN(19) + 5.
```

10. Apply the polygons and text:

```
FOR i = 0, (N_ELEMENTS(fillColor) - 1) DO BEGIN & $
   POLYFILL, x, y + offset[i], COLOR = fillColor[i], $
   /DEVICE & $
   XYOUTS, x[0] + 5., y[0] + offset[i] + 5., $
   temperature[i], COLOR = 255*(fillColor[i] LT 255), $
   /DEVICE & $
ENDFOR
```

**Note**

The & after BEGIN and the $ allow you to use the FOR/DO loop at the IDL command line. These & and $ symbols are not required when the FOR/DO loop in placed in an IDL program as shown in "Example Code: Applying Color Annotations to Indexed Images in Direct Graphics" on page 156.

The following figure displays the colorbar annotation applied to the image.

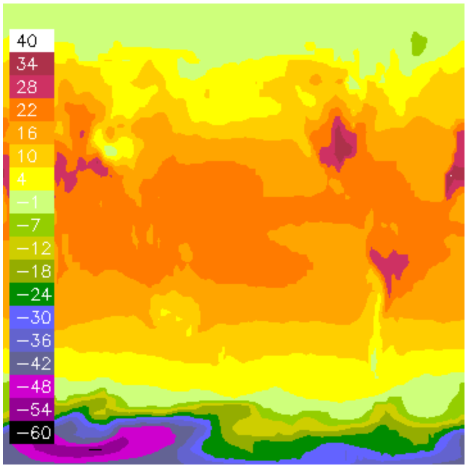


*Figure 3-33: Temperature Image and Colorbar (Direct Graphics)*

## Example Code: Applying Color Annotations to Indexed Images in Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `ApplyColorbar_Indexed_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO ApplyColorbar_Indexed_Direct

; Determine path to "worldtmp.png" file.
worldtmpFile = FILEPATH('worldtmp.png', $
   SUBDIRECTORY = ['examples', 'demo', 'demodata'])

; Import image from file into IDL.
worldtmpImage = READ_PNG(worldtmpFile)

; Determine size of imported image.
worldtmpSize = SIZE(worldtmpImage, /DIMENSIONS)

; Initialize display.
DEVICE, DECOMPOSED = 0
LOADCT, 38
WINDOW, 0, XSIZE = worldtmpSize[0], $
   YSIZE = worldtmpSize[1], $
   TITLE = 'Average World Temperature (in Celsius)'

; Display image.
TV, worldtmpImage

; Initialize color level parameter.
fillColor = BYTSCL(INDGEN(18))

; Initialize text variable.
temperature = STRTRIM(FIX(((20.*fillColor)/51.) - 60), 2)

; Initialize polygon and text location parameters.
x = [5., 40., 40., 5., 5.]
y = [5., 5., 23., 23., 5.] + 5.
offset = 18.*FINDGEN(19) + 5.

; Apply polygons and text.
FOR i = 0, (N_ELEMENTS(fillColor) - 1) DO BEGIN
   POLYFILL, x, y + offset[i], COLOR = fillColor[i], $
   /DEVICE
   XYOUTS, x[0] + 5., y[0] + offset[i] + 5., $
   temperature[i], COLOR = 255*(fillColor[i] LT 255), $
   /DEVICE
ENDFOR

END
```

## Applying Color Annotations to Indexed Images in Object Graphics

When using Object Graphics, the original color table does not need to be modified. The color table (palette) pertains only to the image object not the window, view, model, polygon, or text objects. Color annotations are usually applied to label each color level within the image or to allow color comparisons. This section shows how to label each color level on an indexed image in Object Graphics. As an example, an image of average world temperature is imported from the `worldtmp.png` file. This file does not contain a color table associated with this image, so a pre-defined color table will be applied. This table provides the colors for the polygons and text used to make a colorbar for this image. Each polygon uses the color of each level in the table. The text represents the average temperature (in Celsius) of each level.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Applying Color Annotations to Indexed Images in Object Graphics" on page 162 or complete the following steps for a detailed description of the process.

1. Determine the path to the `worldtmp.png` file:

```
worldtmpFile = FILEPATH('worldtmp.png', $
   SUBDIRECTORY = ['examples', 'demo', 'demodata'])
```

2. Import the image from the `worldtmp.png` file into IDL:

```
worldtmpImage = READ_PNG(worldtmpFile)
```

3. Determine the size of the imported image:

```
worldtmpSize = SIZE(worldtmpImage, /DIMENSIONS)
```

4. Initialize the display objects necessary for an Object Graphics display:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [worldtmpSize[0], worldtmpSize[1]], $
   TITLE = 'Average World Temperature (in Celsius)')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0, 0, worldtmpSize[0], $
   worldtmpSize[1]])
oModel = OBJ_NEW('IDLgrModel')
```

5. Initialize the palette object, load the Rainbow18 color table into the palette, and then apply the palette to an image object:

```
oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LoadCT, 38
oImage = OBJ_NEW('IDLgrImage', worldtmpImage, $
   PALETTE = oPalette)
```

6. Add the image to the model, then add the model to the view, and finally draw the view in the window:

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

   The following figure is displayed.

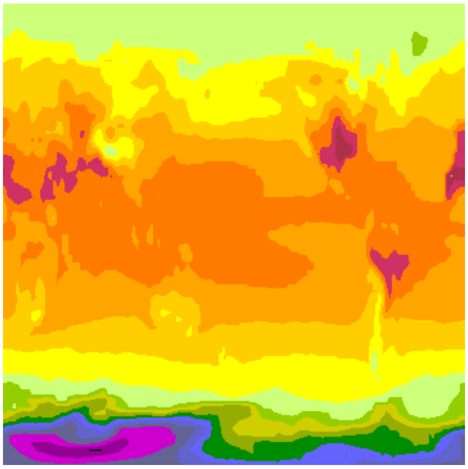


*Figure 3-34: Temperature Image and Rainbow18 Color Table (Object Graphics)*

Before applying the color polygons and text of each level, you must first initialize their color values and their locations. The Rainbow18 color table has only 18 different color levels, but still has 256 elements. You can use the INDGEN routine to make an array of 18 elements ranging from 0 to 17 in value, where each element contains the index of that element. Then you can use the BYTSCL routine to scale these values to range from 0 to 255. The resulting array contains the initial color value (from 0 to 255) of the associated range (from 0 to 17, equalling 18 elements).

7. Initialize the color level parameter:

   ```
   fillColor = BYTSCL(INDGEN(18))
   ```

8. Initialize the average temperature of each level, which directly depends on the initial color value of each range. Temperature is linearly scaled to range from -60 to 40 Celsius. You can convert the resulting temperature value to a string variable to be used as text:

   ```
   temperature = STRTRIM(FIX(((20.*fillColor)/51.) - 60), 2)
   ```

**Note**

When the *fillColor* variable in the previous statement is multiplied by the floating-point value of 20 (denoted by the decimal after the number), the elements of the array are converted from byte values to floating-point values. These elements are then converted to integer values with the FIX routine so the decimal part will not be displayed. The STRTRIM routine converts the integer values to string values to be displayed as text. The second argument to STRTRIM is set to 2 to note the leading and trailing black values should be trimmed away when the integer values are converted to string values.

With the polygon color and text now defined, you can determine their locations. You can use a polygon object to draw each polygon and text objects to display each element of text. The process is repetitive from level to level, so a FOR/DO loop is used to display the entire colorbar. Since each polygon and text is drawn individually within the loop, you only need to determine the location of a single polygon and an array of offsets for each step in the loop. The following two steps describe this process.

9. Initialize the polygon and the text location parameters. Each polygon is 35 pixels in width and 18 pixels in height. The offset will move the y-location 18 pixels every time a new polygon is displayed:

   ```
   x = [5., 40., 40., 5., 5.]
   y = [5., 5., 23., 23., 5.] + 5.
   offset = 18.*FINDGEN(19) + 5.
   ```

10. Initialize the polygon and text objects:

    ```
    oPolygon = OBJARR(18)
    oText = OBJARR(18)
    FOR i = 0, (N_ELEMENTS(oPolygon) - 1) DO BEGIN & $
       oPolygon[i] = OBJ_NEW('IDLgrPolygon', x, $
       y + offset[i], COLOR = fillColor[i], $
       PALETTE = oPalette) & $
       oText[i] = OBJ_NEW('IDLgrText', temperature[i], $
       LOCATIONS = [x[0] + 3., y[0] + offset[i] + 3.], $
       COLOR = 255*(fillColor[i] LT 255), $
       PALETTE = oPalette) & $
    ENDFOR
    ```

**Note**

The & after BEGIN and the $ allow you to use the FOR/DO loop at the IDL command line. These & and $ symbols are not required when the FOR/DO loop in placed in an IDL program as shown in "Example Code: Applying Color Annotations to Indexed Images in Object Graphics" on page 162.

11. Add the polygons and text to the model, then add the model to the view, and finally redraw the view in the window:

    ```
    oModel -> Add, oPolygon
    oModel -> Add, oText
    oWindow -> Draw, oView
    ```

    The following figure displays the colorbar annotation applied to the image.

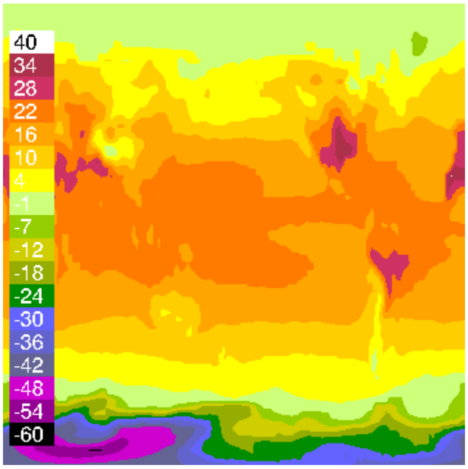


*Figure 3-35: Temperature Image and Colorbar (Object Graphics*

12. Clean up 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 and the palette objects:

    ```
    OBJ_DESTROY, [oView, oPalette]
    ```

## Example Code: Applying Color Annotations to Indexed Images in Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `ApplyColorbar_Indexed_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO ApplyColorbar_Indexed_Object

; Determine path to "worldtmp.png" file.
worldtmpFile = FILEPATH('worldtmp.png', $
   SUBDIRECTORY = ['examples', 'demo', 'demodata'])

; Import image from file into IDL.
worldtmpImage = READ_PNG(worldtmpFile)

; Determine size of imported image.
worldtmpSize = SIZE(worldtmpImage, /DIMENSIONS)

; Initialize objects.
; Initialize display.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [worldtmpSize[0], worldtmpSize[1]], $
   TITLE = 'Average World Temperature (in Celsius)')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0, 0, worldtmpSize[0], $
   worldtmpSize[1]])
oModel = OBJ_NEW('IDLgrModel')
; Initialize palette and image.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LoadCT, 38
oImage = OBJ_NEW('IDLgrImage', worldtmpImage, $
   PALETTE = oPalette)

; Add image to model, which is added to view, and then
; display view in window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Initialize color level parameter.
fillColor = BYTSCL(INDGEN(18))

; Initialize text variable.
temperature = STRTRIM(FIX(((20.*fillColor)/51.) - 60), 2)

; Initialize polygon and text location parameters.
x = [5., 40., 40., 5., 5.]
```

```
y = [5., 5., 23., 23., 5.] + 5.
offset = 18.*FINDGEN(19) + 5.

; Initialize polygon and text objects.
oPolygon = OBJARR(18)
oText = OBJARR(18)
FOR i = 0, (N_ELEMENTS(oPolygon) - 1) DO BEGIN
   oPolygon[i] = OBJ_NEW('IDLgrPolygon', x, $
   y + offset[i], COLOR = fillColor[i], $
   PALETTE = oPalette)
   oText[i] = OBJ_NEW('IDLgrText', temperature[i], $
   LOCATIONS = [x[0] + 3., y[0] + offset[i] + 3.], $
   COLOR = 255*(fillColor[i] LT 255), $
   PALETTE = oPalette)
ENDFOR

; Add polygons and text to model and then re-display
; view in window.
oModel -> Add, oPolygon
oModel -> Add, oText
oWindow -> Draw, oView

; Clean up object references.
OBJ_DESTROY, [oView, oPalette]

END
```

## Applying Color Annotations to RGB Images in Direct Graphics

RGB images contain their own color information. Color tables do not apply to RGB images. With Direct Graphics the color of the annotations on an RGB image do not depend on a color table.

**Tip**

If you are running IDL on a PseudoColor display, use the COLOR_QUAN routine to convert the RGB image to an indexed image with an associated color table to display the image and see the previous section, "Applying Color Annotations to Indexed Images in Direct Graphics" on page 153.

If you want to apply a specific color to data or an annotation, you must provide the TrueColor index for that color. The TrueColor index ranges from 0 to 16,777,216. You can derive a TrueColor index from its red, green, and blue values:

```
red = 255
green = 128
blue = 0
trueColorIndex = red + (256L*green) + ((256L^2)*blue)
PRINT, trueColorIndex
     33023
```

where *red*, *green*, and *blue* are either scalars or vectors of values ranging from 0 to 255 and representing the amount of red, green, and blue in the resulting color. The L after the numbers defines that number as a longword integer data type. The above red, green, and blue combination creates the color of orange, which has a TrueColor index of 33,023.

In this example, a color spectrum of additive and subtractive primary colors will be drawn on an RGB image for comparison with the colors in an image. The `glowing_gas.jpg` file (which is provided by the Hubble Heritage Team, made of AURA, STScI, and NASA.) contains an RGB image of an expanding shell of glowing gas surrounding a hot, massive star in our Milky Way Galaxy. This image contains all the colors of this spectrum.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Applying Color Annotations to Indexed Images in Direct Graphics" on page 156 or complete the following steps for a detailed description of the process.

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

   ```
   cosmicFile = FILEPATH('glowing_gas.jpg', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Import the image from the `glowing_gas.jpg` file into IDL:

   ```
   READ_JPEG, cosmicFile, cosmicImage
   ```

3. Determine the size of the imported image. The image contained within this file is pixel-interleaved (the color information is contained within the first dimension). You can use the SIZE routine to determine the other dimensions of this image:

   ```
   cosmicSize = SIZE(cosmicImage, /DIMENSIONS)
   ```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to one before your first RGB image is displayed within an IDL session or program. See "Foreground Color" on page 95 for more information:

```
DEVICE, DECOMPOSED = 1
```

5. Use the dimensions determined in the previous step to initialize the display:

```
WINDOW, 0, XSIZE = cosmicSize[1], YSIZE = cosmicSize[2], $
   TITLE = ‘glowing_gas.jpg'
```

6. Now display the image with the TRUE keyword set to 1 since the image is pixel interleaved:

```
TV, cosmicImage, TRUE = 1
```

The following figure shows that the image contains all of the colors of the additive and subtractive primary spectrum. In the following steps, a colorbar annotation will be added to allow you to compare the colors of that spectrum and the colors within the image.

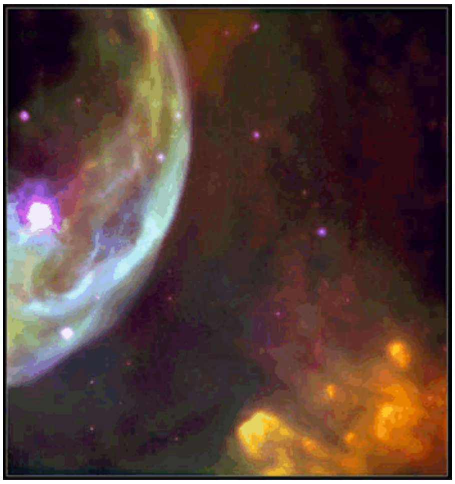


*Figure 3-36: Cosmic RGB Image (Direct Graphics)*

You can use the following to determine the color and location parameters for each polygon.

7. Initialize the color parameters:

   ```
   red = BYTARR(8) & green = BYTARR(8) & blue = BYTARR(8)
   red[0] = 0 & green[0] = 0 & blue[0] = 0 ; black
   red[1] = 255 & green[1] = 0 & blue[1] = 0 ; red
   red[2] = 255 & green[2] = 255 & blue[2] = 0 ; yellow
   red[3] = 0 & green[3] = 255 & blue[3] = 0 ; green
   red[4] = 0 & green[4] = 255 & blue[4] = 255 ; cyan
   red[5] = 0 & green[5] = 0 & blue[5] = 255 ; blue
   red[6] = 255 & green[6] = 0 & blue[6] = 255 ; magenta
   red[7] = 255 & green[7] = 255 & blue[7] = 255 ; white
   fillColor = red + (256L*green) + ((256L^2)*blue)
   ```

8. After defining the polygon colors, you can determine their locations. Initialize polygon location parameters:

   ```
   x = [5., 25., 25., 5., 5.]
   y = [5., 5., 25., 25., 5.] + 5.
   offset = 20.*FINDGEN(9) + 5.
   ```

   The *x* and *y* variables pertain to the x and y locations (in pixel units) of each box of color. The *offset* maintains the spacing (in pixel units) of each box. Since the image is made up of mostly a black background, the x border of the colorbar is also determined to draw a white border around the polygons.

9. Initialize location of colorbar border:

   ```
   x_border = [x[0] + offset[0], x[1] + offset[7], $
      x[2] + offset[7], x[3] + offset[0], x[4] + offset[0]]
   ```

   The y border is already defined by the y variable.

   These parameters are used with POLYFILL and PLOTS to draw the boxes of the color spectrum and the colorbar border. Each polygon is 20 pixels wide and 20 pixels high. The offset will move the y-location 20 pixels every time a new polygon is displayed.

10. Apply the polygons and border. You can use the POLYFILL routine to draw each polygon. The process is repetitive from level to level, so a FOR/DO loop is used to display the entire colorbar. Since each polygon is drawn individually within the loop, you only need to determine the location of a single polygon and an array of offsets for each step in the loop:

    ```
    FOR i = 0, (N_ELEMENTS(fillColor) - 1) DO POLYFILL, $
       x + offset[i], y, COLOR = fillColor[i], /DEVICE
    PLOTS, x_border, y, COLOR = fillColor[7], /DEVICE
    ```

The POLYFILL and PLOTS routines result in the following display.

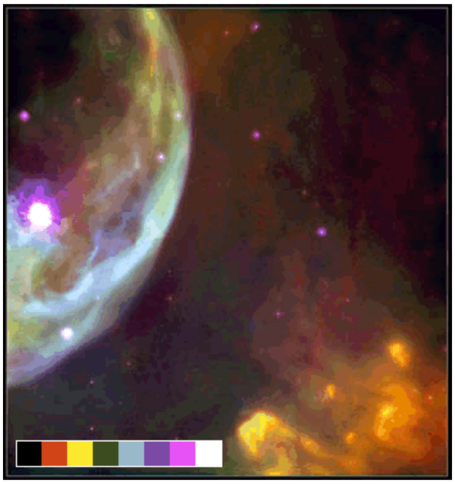


*Figure 3-37: Specified Colors on an RGB Image (Direct Graphics)*

## Example Code: Applying Color Annotations to RGB Images in Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `ApplyColorbar_RGB_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO ApplyingColorbar_RGB_Direct

; Determine path to "glowing_gas.jpg" file.
cosmicFile = FILEPATH('glowing_gas.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Import image from file into IDL.
READ_JPEG, cosmicFile, cosmicImage

; Determine size of image.
cosmicSize = SIZE(cosmicImage, /DIMENSIONS)

; Initialize display.
DEVICE, DECOMPOSED = 1
WINDOW, 0, TITLE = 'glowing_gas.jpg', $
   XSIZE = cosmicSize[1], YSIZE = cosmicSize[2]
```

```
; Diplay image.
TV, cosmicImage, TRUE = 1

; Initialize color parameters.
red = BYTARR(8) & green = BYTARR(8) & blue = BYTARR(8)
red[0] = 0 & green[0] = 0 & blue[0] = 0 ; black
red[1] = 255 & green[1] = 0 & blue[1] = 0 ; red
red[2] = 255 & green[2] = 255 & blue[2] = 0 ; yellow
red[3] = 0 & green[3] = 255 & blue[3] = 0 ; green
red[4] = 0 & green[4] = 255 & blue[4] = 255 ; cyan
red[5] = 0 & green[5] = 0 & blue[5] = 255 ; blue
red[6] = 255 & green[6] = 0 & blue[6] = 255 ; magenta
red[7] = 255 & green[7] = 255 & blue[7] = 255 ; white
fillColor = red + (256L*green) + ((256L^2)*blue)

; Initialize polygon location parameters.
x = [5., 25., 25., 5., 5.]
y = [5., 5., 25., 25., 5.] + 5.
offset = 20.*FINDGEN(9) + 5.

; Initialize location of colorbar border.
x_border = [x[0] + offset[0], x[1] + offset[7], $
   x[2] + offset[7], x[3] + offset[0], x[4] + offset[0]]

; Apply polygons and border.
FOR i = 0, (N_ELEMENTS(fillColor) - 1) DO POLYFILL, $
   x + offset[i], y, COLOR = fillColor[i], /DEVICE
PLOTS, x_border, y, /DEVICE, COLOR = fillColor[7]

END
```

## Applying Color Annotations to RGB Images in Object Graphics

When using Object Graphics, colors can be defined just by the values of their red, green, and blue components. The TrueColor index conversion equation is not required for Object Graphics. In this example, a color spectrum of additive and subtractive primary colors will be drawn on an RGB image for comparison with the colors in that image. The `glowing_gas.jpg` file (which is provided by the Hubble Heritage Team, made up of AURA, STScI, and NASA) contains an RGB image of an expanding shell of glowing gas surrounding a hot, massive star in our Milky Way Galaxy. This image contains all the colors of this spectrum.

For code that you can copy and paste into an IDL Editor window, see “Example Code: Applying Color Annotations to RGB Images in Object Graphics” on page 172 or complete the following steps for a detailed description of the process.

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

   ```
   cosmicFile = FILEPATH('glowing_gas.jpg', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Import the image from the `glowing_gas.jpg` file into IDL:

   ```
   READ_JPEG, cosmicFile, cosmicImage
   ```

3. Determine the size of the imported image. The image contained within this file is pixel-interleaved (the color information is contained within the first dimension). You can use the SIZE routine to determine the other dimensions of this image:

   ```
   cosmicSize = SIZE(cosmicImage, /DIMENSIONS)
   ```

4. Initialize the display objects required for an Object Graphics display:

   ```
   oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
      DIMENSIONS = [cosmicSize[1], cosmicSize[2]], $
      TITLE = 'glowing_gas.jpeg')
   oView = OBJ_NEW('IDLgrView', $
      VIEWPLANE_RECT = [0., 0., cosmicSize[1], $
      cosmicSize[2]])
   oModel = OBJ_NEW('IDLgrModel')
   ```

5. Initialize the image object. The INTERLEAVE keyword is set to 0 because the RGB image is pixel-interleaved:

   ```
   oImage = OBJ_NEW('IDLgrImage', cosmicImage, $
      INTERLEAVE = 0, DIMENSIONS = [cosmicSize[1], $
      cosmicSize[2]])
   ```

6. Add the image to the model, then add the model to the view, and finally draw the view in the window:

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

The following image contains all of the colors of the additive and subtractive primary spectrum. A colorbar annotation can be added to compare the colors of that spectrum and the colors within the image. The color of each box is defined in the following array.

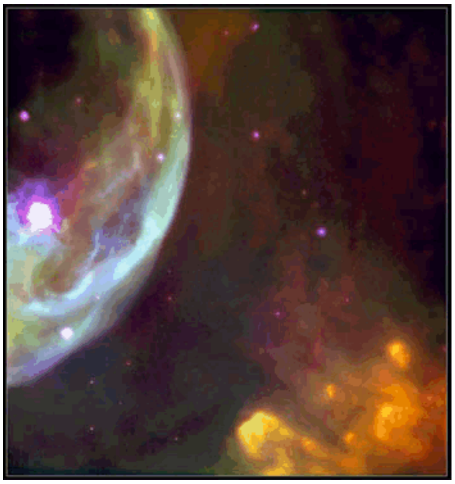


*Figure 3-38: Cosmic RGB Image (Object Graphics)*

You can use the following to determine the color and location parameters for each polygon.

7. Initialize the color parameters:

```
fillColor = [[0, 0, 0], $ ; black
   [255, 0, 0], $ ; red
   [255, 255, 0], $ ; yellow
   [0, 255, 0], $ ; green
   [0, 255, 255], $ ; cyan
   [0, 0, 255], $ ; blue
   [255, 0, 255], $ ; magenta
   [255, 255, 255]] ; white
```

8. After defining the polygon colors, you can determine their locations. Initialize polygon location parameters:

```
x = [5., 25., 25., 5., 5.]
y = [5., 5., 25., 25., 5.] + 5.
offset = 20.*FINDGEN(9) + 5.
```

The *x* and *y* variables pertain to the x and y locations (in pixel units) of each box of color. The *offset* maintains the spacing (in pixel units) of each box. Since the image is made up of mostly a black background, the x border of the colorbar is also determined to draw a white border around the polygons.

9. Initialize location of colorbar border:

```
x_border = [x[0] + offset[0], x[1] + offset[7], $
   x[2] + offset[7], x[3] + offset[0], x[4] + offset[0]]
```

   The y border is already defined by the y variable.

   These parameters are used when initializing the polygon and polyline objects These objects will be used draw the boxes of the color spectrum and the colorbar border. Each polygon is 20 pixels wide and 20 pixels high. The offset will move the y-location 20 pixels every time a new polygon is displayed.

10. Initialize the polygon objects. The process is repetitive from level to level, so a FOR/DO loop will be used to display the entire colorbar. Since each polygon is drawn individually within the loop, you only need to determine the location of a single polygon and an array of offsets for each step in the loop:

```
oPolygon = OBJARR(8)
FOR i = 0, (N_ELEMENTS(oPolygon) - 1) DO oPolygon[i] = $
   OBJ_NEW('IDLgrPolygon', x + offset[i], y, $
   COLOR = fillColor[*, i])
```

11. The colorbar border is produced with a polyline object. This polyline object requires a *z* variable to define it slightly above the polygons and image. The z variable is required to place the polyline in front of the polygons. Initialize the polyline (border) object:

```
z = [0.001, 0.001, 0.001, 0.001, 0.001]
oPolyline = OBJ_NEW('IDLgrPolyline', x_border, y, z, $
   COLOR = [255, 255, 255])
```

12. The polygon and polyline objects can now be added to the model and then displayed (re-drawn) in the window. Add the polygons and polyline to the model, then add the model to the view, and finally redraw the view in the window:

```
oModel -> Add, oPolygon
oModel -> Add, oPolyline
oWindow -> Draw, oView
```

The following figure shows the colorbar annotation applied to the image.

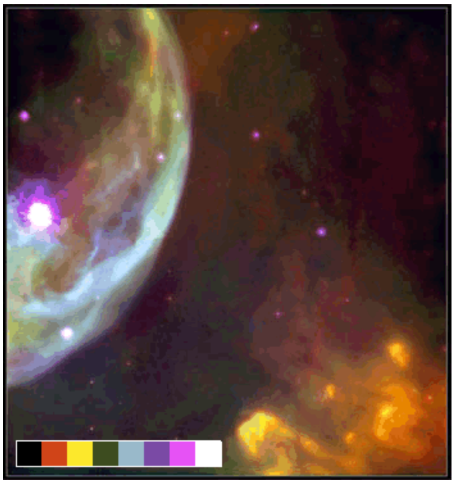


*Figure 3-39: Specified Colors on an RGB Image (Object Graphics)*

13. Clean up 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
    ```

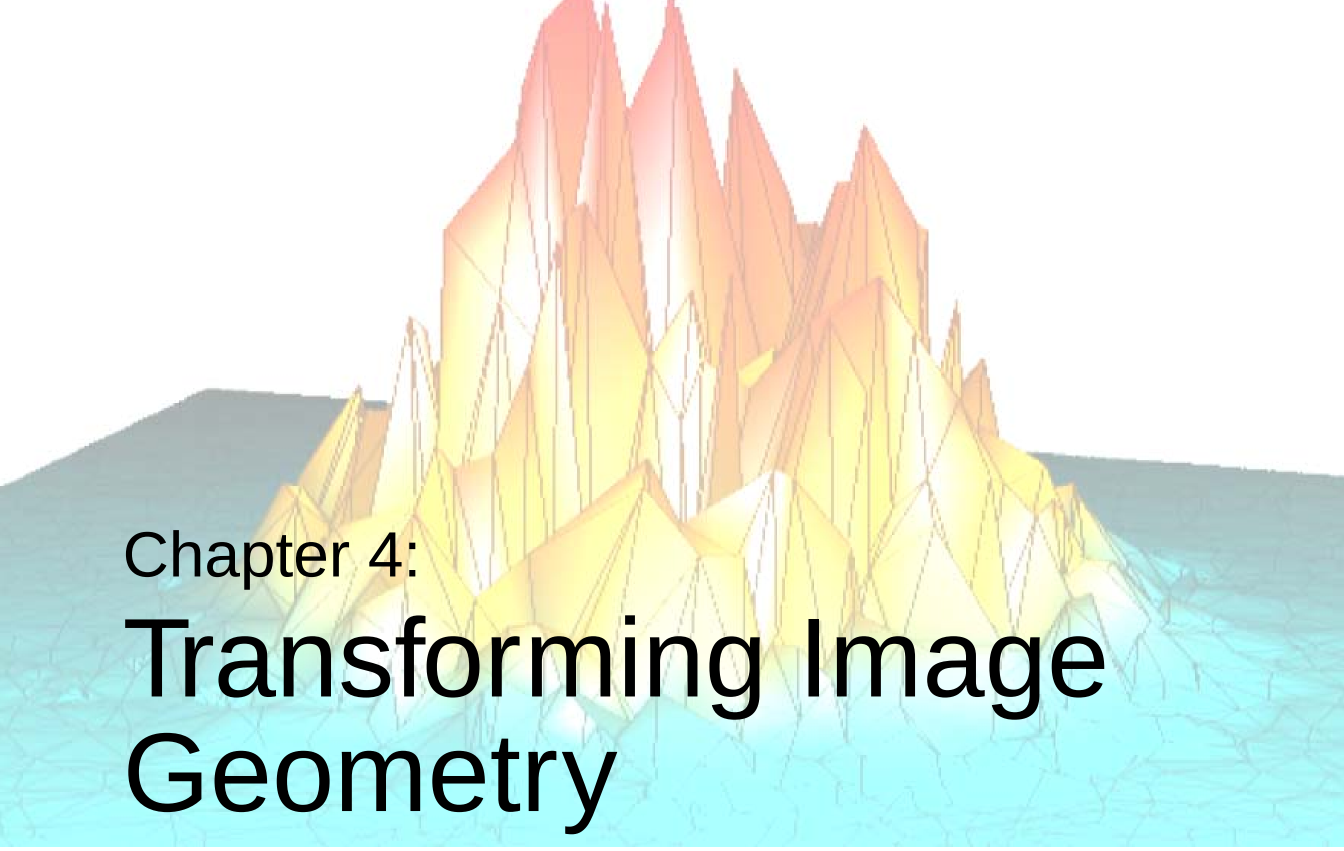
## Example Code: Applying Color Annotations to RGB Images in Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `ApplyColorbar_RGB_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO ApplyColorbar_RGB_Object

; Determine path to "glowing_gas.jpg" file.
cosmicFile = FILEPATH('glowing_gas.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Import image from file into IDL.
READ_JPEG, cosmicFile, cosmicImage

; Determine size of image.
```

```
cosmicSize = SIZE(cosmicImage, /DIMENSIONS)

; Initialize objects.
; Initialize display.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [cosmicSize[1], cosmicSize[2]], $
   TITLE = 'glowing_gas.jpg')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., cosmicSize[1], $
   cosmicSize[2]])
oModel = OBJ_NEW('IDLgrModel')
; Initialize image.
oImage = OBJ_NEW('IDLgrImage', cosmicImage, $
   INTERLEAVE = 0, DIMENSIONS = [cosmicSize[1], $
   cosmicSize[2]])

; Add image to model, which is added to view, and then
; display view in window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Initialize color parameter.
fillColor = [[0, 0, 0], $ ; black
   [255, 0, 0], $ ; red
   [255, 255, 0], $ ; yellow
   [0, 255, 0], $ ; green
   [0, 255, 255], $ ; cyan
   [0, 0, 255], $ ; blue
   [255, 0, 255], $ ; magenta
   [255, 255, 255]] ; white

; Initialize polygon location parameters.
x = [5., 25., 25., 5., 5.]
y = [5., 5., 25., 25., 5.] + 5.
offset = 20.*FINDGEN(9) + 5.

; Initialize location of colorbar border.
x_border = [x[0] + offset[0], x[1] + offset[7], $
   x[2] + offset[7], x[3] + offset[0], x[4] + offset[0]]

; Initialize polygon objects.
oPolygon = OBJARR(8)
FOR i = 0, (N_ELEMENTS(oPolygon) - 1) DO oPolygon[i] = $
   OBJ_NEW('IDLgrPolygon', x + offset[i], y, $
   COLOR = fillColor[*, i])

; Initialize polyline (border) object.
z = [0.001, 0.001, 0.001, 0.001, 0.001]
```

```
oPolyline = OBJ_NEW('IDLgrPolyline', x_border, y, z, $
   COLOR = [255, 255, 255])

; Add polgons and polyline to model and then re-display
; view in window.
oModel -> Add, oPolygon
oModel -> Add, oPolyline
oWindow -> Draw, oView

; Clean up object references.
OBJ_DESTROY, oView

END
```



Chapter 4:

# Transforming Image Geometry

This chapter describes the following topics:

## Overview of Geometric Transformations

Geometric image transformation functions use mathematical transformations to crop, pad, scale, rotate, transpose or otherwise alter an image array to produce a modified view of an image. The transformations described in this chapter are linear transformations. For a description of non-linear geometric transformations, see Chapter 7, "Warping Images".

When an image undergoes a geometric transformation, some or all of the pixels within the source image are relocated from their original spatial coordinates to a new position in the output image. When a relocated pixel does not map directly onto the center of a pixel location, but falls somewhere in between the centers of pixel locations, the pixel's value is computed by sampling the values of the neighboring pixels. This resampling, also known as interpolation, affects the quality of the output image. See "Interpolation Methods" on page 178 for more information.

**Note**

In this book, Direct Graphics examples are provided by default. Object Graphics examples are provided in cases where significantly different methods are required.

The following list introduces image processing tasks and associated IDL image processing routines covered in this chapter.

| Task | Routine(s) | Description |
|---|---|---|
| "Cropping Images" on page 180. | SIZE<br>CURSOR | Focuses attention on important image features by creating a rectangular region of interest. |
| "Padding Images" on page 184. | SIZE | Creates a border around the perimeter of an image for presentation or advanced filtering purposes. |
| "Resizing Images" on page 188. | CONGRID<br>REBIN | Enlarges or shrinks an image. |
| "Shifting Images" on page 191. | SHIFT | Shifts image pixel values along any image dimension. |

*Table 4-1: Image Processing Tasks and Related Image Processing Routines*

| Task | Routine(s) | Description |
|---|---|---|
| "Reversing Images" on page 194. | REVERSE | Reverses array elements to flip an image horizontally or vertically. |
| "Transposing Images" on page 197. | TRANSPOSE | Interchanges array dimensions, reflecting the image about a 45 degree line. |
| "Rotating Images" on page 200. | ROTATE<br>ROT | Rotates an image to any orientation, using 90 degree or arbitrary increments. |
| "Planar Slicing of Volumetric Data" on page 206. | EXTRACT_SLICE<br>SLICER3<br>XVOLUME | Displays a single slice or a series of planar slices in a single window or interactively extracts planar slices of volumetric data. |

*Table 4-1: Image Processing Tasks and Related Image Processing Routines*

**Note**

This chapter uses data files from the `IDL examples/data` directory. Two files, `data.txt` and `index.txt`, contain descriptions of the files, including array sizes.

## Interpolation Methods

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

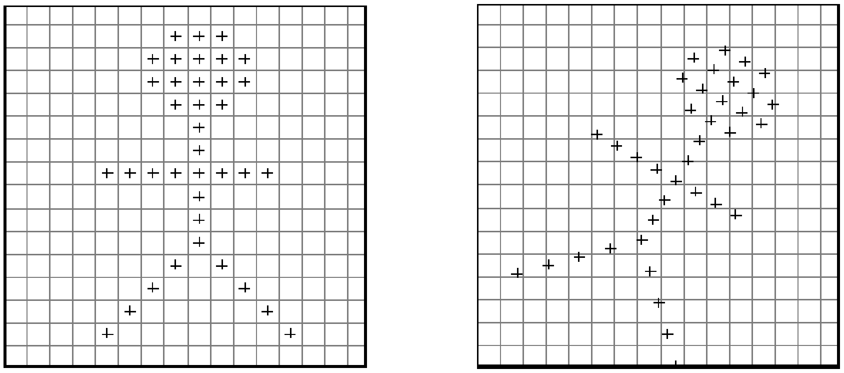


*Figure 4-1: Original Pixel Center Locations (Left) and Rotated Pixel Center Locations (Right)*

When the transformed pixel center does not directly coincide with a pixel in the output image, 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.

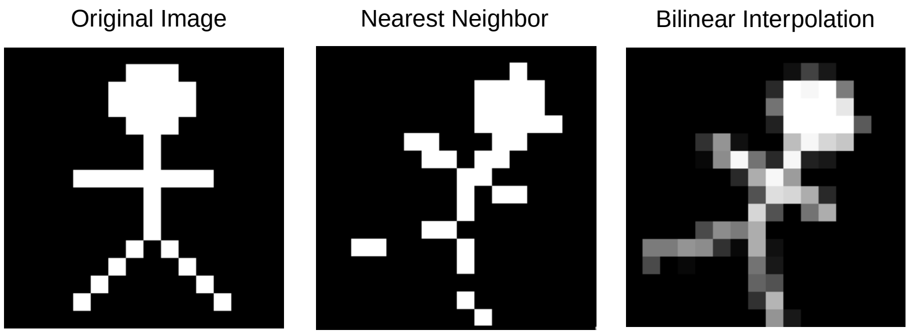


*Figure 4-2: Simple Examples of Image Interpolation*

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 image. 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**

See the *IDL Reference Guide* for complete details about the interpolation options available with each geometric image transformation function.

## Cropping Images

Cropping an image extracts a rectangular region of interest from the original image. This focuses the viewer's attention on a specific portion of the image and discards areas of the image that contain less useful information. Using image cropping in conjunction with image magnification allows you to zoom in on a specific portion of the image. This section describes how to exactly define the portion of the image you wish to extract to create a cropped image. For information on how to magnify a cropped image, see "Resizing Images" on page 188.

Image cropping requires a pair of (*x*, *y*) coordinates that define the corners of the new, cropped image. The following example extracts the African continent from an image of the world. For code that you can copy and paste into an IDL Editor window, see "Example Code: Cropping an Image" on page 183.

1. Open the world image file, using the `R,G,B` arguments to obtain the image's color information:

```
world = READ_PNG (FILEPATH ('avhrr.png', $
   SUBDIRECTORY = ['examples', 'data']), R, G, B)
```

2. Prepare the display device and load the color table with the red, green and blue values retrieved from the image file in the previous step:

```
DEVICE, RETAIN = 2, DECOMPOSED = 0
TVLCT, R, G, B
```

3. Get the size of the image and prepare the window display using the dimensions returned by the SIZE command:

```
worldSize = SIZE(world, /DIMENSIONS)
WINDOW, 0, XSIZE = worldSize[0], YSIZE = worldSize[1]
```

4. Display the image:

```
TV, world
```

In this example, we will crop the image to display only the African continent as shown in the following figure. Two sets of coordinates, (*LeftLowX*, *LeftLowY*) and (*RightTopX*, *RightTopY*), will be used to create the new, cropped image array.

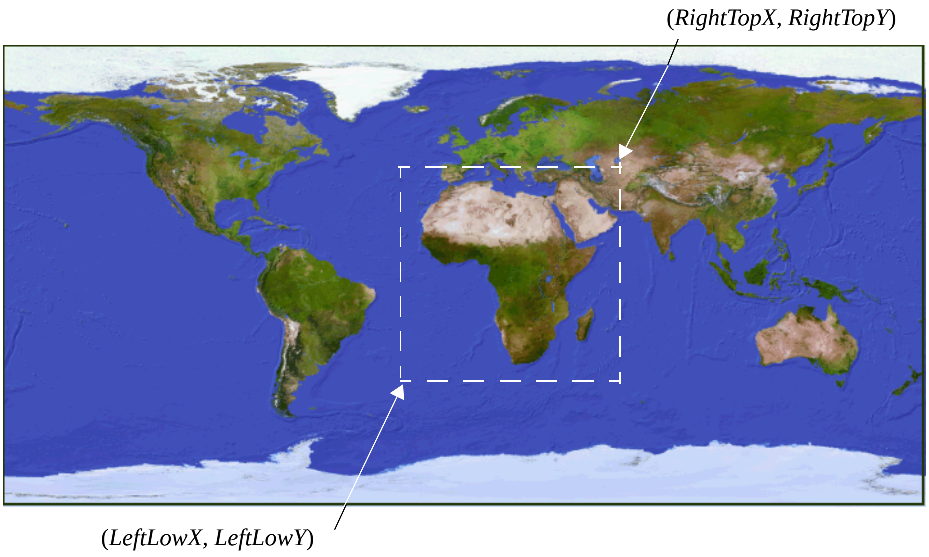


*Figure 4-3: Defining the Boundaries of the Cropped Image Array*

In the following step, use the CURSOR function to define the boundaries of the cropped image. The values returned by the CURSOR function will be defined as the variables shown in the previous image.

**Note**

To crop an image without interactively defining the cursor position, you can use the actual coordinates of the cropped image array in place of the coordinate variables, (*LeftLowX*, *LeftLowY*) and (*RightTopX*, *RightTopY*). See "Example Code: Cropping an Image" on page 183 for an example.

5. Use the cursor function to define the lower-left corner of the cropped image by entering the following line:

   ```
   CURSOR, LeftLowX, LeftLowY, /DEVICE
   ```

   The cursor changes to a cross hair symbol when it is positioned over the graphics window. Click in the area to the left and below the African continent.

**Note**

The values for *`LeftLowX`* and *`LeftLowY`* appear in the IDLDE Variable Watch window. Alternately, use `PRINT, LeftLowX, LeftLowY` to display these values.

6. Define the upper-right corner of the cropped image. Enter the following line and then click above and to the right of the African continent.

```
CURSOR, RightTopX, RightTopY, /DEVICE
```

7. Name the cropped image and define its array using the lower-left and upper-right *x* and *y* variables:

```
africa = world[LeftLowX:RightTopX, LeftLowY:RightTopY]
```

8. Prepare a window based on the size of the new array:

```
WINDOW, 2, XSIZE = (RightTopX - LeftLowX + 1), $
   YSIZE = (RightTopY - LeftLowY + 1)
```

9. Display the cropped image:

```
TV, africa
```

Your image should appear similar to the following figure.

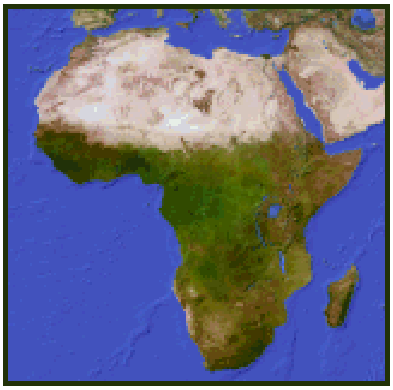


*Figure 4-4: Result of the Cropped Image Example*

## Example Code: Cropping an Image

The following program creates the same cropped image as the previous example but uses numeric coordinates instead of named variable coordinates defined using the interactive CURSOR function. Copy the following text into an IDL Editor window. After saving the file as `CropWorld.pro`, compile and run the program to reproduce the cropped image example.

```
PRO CropWorld

; Read in the image file.
world = READ_PNG(FILEPATH('avhrr.png', $
   SUBDIRECTORY = ['examples', 'data']), R,G,B)

; Prepare the display device and load the color table.
DEVICE, DECOMPOSED = 0, RETAIN = 2
TVLCT, R, G, B

; Get the size of the image array.
worldSize = SIZE(world, /DIMENSIONS)

; Use the returned dimensions to create a display window
; and display the original image.
WINDOW, 0, XSIZE = worldSize[0], YSIZE = worldSize[1]
TV, world

; Note: the following  section uses numeric coordinates to
crop
; the array instead of defining coordinates using the CURSOR
; function. Compared to the step-by-step example, this line
has
; the following structure:
; africa = world[LeftLowX:RightTopX, LeftLowY:RightTopY]
africa = world [312:475, 103:264]

; Define the window size based on the size of the cropped
array
; using XSIZE = (RightTopX - LeftLowX + 1),
; YSIZE = (RightTopY - LeftLowY + 1)
WINDOW, 2, XSIZE =(475-312 + 1), YSIZE =(264-103 + 1)

; Display the cropped image.
TV, africa

END
```

## Padding Images

Image padding introduces new pixels around the edges of an image. The border provides space for annotations or acts as a boundary when using advanced filtering techniques.

This exercise adds a 10-pixel border to left, right and bottom of the image and a 30-pixel border at the top allowing space for annotation. The diagonal lines in the following image represent the area that will be added to the original image. For an example of padding an image, complete the following steps. If you prefer to cut and paste the entire example into an IDL Editor window, see "Example Code: Padding an Image" on page 187.

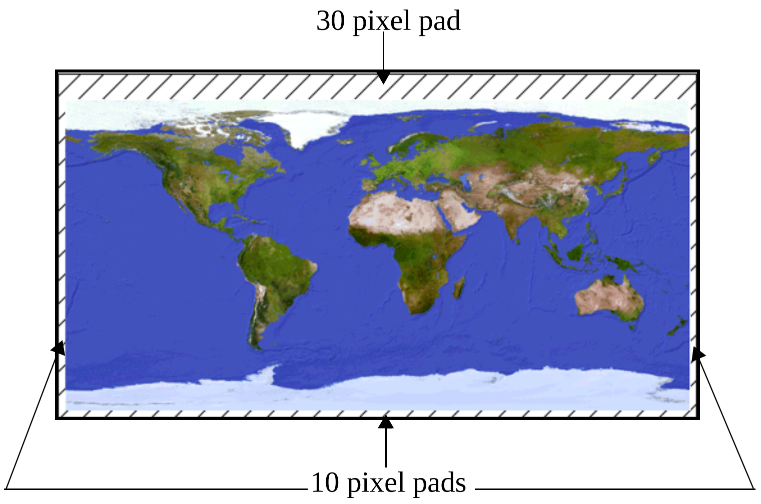


*Figure 4-5: Diagonal Lines Indicate Padding*

To add a border around the earth image, complete the following steps:

1. Open the world image file:

```
earth = READ_PNG(FILEPATH('avhrr.png', $
   SUBDIRECTORY = ['examples', 'data']), R, G, B)
```

2. Prepare the display device:

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

3. Load the color table with the red, green and blue values retrieved from the image in step 1 and modify the color table so that the final index value of each color band is the maximum color value (white):

   ```
   TVLCT, R, G, B
   maxColor = !D.TABLE_SIZE - 1
   TVLCT, 255, 255, 255, maxColor
   ```

4. Get the size of the image by entering the following line:

   ```
   earthSize = SIZE(earth, /DIMENSIONS)
   ```

5. Define the amount of padding you want to add to the image. This example adds 10 pixels to the right and left sides of the image equalling a total of 20 pixels along the x-axis. We also add 30 pixels to the top and 10 pixels to the bottom of the image for a total of 40 pixels along the y-axis.

   Using the REPLICATE syntax, *Result* = REPLICATE (*Value*, D1 [, ..., D8]), create an array of the specified dimensions, and set *Value* equal to the byte value of the final color index to make the white border:

   ```
   paddedEarth = REPLICATE(BYTE(maxColor), earthSize[0] + 20, $
      earthSize[1] + 40)
   ```

**Note**

The argument `BYTE(maxColor)` in the previous line produces a white background only when white is designated as the final index value for the red, green and blue bands of the color table you are using. As shown in step 3, this can be accomplished by setting each color component (of the color table entry indexed by *maxColor*) to 255.

See Chapter 3, “Working with Color” for detailed information about modifying color tables.

6. Copy the original image, *earth*, into the appropriate portion of the padded array. The following line places the lower-left corner of the original image array at the coordinates (10, 10) of the padded array:

   ```
   paddedEarth [10,10] = earth
   ```

7. Prepare a window to display the image using the size of the original image plus the amount of padding added along the x and y axes:

   ```
   WINDOW, 0, XSIZE = earthSize[0] + 20, $
      YSIZE = earthSize[1] + 40
   ```

8. Display the padded image.

```
TV, paddedEarth
```

9. Place a title at the top of the image using the XYOUTS procedure.

```
x = (earthSize[0]/2) + 10
y = earthSize[1] + 15
XYOUTS, x, y, 'World Map', ALIGNMENT = 0.5, COLOR = 0, $
   /DEVICE
```

The resulting image should appear similar to the following figure.

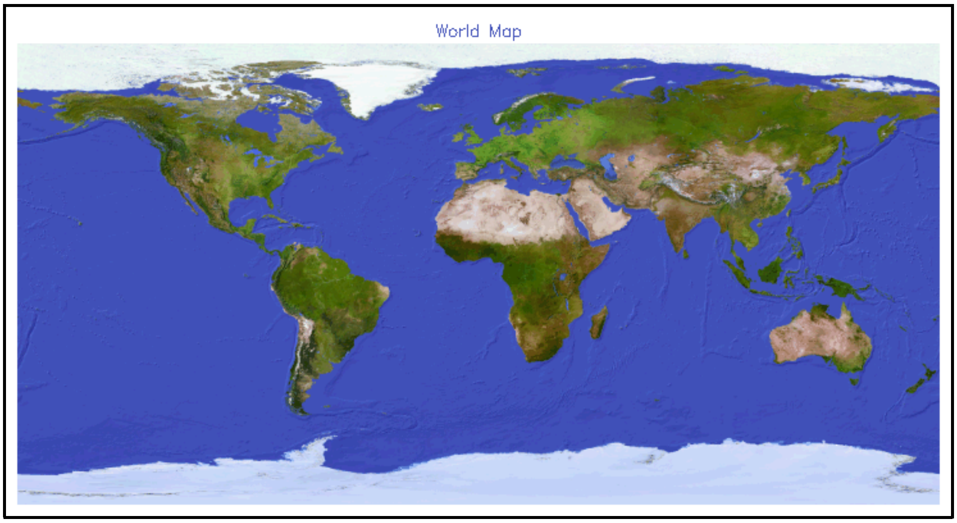


*Figure 4-6: Resulting Padded Image*

## Example Code: Padding an Image

Copy the following code into the IDL Editor window and save it as `PaddedImage.pro`. Compile and run the program to duplicate the image padding example.

```
PRO PaddedImage

; 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)

; Return an array with the given dimensions.
paddedEarth = REPLICATE(BYTE(maxColor), earthSize[0] + 20, $
   earthSize[1] + 40)

; Copy the original image into the appropriate portion
; of the new array.
paddedEarth [10,10] = earth

; Prepare a window and display the new image.
WINDOW, 0, XSIZE = earthSize[0] + 20, $
   YSIZE = earthSize[1] + 40
TV, paddedEarth

; Place a title at the top of the image using the
; XYOUTS procedure.
x = (earthSize[0]/2) + 10
y = earthSize[1] + 15
XYOUTS, x, y, 'World Map', ALIGNMENT = 0.5, COLOR = 0, $
   /DEVICE

END
```

## Resizing Images

Image resizing, or scaling, supports further image analysis by either shrinking or expanding an image. Both the CONGRID and the REBIN functions resize one-, two- or three-dimensional arrays. The CONGRID function resizes an image array by any arbitrary amount. The REBIN function requires that the output dimensions of the new array be an integer multiple of the original image's dimensions.

When magnifying an image, new values are interpolated from the source image to produce additional pixels in the output image.When shrinking an image, pixels are resampled to produce a lower number of pixels in the output image. The default interpolation method varies according to whether you are magnifying or shrinking the image.

When magnifying an image:

- CONGRID defaults to nearest-neighbor sampling with 1D or 2D arrays and automatically uses bilinear interpolation with 3D arrays.
- REBIN defaults to bilinear interpolation.

When shrinking an image:

- CONGRID uses nearest-neighbor interpolation to resample the image.
- REBIN averages neighboring pixel values in the source image that contribute to a single pixel value in the output image.

The following example uses CONGRID since it offers more flexibility. However, if you wish to resize an array proportionally, REBIN returns results more quickly. For an example of magnifying an image using the CONGRID function, complete the following steps. For code that you can copy and paste into an IDL Editor window, see "Example: Resizing an Image Using CONGRID" on page 190.

1. Select the file and read in the data, specifying known data dimensions:

```
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = [248, 248])
```

2. Load a color table and prepare the display device:

```
LOADCT, 28
DEVICE, DECOMPOSED = 0, RETAIN = 2
```

3. Prepare the window and display the original image:

```
WINDOW, 0, XSIZE = 248, YSIZE = 248
TV, image
```

4. Use the CONGRID function to increase the image array size to 600 by 600 pixels and force bilinear interpolation:

```
magnifiedImg = CONGRID(image, 600, 600, /INTERP)
```

5. Display the magnified image in a new window:

```
WINDOW, 1, XSIZE = 600, YSIZE = 600
TV, magnifiedImg
```

The following figure displays the original image (left) and the magnified view of the image (right).

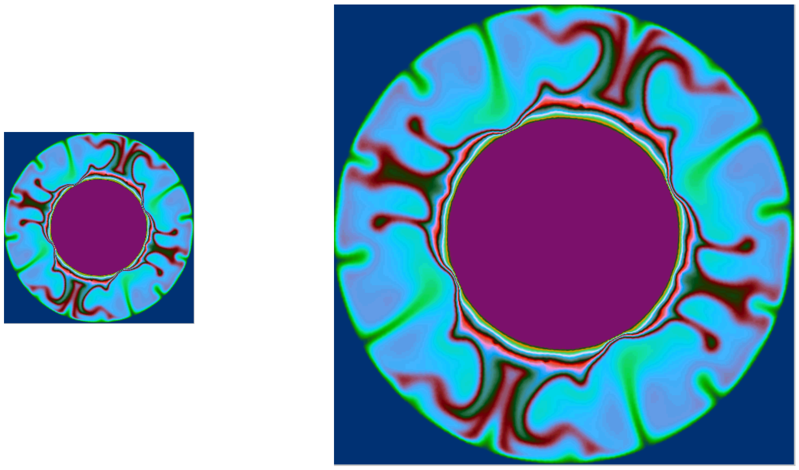


*Figure 4-7: Original Image and Magnified Image*

## Example: Resizing an Image Using CONGRID

Copy and paste the following text into the Editor window. After saving the file as `MagnifyImage.pro`, compile and run the program to reproduce the CONGRID function example.

```
PRO MagnifyImage

; Select the file, and read in the data using known dimensions.
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = [248, 248])

; Load a color table and prepare to display the image.
LOADCT, 28
DEVICE, DECOMPOSED = 0, RETAIN = 2
WINDOW, 0, XSIZE = 248, YSIZE = 248

; Display the original image.
TV, image

; Magnify the image and display it in a new window.
magnifiedImg = CONGRID(image, 600, 600, /INTERP)
WINDOW, 1, XSIZE = 600, YSIZE = 600
TV, magnifiedImg

END
```

## Shifting Images

The SHIFT function moves elements of a vector or array along any dimension by any number of elements. All shifts are circular. Elements shifted off one end are wrapped around, appearing at the opposite end of the vector or array.

Occasionally, image files are saved with array elements offset. The SHIFT function allows you to easily correct such images assuming you know the amounts of the vertical and horizontal offsets. In the following example, the x-axis of original image is offset by a quarter of the image width, and the y-axis is offset by a third of the height.

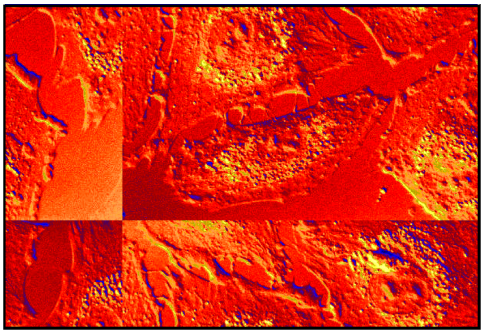


*Figure 4-8: Example of Misaligned Image Array Elements*

Using the SHIFT syntax, `Result = SHIFT(Array, S1, ..., Sn)`, we will enter negative values for the *S* (dimension) amounts in order to correct the image offset. For code that can be pasted into the Editor window, see "Example Code: Using Shift to Correct an Image" on page 193.

1. Select the image file and read it into memory:

```
file = FILEPATH('shifted_endocell.png', $
   SUBDIRECTORY = ['examples','data'])
image = READ_PNG(file, R, G, B)
```

2. Prepare the display device and load the image's associated color table:

```
DEVICE, DECOMPOSED = 0, RETAIN = 2
TVLCT, R, G, B
```

3. Get the size of the image, prepare a window based upon the values returned by the SIZE function, and display the image to be corrected:

```
imageSize = SIZE(image, /DIMENSIONS)
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original Image'
TV, image
```

4. Use SHIFT to correct the original image. Move the elements along the x-axis to the left, using a quarter of the array width as the x-dimension values. Move the y-axis elements, using one third of the array height as the number of elements to be shifted. By entering negative values for the amount the image dimensions are to be shifted, the array elements move toward the x and y axes.

```
image = SHIFT(image, -(imageSize[0]/4), -(imageSize[1]/3))
```

5. Display the corrected image in a second window:

```
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE='Shifted Image'
TV, image
```

The following figure displays the corrected image.

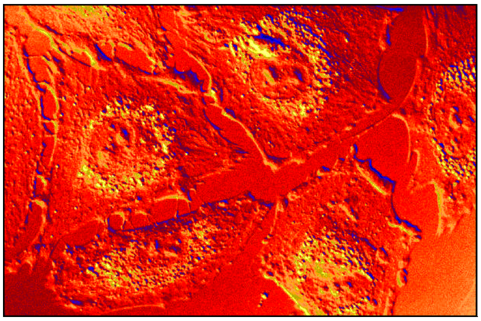


*Figure 4-9: Resulting Shifted Array*

## Example Code: Using Shift to Correct an Image

Copy and paste the following text into the IDLDE Editor window. After saving the file as `ShiftImageOffset.pro`, compile and run the program to reproduce the previous example.

```
PRO ShiftImageOffset

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

; Prepare the display device and load the
; color translation tables.
DEVICE, DECOMPOSED = 0, RETAIN = 2
TVLCT, R, G, B
HELP, image

; Get the image size.
imageSize = SIZE(image, /DIMENSIONS)

; Prepare the display window.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original Image'

; Display the original image.
TV, image

; Shift the original image to correct for the misalignment.
image = SHIFT(image, -imageSize[0]/4, -imageSize[1]/3)

; Display the shifted image.
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Shifted Image'
TV, image

END
```

## Reversing Images

The REVERSE function allows you to reverse any dimension of an array. This allows you to quickly change the viewing orientation of an image (flipping it horizontally or vertically).

Note that in the REVERSE syntax,

```
Result = REVERSE(Array [, Subscript_Index][,/OVERWRITE])
```

`Subscript_Index` specifies the dimension number beginning with 1, not 0 as with some other functions.

The following example demonstrates reversing the x-axis values (dimension 1) and the y-axis values (dimension 2) of an image of a knee. For code that can be copied into and IDL Editor window, see "Example Code: Reversing Images" on page 196.

1. Select the DICOM image of the knee and get the image's dimensions:

```
image = READ_DICOM (FILEPATH('mr_knee.dcm', $
   SUBDIRECTORY = ['examples', 'data']))
imgSize = SIZE (image, /DIMENSIONS)
```

2. Prepare the display device and load the gray scale color table:

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

3. Use the REVERSE function to reverse the x-axis values (`flipHorzImg`) and y-axis values (`flipVertImg`):

```
flipHorzImg = REVERSE(image, 1)
flipVertImg = REVERSE(image, 2)
```

4. Create an output window that is 2 times the size of the x-dimension of the image and 2 times the size of the y-dimension of the image:

```
WINDOW, 0, XSIZE = 2*imgSize[0], YSIZE = 2*imgSize[1], $
    TITLE = 'Original (Top) & Flipped Images (Bottom)'
```

5. Display the images, controlling their placement in the graphics window by using the *Position* argument to the TV command:

```
TV, image, 0
TV, flipHorzImg, 2
TV, flipVertImg, 3
```

Your output should appear similar to the following figure.

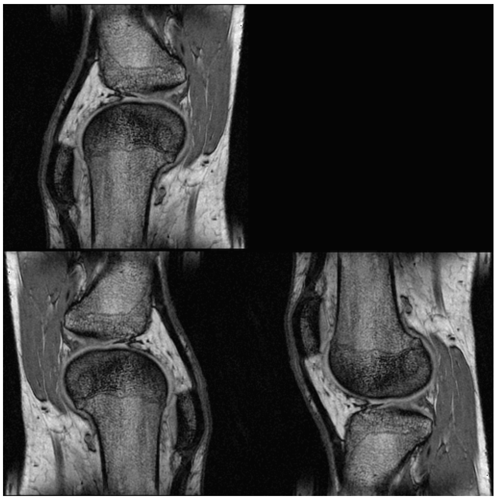


*Figure 4-10: Original Image (Top); Reversed Dimension 1 (Bottom Left); and Reversed Dimension 2 (Bottom Right)*

## Example Code: Reversing Images

Copy and paste the following text into the IDL Editor window. After saving the file as `ReverseImage.pro`, compile and run it to reproduce the REVERSE function example.

```
PRO ReverseImage

; Select the file and get the image dimensions.
image = READ_DICOM (FILEPATH('mr_knee.dcm', $
   SUBDIRECTORY = ['examples', 'data']))
imgSize = SIZE (image, /DIMENSIONS)

;Prepare the display device and load a color table.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 0

; Reverse dimension 1 to flip the image horizontally.
flipHorzImg = REVERSE(image, 1)

; Reverse dimesion 2 to flip the image vertically.
flipVertImg = REVERSE(image, 2)

; Prepare the window and display the original image.
WINDOW, 0, XSIZE = 2*imgSize[0], YSIZE = 2*imgSize[1], $
    TITLE = 'Original (Top) & Flipped Images (Bottom)'
TV, image, 0

; Display the reversed images.
TV, flipHorzImg, 2
TV, flipVertImg, 3

END
```

## Transposing Images

Transposing an image array interchanges array dimensions, reflecting an image about a diagonal (for example, reflecting a square image about a 45 degree line). By default, the TRANSPOSE function reverses the order of the dimensions. However, you can control how the dimensions are altered by specifying the optional vector, *P*, in the following statement:

```
Result = TRANSPOSE(Array[,P])
```

The values for *P* start at zero and correspond to the dimensions of the array. The following example transposes a photomicrograph of smooth muscle cells. For code that can be copied into the IDL Editor window, see “Example Code: Transposing an Image” on page 199.

1. Open the file and prepare to display it with a color table:

```
READ_JPEG, FILEPATH('muscle.jpg', $
   SUBDIRECTORY=['examples', 'data']), image
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 0
```

2. Display the original image:

```
WINDOW, 0, XSIZE = 652, YSIZE = 444, TITLE = 'Original Image'
TV, image
```

3. Reduce the image size for display purposes:

```
smallImg = CONGRID(image, 183, 111)
```

4. Using the TRANSPOSE function, reverse the array dimensions. This essentially flips the image across its main diagonal axis, moving the upper left corner of the image to the lower right corner.

```
transposeImg1 = TRANSPOSE(smallImg)
WINDOW, 1, XSIZE = 600, YSIZE = 183, TITLE = 'Transposed
Images'
TV, transposeImg1, 0
```

5. Specifying the reversal of the array dimensions leads to the same result since this is the default behavior of the TRANSPOSE function.

```
transposeImg2 = TRANSPOSE(smallImg, [1,0])
TV, transposeImg2, 2
```

6. However, specifying the original arrangement of the array dimensions results in no image transposition.

```
transposeImg3 = TRANSPOSE(smallImg, [0,1])
TV, transposeImg3, 2
```

The following figure displays the original image (top) and the results of the various TRANSPOSE statements (bottom).

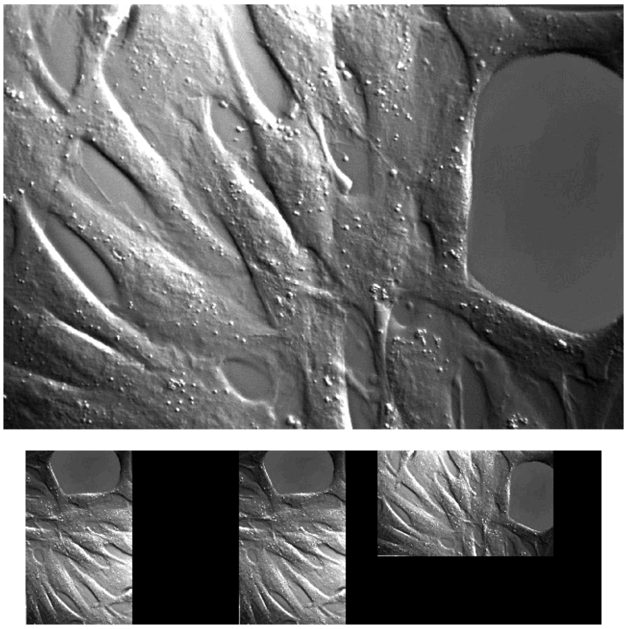


*Figure 4-11: Original (Top) and Transposed Images (Bottom) from Left to Right, transposeImg1, transposeImg2, and transposeImg3*

## Example Code: Transposing an Image

Copy and paste the following text into the Editor window. After saving the file as `TransposeImage.pro`, compile and run the program to reproduce the TRANSPOSE function example.

```
PRO TransposeImage

; Open the file and prepare to display it with a color table.
READ_JPEG, FILEPATH('muscle.jpg', $
   SUBDIRECTORY=['examples', 'data']), image
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Display the original image.
WINDOW, 0, XSIZE = 652, YSIZE = 444, TITLE = 'Original Image'
TV, image

; Reduce the image size for display purposes.
smallImg = CONGRID(image, 183, 111)

; Flip the image across its main diagonal axis,
; placing the upper left corner in the lower right corner.
transposeImg1 = TRANSPOSE(smallImg)

; Specifying the reversal of array dimensions leads
; to the same result.
transposeImg2 = TRANSPOSE(smallImg, [1,0])

; Specifying the original array arrangement results in
; no transposition.
transposeImg3 = TRANSPOSE(smallImg, [0,1])

; Display the transposed images.
WINDOW, 1, XSIZE= 600, YSIZE=183, TITLE='Transposed Images'
TV, transposeImg1, 0
TV, transposeImg2, 2
TV, transposeImg3, 2

END
```

## Rotating Images

To change the orientation of an image in IDL, use either the ROTATE or the ROT function. The ROTATE function changes the orientation of an image by 90 degree increments and/or transposes the array. The ROT function rotates an image by any amount and offers additional resizing options. For more information, see "Using the ROT Function for Arbitrary Rotations" on page 203.

### Rotating an Image by 90 Degree Increments

The following example changes the orientation of an image by rotating it 270°. For code that you can copy and paste into the Editor window, see "Example Code: Using ROTATE" on page 202.

1. Select the file and read in the data, specifying known data dimensions:

```
file = FILEPATH('galaxy.dat', $
   SUBDIRECTORY=['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = [256, 256])
```

2. Prepare the display device, load a color table, create a window, and display the image:

```
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 4
WINDOW, 0, XSIZE = 256, YSIZE = 256
TVSCL, image
```

3. Using the ROTATE syntax, *Result* = ROTATE (*Array*, *Direction*), rotate the galaxy image 270° counterclockwise by setting the *Direction* argument equal to 3. See "ROTATE Direction Argument Options" on page 201 for more information.

```
rotateImg = ROTATE(image, 3)
```

4. Display the rotated image.

```
Window, 1, XSIZE = 256, YSIZE = 256,
TVSCL, rotateImg
```

The following figure displays the original (left) and the rotated image (right).

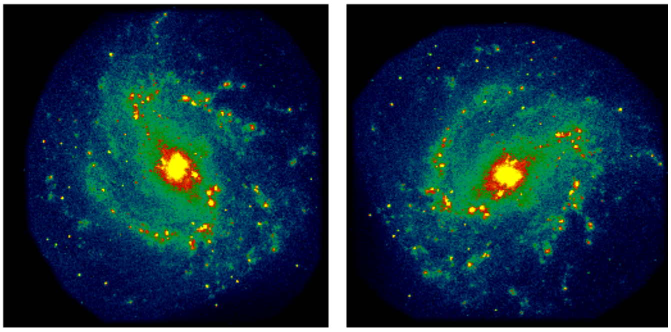


*Figure 4-12: Using ROTATE to Alter Image Orientation*

## ROTATE *Direction* Argument Options

The following table describes the *Direction* options available with the ROTATE function syntax, *Result* = ROTATE (*Array*, *Direction*).

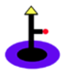
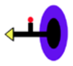
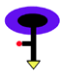
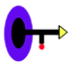
| Direction | Transpose? | Rotation Counterclockwise | Sample Image |
|---|---|---|---|
| 0 | No | None |  |
| 1 | No | 90° |  |
| 2 | No | 180° |  |
| 3 | No | 270° |  |

*Table 4-2: Direction Options Available with ROTATE*

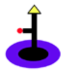
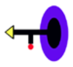
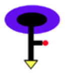
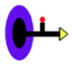
| Direction | Transpose? | Rotation Counterclockwise | Sample Image |
|---|---|---|---|
| 4 | Yes | None |  |
| 5 | Yes | 90° |  |
| 6 | Yes | 180° |  |
| 7 | Yes | 270° |  |

*Table 4-2: Direction Options Available with ROTATE*

## Example Code: Using ROTATE

Copy and paste the following text into the IDL Editor window. After saving the file as `RotateImage.pro`, compile and run the program to reproduce the previous example.

```
PRO RotateImage

; Select the file and read in the data using known dimensions.
file = FILEPATH('galaxy.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = [256, 256])

; Prepare the display device and load a color table.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 4

; Create a window and display the original image.
WINDOW, 0, XSIZE = 256, YSIZE = 256
TVSCL, image

; Rotate the galaxy 270 degrees counterclockwise.
rotateImg = ROTATE(image, 3)

; Display the rotated image in a new window.
WINDOW, 1, XSIZE = 256, YSIZE = 256
```

```
TVSCL, rotateImg

END
```

## Using the ROT Function for Arbitrary Rotations

The ROT function supports *clockwise* rotation of an image by any specified amount (not limited to 90 degree increments). Keywords also provide a means of optionally magnifying the image, selecting the pivot point around which the image rotates, and using either bilinear or cubic interpolation. If you wish to rotate an image only by 90 degree increments, ROTATE produces faster results.

The following example opens a image of a whirlpool galaxy, rotates it 33° clockwise and shrinks it to 50% of its original size. To copy and paste this example into an IDL Editor window, see "Example Code: Image Rotation Using the ROT Function" on page 205.

1. Select the file and read in the data, specifying known data dimensions:

   ```
   file = FILEPATH('m51.dat', SUBDIRECTORY = ['examples',
   'data'])
   image = READ_BINARY(file, DATA_DIMS = [340, 440])
   ```

2. Prepare the display device and load a black and white color table:

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

3. Create a window and display the original image:

   ```
   WINDOW, 0, XSIZE = 340, YSIZE = 440
   TVSCL, image
   ```

4. Using the ROT function syntax,

   *Result*=ROT(*A*, *Angle*, [*Mag*, $X_0$, $Y_0$] [,/INTERP]
   [,CUBIC=*value*{-1 to 0}] [, MISSING=*value*] [,/PIVOT])

   enter the following line to rotate the image 33°, shrink it to 50% of its original size, and fill the image display with a neutral gray color where there are no original pixel values:

   ```
   arbitraryImg = ROT(image, 33, .5, /INTERP, MISSING = 127)
   ```

5. Display the rotated image in a new window by entering the following two lines:

   ```
   WINDOW, 1, XSIZE = 340, YSIZE = 440
   TVSCL, arbitraryImg
   ```

Your output should appear similar to the following figure.

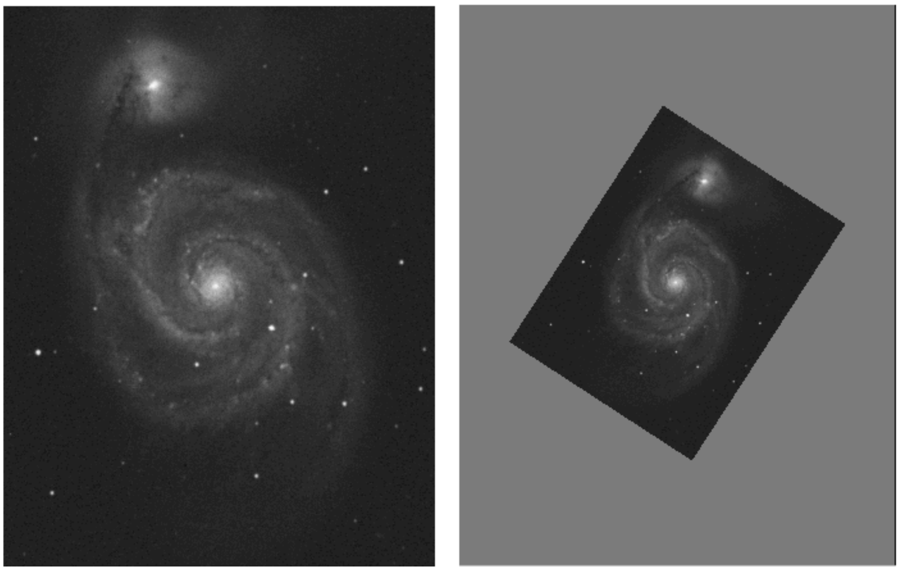


*Figure 4-13: The Original Image (Left) and Modified Image (Right)*

The `MISSING` keyword maintains the original image's boundaries, keeping the interpolation from extending beyond the original image size. Replacing `MISSING = 127` with `MISSING = 0` in the previous example creates a black background by using the default pixel color value of 0. Removing the `MISSING` keyword from the same statement allows the image interpolation to extend beyond the image's original boundaries.

## Example Code: Image Rotation Using the ROT Function

Copy and paste the following text into the IDL Editor window. After saving the file as `ArbitraryRotation.pro`, compile and run the program to reproduce the previous example.

```
PRO ArbitraryRotation

; Select the file and read in the data using known
; dimensions.
file = FILEPATH('m51.dat', SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = [340, 440])

; Prepare the display device and load a black and white
; color table.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 0

;Create a window and display the original image.
WINDOW, 0, XSIZE = 340, YSIZE = 440
TVSCL, image

; Rotate the new image 33 degrees clockwise and shrink
; it by 50 % and keep the interpolation from extending
; beyond the array boundaries.
arbitraryImg = ROT(image, 33, .5, /INTERP, MISSING = 127)

; Display the rotated image.
WINDOW, 1, XSIZE = 340, YSIZE = 440
TVSCL, arbitraryImg

END
```

# Planar Slicing of Volumetric Data

Volumetric displays are composed of a series of 2D slices of data which are layered to produce the volume. IDL provides routines that allow you to display a series of the 2D slices in a single image window, display single orthogonal or non-orthogonal slices of volumetric data, or interactively extract slices from a 3D volume. For more information, see the following sections:

- "Displaying a Series of Planar Slices" in the following section
- "Extracting a Slice of Volumetric Data" on page 209
- "Interactive Planar Slicing of Volumetric Data" on page 211

## Displaying a Series of Planar Slices

The following example displays 57 Magnetic Resonance Imaging (MRI) slices of a human head within a single window as well as a single slice which is perpendicular to the MRI data. For code that you can copy and paste into an IDL Editor window, see "Example Code: Displaying a Series of Planar Slices" on page 208.

1. Select the file and read in the data, specifying known data dimensions:

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

2. Load a color table to more easily distinguish between data values and prepare the display device:

   ```
   LOADCT, 5
   DEVICE, DECOMPOSED = 0, RETAIN = 2
   ```

3. Create the display window. When displaying all 57 slices of the array in a single window, the image size (80 by 100) and the number of slices (57) determine the window size. In this case, 10 columns and 6 rows will contain all 57 slices of the volumetric data.

   ```
   WINDOW, 0, XSIZE = 800, YSIZE = 600
   ```

4. Use the variable *`i`* in the following FOR statement to incrementally display each image in the array. The *`i`* also functions to control the positioning which, by default, uses the upper left corner as the starting point. Use `255b -` *`array`* to display the images using the inverse of the selected color table and the `ORDER` keyword to draw each image from the top down instead of the bottom up.

   ```
   FOR i = 0, 56,1 DO TVSCL, 255b - image [*,*,i], /ORDER, i
   ```

5. To extract a central slice from the y, z plane, which is perpendicular to the x, y plane of the MRI scans, specify 40 for the x-dimension value. Use REFORM to decrease the number of array dimensions so that TV can display the image:

   ```
   sliceImg = REFORM(image[40,*,*])
   ```

   This results in a 100 by 57 array.

6. Use CONGRID to compensate for the sampling rate of the scan slices:

   ```
   sliceImg = CONGRID(sliceImg, 100, 100)
   ```

7. Display the slice in the 47th window position:

   ```
   TVSCL, 255b - sliceImg, 47
   ```

   Since the image size is now 100 x 100 pixels, the 47th position in the 800 by 600 window is the final position.

Your output should be similar to the following figure.

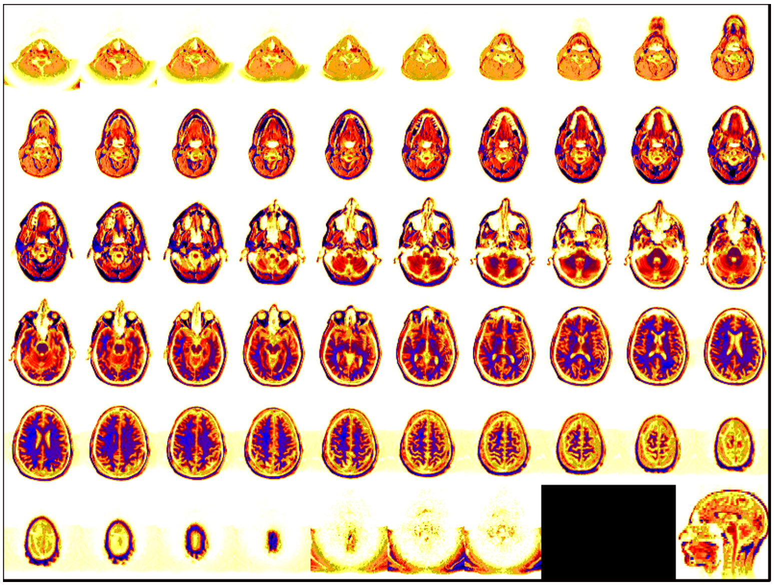


*Figure 4-14: Planar Slices of a MRI Scan of a Human Head*

**Note**

This method of extracting slices of data is limited to orthogonal slices only. You can extract single orthogonal and non-orthogonal slices of volumetric data using EXTRACT_SLICE, described in the following section. See "Extracting a Slice of Volumetric Data" on page 209 for more information.

## Example Code: Displaying a Series of Planar Slices

Copy and paste the following text into the Editor window. Save the file as `DisplaySlices.pro`, compile it and run it to reproduce the previous example.

```
PRO DisplaySlices

; Select the file, create an array and read in the data.
file = FILEPATH('head.dat', SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = [80, 100, 57])

; Load a color table and prepare the display window.
LOADCT,5
DEVICE, DECOMPOSED = 0, RETAIN = 2
```

```
WINDOW, 0, XSIZE = 800, YSIZE = 600

; Initialize the FOR statement. Use i as the loop element
; for the slice and the position. Use "255b -" to display
; the images with the inverse of the selected color table
; and use /ORDER to draw the image from the top down.
FOR i = 0, 56, 1 DO TVSCL, 255b - image [*,*,i], /ORDER, i

; Now extract a single perpendicular slice of data.
sliceImg = REFORM(image[40,*,*])

; Compensate for the sampling rate of the scan slices
; and display the image.
sliceImg = CONGRID(sliceImg, 100, 100)
TVSCL, 255b - sliceImg, 47

END
```

## Extracting a Slice of Volumetric Data

The EXTRACT_SLICE function extracts a single two-dimensional planar slice of data from a three-dimensional volume. By setting arguments that specify the orientation of the slice and a point in its center using the following syntax, you can precisely control the orientation of the slicing plane.

```
Result = EXTRACT_SLICE( Vol, Xsize, Ysize, Xcenter, Ycenter,
Zcenter, Xrot, Yrot, Zrot [, ANISOTROPY=[xspacing, yspacing,
zspacing]] [, OUT_VAL=value] [, /RADIANS] [, /SAMPLE]
[, VERTICES=variable] )
```

The following example demonstrates how to use EXTRACT_SLICE to extract the same singular slice as that shown in the previous example. Complete the following steps or see “Example Code: Extracting a Slice of Volumetric Data” on page 211 for an example you can copy and paste into the Editor window.

1. Select the file and read in the data, specifying known data dimensions:

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

2. Prepare the display device and load the grayscale color table.

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

3. Enter the following line to extract a sagittal planar slice from the MRI volume of the head.

```
sliceImg = EXTRACT_SLICE $
   (volume, 110, 110, 40, 50, 28, 90.0, 90.0, 0.0, OUT_VAL =
0)
```

**Note**

The code within the previous parentheses specifies: the volume (`Data`), a size greater than the *Xsize* and *Ysize* of the volume (`110,110`), the *Xcenter*, *Ycenter* and *Zcenter* (`40, 50, 28`) denoting the *x*, *y*, and *z* index points through which the slice will pass, the degree of *x*, *y*, and *z* rotation of the slicing plane (`90.0, 90.0, 0.0`) and the `OUT_VAL = 0` indicating that elements of the output array which fall outside the original values will be given the value of 0 or black.

4. Use CONGRID to resize the output array to an easily viewable size. This is also used to compensate for the sampling rate of the scan images.

```
bigImg = CONGRID (sliceImg, 400, 650, /INTERP)
```

5. Prepare a display window based on the resized array and display the image.

```
WINDOW, 0, XSIZE = 400, YSIZE = 650
TVSCL, bigImg
```

The image created by this example should appear similar to the following figure.

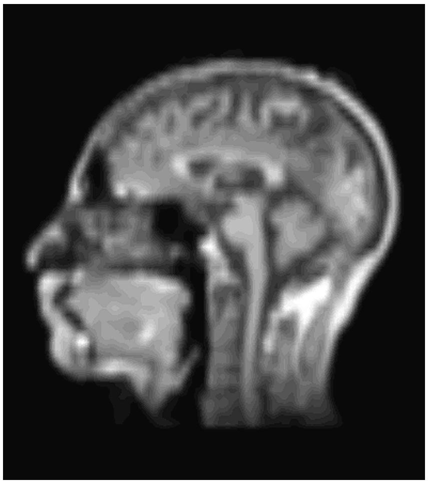


*Figure 4-15: Example of Extracting a Slice of Data From a Volume*

### Example Code: Extracting a Slice of Volumetric Data

Copy and paste the following text into an IDL Editor window. After saving the file as `ExtractSlice.pro`, compile and run the program to reproduce the previous example.

```
PRO ExtractSlice

; Select the file and define the image array.
file = FILEPATH('head.dat', SUBDIRECTORY = ['examples', 'data'])
volume = READ_BINARY(file, DATA_DIMS =[80, 100, 57])

; Prepare the display device and load a color table.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 0

; Extract a slice from the volume.
sliceImg = EXTRACT_SLICE(volume, 110, 110, 40, 50, 28, $
    90.0, 90.0, 0.0, OUT_VAL = 0)

; Enlarge the array.
bigImg = CONGRID(sliceImg, 400, 650, /INTERP)

; Display the image.
WINDOW, 0, XSIZE = 400, YSIZE = 650
TVSCL, bigImg

END
```

## Interactive Planar Slicing of Volumetric Data

The series of two-dimensional images created by the magnetic resonance imaging scan, shown in the section, "Displaying a Series of Planar Slices" on page 206, can also be visualized as a three-dimensional volume using either of IDL's interactive volume visualization tools, SLICER3 or XVOLUME.

SLICER3 quickly creates visualizations of 3D data using IDL Direct Graphics. The XVOLUME procedure employs IDL Object Graphics to create highly interactive visualizations that take advantage of OpenGL hardware acceleration and multiple processors for volume rendering. Since Object Graphics are rendered in memory and not simply drawn, both the time and amount of virtual memory required to create a XVOLUME visualization exceed those needed to create a Direct Graphics, SLICER3 visualization. For more information about XVOLUME, see "Displaying Volumes Using XVOLUME" on page 216.

## Displaying Volumetric Data Using SLICER3

The Direct Graphics SLICER3 widget-based application allows you to view single or multiple slices of a volume or to create an isosurface of the three-dimensional data. Complete the following steps to load the `head.dat` volume into the SLICER3 application or see “Example Code: Displaying Volumetric Data Using SLICER3” on page 215 for an example you can copy and paste into an IDL Editor window.

1. Select the data file and read in the data using known dimensions:

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

2. To display all slices of the `head.dat` file as a volume in SLICER3, create a pointer called `pdata` which passes the `data` array information to the SLICER3 application.

```
pData = PTR_NEW(volume)
```

**Note**

You can load multiple arrays into the SLICER3 application by creating a pointer for each array. Each array must have the same dimensions.

3. Load the data into the SLICER3 application. The DATA_NAMES designates the data set in the application’s **Data** list. This field will be greyed out if only one volumetric array has been loaded.

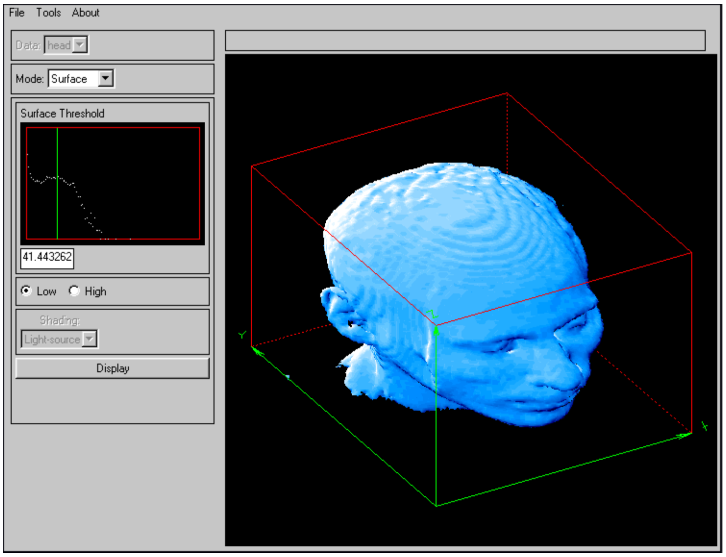
```
SLICER3, pData, DATA_NAMES ='head'
```

At first it is not apparent that your data has been passed to the SLICER3 application. See the following section, “Manipulating Volumetric Data Using SLICER3” for details on how to use this interface.

## Manipulating Volumetric Data Using SLICER3

Once you have loaded a three-dimensional array into the SLICER3 application, the interface offers numerous ways to visualize the data. The following steps cover creating an isosurface, viewing a slice of data within the volume and rotating the display.

1. In the SLICER3 application, select **Surface** from the **Mode:** list. Left-click in the Surface Threshold window containing the logarithmic histogram plot of the data and drag the green line to change the threshold value of the display. A value in the low to mid 40’s works well for this image. Click **Display** to view the isosurface of the data.



Mode: Surface

41.443262

(● Low ○ High)

*Figure 4-16: An Isosurface of Volumetric Data*

**Note**

To undo an action resulting in an unwanted image in the SLICER3 window, you can either choose **Tools** → **Delete** and select the last item on the list to undo the last action or choose **Tools** → **Erase** to erase the entire image.

2. Select **Slice** from the **Mode** list. Select the **Expose**, **Orthogonal**, and **X** options. Left-click in the image window and drag the mouse halfway along the X axis and then release the mouse button. The planar slice of volumetric data appears at the point where you release the mouse button.

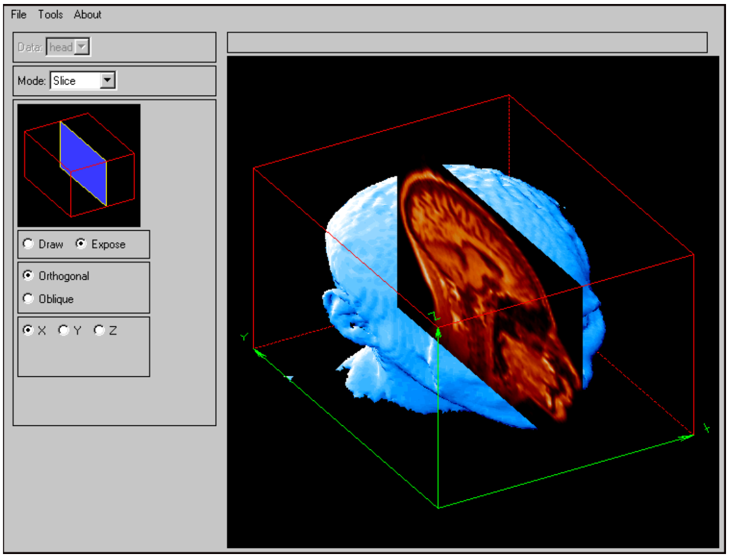


*Figure 4-17: Visualizing a Slice of Volumetric Data*

3. Change the colors used to display the slice by selecting **Tools** → **Colors** → **Slice/Block**. In the color table widget, select **STD Gamma-II** from the list and click **Done** to load the new color table.

4. Change the view of the display by selecting **View** from the **Mode** list. Here you can change the rotation and zoom factors of the displayed image. Use the slider bars to rotate the orientation cube. A preview of the cube's orientation appears in the small window above the controls. To create the orientation shown in the following figure, move the slider to a rotation of -18 for Z and -80 for X. Click **Display** to change the orientation of the image in the window.

The following figure displays the final image.

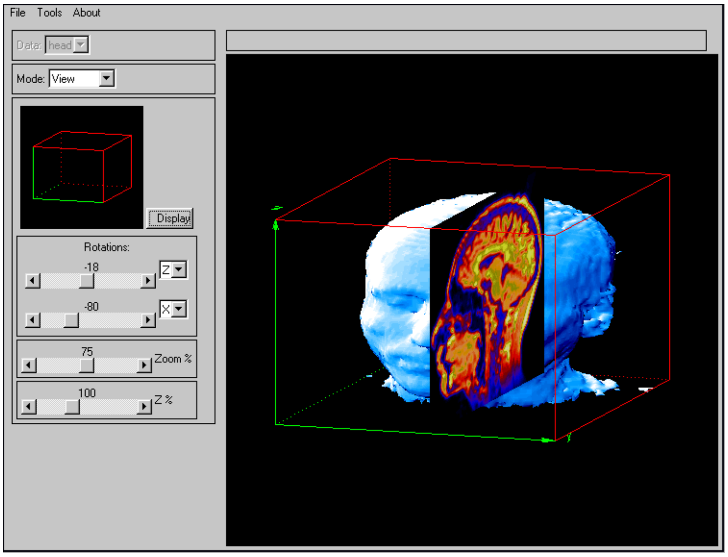


*Figure 4-18: A Slice Overlaying an Isosurface*

To save the image currently in the display window, select **File** → **Save** → **Save TIFF Image**. For more information about using the SLICER3 interface to manipulate volumetric data, see "SLICER3" in the *IDL Reference Guide*.

**Note**

Enter the following line after closing the SLICER3 application to release memory used by the pointer: `PTR_FREE, pData`

## Example Code: Displaying Volumetric Data Using SLICER3

The following code can be copied and pasted into the IDL Editor window to quickly pass the volumetric data contained in the `head.dat` file to the SLICER3 application. Paste the following text into an IDL Editor window and save the program as `DisplaySLICER3.pro` before compiling and running the program.

```
PRO DisplaySLICER3

; Select the file and define the array.
file = FILEPATH('head.dat', SUBDIRECTORY=['examples', 'data'])
volume = READ_BINARY(file, DATA_DIMS = [80, 100, 57])
```

```
; Create a pointer to the image data passed to SLICER3.
pData = PTR_NEW(volume)

; Load the data into the SLICER3 application.
SLICER3, pData, DATA_NAMES = 'head', /MODAL

; Release memory used by the pointer.
PTR_FREE, pData

END
```

**Note**

After running this program to load the data into the SLICER3 application, see "Manipulating Volumetric Data Using SLICER3" on page 213 for tips on using the interface.

## Displaying Volumes Using XVOLUME

Unlike SLICER3, the IDL Object Graphics procedure, XVOLUME, allows you to interactively manipulate 3D volumes and isosurfaces. While the following example requires more processing time to display the same data (`head.dat`) as that previously displayed with SLICER3, remember that the output is not the same. The XVOLUME example is rendering an opaque volume of the data set whereas the previous SLICER3 example simply displayed an isosurface. Although Object Graphics display methods can require more processing time, they also offer significant advantages including greater interactivity, true volume rendering with the ability to specify opacities, and finer control over image and volumetric data.

Complete the following steps to load the `head.dat` volume into the XVOLUME application or see "Example Code: Displaying Volumetric Data Using XVOLUME" on page 219 for an example you can copy and paste into an IDL Editor window.

1. Select the file and read in the data, specifying known data dimensions:

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

2. Reduce the size of the original array to speed up processing:

   ```
   smallVol = CONGRID(volume, 40, 50, 27)
   ```

3. Using the INTERPOLATE keyword to smooth the data, display the volume using the XVOLUME procedure:

   ```
   XVOLUME, smallVol, /INTERPOLATE
   ```

After the data is passed to the XVOLUME application, an image similar to the following figure appears.

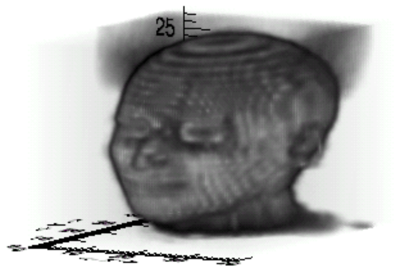


*Figure 4-19: Visualizing a Volume with XVOLUME*

## Manipulating Volumetric Data Using XVOLUME

Once data has been loaded into the XVOLUME application, you can create color coded contoured slices of data. Complete the following steps to create x-, y- and z-dimensional contours of the `head.dat` volume.

1. Rotate the image of the head so that the nose is facing toward the right. Click in the display window and, with your mouse button depressed, drag the mouse cursor to reposition the image display.
2. Select the **X**, **Y**, and **Z** “Contours” options, located on the upper-left portion of the XVOLUME interface.

**Note**

Turning off the XVOLUME “Auto-Render” feature produces faster responses to processing requests.

3. Move the **X Plane** slider to a value of 22. A contour line appears in the display window, running down the center of the image of the head. When you click in the display window, the planar slice is visible.
4. Move the **Y Plane** slider to a value of 27. A contour line appears along the middle of the y-dimension.

5. Move the **Z Plane** slider to a value of 12. Another contour line appears near the middle of the z-dimension.

The XVOLUME interface should appear similar to the following figure.

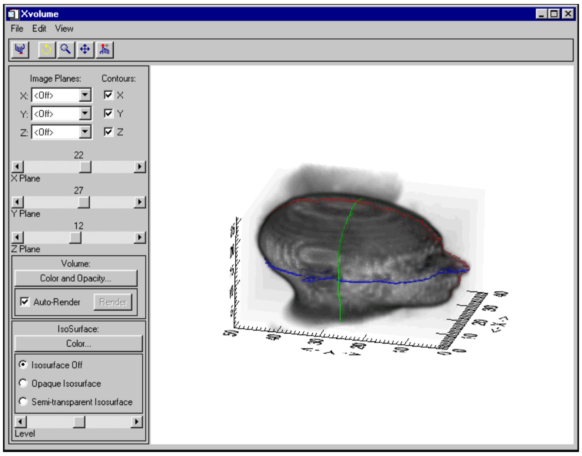


Image Planes: X: <Off> Y: <Off> Z: <Off>
Contours: ☑ X ☑ Y ☑ Z
X Plane 22
Y Plane 27
Z Plane 12
☑ Auto-Render
● Isosurface Off
○ Opaque Isosurface
○ Semi-transparent Isosurface

Figure 4-20: Creating Dimensional Contours Using XVOLUME

Click in the image display window to show the contour lines.

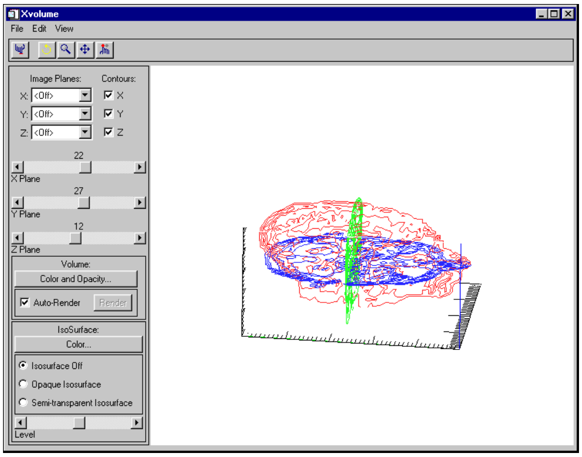


Image Planes: X: <Off> Y: <Off> Z: <Off>
Contours: ☑ X ☑ Y ☑ Z
X Plane 22
Y Plane 27
Z Plane 12
☑ Auto-Render
● Isosurface Off
○ Opaque Isosurface
○ Semi-transparent Isosurface

Figure 4-21: Displaying Contours of Planar Slices Using XVOLUME

**Tip**

After a volumetric array has been loaded into the XVOLUME application, it can be animated using the XVOLUME_ROTATE procedure. To rotate the image above, run the example program for "XVOLUME_ROTATE" in the *IDL Reference Guide*.

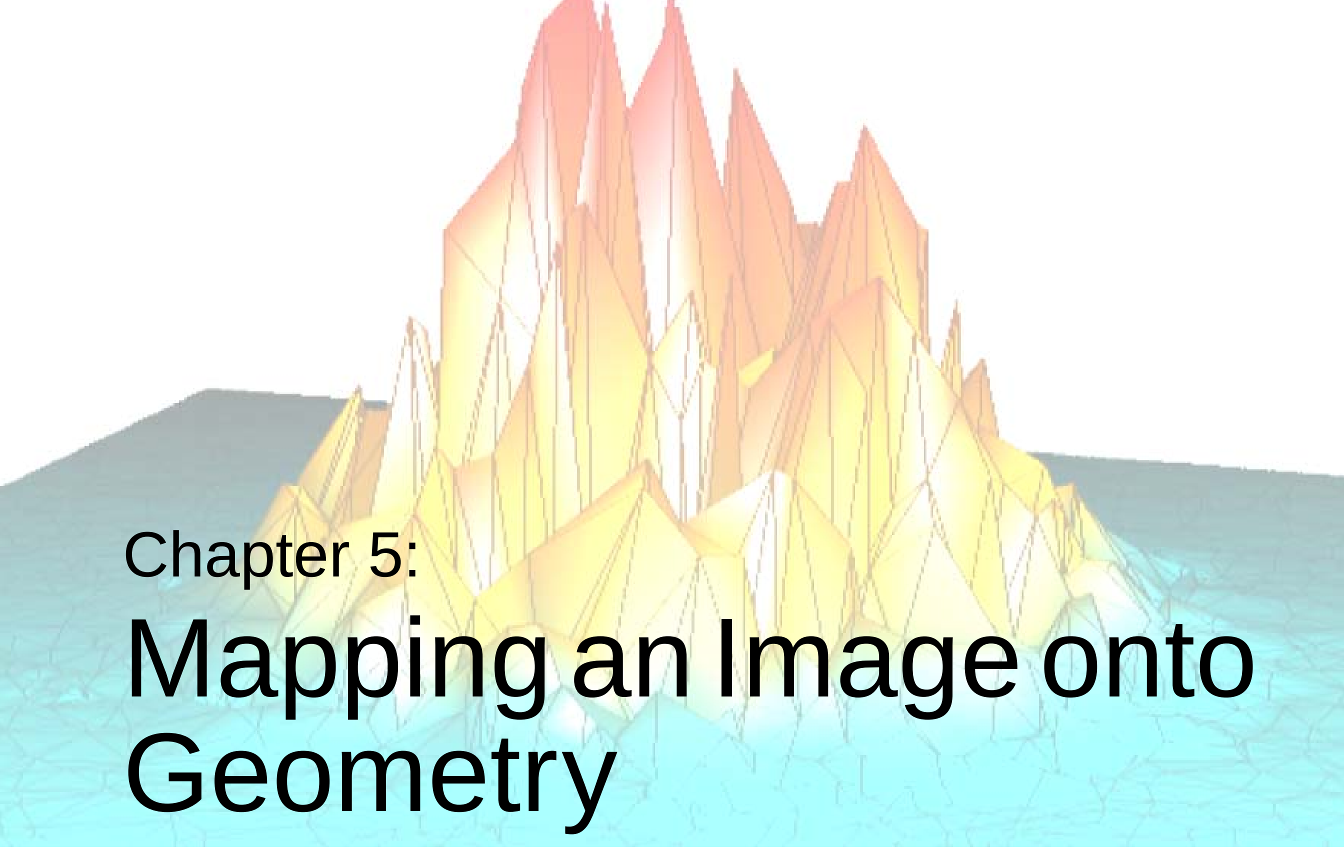
## Example Code: Displaying Volumetric Data Using XVOLUME

The following code can be copied and pasted into the IDL Editor window to quickly pass the volumetric data contained in the `head.dat` file to the XVOLUME application. After saving the file as `DisplayXVOLUME.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayXVOLUME

; Select the file and read in the data using known dimensions.
file = FILEPATH('head.dat', SUBDIRECTORY = ['examples', 'data'])
volume = READ_BINARY(file, DATA_DIMS = [80, 100, 57])

; Decrease the size of the array to speed up processing.
smallVol = CONGRID(volume, 40, 50, 27)

; Display the data using XVOLUME.
XVOLUME, smallVol, /INTERPOLATE

END
```

**Tip**

For information about manipulating data in the XVOLUME interface, see "Manipulating Volumetric Data Using XVOLUME" on page 217.



# Chapter 5: Mapping an Image onto Geometry

This chapter describes the following topics:

## Overview of Mapping Images onto Geometric Surfaces

Mapping an image onto geometry, also known as texture mapping, involves overlaying an image or function onto a geometric surface. Images may be realistic, such as satellite images, or representational, such as color-coded functions of temperature or elevation. Unlike volume visualizations, which render each voxel (volume element) of a three-dimensional scene, mapping an image onto geometry efficiently creates the appearance of complexity by simply layering an image onto a surface. The resulting realism of the display also provides information that is not as readily apparent as with a simple display of either the image or the geometric surface.

Mapping an image onto a geometric surface is a two step process. First, the image is mapped onto the geometric surface in *object space*. Second, the surface undergoes view transformations (relating to the viewpoint of the observer) and is then displayed in 2D *screen space*. You can use IDL Direct Graphics or Object Graphics to display images mapped onto geometric surfaces.

The following table introduces the tasks and routines covered in this chapter.

| Task | Routine(s)/Object(s) | Description |
|---|---|---|
| "Mapping an Image onto Elevation Data" on page 224. | SHADE_SURF | Display the elevation data. |
| | IDLgrWindow::Init<br>IDLgrView::Init<br>IDLgrModel::Init | Initialize the objects necessary for an Object Graphics display. |
| | IDLgrSurface::Init | Initialize a surface object containing the elevation data. |
| | IDLgrImage::Init | Initialize an image object containing the satellite image. |
| | XOBJVIEW | Display the object in an interactive IDL utility allowing rotation and resizing. |

*Table 5-1: Tasks and Routines Associated with Mapping an Image onto Geometry*

| Task | Routine(s)/Object(s) | Description |
|---|---|---|
| "Mapping an Image onto a Sphere Using Direct Graphics" on page 233. | MESH_OBJ<br>REPLICATE | Create a sphere. |
| | SCALE3 | Specify system variables required for 3D viewing. |
| | SET_SHADING | Control the light source used by POLYSHADE. |
| | TVSCL<br>POLYSHADE | Map the image onto the sphere using POLYSHADE and display the example with TVSCL. |
| "Mapping an Image onto a Sphere Using Object Graphics" on page 237. | MESH_OBJ<br>REPLICATE | Create a sphere. |
| | IDLgrModel::Init<br>IDLgrPalette::Init<br>IDLgrImage::Init | Initialize model, palette and image objects. |
| | FINDGEN<br>REPLICATE | Create normalized coordinates in order to map the image onto the sphere. |
| | IDLgrPolygon::Init | Assign the sphere to a polygon object and apply the image object. |
| | XOBJVIEW | Display the object in an interactive IDL utility allowing rotation and resizing. |

*Table 5-1: Tasks and Routines Associated with Mapping an Image onto Geometry (Continued)*

## Mapping an Image onto Elevation Data

The following Object Graphics example maps a satellite image from the Los Angeles, California vicinity onto a DEM (Digital Elevation Model) containing the area's topographical features. The realism resulting from mapping the image onto the corresponding elevation data provides a more informative view of the area's topography. The process is segmented into the following three sections:

- "Opening Image and Geometry Files", in the following section
- "Initializing the IDL Display Objects" on page 226
- "Displaying the Image and Geometric Surface Objects" on page 227

**Note**

Data can be either regularly gridded (defined by a 2D array) or irregularly gridded (defined by irregular *x*, *y*, *z* points). Both the image and elevation data used in this example are regularly gridded. If you are dealing with irregularly gridded data, use GRIDDATA to map the data to a regular grid.

See "Example Code: Mapping an Image onto a DEM" on page 231 for an example that you can copy and paste into an Editor window or complete the following steps for a detailed description of the process.

### Opening Image and Geometry Files

The following steps read in the satellite image and DEM files and display the elevation data.

1. Select the satellite image:

```
imageFile = FILEPATH('elev_t.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Import the JPEG file:

```
READ_JPEG, imageFile, image
```

3. Select the DEM file:

```
demFile = FILEPATH('elevbin.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

4. Define an array for the elevation data, open the file, read in the data and close the file:

   ```
   dem = READ_BINARY(demFile, DATA_DIMS = [64, 64])
   ```

5. Enlarge the size of the elevation array for display purposes:

   ```
   dem = CONGRID(dem, 128, 128, /INTERP)
   ```

6. To quickly visualize the elevation data before continuing on to the Object Graphics section, initialize the display, create a window and display the elevation data using the SHADE_SURF command:

   ```
   DEVICE, DECOMPOSED = 0
   WINDOW, 0, TITLE = 'Elevation Data'
   SHADE_SURF, dem
   ```

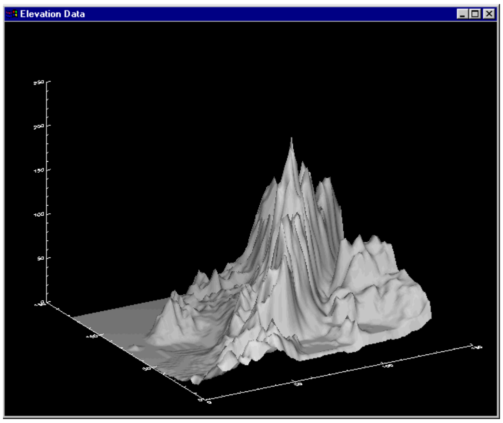


*Figure 5-1: Visual Display of the Elevation Data*

After reading in the satellite image and DEM data, continue with the next section to create the objects necessary to map the satellite image onto the elevation surface.

## Initializing the IDL Display Objects

After reading in the image and surface data in the previous steps, you will need to create objects containing the data. When creating an IDL Object Graphics display, it is necessary to create a window object (*oWindow*), a view object (*oView*) and a model object (*oModel*). These display objects, shown in the conceptual representation in the following figure, will contain a geometric surface object (the DEM data) and an image object (the satellite image). These user-defined objects are instances of existing IDL object classes and provide access to the properties and methods associated with each object class.

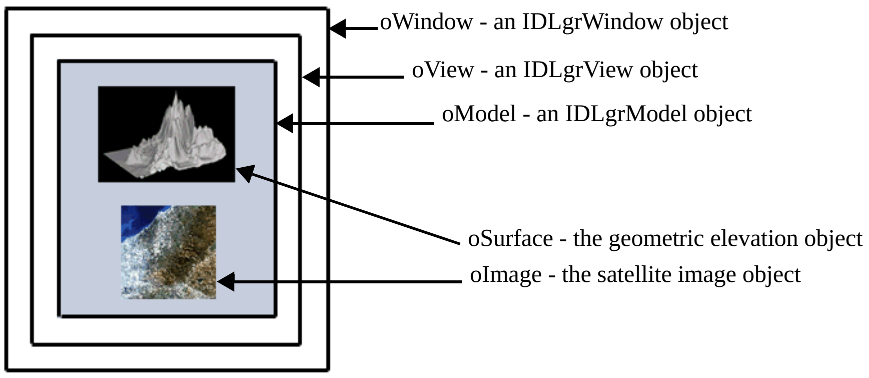

oWindow - an IDLgrWindow object
oView - an IDLgrView object
oModel - an IDLgrModel object
oSurface - the geometric elevation object
oImage - the satellite image object

*Figure 5-2: Conceptualization of Object Graphics Display Example*

**Note**

The XOBJVIEW utility (described in "Mapping an Image onto a Sphere Using Object Graphics" on page 237) automatically creates window and view objects.

Complete the following steps to initialize the necessary IDL objects.

1. Initialize the window, view and model display objects. For detailed syntax, arguments and keywords available with each object initialization, see IDLgrWindow::Init, IDLgrView::Init and IDLgrModel::Init. The following three lines use the basic syntax `oNewObject = OBJ_NEW('Class_Name')` to create these objects:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, COLOR_MODEL = 0)
oView = OBJ_NEW('IDLgrView')
oModel = OBJ_NEW('IDLgrModel')
```

2. Assign the elevation surface data, *dem*, to an IDLgrSurface object. The IDLgrSurface::Init keyword, `STYLE = 2`, draws the elevation data using a filled line style:

   ```
   oSurface = OBJ_NEW('IDLgrSurface', dem, STYLE = 2)
   ```

3. Assign the satellite image to a user-defined IDLgrImage object using IDLgrImage::Init:

   ```
   oImage = OBJ_NEW('IDLgrImage', image, INTERLEAVE = 0, $
      /INTERPOLATE)
   ```

   `INTERLEAVE = 0` indicates that the satellite image is organized using pixel interleaving, and therefore has the dimensions (3, *m*, *n*). The `INTERPOLATE` keyword forces bilinear interpolation instead of using the default nearest-neighbor interpolation method.

## Displaying the Image and Geometric Surface Objects

This section displays the objects created in the previous steps. The image and surface objects will first be displayed in an IDL Object Graphics window and then with the interactive XOBJVIEW utility.

1. Center the elevation surface object in the display window. The default object graphics coordinate system is [–1,–1], [1,1]. To center the object in the window, position the lower left corner of the surface data at [–0.5,–0.5, –0.5] for the *x*, *y* and *z* dimensions:

   ```
   oSurface -> GETPROPERTY, XRANGE = xr, YRANGE = yr, $
      ZRANGE = zr
   xs = NORM_COORD(xr)
   xs[0] = xs[0] - 0.5
   ys = NORM_COORD(yr)
   ys[0] = ys[0] - 0.5
   zs = NORM_COORD(zr)
   zs[0] = zs[0] - 0.5
   oSurface -> SETPROPERTY, XCOORD_CONV = xs, $
      YCOORD_CONV = ys, ZCOORD = zs
   ```

2. Map the satellite image onto the geometric elevation surface using the IDLgrSurface::Init TEXTURE_MAP keyword:

   ```
   oSurface -> SetProperty, TEXTURE_MAP = oImage, $
      COLOR = [255, 255, 255]
   ```

   For clearest display of the texture map, set COLOR = [255, 255, 255]. If the image does not have dimensions that are exact powers of 2, IDL resamples the image into a larger size that has dimensions which are the next powers of two greater than the original dimensions. This resampling may cause unwanted sampling artifacts. In this example, the image does have dimensions that are exact powers of two, so no resampling occurs.

**Note**

If your texture does not have dimensions that are exact powers of 2 and you do not want to introduce resampling artifacts, you can pad the texture with unused data to a power of two and tell IDL to map only a subset of the texture onto the surface.

For example, if your image is 40 by 40, create a 64 by 64 image and fill part of it with the image data:

```
textureImage = BYTARR(64, 64)
textureImage[0:39, 0:39] = image ; image is 40 by 40
oImage = OBJ_NEW('IDLgrImage', textureImage)
```

Then, construct texture coordinates that map the active part of the texture to a surface (oSurface):

```
textureCoords = [[], [], [], []]
oSurface -> SetProperty, TEXTURE_COORD = textureCoords
```

The surface object in IDL 5.6 is has been enhanced to automatically perform the above calculation. In the above example, just use the image data (the 40 by 40 array) to create the image texture and do not supply texture coordinates. IDL computes the appropriate texture coordinates to correctly use the 40 by 40 image.

**Note**

Some graphic devices have a limit for the maximum texture size. If your texture is larger than the maximum size, IDL scales it down into dimensions that work on the device. This rescaling may introduce resampling artifacts and loss of detail in the texture. To avoid this, use the TEXTURE_HIGHRES keyword to tell IDL to draw the surface in smaller pieces that can be texture mapped without loss of detail.

3. Add the surface object, covered by the satellite image, to the model object. Then add the model to the view object:

```
oModel -> Add, oSurface
oView -> Add, oModel
```

4. Rotate the model for better display in the object window. Without rotating the model, the surface is displayed at a 90° elevation angle, containing no depth information. The following lines rotate the model 90° away from the viewer along the *x*-axis and 30° clockwise along the *y*-axis and the *x*-axis:

```
oModel -> ROTATE, [1, 0, 0], -90
oModel -> ROTATE, [0, 1, 0], 30
oModel -> ROTATE, [1, 0, 0], 30
```

5. Display the result in the Object Graphics window:

```
oWindow -> Draw, oView
```

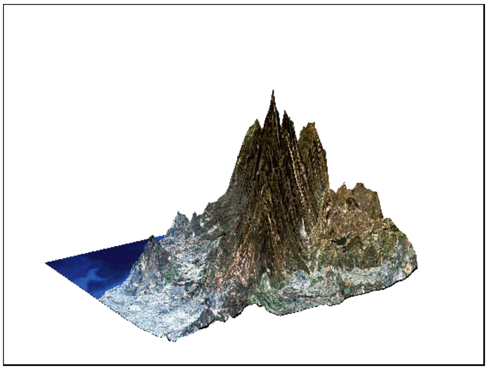


*Figure 5-3: Image Mapped onto a Surface in an Object Graphics Window*

6. Display the results using XOBJVIEW, setting the `SCALE = 1` (instead of the default value of 1/SQRT3) to increase the size of the initial display:

   ```
   XOBJVIEW, oModel, /BLOCK, SCALE = 1
   ```

   This results in the following display.

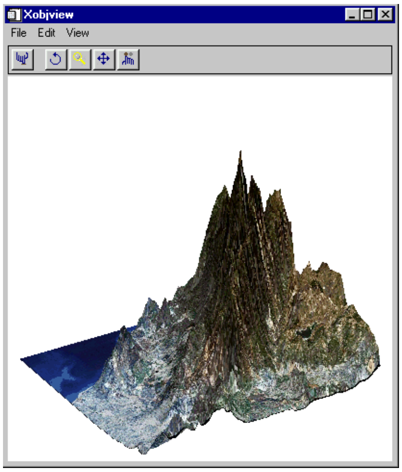


*Figure 5-4: Displaying the Image Mapped onto the Surface in XOBJVIEW*

   After displaying the model, you can rotate it by clicking in the application window and dragging your mouse. Select the magnify button, then click near the middle of the image. Drag your mouse away from the center of the display to magnify the image or toward the center of the display to shrink the image. Select the left-most button on the XOBJVIEW toolbar to reset the display.

7. Destroy unneeded object references after closing the display windows:

   ```
   OBJ_DESTROY, [oView, oImage]
   ```

   The *oModel* and *oSurface* objects are automatically destroyed when *oView* is destroyed.

For an example of mapping an image onto a regular surface using both Direct and Object Graphics displays, see "Mapping an Image onto a Sphere" on page 233.

## Example Code: Mapping an Image onto a DEM

Copy and paste the following text into the IDL Editor window. After saving the file as `Elevation_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO Elevation_Object

; Obtaining path to image file.
imageFile = FILEPATH('elev_t.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Importing image file.
READ_JPEG, imageFile, image

; Obtaining path to DEM data file.
demFile = FILEPATH('elevbin.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Importing data.
dem = READ_BINARY(demFile, DATA_DIMS = [64, 64])
dem = CONGRID(dem, 128, 128, /INTERP)

; Initialize the display.
DEVICE, DECOMPOSED = 0, RETAIN = 2

; Displaying original DEM elevation data.
WINDOW, 0, TITLE = 'Elevation Data'
SHADE_SURF, dem

; Initialize the  display objects.
oModel = OBJ_NEW('IDLgrModel')
oView = OBJ_NEW('IDLgrView')
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   COLOR_MODEL = 0)
oSurface = OBJ_NEW('IDLgrSurface', dem, STYLE = 2)
oImage = OBJ_NEW('IDLgrImage', image, $
   INTERLEAVE = 0, /INTERPOLATE)

; Calculating normalized conversion factors and
; shifting -.5 in every direction to center object
; in the window.
; Keep in mind that your view default coordinate
; system is [-1,-1], [1, 1]
oSurface -> GetProperty, XRANGE = xr, $
   YRANGE = yr, ZRANGE = zr
xs = NORM_COORD(xr)
xs[0] = xs[0] - 0.5
ys = NORM_COORD(yr)
```

```
ys[0] = ys[0] - 0.5
zs = NORM_COORD(zr)
zs[0] = zs[0] - 0.5
oSurface -> SetProperty, XCOORD_CONV = xs, $
   YCOORD_CONV = ys, ZCOORD = zs

; Applying image to surface (texture mapping).
oSurface -> SetProperty, TEXTURE_MAP = oImage, $
   COLOR = [255, 255, 255]

; Adding objects to model,then adding model to view.
oModel -> Add, oSurface
oView -> Add, oModel

; Rotating model for better display of surface
; in the object window.
oModel -> ROTATE, [1, 0, 0], -90
oModel -> ROTATE, [0, 1, 0], 30
oModel -> ROTATE, [1, 0, 0], 30

; Drawing the view of the surface (Displaying the
; results).
oWindow -> Draw, oView

; Displaying results in XOBJVIEW utility to allow
; rotation
XOBJVIEW, oModel, /BLOCK, SCALE = 1

; Destroying object references, which are no longer
; needed.
OBJ_DESTROY, [oView, oImage]

END
```

## Mapping an Image onto a Sphere

The following example maps an image containing a color representation of world elevation onto a sphere using both Direct and Object Graphics displays. The example is broken down into two sections:

- "Mapping an Image onto a Sphere Using Direct Graphics"
- "Mapping an Image onto a Sphere Using Object Graphics" on page 237

### Mapping an Image onto a Sphere Using Direct Graphics

See "Example Code: Mapping an Image onto a Sphere Using Direct Graphics" on page 236 for an example that you can copy and paste into an Editor window or complete the following steps for a detailed description of the process.

1. Select the file containing the world elevation image. Define the array, read in the data and close the file:

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

2. Prepare the display device to display a PseudoColor image:

```
DEVICE, DECOMPOSED = 0
```

3. Load a color table and using TVLCT, set the final index value of the red, green and blue bands to 255 (white). Setting these index values to white provides for the creation of a white window background in a later step.

```
LOADCT, 33
TVLCT, 255,255,255, !D.TABLE.SIZE - 1
```

   (For comparison, `TVLCT, 0, 0, 0, !D.TABLE_SIZE+1` would designate a black window background.)

4. Create a window and display the image containing the world elevation data:

```
WINDOW, 0, XSIZE = 360, YSIZE = 360
TVSCL, image
```

This image, shown in the following figure, will be mapped onto the sphere.

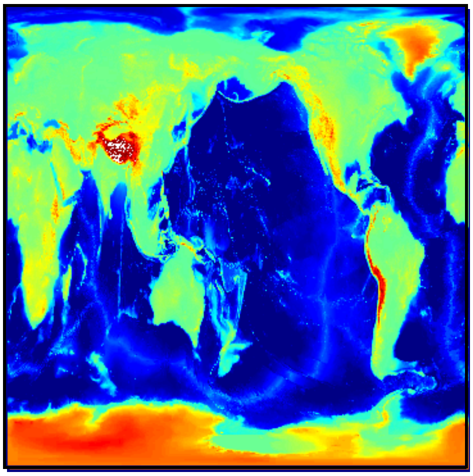


*Figure 5-5: World Elevation Image*

5. Use MESH_OBJ to create a sphere onto which the image will be mapped. The following line specifies a value of 4, indicating a spherical surface type:

   ```
   MESH_OBJ, 4, vertices, polygons, REPLICATE(0.25, 360, 360), $
      /CLOSED
   ```

   The *vertices* and *polygons* variables are the lists that contain the mesh vertices and mesh indices of the sphere. REPLICATE generates a 360 by 360 array, each element of which will contain the value 0.25. Using REPLICATE in the *Array1* argument of MESH_OBJ specifies that the *vertices* variable is to consist of 360 by 360 vertices, each positioned at a constant radius of 0.25 from the center of the sphere.

6. Create a window and define the 3D view. Use SCALE3 to designate transformation and scaling parameters for 3D viewing. The AX and AZ keywords specify the rotation, in degrees about the *x* and *z* axes:

   ```
   WINDOW, 1, XSIZE = 512, YSIZE = 512
   SCALE3, XRANGE = [-0.25,0.25], YRANGE = [-0.25,0.25], $
      ZRANGE = [-0.25,0.25], AX = 0, AZ = -90
   ```

7. Set the light source to control the shading used by the POLYSHADE function. Use SET_SHADING to modify the light source, moving it from the default position of [0,0,1] with rays parallel to the *z*-axis to a light source position of [-0.5, 0.5, 2.0]:

    ```
    SET_SHADING, LIGHT = [-0.5, 0.5, 2.0]
    ```

8. Set the system background color to the default color index, defining a white window background:

    ```
    !P.BACKGROUND = !P.COLOR
    ```

9. Use TVSCL to display the world elevation image mapped onto the sphere. POLYSHADE references the sphere created with the MESH_OBJ routine, sets `SHADES = image` to map the image onto the sphere and uses the image transformation defined by the T3D transformation matrix:

    ```
    TVSCL, POLYSHADE(vertices, polygons, SHADES = image, /T3D)
    ```

The specified view of the image mapped onto the sphere is displayed in a Direct Graphics window as shown in the following figure.

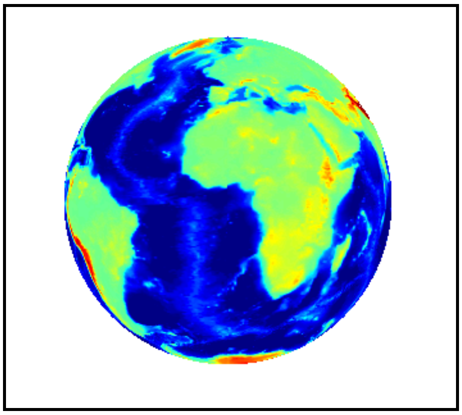


*Figure 5-6: Direct Graphics Display of an Image Mapped onto a Sphere*

10. After displaying the image, restore the system's default background color:

    ```
    !P.BACKGROUND = 0
    ```

To create a Object Graphics display featuring a sphere that can be interactively rotated and resized, complete the steps contained in the section, "Mapping an Image onto a Sphere Using Object Graphics" on page 237.

## Example Code: Mapping an Image onto a Sphere Using Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `MapOnSphere_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO MapOnSphere_Direct

; Importing image into IDL.
file = FILEPATH('worldelv.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = [360, 360])

; Initializing color table and setting the final
; index values to white.
DEVICE, DECOMPOSED = 0
LOADCT, 33
TVLCT, 255, 255, 255, !D.TABLE_SIZE - 1

; Displaying the original image.
WINDOW, 0, XSIZE = 360, YSIZE = 360
TVSCL, image

; Creating a 360x360 sphere with a constant radius of
; 0.25 to use as the data.
MESH_OBJ, 4, vertices, polygons, REPLICATE(0.25, 360, 360), $
   /CLOSED

; Creating the window defining the view.
WINDOW, 2, XSIZE = 512, YSIZE = 512
SCALE3, XRANGE = [-0.25,0.25], YRANGE = [-0.25,0.25], $
   ZRANGE = [-0.25,0.25], AX = 0, AZ = -90

; Displaying data with image as texture map.
SET_SHADING, LIGHT = [-0.5, 0.5, 2.0]
!P.BACKGROUND = !P.COLOR
TVSCL, POLYSHADE(vertices, polygons, SHADES = image, /T3D)
!P.BACKGROUND = 0

END
```

## Mapping an Image onto a Sphere Using Object Graphics

This example maps an image containing world elevation data onto the surface of a sphere and displays the result using the XOBJVIEW utility. This utility automatically creates the window object and the view object, previously shown in the section, "Initializing the IDL Display Objects" on page 226. Therefore, this example creates an object based on IDLgrModel that contains the sphere, the image and the image palette, as shown in the conceptual representation in the following figure.

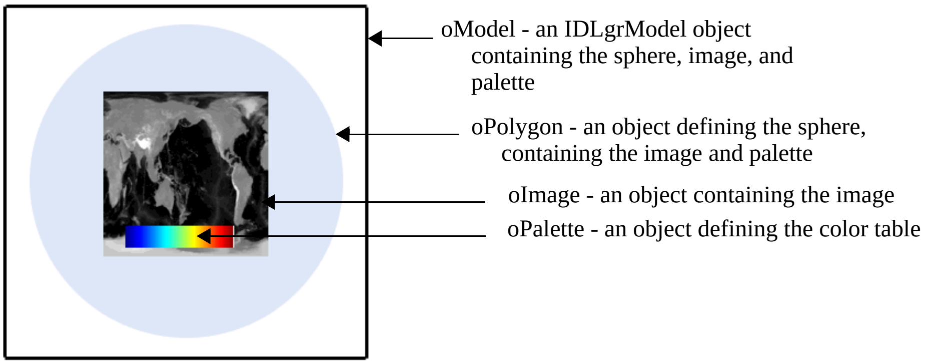

oModel - an IDLgrModel object containing the sphere, image, and palette
oPolygon - an object defining the sphere, containing the image and palette
oImage - an object containing the image
oPalette - an object defining the color table

*Figure 5-7: Conceptualization of XOBJVIEW Object Graphics Example*

See "Example Code: Mapping an Image onto a Sphere Using Object Graphics" on page 241 for an example that you can copy and paste into an Editor window or complete the following steps for a detailed description of the process.

**Note**

If you are continuing the exercise from the previous section, "Mapping an Image onto a Sphere Using Direct Graphics", skip steps 1, and 2. Proceed with step 3 to create the necessary objects.

1. Select the world elevation image. Define the array, read in the data and close the file.

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

2. Use the MESH_OBJ procedure to create a sphere onto which the image will be mapped. The following invocation of MESH_OBJ uses a value of 4, which represents a spherical mesh:

   ```
   MESH_OBJ, 4, vertices, polygons, REPLICATE(0.25, 101, 101)
   ```

   When the MESH_OBJ procedure completes, the *vertices* and *polygons* variables contain the mesh vertices and polygonal mesh connectivity information, respectively. Although our image is 360 by 360, we can texture map the image to a mesh that has fewer vertices. IDL interpolates the image data across the mesh, retaining all the image detail between polygon vertices. The number of mesh vertices determines how close to perfectly round the sphere will be. Fewer vertices produce a sphere with larger facets, while more vertices make a sphere with smaller facets and more closely approximates a perfect sphere. A large number of mesh vertices will increase the time required to draw the sphere. In this example, MESH_OBJ produces a 101 by 101 array of vertices that are located in a sphere shape with a radius of 0.25.

3. Initialize the display objects. In this example, it is necessary to define a model object that will contain the sphere, the image and the color table palette. Using the syntax, `*oNewObject* = OBJ_NEW('*Class_Name*')`, create the model, palette and image objects:

   ```
   oModel = OBJ_NEW('IDLgrModel')
   oPalette = OBJ_NEW('IDLgrPalette')
   oPalette -> LOADCT, 33
   oPalette -> SetRGB, 255, 255, 255, 255
   oImage = OBJ_NEW('IDLgrImage', image, PALETTE = oPalette)
   ```

   The previous lines initialize the *oPalette* object with the color table and then set the final index value of the red, green and blue bands to 255 (white) in order to use white (instead of black) to designate the highest areas of elevation. The palette object is created before the image object so that the palette can be applied when initializing the image object. For more information, see IDLgrModel::Init, IDLgrPalette::Init and IDLgrImage::Init.

4. Create texture coordinates that define how the texture map is applied to the mesh. A texture coordinate is associated with each vertex in the mesh. The value of the texture coordinate at a vertex determines what part of the texture will be mapped to the mesh at that vertex. Texture coordinates run from 0.0 to 1.0 across a texture, so a texture coordinate of [0.5, 0.5] at a vertex specifies that the image pixel at the exact center of the image is mapped to the mesh at that vertex.

In this example, we want to do a simple linear mapping of the texture around the sphere, so we create a convenience vector that describes the mapping in each of the texture's *x*- and *y*-directions, and then create these texture coordinates:

```
vector = FINDGEN(101)/100.
texure_coordinates = FLTARR(2, 101, 101)
texure_coordinates[0, *, *] = vector # REPLICATE(1., 101)
texure_coordinates[1, *, *] = REPLICATE(1., 101) # vector
```

The code above copies the convenience vector through the array in each direction.

5. Enter the following line to initialize a polygon object with the image and geometry data using the IDLgrPolygon::Init function. Set `SHADING = 1` for gouraud (smoother) shading. Set the `DATA` keyword equal to the sphere defined with the MESH_OBJ function. Set `COLOR` to draw a white sphere onto which the image will be mapped. Set `TEXTURE_COORD` equal to the texture coordinates created in the previous steps. Assign the image object to the polygon object using the `TEXTURE_MAP` keyword and force bilinear interpolation:

```
oPolygons = OBJ_NEW('IDLgrPolygon', SHADING = 1, $
   DATA = vertices, POLYGONS = polygons, $
   COLOR = [255, 255, 255], $
   TEXTURE_COORD = texure_coordinates, $
   TEXTURE_MAP = oImage, /TEXTURE_INTERP)
```

**Note**

When mapping an image onto an IDLgrPolygon object, you must specify both TEXTURE_MAP and TEXTURE_COORD keywords.

6. Add the polygon containing the image and the palette to the model object:

```
oModel -> ADD, oPolygons
```

7. Rotate the model -90° along the *x*-axis and *y*-axis:

```
oModel -> ROTATE, [1, 0, 0], -90
oModel -> ROTATE, [0, 1, 0], -90
```

8. Display the results using XOBJVIEW, an interactive utility allowing you to rotate and resize objects:

```
XOBJVIEW, oModel, /BLOCK
```

After displaying the object, you can rotate the sphere by clicking in the display window and dragging your mouse. Select the magnify button and click near the middle of the sphere. Drag your mouse away from the center of the display to magnify the image or toward the center of the display to shrink the image. Select the left-most button on the XOBJVIEW toolbar to reset the display. The following figure shows a rotated and magnified view of the world elevation object.

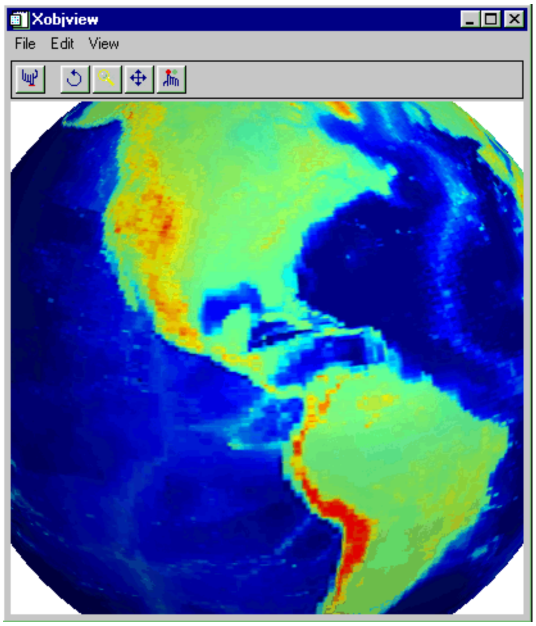


*Figure 5-8: Magnified View of World Elevation Object*

9. After closing the XOBJVIEW display, remove unneeded object references:

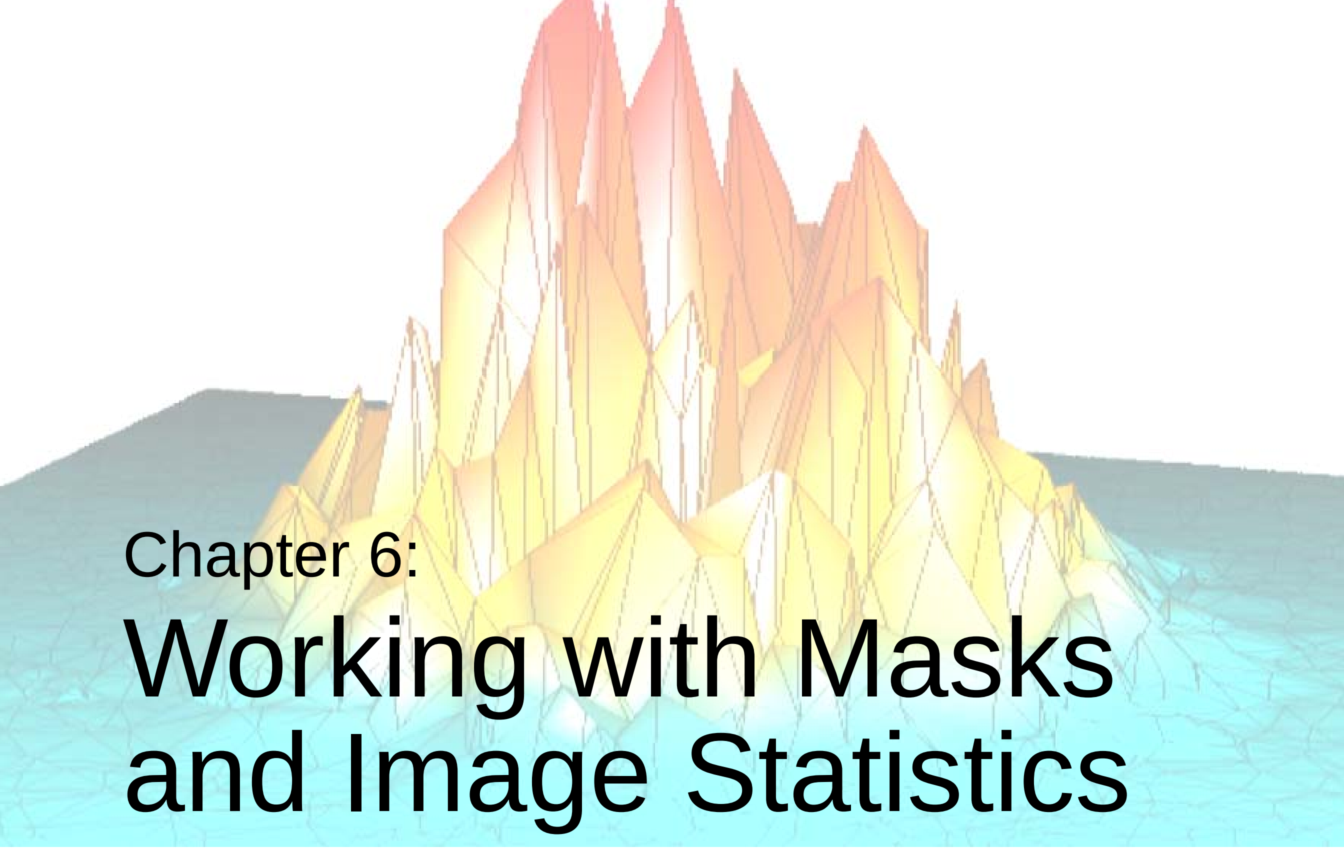
```
OBJ_DESTROY, [oModel, oImage, oPalette]
```

## Example Code: Mapping an Image onto a Sphere Using Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `MapOnSphere_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO MapOnSphere_Object

; Importing image into IDL.
file = FILEPATH('worldelv.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = [360, 360])

; Creating a 51x51 sphere with a constant radius of
; 0.25 to use as the data.
MESH_OBJ, 4, vertices, polygons, $
   REPLICATE(0.25, 101, 101)

; Creating a model object to contain the display.
oModel = OBJ_NEW('IDLgrModel')

; Creating image and palette objects to contain the
; imported image and color table.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LoadCT, 33
oPalette -> SetRGB, 255,255,255,255
oImage = OBJ_NEW('IDLgrImage', image, $
   PALETTE = oPalette)

; Deriving texture map coordinates.
vector = FINDGEN(101)/100.
texure_coordinates = FLTARR(2, 101, 101)
texure_coordinates[0, *, *] = vector # REPLICATE(1., 101)
texure_coordinates[1, *, *] = REPLICATE(1., 101) # vector

; Creating the polygon object containing the data.
oPolygons = OBJ_NEW('IDLgrPolygon', SHADING = 1, $
   DATA = vertices, POLYGONS = polygons, $
   COLOR = [255,255,255], $
   TEXTURE_COORD = texure_coordinates, $
   TEXTURE_MAP = oImage, /TEXTURE_INTERP)

; Adding polygon to model container.  NOTE:  the polygon
; object already contains the texture map image and its
; related palette.
oModel -> ADD, oPolygons
```

```
; Rotating model to display zero degrees latitude and
; zero degrees longitude as front.
oModel -> ROTATE, [1, 0, 0], -90
oModel -> ROTATE, [0, 1, 0], -90

; Displaying results.
XOBJVIEW, oModel, /BLOCK

; Cleaning up object references.
OBJ_DESTROY, [oModel, oImage, oPalette]

END
```



# Chapter 6: Working with Masks and Image Statistics

This chapter describes the following topics:

## Overview of Masks and Image Statistics

Mathematical operations used with images include logic (conditional) operations and statistics. Logic operations are used to make masks to apply threshold levels to clip the pixel values of an image, and to locate pixel values. These operations help to segment features in an image, after which statistics can be derived to provide a means of comparison.

Masks are used to isolate specific features. A mask is a binary image, made by using relational operators. A binary mask is multiplied by the original image to omit specific areas. For more information, see "Masking Images" on page 246.

Threshold levels can be applied to an image to clip the pixel values to a floor or a ceiling. Clipping enhances specific features, and is applied through minimum and maximum operators. After the resulting images are byte-scaled, the specific features remain while the other areas become part of the background. For more information, see "Clipping Images" on page 251.

Locating pixel values is another way to segment specific features. Mathematical expressions are used to determine the location of pixels with particular values within the two-dimensional array representing the image. For more information, see "Locating Pixel Values in an Image" on page 256.

When specific features have been segmented, image statistics (such as total, mean, standard deviation, and variance) can be derived to quantify and compare them. For more information, see "Calculating Image Statistics" on page 262.

**Note**

In this book, Direct Graphics examples are provided by default. Object Graphics examples are provided in cases where significantly different methods are required.

The following list introduces image math operations and associated IDL math operators and routines covered in this chapter.

| Task | Operator(s) and Routine(s) | Description |
|---|---|---|
| "Masking Images" on page 246. | Relational Operators<br>Mathematical Operators | Make masks and apply them to images. |

*Table 6-1: Image Math Tasks and Related Image Math Operators and Routines*

| Task | Operator(s) and Routine(s) | Description |
|---|---|---|
| "Clipping Images" on page 251. | Minimum and Maximum Operators<br>Mathematical Operators | Clip the pixel values of an image to highlight specific features. |
| "Locating Pixel Values in an Image" on page 256. | WHERE<br>Mathematical Operators | Locate specific pixel values within an image. |
| "Calculating Image Statistics" on page 262 | Mathematical Operators<br>IMAGE_STATISTICS | Calculate the sum, mean, standard deviation, and variance of the pixel values within an image. |

*Table 6-1: Image Math Tasks and Related Image Math Operators and Routines (Continued)*

**Note**

This chapter uses data files from the `IDL examples/data` and `examples/demo/demodata` directories. Two files, `data.txt` and `index.txt`, contain descriptions of the files, including array sizes.

## Masking Images

Masking (also known as thresholding) is used to isolate features within an image above, below, or equal to a specified pixel value. The value (known as the threshold level) determines how masking occurs. In IDL, masking is performed with the relational operators. IDL's relational operators are shown in the following table.

| Operator | Description |
|---|---|
| EQ | Equal to |
| NE | Not equal to |
| GE | Greater than or equal to |
| GT | Greater than |
| LE | Less than or equal to |
| LT | Less than |

*Table 6-2: IDL's Relational Operators*

For example, if you have an *image* variable and you want to mask it to include only the pixel values equaling 125, the resulting *mask* variable is created with the following IDL statement.

```
mask = image EQ 125
```

The mask level is applied to every element in the image array, which results in a binary image.

**Note**

You can also provide both upper and lower bounds to masks by using the bitwise operators; AND, NOT, OR, and XOR. See Bitwise Operators *in the Building IDL Applications* for more information on these operators.

The following example uses masks derived from the image contained in the `worldelv.dat` file, which is in the `examples/data` directory. Masks are derived to extract the oceans and land. These masks are applied back to the image to show only on the oceans or the land. Masks are applied by multiplying them with the original image.

For code that you can copy and paste into an Editor window, see "Example Code: Masking Images" on page 249 or complete the following steps for a detailed description of the process.

1. Determine the path to the file:

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

2. Initialize the image size parameter:

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

3. Import the image from the file:

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

4. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 38
```

5. Create a window and display the image:

```
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'World Elevation'
TV, image
```

The following figure shows the original image, which represents the elevation levels of the world.

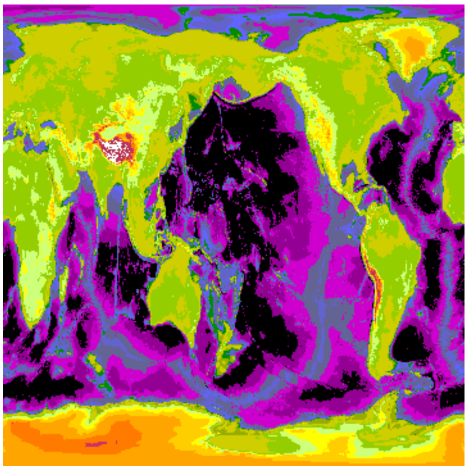


*Figure 6-1: World Elevation Image*

6. Make a mask of the oceans:

```
oceanMask = image LT 125
```

7. Multiply the ocean mask by the original image:

```
maskedImage = image*oceanMask
```

8. Create another window and display the mask and the results of the multiplication:

```
WINDOW, 1, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Oceans Mask (left) and Resulting Image (right)'
TVSCL, oceanMask, 0
TV, maskedImage, 1
```

The following figure shows the mask of the world's oceans and the results of applying it to the original image.

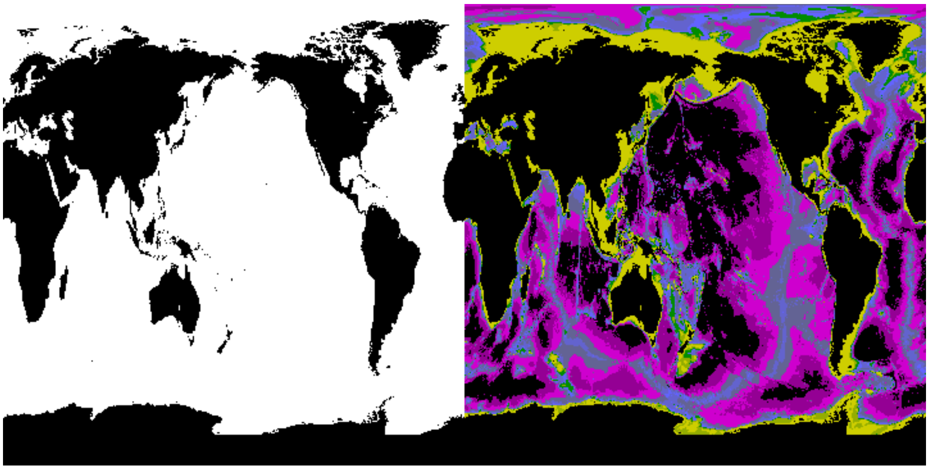


*Figure 6-2: Oceans Mask (left) and the Resulting Image (right)*

9. Make a mask of the land:

```
landMask = image GE 125
```

10. Multiply the land mask by the original image:

```
maskedImage = image*landMask
```

11. Create another window and display the mask and the results of the multiplication:

```
WINDOW, 2, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Land Mask (left) and Resulting Image (right)'
TVSCL, landMask, 0
TV, maskedImage, 1
```

The following figure shows the mask of the land masses of the world and the results of applying it to the original image.

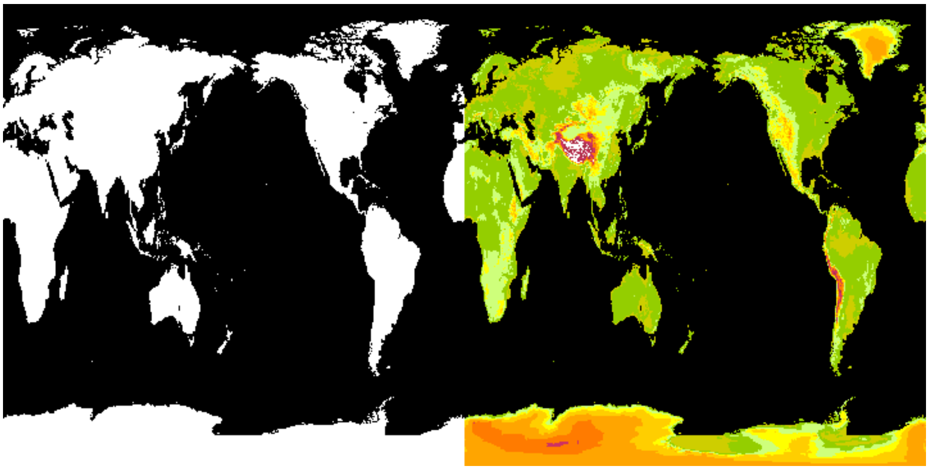


*Figure 6-3: Land Mask (left) and the Resulting Image (right)*

## Example Code: Masking Images

Copy and paste the following text into the IDL Editor window. After saving the file as `MaskingImages.pro`, compile and run the program to reproduce the previous example.

```
PRO MaskingImages

; Determine the path to the file.
file = FILEPATH('worldelv.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameters.
imageSize = [360, 360]

; Import the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

```
; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 38

; Create a window and display the image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'World Elevation'
TV, image

; Make a mask of the oceans.
oceanMask = image LT 125

; Multiply the ocean mask by the original image.
maskedImage = image*oceanMask

; Create another window and display the mask and the
; results of the multiplication.
WINDOW, 1, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Oceans Mask (left) and Resulting Image (right)'
TVSCL, oceanMask, 0
TV, maskedImage, 1

; Make a mask of the land.
landMask = image GE 125

; Multiply the land mask by the original image.
maskedImage = image*landMask

; Create another window and display the mask and the
; results of the multiplication.
WINDOW, 2, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Land Mask (left) and Resulting Image (right)'
TVSCL, landMask, 0
TV, maskedImage, 1

END
```

## Clipping Images

Clipping is used to enhance features within an image. You provide a threshold level to determine how the clipping occurs. The values above (or below) the threshold level remain the same while the other values are set equal to the level.

In IDL, clipping is performed with the minimum and maximum operators. IDL's minimum and maximum operators are shown in the following table.

| Operator | Description |
|---|---|
| < | Less than or equal to |
| > | Greater than or equal to |

*Table 6-3: IDL's Minimum and Maximum Operators*

The operators are used in an expression that contains an image array, the operator, and then the threshold level. For example, if you have an *image* variable and you want to scale it to include only the values greater than or equal to 125, the resulting *clippedImage* variable is created with the following IDL statement.

```
clippedImage = image > 125
```

The threshold level is applied to every element in the image array. If the element value is less than 125, it is set equal to 125. If the value is greater than or equal to 125, it is left unchanged.

**Note**

When clipping is combined with byte-scaling, this is equivalent to performing a stretch on an image. See "Determining Intensity Values When Thresholding and Stretching Images" in Chapter 11 for more information.

The following example shows how to threshold an image of Hurricane Gilbert, which is in the `hurric.dat` file in the `examples/data` directory. Two clipped images are created. One contains all data values greater than 125 and the other contains all values less than 125. Since these clipped images are grayscale images and do not use the entire 0 to 255 range, they are displayed with the TV procedure and then scaled with the TVSCL procedure, which scales the range of the image from 0 to 255.

For code that you can copy and paste into an Editor window, see "Example Code: Thresholding Images" on page 254 or complete the following steps for a detailed description of the process.

1. Determine the path to the `worldtmp.png` file:

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

2. Define the image size parameter:

   ```
   imageSize = [440, 340]
   ```

3. Import the image from the file:

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

4. Initialize the display:

   ```
   DEVICE, DECOMPOSED = 0
   LOADCT, 0
   ```

5. Create a window and display the image:

   ```
   WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'Hurricane Gilbert'
   TV, image
   ```

   The following figure shows the original image of Hurricane Gilbert.

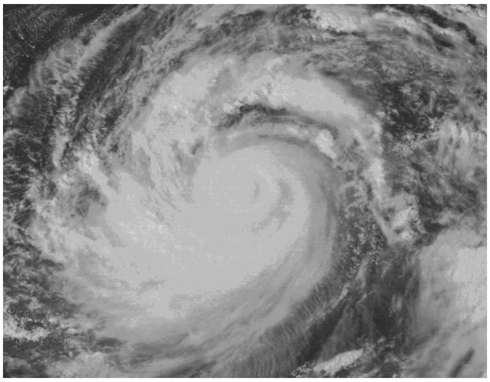


*Figure 6-4: Image of Hurricane Gilbert*

6. Clip the image to determine which pixel values are greater than 125:

   ```
   topClippedImage = image > 125
   ```

7. Create another window and display the clipped image with the TV (left) and the TVSCL (right) procedures:

```
WINDOW, 1, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Image Greater Than 125, TV (left) ' + $
   'and TVSCL (right)'
TV, topClippedImage, 0
TVSCL, topClippedImage, 1
```

   The following figure shows the resulting image of pixel values greater than 125 with the TV and TVSCL procedures.

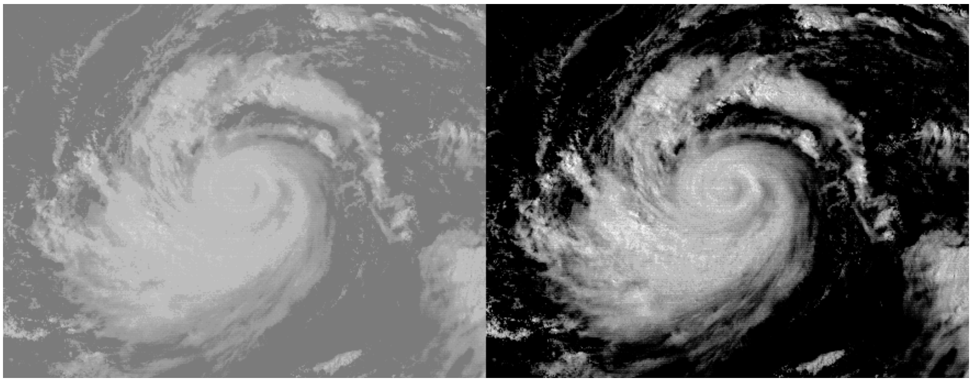


*Figure 6-5: Pixel Values Greater Than 125, TV (left) and TVSCL (right)*

8. Clip the image to determine which pixel values are less than a 125:

```
bottomClippedImage = image < 125
```

9. Create another window and display the clipped image with the TV and the TVSCL procedures:

```
WINDOW, 1, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Image Less Than 125, TV (left) ' + $
   'and TVSCL (right)'
TV, bottomClippedImage, 0
TVSCL, bottomClippedImage, 1
```

The following figure shows the resulting image of pixel values less than 125 with the TV (left) and TVSCL (right) procedures.

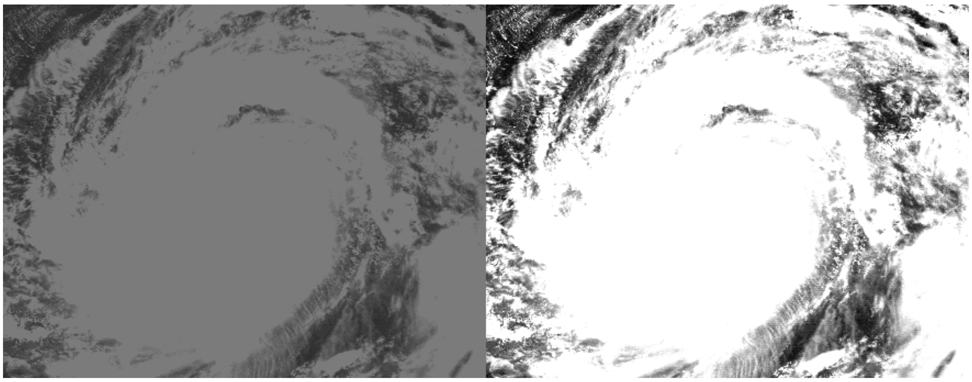


*Figure 6-6: Pixel Values Less Than 125, TV (left) and TVSCL (right)*

## Example Code: Thresholding Images

Copy and paste the following text into the IDL Editor window. After saving the file as `ClippingImages.pro`, compile and run the program to reproduce the previous example.

```
PRO ClippingImages

; Determine the path to the file.
file = FILEPATH('hurric.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Define the image size parameter.
imageSize = [440, 340]

; Import image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Hurricane Gilbert'
TV, image
```

```
; Threshold the image by determining which pixel values
; are greater than 125.
topThreshold = image > 125

; Create another window and display the threshold image
; with the TV (left) and the TVSCL (right) procedures.
WINDOW, 1, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Image Greater Than 125, TV (left) ' + $
   'and TVSCL (right)'
TV, topThreshold, 0
TVSCL, topThreshold, 1

; Threshold the image by determining which pixel values
; are less than 125.
bottomThreshold = image < 125

; Create another window and display the threshold image
; with the TV (left) and the TVSCL (right) procedures.
WINDOW, 2, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Image Less Than 125, TV (left) ' + $
   'and TVSCL (right)'
TV, bottomThreshold, 0
TVSCL, bottomThreshold, 1

END
```

## Locating Pixel Values in an Image

Locating pixel values within an image helps to segment features. You can use IDL's WHERE function to determine where features characterized by specific values appear within the image. The WHERE function returns a vector of one-dimensional indices, locating where the specified values occur within the image. The values are specified with an expression input argument to the WHERE function. The expression is defined with the relational operators, similar to how masking is performed. See "Masking Images" on page 246 for more information on relational operators.

Since the WHERE function only returns the one-dimensional indices, you must derive the column and row locations with the following statements.

```
column = index MOD imageSize[0]
row = index/imageSize[0]
```

where *index* is the result from the WHERE function and *imageSize[0]* is the width of the image.

The WHERE function returns one-dimensional indices to allow you to easily use these results as subscripts within the original image array or another array. This ability allows you to combine values from one image with another image. The following example combines specific values from the image within the `worldelv.dat` file with the image within the `worldtmp.png` file. The `worldelv.dat` file is in the `examples/data` directory and the `worldtmp.png` file is in the `examples/demo/demodata` directory. First, the temperature data is shown in the oceans and the elevation data is shown on the land. Then, the elevation data is shown in the oceans and the temperature data is shown on the land.

For code that you can copy and paste into an Editor window, see "Example Code: Locating Pixel Values in an Images" on page 260 or complete the following steps for a detailed description of the process.

1. Determine the path to the file:

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

2. Initialize the image size parameter:

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

3. Import the elevation image from the file:

   ```
   elvImage = READ_BINARY(file, DATA_DIMS = imageSize)
   ```

4. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 38
```

5. Create a window and display the elevation image:

```
WINDOW, 0, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'World Elevation (left) and Temperature (right)'
TV, elvImage, 0
```

6. Determine the path to the other file:

```
file = FILEPATH('worldtmp.png', $
   SUBDIRECTORY = ['examples', 'demo', 'demodata'])
```

7. Import the temperature image:

```
tmpImage = READ_PNG(file)
```

8. Display the temperature image:

```
TV, tmpImage, 1
```

The following figure shows the original world elevation and temperature images.

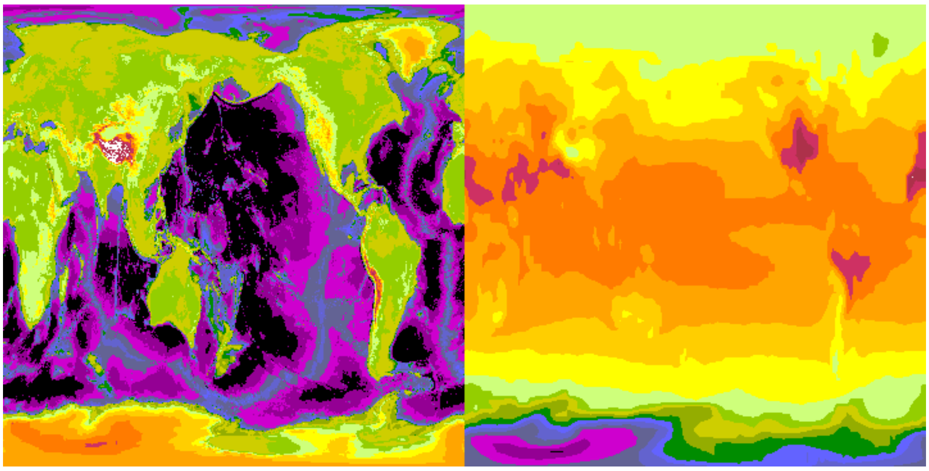


*Figure 6-7: World Elevation (left) and Temperature (right)*

9. Determine where the oceans are located within the elevation image:

```
ocean = WHERE(elvImage LT 125)
```

10. Set the temperature image as the background:

    ```
    image = tmpImage
    ```

11. Replace values from the temperature image with the values from the elevation image only where the ocean pixels are located:

    ```
    image[ocean] = elvImage[ocean]
    ```

12. Create another window and display the resulting temperature over land image:

    ```
    WINDOW, 1, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
       TITLE = 'Temperature Over Land (left) ' +
       'and Over Oceans (right)'
    TV, image, 0
    ```

13. Determine where the land is located within the elevation image:

    ```
    land = WHERE(elvImage GE 125)
    ```

14. Set the temperature image as the background:

    ```
    image = tmpImage
    ```

15. Replace values from the temperature image with the values from the elevation image only where the land pixels are located:

    ```
    image[land] = elvImage[land]
    ```

16. Display the resulting temperature over oceans image:

    ```
    TV, image, 1
    ```

The following figure shows two possible image combinations using the world elevation and temperature images.

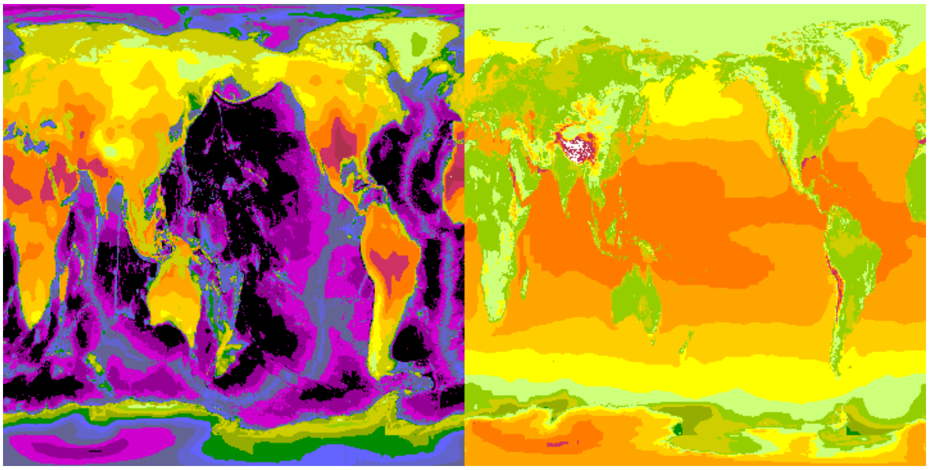


*Figure 6-8: Temperature Over Land (left) and Over Oceans (right)*

**Tip**

You could also construct the same image using masks and adding them together. For example, to create the second image (temperature over oceans), you could have done the following:

```
mask = elvImage GE 125
image = (tmpImage*(1 - mask)) + (elvImage*mask)
```

For large images, using masks may be faster than using the WHERE routine.

## Example Code: Locating Pixel Values in an Images

Copy and paste the following text into the IDL Editor window. After saving the file as `CombiningImages.pro`, compile and run the program to reproduce the previous example.

```
PRO CombiningImages

; Determine the path to the file.
file = FILEPATH('worldelv.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize image size parameter.
imageSize = [360, 360]

; Import the elevation image from the file.
elvImage = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 38

; Create a window and display the elevation image.
WINDOW, 0, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'World Elevation (left) and Temperature (right)'
TV, elvImage, 0

; Determine the path to the other file.
file = FILEPATH('worldtmp.png', $
   SUBDIRECTORY = ['examples', 'demo', 'demodata'])

; Import the temperature image from the other file.
tmpImage = READ_PNG(file)

; Display the temperature image.
TV, tmpImage, 1

; Determine where the oceans are located within the
; elevation image.
ocean = WHERE(elvImage LT 125)

; Set the temperature image as the background.
image = tmpImage

; Replace values from the temperature image with values
; from the elevation image only where the ocean pixels
; are located.
image[ocean] = elvImage[ocean]
```

```
; Create another window and display the resulting
; temperature over land image.
WINDOW, 1, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Temperature Over Land (left) ' + $
   'and Over Oceans (right)'
TV, image, 0

; Determine where the land is located within the
; elevation image.
land = WHERE(elvImage GE 125)

; Set the temperature image as the background.
image = tmpImage

; Replace values from the temperature image with values
; from the elevation image only where the land pixels
; are located.
image[land] = elvImage[land]

; Display the resulting temperature over oceans image.
TV, image, 1

END
```

## Calculating Image Statistics

The statistical properties of an image provide useful information, such as the total, mean, standard deviation, and variance of the pixel values. IDL's IMAGE_STATISTICS procedure can be used to calculate these statistical properties. The MOMENT, N_ELEMENTS, TOTAL, MAX, MEAN, MIN, STDDEV, and VARIANCE routines can also be used to calculate individual statistics, but most of these values are already provided by the IMAGE_STATISTICS procedure.

The following example shows how to use the IMAGE_STATISTICS procedure to calculate the statistical properties of an image. First, a mask is used to subtract the convection of the earth's core from the convection image contained in the `convec.dat` file, which is in the `examples/data` directory. The resulting difference represents the convection of just the earth's mantle. The IMAGE_STATISTICS procedure is applied to this difference image, and the resulting values are displayed in the Output Log. Then, a mask is derived for the non-zero values of the difference image, and the IMAGE_STATISTICS procedure is used again, this time with the mask applied through the MASK keyword. The resulting statistics can than be compared. The color table associated with this example is white for zero values and dark red for 255 values.

For code that you can copy and paste into an Editor window, see step 9, "Determine the statistics of the difference image:" on page 264 or complete the following steps for a detailed description of the process.

1. Determine the path to the file:

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

2. Initialize the image size parameter.

```
imageSize = [248, 248]
```

3. Import the image from the file:

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

4. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 27
```

5. Create a window and display the image:

   ```
   WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'Earth Mantle Convection'
   TV, image
   ```

   The following figure shows the original convection image.

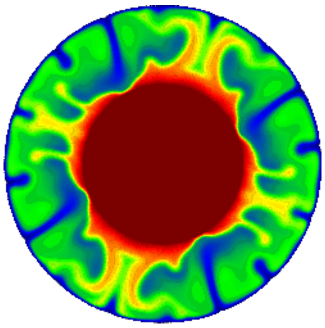


*Figure 6-9: Earth Mantle Convection*

6. Make a mask of the core and scale it to range from 0 to 255:

   ```
   core = BYTSCL(image EQ 255)
   ```

7. Subtract the scaled mask from the original image:

   ```
   difference = image - core
   ```

8. Create another window and display the difference of the original image and the scaled mask:

   ```
   WINDOW, 2, XSIZE = imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'Difference of Original & Core'
   TV, difference
   ```

The following figure shows the convection of just the earth’s mantle.

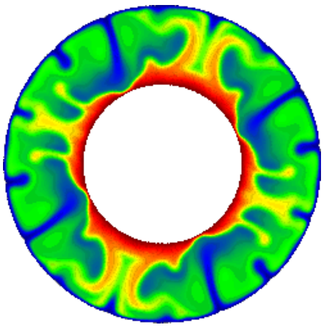


*Figure 6-10: The Difference of the Original Image and the Core*

9. Determine the statistics of the difference image:

```
IMAGE_STATISTICS, difference, COUNT = pixelNumber, $
   DATA_SUM = pixelTotal, MAXIMUM = pixelMax, $
   MEAN = pixelMean, MINIMUM = pixelMin, $
   STDDEV = pixelDeviation, $
   SUM_OF_SQUARES = pixelSquareSum, $
   VARIANCE = pixelVariance
```

10. Print out the resulting statistics:

```
PRINT, ''
PRINT, 'IMAGE STATISTICS:'
PRINT, 'Total Number of Pixels = ', pixelNumber
PRINT, 'Total of Pixel Values = ', pixelTotal
PRINT, 'Maximum Pixel Value = ', pixelMax
PRINT, 'Mean of Pixel Values = ', pixelMean
PRINT, 'Minimum Pixel Value = ', pixelMin
PRINT, 'Standard Deviation of Pixel Values = ', $
   pixelDeviation
PRINT, 'Total of Squared Pixel Values = ', $
   pixelSquareSum
PRINT, 'Variance of Pixel Values = ', pixelVariance
```

IDL prints:

```
IMAGE STATISTICS:
Total Number of Pixels = 61504
Total of Pixel Values = 2.61691e+006
Maximum Pixel Value = 253.000
Mean of Pixel Values = 42.5486
Minimum Pixel Value = 0.000000
Standard Deviation of Pixel Values = 48.7946
Total of Squared Pixel Values = 2.57779e+008
Variance of Pixel Values = 2380.91
```

11. Derive a mask of the non-zero values of the image:

```
nonzeroMask = difference NE 0
```

12. Determine the statistics of the image with the mask applied:

```
IMAGE_STATISTICS, difference, COUNT = pixelNumber, $
   DATA_SUM = pixelTotal, MASK = nonzeroMask, $
   MAXIMUM = pixelMax, MEAN = pixelMean, $
   MINIMUM = pixelMin, STDDEV = pixelDeviation, $
   SUM_OF_SQUARES = pixelSquareSum, $
   VARIANCE = pixelVariance
```

13. Print out the resulting statistics:

```
PRINT, ''
PRINT, 'MASKED IMAGE STATISTICS:'
PRINT, 'Total Number of Pixels = ', pixelNumber
PRINT, 'Total of Pixel Values = ', pixelTotal
PRINT, 'Maximum Pixel Value = ', pixelMax
PRINT, 'Mean of Pixel Values = ', pixelMean
PRINT, 'Minimum Pixel Value = ', pixelMin
PRINT, 'Standard Deviation of Pixel Values = ', $
   pixelDeviation
PRINT, 'Total of Squared Pixel Values = ', $
   pixelSquareSum
PRINT, 'Variance of Pixel Values = ', pixelVariance
```

IDL prints:

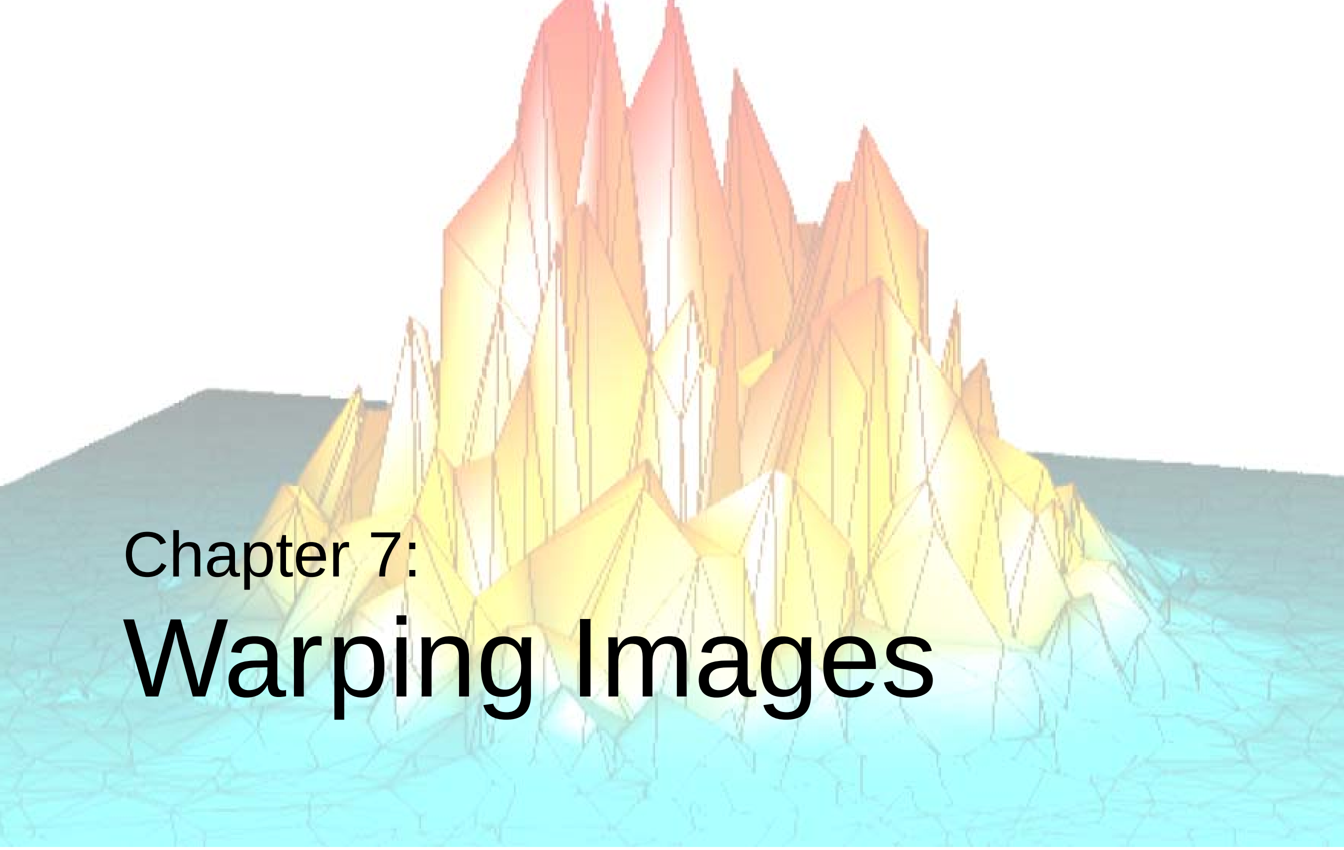
```
MASKED IMAGE STATISTICS:
Total Number of Pixels = 36325
Total of Pixel Values = 2.61691e+006
Maximum Pixel Value = 253.000
Mean of Pixel Values = 72.0416
Minimum Pixel Value = 1.00000
Standard Deviation of Pixel Values = 43.6638
Total of Squared Pixel Values = 2.57779e+008
Variance of Pixel Values = 1906.53
```

The difference in the resulting statistics are because of the zero values, which are a part of the calculations for the image before the mask is applied.

## Example Code: Calculating Image Statistics

Copy and paste the following text into the IDL Editor window. After saving the file as `CalculatingStatistics.pro`, compile and run the program to reproduce the previous example.

```
PRO CalculatingStatistics

; Determine the path to the file.
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [248, 248]

; Import the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 27

; Create a window and display the image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Earth Mantle Convection'
TV, image

; Make a mask of the core and scale it to range from 0
; to 255.
core = BYTSCL(image EQ 255)

; Subtract the scaled mask from the original image.
difference = image - core

; Create another window and display the difference of
; the original image and the scaled mask.
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Difference of Original & Core'
TV, difference

; Determine the statistics of the image.
IMAGE_STATISTICS, difference, COUNT = pixelNumber, $
   DATA_SUM = pixelTotal, MAXIMUM = pixelMax, $
   MEAN = pixelMean, MINIMUM = pixelMin, $
   STDDEV = pixelDeviation, $
   SUM_OF_SQUARES = pixelSquareSum, $
   VARIANCE = pixelVariance

; Print out the resulting statistics.
PRINT, ''
PRINT, 'IMAGE STATISTICS:'
PRINT, 'Total Number of Pixels = ', pixelNumber
PRINT, 'Total of Pixel Values = ', pixelTotal
PRINT, 'Maximum Pixel Value = ', pixelMax
PRINT, 'Mean of Pixel Values = ', pixelMean
PRINT, 'Minimum Pixel Value = ', pixelMin
PRINT, 'Standard Deviation of Pixel Values = ', $
   pixelDeviation
PRINT, 'Total of Squared Pixel Values = ', $
   pixelSquareSum
PRINT, 'Variance of Pixel Values = ', pixelVariance

; Derive a mask of the non-zero values of the image.
nonzeroMask = difference NE 0

; Determine the statistics of the image with the
; mask applied.
IMAGE_STATISTICS, difference, COUNT = pixelNumber, $
   DATA_SUM = pixelTotal, MASK = nonzeroMask, $
   MAXIMUM = pixelMax, MEAN = pixelMean, $
   MINIMUM = pixelMin, STDDEV = pixelDeviation, $
   SUM_OF_SQUARES = pixelSquareSum, $
   VARIANCE = pixelVariance

; Print out the resulting statistics.
PRINT, ''
PRINT, 'MASKED IMAGE STATISTICS:'
PRINT, 'Total Number of Pixels = ', pixelNumber
PRINT, 'Total of Pixel Values = ', pixelTotal
PRINT, 'Maximum Pixel Value = ', pixelMax
PRINT, 'Mean of Pixel Values = ', pixelMean
PRINT, 'Minimum Pixel Value = ', pixelMin
PRINT, 'Standard Deviation of Pixel Values = ', $
   pixelDeviation
PRINT, 'Total of Squared Pixel Values = ', $
   pixelSquareSum
PRINT, 'Variance of Pixel Values = ', pixelVariance

END
```



# Chapter 7: Warping Images

This chapter describes the following topics:

## Overview of Warping Images

In image processing, image warping is used primarily to correct optical distortions introduced by camera lenses, or to register images acquired from either different perspectives or different sensors. When correcting optical distortions, the original image may be registered to a regular grid rather than to another image. In image warping, corresponding *control points* (selected in the input and reference images) control the geometry of the warping transformation. The arrays of control points from the original input image, *Xi* and *Yi*, are stretched to conform to the control point arrays *Xo* and *Yo*, designated in the reference image. Because these transformations are frequently nonlinear, image warping is often known as *rubber sheeting*. For general tips regarding control point selection see "Tips for Selecting Control Points" on page 271.

Image warping in IDL is a three-step process. First, control points are selected between two displayed images or between an image and a grid. Second, the resulting arrays of control points, *Xi*, *Yi*, *Xo*, and *Yo*, are then input into one of IDL's warping routines. Third, the warped image resulting from the translation of the *Xi*, *Yi* points to the *Xo*, *Yo* points, is displayed. It is often useful to display the warped image as a transparency, overlaying the reference image. For more information on creating transparencies with Direct and Object Graphics, see "Creating Transparent Image Overlays" on page 272.

The following table introduces the tasks and routines covered in this chapter.

| Task | Routine | Description |
|---|---|---|
| Creating a Direct Graphics Display of Image Warping<br>See "Warping Images Using Direct Graphics" on page 274. | WSET<br>CURSOR | Set the window focus and select control point coordinates. |
| | WARP_TRI | Warp the images using WARP_TRI's triangulation and interpolation. |
| | POLYWARP | Create arrays of polynomial coefficients from the control point arrays before using POLY_2D. |
| | POLY_2D | Warp the images using the polynomial warping functions of POLY_2D. |
| | XPALETTE | Use XPALETTE to view a color table. |

Table 7-1: Image Warping Tasks and Routines

| Task | Routine | Description |
|---|---|---|
| Creating an Object Graphics Display of Image Warping<br>See "Warping Images Using Object Graphics" on page 285. | IDLgrPalette::Init | Create a palette object. |
| | XROI | Select control points using the XROI utility. |
| | WARP_TRI | Warp the input image to the reference image using the triangulation and interpolation functions of WARP_TRI. |
| | SIZE<br>BYTARR | Change the warped image into a RGB image containing an alpha channel to enable transparency. |
| | IDLgrImage::Init | Initialize transparent image and base image objects. |
| | IDLgrWindow::Init<br>IDLgrView::Init<br>IDLgrModel::Init | Initialize the objects necessary for an Object Graphics display. |

*Table 7-1: Image Warping Tasks and Routines (Continued)*

## Tips for Selecting Control Points

Both examples in this chapter use control points to define the image warping transformation. To produce accurate results, use the following guidelines when selecting corresponding control points:

- Select numerous control points. A warping transformation based on many control points produces a more accurate result than one based on only a few control points.
- Select control points near the edges of the image in addition to control points near the center of the image.
- Select a higher density of control points in irregular or highly varying areas of the image.
- Select points in which you are confident. Including points with poor accuracy may generate worse results then a warp model with fewer points.

## Creating Transparent Image Overlays

It is possible to create and display a transparent image using either IDL Direct Graphics or IDL Object Graphics. Creating a transparent image is useful in the warping process when you want to overlay a transparency of the warped image onto the reference image (the image in which *Xo*, *Yo* control points were selected). The method used to create and display the transparent image depends on whether the resulting image is being displayed with Direct Graphics or Object Graphics.

### Displaying Image Transparencies Using Direct Graphics

Creating a transparent overlay in Direct Graphics requires devising a mask to alter the array of the image that is to be displayed as a transparency. The mask retains only the pixel values that will appear in the transparent overlay. The base image and the transparent warped image can then be displayed as a blended image in a Direct Graphics window.

With Direct Graphics displays, only a single color table can be applied to the blended image in a display window. For an example of creating a blended image, combining a warped image and a base image, see "Warping Images Using Direct Graphics" on page 274.

**Note**

For precise control over the color tables associated with the reference image and the warped image transparency, consider using Object Graphics.

### Displaying Image Transparencies Using Object Graphics

In Object Graphics, a transparent image is created by adding an alpha channel to the image array. The alpha channel is used to define the level of transparency in an image object. The Object Graphics example in this chapter uses the IDLgrImage object to create an image object and employs the BLEND_FUNCTION keyword to specify how the transparency of the alpha channel is applied. Other methods of applying a transparent object include using the TEXTURE_MAP keyword in conjunction with either an IDLgrPolygon or IDLgrSurface object.

Object Graphics allows precise control over the color palettes used to display image objects. By initializing a palette object, both the reference image object and the transparent, warped image object can be displayed using individual color palettes. For an example, see "Warping Images Using Object Graphics" on page 285.

## Warping Images Using Direct Graphics

Image warping requires selection of corresponding control points in an input image and either a reference image or a regular grid. The input image is warped so that the input image control points match the control points specified in the reference image.

Using Direct Graphics, the following example warps the input image, a Magnetic Resonance Image (MRI) proton density scan of a human thoracic cavity, to the reference image, a Computed Tomography (CT) bone scan of the same region. For code that you can copy and paste into an IDL Editor window, see "Example Code: Direct Graphics Display of Image Warping" on page 282 or complete the following steps for a detailed description of the process.

1. Select the MRI proton density image file:

```
mriFile= FILEPATH('pdthorax124.jpg', $
   Subdirectory = ['examples', 'data'])
```

2. Use READ_JPEG to read in the input image, which will be warped to the CT bone scan image. Then prepare the display device, load a grayscale color table, create a window and display the image:

```
READ_JPEG, mriFile, mriImg
DEVICE, DECOMPOSED = 0
LOADCT, 0
WINDOW, 0, XSIZE = 256, YSIZE = 256, $
   TITLE = 'MRI Proton Density Input Image'
TV, mriImg
```

3. Select the CT bone scan image file:

```
ctboneFile = FILEPATH('ctbone157.jpg', $
   Subdirectory = ['examples', 'data'])
```

4. Use READ_JPEG to read in the reference image and create a window:

```
READ_JPEG, ctboneFile, ctboneImg
WINDOW, 2, XSIZE = 483, YSIZE = 410, $
   TITLE = 'CT Bone Scan Reference Image'
```

5. Load the "Hue Sat Lightness 2" color table, making the image's features easier to distinguish. After displaying the image, return to the gray scale color table.

```
LOADCT, 20
TV, ctboneImg
LOADCT, 0
```

Proceed with the following section to begin selecting control points.

## Direct Graphics Example: Selecting Control Points

This section describes selecting corresponding control points in the two displayed images. The array of control points (*Xi*, *Yi)* in the input image will be mapped to the array of points (*Xo*, *Yo)* selected in the reference image. The following image shows the points to be selected in the input image.

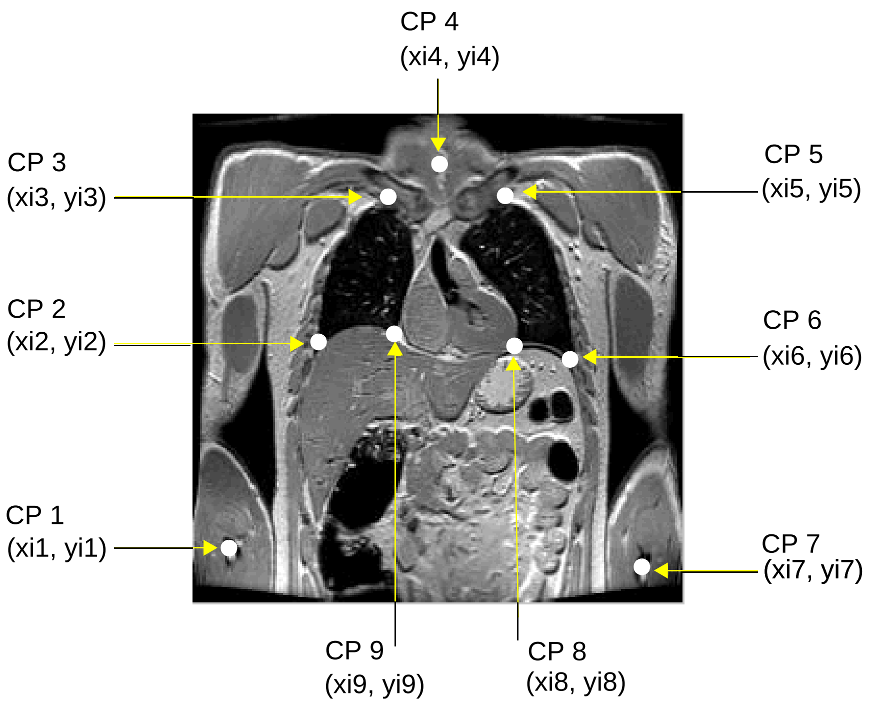


*Figure 7-1: Control Points (CP) Selection in the Input Image*

1. Set focus on the first image window:

   ```
   WSET, 0
   ```

2. Select the first control point using the CURSOR function. After entering the following line, the cursor changes to a cross hair when positioned over the image window. Position the cross hair so that it is on the first control point, “CP 1", depicted by a white circle in the lower-left corned of the previous figure, and click the left mouse button. The *x*, *y* coordinate values of the first control point will be saved in the variables *xi1*, *yi1*:

   ```
   CURSOR, xi1, yi1, /DEVICE
   ```

**Note**

The values for *xi1* and *yi1* are displayed in the IDLDE Variable Watch window. If you are not running the IDLDE, you can type `PRINT, xi1, yi1` to see the values.

**Tip**

After entering the first line and selecting the first control point in the display window, place your cursor in the IDL command line and press the Up Arrow key. The last line entered is displayed and can be easily modified.

3. Continue selecting control points. After you enter each of the following lines, select the appropriate control point in the input image as shown in the previous figure:

```
CURSOR, xi2, yi2, /DEVICE
CURSOR, xi3, yi3, /DEVICE
CURSOR, xi4, yi4, /DEVICE
CURSOR, xi5, yi5, /DEVICE
CURSOR, xi6, yi6, /DEVICE
CURSOR, xi7, yi7, /DEVICE
CURSOR, xi8, yi8, /DEVICE
CURSOR, xi9, yi9, /DEVICE
```

4. Set the focus on the window containing the reference image to prepare to select corresponding control points:

```
WSET, 2
```

**Note**

The *Xi* and *Yi* vectors and the *Xo* and *Yo* vectors must be the same length, meaning that you must select the same number of control points in the reference image as you selected in the input image. The control points must also be selected in the same order since the point Xi1, Yi1 will be warped to Xo1, Yo1.

The following figure displays the control points to be selected in the next step.

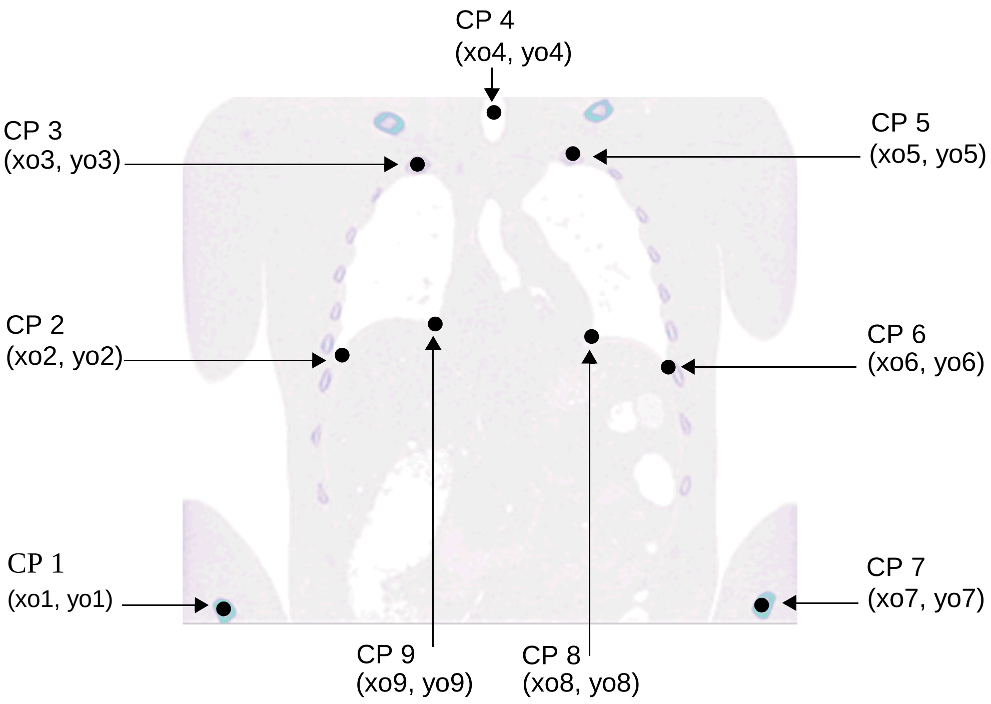


*Figure 7-2: Control Point (CP) Selection in the Reference Image*

5. Select the control points in the reference image. These are the corresponding points to which the input image control points will be warped. After entering each line, select the appropriate control point as shown in the previous figure:

```
CURSOR, xo1, yo1, /DEVICE
CURSOR, xo2, yo2, /DEVICE
CURSOR, xo3, yo3, /DEVICE
CURSOR, xo4, yo4, /DEVICE
CURSOR, xo5, yo5, /DEVICE
CURSOR, xo6, yo6, /DEVICE
CURSOR, xo7, yo7, /DEVICE
CURSOR, xo8, yo8, /DEVICE
CURSOR, xo9, yo9, /DEVICE
```

6. Place the control points into vectors (one-dimensional arrays) required by IDL warping routines. WARP_TRI and POLYWARP use the variables *Xi*, *Yi* and *Xo*, *Yo* as containers for the control points selected in the original input and reference images. Geometric transformations control the warping of the input image (*Xi*, *Yi*) values to the reference image (*Xo*, *Yo*) values. Enter the following lines to load the control point values into the one-dimensional arrays:

```
Xi = [xi1, xi2, xi3, xi4, xi5, xi6, xi7, xi8, xi9]
Yi = [yi1, yi2, yi3, yi4, yi5, yi6, yi7, yi8, yi9]
Xo = [xo1, xo2, xo3, xo4, xo5, xo6, xo7, xo8, xo9]
Yo = [yo1, yo2, yo3, yo4, yo5, yo6, yo7, yo8, yo9]
```

## Example Code: Warping and Displaying a Transparent Image Using Direct Graphics

This section uses the control points defined in the previous section to warp the original MRI scan to the CT scan, using both of IDL's warping routines, WARP_TRI and POLY_2D. After outputting the warped image, it will be altered for display as a transparency in Direct Graphics.

1. Warp the input image, *mriImg*, onto the reference image using WARP_TRI. This function uses the irregular grid of the reference image, defined by *Xo*, *Yo*, as a basis for triangulation, defining the surfaces associated with (*Xo*, *Yo*, *Xi*) and (*Xo*, *Yo*, *Yi*). Each pixel in the input image is then transferred to the appropriate position in the resulting output image as designated by interpolation. Using the WARP_TRI syntax,

```
Result = WARP_TRI(Xo, Yo, Xi, Yi, Image, OUTPUT_SIZE=vector]
   [, /QUINTIC] [, /EXTRAPOLATE])
```

set the OUTPUT_SIZE equal to the reference image dimensions since this image forms the basis of the warped, output image. Use the EXTRAPOLATE keyword to display the portions of the image which fall outside of the boundary of the selected control points:

```
warpTriImg = WARP_TRI(Xo, Yo, Xi, Yi, mriImg, $
   OUTPUT_SIZE=[483, 410], /EXTRAPOLATE)
```

**Note**

Images requiring more aggressive warp models may not have good results outside of the extent of the control points when WARP_TRI is used with the /EXTRAPOLATE keyword.

2. Create a new window and display the warped image:

```
WINDOW, 3, XSIZE = 483, YSIZE = 410, TITLE = 'WARP_TRI image'
TV, warpTriImg
```

You can see the how precisely the control points were selected by the amount of distortion in the resulting warped image. The following figure shows little distortion.

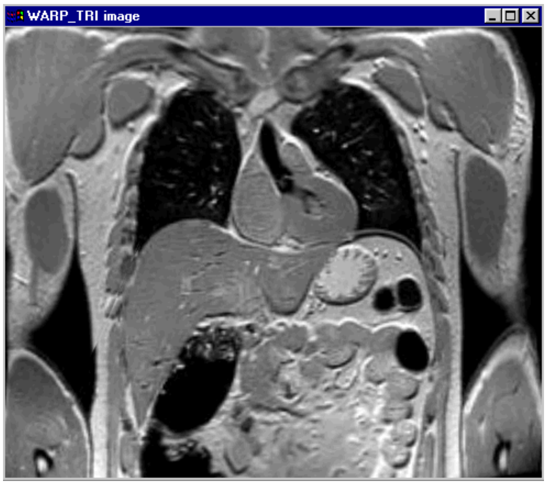


*Figure 7-3: Warped Image Produced with WARP_TRI*

3. Use POLYWARP in conjunction with POLY_2D to create another warped image for comparison with the WARP_TRI image. First use the POLYWARP procedure to create arrays (*p*, *q*) containing the polynomial coefficients required by the POLY_2D function:

```
POLYWARP, Xi, Yi, Xo, Yo, 1, p, q
```

4. Using the *p*, *q* array values generated by POLYWARP, warp the original image, *mriImg*, onto the CT bone scan using the POLY_2D function syntax,

```
Result = POLY_2D( Array, P, Q [, Interp [, Dimx, Dimy]]
    [, CUBIC={-1 to 0}] [, MISSING=value] )
```

Specify a value of `1` for the `Interp` argument to use bilinear interpolation and set *`DimX, DimY`* equal to the reference image dimensions:

```
warpPolyImg = POLY_2D(mriImg, p, q, 1, 483, 410)
```

5. Create a new window and display the image created using POLY_2D:

```
WINDOW, 4, XSIZE = 483, YSIZE = 410, TITLE = 'Poly_2D image'
TV, warpPolyImg
```

The following image shows little difference from the WARP_TRI image other than more accurate placement in the display window.

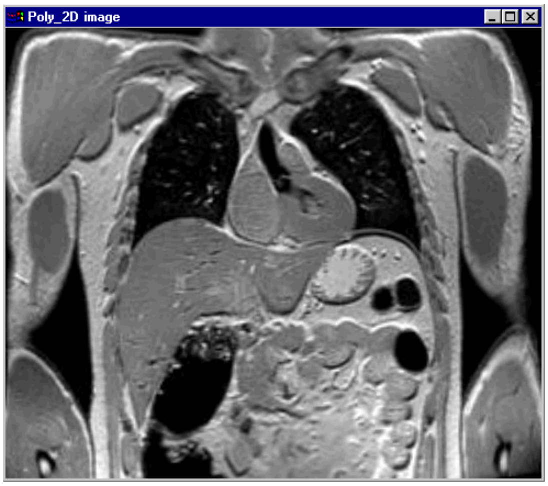


*Figure 7-4: Warped Image Produced with POLY_2D*

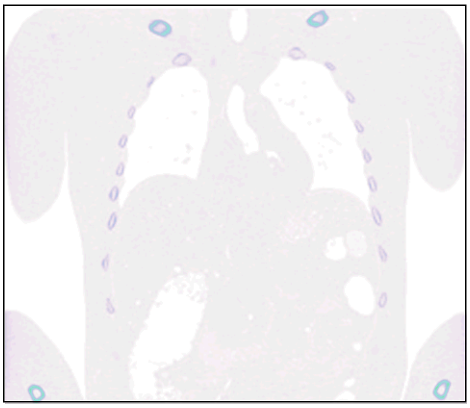
Direct Graphics displays in IDL allow you to display a combination of images in the same Direct Graphics window. The following steps display various intensities of the warped image and the reference image in a Direct Graphics window.

6. Use the XPALETTE tool to view the color table applied to the bone scan image by first entering:

```
XPALETTE
```

In the XPALETTE utility, display a color table by selecting the **Predefined** button. In the resulting XLOADCT dialog, scroll down and select **Hue Saturation Lightness 2**. Click **Done**. In the XPALETTE utility, click **Redraw**. Compare the bone scan image, displayed in window 2, to the displayed color table. To mask out the less important background information, select a color close to that of the body color in the image.

The following figure displays a portion of the XPALETTE utility with such a selection.



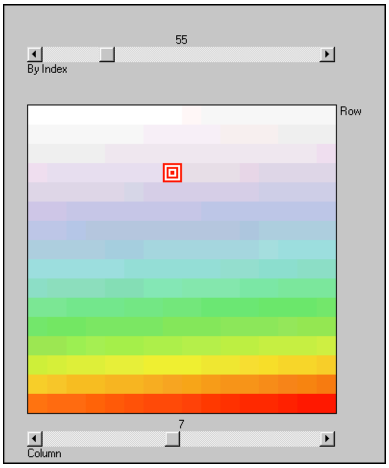


*Figure 7-5: Using XPALETTE to Identify Mask Values*

7. Using the knowledge that the body color’s index number is 55, mask out the less important background information of the bone scan image by creating an array containing only pixel values greater than 55. Multiply the mask by the image to retain the color information and use BYTSCL to scale the resulting array from 0 to 255:

```
ctboneMask = BYTSCL((ctboneImg GT 55) * ctboneImg)
```

8. Display a blended image using the full intensity of the bone scan image and a 75% intensity of the warped image. The following statement displays the pixels in the bone scan with the full range of colors in the color table while using the lower 75% of the color table values for the warped image. After adding the arrays, scale the results for display purposes:

```
blendImg = BYTSCL(ctboneMask + 0.75 * warpPolyImg)
```

9. Create a window and display the result:

```
WINDOW, 5, XSIZE = 483, YSIZE = 410, TITLE = 'Blended Image'
TV, blendImg
```

The clavicles and rib bones of the reference image are clearly displayed in the following figure.

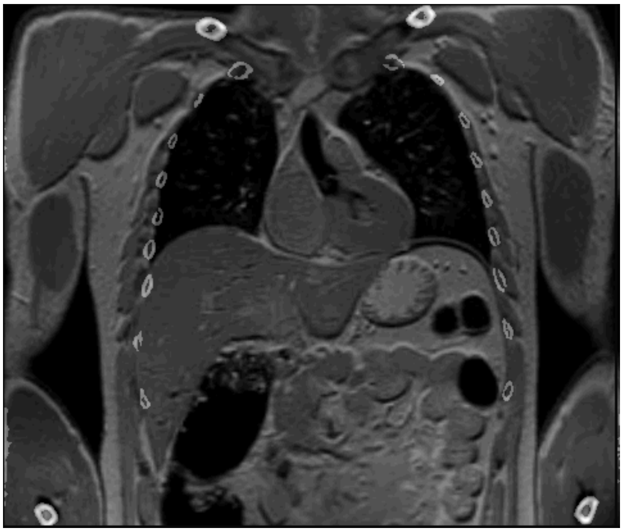


*Figure 7-6: Direct Graphics Display of a Transparent Blended Image*

While Direct Graphics supports displaying indexed images as transparent blended images, you could also apply alpha blending to RGB images that are output to a TrueColor display. However, creating image transparencies which retain their color information is more easily accomplished using Object Graphics. For an example of using Object Graphics to display a warped image transparency over another image see "Warping Images Using Object Graphics" on page 285.

## Example Code: Direct Graphics Display of Image Warping

Copy and paste the following text into an IDL Editor window. After saving the file as `MRIWarping_direct.pro`, compile and run the program to reproduce the previous example.

```
PRO MRIWarping_Direct

; Select the MRI proton density scan file.
mriFile = FILEPATH('pdthorax124.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Read in the MRI file, prepare the display,
; load the gray scale color table and display the image.
```

```
READ_JPEG, mriFile, mriImg
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 0
WINDOW, 0, XSIZE = 256, YSIZE = 256, $
   TITLE = 'MRI Proton Density Input Image'
TV, mriImg

; Select the CT bone scan file.
ctboneFile = FILEPATH('ctbone157.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Read in the file and create another window.
READ_JPEG, ctboneFile, ctboneImg
WINDOW, 2, XSIZE = 483, YSIZE = 410, $
   TITLE = 'CT Bone Scan Reference Image'

; Display the image with a color table to highlight
; features.
LOADCT, 20
TV, ctboneImg

; Return to the gray scale palette
LOADCT, 0

; Designate the control points in the input image
; (mriImg) in window 0). In the example steps, these
; points were selected with the CURSOR function.
Xi = [21, 65, 104, 129, 161, 198, 235, 170, 107]
Yi = [25, 131, 207, 229, 211, 121, 16, 134, 140]

; Designate the corresponding control points in the
; reference image (ctboneImg in window 2). In the
; example steps, these points were selected with the
; CURSOR function.
Xo = [34, 121, 183, 243, 303, 377, 454, 319, 198]
Yo = [10, 207, 357, 400, 363, 206, 12, 222, 233]

; Using the control points, warp the input image
; (mriImg) onto the reference image (ctboneImg).
warpTriImg = WARP_TRI(Xo, Yo, Xi, Yi, mriImg, $
   OUTPUT_SIZE = [483, 410], /EXTRAPOLATE)
;Display new image.
WINDOW, 3, XSIZE = 483, YSIZE = 410, $
   TITLE = 'WARP_TRI image'
TV, warpTriImg

; Use POLYWARP to create the variables (p,q) needed by
; POLY_2D.
POLYWARP, Xi, Yi, Xo, Yo, 1, p, q
```

```
; Using the values generated by POLYWARP, warp the
; original image. Specify 1 for  bilinear interpolation
; and set the output size equal to the ctboneImg image
; dimensions.
warpPolyImg = POLY_2D(mriImg, p, q, 1, 483, 410)

; Create a window and display the image.
WINDOW, 4, XSIZE = 483, YSIZE = 410, $
   TITLE = 'Poly_2D image'
TV, warpPolyImg

; Mask out the lower pixel values in the ctboneImg
; image.
ctboneMask = BYTSCL((ctboneImg GT 55) * ctboneImg)

; Display an image using the full intensity of the bone
; scan image and a 75% intensity of the warped image.
blendImg = BYTSCL(ctboneMask + 0.75 * warpPolyImg)

; Display the blended image.
WINDOW, 5, XSIZE = 483, YSIZE = 410, $
   TITLE = 'Blended Image'
TV, blendImg

END
```

# Warping Images Using Object Graphics

The following example warps an African land-cover characteristics image to a political map of the continent. After displaying the images and selecting control points in each image using the XROI utility, the resulting warped image is altered to include an alpha channel, enabling transparency. Image objects are then created and displayed in an IDL Object Graphics display. For code that you can copy and paste into an IDL Editor window, see "Example Code: Object Graphics Display of Image Warping" on page 295 or complete the following steps for a detailed description of the process.

1. Select the political map image. This is the reference image to which the land cover image will be warped:

```
mapFile= FILEPATH('afrpolitsm.png', $
   Subdirectory = ['examples', 'data'])
```

2. Use READ_PNG routine to read in the file. Specify *mapR, mapG, mapB* to read in the image's associated color table:

```
mapImg = READ_PNG(mapFile, mapR, mapG, mapB)
```

3. Using IDLgrPalette::Init, assign the image's color table to a palette object, which will be applied to an image object in a later step:

```
mapPalette = OBJ_NEW('IDLgrPalette', mapR, mapG, mapB)
```

4. Select and open the land cover input image, which will be warped to the map:

```
landFile = FILEPATH('africavlc.png', $
   Subdirectory = ['examples', 'data'])
landImg = READ_PNG (landFile, landR, landG, landB)
```

## Object Graphics Example: Selecting Control Points

This section describes using the XROI utility to select corresponding control points in the two images. The arrays of control points in the input image, (*Xi*, *Yi*), will be mapped to the array of points selected in the reference image, (*Xo*, *Yo*).

**Note**

The *Xi* and *Yi* vectors and the *Xo* and *Yo* vectors must be the same length, meaning that you must select the same number of control points in the reference image as you select in the input image. The control points must also be selected in the same order since the point *Xi1*, *Yi1* will be warped to *Xo1*, *Yo1*.

The following figure shows the points to be selected in the input image.

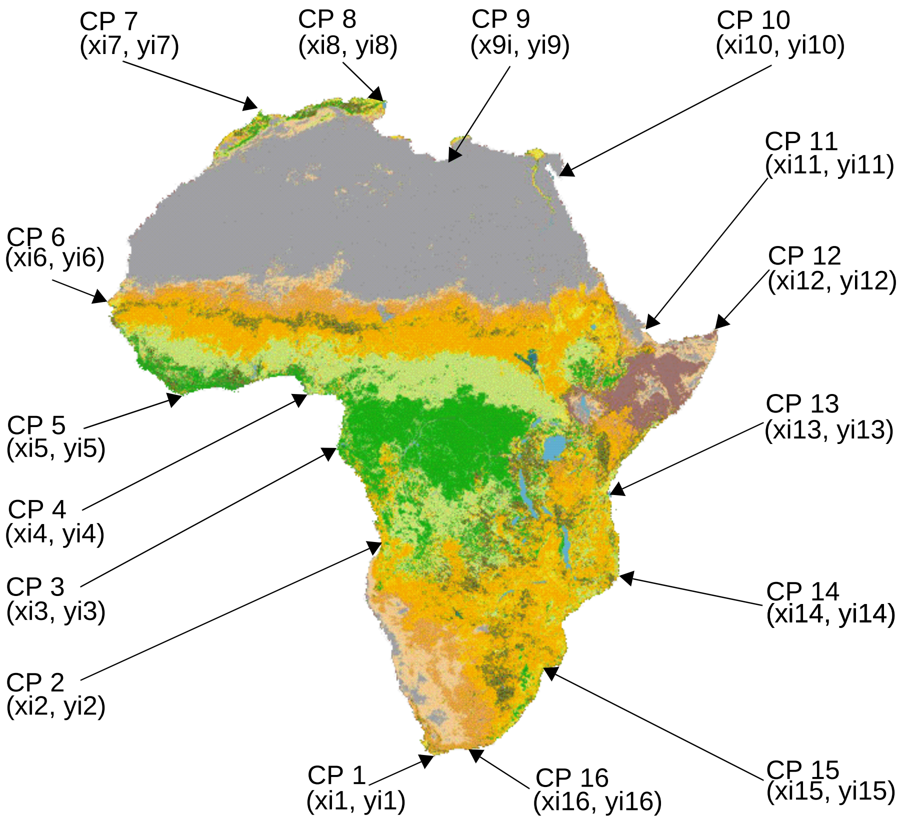


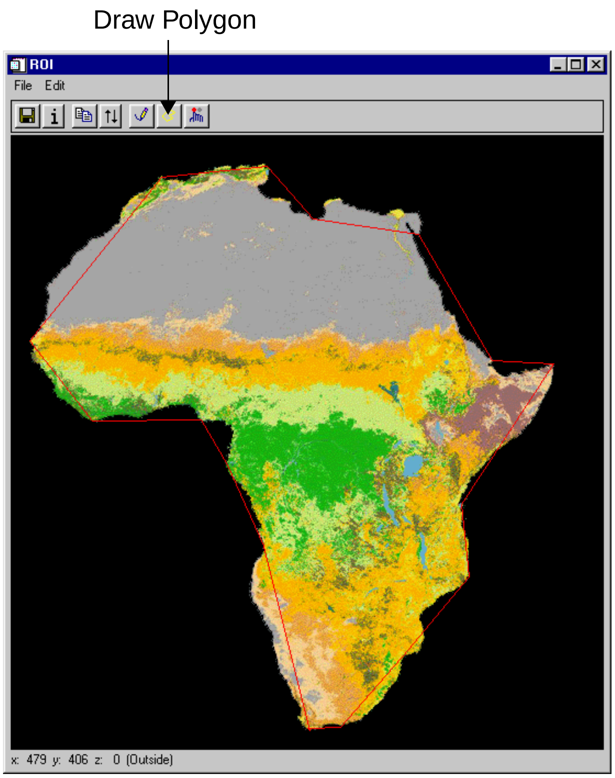
*Figure 7-7: Selecting Control Points in the Input Image*

Reasonably precise warping of the land classification image to the political map requires selecting numerous control points because of the irregularity of the continent's border. Select the control points in the land classification image as described in the following steps.

1. Load the image and its associated color table into the XROI utility, specifying the REGIONS_OUT keyword to save the region defined by the control points in the *landROIout* object:

```
XROI, landImg, landR, landG, landB, $
   REGIONS_OUT = landROIout, /BLOCK
```

Select the **Draw Polygon** button from the XROI utility toolbar shown in the following figure. Position the crosshairs symbol over CP1, shown in the previous figure, and click the left mouse button. Repeat this action for each successive control point. After selecting the sixteenth control point, position the crosshairs over the first point selected and click the right mouse button to close the region. Your display should appear similar to the following figure.



x: 479 y: 406 z: 0 (Outside)

*Figure 7-8: Selecting Control Points Using XROI*

**Note**

It is of no concern that portions of the continent lie outside the polygonal boundary. The EXTRAPOLATE keyword to WARP_TRI enables warping of the image areas lying outside of the boundary of control points. However, images requiring more aggressive warp models may not have good results outside of the extent of the control points when WARP_TRI is used with the /EXTRAPOLATE keyword.

2. Close the XROI window and assign the *landROIout* object data to the *Xi* and *Yi* control point vectors:

```
landROIout -> GetProperty, DATA = landROIdata
Xi = landROIdata[0,*]
Yi = landROIdata[1,*]
```

The following figure displays the corresponding control points to be selected in the reference image of the political map. These control points will make up the *Xo* and *Yo* arrays required by the IDL warping routines.

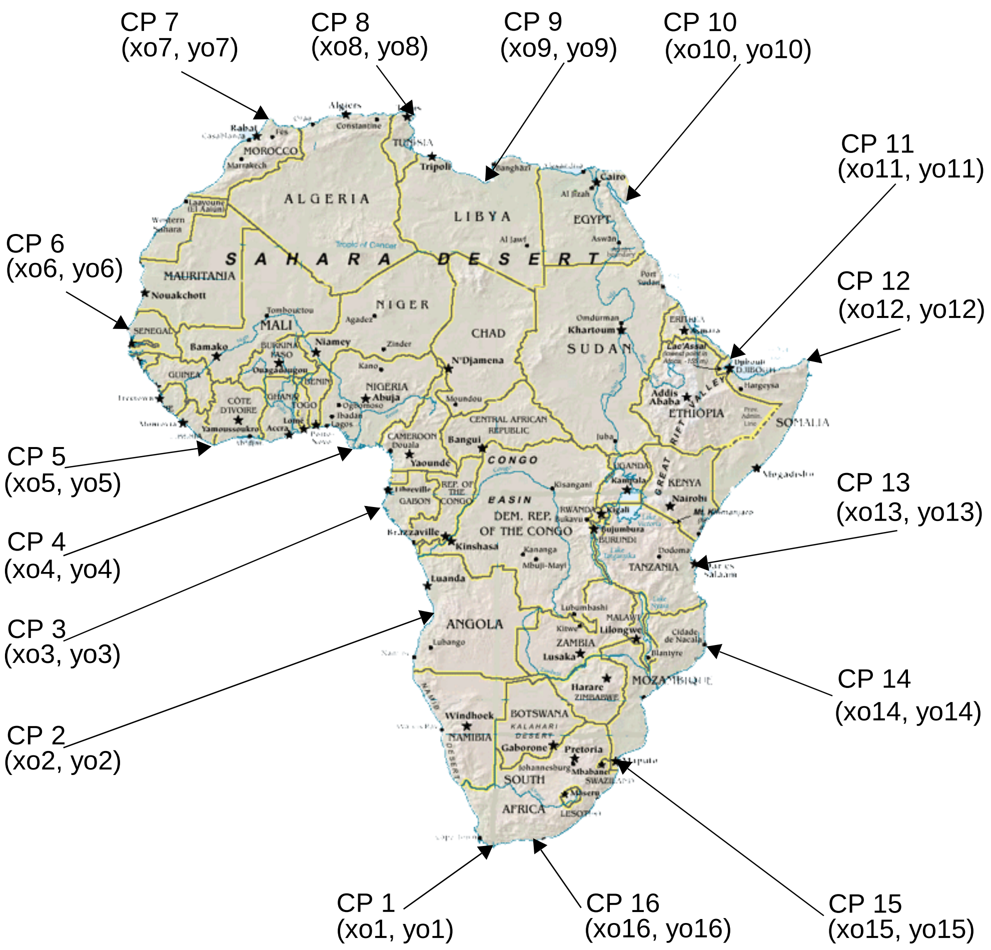


*Figure 7-9: Control Points to be Selected in the Reference Image*

3. Load the image of the political map and its associated color table into the XROI utility, specifying the REGIONS_OUT keyword to save the selected region in the *mapROIout* object:

```
XROI, mapImg, mapR, mapG, mapB, $
   REGIONS_OUT=mapROIout,/BLOCK
```

   Select the **Draw Polygon** button from the XROI utility toolbar. Position the crosshairs symbol over CP1, shown in the previous figure, and click the left mouse button. Repeat this action for each successive control point. After selecting the sixteenth control point, position the crosshairs over the first point selected and click the right mouse button to close the region. Your display should appear similar to the following figure.

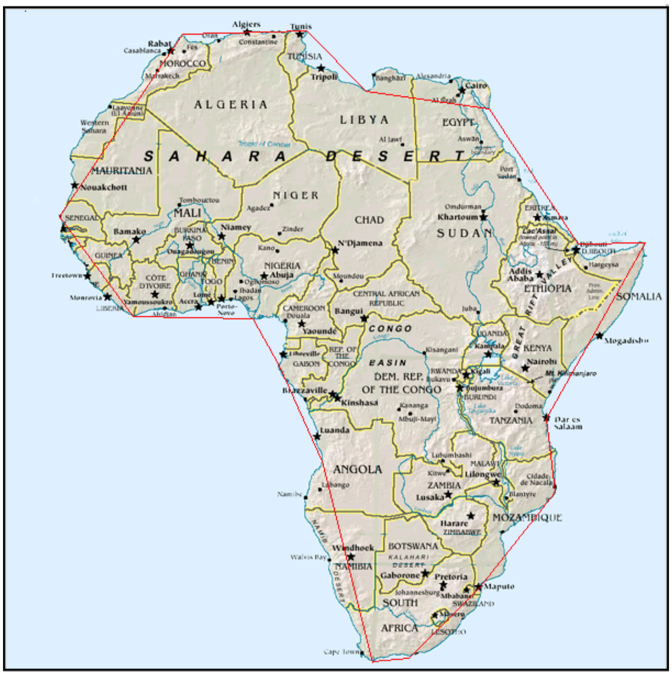


*Figure 7-10: Selecting Control Points Using XROI*

4. Close the XROI window and assign the *mapROIout* object data to the *Xo* and *Yo* control point vectors:

```
mapROIout -> GetProperty, DATA=mapROIdata
Xo = mapROIdata[0,*]
Yo = mapROIdata[1,*]
```

## Object Graphics Example: Warping and Displaying a Transparent Image

The following section describes warping the land cover image to the political map and creating image objects. The resulting warped image will then be made into a transparency by creating an alpha channel for the image. Finally, the transparent object will be displayed as an overlay to the original political map.

1. Warp the input image, *landImg*, onto the reference image using WARP_TRI. This function uses the irregular grid of the reference image, defined by *Xo*, *Yo*, as a basis for triangulation, defining the surfaces associated with (*Xo*, *Yo*, *Xi*) and (*Xo*, *Yo*, *Yi*). Each pixel in the input image is then transferred to the appropriate position in the resulting output image as designated by interpolation. Using the WARP_TRI syntax,

   ```
   Result = WARP_TRI( Xo, Yo, Xi, Yi, Image
      [, OUTPUT_SIZE=vector][, /QUINTIC] [, /EXTRAPOLATE]
   )
   ```

   set the OUTPUT_SIZE equal to the reference image dimensions since this image forms the basis of the warped, output image. Use the EXTRAPOLATE keyword to display the portions of the image which fall outside of the boundary of selected control points:

   ```
   warpImg = WARP_TRI(Xo, Yo, Xi, Yi, landImg, $
      OUTPUT_SIZE=[600, 600], /EXTRAPOLATE)
   ```

2. While not required, you can quickly check the precision of the warp in a Direct Graphics display before proceeding with creating a transparency by entering the following lines:

   ```
   DEVICE, DECOMPOSED = 0
   TVLCT, landR, landG, landB
   WINDOW, 3, XSIZE = 600, YSIZE = 600, $
      TITLE = 'Image Warped with WARP_TRI'
   TV, warpImg
   ```

Precise control point selection results in accurate warping. If there is little distortion, as in the following figure, control points were successfully selected in nearly corresponding positions in both images.

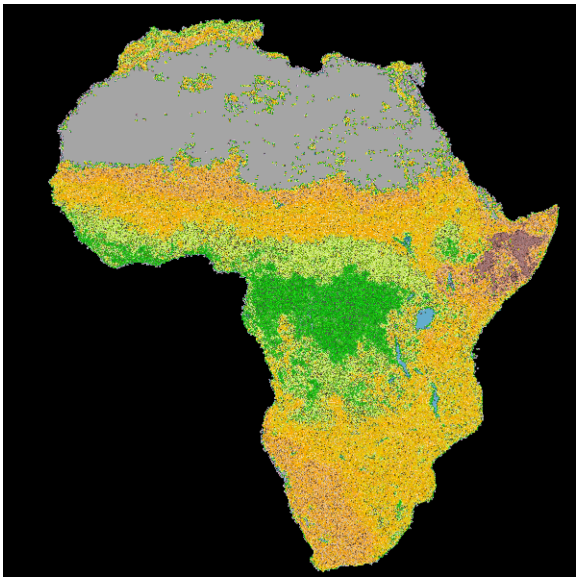


*Figure 7-11: Resulting Warped Image*

3. A transparent image object must be a grayscale or an RGB (24-bit) image containing an alpha channel. The alpha channel controls the transparency of the pixels. See IDLgrImage::Init for more information.

   The following lines convert the warped image and its associated color table into a RGB image containing four channels (red, green, blue, and alpha). First, get the dimensions of the warped image and then use BYTARR to create *alphaWarp*, a 4-channel by *xdim* by *ydim* array, where (*xdim*, *ydim*) are the dimensions of the warped image:

   ```
   warpImgDims = SIZE(warpImg, /Dimensions)
   alphaWarp = BYTARR(4, warpImgDims[0], warpImgDims[1])
   ```

4. Load the red, green and blue channels of the warped land characteristics image into the first three channels of the *alphaWarp* array:

   ```
   alphaWarp[0, *, *] = landR[warpImg]
   alphaWarp[1, *, *] = landG[warpImg]
   alphaWarp[2, *, *] = landB[warpImg]
   ```

5. Define the transparency of the alpha channel. First, create an array, masking out the black background of the warped image (where pixel values equal 0) by retaining only pixels with values greater than 0:

   ```
   mask = (warpImg GT 0)
   ```

   Apply the resulting mask to the alpha channel, the fourth channel of the array. This channel creates a 50% transparency of the pixels of the first three channels (red, green, blue) of the *alphaWarp* by multiplying the *mask* by 128B (byte). Alpha channel values range from 0 (completely transparent) to 255 (completely opaque):

   ```
   alphaWarp [3, *, *] = mask*128B
   ```

**Note**

You can set the transparency of an entire image. To set the transparency of *all* pixels at 50% in this example, your could replace the two previous steps with the following two lines:

```
mask = BYTARR(s[0], s[1]) + 128
alphaWarp [3, *, *] = mask
```

6. Initialize the transparent image object using IDLgrImage::Init. Specify the BLEND_FUNCTION property of the image object to control how the alpha channel is interpreted. Setting the BLEND_FUNCTION to [3, 4] allows you to see through the foreground image to the background. The foreground opacity is defined by the alpha channel value, specified in the previous step:

   ```
   oAlphaWarp = OBJ_NEW('IDLgrImage', alphaWarp, $
      DIMENSIONS = [600, 600], BLEND_FUNCTION = [3, 4])
   ```

7. Initialize the reference image object, applying the palette created earlier:

   ```
   oMapImg = OBJ_NEW('IDLgrImage', mapImg, $
      DIMENSIONS = [600,600], PALETTE = mapPalette)
   ```

8. Using IDLgrWindow::Init, initialize a window object in which to display the images:

   ```
   oWindow = OBJ_NEW('IDLgrWindow', DIMENSIONS = [600, 600], $
      RETAIN = 2, TITLE = 'Overlay of Land Cover Transparency')
   ```

9. Create a view object using IDLgrView::Init. The VIEWPLANE_RECT keyword controls the image display in the Object Graphics window. First create an array, *viewRect*, which specifies the *x-placement*, *y-placement*, *width*, and *height* of the view surface. The values `0, 0` place the (0, 0) coordinate of viewing surface in the lower-left corner of the Object Graphics window:

```
viewRect = [0, 0, 600, 600]
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT = viewRect)
```

10. Using IDLgrModel::Init, initialize a model object to which the images will be applied. Add the base image and the transparent alpha image to the model:

```
oModel = OBJ_NEW('IDLgrModel')
oModel -> Add, oMapImg
oModel -> Add, oAlphaWarp
```

**Note**

Image objects appear in the Object Graphics window in the order in which they are added to the model. If a transparent object is added to the model before an opaque object, it will not be visible.

11. Add the model, containing the images, to the view and draw the view in the window:

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

The following figure shows the warped image transparency overlaid onto the original reference image, the political map.

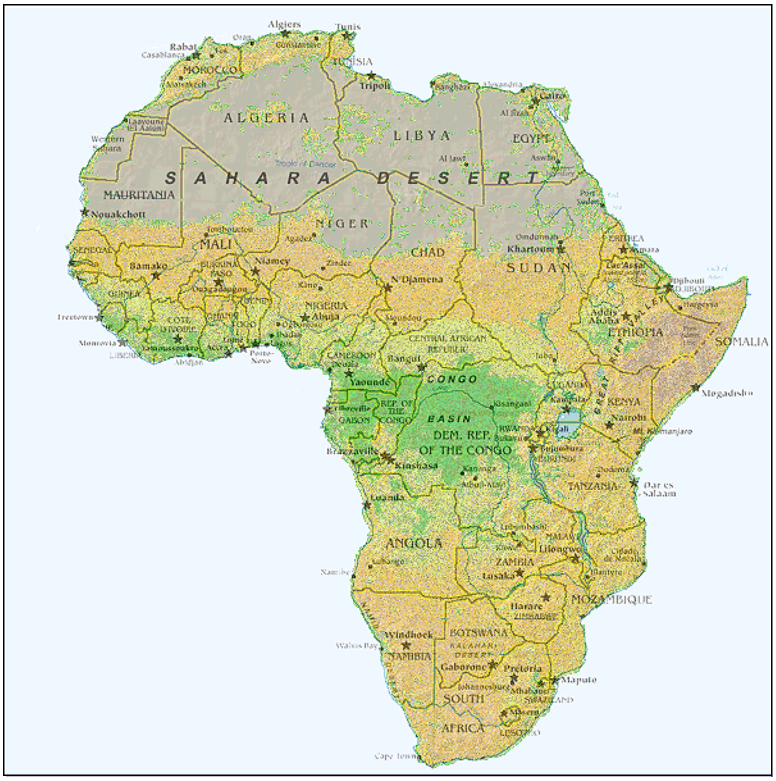


*Figure 7-12: Object Graphics Display of the Political Map with a Transparent Land Cover Overlay*

12. Use OBJ_DESTROY to clean up unneeded object references including the region objects:

```
OBJ_DESTROY, [oView, oMapImg, oAlphaWarp, $
   mapPalette, landROIout, mapROIout]
```

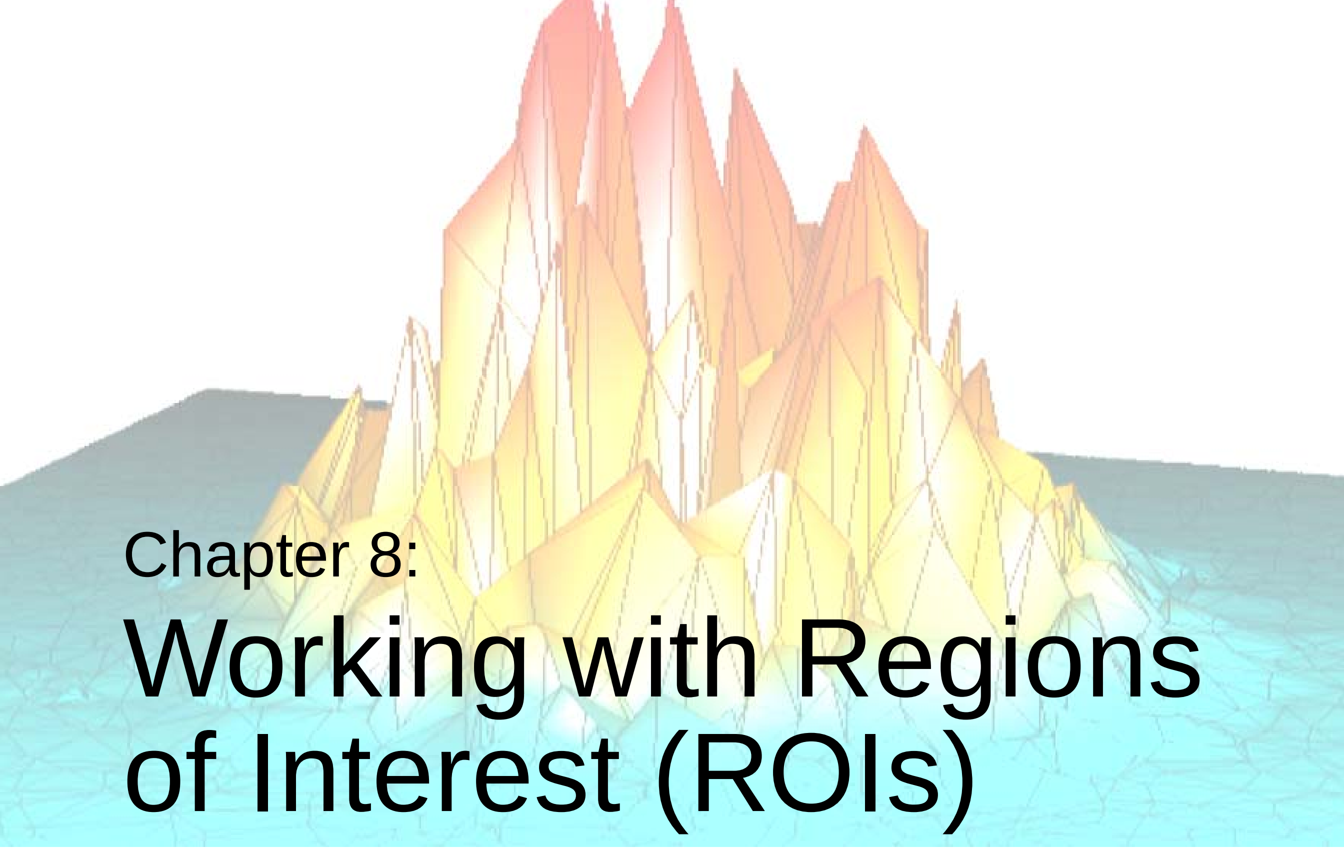
## Example Code: Object Graphics Display of Image Warping

Copy and paste the following text into an IDL Editor window. After saving the file as `TransparentWarping_object.pro`, compile and run the program to reproduce the previous example.

```
PRO TransparentWarping_Object

; Open the political map, the base image to which the
; land cover image will be warped.
mapFile= FILEPATH('afrpolitsm.png', $
   SUBDIRECTORY = ['examples', 'data'])
mapImg = READ_PNG(mapFile, mapR, mapG, mapB)

; Assign the mapImg's color table to a palette object.
mapPalette = OBJ_NEW('IDLgrPalette', mapR, mapG, mapB)

; Open the land cover characteristics image
; that will be warped to the political map.
landFile = FILEPATH('africavlc.png', $
   SUBDIRECTORY = ['examples', 'data'])
landImg = READ_PNG (landFile, landR, landG, landB)

; Select the control point using the polygon tool in
; XROI.
XROI, landImg, landR, landG, landB, $
   REGIONS_OUT = landROIout, /BLOCK
PRINT, 'Select control points using Draw Polygon tool'
; Assign the ROI data to the Xi and Yi control point
; vectors.
landROIout -> GetProperty, DATA = landROIdata
Xi = landROIdata[0,*]
Yi = landROIdata[1,*]

; Select the control point in the reference image
; using the polygon tool in XROI.
XROI, mapImg, mapR, mapG, mapB, $
   REGIONS_OUT = mapROIout, /BLOCK
PRINT, 'Select control points using Draw Polygon tool'
; Assign the ROI data to the Xo and Yo control point
; vectors.
mapROIout -> GetProperty, DATA = mapROIdata
Xo = mapROIdata[0,*]
Yo = mapROIdata[1,*]

; Using the control point vectors, warp the land
; classification image to the political map.
warpImg = WARP_TRI(Xo, Yo, Xi, Yi, landImg, $
   OUTPUT_SIZE = [600, 600], /EXTRAPOLATE)
```

```
; Quickly display the warped image in a Direct Graphics
; window to check the precision of the warp. Load the
; image's associated color table and display it.
DEVICE, DECOMPOSED = 0
TVLCT, landR, landG, landB
WINDOW, 3, XSIZE = 600, YSIZE = 600, $
   TITLE = 'Image Warped with WARP_TRI'
TV, warpImg

; Make the warped land classification image into a
; 24-bit RGB image in order to use alpha blending.
warpImgDims = SIZE(warpImg, /Dimensions)
alphaWarp = BYTARR(4, warpImgDims[0], warpImgDims[1])

; Get the red, green and blue values used by the image
; and assign them to the first three channels of the
; alpha image array.
alphaWarp[0, *, *] = landR[warpImg]
alphaWarp[1, *, *] = landG[warpImg]
alphaWarp[2, *, *] = landB[warpImg]

; Create a transparency mask, the same size as the
; warpImg array. Mask out the black pixels with a
; values of 0. Set the alpha channel by multiplying
; the mask by 128, resulting in a 50% transparency.
mask = (warpImg GT 0)
alphaWarp [3, *, *] = mask*128B
; To alter the transparency, change the value 128. This
; value can range from 0 (completely transparent) to
; 255 (completely opaque).

; Create the objects necessary for the Object Graphics
; display. Create the transparent overlay image object.
oAlphaWarp = OBJ_NEW('IDLgrImage', alphaWarp, $
   DIMENSIONS = [600, 600], BLEND_FUNCTION = [3,4])

; Create the background, mapImg object and apply its
; palette.
oMapImg = OBJ_NEW('IDLgrImage', mapImg, $
   DIMENSIONS = [600, 600], PALETTE = mapPalette)

; Create a window in which to display the objects.
oWindow = OBJ_NEW('IDLgrWindow', $
   DIMENSIONS = [600, 600], RETAIN = 2, $
   TITLE = 'Overlay of Land Cover Transparency')

; Create a view.
viewRect = [0, 0, 600, 600]
```

```
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT = viewRect)

; Create a model object.
oModel = OBJ_NEW('IDLgrModel')

; Add the transparent image after the adding the base
; image.
oModel -> Add, oMapImg
oModel -> Add, oAlphaWarp

; Add the model containing the images to the view.
oView -> Add, oModel

; Draw the view in the window.
oWindow -> Draw, oView

; Clean up objects.
OBJ_DESTROY, [oView, oMapImg, oAlphaWarp, $
   mapPalette, landROIout, mapROIout]

END
```



# Chapter 8: Working with Regions of Interest (ROIs)

This chapter describes creating and analyzing regions of interest (ROIs) and includes the following topics:

## Overview of Working with ROIs

A region of interest (ROI) is an area of an image defined for further analysis or processing. There are several ways to define ROIs. The XROI utility enables the interactive definition of single or multiple regions from an image using the mouse. Routines such as CONTOUR or REGION_GROW enable the programmatic definition of ROIS. CONTOUR traces the outlines of thresholded ROIs while the REGION_GROW routine expands an initial region to include all connected, neighboring pixels that meet given conditions. Once an ROI is defined, it can be displayed or undergo further analysis.

An ROI can be displayed using either Direct Graphics or Object Graphics. In Direct Graphics, the DRAW_ROI routine quickly displays single or multiple ROI objects or an ROI group. In Object Graphics, the XROI utility displays defined ROIs and can output ROI data to specified ROI objects. Any ROI object, whether defined programmatically or interactively, can undergo further processing as an analysis-oriented IDLanROI object, or can be used for display as an IDLgrROI object. See IDLanROI and IDLgrROI in the *IDL Reference Guide* for more information.

**Note**

When computing ROI geometry, there is a difference between a region's area when it is displayed on a screen versus the region's computed, geometric area. See "Contrasting an ROI's Geometric Area and Mask Area" on page 302 for details.

Multiple ROIs can also be defined from a multi-image data set and added to an IDLanROIGroup object for triangulation into a 3D mesh. Alternatively, multiple ROIs can be defined in a single image and added to a group object. ROI groups can be displayed in a Direct Graphics window with DRAW_ROI or with the Object Graphics XOBJVIEW utility.

The following table introduces the tasks and routines covered in this chapter.

| Task | Routine(s)/Object(s) | Description |
|---|---|---|
| "Defining Regions of Interest" on page 303. | XROI | Create an ROI interactively, prior to analysis or display. |

*Table 8-1: Tasks and Routines Associated with Regions of Interest*

| Task | Routine(s)/Object(s) | Description |
|---|---|---|
| "Displaying ROI Objects in a Direct Graphics Window" on page 306. | DRAW_ROI | Display ROI objects in a Direct Graphics window. |
| "Programmatically Defining ROIs" on page 311. | CONTOUR<br>DRAW_ROI<br>IDLanROI::ComputeMask<br>IMAGE_STATISTICS<br>IDLanROI::ComputeGeometry | Define ROIs using CONTOUR and display them using DRAW_ROI. Return various statistics for each ROI. |
| "Growing a Region" on page 317. | REGION_GROW | Expand an original region to include all connected, neighboring pixels which meet specified constraints. |
| "Creating and Displaying an ROI Mask" on page 324. | IDLanROI::ComputeMask | Create a 2D mask of an ROI, compute the area of the mask and display a magnified view of the image region. |
| "Testing an ROI for Point Containment" on page 330. | IDLanROI::ContainsPoints | Determine whether a point lies within the boundary of a region. |
| "Creating a Surface Mesh of an ROI Group" on page 334. | IDLanROIGroup::Add<br>IDLanROIGroup::ComputeMesh<br>XOBJVIEW | Add ROIs to an ROI group object, triangulate a surface mesh and display the group object using XOBJVIEW. |

*Table 8-1: Tasks and Routines Associated with Regions of Interest (Continued)*

## Contrasting an ROI's Geometric Area and Mask Area

When working with ROIs, many users note a discrepancy between the computation of an ROI's geometric area and the computation of the mask area (the number of pixels an ROI contains when displayed). Intuition might lead one to believe that the results should be the same. However, as the following figure shows, the computed geometric area (the result of a pure mathematical calculation) differs from the displayed (masked) area, which is subject to the artifacts of digital sampling.

When displaying a region (or computing the area of its mask), each vertex of the region is mapped to a corresponding discrete pixel location. No matter where the vertex falls within the pixel, the entire pixel location is set since the region is being displayed. For example, for any vertex coordinate (x, y) where:

```
1.5 ≤ x < 2.5 and 1.5 ≤ y < 2.5
```

the vertex coordinate is assigned a value of (2, 2). Therefore, the area of the displayed (masked) region is typically larger than the computed geometric area. While the geometric area of a 2 by 2 region equals 4 as expected, the mask area of the identical region equals 9 due to the centering of the pixels when the region is displayed.

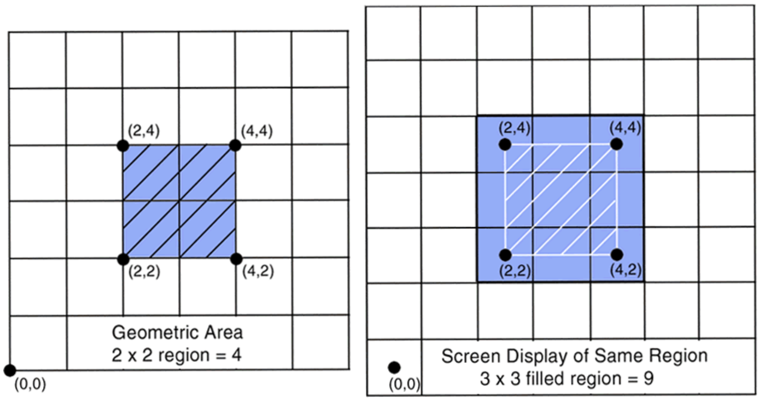


*Figure 8-1: A Region's Undisplayed Area (left) vs. Displayed Area (right)*

The ROI Information dialog of the XROI utility reports the region's "Area" (geometric area) and "# Pixels" (mask area). To programmatically compute an ROI's geometric area, use IDLanROI::ComputeGeometry. To programmatically compute the area of a displayed region, use IDLanROI::ComputeMask in conjunction with IMAGE_STATISTICS. See "Programmatically Defining ROIs" on page 311 for examples of these computations.

## Defining Regions of Interest

The XROI utility allows you to quickly load an image file, define single or multiple ROIs, and obtain geometry and statistical data about the ROIs. While regions can be defined programmatically (see "Programmatically Defining ROIs" on page 311 and "Growing a Region" on page 317), the XROI utility enables the interactive creation and selection of an ROI using the mouse.

For a quick introduction to creating ROIs using XROI, complete the following steps:

1. Open XROI by typing the following at the command line:

   ```
   XROI
   ```

2. Load an image using the image file selection dialog. Select `earth.jpg` from the `examples/demo/demodata` directory. Click **Open**.The image appears in the XROI utility.

The XROI toolbar contains the following buttons:

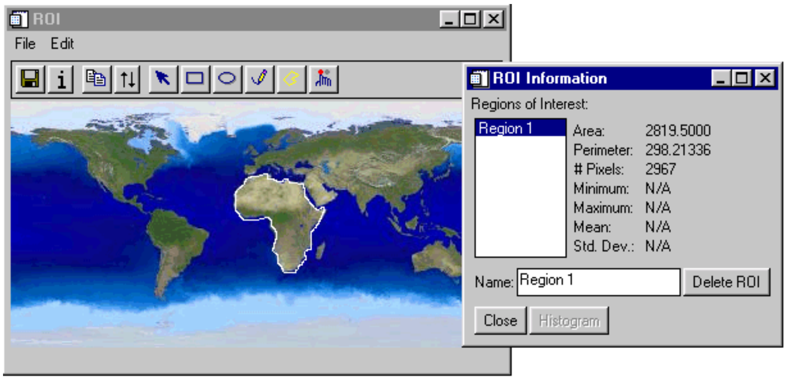
| | | |
|---|---|---|
| 💾 | **Save:** | Opens a file selection dialog for saving ROIs to a `.sav` file. |
| i | **Info:** | Opens the **ROI Information** window for the currently defined ROI. |
| 📋 | **Copy:** | Copies the contents of the display area to the clipboard. |
| ↑↓ | **Flip:** | Flips the image vertically. Any defined ROIs do not move. |

Depending on the value of the TOOLS keyword, the XROI toolbar may also contain the following buttons:

| | | |
|---|---|---|
| ↖ | **Translate/ Scale:** | Click this button to translate or scale ROIs. Mouse down inside the bounding box selects a region, mouse motion translates (repositions) the region. Mouse down on a scale handle of the bounding box enables scaling (stretching, enlarging and shrinking) of the region according to mouse motion. Mouse up finishes the translation or scaling. |
| ▭ | **Draw Rectangle:** | Click this button to draw rectangular ROIs. Mouse down positions one corner of the rectangle, mouse motions creates the rectangle, positioning the rectangle's opposite corner, mouse up finishes the rectangular region. |

| | |
|---|---|
| **Draw Ellipse:** | Click this button to draw elliptical ROIs. Mouse down positions the center of the ellipse, mouse motion positions the corner of the ellipse's bounding box, mouse up finishes the elliptical region. |
| **Draw Freehand:** | Draw freehand ROIs. Mouse down begins an ROI, mouse motion defines the ROI vertices (following the path of the mouse), mouse up closes the ROI. |
| **Draw Polygon:** | Draw polygon ROIs. Mouse down begins an ROI, subsequent mouse clicks add vertices, double-click closes the ROI. |
| **Select:** | Select an ROI. Clicking the image draws a cross hairs symbol at the nearest vertex of the selected ROI. |

3. Flip the image vertically to display it right-side-up by clicking the **Flip** button.
4. Select the **Draw Freehand** button and use the mouse to interactively define an ROI encompassing the African continent. Your image should be similar to the following figure.



Area: 2819.5000
Perimeter: 298.21336
# Pixels: 2967
Minimum: N/A
Maximum: N/A
Mean: N/A
Std. Dev.: N/A
Name: Region 1

*Figure 8-2: Defining an ROI of Africa and Showing the ROI Information Dialog*

5. After releasing the mouse button, the ROI Information dialog appears, displaying ROI statistics. You can now define another ROI, save the defined ROI as a `.sav` file or exit the XROI utility.

Using XROI syntax allows you to programmatically load an image and specify a variable for REGIONS_OUT that will contain the ROI data. The region data can then undergo further analysis and processing. The following code lines open the previously opened image for ROI creation and selection and specify to save the region data as *oROIAfrica*.

```
; Select the file, read the data and load the image’s color table.
imgFile = FILEPATH('earth.jpg', $
   SUBDIRECTORY = ['examples', 'demo', 'demodata'])
image = READ_IMAGE(imgFile, R, G, B)
TVLCT, R, G, B

; Display the image using XROI. Specify a variable for REGIONS_OUT
; to save the ROI information.
XROI, image, R, G, B, REGIONS_OUT = oROIAfrica
```

The ROI information, *oROIAfrica*, can then be analyzed using IDLanROI methods or the REGION_GROW procedure. The ROI data can also be displayed using DRAW_ROI or as an IDLgrROI object. Such tasks are covered in the following sections.

# Displaying ROI Objects in a Direct Graphics Window

The DRAW_ROI procedure displays single or multiple IDLanROI objects in a Direct Graphics window. The procedure allows you to layer the ROIs over the original image and specify the line style and color with which each region is drawn. The DRAW_ROI procedure also provides a means of easily displaying interior regions or "holes" within a defined ROI.

The following example uses the XROI utility to define two regions, a femur and tibia from a DICOM image of a knee, and draws them in a Direct Graphics window. For code that you can copy and paste into an Editor window, see "Example Code: Displaying ROIs in a Direct Graphics Window" on page 309 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load a grayscale color table:

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

2. Select and open the image file using the READ_DICOM function and get its size:

```
kneeImg = READ_DICOM(FILEPATH('mr_knee.dcm', $
   SUBDIRECTORY = ['examples','data']))
dims = SIZE(kneeImg, /DIMENSIONS)
```

3. Rotate the image 180 degrees so that the femur will be at the top of the display:

```
kneeImg = ROTATE(BYTSCL(kneeImg), 2)
```

4. Open the file in the XROI utility to create an ROI containing the femur. The following line includes the ROI_GEOMETRY and STATISTICS keywords so that specific ROI information can be retained for printing in a later step:

```
XROI, kneeImg, REGIONS_OUT = femurROIout, $
   ROI_GEOMETRY = femurGeom,$
   STATISTICS = femurStats, /BLOCK
```

Select the **Draw Polygon** button from the XROI utility toolbar, shown in the following figure. Position the crosshairs anywhere along the border of the femur and click the left mouse button to begin defining the ROI. Move your mouse to another point along the border and left-click again. Repeat the process until you have defined the outline for the ROI. To close the region, double-click the left mouse button. Your display should appear similar to the following figure.

Close the XROI utility to store the ROI information in the variable, *femurROIout*.

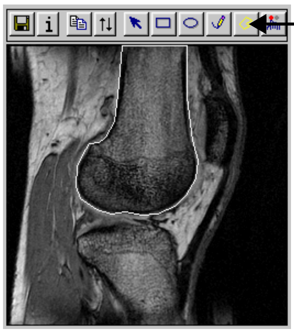


*Figure 8-3: Defining the Femur ROI*

5. Create an ROI containing the tibia, using the following XROI statement:

```
XROI, kneeImg, REGIONS_OUT = tibiaROIout, $
   ROI_GEOMETRY = tibiaGeom, $
   STATISTICS = tibiaStats, /BLOCK
```

Select the **Draw Polygon** button from the XROI utility toolbar. Position the crosshairs symbol anywhere along the border of the tibia and draw the region shown in the following figure, repeating the same steps as those used to define the femur ROI. Close the XROI utility to store the ROI information in the specified variables.

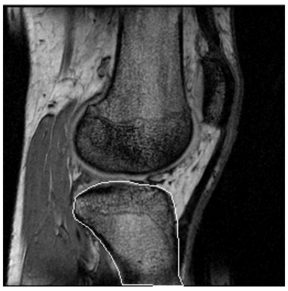


*Figure 8-4: Defining the Tibia ROI*

6. Create a Direct Graphics display containing the original image:

   ```
   WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1]
   TVSCL, kneeImg
   ```

7. Load the 16-level color table to display the regions using different colors. Use DRAW_ROI statements to specify how each ROI is drawn:

   ```
   LOADCT, 12
   DRAW_ROI, femurROIout, /LINE_FILL, COLOR = 80, SPACING = 0.1,
   $
      ORIENTATION = 315,  /DEVICE
   DRAW_ROI, tibiaROIout, /LINE_FILL, COLOR = 42, SPACING = 0.1,
   $
      ORIENTATION = 30,  /DEVICE
   ```

   In the previous statements, the ORIENTATION keyword specifies the degree of rotation of the lines used to fill the drawn regions. The DEVICE keyword indicates that the vertices of the regions are defined in terms of the device coordinate system where the origin (0,0) is in the lower-left corner of the display.

Your results should appear similar to the following figure, with the ROI objects layered over the original image.

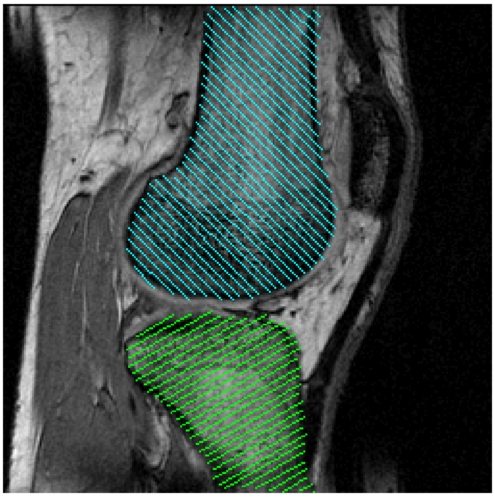


*Figure 8-5: Defined Region Objects Overlaid onto Original Image*

8. Print the statistics for the femur and tibia ROIs. This information has been stored in the *femurGeom*, *femurStat*, *tibiaGeom* and *tibiaStat* variable structures, defined in the previous XROI statements. Use the following lines to print geometrical and statistical data for each ROI:

```
PRINT, 'FEMUR Region Geometry and Statistics'
PRINT, 'area =', femurGeom.area, $
   'perimeter = ', femurGeom.perimeter, $
   'population =', femurStats.count
PRINT, ' '
PRINT, 'TIBIA Region Geometry and Statistics'
PRINT, 'area =', tibiaGeom.area, $
   'perimeter = ', tibiaGeom.perimeter, $
   'population =', tibiaStats.count
```

**Note**

Notice the difference between the “area” value, indicating the region’s geometric area, and the “population” value, indicating the number of pixels covered by the region when it is displayed. This difference is expected and is explained in the section, “Contrasting an ROI’s Geometric Area and Mask Area” on page 302.

9. Clean up object references that are not destroyed by the window manager when you close the Object Graphics displays:

```
OBJ_DESTROY, [femurROIout, tibiaROIout]
```

## Example Code: Displaying ROIs in a Direct Graphics Window

Copy and paste the following text into the IDL Editor window. After saving the file as `DrawROIex.pro`, compile and run the program to reproduce the previous example.

```
PRO DrawROIex

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

; Select and open the image file and get its size.
kneeImg = READ_DICOM(FILEPATH('mr_knee.dcm',$
   SUBDIRECTORY = ['examples','data']))
dims = SIZE(kneeImg, /DIMENSIONS)

; Flip the image vertically.
kneeImg = ROTATE(BYTSCL(kneeImg), 2)

; Open the file in the XROI utility to select the femur region.
```

```
XROI, kneeImg, REGIONS_OUT = femurROIout, $
   ROI_GEOMETRY = femurGeom,$
   STATISTICS = femurStats, /BLOCK

; Open the file in XROI to select tibia region.
XROI, kneeImg, REGIONS_OUT = tibiaROIout, $
   ROI_GEOMETRY = tibiaGeom, $
   STATISTICS = tibiaStats, /BLOCK

; Create a window and display the original image.
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1]
TVSCL, kneeImg

; Load the 16-level colortable to display regions in color
; and draw them in a Direct Graphics window.
LOADCT, 12
DRAW_ROI, femurROIout, /LINE_FILL, COLOR = 80, SPACING = 0.1, $
   ORIENTATION = 315, /DEVICE
DRAW_ROI, tibiaROIout, /LINE_FILL, COLOR = 42, SPACING = 0.1, $
   ORIENTATION = 30, /DEVICE

; Print selected stats for the femur and tibia.
PRINT, 'FEMUR Region Geometry and Statistics'
PRINT, 'area =', femurGeom.area, $
   '    perimeter = ', femurGeom.perimeter, $
   '    population =',  femurStats.count
PRINT, ' '
PRINT, 'TIBIA Region Geometry and Statistics'
PRINT, 'area =', tibiaGeom.area, $
   '    perimeter = ', tibiaGeom.perimeter, $
   '    population =', tibiaStats.count

; Destroy object references.
OBJ_DESTROY, [femurROIout, tibiaROIout]

END
```

## Programmatically Defining ROIs

While most examples in this chapter use interactive methods to define ROIs, a region can also be defined programmatically. The following example uses thresholding and the CONTOUR function to programmatically trace region outlines. After the path information of the regions has been input into ROI objects, the DRAW_ROI procedure displays each region. The example then computes and returns the geometric area and perimeter of each region as well as the number of pixels making up each region when it is displayed.

For code that you can copy and paste into an Editor window, see “Example Code: Defining an ROI and Computing ROI Statistics” on page 314 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load a color table:

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

2. Select and open the image file and get its dimensions:

```
img = READ_PNG(FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data']))
dims = SIZE(img, /DIMENSIONS)
```

3. Create a window and display the original image:

```
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1]
TVSCL, img
```

The following figure displays the initial image.

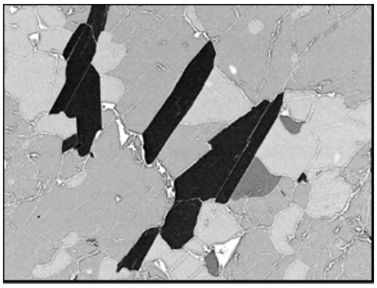


*Figure 8-6: Initial Image*

4. Create a mask that identifies the darkest pixels, whose values are less than 50:

```
threshImg = (img LT 50)
```

**Note**

See “Determining Intensity Values When Thresholding and Stretching Images” on page 486 for a useful strategy to use when determining threshold values.

5. Create and apply a 3x3 square structuring element, using the erosion and dilation operators to close gaps in the thresholded image:

```
strucElem = REPLICATE(1, 3, 3)
threshImg = ERODE(DILATE(TEMPORARY(threshImg), $
   strucElem), strucElem)
```

6. Use the CONTOUR procedure to extract the boundaries of the thresholded regions. Store the path information and coordinates of the contours in the variables *pathInfo* and *pathXY* as follows:

```
CONTOUR, threshImg, LEVEL = 1,  $
   XMARGIN = [0, 0], YMARGIN = [0, 0], $
   /NOERASE, PATH_INFO = pathInfo, PATH_XY = pathXY, $
   XSTYLE = 5, YSTYLE = 5, /PATH_DATA_COORDS
```

The PATH_INFO variable contains the path information for the contours. When used in conjunction with the PATH_XY variable, containing the coordinates of the contours, the CONTOUR procedure records the outline of closed regions. See CONTOUR in the *IDL Reference Guide* for full details.

7. Display the original image in a second window and load a discrete color table:

```
WINDOW, 2, XSIZE = dims[0], YSIXE = dims[1]
TVSCL, img
LOADCT, 12
```

8. Input the data of each of the contour paths into IDLanROI objects:

```
FOR I = 0,(N_ELEMENTS(PathInfo) - 1 ) DO BEGIN & $
```

**Note**

The & after BEGIN and the $ allow you to use the FOR/DO loop at the IDL command line. These & and $ symbols are not required when the FOR/DO loop in placed in an IDL program as shown in “Example Code: Defining an ROI and Computing ROI Statistics” on page 314.

9. Initialize *oROI* with the contour information of the current region:

```
line = [LINDGEN(PathInfo(I).N), 0] & $
oROI = OBJ_NEW('IDLanROI', $
   (pathXY(*, pathInfo(I).OFFSET + line))[0, *], $
   (pathXY(*, pathInfo(I).OFFSET + line))[1, *]) & $
```

10. Draw the ROI object in a Direct Graphics window using DRAW_ROI:

```
DRAW_ROI, oROI, COLOR = 80 & $
```

11. Use the IDLanROI::ComputeMask function in conjunction with IMAGE_STATISTICS to obtain *maskArea*, the number of pixels covered by the region when it is displayed. The variable, *maskResult*, is input as the value of MASK in the second statement in order to return the *maskArea*:

```
maskResult = oROI -> ComputeMask( $
   DIMENSIONS = [dims[0], dims[1]]) & $
IMAGE_STATISTICS, img, MASK = maskResult, $
   COUNT = maskArea & $
```

12. Use the IDLanROI::ComputeGeometry function to return the geometric area and perimeter of each region. In the following example, SPATIAL_SCALE defines that each pixel represents 1.2 by 1.2 millimeters:

```
ROIStats = oROI -> ComputeGeometry( $
   AREA = geomArea, PERIMETER = perimeter, $
   SPATIAL_SCALE = [1.2, 1.2, 1.0]) & $
```

**Note**

The value for SPATIAL _SCALE in the previous statement is used only as an example. The actual spatial scale value is typically known based upon equipment used to gather the data.

13. Print the statistics for each ROI when it is displayed and wait 3 seconds before proceeding to the display and analysis of the next region:

```
PRINT, ' ' & $
PRINT, 'Region''s mask area =   ', $
   FIX(maskArea), ' pixels' & $
PRINT, 'Region''s geometric area =   ', $
   FIX(geomArea), ' mm' & $
PRINT, 'Region''s perimeter =    ', $
   FIX(perimeter),' mm' & $
WAIT, 3
```

14. Remove each unneeded object reference after displaying the region:

```
OBJ_DESTROY, oROI & $
```

15. End the FOR loop:

```
ENDFOR
```

The outlines of the ROIs recorded by the CONTOUR function have been translated into ROI objects and displayed using DRAW_ROI. Each region's "mask area," (computed using IDLanROI::ComputeMask in conjunction with IMAGE_STATISTICS) shows the number of pixels covered by the region when it is displayed on the screen.

Each region's geometric area and perimeter, (computed using IDLanROI::ComputeGeometry's SPATIAL_SCALE keyword) results in the following geometric area and perimeter measurements in millimeters.

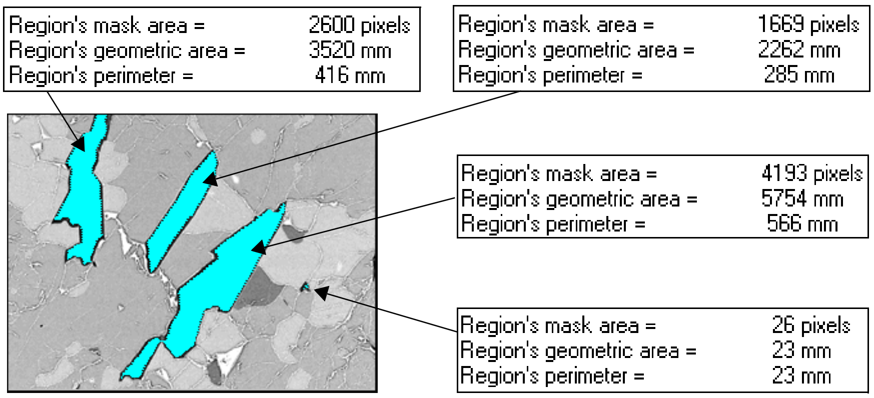

Region's mask area = 2600 pixels
Region's geometric area = 3520 mm
Region's perimeter = 416 mm
Region's mask area = 1669 pixels
Region's geometric area = 2262 mm
Region's perimeter = 285 mm
Region's mask area = 4193 pixels
Region's geometric area = 5754 mm
Region's perimeter = 566 mm
Region's mask area = 26 pixels
Region's geometric area = 23 mm
Region's perimeter = 23 mm

*Figure 8-7: Display of Programmatically Defined Regions*

## Example Code: Defining an ROI and Computing ROI Statistics

Copy and paste the following text into the IDL Editor window. After saving the file as `ProgramDefineROI.pro`, compile and run the program to reproduce the previous example.

```
PRO ProgramDefineROI

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

; Select and open the image file and get its size.
img = READ_PNG(FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data']))
dims = SIZE(img, /DIMENSIONS)
```

```
; Create a window and display the original image.
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1]
TVSCL, img, 0

; Create a mask that identifies the darkest pixels,
; whose values are less than 50.
threshImg = (img LT 50)

; Get rid of gaps, applying a 3x3 element to the image
; using the erosion and dilation morphological
; operators.
strucElem = REPLICATE(1, 3, 3)
threshImg = ERODE(DILATE(TEMPORARY(threshImg), $
   strucElem), strucElem)

; Extract the contours of the thresholded image.
CONTOUR, threshImg, LEVEL = 1,  $
   XMARGIN = [0, 0], YMARGIN = [0, 0], $
   /NOERASE, PATH_INFO = pathInfo, PATH_XY = pathXY, $
   XSTYLE = 5, YSTYLE = 5, /PATH_DATA_COORDS

; Display the original image in a second window and
; load a discrete color table.
WINDOW, 2, XSIZE = dims[0], YSIZE = dims[1]
TVSCL, img
LOADCT, 12

; For each region, feed the contours into an IDLgrROI
; object for display with DRAW_ROI.
FOR I = 0, (N_ELEMENTS(pathInfo) - 1 ) DO BEGIN

   ; Initialize the IDLgrROI object with the contour
   ; information of the current region with the FOR
   ; loop.
   line = [LINDGEN(pathInfo(I).N), 0]
   oROI = OBJ_NEW('IDLanROI', $
      (pathXY(*,pathInfo(I).OFFSET + line))[0, *], $
      (pathXY(*,pathInfo(I).OFFSET + line))[1, *])

   ; Draw each ROI defined by thresholding and
   ; contouring.
   DRAW_ROI, oROI, COLOR = 80

   ; Use ComputeMask in conjunction with
   ; IMAGE_STATISTICS to obtain the number of pixels
   ; covered by the regions when displayed.
   maskResult = oROI -> ComputeMask(DIMENSIONS = $
      [dims[0], dims[1]])
   IMAGE_STATISTICS, img, MASK = maskResult, $
      COUNT = maskArea

   ; Use ComputeGeometry to obtain the geometric area
   ; and perimeter of each region where 1 pixel =
   ; 1.2 x 1.2 mm.
   ROIStats = oROI -> ComputeGeometry($
      AREA = geomArea, PERIMETER = perimeter, $
      SPATIAL_SCALE = [1.2, 1.2, 1.0])

   ; Print the statistics of each ROI when it is
   ; displayed and wait 3 seconds before proceeding to
   ; next region.
   PRINT, ' '
   PRINT, 'Region''s mask area =', $
      FIX(maskArea), ' pixels'
   PRINT, 'Region''s geometric area =', $
      FIX(geomArea), ' mm'
   PRINT, 'Region''s perimeter = ', $
      FIX(perimeter), ' mm'
   WAIT, 3

   ; Remove each unneeded object reference after
   ; displaying it.
   OBJ_DESTROY, oROI

; End the FOR loop.
ENDFOR

END
```

## Growing a Region

The REGION_GROW function is an analysis routine that allows you to identify a complicated region without having to manually draw intricate boundaries. This function expands a given region based upon the constraints imposed by either a threshold range (minimum and maximum pixel values) or by a multiplier of the standard deviation of the original region. REGION_GROW expands an original region to include all connected neighboring pixels that fall within the specified limits.

The following example interactively defines an initial region within a cross-section of a human skull. The initial region is then expanded using both methods of region expansion, thresholding and standard deviation multiplication.

For code that you can copy and paste into an Editor window, see "Example Code: Growing an ROI" on page 322 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load a grayscale color table:

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

2. Select the file, read in the data and get the image dimensions:

```
file = FILEPATH('md1107g8a.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, img, /GRAYSCALE
dims = SIZE(img, /DIMENSIONS)
```

3. Double the size of the image for display purposes and compute the new dimensions:

```
img = REBIN(BYTSCL(img), dims[0]*2, dims[1]*2)
dims = 2*dims
```

4. Create a window and display the original image:

```
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1], $
   TITLE = 'Click on Image to Select Point of ROI'
TVSCL, img
```

The following figure shows the initial image.

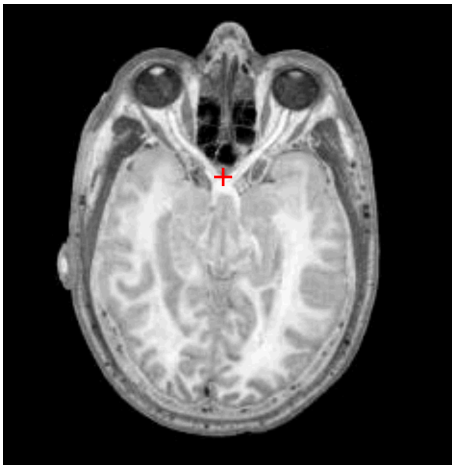


*Figure 8-8: Original Image Showing Region to be Selected*

5. Define the original region pixels. Using the CURSOR function, select the original region by positioning your cursor over the image and clicking on the region indicated in the previous figure by the “+” symbol. Then create a 10 by 10 square ROI, named *roipixels*, at the selected x, y, coordinates:

```
CURSOR, xi, yi, /DEVICE
x = LINDGEN(10*10) MOD 10 + xi
y = LINDGEN(10*10) / 10 + yi
roiPixels = x + y * dims[0]
```

**Note**

A region can also be defined and grown using the XROI utility. See the XROI procedure in the *IDL Reference Guide* for more information.

6. Delete the window after selecting the point:

```
WDELETE, 0
```

7. Set the topmost color table entry to red:

```
topClr = !D.TABLE_SIZE - 1
TVLCT, 255, 0, 0, topClr
```

8. Display the initial region using the previously defined color:

```
regionPts = BYTSCL(img, TOP = (topClr - 1))
regionPts[roiPixels] = topClr
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1], $
   TITLE = 'Original Region'
TV, regionPts
```

The following figure shows the initial ROI that will be input and expanded with the REGION_GROW function.

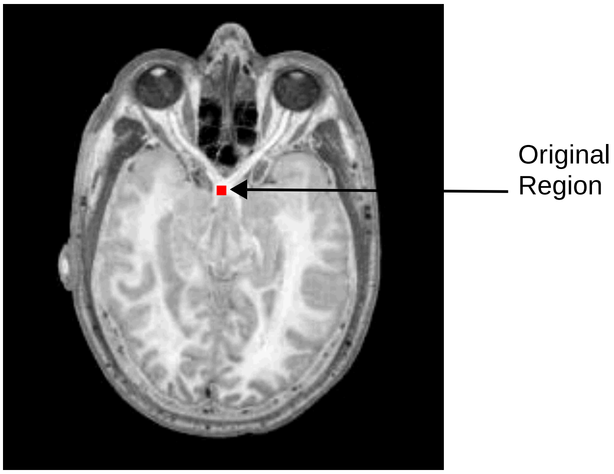


*Figure 8-9: Square ROI at Selected Coordinates*

9. Using the REGION_GROW function syntax,

```
Result = REGION_GROW(Array, ROIPixels [, /ALL_NEIGHBORS]
[, STDDEV_MULTIPLIER=value | THRESHOLD=[min,max]] )
```

input the original region, *roipixels*, and expand the region to include all connected pixels which fall within the specified THRESHOLD range:

```
newROIPixels = REGION_GROW(img, roiPixels, $
   THRESHOLD = [215,255])
```

**Note**

If neither the THRESHOLD nor the STDDEV_MULTIPLIER keywords are specified, REGION_GROW automatically applies THRESHOLD, using the minimum and maximum pixels values occurring within the original region.

10. Show the results of growing the original region using threshold values:

```
regionImg = BYTSCL(img, TOP = (topClr-1))
regionImg[newROIPixels] = topClr
WINDOW, 2, XSIZE = dims[0], YSIZE = dims[1], $
   TITLE = 'THRESHOLD Grown Region'
TV, regionImg
```

**Note**

An error message such as "`Attempt to subscript REGIONIMG with NEWROIPIXELS is out of range`" indicates that the pixel values within the defined region fall outside of the minimum and maximum THRESHOLD values. Either define a region containing pixel values that occur within the threshold range or alter the minimum and maximum values.

The left-hand image in the following figure shows that the region has been expanded to clearly identify the optic nerves. Now expand the original region by specifying a standard deviation multiplier value as described in the following step.

11. Expand the original region using a value of 7 for STDDEV_MULTIPLIER:

```
stddevPixels = REGION_GROW(img, roiPixels, $
   STDDEV_MULTIPLIER = 7)
```

12. Create a new window and show the resulting ROI:

```
WINDOW, 3, XSIZE = dims[0], YSIZE = dims[1], $
   TITLE = "STDDEV_MULTIPLIER Grown Region"
regionImg2 = BYTSCL(img, TOP = (topClr - 1))
regionImg2[stddevPixels] = topClr
TV, regionImg2
```

The following figure displays the results of growing the original region using thresholding (left) and standard deviation multiplication (right).

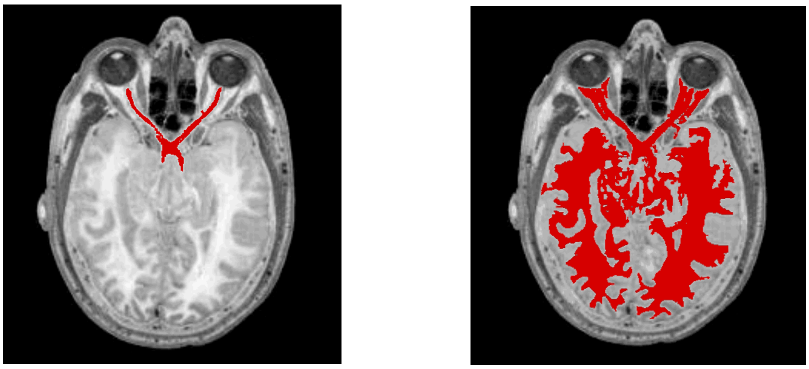


*Figure 8-10: Regions Expanded Using REGION_GROW*

**Note**

Your results for the right-hand image may differ. Results of growing a region using a standard deviation multiplier will vary according to the exact mean and deviation of the pixel values within the original region.

## Example Code: Growing an ROI

Copy and paste the following text into the IDL Editor window. After saving the file as `RegionGrowEx.pro`, compile and run the program to reproduce the previous example.

```
PRO RegionGrowEx

; Prepare the display device and load a grayscale color
; table.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 0

; Load an image and get the image dimensions.
file = FILEPATH('md1107g8a.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, img, /GRAYSCALE
dims = SIZE(img, /DIMENSIONS)

; Double the size of the image for display purposes and
; get the new dimensions.
img = REBIN(BYTSCL(img), dims[0]*2, dims[1]*2)
dims = 2*dims

; Create a window and display the image.
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1], $
   TITLE = 'Click on Image to Select Point of ROI'
TVSCL, img

; Define the original region pixels. Use the CURSOR
; function to select the region, making a 10x10 square
; at the selected x,y, coordinates.
CURSOR, xi, yi, /DEVICE
x = LINDGEN(10*10) MOD 10 + xi
y = LINDGEN(10*10) / 10 + yi
roiPixels = x + y * dims[0]

; Delete the window after selecting the point.
WDELETE, 0

; Set the topmost color table entry to red.
topClr = !D.TABLE_SIZE - 1
TVLCT, 255, 0, 0, topClr

; Scale the array, setting the maximum array value
; equal to one less than the value of topClr.
regionPts = BYTSCL(img, TOP = (topClr - 1))

; Show the results of the original region selection.
```

```
regionPts[roiPixels] = topClr
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1], $
   TITLE = 'Original Region'
TV, regionPts

; Grow the region. The THRESHOLD values are determined
; empirically.
newROIPixels = REGION_GROW(img, roiPixels, $
   THRESHOLD = [215,255])

; Show the result of the region grown using
; thresholding.
regionImg = BYTSCL(img, TOP = (topClr - 1))
regionImg[newROIPixels] = topClr
WINDOW, 2, XSIZE = dims[0], YSIZE = dims[1], $
   TITLE = 'THRESHOLD Grown Region'
TV, regionImg

; Show the results of growing the region using
; STDDEV_MULTIPLIER in a new window.
stddevPixels = REGION_GROW(img, roiPixels, $
   STDDEV_MULTIPLIER = 7)

WINDOW, 3, XSIZE = dims[0], YSIZE = dims[1], $
   TITLE = "STDDEV_MULTIPLIER Grown Region"
regionImg2 = BYTSCL(img, TOP = (topClr - 1))
regionImg2[stddevPixels] = topClr
TV, regionImg2

END
```

## Creating and Displaying an ROI Mask

The IDLanROI::ComputeMask function method defines a 2D mask of a region object, returning an array in which all pixels that lie outside of the region have a value of 0. The mask can then be used to extract the portion of the original image that lies within the ROI. The following example defines an ROI, computes a mask, applies the mask to retain only the portion of the image defined by the ROI, and produces a magnified view of the ROI. For code that you can copy and paste into an Editor window, see “Example Code: Defining an ROI Mask” on page 327 or complete the following steps for a detailed description of the process.

1. Select the file, read in the data and get the image dimensions:

```
file = FILEPATH('md5290fc1.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, img, /GRAYSCALE
dims = SIZE(img, /DIMENSIONS)
```

2. Pass the image to XROI and use the Draw Polygon tool to define the region shown in the following figure:

```
XROI, img, REGIONS_OUT = ROIout, /BLOCK
```

   Close the XROI window to save the region object data in the variable, *ROIout*.

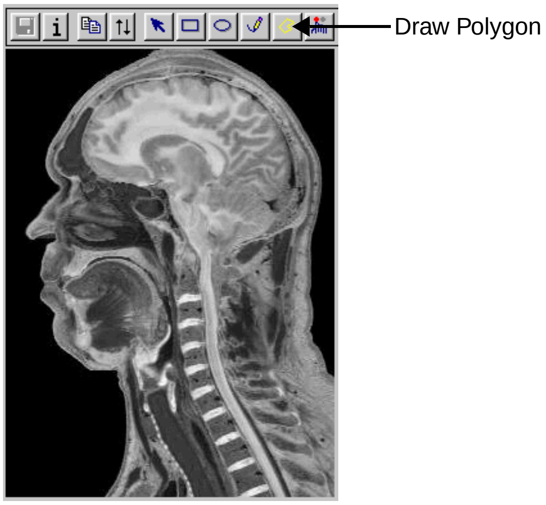


*Figure 8-11: ROI Definition in XROI*

3. Assign the ROI data to the arrays, *x* and *y*:

```
ROIout -> GetProperty, DATA = ROIdata
x = ROIdata[0,*]
y = ROIdata[1,*]
```

4. Set the properties of the ROI:

```
ROIout -> SetProperty, COLOR = [255,255,255], THICK = 2
```

5. Initialize an IDLgrImage object containing the original image data:

```
oImg = OBJ_NEW('IDLgrImage', img,$
   DIMENSIONS = dims)
```

6. Create a window in which to display the image and the ROI:

```
oWindow = OBJ_NEW('IDLgrWindow', DIMENSIONS = dims, $
   RETAIN = 2, TITLE = 'Selected ROI')
```

7. Create the view plane and initialize the view:

```
viewRect = [0, 0, dims[0], dims[1]]
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT = viewRect)
```

8. Initialize a model object and add the image and ROI to the model. Add the model to the view and draw the view in the window to display the ROI overlaid onto the original image:

```
oModel = OBJ_NEW('IDLgrModel')
oModel -> Add, oImg
oModel -> Add, ROIout
oView -> Add, oModel
oWindow -> Draw, oView
```

9. Use the IDLanROI::ComputeMask function to create a 2D mask of the region. Pixels that fall outside of the ROI will be assigned a value of 0:

```
maskResult = ROIout -> ComputeMask(DIMENSIONS = dims)
```

10. Use the IMAGE_STATISTICS procedure to compute the area of the mask, inputting *maskResult* as the MASK value. Print *count* to view the number of pixels occurring within the masked region:

```
IMAGE_STATISTICS, img, MASK = MaskResult, COUNT = count
PRINT, 'area of mask =  ', count,' pixels'
```

**Note**

The COUNT keyword to IMAGE_STATISTICS returns the number of pixels covered by the ROI when it is displayed, the same value as that shown in the "# Pixels" field of XROI's ROI Information dialog.

11. From the ROI mask, create a binary mask, consisting of only zeros and ones. Multiply the binary mask times the original image to retain only the portion of the image that was defined in the original ROI:

    ```
    mask = (maskResult GT 0)
    maskImg = img * mask
    ```

12. Using the minimum and maximum values of the ROI array, create a cropped array, *cropImg*, and get its dimensions:

    ```
    cropImg = maskImg[min(x):max(x), min(y): max(y)]
    cropDims = SIZE(cropImg, /DIMENSIONS)
    ```

13. Initialize an image object with the cropped region data:

    ```
    oMaskImg = OBJ_NEW('IDLgrImage', cropImg, $
       DIMENSIONS = dims)
    ```

14. Using the cropped region dimensions, create an offset window. Multiply the *x* and *y* dimensions times the value by which you wish to magnify the ROI:

    ```
    oMaskWindow = OBJ_NEW('IDLgrWindow', $
       DIMENSIONS = 2 * cropDims, RETAIN = 2, $
       TITLE = 'Magnified ROI', LOCATION = dims)
    ```

15. Create the display objects and display the cropped and magnified ROI:

    ```
    oMaskView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT = viewRect)
    oMaskModel = OBJ_NEW('IDLgrModel')
    oMaskModel -> Add, oMaskImg
    oMaskView -> Add, oMaskModel
    OMaskWindow -> Draw, oMaskView
    ```

The original and the magnified view of the ROI are shown in the following figure.

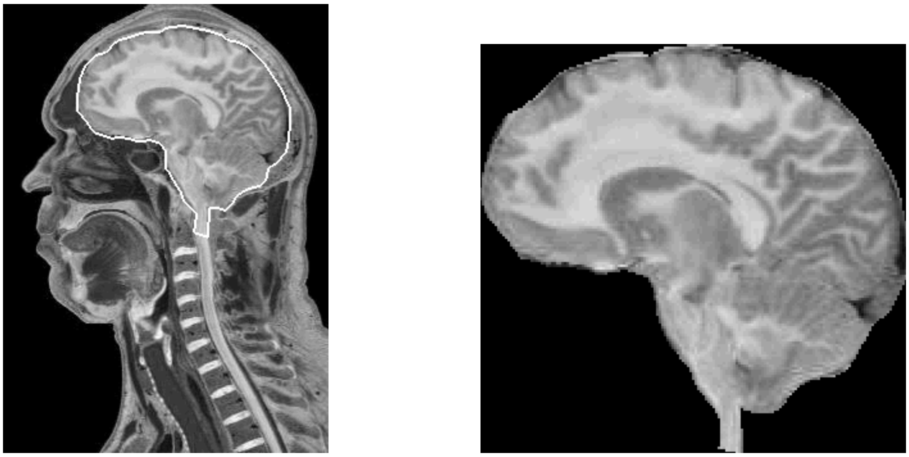


*Figure 8-12: Original and Magnified View of the ROI*

16. Clean up object references that are not destroyed by the window manager when you close the Object Graphics displays:

```
OBJ_DESTROY, [oView, oMaskView, ROIout]
```

## Example Code: Defining an ROI Mask

Copy and paste the following text into the IDL Editor window. After saving the file as `ScaleMask_object.pro`, compile and run the program to reproduce the previous example.

```
PRO ScaleMask_Object

; Select the image file and get the image dimensions.
file= FILEPATH('md5290fc1.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, img, /GRAYSCALE
dims = SIZE(img, /DIMENSIONS)

; Pass the image to XROI and define the region.
XROI, img, REGIONS_OUT = ROIout, /BLOCK

; Assign the ROI data to the arrays, x and y.
ROIout -> GetProperty, DATA = ROIdata
x = ROIdata[0,*]
y = ROIdata[1,*]
```

```
; Set the properties of the ROI.
ROIout -> SetProperty, COLOR = [255,255,255], THICK = 2

; Create the image object.
oImg = OBJ_NEW('IDLgrImage', img,$
   DIMENSIONS = dims)

; Create a window in which to display the selected ROI.
oWindow = OBJ_NEW('IDLgrWindow', DIMENSIONS = dims, $
   RETAIN = 2, TITLE = 'Selected ROI')

; Create the display objects and display the region.
viewRect = [0, 0, dims[0], dims[1]]
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT = viewRect)

oModel = OBJ_NEW('IDLgrModel')
oModel -> Add, oImg
oModel -> Add, ROIout
oView -> Add, oModel
oWindow -> Draw, oView

; Create a mask and print area of the mask.
maskResult = ROIout -> ComputeMask( $
   DIMENSIONS = dims)
IMAGE_STATISTICS, img, MASK = MaskResult, COUNT = count
PRINT, 'area of mask =  ', count,' pixels'

; Mask out all portions of the image except for the ROI.
mask = (maskResult GT 0)
maskImg = img*mask

; Create a image containing only the cropped ROI.
cropImg = maskImg[min(x):max(x), min(y): max(y)]
cropDims = SIZE(cropImg, /DIMENSIONS)
oMaskImg = OBJ_NEW('IDLgrImage', cropImg, $
   DIMENSIONS = dims)

; Create a window in which to display the cropped ROI.
; Multiply the dimensions times the value you wish to
; magnify the ROI.
oMaskWindow = OBJ_NEW('IDLgrWindow', $
   DIMENSIONS = 2 * cropDims, RETAIN = 2, $
   TITLE = 'Magnified ROI', LOCATION = dims)

; Create the display objects and display the cropped
; ROI.
oMaskView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = viewRect)
```

```
oMaskModel = OBJ_NEW('IDLgrModel')
oMaskModel -> Add, oMaskImg
oMaskView -> Add, oMaskModel
OMaskWindow -> Draw, oMaskView

; Clean up objects.
OBJ_DESTROY, [oView, oMaskView, ROIout]

END
```

## Testing an ROI for Point Containment

The IDLanROI::ContainsPoints function method determines whether a point having given coordinates lies inside, outside, on the boundary of, or on the vertex of a designated ROI. The following example allows the creation of an ROI within an image of the world using XROI. After exiting XROI, a point is selected and tested to determine its relationship to the ROI. The example then creates textual and graphical displays of the results.

For code that you can copy and paste into an Editor window, see "Example Code: Testing an ROI Object for Point Containment" on page 332 or complete the following steps for a detailed description of the process.

1. Prepare the display device:

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

2. Select and open the image file and get its dimensions:

```
img = READ_PNG(FILEPATH('avhrr.png', $
   SUBDIRECTORY = ['examples', 'data']), R, G, B)
dims = SIZE(img, /DIMENSIONS)
```

3. Open the file in the XROI utility to create an ROI:

```
XROI, img, REGIONS_OUT = ROIout, R, G, B, /BLOCK, $
   TITLE = 'Create ROI and Close Window'
```

   After creating any region using the tool of your choice, close the XROI utility to save the ROI object data in the variable, *ROIout*.

4. Load the image color table and display the image in a new window:

```
TVLCT, R, G, B
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1], $
   TITLE = 'Left-Click Anywhere in Image'
TV, img
```

5. The CURSOR function allows you to select and define the coordinates of a point. After entering the following line, position your cursor anywhere in the image window and click the left mouse button to select a point:

```
CURSOR, xi, yi, /DEVICE
```

6. Delete the window after selecting the point:

```
WDELETE, 0
```

7. Using the coordinates returned by the CURSOR function, determine the placement of the point in relation to the ROI object using IDLanROI::ContainsPoints:

```
ptTest = ROIout -> ContainsPoints(xi,yi)
```

8. The value of *ptTest*, returned by the previous statement, ranges from 0 to 3. Create the following vector of string data where the index value of the string element relates to value of *ptTest*. Print the actual and textual value of *ptTest*:

```
containResults = [ $
   'Point lies outside ROI', $
   'Point lies inside ROI', $
   'Point lies on the edge of the ROI', $
   'Point lies on vertex of the ROI']

PRINT, 'Result =',ptTest,':   ', containResults[ptTest]
```

9. Complete the following steps to create a visual display of the ROI and the point that you have defined. First, create a 7 by 7 ROI indicating the point:

```
x = LINDGEN(7*7) MOD 7 + xi
y = LINDGEN(7*7) / 7 + yi
point = x + y * dims[0]
```

10. Define the color with which the ROI and point are drawn:

```
maxClr = !D.TABLE_SIZE - 1
TVLCT, 255, 255, 255, maxClr
```

11. Draw the point within the original image and display it:

```
regionPt = img
regionPt[point] = maxClr
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1], $
   TITLE='Containment Test Results'
TV, regionPt
```

12. Draw the ROI over the image using DRAW_ROI:

```
DRAW_ROI, ROIout, COLOR = maxClr, /LINE_FILL, $
   THICK = 2, LINESTYLE = 0, ORIENTATION = 315, /DEVICE
```

13. Clean up object references that are not destroyed by the window manager:

```
OBJ_DESTROY, ROIout
```

The following figure displays a region covering South America and a point within the African continent. Your results will depend upon the ROI and point you have defined when running this program.

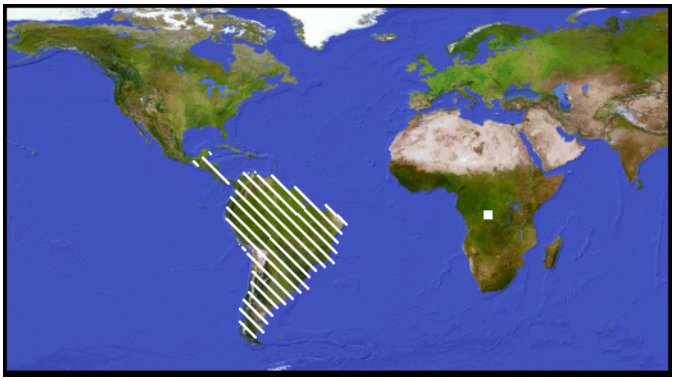


*Figure 8-13: Detail of Point Containment Test*

## Example Code: Testing an ROI Object for Point Containment

Copy and paste the following text into the IDL Editor window. After saving the file as `ContainmentTest.pro`, compile and run the program to reproduce the previous example.

```
PRO ContainmentTest

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

; Select and open the image file and get its size.
img = READ_PNG(FILEPATH('avhrr.png', $
   SUBDIRECTORY = ['examples', 'data']), R, G, B)
dims = SIZE(img, /DIMENSIONS)

; Open the file in the XROI utility to select a ROI.
XROI, img, REGIONS_OUT = ROIout, R, G, B, /BLOCK, $
   TITLE = 'Create ROI and Close Window'

; Load the image color table and display the image.
TVLCT, R, G, B
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1], $
   TITLE = 'Left-Click Anywhere in Image'
TV, img
```

```
; Select and define the coordinates of a point and
; delete window.
PRINT, 'Left-click anywhere in the image.'
CURSOR, xi, yi, /DEVICE
WDELETE, 0

; Test for point containment within the ROI
; and print result of the containment test.
ptTest = ROIout -> ContainsPoints(xi,yi)
containResults = [ $
   'Point lies outside ROI', $
   'Point lies inside ROI', $
   'Point lies on the edge of the ROI', $
   'Point lies on vertex of the ROI']

PRINT, 'Result =', ptTest, ':', $
   containResults[ptTest]

; Create a 7x7 square indicating original point.
x = LINDGEN(7*7) MOD 7 + xi
y = LINDGEN(7*7) / 7 + yi
point = x + y * dims[0]

; Define the color to use for the ROI and point.
maxClr = !D.TABLE_SIZE - 1
TVLCT, 255, 255, 255, maxClr

; Identify the point within the original image.
regionPt = img
regionPt[point] = maxClr

; Create a window and display the point and ROI.
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1], $
   TITLE='Containment Test Results'
TV, regionPt
DRAW_ROI, ROIout, COLOR = maxClr, /LINE_FILL, $
   THICK = 2, LINESTYLE = 0, ORIENTATION = 315, /DEVICE

; Destroy object references.
OBJ_DESTROY, ROIout

END
```

## Creating a Surface Mesh of an ROI Group

An IDLanROIGroup contains multiple ROIs. The ROI group consists of either several ROIs defined in a single image, or a stack of ROIs, each of which has been defined from a separate slice of a multi-image data set. An ROI group can be translated into a surface mesh, a mask, or tested for point containment. The following example defines ROIs from a data set containing 57 MRI images of a human head. After all ROIs have been defined with the utility and each region has been added to the group, IDLanROI::ComputeMesh triangulates a surface mesh. The resulting vertices and connectivity array are used to create a polygon object that is displayed using XOBJVIEW.

For code that you can copy and paste into an Editor window, see "Example Code: Creating an ROI Mesh from an ROI Group" on page 337 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load a color table to more easily distinguish image features:

```
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 5
TVLCT, R, G, B, /GET
```

2. Select and open the file:

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

3. Resize the array for display purposes and to compensate for the sampling rate of the scan slices:

```
img = CONGRID(img, 200, 225, 57)
```

4. Initialize an IDLanROIGroup object to which individual ROIs will be added:

```
oROIGroup = OBJ_NEW('IDLgrROIGroup')
```

5. Use a FOR loop to define an ROI within every fifth slice of data. Add each ROI to the group:

```
FOR i=0, 54, 5  DO BEGIN & $
   XROI, img[*, *,i], R, G, B, REGIONS_OUT = oROI, $
      /BLOCK, ROI_SELECT_COLOR = [255, 255, 255] & $
   oROI -> GetProperty, DATA = roiData & $
   roiData[2, *] = 2.2*i & $
   oRoi -> ReplaceData, roiData & $
   oRoiGroup -> Add, oRoi & $
ENDFOR
```

**Note**

The & after BEGIN and the $ allow you to use the FOR/DO loop at the IDL command line. These & and $ symbols are not required when the FOR/DO loop in placed in an IDL program as shown in "Example Code: Defining an ROI and Computing ROI Statistics" on page 314.

The following image shows samples of the ROIs to be defined.

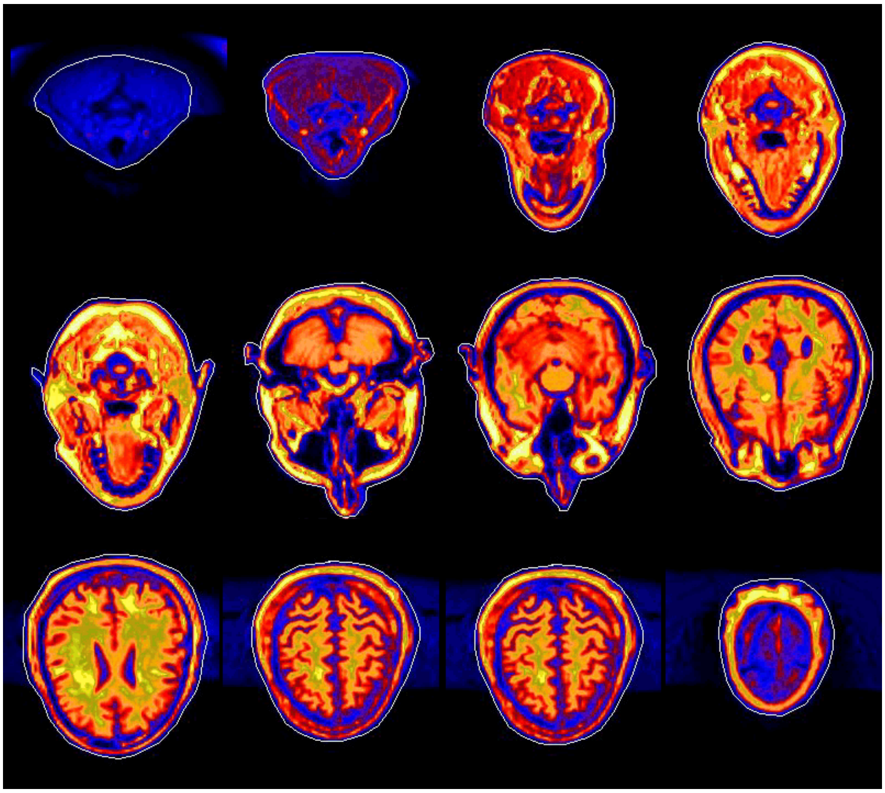


*Figure 8-14: ROIs to be Defined*

To limit the time needed complete this exercise, the previous FOR statement arranges to display every fifth slice of data for ROI selection. To obtain higher quality results, consider selecting an ROI in every other slice of data.

6. Compute the mesh for the ROI group using IDLanROIGroup::ComputeMesh:

```
result = oROIGroup -> ComputeMesh(verts, conn)
```

**Note**

The ComputeMesh function will fail if the ROIs contain interior regions (holes), are self-intersecting or are of a TYPE other than the default, closed polygon.

7. Prepare to display the mesh, scaling and translating the array for display in XOBJVIEW:

```
nImg = 57
xymax = 200.0
zmax = float(nImg)
oModel = OBJ_NEW('IDLgrModel')
oModel -> Scale, 1./xymax,1./xymax, 1.0/zmax
oModel -> Translate, -0.5, -0.5, -0.5
oModel -> Rotate, [1,0,0], -90
oModel -> Rotate, [0, 1, 0], 30
oModel -> Rotate, [1,0,0], 30
```

8. Create an IDLgrPolygon object using the results of ComputeMesh:

```
oPoly = OBJ_NEW('IDLgrPolygon', verts, POLYGON = conn, $
   COLOR = [128, 128, 128], SHADING = 1)
```

9. Add the polygon to the model and display the polygon object in XOBJVIEW:

```
oModel -> Add, oPoly
XOBJVIEW, oModel, /BLOCK
```

10. Clean up object references that are not destroyed by the window manager when you close the Object Graphics displays:

```
OBJ_DESTROY, [oROI, oROIGroup, oPoly, oModel]
```

The following figure displays the mesh created by defining an ROI in every other slice of data instead of from every fifth slice as described in this example. Therefore, your results will likely vary.

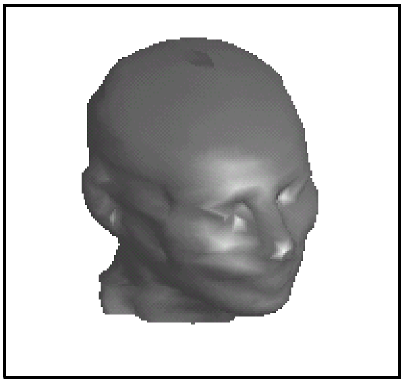


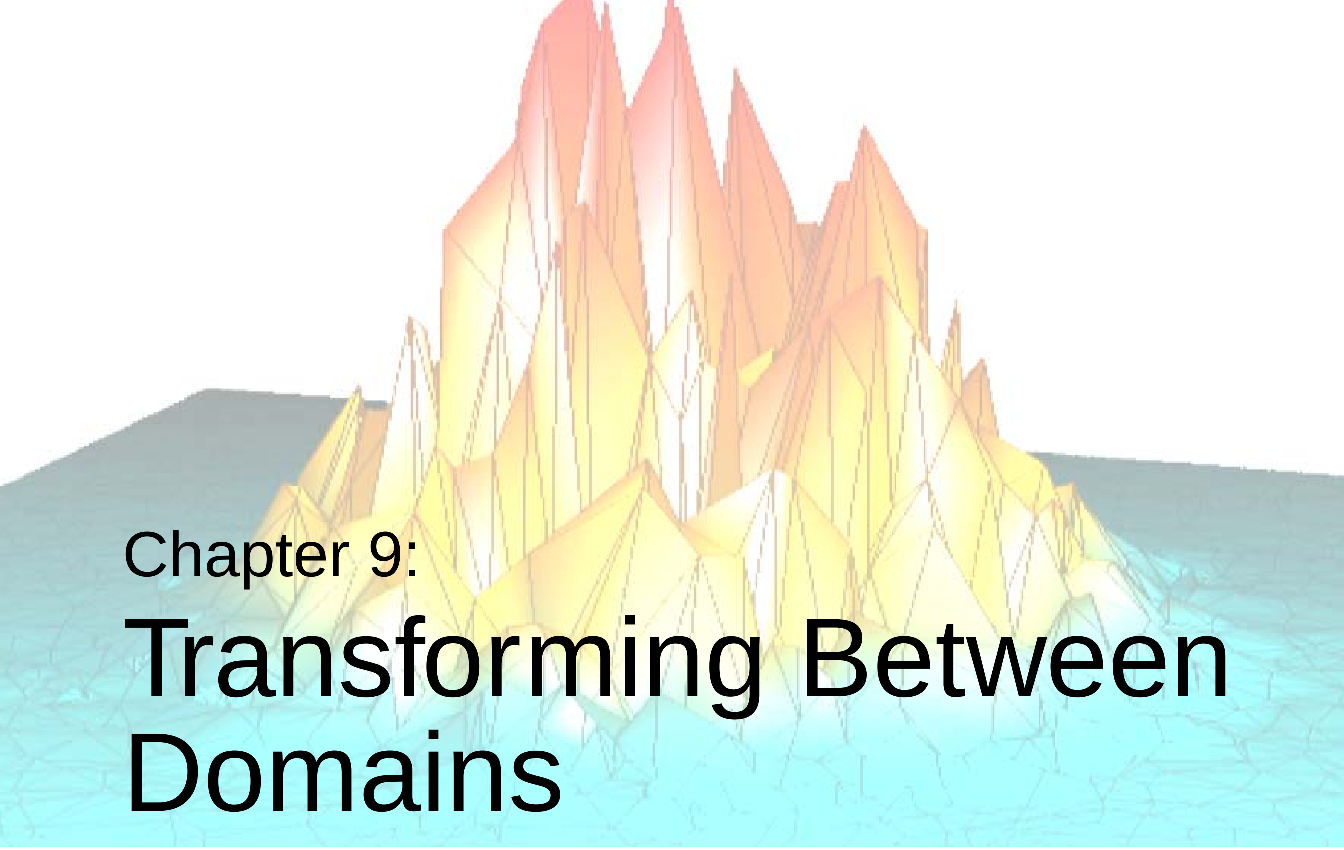
*Figure 8-15: Result of Creating a Mesh from a Group of ROIs*

## Example Code: Creating an ROI Mesh from an ROI Group

Copy and paste the following text into the IDL Editor window. After saving the file as `GroupROIMesh.pro`, compile and run the program to reproduce the previous example.

```
Pro GroupROIMesh

; Prepare the display device and load a color table
; to more easily distinguish image features.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 5
TVLCT, R, G, B, /GET

; Select and open the file.
file = FILEPATH('head.dat', $
   SUBDIRECTORY = ['examples', 'data'])
img = READ_BINARY(file, DATA_DIMS = [80,100,57])

; Resize the array for display purposes and to
; compensate for the sampling rate of the scan slices.
img = CONGRID(img, 200, 225, 57)

; Initialize a ROI group object to which individual
; ROIs will be added.
oROIGroup = OBJ_NEW('IDLgrROIGroup')
```

```
; Use a FOR loop to define ROIs with which to create
; the mesh. Add each ROI to the group.
FOR i=0, 54, 5  DO BEGIN
   XROI, img[*, *,i], R, G, B, REGIONS_OUT = oROI, $
      /BLOCK, ROI_SELECT_COLOR = [255, 255, 255]
   oROI -> GetProperty, DATA = roiData
   roiData[2, *] = 2.2*i
   oRoi -> ReplaceData, roiData
   oRoiGroup -> Add, oRoi
ENDFOR

; Compute the mesh for the group.
result = oROIGroup -> ComputeMesh(verts, conn)

; Prepare to display the mesh, scaling and translating
; the array for display in XOBJVIEW.
nImg = 57
xymax = 200.0
zmax = float(nImg)
oModel = OBJ_NEW('IDLgrModel')
oModel -> Scale, 1./xymax,1./xymax, 1.0/zmax
oModel -> Translate, -0.5, -0.5, -0.5
oModel -> Rotate, [1, 0, 0], -90
oModel -> Rotate, [0, 1, 0], 30
oModel -> Rotate, [1, 0, 0], 30

; Create a polygon object using the results of
; ComputeMesh.
oPoly = OBJ_NEW('IDLgrPolygon', verts, POLYGON = conn, $
   COLOR = [128, 128, 128], SHADING = 1)

; Add the polygon to the model and display the polygon
; object in XOBJVIEW.
oModel -> Add, oPoly
XOBJVIEW, oModel, /BLOCK

; Clean up object references.
OBJ_DESTROY, [oROI, oROIGroup, oPoly, oModel]

END
```



# Chapter 9: Transforming Between Domains

This chapter describes the following topics:

## Overview of Transforming Between Image Domains

Some processes performed on an image in the spatial domain may be very computationally expensive. These same processes may be significantly easier to perform after transforming an image to a different domain. These transformations are the basis for many image filters, applied to remove noise, to sharpen, or extract features. Domain transformations also provide additional information about an image and can offer compression benefits.

The most common representation of a pixel's value and location is spatial, where it appears in three dimensions (*x*, *y*, and *z*). Pixel value and location in this space is usually referred to by column (*x*), row (*y*), and value (*z*), and is known as the spatial domain. However, a pixel's value and location can be represented in other domains.

In the frequency or Fourier domain, the value and location are represented by sinusoidal relationships that depend upon the frequency of a pixel occurring within an image. In this domain, pixel location is represented by its *x*- and *y*-frequencies and its value is represented by an amplitude. Images can be transformed into the frequency domain to determine which pixels contain more important information and whether repeating patterns occur. See "Transforming to and from the Frequency Domain with FFT" on page 343 for more information on the frequency domain.

In the time-frequency or wavelet domain, the value and location are represented by sinusoidal relationships that only partially transform the image into the frequency domain. Like the transformation to the full frequency domain, the transformation to the time-frequency domain helps to determine the important information in an image. See "Transforming to and from the Time-Frequency Domain with Wavelets" on page 365 for more information on the time-frequency domain.

In the Hough domain, pixels are presented by sinusoidal lines. Since straight lines within an image are transformed into the Hough domain as intersecting sinusoidal lines, these intersections can be used to determine if and where straight lines occur within an image. See "Transforming to and from the Hough and Radon Domains" on page 383 for more information on the Hough domain.

In the Radon domain, a line of pixels occurring in an image is represented by a single point. This transformation is useful for detecting specific features and image compression. Since transforming images to and from the Hough and Radon domains use similar methods, the Radon image representation is described in the same section as the Hough representation. See "Transforming to and from the Hough and Radon Domains" on page 383 for more information on the Radon domain.

**Note**

In this book, Direct Graphics examples are provided by default. Object Graphics examples are provided in cases where significantly different methods are required.

The following list introduces the image domain transformations and associated IDL image transformation routines covered in this chapter.

| Task | Routine(s) | Description |
|---|---|---|
| "Transforming to and from the Frequency Domain with FFT" on page 343 | FFT | Transform images into the frequency domain and back into the spatial domain with the Fast Fourier Transform. Then show how to use this process to remove noise from an image. |
| "Transforming to and from the Time-Frequency Domain with Wavelets" on page 365 | WTN | Transform images into the time-frequency domain and back into the spatial domain with the Wavelet transform. Then show how to use this process to remove noise from an image. |

*Table 9-1: Image Transformation Tasks and Related Routines*

| Task | Routine(s) | Description |
|---|---|---|
| "Transforming to and from the Hough and Radon Domains" on page 383 | HOUGH<br>RADON | Transform images into the Hough and the Radon domains and back into the spatial domain with the Hough and Radon transforms. Then show how to use these processes to detect straight lines and improve contrast within an image. |

*Table 9-1: Image Transformation Tasks and Related Routines (Continued)*

**Note**

This chapter uses data files from the `IDL examples/data` directory. Two files, `data.txt` and `index.txt`, contain descriptions of the files, including array sizes.

# Transforming to and from the Frequency Domain with FFT

The Fast Fourier Transform (FFT) is used in numerical analysis to transform an image between spatial and frequency domains. The FFT decomposes an image into sines and cosines of varying amplitudes and phases. The values of the resulting transform represent the amplitudes of particular horizontal and vertical frequencies. This image information in the frequency domain shows how often patterns are repeated within an image. Low frequencies represent gradual variations in an image, while high frequencies correspond to abrupt variations in the image.

Low frequencies tend to contain the most information because they determine the overall shape or pattern in the image. High frequencies provide detail in the image, but they are often contaminated by the spurious effects of noise. Masks can be easily applied to the image within the frequency domain to remove the noise.

The following sections introduce the concepts needed to work with images and Fast Fourier Transforms (FFTs):

- "Transforming to the Frequency Domain"
- "Displaying Images in the Frequency Domain" on page 349
- "Transforming from the Frequency Domain" on page 354

The FFT process is the basis for many filters used in image processing. One of the easiest FFT filters to understand is the one used for background noise removal. This filter is simply a mask applied to the image in the frequency domain. See "Removing Noise with the FFT" on page 358 for an example of how to use this type of filter.

## Transforming to the Frequency Domain

When an image is transformed with FFT from the spatial domain to the frequency domain, the transformation process is referred to as a forward FFT. The forward FFT process can be performed with IDL's FFT function.

In the frequency domain, the lowest frequencies usually contain most of the information, which is shown by the large peak in the center of the data. If the transform is shown as a surface, the peak of low frequencies appears as a spike. If the transform is shown as an image, the peak of low frequencies is composed of the brightest pixels.

If the image does not contain any background noise, the rest of the data frequencies are very close to zero. However, the results of the FFT function have a very wide range. An initial display may not show any variations from zero, but a smaller range will show that the image does actually contain background noise. Since scaling a range can sometimes be quite arbitrary, different methods are used. See "Displaying Images in the Frequency Domain" on page 349 for more information on displaying the results of a forward FFT.

The following example shows how to use IDL's FFT function to compute a forward FFT. This example uses the first image within the `abnorm.dat` file in the `examples/data` directory. The results of the FFT function are shifted to move the origin (0, 0) of the *x*- and *y*-frequencies to the center of the data. Frequency magnitude then increases with distance from the origin. If the results are not centered, then the negative frequencies appear after the positive frequencies because of the storage scheme of the FFT process. See the FFT description *in the IDL Reference Guide* for more information on this storage scheme.

For code that you can copy and paste into an Editor window, see "Example Code: Transforming to the Frequency Domain" on page 347 or complete the following steps for a detailed description of the process.

1. Import the first image from the `abnorm.dat` file:

```
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Define a display size parameter to resize the image when displaying it:

```
displaySize = 2*imageSize
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

4. Create a window and display the image:

```
WINDOW, 0, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], TITLE = 'Original Image'
TVSCL, CONGRID(image, displaySize[0], $
   displaySize[1])
```

The following figure shows the original image.

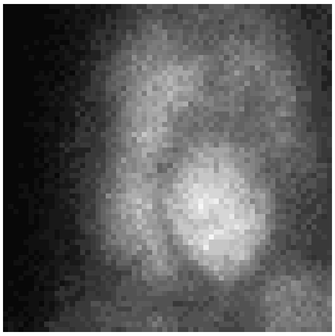


*Figure 9-1: Original Gated Blood Pool Image*

5. With the FFT function, transform the image into the frequency domain:

```
ffTransform = FFT(image)
```

6. Shift the zero frequency location from (0, 0) to the center of the display:

```
center = imageSize/2 + 1
fftShifted = SHIFT(ffTransform, center)
```

7. Calculate the horizontal and vertical frequency values, which will be used as the values for the display axes.

```
interval = 1.
hFrequency = INDGEN(imageSize[0])
hFrequency[center[0]] = center[0] - imageSize[0] + $
   FINDGEN(center[0] - 2)
hFrequency = hFrequency/(imageSize[0]/interval)
hFreqShifted = SHIFT(hFrequency, -center[0])
vFrequency = INDGEN(imageSize[1])
vFrequency[center[1]] = center[1] - imageSize[1] + $
   FINDGEN(center[1] - 2)
vFrequency = vFrequency/(imageSize[1]/interval)
vFreqShifted = SHIFT(vFrequency, -center[1])
```

**Note**

The previous two steps were performed because of the storage scheme of the FFT process. See the FFT description *in the IDL Reference Guide* for more information on this storage scheme.

8. Create another window and display the frequency transform:

```
WINDOW, 1, TITLE = 'FFT: Transform'
SHADE_SURF, fftShifted, hFreqShifted, vFreqShifted, $
   /XSTYLE, /YSTYLE, /ZSTYLE, $
   TITLE = 'Transform of Image', $
   XTITLE = 'Horizontal Frequency', $
   YTITLE = 'Vertical Frequency', $
   ZTITLE = 'Real Part of Transform', CHARSIZE = 1.5
```

The following figure shows the results of applying the FFT to the image. The data at the high frequencies seem to be close to zero, but the peak (spike) along the *z*-axis is so large that a closer look is needed.

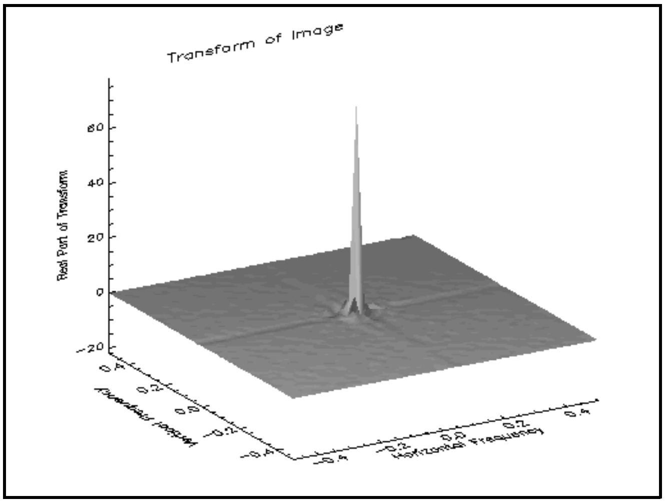


*Figure 9-2: FFT of the Gated Blood Pool Image*

**Note**

The data type of the array returned by the FFT function is complex, which contains real and imaginary parts. The amplitude is the absolute value of the FFT, while the phase is the angle of the complex number, computed using the arctangent. In the above surface, we are only displaying the real part. In most cases, the imaginary part will look the same as the real part.

9. Create another window and display the frequency transform with a data (*z*) range of 0 to 5:

```
WINDOW, 2, TITLE = 'FFT: Transform (Closer Look)'
SHADE_SURF,fftShifted, hFreqShifted, vFreqShifted, $
   /XSTYLE, /YSTYLE, /ZSTYLE, $
   TITLE = 'Transform of Image', $
   XTITLE = 'Horizontal Frequency', $
   YTITLE = 'Vertical Frequency', $
   ZTITLE = 'Real Part of Transform', CHARSIZE = 1.5, $
   ZRANGE = [0., 5.]
```

The following figure shows the resulting transform after scaling the *z*-axis range from 0 to 5. You can now see that the central peak is surrounded by smaller peaks containing both high frequency information and noise.

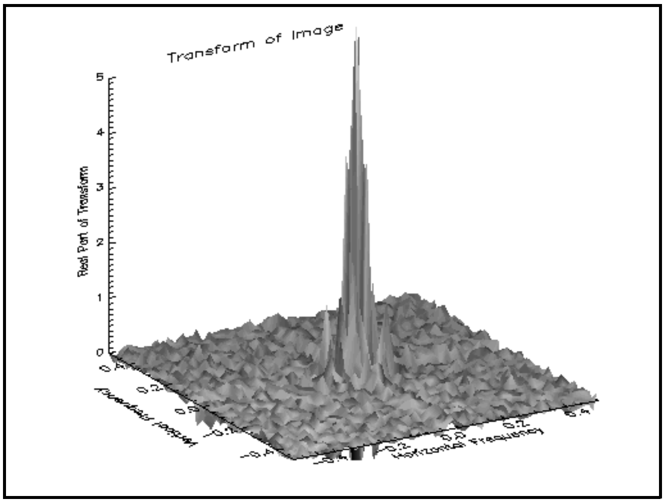


*Figure 9-3: FFT of the Gated Blood Pool Image Scaled Between 0 and 5*

## Example Code: Transforming to the Frequency Domain

Copy and paste the following text into an IDL Editor window. After saving the file as `ForwardFFT.pro`, compile and run the program to reproduce the previous example.

```
PRO ForwardFFT

; Import the image from the file.
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize a display size parameter to resize the
; image when displaying it.
displaySize = 2*imageSize

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the image.
WINDOW, 0, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], TITLE = 'Original Image'
TVSCL, CONGRID(image, displaySize[0], $
   displaySize[1])

; Transform the image into the frequency domain.
ffTransform = FFT(image)

; Shift the zero frequency location from (0, 0) to
; the center of the display.
center = imageSize/2 + 1
fftShifted = SHIFT(ffTransform, center)

; Calculate the horizontal and vertical frequency
; values, which will be used as the values for the
; axes of the display.
interval = 1.
hFrequency = INDGEN(imageSize[0])
hFrequency[center[0]] = center[0] - imageSize[0] + $
   FINDGEN(center[0] - 2)
hFrequency = hFrequency/(imageSize[0]/interval)
hFreqShifted = SHIFT(hFrequency, -center[0])
vFrequency = INDGEN(imageSize[1])
vFrequency[center[1]] = center[1] - imageSize[1] + $
   FINDGEN(center[1] - 2)
vFrequency = vFrequency/(imageSize[1]/interval)
vFreqShifted = SHIFT(vFrequency, -center[1])

; Create another window and display the frequency
; transform.
WINDOW, 1, TITLE = 'FFT: Transform'
SHADE_SURF, fftShifted, hFreqShifted, vFreqShifted, $
   /XSTYLE, /YSTYLE, /ZSTYLE, $
   TITLE = 'Transform of Image', $
   XTITLE = 'Horizontal Frequency', $
   YTITLE = 'Vertical Frequency', $
   ZTITLE = 'Amplitude', CHARSIZE = 1.5
```

```
; Create another window and display the frequency
; transform within the data (z) range of 0 to 5.
WINDOW, 2, TITLE = 'FFT: Transform (Closer Look)'
SHADE_SURF, fftShifted, hFreqShifted, vFreqShifted, $
   /XSTYLE, /YSTYLE, /ZSTYLE, $
   TITLE = 'Transform of Image', $
   XTITLE = 'Horizontal Frequency', $
   YTITLE = 'Vertical Frequency', $
   ZTITLE = 'Amplitude', CHARSIZE = 1.5, $
   ZRANGE = [0., 5.]

END
```

## Displaying Images in the Frequency Domain

Within the frequency domain, the range of values from the peak to the high frequency noise is extreme. You can use a logarithmic scale to retain the shape of the surface, but reduce its range. Since the logarithmic scale only applies to positive values, you should first compute the power spectrum, which is the absolute value squared of the transform.

The following example shows how to display the results of IDL's FFT function. This example also uses the first image within the `abnorm.dat` file in the `examples/data` directory. The results of the transform are shifted to move the origin (0, 0) of the horizontal and vertical frequencies to the center of the display. If the results are not centered then the negative frequencies appear after the positive frequencies because of the storage scheme of the FFT process. See FFT for more information on its storage scheme.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Images in the Frequency Domain" on page 352 or complete the following steps for a detailed description of the process.

1. Import the first image from the `abnorm.dat` file:

   ```
   imageSize = [64, 64]
   file = FILEPATH('abnorm.dat', $
      SUBDIRECTORY = ['examples', 'data'])
   image = READ_BINARY(file, DATA_DIMS = imageSize)
   ```

2. Initialize a display size parameter to resize the image when displaying it:

   ```
   displaySize = 2*imageSize
   ```

3. Initialize the display:

   ```
   DEVICE, DECOMPOSED = 0
   LOADCT, 0
   ```

4. Create a window and display the image:

```
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Original Image'
TVSCL, CONGRID(image, displaySize[0], $
   displaySize[1])
```

The following figure shows the original image.

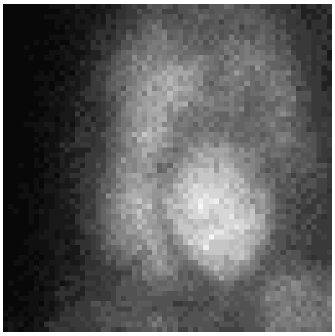


*Figure 9-4: Original Gated Blood Pool Image*

5. Transform the image into the frequency domain:

```
ffTransform = FFT(image)
```

6. Shift the zero frequency location from (0, 0) to the center of the display:

```
center = imageSize/2 + 1
fftShifted = SHIFT(ffTransform, center)
```

7. Calculate the horizontal and vertical frequency values, which will be used as the values for the display axes.

```
interval = 1.
hFrequency = INDGEN(imageSize[0])
hFrequency[center[0]] = center[0] - imageSize[0] + $
   FINDGEN(center[0] - 2)
hFrequency = hFrequency/(imageSize[0]/interval)
hFreqShifted = SHIFT(hFrequency, -center[0])
vFrequency = INDGEN(imageSize[1])
vFrequency[center[1]] = center[1] - imageSize[1] + $
   FINDGEN(center[1] - 2)
vFrequency = vFrequency/(imageSize[1]/interval)
vFreqShifted = SHIFT(vFrequency, -center[1])
```

**Note**

The previous two steps were performed because of the storage scheme of the FFT process. See the FFT description *in the IDL Reference Guide* for more information on this storage scheme.

8. Compute the power spectrum of the transform:

```
powerSpectrum = ABS(fftShifted)^2
```

9. Apply a logarithmic scale to values of the power spectrum:

```
scaledPowerSpect = ALOG10(powerSpectrum)
```

10. Create another window and display the power spectrum as a surface:

```
WINDOW, 1, TITLE = 'FFT Power Spectrum: '+ $
   'Logarithmic Scale (surface)'
SHADE_SURF, scaledPowerSpect, hFreqShifted, vFreqShifted, $
   /XSTYLE, /YSTYLE, /ZSTYLE, $
   TITLE = 'Log-scaled Power Spectrum', $
   XTITLE = 'Horizontal Frequency', $
   YTITLE = 'Vertical Frequency', $
   ZTITLE = 'Log(Squared Amplitude)', CHARSIZE = 1.5
```

The following figure shows the log-scaled power spectrum as a surface. Both low and high frequency information are visible in this display.

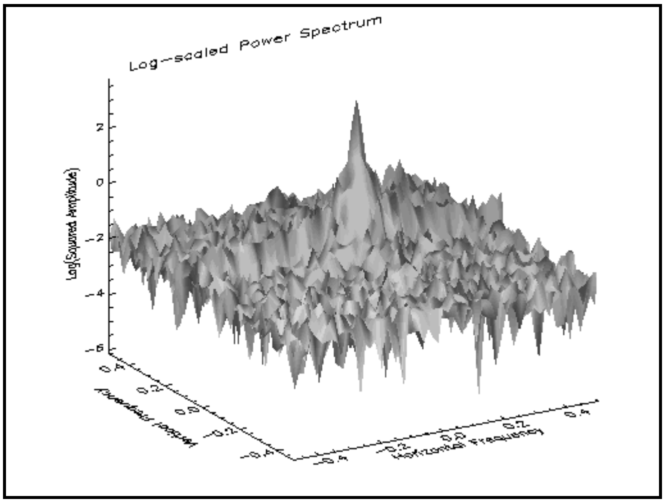


*Figure 9-5: Log-scaled FFT Power Spectrum of Image (as a surface)*

**Note**

The data type of the array returned by the FFT function is complex, which contains real and imaginary parts. The amplitude is the absolute value of the FFT, while the phase is the angle of the complex number, computed using the arctangent. In the above surface, we are only displaying the real part. In most cases, the imaginary part will look the same as the real part.

11. Create another window and display the log-scaled transform as an image:

```
WINDOW, 2, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'FFT Power Spectrum: Logarithmic Scale (image)'
TVSCL, CONGRID(scaledPowerSpect, displaySize[0], $
   displaySize[1])
```

The following figure shows the log-scaled power spectrum as an image. The brighter pixels near the center of the display represent the low frequency peak of information-containing data. The noise appears as random darker pixels within the image.

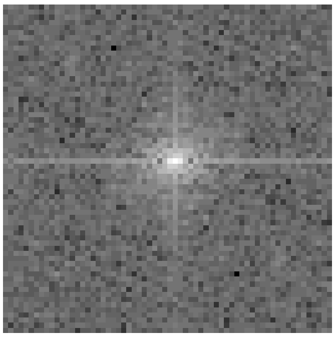


*Figure 9-6: Log-scaled FFT Power Spectrum of Image (as an image)*

## Example Code: Displaying Images in the Frequency Domain

Copy and paste the following text into an IDL Editor window. After saving the file as `DisplayFFT.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayFFT

; Import the image from the file.
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

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

; Initialize a display size parameter to resize the
; image when displaying it.
displaySize = 2*imageSize

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the image.
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Original Image'
TVSCL, CONGRID(image, displaySize[0], $
   displaySize[1])

; Transform the image into the frequency domain.
ffTransform = FFT(image)

; Shift the zero frequency location from (0, 0) to
; the center of the display.
center = imageSize/2 + 1
fftShifted = SHIFT(ffTransform, center)

; Calculate the horizontal and vertical frequency
; values, which will be used as the values for the
; axes of the display.
interval = 1.
hFrequency = INDGEN(imageSize[0])
hFrequency[center[0]] = center[0] - imageSize[0] + $
   FINDGEN(center[0] - 2)
hFrequency = hFrequency/(imageSize[0]/interval)
hFreqShifted = SHIFT(hFrequency, -center[0])
vFrequency = INDGEN(imageSize[1])
vFrequency[center[1]] = center[1] - imageSize[1] + $
   FINDGEN(center[1] - 2)
vFrequency = vFrequency/(imageSize[1]/interval)
vFreqShifted = SHIFT(vFrequency, -center[1])

; Compute the power spectrum of the transform.
powerSpectrum = ABS(fftShifted)^2

; Apply a logarithmic scale to the power spectrum.
scaledPowerSpect = ALOG10(powerSpectrum)

; Create another window and display the log-scaled
; power spectrum as a surface.
WINDOW, 1, TITLE = 'FFT Power Spectrum: ' + $
   'Logarithmic Scale (surface)'
```

```
SHADE_SURF, scaledPowerSpect, hFreqShifted, vFreqShifted, $
   /XSTYLE, /YSTYLE, /ZSTYLE, $
   TITLE = 'Log-scaled Power Spectrum', $
   XTITLE = 'Horizontal Frequency', $
   YTITLE = 'Vertical Frequency', $
   ZTITLE = 'Log(Abs(Amplitude^2))', CHARSIZE = 1.5

; Create another window and display the log-scaled
; power spectrum as an image.
WINDOW, 2, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'FFT Power Spectrum: Logarithmic Scale (image)'
TVSCL, CONGRID(scaledPowerSpect, displaySize[0], $
   displaySize[1])

END
```

## Transforming from the Frequency Domain

After manipulating an image within the frequency domain, you will need to transform the image back to the spatial domain. This transformation process is referred to as an inverse FFT. The inverse FFT process can be performed with IDL's FFT function by setting the INVERSE keyword.

The following example shows how to use IDL's FFT function to compute an inverse FFT. This example uses the first image within the `abnorm.dat` file in the `examples/data` directory. The image is not manipulated in this example while it is in the frequency domain to show that no information is lost when using the FFT. However, manipulating spurious high frequency data within the frequency domain is a useful way to remove background noise from an image, as shown in "Removing Noise with the FFT" on page 358.

For code that you can copy and paste into an Editor window, see "Example Code: Transforming from the Frequency Domain" on page 356 or complete the following steps for a detailed description of the process.

1. Import the first image from the `abnorm.dat` file:

```
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Initialize a display size parameter to resize the image when displaying it:

```
displaySize = 2*imageSize
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

4. With the FFT function, transform the image into the frequency domain:

```
ffTransform = FFT(image)
```

5. Shift the zero frequency location from (0, 0) to the center of the display:

```
center = imageSize/2 + 1
fftShifted = SHIFT(ffTransform, center)
```

**Note**

This step was performed because of the storage scheme of the FFT process. See the FFT description *in the IDL Reference Guide* for more information on this storage scheme.

6. Compute the power spectrum of the transform:

```
powerSpectrum = ABS(fftShifted)^2
```

7. Apply a logarithmic scale to values of the power spectrum:

```
scaledPowerSpect = ALOG10(powerSpectrum)
```

8. Create a window and display the power spectrum as an image:

```
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Power Spectrum Image'
TVSCL, CONGRID(scaledPowerSpect, displaySize[0], $
   displaySize[1])
```

The following figure shows the log-scaled power spectrum.

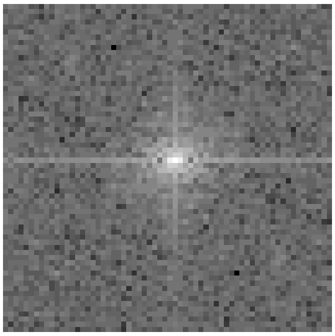


*Figure 9-7: Log-scaled FFT Power Spectrum of the Gated Blood Pool Image*

9. With the FFT function, transform the frequency domain data back to the original image (obtain the inverse transform):

```
fftInverse = REAL_PART(FFT(ffTransform, /INVERSE))
```

**Note**

The data type of the array returned by the FFT function is complex, which contains real and imaginary parts. The amplitude is the absolute value of the FFT, while the phase is the angle of the complex number, computed using the arctangent. In the above surface, we are only displaying the real part. In most cases, the imaginary part will look the same as the real part.

10. Create another window and display the inverse transform as an image:

```
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'FFT: Inverse Transform'
TVSCL, CONGRID(fftInverse, displaySize[0], $
   displaySize[1])
```

The inverse transform is the same as the original image as shown in the following figure. Unlike some domain transformations, all image information is retained when transforming data to and from the frequency domain.

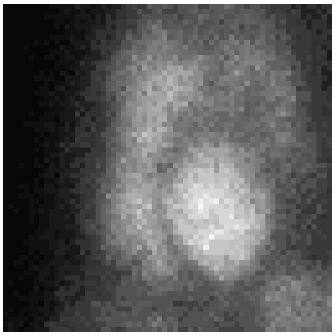


*Figure 9-8: Inverse FFT of the Gated Blood Pool Image*

## Example Code: Transforming from the Frequency Domain

Copy and paste the following text into an IDL Editor window. After saving the file as `InverseFFT.pro`, compile and run the program to reproduce the previous example.

```
PRO InverseFFT

; Import the image from the file.
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize a display size parameter to resize the
; image when displaying it.
displaySize = 2*imageSize

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Transform the image into the frequency domain.
ffTransform = FFT(image)

; Shift the zero frequency location from (0, 0) to
; the center of the display.
center = imageSize/2 + 1
fftShifted = SHIFT(ffTransform, center)

; Compute the power spectrum of the transform.
powerSpectrum = ABS(fftShifted)^2

; Apply a logarithmic scale to the power spectrum.
scaledPowerSpect = ALOG10(powerSpectrum)

; Create a window and display the power spectrum.
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Power Spectrum Image'
TVSCL, CONGRID(scaledPowerSpect, displaySize[0], $
   displaySize[1])

; Compute the inverse
fftInverse = REAL_PART(FFT(ffTransform, /INVERSE))

; Create another window and display the inverse
; transform as an image
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'FFT: Inverse Transform'
TVSCL, CONGRID(fftInverse, displaySize[0], $
   displaySize[1])

END
```

## Removing Noise with the FFT

This example uses IDL's FFT function to remove noise from an image. The image comes from the `abnorm.dat` file found in the `examples/data` directory. The first display contains the original image and its transform. The noise is very evident in the transform. A surface representation of the power spectrum helps to determine the threshold necessary to remove the noise from the image. In the surface representation, the noise appears random and below a ridge containing the spike. The ridge and spike represent coherent information within the image. A mask is applied to the transform to remove the noise and the inverse transform is applied, resulting in a clearer image.

For code that you can copy and paste into an Editor window, see "Example Code: Removing Noise with the FFT" on page 362 or complete the following steps for a detailed description of the process.

1. Import the first image from the `abnorm.dat` file:

```
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Initialize a display size parameter to resize the image when displaying it:

```
displaySize = 2*imageSize
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

4. Create a window and display the original image

```
WINDOW, 0, XSIZE = 2*displaySize[0], $
   YSIZE = displaySize[1], $
   TITLE = 'Original Image and Power Spectrum'
TVSCL, CONGRID(image, displaySize[0], displaySize[1]), 0
```

5. Transform the image into the frequency domain:

```
ffTransform = FFT(image)
```

6. Shift the zero frequency location from (0, 0) to the center of the display:

```
center = imageSize/2 + 1
fftShifted = SHIFT(ffTransform, center)
```

7. Calculate the horizontal and vertical frequency values, which will be used as the values for the axes of the display.

```
interval = 1.
hFrequency = INDGEN(imageSize[0])
hFrequency[center[0]] = center[0] - imageSize[0] + $
   FINDGEN(center[0] - 2)
hFrequency = hFrequency/(imageSize[0]/interval)
hFreqShifted = SHIFT(hFrequency, -center[0])
vFrequency = INDGEN(imageSize[1])
vFrequency[center[1]] = center[1] - imageSize[1] + $
   FINDGEN(center[1] - 2)
vFrequency = vFrequency/(imageSize[1]/interval)
vFreqShifted = SHIFT(vFrequency, -center[1])
```

**Note**

The previous two steps were performed because of the storage scheme of the FFT process. See the FFT description *in the IDL Reference Guide* for more information on this storage scheme.

8. Compute the power spectrum of the transform:

```
powerSpectrum = ABS(fftShifted)^2
```

9. Apply a logarithmic scale to values of the power spectrum:

```
scaledPowerSpect = ALOG10(powerSpectrum)
```

10. Display the log-scaled power spectrum:

```
TVSCL, CONGRID(scaledPowerSpect, displaySize[0], $
   displaySize[1]), 1
```

The following figure shows the original image and its log-scaled power spectrum. The black pixels (which appear random) in the power spectrum represent noise.

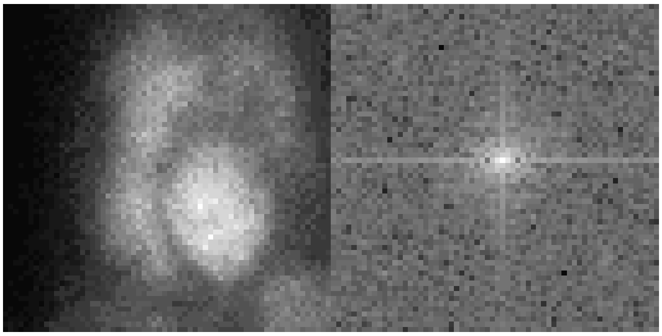


*Figure 9-9: Original Image and Its FFT Power Spectrum*

11. Scale the power spectrum to make its maximum value equal to zero:

```
scaledPS0 = scaledPowerSpect - MAX(scaledPowerSpect)
```

12. Create another window and display the scaled transform as a surface:

```
WINDOW, 1, $
   TITLE = 'Power Spectrum Scaled to a Zero Maximum'
SHADE_SURF, scaledPS0, hFreqShifted, vFreqShifted, $
   /XSTYLE, /YSTYLE, /ZSTYLE, $
   TITLE = 'Zero Maximum Power Spectrum', $
   XTITLE = 'Horizontal Frequency', $
   YTITLE = 'Vertical Frequency', $
   ZTITLE = 'Max-Scaled(Log(Power Spectrum))', $
   CHARSIZE = 1.5
```

The following figure shows the resulting log-scaled power spectrum as a surface.

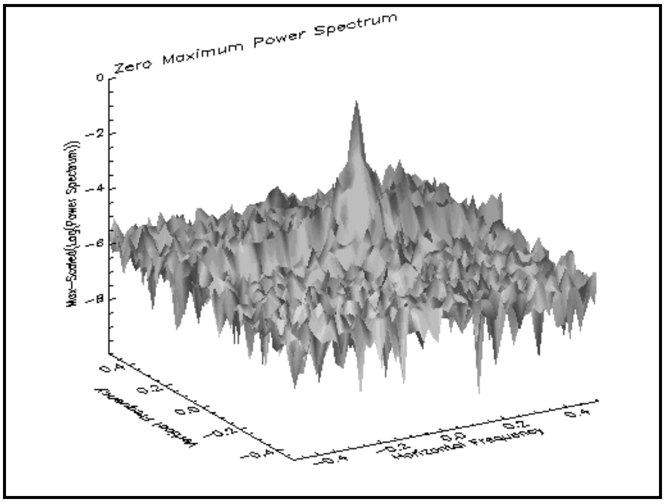


*Figure 9-10: FFT Power Spectrum of the Image Scaled to a Zero Maximum*

**Note**

The data type of the array returned by the FFT function is complex, which contains real and imaginary parts. The real part is the amplitude, and the imaginary part is the phase. In image processing, we are more concerned with the amplitude, which is the only part represented in the surface and displays of the results of the transformation. However, the imaginary part is retained for the inverse transform back into the spatial domain.

13. Threshold the image at a value of -5.25, which is just below the peak of the power spectrum, to remove the noise:

```
mask = REAL_PART(scaledPS0) GT -5.25
```

14. Apply the mask to the transform to exclude the noise:

```
maskedTransform = fftShifted*mask
```

15. Create another window and display the power spectrum of the masked transform:

```
WINDOW, 2, XSIZE = 2*displaySize[0], $
   YSIZE = displaySize[1], $
   TITLE = 'Power Spectrum of Masked Transform and Results'
TVSCL, CONGRID(ALOG10(ABS(maskedTransform^2)), $
   displaySize[0], displaySize[1]), 0, /NAN
```

16. Shift the masked transform to the position of the original transform:

```
maskedShiftedTrans = SHIFT(maskedTransform, -center)
```

17. Apply the inverse transformation to the masked transform:

```
inverseTransform = REAL_PART(FFT(maskedShiftedTrans, $
   /INVERSE))
```

18. Display the results of the inverse transformation:

```
TVSCL, CONGRID(inverseTransform, displaySize[0], $
   displaySize[1]), 1
```

The following figure shows the power spectrum of the masked transform and its inverse, which contains less noise than the original image.

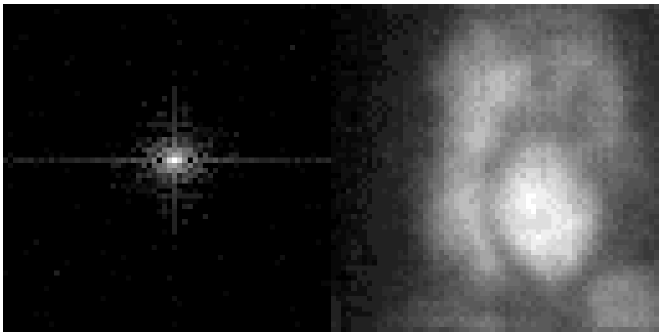


*Figure 9-11: Masked FFT Power Spectrum and Resulting Inverse Transform*

## Example Code: Removing Noise with the FFT

Copy and paste the following text into an IDL Editor window. After saving the file as `RemovingNoiseWithFFT.pro`, compile and run the program to reproduce the previous example.

```
PRO RemovingNoiseWithFFT

; Import the image from the file.
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize a display size parameter to resize the
; image when displaying it.
displaySize = 2*imageSize

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the original image.
WINDOW, 0, XSIZE = 2*displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Original Image and Power Spectrum'
TVSCL, CONGRID(image, displaySize[0], displaySize[1]), 0

; Transform the image into the frequency domain.
ffTransform = FFT(image)

; Shift the zero frequency location from (0, 0) to
; the center of the display.
center = imageSize/2 + 1
fftShifted = SHIFT(ffTransform, center)

; Calculate the horizontal and vertical frequency
; values, which will be used as the values for the
; axes of the display.
interval = 1.
hFrequency = INDGEN(imageSize[0])
hFrequency[center[0]] = center[0] - imageSize[0] + $
   FINDGEN(center[0] - 2)
hFrequency = hFrequency/(imageSize[0]/interval)
hFreqShifted = SHIFT(hFrequency, -center[0])
vFrequency = INDGEN(imageSize[1])
vFrequency[center[1]] = center[1] - imageSize[1] + $
   FINDGEN(center[1] - 2)
vFrequency = vFrequency/(imageSize[1]/interval)
vFreqShifted = SHIFT(vFrequency, -center[1])

; Compute the power spectrum of the transform.
powerSpectrum = ABS(fftShifted)^2

; Apply a logarithmic scale to the power spectrum.
scaledPowerSpect = ALOG10(powerSpectrum)
```

```
; Display the log-scaled power spectrum.
TVSCL, CONGRID(scaledPowerSpect, displaySize[0], $
   displaySize[1]), 1

; Scale the power spectrum to a zero maximum.
scaledPS0 = scaledPowerSpect - MAX(scaledPowerSpect)

; Create another window and display the scaled transform
; as a surface.
WINDOW, 1, $
   TITLE = 'Power Spectrum Scaled to a Zero Maximum'
SHADE_SURF, scaledPS0, hFreqShifted, vFreqShifted, $
   /XSTYLE, /YSTYLE, /ZSTYLE, $
   TITLE = 'Zero Maximum Power Spectrum', $
   XTITLE = 'Horizontal Frequency', $
   YTITLE = 'Vertical Frequency', $
   ZTITLE = 'Max-Scaled(Log(Power Spectrum))', $
   CHARSIZE = 1.5

; Threshold the image using -5.25, which is just below
; the peak of the transform, to remove the noise.
mask = REAL_PART(scaledPS0) GT -5.25

; Mask the transform to exclude the noise.
maskedTransform = fftShifted*mask

; Create another window and display the power spectrum
; of the masked transform.
WINDOW, 2, XSIZE = 2*displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Power Spectrum of Masked Transform and Results'
TVSCL, CONGRID(ALOG10(ABS(maskedTransform^2)), $
   displaySize[0], displaySize[1]), 0, /NAN

; Shift the masked transform to the position of the
; original transform.
maskedShiftedTrans = SHIFT(maskedTransform, -center)

; Apply the inverse transformation to masked transform.
inverseTransform = REAL_PART(FFT(maskedShiftedTrans, $
   /INVERSE))

; Display results of inverse transformation.
TVSCL, CONGRID(inverseTransform, displaySize[0], $
   displaySize[1]), 1

END
```

## Transforming to and from the Time-Frequency Domain with Wavelets

Images do not have to be completely transformed into the frequency domain. Some transformations only partially convert an image into the frequency domain. One of the most common types of these transformations is into the time-frequency or wavelet domain.

The Discrete Wavelet Transform (DWT) is used in numerical analysis to transform an image from the spatial domain to the time-frequency domain and back again. This transform is different from the FFT. The FFT decomposes an image with sines and cosines over the entire image. In contrast, the wavelet functions are applied multiple times over portions.

The image information within the time-frequency domain shows the frequency of patterns within an image, and how these patterns vary over the image. The low frequencies typically contain most of the information, which is commonly seen as a peak (spike) of data within the time-frequency domain. The information at the high frequencies is usually noise. The image can easily be altered within the time-frequency domain to remove the noise.

The following sections introduce the concepts needed to work with images and Discrete Wavelet Transforms (DWTs):

- “Transforming to the Time-Frequency Domain”
- “Displaying Images in the Time-Frequency Domain” on page 370
- “Transforming from the Time-Frequency Domain” on page 374

The wavelet transformation process is the basis for many image compression algorithms. See “Removing Noise with the Wavelet Transform” on page 378 for an example of how wavelets can be used to compress data and remove noise.

### Transforming to the Time-Frequency Domain

When an image is transformed with a DWT from the spatial domain to the time-frequency domain, the transformation process is referred to as a forward DWT. The forward DWT process can be performed with IDL’s WTN function.

The low frequencies usually contain most of the useful information within the image, which is shown by the peak (spike) of data around the origin within the time-frequency domain. If the image does not contain any background noise, the rest of the data frequency values are very close to zero. However, the results of the WTN function have a very wide range. An initial display may not show any variations from zero, but a smaller surface range will show that the image does actually contain background noise. Since scaling a range can sometimes be quite arbitrary, different methods are used. See "Displaying Images in the Time-Frequency Domain" on page 370 for more information on displaying the results of a forward DWT.

The following example shows how to use IDL's WTN function to compute a forward DWT. This example uses the first image within the `abnorm.dat` file, which is in the `examples/data` directory.

For code that you can copy and paste into an Editor window, see "Example Code: Transforming to the Time-Frequency Domain" on page 369 or complete the following steps for a detailed description of the process.

1. Import the first image from the `abnorm.dat` file:

   ```
   imageSize = [64, 64]
   file = FILEPATH('abnorm.dat', $
      SUBDIRECTORY = ['examples', 'data'])
   image = READ_BINARY(file, DATA_DIMS = imageSize)
   ```

2. Initialize a display size parameter to resize the image when displaying it:

   ```
   displaySize = 2*imageSize
   ```

3. Initialize the display:

   ```
   DEVICE, DECOMPOSED = 0
   LOADCT, 0
   ```

4. Create a window and display the image:

   ```
   WINDOW, 0, XSIZE = displaySize[0], $
      YSIZE = displaySize[1], TITLE = 'Original Image'
   TVSCL, CONGRID(image, displaySize[0], $
      displaySize[1])
   ```

The following figure shows the original image.

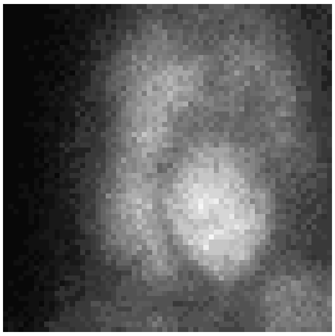


*Figure 9-12: Original Gated Blood Pool Image*

5. With the WTN function, transform the image into the wavelet domain:

   ```
   waveletTransform = WTN(image, 20)
   ```

   The *Coef* argument is set to 20 to specify 20 wavelet filter coefficients to provide the most efficient wavelet estimate possible. Less wavelet filter coefficients can be used with larger images to decrease computation time.

6. Create another window and display the wavelet transform:

   ```
   WINDOW, 1, TITLE = 'Wavelet: Transform'
   SHADE_SURF, waveletTransform, /XSTYLE, /YSTYLE, $
      /ZSTYLE, TITLE = 'Transform of Image', $
      XTITLE = 'Horizontal Number', $
      YTITLE = 'Vertical Number', $
      ZTITLE = 'Amplitude', CHARSIZE = 1.5
   ```

The following figure shows the wavelet transform. The data at the high frequencies seems to be close to zero, but the peak (spike) in the *z* range is so large that a closer look is needed.

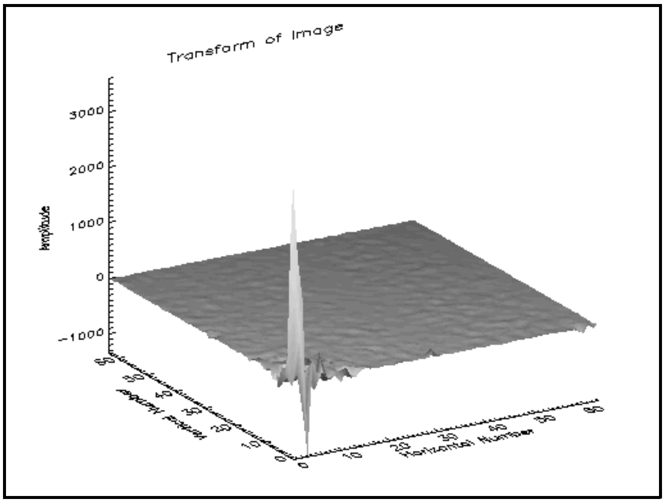


*Figure 9-13: Wavelet Transform of Gated Blood Pool Image*

7. Create another window and display the wavelet transform, scaling the data (*z*) range from 0 to 200:

```
WINDOW, 2, TITLE = 'Wavelet: Transform (Closer Look)'
SHADE_SURF, waveletTransform, /XSTYLE, /YSTYLE, $
   /ZSTYLE, TITLE = 'Transform of Image', $
   XTITLE = 'Horizontal Number', $
   YTITLE = 'Vertical Number',
   ZTITLE = 'Amplitude', CHARSIZE = 1.5, $
   ZRANGE = [0., 200.]
```

The following figure shows the wavelet transform with the *z*-axis ranging from 0 to 200. A closer looks shows that the image does contain background noise.

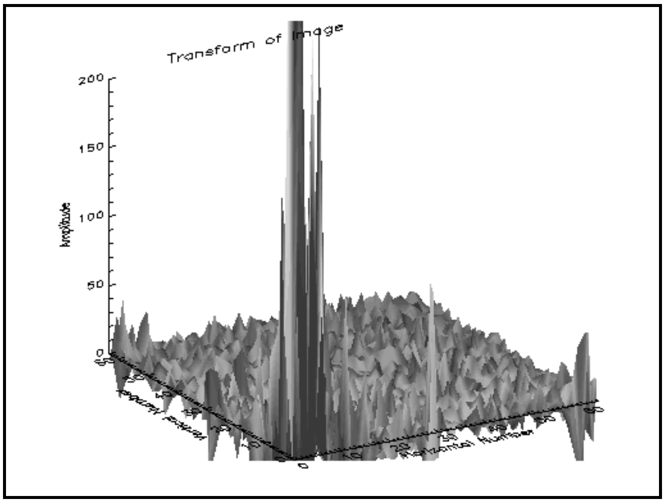


*Figure 9-14: Wavelet Transform of Image Scaled Between 0 and 200*

## Example Code: Transforming to the Time-Frequency Domain

Copy and paste the following text into an IDL Editor window. After saving the file as `ForwardWavelet.pro`, compile and run the program to reproduce the previous example.

```
PRO ForwardWavelet

; Import the image from the file.
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize a display size parameter to resize the
; image when displaying it.
displaySize = 2*imageSize

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the image.
```

```
WINDOW, 0, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], TITLE = 'Original Image'
TVSCL, CONGRID(image, displaySize[0], $
   displaySize[1])

; Transform the image into the wavelet domain.
waveletTransform = WTN(image, 20)

; Create another window and display the frequency
; transform.
WINDOW, 1, TITLE = 'Wavelet: Transform'
SHADE_SURF, waveletTransform, /XSTYLE, /YSTYLE, $
   /ZSTYLE, TITLE = 'Transform of Image', $
   XTITLE = 'Horizontal Number', $
   YTITLE = 'Vertical Number', $
   ZTITLE = 'Amplitude', CHARSIZE = 1.5

; Create another window and display the frequency
; transform within the data (z) range of 0 to 200.
WINDOW, 2, TITLE = 'Wavelet: Transform (Closer Look)'
SHADE_SURF, waveletTransform, /XSTYLE, /YSTYLE, $
   /ZSTYLE, TITLE = 'Transform of Image', $
   XTITLE = 'Horizontal Number', $
   YTITLE = 'Vertical Number', $
   ZTITLE = 'Amplitude', CHARSIZE = 1.5, $
   ZRANGE = [0., 200.]

END
```

## Displaying Images in the Time-Frequency Domain

Within the time-frequency domain, the range of values from the peak to the spurious high frequency data is extreme. The logarithmic scale is applied to retain the shape of the surface, but reduce its range. Since the logarithmic scale only applies to positive values, you should first compute the power spectrum, which is the absolute value squared of the transform.

The following example shows how to display the results of IDL's WTN function. This example also uses the first image within the `abnorm.dat` file, which is in the `examples/data` directory.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Images in the Time-Frequency Domain" on page 373 or complete the following steps for a detailed description of the process.

1. Import the first image from the `abnorm.dat` file:

```
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Initialize a display size parameter to resize the image when displaying it:

```
displaySize = 2*imageSize
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

4. Create a window and display the image:

```
WINDOW, 0, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], TITLE = 'Original Image'
TVSCL, CONGRID(image, displaySize[0], $
   displaySize[1])
```

The following figure shows the original image.

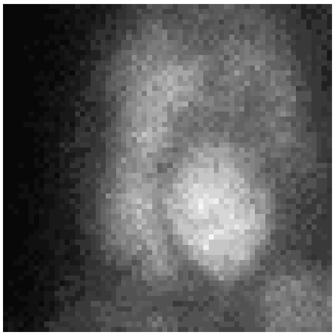


*Figure 9-15: Original Gated Blood Pool Image*

5. Transform the image into the time-frequency domain.

```
waveletTransform = WTN(image, 20)
```

The *Coef* argument is set to 20 to specify 20 wavelet filter coefficients to provide the most efficient wavelet estimate possible. Less wavelet filter coefficients can be used with larger images to decrease computation time.

6. Compute the power spectrum.

   ```
   powerSpectrum = ABS(waveletTransform)^2
   ```

7. Apply a logarithmic scale to the power spectrum:

   ```
   scaledPowerSpect = ALOG10(powerSpectrum)
   ```

8. Create another window and display the log-scaled power spectrum as a surface:

   ```
   WINDOW, 1, TITLE = 'Wavelet Power Spectrum: ' + $
      'Logarithmic Scale (surface)'
   SHADE_SURF, scaledPowerSpect, /XSTYLE, /YSTYLE, /ZSTYLE, $
      TITLE = 'Log-scaled Power Spectrum of Image', $
      XTITLE = 'Horizontal Number', $
      YTITLE = 'Vertical Number', $
      ZTITLE = 'Log(Squared Amplitude)', CHARSIZE = 1.5
   ```

   The following figure shows the log-scaled power spectrum of the wavelet transform as a surface.

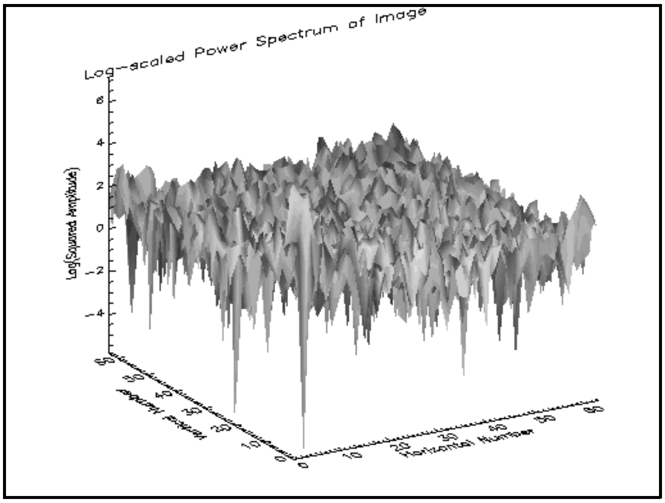


*Figure 9-16: Log-scaled Wavelet Power Spectrum of Image (as a surface)*

9. Create another window and display the log-scaled power spectrum as an image:

```
WINDOW, 2, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Wavelet Power Spectrum: Logarithmic Scale
(image)'
TVSCL, CONGRID(scaledPowerSpect, displaySize[0], $
   displaySize[1])
```

The following figure shows the log-scaled power spectrum as an image. Most of the signal is located near the origin (the lower left of the power spectrum image). This data is shown as bright pixels at the origin. The noise appears in the rest of the image.

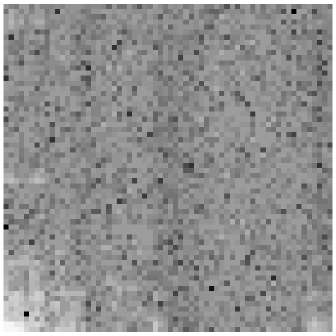


*Figure 9-17: Log-scaled Wavelet Power Spectrum of Image (as am image)*

## Example Code: Displaying Images in the Time-Frequency Domain

Copy and paste the following text into an IDL Editor window. After saving the file as `DisplayWavelet.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayWavelet

; Import the image from the file.
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize a display size parameter to resize the
; image when displaying it.
```

```
displaySize = 2*imageSize

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the image.
WINDOW, 0, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], TITLE = 'Original Image'
TVSCL, CONGRID(image, displaySize[0], $
   displaySize[1])

; Transform the image into the time-frequency domain.
waveletTransform = WTN(image, 20)

; Compute the power spectrum.
powerSpectrum = ABS(waveletTransform)^2

; Apply a logarithmic scale to the power spectrum.
scaledPowerSpect = ALOG10(powerSpectrum)

; Create another window and display the log-scaled
; power spectrum as a surface.
WINDOW, 1, TITLE = 'Wavelet Power Spectrum: ' + $
   'Logarithmic Scale (surface)'
SHADE_SURF, scaledPowerSpect, /XSTYLE, /YSTYLE, /ZSTYLE, $
   TITLE = 'Log-scaled Power Spectrum of Image', $
   XTITLE = 'Horizontal Number', $
   YTITLE = 'Vertical Number', $
   ZTITLE = 'Log(Abs(Amplitude^2))', CHARSIZE = 1.5

; Create another window and display the log-scaled
; power spectrum as an image.
WINDOW, 2, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Wavelet Power Secptrum: Logarithmic Scale (image)'
TVSCL, CONGRID(scaledPowerSpect, displaySize[0], $
   displaySize[1])

END
```

## Transforming from the Time-Frequency Domain

After manipulating an image within the time-frequency domain, you will need to transform it back to the spatial domain. This transformation process is referred to as an inverse DWT. The inverse DWT process can be performed with IDL’s WTN function by setting the INVERSE keyword.

The following example shows how to use IDL's WTN function to compute an inverse DWT. This example uses the first image within the `abnorm.dat` file, which is in the `examples/data` directory. The image is not manipulated while it is in the time-frequency domain to show that no data is lost when using the DWT. However, manipulating data within the time-frequency domain is a useful way to compress data and remove background noise from an image, as shown in "Removing Noise with the Wavelet Transform" on page 378.

For code that you can copy and paste into an Editor window, see "Example Code: Transforming from the Time-Frequency Domain" on page 377 or complete the following steps for a detailed description of the process.

1. Import the first image from the `abnorm.dat` file:

```
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Initialize a display size parameter to resize the image when displaying it:

```
displaySize = 2*imageSize
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

4. With the WTN function, transform the image into the wavelet domain:

```
waveletTransform = WTN(image, 20)
```

   The *Coef* argument is set to 20 to specify 20 wavelet filter coefficients to provide the most efficient wavelet estimate possible. Fewer wavelet filter coefficients can be used with larger images to decrease computation time.

5. Compute the power spectrum:

```
powerSpectrum = ABS(waveletTransform)^2
```

6. Apply a logarithmic scale to the power spectrum:

```
scaledPowerSpect = ALOG10(powerSpectrum)
```

7. Create a window and display the log-scaled power spectrum as an image:

```
; Create a window and display the transform.
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Power Spectrum Image'
TVSCL, CONGRID(scaledPowerSpect, displaySize[0], $
   displaySize[1])
```

The following figure shows the log-scaled power spectrum of the image.

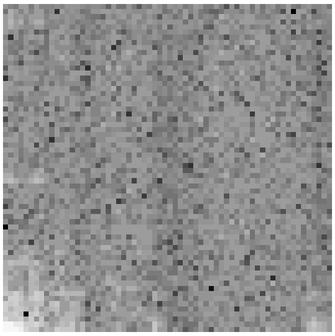


*Figure 9-18: Log-scaled Wavelet Power Spectrum of Image*

8. With the WTN function, transform the wavelet domain data back to the original image (obtain the inverse transform):

```
waveletInverse = WTN(waveletTransform, 20, /INVERSE)
```

9. Create another window and display the inverse transform as an image:

```
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Wavelet: Inverse Transform'
TVSCL, CONGRID(waveletInverse, displaySize[0], $
   displaySize[1])
```

The inverse transform is the same as the original image as shown in the following figure. No image data is lost when transforming an image to and from the time-frequency domain.

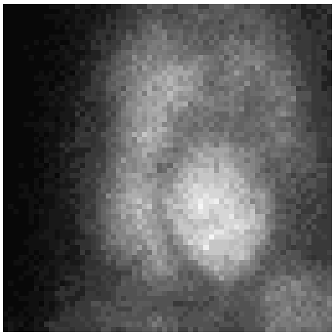


*Figure 9-19: Inverse of the Wavelet Transform of the Gated Blood Pool Image*

## Example Code: Transforming from the Time-Frequency Domain

Copy and paste the following text into an IDL Editor window. After saving the file as `InverseWavelet.pro`, compile and run the program to reproduce the previous example.

```
PRO InverseWavelet

; Import the image from the file.
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize a display size parameter to resize the
; image when displaying it.
displaySize = 2*imageSize

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Transform the image into the frequency domain.
waveletTransform = WTN(image, 20)

; Compute the power spectrum.
```

```
powerSpectrum = ABS(waveletTransform)^2

; Apply a logarithmic scale to the power spectrum.
scaledPowerSpectrum = ALOG10(powerSpectrum)

; Create a window and display the transform.
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Power Spectrum Image'
TVSCL, CONGRID(scaledPowerSpectrum, displaySize[0], $
  displaySize[1])

; Compute the inverse
waveletInverse = WTN(waveletTransform, 20, /INVERSE)

; Create another window and display the inverse
; transform as an image
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Wavelet: Inverse Transform'
TVSCL, CONGRID(waveletInverse, displaySize[0], $
   displaySize[1])

END
```

## Removing Noise with the Wavelet Transform

This example uses IDL's WTN function to remove noise from an image. The image comes from the `abnorm.dat` file found in the `examples/data` directory. The first display contains the original image and its wavelet transform. The noise is very evident in the image. A surface of the transform helps to determine beyond which point the noise occurs. Only the important data is kept and noise is replaced by zero values. The inverse transform is then applied, resulting in a cleaner image.

For code that you can copy and paste into an Editor window, see "Example Code: Removing Noise with the Wavelet Transform" on page 381 or complete the following steps for a detailed description of the process.

1. Import the first image from the `abnorm.dat` file:

```
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Initialize a display size parameter to resize the image when displaying it:

```
displaySize = 2*imageSize
```

3. Initialize the display:

   ```
   DEVICE, DECOMPOSED = 0
   LOADCT, 0
   ```

4. Create a window and display the image:

   ```
   WINDOW, 0, XSIZE = 2*displaySize[0], $
      YSIZE = displaySize[1], $
      TITLE = 'Original Image and Power Spectrum'
   TVSCL, CONGRID(image, displaySize[0], $
      displaySize[1]), 0
   ```

5. Determine the wavelet transform of the image:

   ```
   waveletTransform = WTN(image, 20)
   ```

   The *Coef* argument is set to 20 to specify 20 wavelet filter coefficients to provide the most efficient wavelet estimate possible. Fewer wavelet filter coefficients can be used with larger images to decrease computation time.

6. Display the power spectrum of the transform:

   ```
   TVSCL, CONGRID(ALOG10(ABS(waveletTransform^2)), $
      displaySize[0], displaySize[1]), 1
   ```

   The following figure shows the original image and its power spectrum within the time-frequency domain.

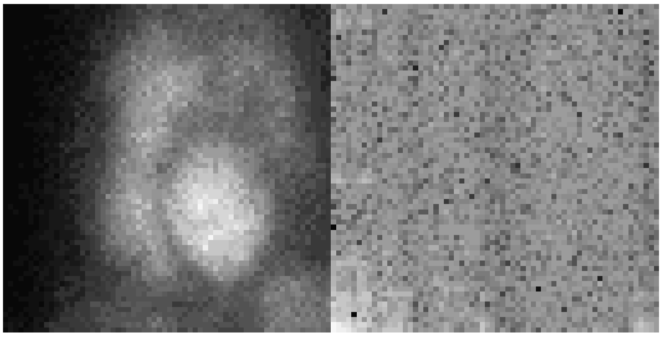


*Figure 9-20: Gated Blood Pool Image and Its Wavelet Power Spectrum*

7. Crop the transform to only include the quadrant of data closest to the spike of low frequency in the lower-left corner:

```
croppedTransform = FLTARR(imageSize[0], imageSize[1])
croppedTransform[0, 0] = waveletTransform[0:(imageSize[0]/2),
$
   0:(imageSize[1]/2)]
```

8. Create another window and display the power spectrum of the cropped transform as an image:

```
WINDOW, 1, XSIZE = 2*displaySize[0], $
   YSIZE = displaySize[1], $
   TITLE = 'Power Spectrum of Cropped Transform and Results'
TVSCL, CONGRID(ALOG10(ABS(croppededTransform^2)), $
   displaySize[0], displaySize[1]), 0, /NAN
```

9. Apply the inverse transformation to the masked power spectrum:

```
inverseTransform = WTN(maskedTransform, 20, /INVERSE)
```

10. Display results of the inverse transform:

```
TVSCL, CONGRID(inverseTransform, displaySize[0], $
   displaySize[1]), 1
```

The following figure shows the power spectrum of the cropped transform and its resulting inverse transform. The cropping process shows that only one quarter of the data was needed to reconstruct the image. The image is compressed by a 4:1 ratio.

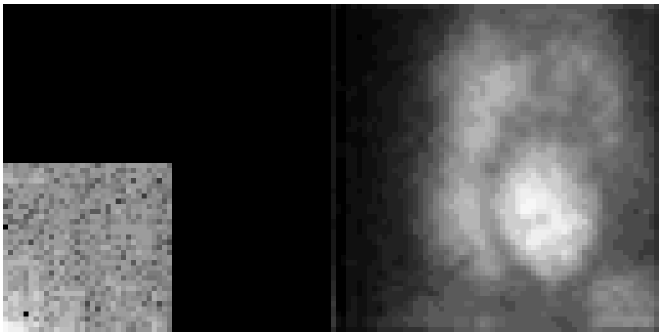


*Figure 9-21: Masked Wavelet Power Spectrum and Its Resulting Inverse Transform*

## Example Code: Removing Noise with the Wavelet Transform

Copy and paste the following text into an IDL Editor window. After saving the file as `RemovingNoiseWithWavelet.pro`, compile and run the program to reproduce the previous example.

```
PRO RemovingNoiseWithWavelet

; Import the image from the file.
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize a display size parameter to resize the
; image when displaying it.
displaySize = 2*imageSize

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the original image.
WINDOW, 0, XSIZE = 2*displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Original Image and Power Spectrum'
TVSCL, CONGRID(image, displaySize[0], displaySize[1]), 0

; Determine the transform of the image.
waveletTransform = WTN(image, 20)

; Display the power spectrum.
TVSCL, CONGRID(ALOG10(ABS(waveletTransform^2)), $
   displaySize[0], displaySize[1]), 1

; Crop the transform to only include data close to
; the spike in the lower-left corner.
croppedTransform = FLTARR(imageSize[0], imageSize[1])
croppedTransform[0, 0] = waveletTransform[0:(imageSize[0]/2), $
   0:(imageSize[1]/2)]

; Create another window and display the power spectrum
; of the cropped transform as an image.
WINDOW, 1, XSIZE = 2*displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Power Spectrum of Cropped Transform and Results'
TVSCL, CONGRID(ALOG10(ABS(croppedTransform^2)), $
   displaySize[0], displaySize[1]), 0, /NAN

; Apply the inverse transformation to cropped transform.
inverseTransform = WTN(croppedTransform, 20, /INVERSE)
```

```
; Display results of inverse transformation.
TVSCL, CONGRID(inverseTransform, displaySize[0], $
   displaySize[1]), 1

END
```

## Transforming to and from the Hough and Radon Domains

The Hough transform is used to transform from the spatial domain to the Hough domain and back again. The image information within the Hough domain shows the pixels of the original (spatial) image as sinusoidal curves. If the points of the original image form a straight line, their related sinusoidal curves in the Hough domain will intersect. Many intersections produce a peak. Masks can be easily applied to the image within the Hough domain to determine if and where straight lines occur.

The Radon transform is used to transform from the spatial domain to the Radon domain and back again. The image information within the Radon domain shows a line through the original image as a point. Specific features (geometries) in the original image produce peaks or collections of points. Masks can be easily applied to the image within the Radon domain to determine if and where these specific features occur.

Unlike transformations to and from the frequency and time-frequency domains, the Hough and Radon transforms do lose some data during the transformation process. These transformations are usually applied to the original image as a mask instead of producing an image from the results of the transform itself. See the HOUGH and RADON descriptions *in the IDL Reference Guide* for more information on Hough and Radon transform theory.

The following sections introduce the concepts needed to work with images and these transforms:

- "Transforming to the Hough and Radon Domains (Projecting)" on page 384
- "Transforming from the Hough and Radon Domains (Backprojecting)" on page 389

The Hough transformation process is used to find straight lines within an image. See "Finding Straight Lines with the Hough Transform" on page 394 for an example. The Radon transformation process is used to enhance contrast within an image. See "Color Density Contrasting with the Radon Transform" on page 402 for an example.

## Transforming to the Hough and Radon Domains (Projecting)

When an image is transformed from the spatial domain to either the Hough or Radon domain, the transformation process is referred to as a Hough or Radon projection. The projection process can be performed with either IDL's HOUGH function or IDL's RADON function, depending on which transform you want to use.

The following example shows how to use IDL's HOUGH and RADON functions to compute and display the Hough and Radon projections. This example uses the first image within the `abnorm.dat` file, which is in the `examples/data` directory.

For code that you can copy and paste into an Editor window, see "Example Code: Hough and Radon Projections" on page 387 or complete the following steps for a detailed description of the process.

1. Import the first image from the `abnorm.dat` file:

```
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Define the display size and offset parameters to resize and position the images when displaying them:

```
displaySize = 2*imageSize
offset = displaySize/3
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

4. Create a window and display the image:

```
WINDOW, 0, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], TITLE = 'Original Image'
TVSCL, CONGRID(image, displaySize[0], $
   displaySize[1])
```

The following figure shows the original image.

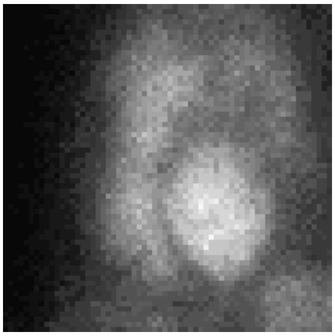


*Figure 9-22: Original Gated Blood Pool Image*

5. With the HOUGH function, transform the image into the Hough domain:

```
houghTransform = HOUGH(image, RHO = houghRadii, $
   THETA = houghAngles, /GRAY)
```

6. Create another window and display the Hough transform with axes provided by the PLOT procedure:

```
WINDOW, 1, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Hough Transform'
TVSCL, CONGRID(houghTransform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, houghAngles, houghRadii, /XSTYLE, /YSTYLE, $
   TITLE = 'Hough Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5
```

The following figure shows the resulting Hough transform.

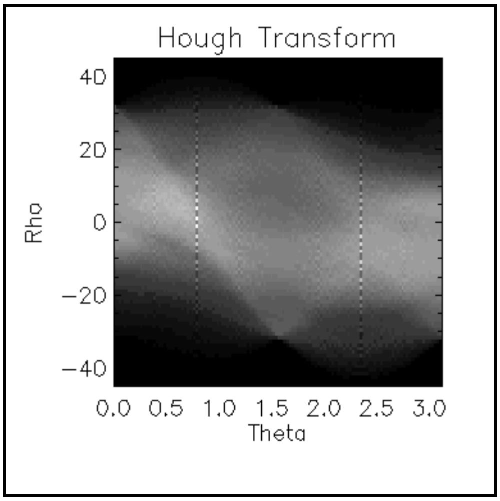


*Figure 9-23: Hough Transform of the Gated Blood Pool Image*

7. With the RADON function, transform the image into the Radon domain with axes provided by the PLOT procedure:

```
radonTransform = RADON(image, RHO = radonRadii, $
   THETA = radonAngles, /GRAY)
```

8. Create another window and display the Radon transform:

```
WINDOW, 2, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Radon Transform'
TVSCL, CONGRID(radonTransform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, radonAngles, radonRadii, /XSTYLE, /YSTYLE, $
   TITLE = 'Radon Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5
```

The following figure shows the resulting Radon transform.

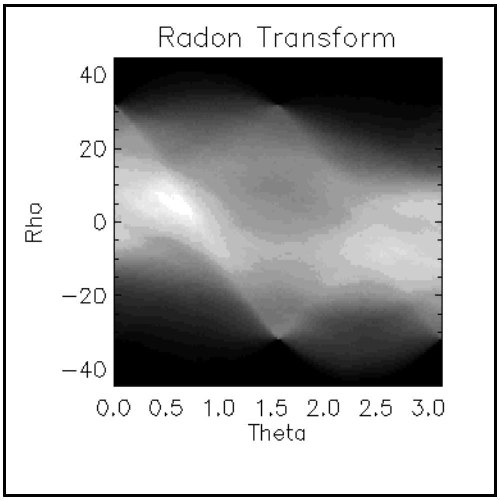


*Figure 9-24: Radon Transform of the Gated Blood Pool Image*

## Example Code: Hough and Radon Projections

Copy and paste the following text into an IDL Editor window. After saving the file as `ForwardHoughAndRadon.pro`, compile and run the program to reproduce the previous example.

```
PRO ProjectHoughAndRadon

; Import the image from the file.
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Define the display size and offset parameters to
; resize and position the images when displaying them.
displaySize = 4*imageSize
offset = displaySize/3

; Initialize the displays.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the image.
```

```
WINDOW, 0, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], TITLE = 'Original Image'
TVSCL, CONGRID(image, displaySize[0], $
   displaySize[1])

; With the HOUGH function, transform the image into the
; Hough domain.
houghTransform = HOUGH(image, RHO = houghRadii, $
   THETA = houghAngles, /GRAY)

; Create another window and display the Hough transform
; with axes.
WINDOW, 1, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Hough Transform'
TVSCL, CONGRID(houghTransform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, houghAngles, houghRadii, /XSTYLE, /YSTYLE, $
   TITLE = 'Hough Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5

; With the RADON function, transform the image into the
; Radon domain.
radonTransform = RADON(image, RHO = radonRadii, $
   THETA = radonAngles, /GRAY)

; Create another window and display the Radon transform
; with axes.
WINDOW, 2, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Radon Transform'
TVSCL, CONGRID(radonTransform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, radonAngles, radonRadii, /XSTYLE, /YSTYLE, $
   TITLE = 'Radon Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5

END
```

## Transforming from the Hough and Radon Domains (Backprojecting)

After manipulating an image within either the Hough or Radon domain, you may need to transform the image from that domain back to the spatial domain. This transformation process is referred to as a Hough or Radon backprojection. The backprojection process can be performed with either IDL's HOUGH function or IDL's RADON function, depending on which domain your image is in. You can perform the backprojection process with these functions by setting the BACKPROJECT keyword.

The following example shows how to use IDL's HOUGH and RADON functions to compute the backprojection from these domains. This example uses the first image within the `abnorm.dat` file, which is in the `examples/data` directory. Although the image is not manipulated while it is in the Hough or Radon domain, information is lost when using these transforms.

For code that you can copy and paste into an Editor window, see "Example Code: Hough and Radon Backprojections" on page 392 or complete the following steps for a detailed description of the process.

1. Import in the first image from the `abnorm.dat` file:

```
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Define the display size and offset parameters to resize and position the images when displaying them:

```
displaySize = 2*imageSize
offset = displaySize/3
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

4. With the HOUGH function, transform the image into the Hough domain:

```
houghTransform = HOUGH(image, RHO = houghRadii, $
   THETA = houghAngles, /GRAY)
```

5. Create another window and display the Hough transform with axes provided by the PLOT procedure:

```
WINDOW, 1, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Hough Transform'
TVSCL, CONGRID(houghTransform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, houghAngles, houghRadii, /XSTYLE, /YSTYLE, $
   TITLE = 'Hough Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5
```

6. With the RADON function, transform the image into the Radon domain with axes provided by the PLOT procedure:

```
radonTransform = RADON(image, RHO = radonRadii, $
   THETA = radonAngles, /GRAY)
```

7. Create another window and display the Radon transform:

```
WINDOW, 2, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Radon Transform'
TVSCL, CONGRID(radonTransform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, radonAngles, radonRadii, /XSTYLE, /YSTYLE, $
   TITLE = 'Radon Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5
```

The following figure shows the Hough and Radon transforms.

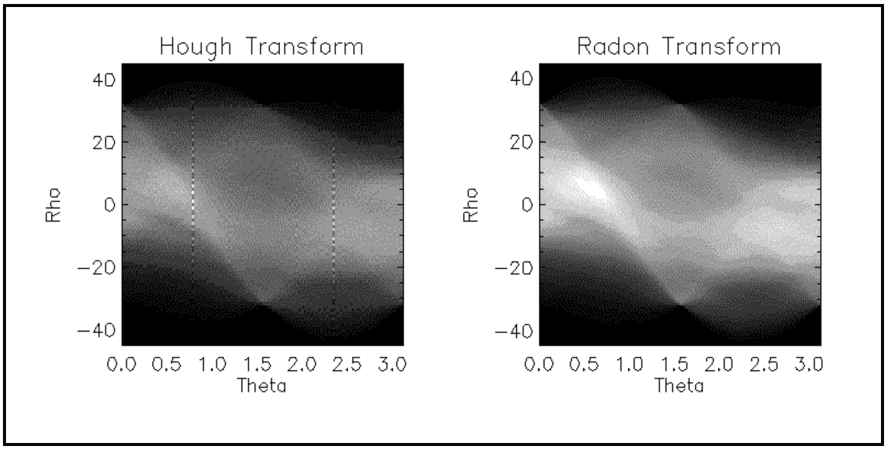


*Figure 9-25: Hough (left) and Radon (right) Transforms of Image*

8. Backproject the Hough and Radon transforms:

```
backprojectHough = HOUGH(houghTransform, /BACKPROJECT, $
   RHO = houghRadii, THETA = houghAngles, $
   NX = imageSize[0], NY = imageSize[1])
backprojectRadon = RADON(radonTransform, /BACKPROJECT, $
   RHO = radonRadii, THETA = radonAngles, $
   NX = imageSize[0], NY = imageSize[1])
```

9. Create another window and display the original image with the Hough and Radon backprojections:

```
WINDOW, 2, XSIZE = (3*displaySize[0]), $
   YSIZE = displaySize[1], $
   TITLE = 'Hough and Radon Backprojections'
TVSCL, CONGRID(image, displaySize[0], $
   displaySize[1]), 0
TVSCL, CONGRID(backprojectHough, displaySize[0], $
   displaySize[1]), 1
TVSCL, CONGRID(backprojectRadon, displaySize[0], $
   displaySize[1]), 2
```

The following figure shows the original image and its Hough and Radon transforms. These resulting images shows information is blurred when using the Hough and Radon transformations.

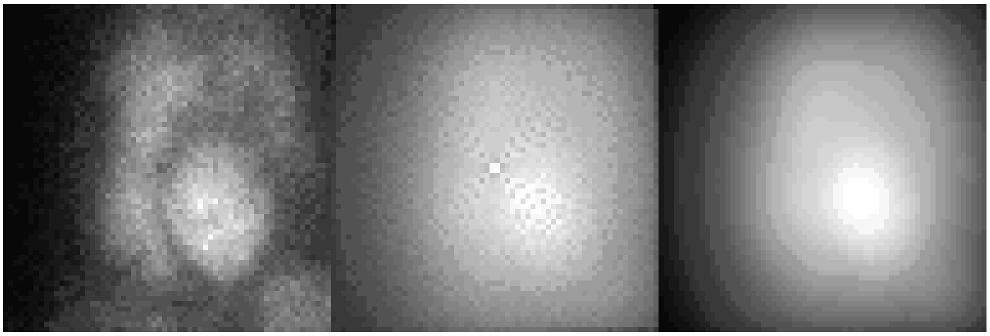


*Figure 9-26: Original Gated Blood Pool Image (left), Hough Backprojection (center), and Radon Backprojection (right)*

## Example Code: Hough and Radon Backprojections

Copy and paste the following text into an IDL Editor window. After saving the file as `BackprojectHoughAndRadon.pro`, compile and run the program to reproduce the previous example.

```
PRO BackprojectHoughAndRadon

; Import the image from the file.
imageSize = [64, 64]
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Define the display size and offset parameters to
; resize and position the images when displaying them.
displaySize = 4*imageSize
offset = displaySize/3

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; With the HOUGH function, transform the image into the
; Hough domain.
houghTransform = HOUGH(image, RHO = houghRadii, $
   THETA = houghAngles, /GRAY)

; Create another window and display the Hough transform
; with axes.
WINDOW, 0, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Hough Transform'
TVSCL, CONGRID(houghTransform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, houghAngles, houghRadii, /XSTYLE, /YSTYLE, $
   TITLE = 'Hough Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5

; With the RADON function, transform the image into the
; Radon domain.
radonTransform = RADON(image, RHO = radonRadii, $
   THETA = radonAngles, /GRAY)

; Create another window and display the Radon transform
; with axes.
WINDOW, 1, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Radon Transform'
TVSCL, CONGRID(radonTransform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, radonAngles, radonRadii, /XSTYLE, /YSTYLE, $
   TITLE = 'Radon Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5

; Backproject the Hough and Radon transforms.
backprojectHough = HOUGH(houghTransform, /BACKPROJECT, $
   RHO = houghRadii, THETA = houghAngles, $
   NX = imageSize[0], NY = imageSize[1])
backprojectRadon = RADON(radonTransform, /BACKPROJECT, $
   RHO = radonRadii, THETA = radonAngles, $
   NX = imageSize[0], NY = imageSize[1])

; Create another window and display the original image
; with the Hough and Radon backjections.
WINDOW, 2, XSIZE = (3*displaySize[0]), $
   YSIZE = displaySize[1], $
   TITLE = 'Hough and Radon Backprojections'
TVSCL, CONGRID(image, displaySize[0], $
   displaySize[1]), 0
TVSCL, CONGRID(backprojectHough, displaySize[0], $
   displaySize[1]), 1
TVSCL, CONGRID(backprojectRadon, displaySize[0], $
   displaySize[1]), 2

END
```

## Finding Straight Lines with the Hough Transform

This example uses the Hough transform to find straight lines within an image. The image comes from the `rockland.png` file found in the `examples/data` directory. The image is a saturation composite of a 24 hour period in Rockland, Maine. A saturation composite is normally used to highlight intensities, but the Hough transform is used in this example to extract the power lines, which are straight lines. The Hough transform is applied to the green band of the image. The results of the transform are scaled to only include lines longer than 85 pixels. The scaled results are then backprojected by the Hough transform to produce an image of only the straight power lines.

For code that you can copy and paste into an Editor window, see "Example Code: Finding Straight Lines with the Hough Transform" on page 399 or complete the following steps for a detailed description of the process.

1. Import the image from the `rockland.png` file:

   ```
   file = FILEPATH('rockland.png', $
      SUBDIRECTORY = ['examples', 'data'])
   image = READ_PNG(file)
   ```

2. Determine the size of the image:

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

3. Initialize a TrueColor display:

   ```
   DEVICE, DECOMPOSED = 1
   ```

4. Create a window and display the original image:

   ```
   WINDOW, 0, XSIZE = imageSize[1], YSIZE = imageSize[2], $
      TITLE = 'Rockland, Maine'
   TV, image, TRUE = 1
   ```

The following figure shows the original image.

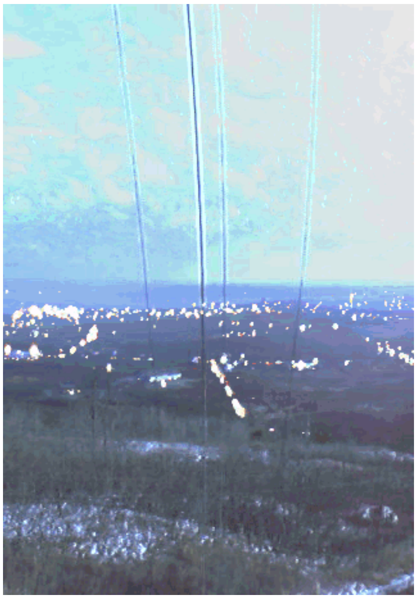


*Figure 9-27: Image of Rockland, Maine*

5. Use the image from green channel to provide an outline of shapes:

```
intensity = REFORM(image[1, *, *])
```

6. Determine the size of the intensity image derived from the green channel:

```
intensitySize = SIZE(intensity, /DIMENSIONS)
```

7. Threshold the intensity image to highlight the power lines:

```
mask = intensity GT 240
```

**Note**

The intensity image values range from 0 to 255. The threshold was derived by iteratively viewing the intensity image at several different values.

8. Initialize the remaining displays:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

9. Create another window and display the masked image:

```
WINDOW, 1, XSIZE = intensitySize[0], $
   YSIZE = intensitySize[1], $
   TITLE = 'Mask to Locate Power Lines'
TVSCL, mask
```

The following figure shows the mask of the original image.

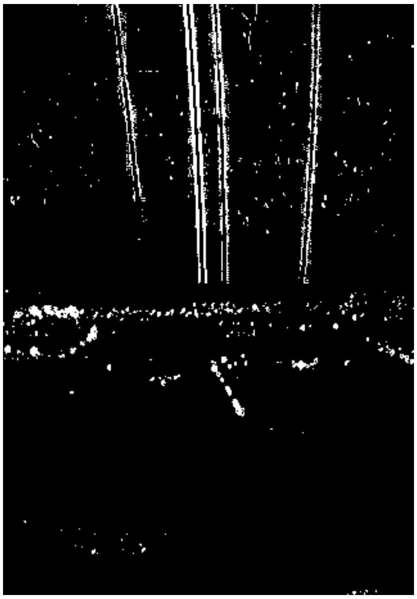


*Figure 9-28: Mask of Rockland Image*

10. Transform the mask with the HOUGH function:

```
transform = HOUGH(mask, RHO = rho, THETA = theta)
```

11. Define the size and offset parameters for the transform displays:

```
displaySize = [256, 256]
offset = displaySize/3
```

12. Reverse the color table to clarify the lines:

```
TVLCT, red, green, blue, /GET
TVLCT, 255 - red, 255 - green, 255 - blue
```

13. Create another window and display the Hough transform with axes provided by the PLOT procedure:

```
WINDOW, 2, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Hough Transform'
TVSCL, CONGRID(transform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, theta, rho, /XSTYLE, /YSTYLE, $
   TITLE = 'Hough Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5, $
   COLOR = !P.BACKGROUND
```

14. Scale the transform to obtain just the power lines, retaining only those lines longer than 85 pixels:

```
transform = (TEMPORARY(transform) - 85) > 0
```

The value of 85 comes from an estimate of the length of the power lines within the original and intensity images.

15. Create another window and display the scaled Hough transform with axes provided by the PLOT procedure:

```
WINDOW, 2, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Scaled Hough Transform'
TVSCL, CONGRID(transform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, theta, rho, /XSTYLE, /YSTYLE, $
   TITLE = 'Scaled Hough Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5, $
   COLOR = !P.BACKGROUND
```

The top image in the following figure shows the Hough transform of the intensity image. This transform is masked to only include straight lines of more than 85 pixels. The bottom image in the following figure shows the results of this mask. Only the far left and right intersections are retained.

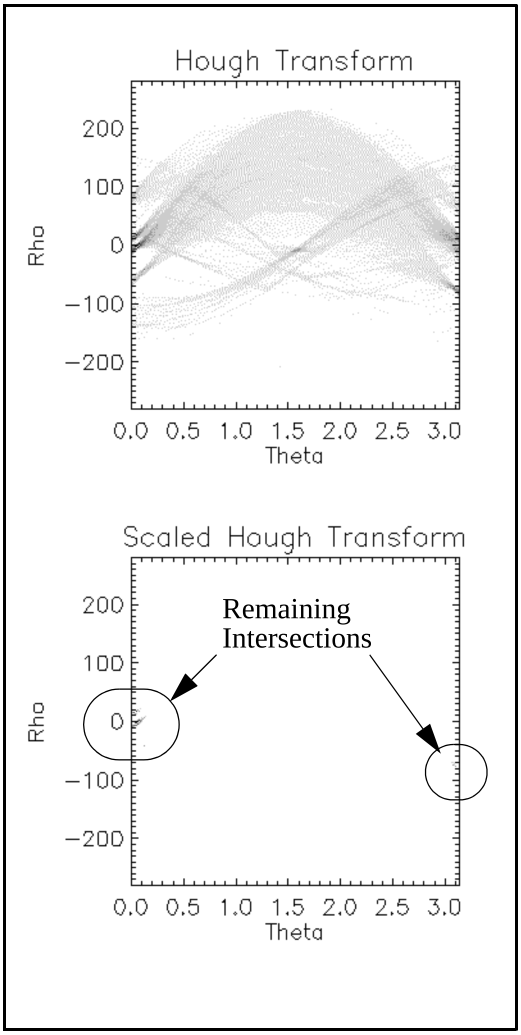


*Figure 9-29: The Hough Transform (top) and the Scaled Transform (bottom) of the Masked Intensity Image*

16. Backproject to compare with the original image:

```
backprojection = HOUGH(transform, /BACKPROJECT, $
   RHO = rho, THETA = theta, $
   NX = intensitySize[0], NY = intensitySize[1])
```

17. Create another window and display the resulting backprojection:

```
WINDOW, 4, XSIZE = intensitySize[0], $
   YSIZE = intensitySize[1], $
   TITLE = 'Resulting Power Lines'
TVSCL, backprojection
```

The following figure shows the resulting backprojection, which contains only the power lines.

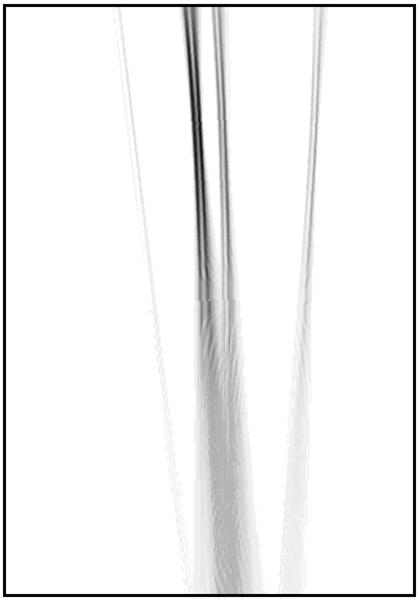


*Figure 9-30: The Resulting Backprojection of the Scaled Hough Transform*

## Example Code: Finding Straight Lines with the Hough Transform

Copy and paste the following text into an IDL Editor window. After saving the file as `FindingLinesWithHough.pro`, compile and run the program to reproduce the previous example.

```
PRO FindingLinesWithHough

; Import the image from file.
file = FILEPATH('rockland.png', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_PNG(file)

; Determine size of image.
imageSize = SIZE(image, /DIMENSIONS)

; Initialize the TrueColor display.
DEVICE, DECOMPOSED = 1

; Create a window and display the original image.
WINDOW, 0, XSIZE = imageSize[1], YSIZE = imageSize[2], $
   TITLE = 'Rockland, Maine'
TV, image, TRUE = 1

; Use the image from green channel to provide outlines
; of shapes.
intensity = REFORM(image[1, *, *])

; Determine size of intensity image.
intensitySize = SIZE(intensity, /DIMENSIONS)

; Mask intensity image to highlight power lines.
mask = intensity GT 240

; Initialize the remaining displays.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create another window and display the masked image.
WINDOW, 1, XSIZE = intensitySize[0], $
   YSIZE = intensitySize[1], $
   TITLE = 'Mask to Locate Power Lines'
TVSCL, mask

; Transform mask.
transform = HOUGH(mask, RHO = rho, THETA = theta)

; Define the size and offset parameters for the
; transform displays.
displaySize = [256, 256]
offset = displaySize/3

; Reverse color table to clarify lines.
TVLCT, red, green, blue, /GET
TVLCT, 255 - red, 255 - green, 255 - blue
```

```
; Create another window and display the Hough transform
; with axes.
WINDOW, 2, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Hough Transform'
TVSCL, CONGRID(transform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, theta, rho, /XSTYLE, /YSTYLE, $
   TITLE = 'Hough Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5, $
   COLOR = !P.BACKGROUND

; Scale transform to obtain just the power lines.
transform = (TEMPORARY(transform) - 85) > 0

; Create another window and display the scaled transform.
WINDOW, 3, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Scaled Hough Transform'
TVSCL, CONGRID(transform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, theta, rho, /XSTYLE, /YSTYLE, $
   TITLE = 'Scaled Hough Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5, $
   COLOR = !P.BACKGROUND

; Backproject to compare with original image.
backprojection = HOUGH(transform, /BACKPROJECT, $
   RHO = rho, THETA = theta, $
   NX = intensitySize[0], NY = intensitySize[1])

; Create another window and display the results.
WINDOW, 4, XSIZE = intensitySize[0], $
   YSIZE = intensitySize[1], $
   TITLE = 'Resulting Power Lines'
TVSCL, backprojection

END
```

## Color Density Contrasting with the Radon Transform

This example uses the Radon transform to provide more contrast within an image based on its color density. The image comes from the `endocell.jpg` file found in the `examples/data` directory. The image is a photomicrograph of cultured endothelial cells. The edges (outlines) within the image are defined by the Roberts filter. The Radon transform is then applied to the filtered image. The transform is scaled to only include the values above the mean of the transform. The scaled results are backprojected by the Radon transform. The resulting backprojection is used as a mask on the original image. The final resulting image shows more color contrast along the edges of the cell nuclei within the image.

For code that you can copy and paste into an Editor window, see “Example Code: Color Density Contrasting with the Radon Transform” on page 406 or complete the following steps for a detailed description of the process.

1. Import in the image from the `endocell.jpg` file:

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

2. Determine the image’s size, but divide it by 4 to reduce the image:

```
imageSize = SIZE(endocellImage, /DIMENSIONS)/4
```

3. Resize the image to a quarter of its original length and width:

```
endocellImage = CONGRID(endocellImage, $
   imageSize[0], imageSize[1])
```

4. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

5. Create a window and display the original image:

```
WINDOW, 0, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original (left) and Filtered (right)'
TV, endocellImage, 0
```

6. Filter the original image to clarify the edges of the cells:

```
image = ROBERTS(endocellImage)
```

7. Display the filtered image:

```
TVSCL, image, 1
```

The following figure shows the original image and the results of the edge detection filter.

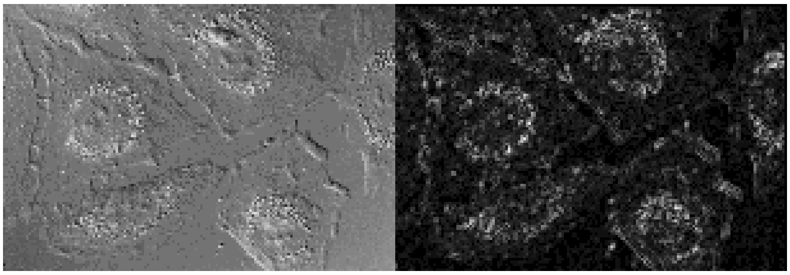


*Figure 9-31: Endothelial Cells Image (left) and the Resulting Edge-Filtered Image (right)*

8. Transform the filtered image:

```
transform = RADON(image, RHO = rho, THETA = theta)
```

9. Define the size and offset parameters for the transform displays:

```
displaySize = [256, 256]
offset = displaySize/3
```

10. Create another window and display the Radon transform with axes provided by the PLOT procedure:

```
WINDOW, 1, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Radon Transform'
TVSCL, CONGRID(transform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, theta, rho, /XSTYLE, /YSTYLE, $
   TITLE = 'Radon Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5
```

11. Scale the transform to include only the density values above the mean of the transform:

    ```
    scaledTransform = transform > MEAN(transform)
    ```

12. Create another window and display the scaled Radon transform with axes provided by the PLOT procedure:

    ```
    WINDOW, 2, XSIZE = displaySize[0] + 1.5*offset[0], $
       YSIZE = displaySize[1] + 1.5*offset[1], $
       TITLE = 'Scaled Radon Transform'
    TVSCL, CONGRID(scaledTransform, displaySize[0], $
       displaySize[1]), offset[0], offset[1]
    PLOT, theta, rho, /XSTYLE, /YSTYLE, $
       TITLE = 'Scaled Radon Transform', XTITLE = 'Theta', $
       YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
       POSITION = [offset[0], offset[1], $
       displaySize[0] + offset[0], $
       displaySize[1] + offset[1]], CHARSIZE = 1.5
    ```

    The following figure shows the original Radon transform of the edge-filtered image and the resulting scaled transform. The high intensity values within the diamond shape of the center of the transform represent high color density within the filtered and original image. The transform is scaled to highlight this segment of intersecting lines.

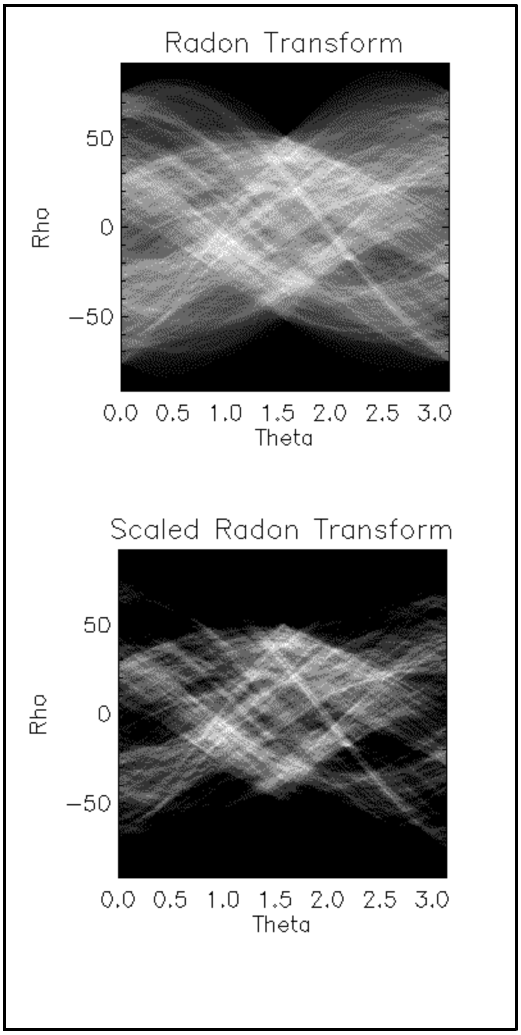


*Figure 9-32: Radon Transform (top) and Scaled Transform (bottom) of the Edge-Filtered Image*

13. Backproject the scaled transform:

```
backprojection = RADON(scaledTransform, /BACKPROJECT, $
   RHO = rho, THETA=theta, NX = imageSize[0], $
   NY = imageSize[1])
```

14. Create another window and display the backprojection:

```
WINDOW, 3, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Backproject (left) and Final Result (right)'
TVSCL, backprojection, 0
```

15. Use the backprojection as a mask to provide a color density contrast of the original image:

```
constrastingImage = endocellImage*backprojection
```

16. Display the resulting contrast image:

```
TVSCL,constrastingImage, 1
```

The following figure shows the Radon backprojection and a combined image of the original and the backprojection. The cell nuclei now have more contrast than the rest of the image.

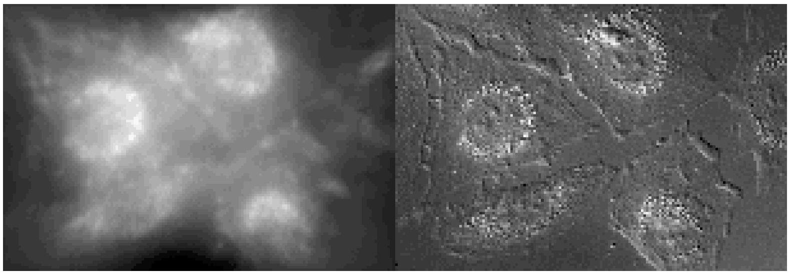
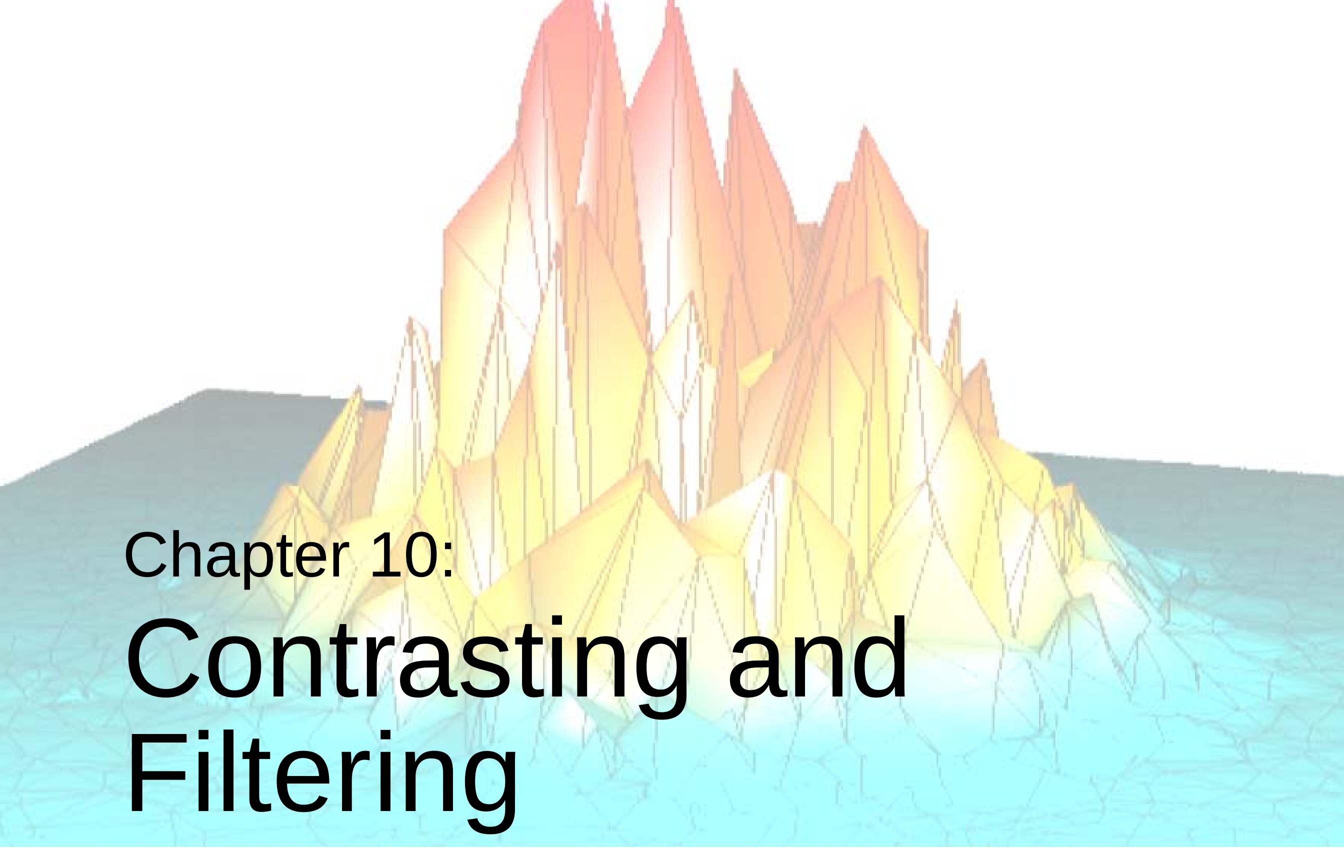


*Figure 9-33: The Backprojection of the Radon Transform and the Resulting Contrast Image*

## Example Code: Color Density Contrasting with the Radon Transform

Copy and paste the following text into an IDL Editor window. After saving the file as `ContrastingCellsWithRadon.pro`, compile and run the program to reproduce the previous example.

```
PRO ContrastingCellsWithRadon

; Import the image from the file.
file = FILEPATH('endocell.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, endocellImage

; Determine image's size, but divide it by 4 to reduce
; the image.
imageSize = SIZE(endocellImage, /DIMENSIONS)/4

; Resize image to a quarter its original length and
; width.
endocellImage = CONGRID(endocellImage, $
   imageSize[0], imageSize[1])

; Initialize the displays.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the original image.
WINDOW, 0, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original (left) and Filtered (right)'
TV, endocellImage, 0

; Filter original image to clarify the edges of the
; cells.
image = ROBERTS(endocellImage)

; Display the filtered image.
TVSCL, image, 1

; Transform the filtered image.
transform = RADON(image, RHO = rho, THETA = theta)

; Define the size and offset parameters for the
; transform displays.
displaySize = [256, 256]
offset = displaySize/3

; Create another window and display the Radon transform
; with axes.
WINDOW, 1, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Radon Transform'
TVSCL, CONGRID(transform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, theta, rho, /XSTYLE, /YSTYLE, $
   TITLE = 'Radon Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5

; Scale the transform to include only the density
; values above the mean of the transform.
scaledTransform = transform > MEAN(transform)

; Create another window and display the scaled Radon
; transform with axes.
WINDOW, 2, XSIZE = displaySize[0] + 1.5*offset[0], $
   YSIZE = displaySize[1] + 1.5*offset[1], $
   TITLE = 'Scaled Radon Transform'
TVSCL, CONGRID(scaledTransform, displaySize[0], $
   displaySize[1]), offset[0], offset[1]
PLOT, theta, rho, /XSTYLE, /YSTYLE, $
   TITLE = 'Scaled Radon Transform', XTITLE = 'Theta', $
   YTITLE = 'Rho', /NODATA, /NOERASE, /DEVICE, $
   POSITION = [offset[0], offset[1], $
   displaySize[0] + offset[0], $
   displaySize[1] + offset[1]], CHARSIZE = 1.5

; Backproject the scaled transform.
backprojection = RADON(scaledTransform, /BACKPROJECT, $
   RHO = rho, THETA=theta, NX = imageSize[0], $
   NY = imageSize[1])

; Create another window and display the backprojection.
WINDOW, 3, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Backproject (left) and Final Result (right)'
TVSCL, backprojection, 0

; Use the backprojection as a mask to provide
; a color density contrast of the original image.
constrastingImage = endocellImage*backprojection

; Display resulting contrast image.
TVSCL, constrastingImage, 1

END
```



# Chapter 10: Contrasting and Filtering

This chapter describes the following topics:

## Overview of Contrasting and Filtering

Contrast within an image is based on the brightness or darkness of a pixel in relation to other pixels. Modifying the contrast among neighboring pixels can enhance the ability to extract information from the image. Operations such as noise removal and smoothing decrease contrast and make neighboring pixel values more similar. Other operations such as scaling pixel values, edge detection and sharpening increase contrast to highlight specific image features.

A simple way to modify contrast is to scale the pixel values within an image. Within IDL, the pixel values of displayed images typically range from 0 to 255. Byte-scaling changes the range of values within an image to a linear progression from a minimum of 0 to a maximum of 255. For images with pixel values exceeding 255, byte-scaling produces a more linear display with the minimum value as the darkest pixel and the maximum value as the brightest pixel. For images with a smaller range in pixel values, byte-scaling increases the contrast and brightens dark areas. See “Byte-Scaling” on page 413 for more information on byte-scaling.

Contrast can also be increased to show more variations within uniform areas of the image using histogram equalization operations. These operations modify the distribution of pixel values within an image. See “Working with Histograms” on page 417 for more information on using histograms to modify contrast.

Filters provide another means of changing contrast within an image. A filter is represented by a kernel, which is an array that is multiplied and added to each pixel (and its surrounding values) within an image. Examples of such filters include low pass, high pass, directional, and Laplacian filters. See “Filtering an Image” on page 428 for more information on these filters. The following list introduces some of the specific operations covered in this section:

- Low pass filtering - a low pass filter provides the basis for smoothing operations. If an image contains too many variations to be able to determine specific features, smoothing can decrease the contrast so that some areas (especially the background) will not distract from viewing other areas of the image. See “Smoothing an Image” on page 448 for more information on smoothing.
- High pass filtering - a high pass filter provides the basis for sharpening operations. Some variations within areas of an image are too slight, causing some features to be indistinguishable from other features (usually the background). Sharpening increases the contrast in these areas, allowing these features to be clearly displayed. See “Sharpening an Image” on page 459 for more information on sharpening.

- Directional and Laplacian filters - these filters are the basis for edge detection operations. Shapes within an image are distinguished by their edges, which typically involve a sharp gradient. Edge detection increases the contrast between the boundary of the shape and the adjoining areas. See "Detecting Edges" on page 464 for more information on edge detection.
- Windowing and adaptive filters - more advanced filters are used to remove noise from an image. The variation in values between the noise and the image data is typically extreme, which detracts from the image clarity. Decreasing the contrast reduces the visible noise and allows the image to be properly viewed. See "Removing Noise" on page 470 for more information on removing noise within an image.

**Note**

In this book, Direct Graphics examples are provided by default. Object Graphics examples are provided in cases where significantly different methods are required.

The following list introduces the image contrasting and filtering tasks and associated IDL image routines covered in this chapter.

| Type of Contrasts or Filters | Routines | Description |
|---|---|---|
| "Byte-Scaling" on page 413 | BYTSCL | Byte-scale the data values of an image to produce a more continuous display or to increase its contrast. |
| "Working with Histograms" on page 417 | HIST_EQUAL<br>ADAPT_HIST_EQUAL | Use histogram equalization to show minor variations in uniform areas. |
| "Filtering an Image" on page 428 | CONVOL | Enhance contrast by applying some basic filters (low pass, high pass, directional, and Laplacian) to images. |

*Table 10-1: Image Contrasting and Filtering Tasks and Related Routines*

| Type of Contrasts or Filters | Routines | Description |
|---|---|---|
| "Smoothing an Image" on page 448 | SMOOTH<br>MEDIAN | Smooth high variations within an image. |
| "Sharpening an Image" on page 459 | CONVOL | Sharpen an image by decreasing too bright pixels and increasing too dark pixels. |
| "Detecting Edges" on page 464 | ROBERTS<br>SOBEL | Use the contrast within an image to detect the possible edges of shapes. |
| "Removing Noise" on page 470 | HANNING<br>LEEFILT | Remove noise from an image by either windowing or using an adaptive filter. |

*Table 10-1: Image Contrasting and Filtering Tasks and Related Routines*

**Note**

This chapter uses data files from the `IDL examples/data` directory. Two files, `data.txt` and `index.txt`, contain descriptions of the files, including array sizes.

## Byte-Scaling

The data values of some images may be greater than 255. When displayed with the TV routine or the IDLgrImage object, the data values above 255 are wrapped around the range of 0 to 255. This type of display may produce discontinuities in the resulting image.

The display can be changed to not wrap around and appear more linear by byte-scaling the image. The scaling process is linear with the minimum data value scaled to 0 and the maximum data value scaled to 255. You can use the BYTSCL function to perform this scaling process.

If the range of the pixel values within an image is less than 0 to 255, you can use the BYTSCL function to increase the range from 0 to 255. This change will increase the contrast within the image by increasing the brightness of darker regions. Keywords to the BYTSCL function also allow you to decrease contrast by setting the highest value of the image to less than 255.

**Note**

The BYTSCL function usually results in a different data type (byte) and range (0 to 255) from the original input data. When converting data with BYTSCL for display purposes, you may want to keep your original data as a separate variable for statistical and numerical analysis.

The following example shows how to use the BYTSCL function to scale data with values greater than 255, producing a more uniform display. This example uses a Magnetic Resonance Image (MRI) of a human brain within the `mr_brain.dcm` file in the `examples/data` directory. The values of this data are unsigned integer and range from 0 to about 800.

For code that you can copy and paste into an Editor window, see “Example Code: Byte-Scaling” on page 416 or complete the following steps for a detailed description of the process.

1. Import the image from the `mr_brain.dcm` file:

```
file = FILEPATH('mr_brain.dcm', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_DICOM(file)
imageSize = SIZE(image, /DIMENSIONS)
```

2. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 5
```

3. Create a window and display the original image:

```
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original Image'
TV, image
```

The following figure shows the original image.

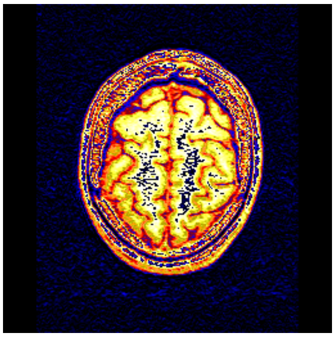


*Figure 10-1: Magnetic Resonance Image (MRI) of a Human Brain*

4. Byte-scale the image:

```
scaledImage = BYTSCL(image)
```

5. Create another window and display the byte-scaled image:

```
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Byte-Scaled Image'
TV, scaledImage
```

The following figure shows the result of byte-scaling. Unlike the original image, the byte-scaled image accurately represents the maximum and minimum pixel values by linearly adjusting the range for display.

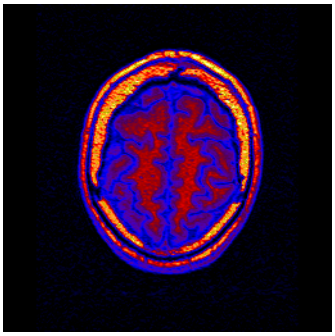


*Figure 10-2: Byte-Scaled MRI*

## Example Code: Byte-Scaling

Copy and paste the following text into an IDL Editor window. After saving the file as `ByteScaling.pro`, compile and run the program to reproduce the previous example.

```
PRO ByteScaling

; Import the image from the file.
file = FILEPATH('mr_brain.dcm', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_DICOM(file)
imageSize = SIZE(image, /DIMENSIONS)

; Initialize the displays.
DEVICE, DECOMPOSED = 0
LOADCT, 5

; Create a window and display the original image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original Image'
TV, image

; Byte-scale the image.
scaledImage = BYTSCL(image)

; Create another window and display the byte-scaled
; image.
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Byte-Scaled Image'
TV, scaledImage

END
```

## Working with Histograms

The histogram of an image shows the number of pixels for each pixel value within the range of the image. Peaks in the histogram represent more common values within the image that usually consist of nearly uniform regions. Valleys in the histogram represent less common values. Empty regions within the histogram indicate that no pixels within the image contain those values.

The following figure shows an example of a histogram and its related image. The most common value in this image is 180, composing the background of the image. Although the background appears nearly uniform, it contains many small variations.

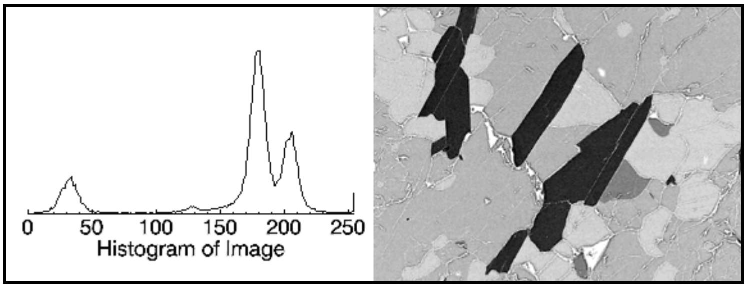


*Figure 10-3: Example of a Histogram (left) and Its Related Image (right)*

The contrast of these variations can be increased by equalizing the image’s histogram. Either the image’s color table or the image itself can be equalized based on the information within the image’s histogram. This section shows how to enhance the contrast within an image by modifying the image itself. See “Showing Variations in Uniform Areas” in Chapter 3 for more information on enhancing contrast by modifying the color table of an image using the image’s histogram information.

During histogram equalization, the values occurring in the empty regions of the histogram are redistributed equally among the peaks and valleys. This process creates intensity gradients within these regions (replacing nearly uniform values), thus highlighting minor variations.

IDL contains the ability to perform histogram equalization and adaptive histogram equalization. The following sections show how to use these forms of histogram equalization to modify images within IDL:

- “Equalizing with Histograms”
- “Adaptive Equalizing with Histograms” on page 422

## Equalizing with Histograms

You can use the HIST_EQUAL function to perform basic histogram equalization within IDL. Unlike histogram equalization methods performed on color tables, the HIST_EQUAL function results in a modified image, which has a different histogram than the original image. The resulting image shows more variations (increased contrast) within uniform areas than the original image.

The following example applies histogram equalization to an image of mineral deposits to reveal previously indistinguishable features. This example uses the `mineral.png` file in the `examples/data` directory.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Equalizing with Histograms" on page 421 or complete the following steps for a detailed description of the process.

1. Import the image and color table from the `mineral.png` file:

```
file = FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_PNG(file, red, green, blue)
imageSize = SIZE(image, /DIMENSIONS)
```

2. Initialize the display:

```
DEVICE, DECOMPOSED = 0
TVLCT, red, green, blue
```

3. Create a window and display the original image with its color table:

```
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original Image'
TV, image
```

The following figure shows the original image.

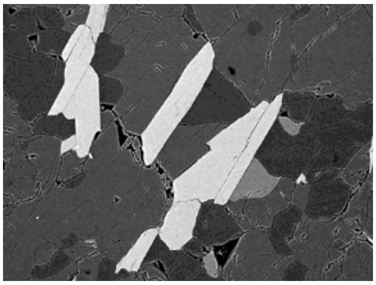


*Figure 10-4: The Mineral Image and Its Related Color Table*

4. Create another window and display the histogram of the original image:

```
WINDOW, 1, TITLE = 'Histogram of Image'
PLOT, HISTOGRAM(image), /XSTYLE, /YSTYLE, $
   TITLE = 'Mineral Image Histogram', $
   XTITLE = 'Intensity Value', $
   YTITLE = 'Number of Pixels of That Value'
```

The following figure shows the original image's histogram.

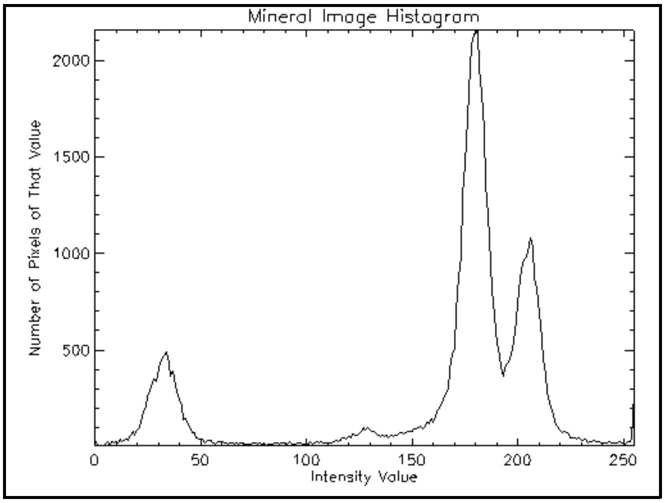


*Figure 10-5: Histogram of the Original Image*

5. Histogram equalize the image:

```
equalizedImage = HIST_EQUAL(image)
```

6. Create another window and display the equalized image:

```
WINDOW, 2, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Equalized Image'
TV, equalizedImage
```

The following figure shows the results of the histogram equalization. Small variations within the uniform regions are now much more noticeable.

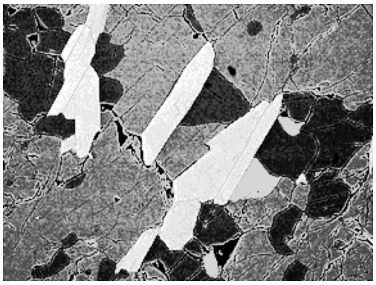


*Figure 10-6: Equalized Mineral Image*

7. Create another window and display the histogram of the equalized image:

```
WINDOW, 3, TITLE = 'Histogram of Equalized Image'
PLOT, HISTOGRAM(equalizedImage), /XSTYLE, /YSTYLE, $
   TITLE = 'Equalized Image Histogram', $
   XTITLE = 'Intensity Value', $
   YTITLE = 'Number of Pixels of That Value'
```

The following figure shows the modified image’s histogram. The resulting histogram is now more uniform than the original histogram.

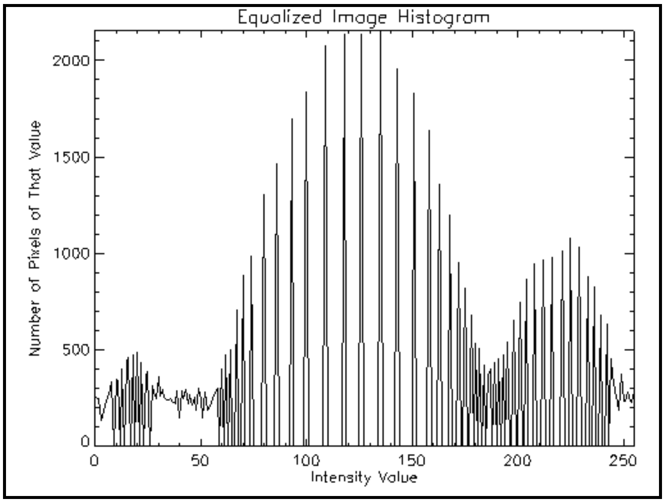


*Figure 10-7: Histogram of the Equalized Image*

## Example Code: Equalizing with Histograms

Copy and paste the following text into an IDL Editor window. After saving the file as `Equalizing.pro`, compile and run the program to reproduce the previous example.

```
PRO Equalizing

; Import the image from the file.
file = FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_PNG(file, red, green, blue)
imageSize = SIZE(image, /DIMENSIONS)

; Initialize the display.
DEVICE, DECOMPOSED = 0
TVLCT, red, green, blue

; Create a window and display the original image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original Image'
TV, image

; Create another window and display the histogram of the
; the original image.
WINDOW, 1, TITLE = 'Histogram of Image'
PLOT, HISTOGRAM(image), /XSTYLE, /YSTYLE, $
   TITLE = 'Mineral Image Histogram', $
   XTITLE = 'Intensity Value', $
   YTITLE = 'Number of Pixels of That Value'

; Histogram-equalize the image.
equalizedImage = HIST_EQUAL(image)

; Create another window and display the equalized image.
WINDOW, 2, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Equalized Image'
TV, equalizedImage

; Create another window and display the histogram of the
; equalizied image.
WINDOW, 3, TITLE = 'Histogram of Equalized Image'
PLOT, HISTOGRAM(equalizedImage), /XSTYLE, /YSTYLE, $
   TITLE = 'Equalized Image Histogram', $
   XTITLE = 'Intensity Value', $
   YTITLE = 'Number of Pixels of That Value'

END
```

## Adaptive Equalizing with Histograms

Adaptive histogram equalization involves applying equalization based on the local region surrounding each pixel. Each pixel is mapped to an intensity proportional to its rank within the surrounding neighborhood. This type of equalization also tends to reduce the disparity between peaks and valleys within the image's histogram.

You can use the ADAPT_HIST_EQUAL function to perform the adaptive histogram equalization process within IDL. Like the HIST_EQUAL function, the ADAPT_HIST_EQUAL function results in a modified image, which has a different histogram than the original image.

The following example applies adaptive histogram equalization to an image of mineral deposits to reveal previously indistinguishable features. This example uses a the `mineral.png` file in the `examples/data` directory.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Adaptive Equalizing with Histograms" on page 426 or complete the following steps for a detailed description of the process.

1. Import the image and color table from the `mineral.png` file:

```
file = FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_PNG(file, red, green, blue)
imageSize = SIZE(image, /DIMENSIONS)
```

2. Initialize the display:

```
DEVICE, DECOMPOSED = 0
TVLCT, red, green, blue
```

3. Create a window and display the original image with its color table:

```
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original Image'
TV, image
```

The following figure shows the original image.

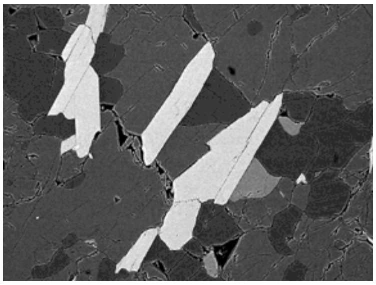


*Figure 10-8: The Mineral Image and Its Related Color Table*

4. Create another window and display the histogram of the original image:

```
WINDOW, 1, TITLE = 'Histogram of Image'
PLOT, HISTOGRAM(image), /XSTYLE, /YSTYLE, $
   TITLE = 'Mineral Image Histogram', $
   XTITLE = 'Intensity Value', $
   YTITLE = 'Number of Pixels of That Value'
```

The following figure shows the resulting display.

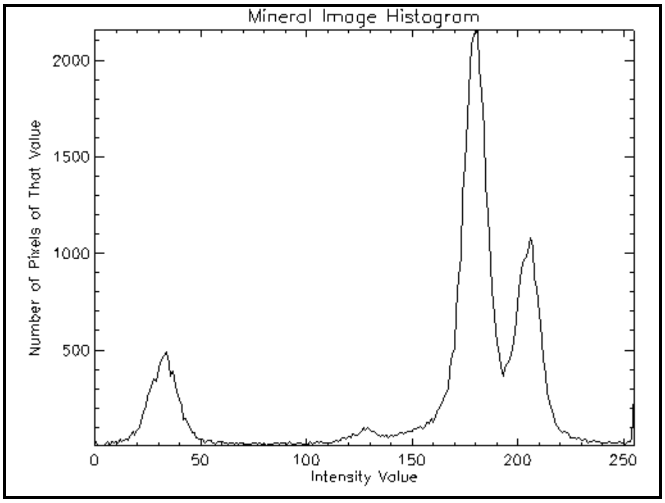


*Figure 10-9: Histogram of the Original Image*

5. Apply adaptive histogram equalization to the image:

```
equalizedImage = ADAPT_HIST_EQUAL(image)
```

6. Create another window and display the equalized image:

```
WINDOW, 2, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Adaptive Equalized Image'
TV, equalizedImage
```

The following figure shows the results of adaptive histogram equalization. All the variations within the image are now noticeable.

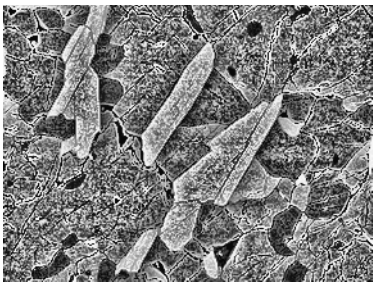


*Figure 10-10: Adaptive Equalized Mineral Image*

7. Create another window and display the histogram of the equalized image:

```
WINDOW, 3, TITLE = 'Histogram of Adaptive Equalized Image'
PLOT, HISTOGRAM(equalizedImage), /XSTYLE, /YSTYLE, $
   TITLE = 'Adaptive Equalized Image Histogram', $
   XTITLE = 'Intensity Value', $
   YTITLE = 'Number of Pixels of That Value'
```

The following figure shows the modified image's histogram. The resulting histogram contains no empty regions and fewer extreme peaks and valleys than the original image.

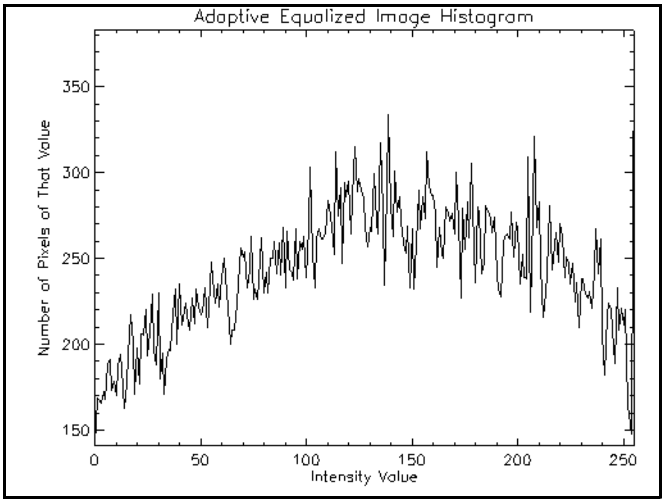


*Figure 10-11: Histogram of the Adaptive Equalized Image*

## Example Code: Adaptive Equalizing with Histograms

Copy and paste the following text into an IDL Editor window. After saving the file as `AdaptiveEqualizing.pro`, compile and run the program to reproduce the previous example.

```
PRO AdaptiveEqualizing

; Import the image from the file.
file = FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_PNG(file, red, green, blue)
imageSize = SIZE(image, /DIMENSIONS)

; Initialize the display.
DEVICE, DECOMPOSED = 0
TVLCT, red, green, blue

; Create a window and display the original image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original Image'
TV, image
```

```
; Create another window and display the histogram of the
; the original image.
WINDOW, 1, TITLE = 'Histogram of Image'
PLOT, HISTOGRAM(image), /XSTYLE, /YSTYLE, $
   TITLE = 'Mineral Image Histogram', $
   XTITLE = 'Intensity Value', $
   YTITLE = 'Number of Pixels of That Value'

; Histogram-equalize the image.
equalizedImage = ADAPT_HIST_EQUAL(image)

; Create another window and display the equalized image.
WINDOW, 2, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Adaptive Equalized Image'
TV, equalizedImage

; Create another window and display the histogram of the
; equalizied image.
WINDOW, 3, TITLE = 'Histogram of Adaptive Equalized Image'
PLOT, HISTOGRAM(equalizedImage), /XSTYLE, /YSTYLE, $
   TITLE = 'Adaptive Equalized Image Histogram', $
   XTITLE = 'Intensity Value', $
   YTITLE = 'Number of Pixels of That Value'

END
```

## Filtering an Image

Image filtering is useful for many applications, including smoothing, sharpening, removing noise, and edge detection. A filter is defined by a kernel, which is a small array applied to each pixel and its neighbors within an image. In most applications, the center of the kernel is aligned with the current pixel, and is a square with an odd number (3, 5, 7, etc.) of elements in each dimension. The process used to apply filters to an image is known as convolution, and may be applied in either the spatial or frequency domain. See Chapter 9, "Overview of Transforming Between Image Domains" for more information on image domains.

Within the spatial domain, the first part of the convolution process multiplies the elements of the kernel by the matching pixel values when the kernel is centered over a pixel. The elements of the resulting array (which is the same size as the kernel) are averaged, and the original pixel value is replaced with this result. The CONVOL function performs this convolution process for an entire image.

Within the frequency domain, convolution can be performed by multiplying the FFT (Fast Fourier Transform) of the image by the FFT of the kernel, and then transforming back into the spatial domain. The kernel is padded with zero values to enlarge it to the same size as the image before the forward FFT is applied. These types of filters are usually specified within the frequency domain and do not need to be transformed. IDL's DIST and HANNING functions are examples of filters already transformed into the frequency domain. See "Windowing to Remove Noise" on page 470 for more information on these types of filters.

The following examples in this section will focus on some of the basic filters applied within the spatial domain using the CONVOL function:

- "Low Pass Filtering" on page 429
- "High Pass Filtering" on page 433
- "Directional Filtering" on page 438
- "Laplacian Filtering" on page 442

Since filters are the building blocks of many image processing methods, these examples merely show how to apply filters, as opposed to showing how a specific filter may be used to enhance a specific image or extract a specific shape. This basic introduction provides the information necessary to accomplish more advanced image-specific processing.

**Note**

The following filters mentioned are not the only filters used in image processing. Most image processing textbooks contain more varieties of filters.

## Low Pass Filtering

A low pass filter is the basis for most smoothing methods. An image is smoothed by decreasing the disparity between pixel values by averaging nearby pixels (see "Smoothing an Image" on page 448 for more information).

Using a low pass filter tends to retain the low frequency information within an image while reducing the high frequency information. An example is an array of ones divided by the number of elements within the kernel, such as the following 3 by 3 kernel:

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

**Note**

The above array is an example of one possible kernel for a low pass filter. Other filters may include more weighting for the center point, or have different smoothing in each dimension.

The following example shows how to use IDL's CONVOL function to smooth an aerial view of New York City within the `nyny.dat` file in the `examples/data` directory.

For code that you can copy and paste into an Editor window, see "Example Code: Low Pass Filtering" on page 432 or complete the following steps for a detailed description of the process.

1. Import the image from the `nyny.dat` file:

```
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [768, 512]
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Crop the image to focus in on the bridges:

```
croppedSize = [96, 96]
croppedImage = image[200:(croppedSize[0] - 1) + 200, $
   180:(croppedSize[1] - 1) + 180]
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
displaySize = [256, 256]
```

4. Create a window and display the cropped image:

```
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Cropped New York Image'
TVSCL, CONGRID(croppedImage, displaySize[0], $
   displaySize[1])
```

The following figure shows the cropped section of the original image.

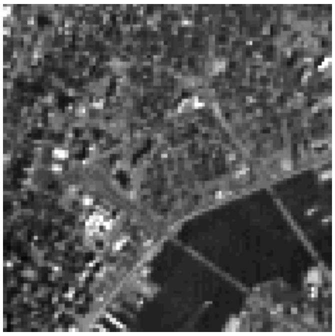


*Figure 10-12: Cropped New York Image*

5. Create a kernel for a low pass filter:

```
kernelSize = [3, 3]
kernel = REPLICATE((1./(kernelSize[0]*kernelSize[1])), $
   kernelSize[0], kernelSize[1])
```

6. Apply the filter to the image:

```
filteredImage = CONVOL(FLOAT(croppedImage), kernel, $
   /CENTER, /EDGE_TRUNCATE)
```

7. Create another window and display the resulting filtered image:

```
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Low Pass Filtered New York Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])
```

   The following figure shows the resulting display. The high frequency pixel values have been blurred as a result of the low pass filter.

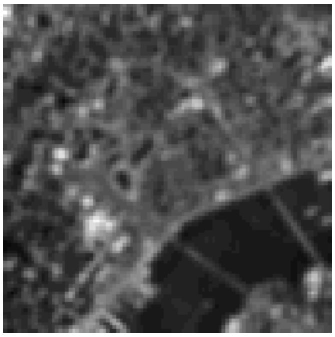


*Figure 10-13: Low Pass Filtered New York Image*

8. Add the original and the filtered image together to show how the filter effects the image.

```
WINDOW, 2, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Low Pass Combined New York Image'
TVSCL, CONGRID(croppedImage + filteredImage, $
   displaySize[0], displaySize[1])
```

The following figure shows the resulting display. In the resulting combined image, the structures within the city are not as pixelated as in the original image. The image is smoothed (blurred) to appear more continuous.

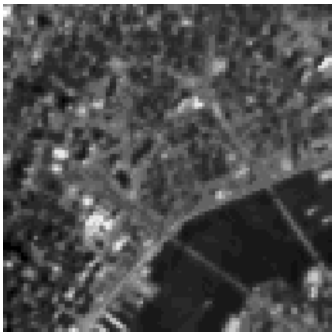


*Figure 10-14: Low Pass Combined New York Image*

## Example Code: Low Pass Filtering

Copy and paste the following text into an IDL Editor window. After saving the file as `LowPassFiltering.pro`, compile and run the program to reproduce the previous example.

```
PRO LowPassFiltering

; Import the image from the file.
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [768, 512]
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Crop the image to focus in on the bridges.
croppedSize = [96, 96]
croppedImage = image[200:(croppedSize[0] - 1) + 200, $
   180:(croppedSize[1] - 1) + 180]

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0
displaySize = [256, 256]

; Create a window and display the cropped image.
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Cropped New York Image'
```

```
TVSCL, CONGRID(croppedImage, displaySize[0], $
   displaySize[1])

; Create a low pass filter.
kernelSize = [3, 3]
kernel = REPLICATE((1./(kernelSize[0]*kernelSize[1])), $
   kernelSize[0], kernelSize[1])

; Apply the filter to the image.
filteredImage = CONVOL(FLOAT(croppedImage), kernel, $
   /CENTER, /EDGE_TRUNCATE)

; Create another window and display the resulting
; filtered image.
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Low Pass Filtered New York Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])

; Create another window and display the combined image.
WINDOW, 2, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Low Pass Combined New York Image'
TVSCL, CONGRID(croppedImage + filteredImage, $
   displaySize[0], displaySize[1])

END
```

## High Pass Filtering

A high pass filter is the basis for most sharpening methods. An image is sharpened when contrast is enhanced between adjoining areas with little variation in brightness or darkness (see "Sharpening an Image" on page 459 for more detailed information).

A high pass filter tends to retain the high frequency information within an image while reducing the low frequency information. The kernel of the high pass filter is designed to increase the brightness of the center pixel relative to neighboring pixels. The kernel array usually contains a single positive value at its center, which is completely surrounded by negative values. The following array is an example of a 3 by 3 kernel for a high pass filter:

$$\begin{bmatrix} -1/9 & -1/9 & -1/9 \\ -1/9 & 8/9 & -1/9 \\ -1/9 & -1/9 & -1/9 \end{bmatrix}$$

**Note**

The above array is an example of one possible kernel for a high pass filter. Other filters may include more weighting for the center point.

The following example shows how to use IDL's CONVOL function with a 3 by 3 high pass filter to sharpen an aerial view of New York City within the `nyny.dat` file in the `examples/data` directory.

For code that you can copy and paste into an Editor window, see "Example Code: High Pass Filtering" on page 437 or complete the following steps for a detailed description of the process.

1. Import the image from the `nyny.dat` file:

```
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [768, 512]
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Crop the image to focus in on the bridges:

```
croppedSize = [96, 96]
croppedImage = image[200:(croppedSize[0] - 1) + 200, $
   180:(croppedSize[1] - 1) + 180]
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
displaySize = [256, 256]
```

4. Create a window and display the cropped image:

```
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Cropped New York Image'
TVSCL, CONGRID(croppedImage, displaySize[0], $
   displaySize[1])
```

The following figure shows the cropped section of the original image.

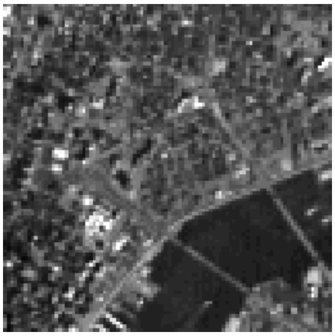


*Figure 10-15: Cropped New York Image*

5. Create a kernel for a high pass filter:

```
kernelSize = [3, 3]
kernel = REPLICATE(-1., kernelSize[0], kernelSize[1])
kernel[1, 1] = 8.
```

6. Apply the filter to the image:

```
filteredImage = CONVOL(FLOAT(croppedImage), kernel, $
   /CENTER, /EDGE_TRUNCATE)
```

7. Create another window and display the resulting filtered image:

```
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'High Pass Filtered New York Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])
```

The following figure shows the results of applying the high pass filter. The high frequency information is retained.

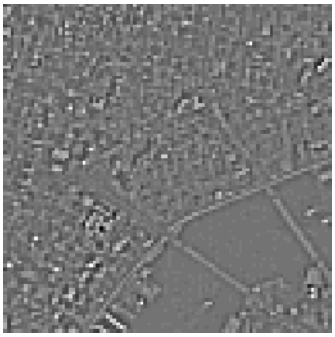


*Figure 10-16: High Pass Filtered New York Image*

8. Add the original and the filtered image together to show how the filter effects the image.

```
WINDOW, 2, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'High Pass Combined New York Image'
TVSCL, CONGRID(croppedImage + filteredImage, $
   displaySize[0], displaySize[1])
```

The following figure shows the resulting display. In the resulting combined image, the structures within the city are more pixelated than in the original image. The pixels are highlighted and appear more discontinuous, exposing the three-dimensional nature of the structures within the image.

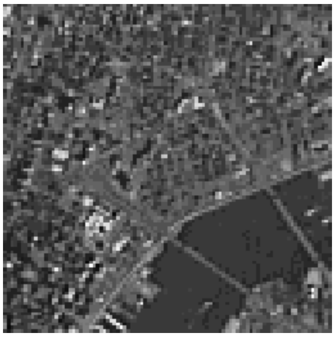


*Figure 10-17: High Pass Combined New York Image*

## Example Code: High Pass Filtering

Copy and paste the following text into an IDL Editor window. After saving the file as `HighPassFiltering.pro`, compile and run the program to reproduce the previous example.

```
PRO HighPassFiltering

; Import the image from the file.
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [768, 512]
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Crop the image to focus in on the bridges.
croppedSize = [96, 96]
croppedImage = image[200:(croppedSize[0] - 1) + 200, $
   180:(croppedSize[1] - 1) + 180]

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0
displaySize = [256, 256]

; Create a window and display the cropped image.
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Cropped New York Image'
TVSCL, CONGRID(croppedImage, displaySize[0], $
   displaySize[1])

; Create a high pass filter.
kernelSize = [3, 3]
kernel = REPLICATE(-1./9., kernelSize[0], kernelSize[1])
kernel[1, 1] = 8./9.

; Apply the filter to the image.
filteredImage = CONVOL(FLOAT(croppedImage), kernel, $
   /CENTER, /EDGE_TRUNCATE)

; Create another window and display the resulting
; filtered image.
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'High Pass Filtered New York Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])

; Create another window and display the combined images.
WINDOW, 2, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'High Pass Combined New York Image'
```

```
TVSCL, CONGRID(croppedImage + filteredImage, $
   displaySize[0], displaySize[1])

END
```

## Directional Filtering

A directional filter forms the basis for some edge detection methods. An edge within an image is visible when a large change (a steep gradient) occurs between adjacent pixel values. This change in values is measured by the first derivatives (often referred to as slopes) of an image. Directional filters can be used to compute the first derivatives of an image (see "Detecting Edges" on page 464 for more information on edge detection).

Directional filters can be designed for any direction within a given space. For images, *x*- and *y*-directional filters are commonly used to compute derivatives in their respective directions. The following array is an example of a 3 by 3 kernel for an *x*-directional filter (the kernel for the *y*-direction is the transpose of this kernel):

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

**Note**

The above array is an example of one possible kernel for a *x*-directional filter. Other filters may include more weighting in the center of the nonzero columns.

The following example shows how to use IDL's CONVOL function to determine the first derivatives of an image in the *x*-direction. The resulting derivatives are then scaled to just show negative, zero, and positive slopes. This example uses the aerial view of New York City within the `nyny.dat` file in the `examples/data` directory.

For code that you can copy and paste into an Editor window, see "Example Code: Directional Filtering" on page 441 or complete the following steps for a detailed description of the process.

1. Import the image from the `nyny.dat` file:

```
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [768, 512]
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Crop the image to focus in on the bridges:

```
croppedSize = [96, 96]
croppedImage = image[200:(croppedSize[0] - 1) + 200, $
   180:(croppedSize[1] - 1) + 180]
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
displaySize = [256, 256]
```

4. Create a window and display the cropped image:

```
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Cropped New York Image'
TVSCL, CONGRID(croppedImage, displaySize[0], $
   displaySize[1])
```

The following figure shows the cropped section of the original image.

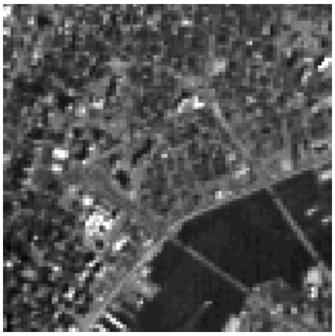


*Figure 10-18: Cropped New York Image*

5. Create a kernel for an *x*-directional filter:

```
kernelSize = [3, 3]
kernel = FLTARR(kernelSize[0], kernelSize[1])
kernel[0, *] = -1.
kernel[2, *] = 1.
```

6. Apply the filter to the image:

```
filteredImage = CONVOL(FLOAT(croppedImage), kernel, $
   /CENTER, /EDGE_TRUNCATE)
```

7. Create another window and display the resulting filtered image:

```
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Direction Filtered New York Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])
```

   The resulting image shows some edge information. The most noticeable edge is seen as a “shadow” for each bridge. This information represents the slopes in the x-direction of the image. The filtered image can then be scaled to highlight these slopes.

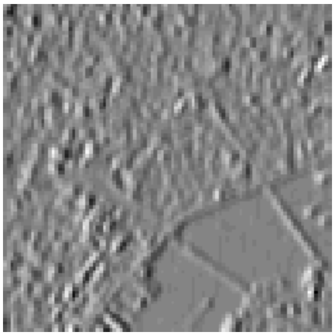


*Figure 10-19: Direction Filtered New York Image*

8. Create another window and display negative slopes as black, zero slopes as gray, and positive slopes as white:

```
WINDOW, 2, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Slopes of Direction Filtered New York Image'
TVSCL, CONGRID(-1 > FIX(filteredImage/50) < 1,
displaySize[0], $
   displaySize[1])
```

   The following figure shows the negative slopes (black areas), zero slopes (gray areas), and positive slopes (white areas) produced by the *x*-directional filter.

The adjacent black and white areas show edges in the *x*-direction, such as along the bridge closest to the right side of the image.

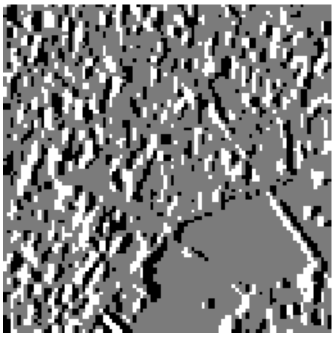


*Figure 10-20: Slopes of Direction Filtered New York Image*

## Example Code: Directional Filtering

Copy and paste the following text into an IDL Editor window. After saving the file as `DirectionFiltering.pro`, compile and run the program to reproduce the previous example.

```
PRO DirectionFiltering

; Import the image from the file.
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [768, 512]
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Crop the image to focus in on the bridges.
croppedSize = [96, 96]
croppedImage = image[200:(croppedSize[0] - 1) + 200, $
   180:(croppedSize[1] - 1) + 180]

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0
displaySize = [256, 256]

; Create a window and display the cropped image.
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Cropped New York Image'
TVSCL, CONGRID(croppedImage, displaySize[0], $
   displaySize[1])
```

```
; Create a directional filter.
kernelSize = [3, 3]
kernel = FLTARR(kernelSize[0], kernelSize[1])
kernel[0, *] = -1.
kernel[2, *] = 1.

; Apply the filter to the image.
filteredImage = CONVOL(FLOAT(croppedImage), kernel, $
   /CENTER, /EDGE_TRUNCATE)

; Create another window and display the resulting
; filtered image.
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Direction Filtered New York Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])

; Create another window and display negative slopes as
; black, zero slopes as gray, and positive slopes as
; white.
WINDOW, 2, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Slopes of Direction Filtered New York Image'
TVSCL, CONGRID(-1 > FIX(filteredImage/50) < 1, displaySize[0], $
   displaySize[1])

END
```

## Laplacian Filtering

A Laplacian filter forms another basis for edge detection methods. A Laplacian filter can be used to compute the second derivatives of an image, which measure the rate at which the first derivatives change. This helps to determine if a change in adjacent pixel values is an edge or a continuous progression (see "Detecting Edges" on page 464 for more information on edge detection).

Kernels of Laplacian filters usually contain negative values in a cross pattern (similar to a plus sign), which is centered within the array. The corners are either zero or positive values. The center value can be either negative or positive. The following array is an example of a 3 by 3 kernel for a Laplacian filter:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

**Note**

The above array is an example of one possible kernel for a Laplacian filter. Other filters may include positive, nonzero values in the corners and more weighting in the centered cross pattern.

The following example shows how to use IDL's CONVOL function with a 3 by 3 Laplacian filter to determine the second derivatives of an image. This type of information is used within edge detection processes to find ridges. This example uses an aerial view of New York City within the `nyny.dat` file in the `examples/data` directory.

For code that you can copy and paste into an Editor window, see "Example Code: Laplacian Filtering" on page 446 or complete the following steps for a detailed description of the process.

1. Import the image from the `nyny.dat` file:

```
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [768, 512]
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Crop the image to focus in on the bridges:

```
croppedSize = [96, 96]
croppedImage = image[200:(croppedSize[0] - 1) + 200, $
   180:(croppedSize[1] - 1) + 180]
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
displaySize = [256, 256]
```

4. Create a window and display the cropped image:

```
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Cropped New York Image'
TVSCL, CONGRID(croppedImage, displaySize[0], $
   displaySize[1])
```

The following figure shows the cropped section of the original image.

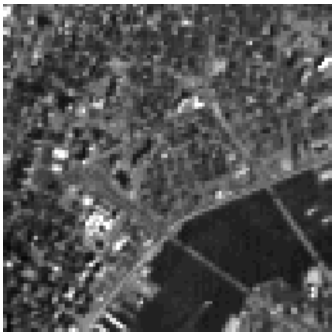


*Figure 10-21: Cropped New York Image*

5. Create a kernel of a Laplacian filter:

```
kernelSize = [3, 3]
kernel = FLTARR(kernelSize[0], kernelSize[1])
kernel[1, *] = -1.
kernel[*, 1] = -1.
kernel[1, 1] = 4.
```

6. Apply the filter to the image:

```
filteredImage = CONVOL(FLOAT(croppedImage), kernel, $
   /CENTER, /EDGE_TRUNCATE)
```

7. Create another window and display the resulting filtered image:

```
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Laplace Filtered New York Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])
```

The following figure contains positive and negative second derivative information. The positive values represent depressions (valleys) and the negative values represent ridges.

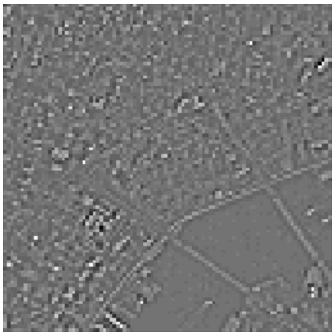


*Figure 10-22: Laplacian Filtered New York Image*

8. Create another window and display only the negative values (ridges) within the image:

```
WINDOW, 2, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Negative Values of Laplace Filtered New York
Image'
TVSCL, CONGRID(filteredImage < 0, $
   displaySize[0], displaySize[1])
```

The following figure shows the negative values produced by the Laplacian filter. The most noticeable ridges in this result are the medians within the wide boulevards of the city.

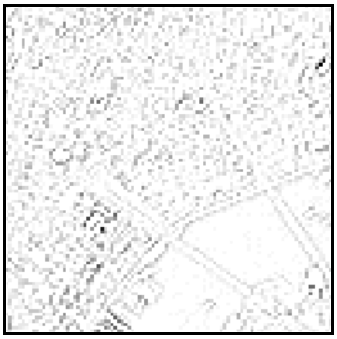


*Figure 10-23: Negative Values of Laplacian Filtered New York Image*

## Example Code: Laplacian Filtering

Copy and paste the following text into an IDL Editor window. After saving the file as `LaplaceFiltering.pro`, compile and run the program to reproduce the previous example.

```
PRO LaplaceFiltering

; Import the image from the file.
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [768, 512]
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Crop the image to focus in on the bridges.
croppedSize = [96, 96]
croppedImage = image[200:(croppedSize[0] - 1) + 200, $
   180:(croppedSize[1] - 1) + 180]

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0
displaySize = [256, 256]

; Create a window and display the cropped image.
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Cropped New York Image'
```

```
TVSCL, CONGRID(croppedImage, displaySize[0], $
   displaySize[1])

; Create a Laplacian filter.
kernelSize = [3, 3]
kernel = FLTARR(kernelSize[0], kernelSize[1])
kernel[1, *] = -1.
kernel[*, 1] = -1.
kernel[1, 1] = 4.

; Apply the filter to the image.
filteredImage = CONVOL(FLOAT(croppedImage), kernel, $
   /CENTER, /EDGE_TRUNCATE)

; Create another window and display the resulting
; filtered image.
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Laplace Filtered New York Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])
PRINT, MIN(filteredImage), MAX(filteredImage)

; Create another window and display only the negative
; values of the image.
WINDOW, 2, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Negative Values of Laplace Filtered New York Image'
TVSCL, CONGRID(filteredImage < 0, $
   displaySize[0], displaySize[1])

END
```

## Smoothing an Image

Smoothing is often used to reduce noise within an image or to produce a less pixelated image. Most smoothing methods are based on low pass filters. See "Low Pass Filtering" on page 429 for more information.

Smoothing is also usually based on a single value representing the image, such as the average value of the image or the middle (median) value. The following examples show how to smooth using average and middle values:

- "Smoothing with Average Values"
- "Smoothing with Median Values" on page 453

### Smoothing with Average Values

The following example shows how to use the SMOOTH function to smooth an image with a moving average. Surfaces of the original and smooth images are displayed to show how discontinuous values are made more continuous. This example uses the photomicrograph image of human red blood cells contained within the `rbcells.jpg` file in the `examples/data` directory.

For code that you can copy and paste into an Editor window, see "Example Code: Smoothing with Average Values" on page 452 or complete the following steps for a detailed description of the process.

1. Import the image from the `rbcells.jpg` file:

```
file = FILEPATH('rbcells.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, image
imageSize = SIZE(image, /DIMENSIONS)
```

2. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

3. Create a window and display the original image:

```
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original Image'
TV, image
```

The following figure shows the original image. This image contains many varying pixel values within the background.

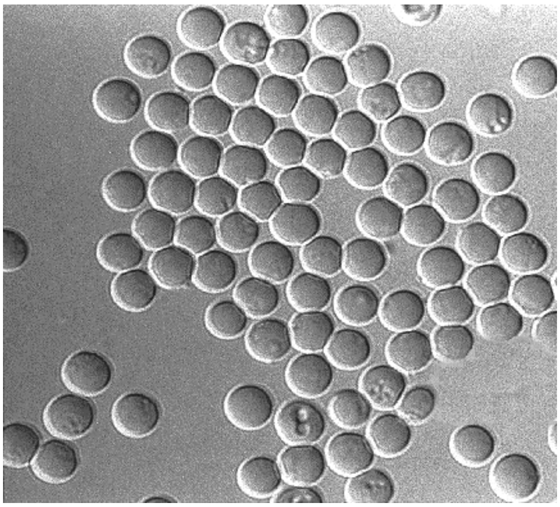


*Figure 10-24: Original Red Blood Cells Image*

4. Create another window and display the original image as a surface:

```
WINDOW, 1, TITLE = 'Original Image as a Surface'
SHADE_SURF, image, /XSTYLE, /YSTYLE, CHARSIZE = 2., $
   XTITLE = 'Width Pixels', $
   YTITLE = 'Height Pixels', $
   ZTITLE = 'Intensity Values', $
   TITLE = 'Red Blood Cell Image'
```

The following figure shows the surface of the original image. This image contains many discontinuous values shown as sharp peaks (spikes) in the middle range of values.

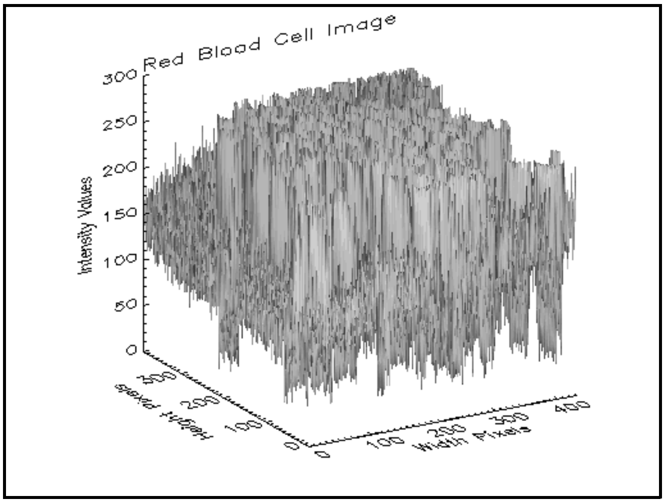


*Figure 10-25: Surface of Original Red Blood Cells Image*

5. Smooth the image with the SMOOTH function, which uses the average value of each group of pixels affected by the 5 by 5 kernel applied to the image:

```
smoothedImage = SMOOTH(image, 5, /EDGE_TRUNCATE)
```

The *width* argument of 5 is used to specify that a 5 by 5 smoothing kernel is to be used.

6. Create another window and display the smoothed image as a surface:

```
WINDOW, 2, TITLE = 'Smoothed Image as a Surface'
SHADE_SURF, smoothedImage, /XSTYLE, /YSTYLE, CHARSIZE = 2., $
   XTITLE = 'Width Pixels', $
   YTITLE = 'Height Pixels', $
   ZTITLE = 'Intensity Values', $
   TITLE = 'Smoothed Cell Image'
```

The following figure shows the surface of the smoothed image. The sharp peaks in the original image have been decreased.

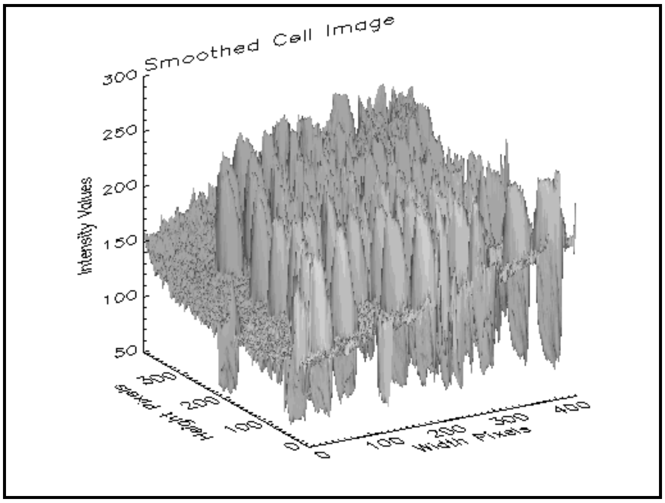


*Figure 10-26: Surface of Average-Smoothed Red Blood Cells Image*

7. Create another window and display the smoothed image:

```
WINDOW, 3, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Smoothed Image'
TV, smoothedImage
```

The following figure shows the smoothed image. Less variations between pixel values occur within the background of the resulting image.

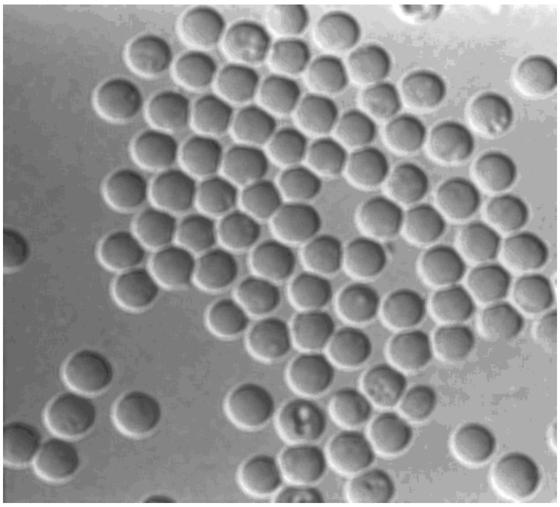


*Figure 10-27: Average-Smoothed Red Blood Cells Image*

## Example Code: Smoothing with Average Values

Copy and paste the following text into an IDL Editor window. After saving the file as `SmoothingWithSMOOTH.pro`, compile and run the program to reproduce the previous example.

```
PRO SmoothingWithSMOOTH

; Import the image from the file.
file = FILEPATH('rbcells.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, image
imageSize = SIZE(image, /DIMENSIONS)

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the original image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original Image'
TV, image
```

```
; Create another window and display the original image
; as a surface.
WINDOW, 1, TITLE = 'Original Image as a Surface'
SHADE_SURF, image, /XSTYLE, /YSTYLE, CHARSIZE = 2., $
   XTITLE = 'Width Pixels', $
   YTITLE = 'Height Pixels', $
   ZTITLE = 'Intensity Values', $
   TITLE = 'Red Blood Cell Image'

; Smooth the image with the SMOOTH function, which uses
; the averages of image values.
smoothedImage = SMOOTH(image, 5, /EDGE_TRUNCATE)

; Create another window and display the smoothed image
; as a surface.
WINDOW, 2, TITLE = 'Smoothed Image as a Surface'
SHADE_SURF, smoothedImage, /XSTYLE, /YSTYLE, CHARSIZE = 2., $
   XTITLE = 'Width Pixels', $
   YTITLE = 'Height Pixels', $
   ZTITLE = 'Intensity Values', $
   TITLE = 'Smoothed Cell Image'

; Create another window and display the smoothed image.
WINDOW, 3, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Smoothed Image'
TV, smoothedImage

END
```

## Smoothing with Median Values

The following example shows how to use IDL’s MEDIAN function to smooth an image by median values. Surfaces of the original and smooth images are displayed to show how discontinuous values are made more continuous. This example uses the photomicrograph image of human red blood cells contained within the `rbcells.jpg` file in the `examples/data` directory.

For code that you can copy and paste into an Editor window, see “Example Code: Smoothing with Median Values” on page 457 or complete the following steps for a detailed description of the process.

1. Import the image from the `rbcells.jpg` file:

```
file = FILEPATH('rbcells.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, image
imageSize = SIZE(image, /DIMENSIONS)
```

2. Initialize the display:

   ```
   DEVICE, DECOMPOSED = 0
   LOADCT, 0
   ```

3. Create a window and display the original image:

   ```
   WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'Original Image'
   TV, image
   ```

   The following figure shows the original image. This image contains many varying pixel values within the background.

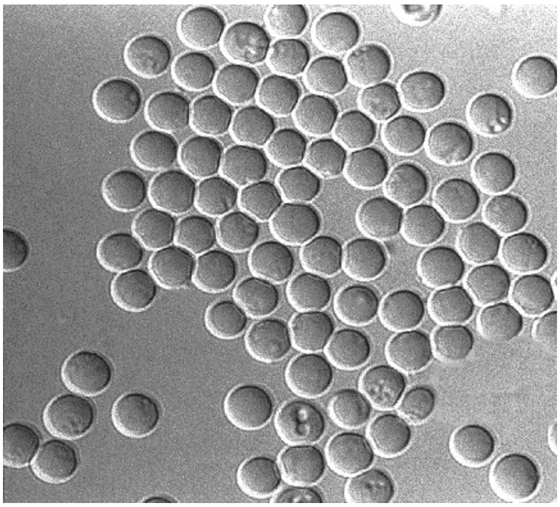


*Figure 10-28: Original Red Blood Cells Image*

4. Create another window and display the original image as a surface:

   ```
   WINDOW, 1, TITLE = 'Original Image as a Surface'
   SHADE_SURF, image, /XSTYLE, /YSTYLE, CHARSIZE = 2., $
      XTITLE = 'Width Pixels', $
      YTITLE = 'Height Pixels', $
      ZTITLE = 'Intensity Values', $
      TITLE = 'Red Blood Cell Image'
   ```

The following figure shows the surface of the original display. This image contains many discontinuous values shown as sharp peaks (spikes) in the middle range of values.

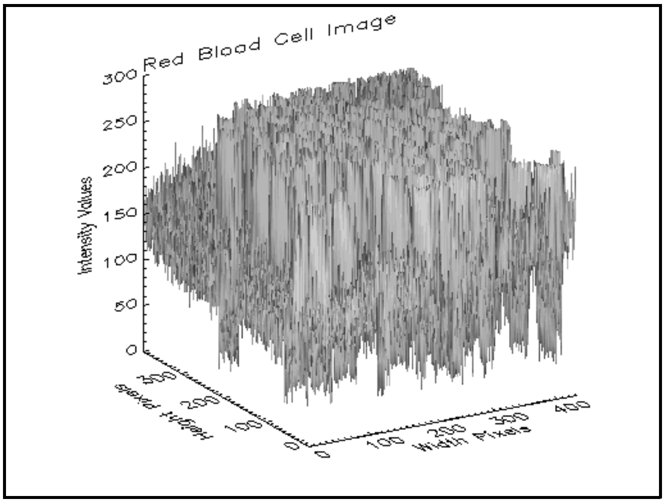


*Figure 10-29: Surface of Original Red Blood Cells Image*

5. Smooth the image with the MEDIAN function, which uses the middle value of each group of pixels affected by the 5 by 5 kernel applied to the image:

```
smoothedImage = MEDIAN(image, 5)
```

6. Create another window and display the smoothed image as a surface:

```
WINDOW, 2, TITLE = 'Smoothed Image as a Surface'
SHADE_SURF, smoothedImage, /XSTYLE, /YSTYLE, CHARSIZE = 2., $
   XTITLE = 'Width Pixels', $
   YTITLE = 'Height Pixels', $
   ZTITLE = 'Intensity Values', $
   TITLE = 'Smoothed Cell Image'
```

The following figure shows the smoothed surface. The sharp peaks in the original image are decreased by the MEDIAN function.

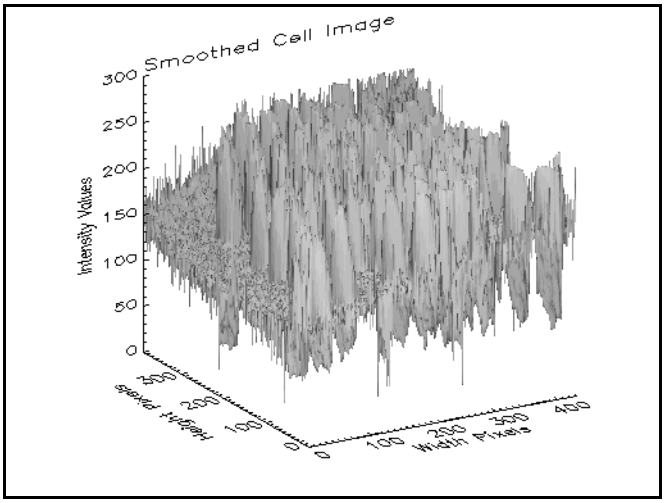


*Figure 10-30: Surface of Middle-Smoothed Red Blood Cells Image*

7. Create another window and display the smoothed image:

```
WINDOW, 3, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Smoothed Image'
TV, smoothedImage
```

The following figure shows the results of applying the median filter. Less variations occur within the background of the resulting image, yet feature edges remain clearly defined.

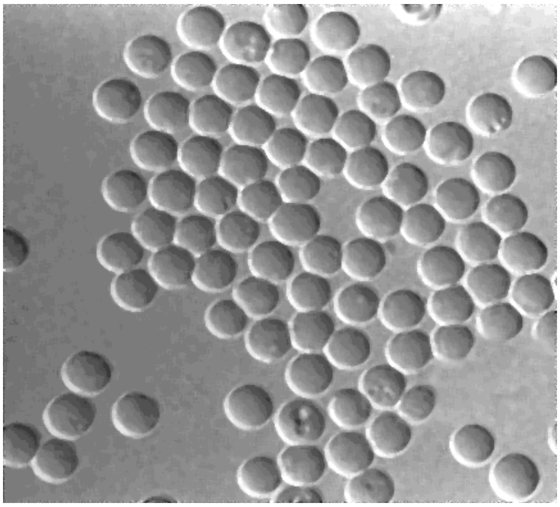


*Figure 10-31: Middle-Smoothed Red Blood Cells Image*

## Example Code: Smoothing with Median Values

Copy and paste the following text into an IDL Editor window. After saving the file as `SmoothingWithMEDIAN.pro`, compile and run the program to reproduce the previous example.

```
PRO SmoothingWithMEDIAN

; Import the image from the file.
file = FILEPATH('rbcells.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, image
imageSize = SIZE(image, /DIMENSIONS)

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the original image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original Image'
TV, image
```

```
; Create another window and display the original image
; as a surface.
WINDOW, 1, TITLE = 'Original Image as a Surface'
SHADE_SURF, image, /XSTYLE, /YSTYLE, CHARSIZE = 2., $
   XTITLE = 'Width Pixels', $
   YTITLE = 'Height Pixels', $
   ZTITLE = 'Intensity Values', $
   TITLE = 'Red Blood Cell Image'

; Smooth the image with the MEDIAN function, which uses
; the middle values of image.
smoothedImage = MEDIAN(image, 5)

; Create another window and display the smoothed image
; as a surface.
WINDOW, 2, TITLE = 'Smoothed Image as a Surface'
SHADE_SURF, smoothedImage, /XSTYLE, /YSTYLE, CHARSIZE = 2., $
   XTITLE = 'Width Pixels', $
   YTITLE = 'Height Pixels', $
   ZTITLE = 'Intensity Values', $
   TITLE = 'Smoothed Cell Image'

; Create another window and display the smoothed image.
WINDOW, 3, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Smoothed Image'
TV, smoothedImage

END
```

## Sharpening an Image

Sharpening an image increases the contrast between bright and dark regions to bring out features.

The sharpening process is basically the application of a high pass filter to an image. The following array is a kernel for a common high pass filter used to sharpen an image:

$$\begin{bmatrix} -1/9 & -1/9 & -1/9 \\ -1/9 & 1 & -1/9 \\ -1/9 & -1/9 & -1/9 \end{bmatrix}$$

**Note**

The above array is an example of one possible kernel for a sharpening filter. Other filters may include more weighting for the center point.

As mentioned in the filtering section of this chapter, filters can be applied to images in IDL with the CONVOL function. See "High Pass Filtering" on page 433 for more information on high pass filters.

The following example shows how to use IDL's CONVOL function and the above high pass filter kernel to sharpen an image. This example uses the Magnetic Resonance Image (MRI) of a human knee contained within the `mr_knee.dcm` file in the `examples/data` directory. Within the original knee MRI, some information is nearly as dark as the background. This image is sharpened to display these dark areas with improved contrast.

For code that you can copy and paste into an Editor window, see "Example Code: Sharpening an Image" on page 462 or complete the following steps for a detailed description of the process.

1. Import the image from the `mr_knee.dcm` file:

```
file = FILEPATH('mr_knee.dcm', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_DICOM(file)
imageSize = SIZE(image, /DIMENSIONS)
```

2. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

3. Create a window and display the original image:

```
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original Knee MRI'
TVSCL, image
```

The following figure shows the original image.

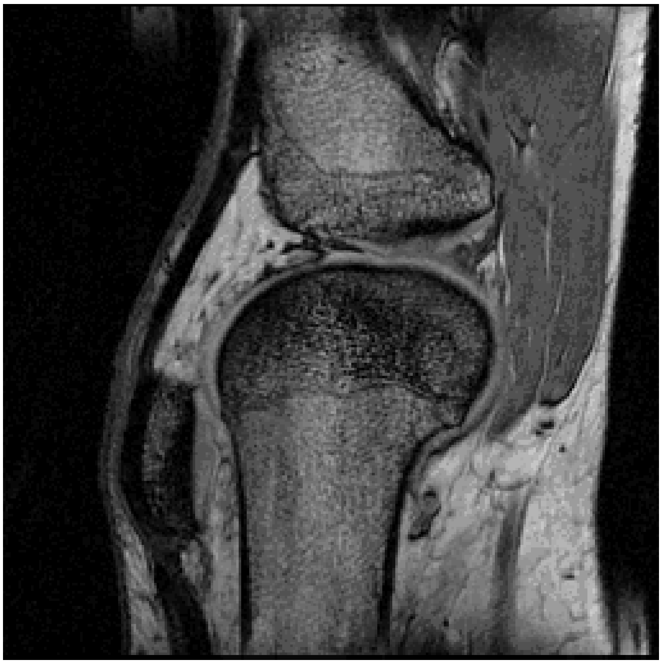


*Figure 10-32: Original Knee MRI*

4. Create a kernel for a sharpening (high pass) filter:

```
kernelSize = [3, 3]
kernel = REPLICATE(-1./9., kernelSize[0], kernelSize[1])
kernel[1, 1] = 1.
```

5. Apply the filter to the image:

```
filteredImage = CONVOL(FLOAT(image), kernel, $
   /CENTER, /EDGE_TRUNCATE)
```

6. Create another window and display the resulting filtered image:

```
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Sharpen Filtered Knee MRI'
TVSCL, filteredImage
```

   The following figure shows the results of applying the sharpening (high pass) filter. Pixels that differ dramatically in contrast with surrounding pixels are brightened.

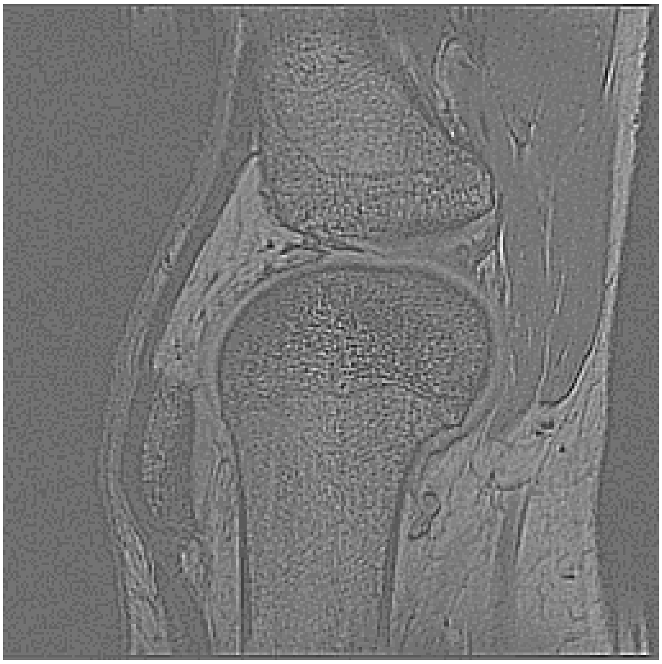


*Figure 10-33: Sharpen FIltered Knee MRI*

7. Create another window and display the combined images:

```
WINDOW, 2, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Sharpened Knee MRI'
TVSCL, image + filteredImage
```

The following figure shows the combination of the sharpened and original images. This image is sharper, containing more information within several regions, especially the tips of the bones.

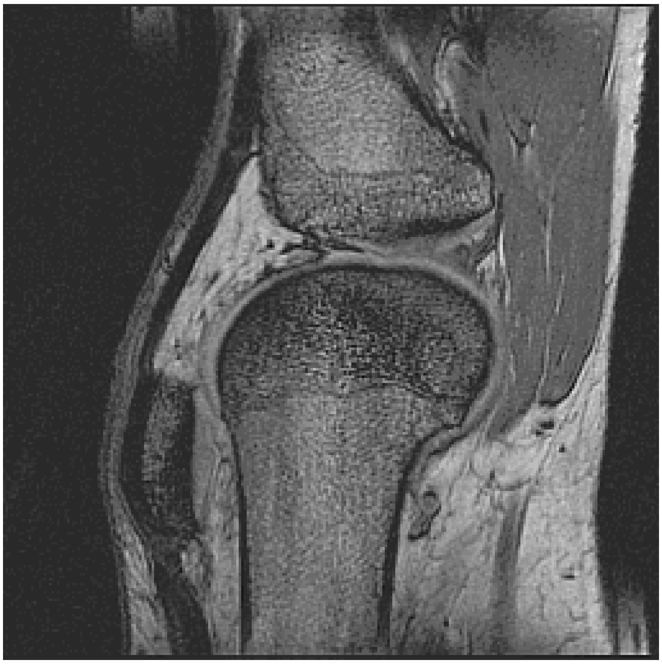


*Figure 10-34: Sharpened Knee MRI*

## Example Code: Sharpening an Image

Copy and paste the following text into an IDL Editor window. After saving the file as `Sharpening.pro`, compile and run the program to reproduce the previous example.

```
PRO Sharpening

; Import the image from the file.
file = FILEPATH('mr_knee.dcm', $
   SUBDIRECTORY = ['examples', 'data'])
image = READ_DICOM(file)
imageSize = SIZE(image, /DIMENSIONS)

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

```
; Create a window and display the original image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Original Knee MRI'
TVSCL, image

; Create a sharpening (high pass) filter.
kernelSize = [3, 3]
kernel = REPLICATE(-1./9., kernelSize[0], kernelSize[1])
kernel[1, 1] = 1.

; Apply the filter to the image.
filteredImage = CONVOL(FLOAT(image), kernel, $
   /CENTER, /EDGE_TRUNCATE)

; Create another window and display the resulting
; filtered image.
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Sharpen Filtered Knee MRI'
TVSCL, filteredImage

; Create another window and display the combined images.
WINDOW, 2, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Sharpened Knee MRI'
TVSCL, image + filteredImage

END
```

# Detecting Edges

Detecting edges is another way to help extract features. Many edge detection methods use either directional or Laplacian filters. See "Directional Filtering" on page 438 and "Laplacian Filtering" on page 442 for more information on directional and Laplacian filters.

IDL contains two basic edge detection routines, the ROBERTS and SOBEL functions. See the ROBERTS and SOBEL descriptions *in the IDL Reference Guide* for more information on these operators. Morphological operators are used for more complex edge detection. See "Detecting Edges of Image Objects" in Chapter 11 for more information on these operators.

The following examples show how to use these routines to detect edges of shapes within an image:

- "Enhancing Edges with the Roberts Operator"
- "Enhancing Edges with the Sobel Operator" on page 467

The results of these edge detection routines can be added or subtracted from the original image to enhance the contrast of the edges within that image. Edge detection results are also used to calculate masks. See "Masking Images" in Chapter 6 for more information on masks.

## Enhancing Edges with the Roberts Operator

The following example shows how to use the ROBERTS function to detect edges within an image. This example uses the aerial view of New York City within the `nyny.dat` file in the `examples/data` directory.

For code that you can copy and paste into an Editor window, see "Example Code: Enhancing edges with the Roberts Operator" on page 466 or complete the following steps for a detailed description of the process.

1. Import the image from the `nyny.dat` file:

```
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [768, 512]
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Crop the image to focus in on the bridges:

```
croppedSize = [96, 96]
croppedImage = image[200:(croppedSize[0] - 1) + 200, $
   180:(croppedSize[1] - 1) + 180]
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
displaySize = [256, 256]
```

4. Create a window and display the cropped image:

```
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Cropped New York Image'
TVSCL, CONGRID(croppedImage, displaySize[0], $
   displaySize[1])
```

The following figure shows the cropped section of the original image.

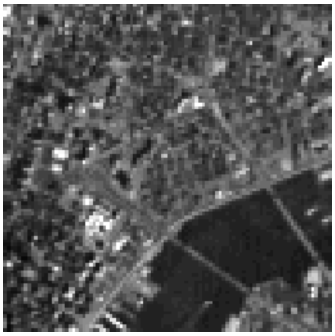


*Figure 10-35: Cropped New York Image*

5. Apply the Roberts filter to the image:

```
filteredImage = ROBERTS(croppedImage)
```

6. Create another window and display the resulting filtered image:

```
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Filtered New York Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])
```

The following figure shows the results of applying the Roberts filter. Edges have been highlighted around all elements separated by significant differences in pixel values.

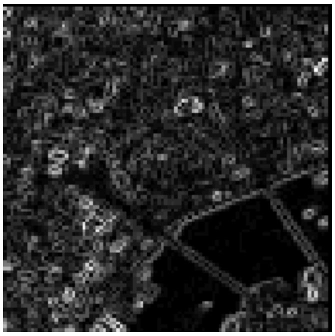


*Figure 10-36: Roberts Filter Applied to the New York Image*

## Example Code: Enhancing edges with the Roberts Operator

Copy and paste the following text into an IDL Editor window. After saving the file as `DetectingEdgesWithROBERTS.pro`, compile and run the program to reproduce the previous example.

```
PRO DetectingEdgesWithROBERTS

; Import the image from the file.
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [768, 512]
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Crop the image to focus in on the bridges.
croppedSize = [96, 96]
croppedImage = image[200:(croppedSize[0] - 1) + 200, $
   180:(croppedSize[1] - 1) + 180]

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0
displaySize = [256, 256]

; Create a window and display the cropped image.
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Cropped New York Image'
```

```
TVSCL, CONGRID(croppedImage, displaySize[0], $
   displaySize[1])

; Apply the filter to the image with the ROBERTS function.
filteredImage = ROBERTS(croppedImage)

; Create another window and display the resulting
; filtered image.
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Filtered New York Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])

END
```

## Enhancing Edges with the Sobel Operator

The following example shows how to use the SOBEL function to detect edges within an image. This example uses the aerial view of New York City within the `nyny.dat` file in the `examples/data` directory.

For code that you can copy and paste into an Editor window, see “Example Code: Enhancing edges with the Sobel Operator” on page 469 or complete the following steps for a detailed description of the process.

1. Import the image from the `nyny.dat` file:

```
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [768, 512]
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Crop the image to focus in on the bridges:

```
croppedSize = [96, 96]
croppedImage = image[200:(croppedSize[0] - 1) + 200, $
   180:(croppedSize[1] - 1) + 180]
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
displaySize = [256, 256]
```

4. Create a window and display the cropped image:

```
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Cropped New York Image'
TVSCL, CONGRID(croppedImage, displaySize[0], $
   displaySize[1])
```

The following figure shows the cropped section of the original image.

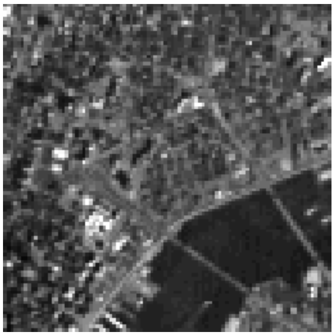


*Figure 10-37: Cropped New York Image*

5. Apply the Sobel filter to the image:

```
filteredImage = SOBEL(croppedImage)
```

6. Create another window and display the resulting filtered image:

```
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Filtered New York Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])
```

The following figure shows the edge enhancement results of applying the Sobel operator.

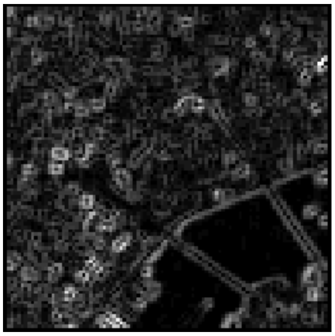


*Figure 10-38: Sobel Filter Applied to the New York Image*

## Example Code: Enhancing edges with the Sobel Operator

Copy and paste the following text into an IDL Editor window. After saving the file as `DetectingEdgesWithSOBEL.pro`, compile and run the program to reproduce the previous example.

```
PRO DetectingEdgesWithSOBEL

; Import the image from the file.
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [768, 512]
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Crop the image to focus in on the bridges.
croppedSize = [96, 96]
croppedImage = image[200:(croppedSize[0] - 1) + 200, $
   180:(croppedSize[1] - 1) + 180]

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0
displaySize = [256, 256]

; Create a window and display the cropped image.
WINDOW, 0, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Cropped New York Image'
TVSCL, CONGRID(croppedImage, displaySize[0], $
   displaySize[1])

; Apply the filter to the image with the SOBEL function.
filteredImage = SOBEL(croppedImage)

; Create another window and display the resulting
; filtered image.
WINDOW, 1, XSIZE = displaySize[0], YSIZE = displaySize[1], $
   TITLE = 'Filtered New York Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])

END
```

## Removing Noise

When a device (such as a camera or scanner) captures an image, the device sometimes adds extraneous noise to the image. This noise must be removed from the image for other image processing operations to return valuable results. Some noise can simply be removed by smoothing an image or masking it within the frequency domain, but most noise requires more involved filtering, such as windowing or adaptive filters. The following example shows how to use windowing and adaptive filters to remove noise from an image within IDL:

- “Windowing to Remove Noise”
- “Lee Filtering to Remove Noise” on page 475

### Windowing to Remove Noise

Within the frequency domain, a filter is applied to an image by multiplying the FFT of that image by the FFT of the filter. When the FFT of a image is multiplied by the FFT of a filter to perform convolution, this process is known as windowing.

The DIST and HANNING functions are examples of windowing filters already transformed into the frequency domain. Windowing with the DIST function has the same effect as applying a high pass filter. The high frequency information is retained, while the effect of the low frequency information is decreased. In contrast, the HANNING function retains the low frequency information. The results of the HANNING function are similar to a mask used to remove noise in an image. The HANNING function can be used to create either a Hanning or Hamming window. Although the DIST and the HANNING functions perform different filtering tasks, these filters are applied the same way, so only one example is provided in this section.

Windowing is different than simply using a mask within the frequency domain. Using a mask omits information within the image, while windowing retains the information, but decreases its effect on the image. See Chapter 9, “Removing Noise with the FFT” for more information on using a mask to remove noise from an image.

The following example shows how to use the HANNING function when windowing an image to remove background noise. This example uses the first image within the `abnorm.dat` file in the `examples/data` directory.

For code that you can copy and paste into an Editor window, see “Example Code: Windowing to Remove Noise” on page 474 or complete the following steps for a detailed description of the process.

1. Import the image from the `abnorm.dat` file:

```
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [64, 64]
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Initialize a display size parameter to resize the image when displaying it:

```
displaySize = 2*imageSize
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

4. Create a window and display the original image:

```
WINDOW, 0, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], $
   TITLE = 'Original Image'
TVSCL, CONGRID(image, displaySize[0], displaySize[1])
```

The following figure shows the original image.

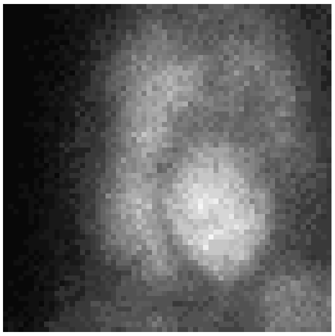


*Figure 10-39: Original Gated Blood Pool Image*

5. Determine the forward Fourier transformation of the image:

```
transform = SHIFT(FFT(image), (imageSize[0]/2), $
   (imageSize[1]/2))
```

6. Create another window and display the power spectrum:

```
WINDOW, 1, TITLE = 'Surface of Forward FFT'
SHADE_SURF, (2.*ALOG10(ABS(transform))), /XSTYLE, /YSTYLE, $
   /ZSTYLE, TITLE = 'Power Spectrum', $
   XTITLE = 'Mode', YTITLE = 'Mode', $
   ZTITLE = 'Amplitude', CHARSIZE = 1.5
```

The following figure shows the power spectrum of the original image. Noise within the image is shown as small peaks.

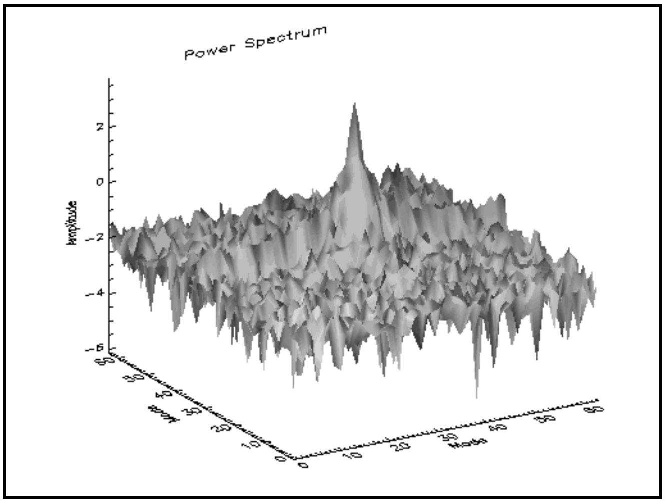


*Figure 10-40: Power Spectrum of the Gated Blood Pool Image*

7. Use a Hanning mask to filter out the noise:

```
mask = HANNING(imageSize[0], imageSize[1])
maskedTransform = transform*mask
```

8. Create another window and display the masked power spectrum:

```
WINDOW, 2, TITLE = 'Surface of Filtered FFT'
SHADE_SURF, (2.*ALOG10(ABS(maskedTransform))), $
   /XSTYLE, /YSTYLE, /ZSTYLE, TITLE = 'Masked Power
Spectrum', $
   XTITLE = 'Mode', YTITLE = 'Mode', $
   ZTITLE = 'Amplitude', CHARSIZE = 1.5
```

The following figure shows the results of applying the Hanning window. The Hanning window gradually smooths the high frequency peaks within the image.

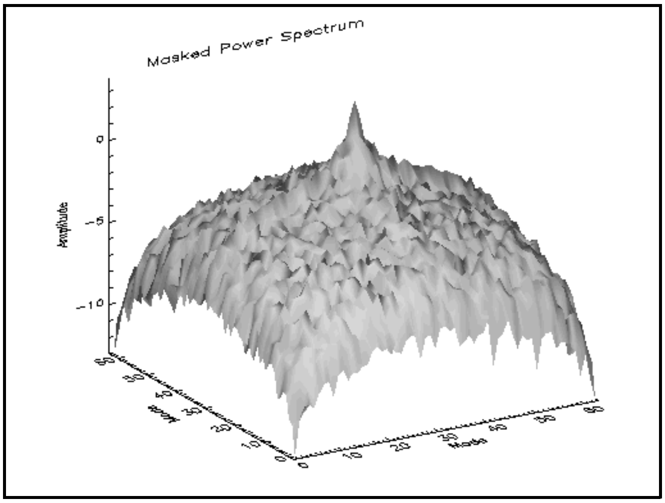


*Figure 10-41: Masked Power Spectrum of the Gated Blood Pool Image*

9. Apply the inverse transformation to the masked frequency domain image:

```
inverseTransform = FFT(SHIFT(maskedTransform, $
   (imageSize[0]/2), (imageSize[1]/2)), /INVERSE)
```

10. Create another window and display the results of the inverse transformation:

```
WINDOW, 3, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], $
   TITLE = 'Hanning Filtered Image'
TVSCL, CONGRID(REAL_PART(inverseTransform), $
   displaySize[0], displaySize[1])
```

The following figure shows the resulting display. Visible noise within the image has been reduced, while the valuable image data has been retained.

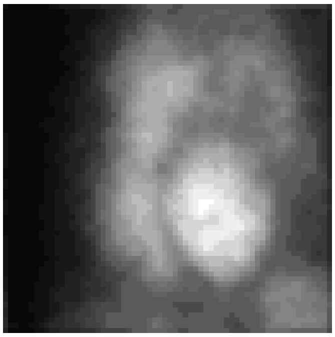


*Figure 10-42: Resulting Hanning Filtered Image*

## Example Code: Windowing to Remove Noise

Copy and paste the following text into an IDL Editor window. After saving the file as `RemovingNoiseWithHANNING.pro`, compile and run the program to reproduce the previous example.

```
PRO RemovingNoiseWithHANNING

; Import the image from the file.
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [64, 64]
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize a display size parameter to resize the
; image when displaying it.
displaySize = 2*imageSize

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the original image.
WINDOW, 0, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], $
   TITLE = 'Original Image'
TVSCL, CONGRID(image, displaySize[0], displaySize[1])

; Determine the forward Fourier transformation of the
; image.
transform = SHIFT(FFT(image), (imageSize[0]/2), $
   (imageSize[1]/2))

; Create another window and display the power spectrum.
WINDOW, 1, TITLE = 'Surface of Forward FFT'
SHADE_SURF, (2.*ALOG10(ABS(transform))), $
   /XSTYLE, /YSTYLE, /ZSTYLE, TITLE = 'Power Spectrum', $
   XTITLE = 'Mode', YTITLE = 'Mode', $
   ZTITLE = 'Amplitude', CHARSIZE = 1.5

; Use a Hanning mask to filter out the noise.
mask = HANNING(imageSize[0], imageSize[1])
maskedTransform = transform*mask

; Create another window and display the masked power
; spectrum.
WINDOW, 2, TITLE = 'Surface of Filtered FFT'
SHADE_SURF, (2.*ALOG10(ABS(maskedTransform))), $
   /XSTYLE, /YSTYLE, /ZSTYLE, $
   TITLE = 'Masked Power Spectrum', $
   XTITLE = 'Mode', YTITLE = 'Mode', $
   ZTITLE = 'Amplitude', CHARSIZE = 1.5

; Apply the inverse transformation to masked frequency
; domain image.
inverseTransform = FFT(SHIFT(maskedTransform, $
   (imageSize[0]/2), (imageSize[1]/2)), /INVERSE)

; Create another window and display the results of
; inverse transformation.
WINDOW, 3, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], $
   TITLE = 'Hanning Filtered Image'
TVSCL, CONGRID(REAL_PART(inverseTransform), $
   displaySize[0], displaySize[1])

END
```

## Lee Filtering to Remove Noise

Unlike the Hanning window, the Lee filter is convolved within the spatial domain. The Lee filter is an adaptive filter, which changes according to the local statistics of the current pixel. The LEEFILT routine applies the Lee filter to an image to remove background noise.

The following example shows how to use the LEEFILT function to remove background noise from an image. This example uses the first image within the `abnorm.dat` file in the `examples/data` directory.

For code that you can copy and paste into an Editor window, see "Example Code: Lee Filtering to Remove Noise" on page 477 or complete the following steps for a detailed description of the process.

1. Import the image from the `abnorm.dat` file:

```
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [64, 64]
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

2. Initialize a display size parameter to resize the image when displaying it:

```
displaySize = 2*imageSize
```

3. Initialize the display:

```
DEVICE, DECOMPOSED = 0
LOADCT, 0
```

4. Create a window and display the original image:

```
WINDOW, 0, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], $
   TITLE = 'Original Image'
TVSCL, CONGRID(image, displaySize[0], displaySize[1])
```

The following figure shows the original image.

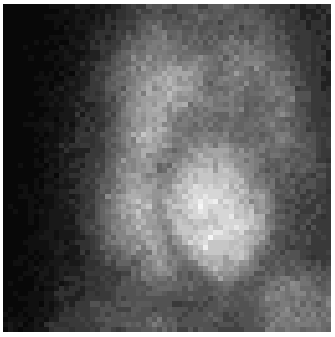


*Figure 10-43: Original Gated Blood Pool Image*

5. Apply the Lee filter to the image:

```
filteredImage = LEEFILT(image, 1)
```

6. Create another window and display the Lee filtered image:

```
WINDOW, 1, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], $
   TITLE = 'Lee Filtered Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])
```

The following figure shows the results of applying the Lee filter, which adaptively smooths areas that contains noise.

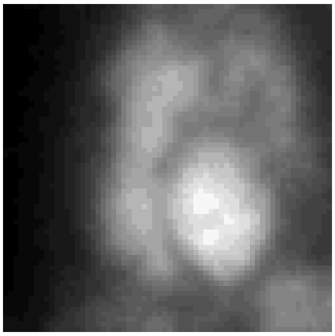


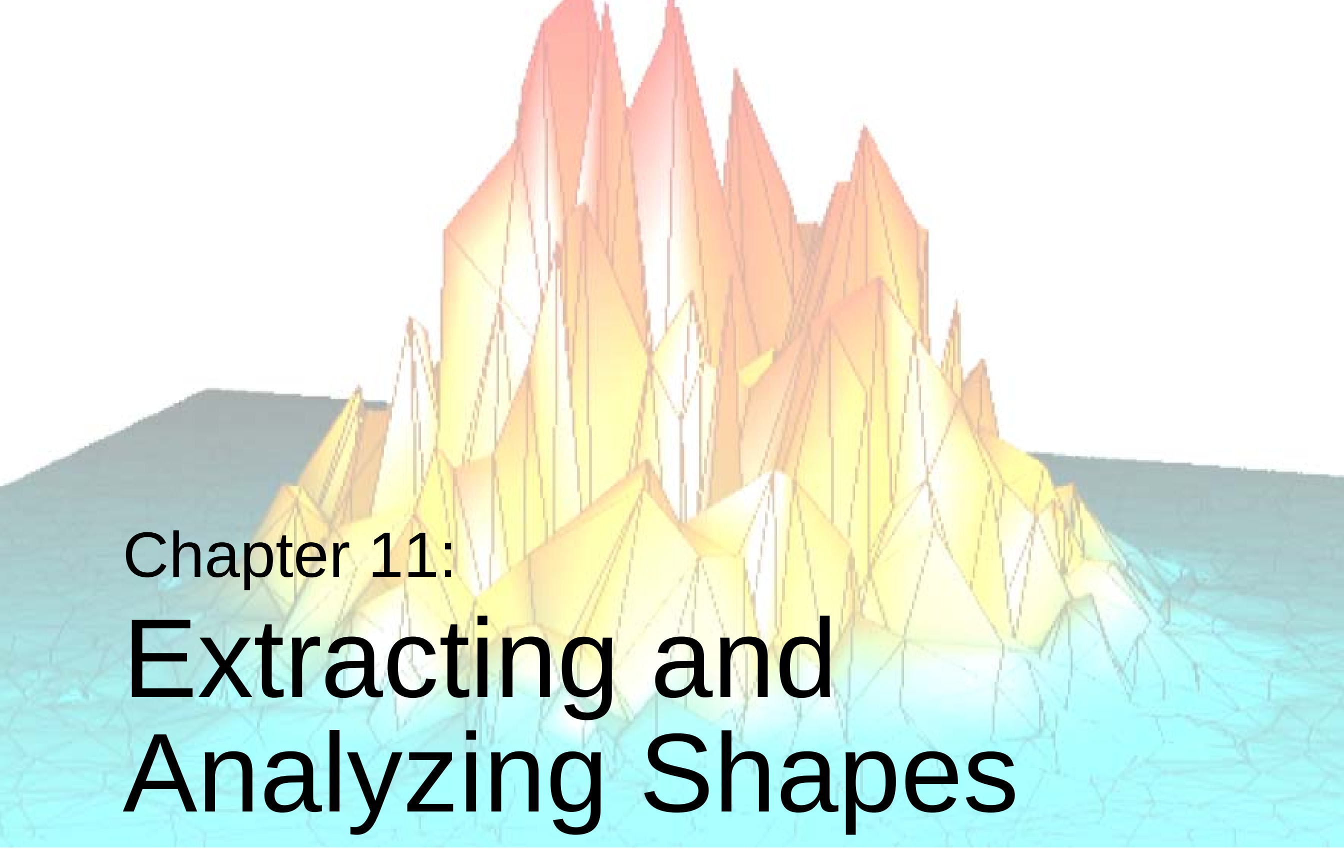
*Figure 10-44: Lee Filtered Gated Blood Pool Image*

## Example Code: Lee Filtering to Remove Noise

Copy and paste the following text into an IDL Editor window. After saving the file as `RemovingNoiseWithLEEFILT.pro`, compile and run the program to reproduce the previous example.

```
PRO RemovingNoiseWithLEEFILT

; Import the image from the file.
file = FILEPATH('abnorm.dat', $
   SUBDIRECTORY = ['examples', 'data'])
imageSize = [64, 64]
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize a display size parameter to resize the
; image when displaying it.
displaySize = 2*imageSize
```

```
; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the original image.
WINDOW, 0, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], $
   TITLE = 'Original Image'
TVSCL, CONGRID(image, displaySize[0], displaySize[1])

; Apply the Lee filter to the image.
filteredImage = LEEFILT(image, 1)

; Create another window and display the Lee filtered
; image
WINDOW, 1, XSIZE = displaySize[0], $
   YSIZE = displaySize[1], $
   TITLE = 'Lee Filtered Image'
TVSCL, CONGRID(filteredImage, displaySize[0], $
   displaySize[1])

END
```



# Chapter 11: Extracting and Analyzing Shapes

This chapter describes using morphological operations in conjunction with image analysis routines to extract and analyze image elements. This chapter includes the following topics:

# Overview of Extracting and Analyzing Image Shapes

Morphological image processing operations reveal the underlying structures and shapes within binary and grayscale images, clarifying basic image features. While individual morphological operations perform simple functions, they can be combined to extract specific information from an image. Morphological operations often precede more advanced pattern recognition and image analysis operations such as segmentation. Shape recognition routines commonly include image thresholding or stretching to separate foreground and background image features. See "Determining Intensity Values When Thresholding and Stretching Images" on page 486 for tips on how to produce the desired results.

This chapter also provides examples of more advanced image analysis routines that return information about specific image elements. One example identifies unique regions within an image and the other finds the area of a specific image feature. See "Analyzing Image Shapes" on page 540 for more information.

**Note**

In this book, Direct Graphics examples are provided by default. Object Graphics examples are provided in cases where significantly different methods are required.

## Applying a Morphological Structuring Element to an Image

Morphological operations apply a *structuring element* or *morphological mask* to an image. A structuring element that is applied to an image must be 2 dimensional, having the same number of dimensions as the array to which it is applied. A morphological operation passes the structuring element, of an empirically determined size and shape, over an image. The operation compares the structuring element to the underlying image and generates an output pixel based upon the function of the morphological operation. The size and shape of the structuring element determines what is extracted or deleted from an image. In general, smaller structuring elements preserve finer details within an image than larger elements. For more information on selecting and creating a structuring element, see "Guidelines for Determining Structuring Element Shapes and Sizes" on page 484.

Morphological operations can be applied to either binary or grayscale images. When applied to a binary image, the operation returns pixels that are either black, having a logical value of 0, or white, having a logical value of 1. Each image pixel and its neighboring pixels are compared against the structuring element to determine the pixel's value in the output image. With grayscale images, pixel values are determined by taking a neighborhood minimum or neighborhood maximum value (as required by the morphological process). The structuring element provides the definition of the shape of the neighborhood.

The following table introduces image processing tasks and associated IDL image processing routines covered in this chapter.

| Task | Routine(s) | Description |
|---|---|---|
| "Eroding and Dilating Image Objects" on page 489. | ERODE | Reduce the size of objects in relation to their background. |
| | DILATE | Expand the size of objects in relation to their background. |
| "Smoothing with MORPH_OPEN" on page 496. | MORPH_OPEN | Apply an erosion operation followed by a dilation operation to a binary or grayscale image. |
| "Smoothing with MORPH_CLOSE" on page 500. | MORPH_CLOSE | Apply a dilation operation followed by an erosion operation to a binary or grayscale image. |
| "Detecting Peaks of Brightness" on page 504. | MORPH_TOPHAT | Retain only the brightest pixels within a grayscale image. |
| "Creating Image Object Boundaries" on page 508. | WATERSHED | Detect boundaries between similar regions in a grayscale image. |

*Table 11-1: Shape Extraction and Analysis Tasks and Routines*

| Task | Routine(s) | Description |
|---|---|---|
| "Selecting Specific Image Objects" on page 514. | MORPH_HITORMISS | Use "hit" and "miss" structures to identify image elements that meet the specified conditions. |
| "Detecting Edges of Image Objects" on page 520. | MORPH_GRADIENT | Subtract an eroded version of a grayscale image from a dilated version of the image, highlighting edges. |
| "Creating Distance Maps" on page 523. | MORPH_DISTANCE | Estimate for each binary foreground pixel the distance to the nearest background pixel, using a given norm. |
| "Thinning Image Objects" on page 527. | MORPH_THIN | Subtract hit-or-miss results from a binary image. Repeated thinning results in pixel-wide linear representations of image objects. |
| "Analyzing Image Shapes" on page 540. | LABEL_REGION | Identify and assign index numbers to discrete regions within a binary image. |
| | CONTOUR | Create a contour plot and extract information about specific contours. |

*Table 11-1: Shape Extraction and Analysis Tasks and Routines (Continued)*

**Note**

For an example that uses a combination of morphological operations to remove bridges from the waterways of New York, see "Combining Morphological Operations" on page 534.

## Guidelines for Determining Structuring Element Shapes and Sizes

Determining the size and shape of a structuring element is largely an empirical process. However, the overall selection of a structuring element depends upon the geometric shapes you are attempting to extract from the image data. For example, if you are dealing with biological or medical images, which contain few straight lines or sharp angles, a circular structuring element is an appropriate choice. When extracting shapes from geographic aerial images of a city, a square or rectangular element will allow you to extract angular features from the image.

While most examples in this chapter use simple structuring elements, you may need to create several different elements or different rotations of a singular element in order to extract the desired shapes from your image. For example, if you wish to extract the rectangular roads from an aerial image, the initial rectangular element will need to be rotated a number of ways to account for multiple orientations of the roads within the image.

The size of the structuring element depends upon what features you wish to extract from the image. Larger structuring elements preserve larger features while smaller elements preserve the finer details of image features.

The following table shows how to easily create simple disk-shaped, square, rectangle, diagonal and custom structuring elements using IDL. The visual representations of the structures, shown in the right-hand column, indicate that the shape of each binary structuring element is defined by foreground pixels having a value of one.

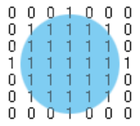
| IDL Code For Structuring Element Shapes | Examples |
|---|---|
| **Disk-Shaped Structuring Element**<br>Use SHIFT in conjunction with DIST to create the disk shape.<br>`radius = 3`<br>`strucElem = SHIFT(DIST(2*radius+1), radius, $`<br>`     radius) LE radius`<br>Change *radius* to alter the size of the structuring element. |  0 0 0 1 0 0 0<br>0 1 1 1 1 1 0<br>0 1 1 1 1 1 0<br>1 1 1 1 1 1 1<br>0 1 1 1 1 1 0<br>0 1 1 1 1 1 0<br>0 0 0 1 0 0 0 |

*Table 11-2: Creating Various Structuring Elements Shapes with IDL*

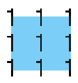
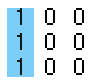
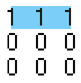
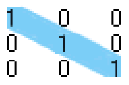
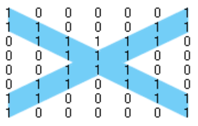
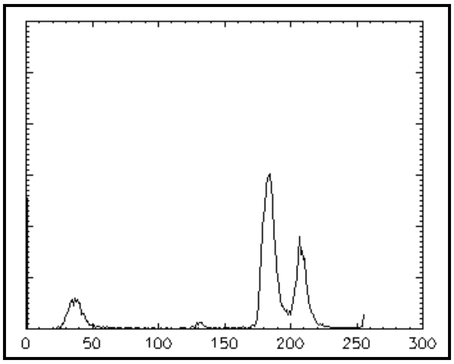
| IDL Code For Structuring Element Shapes | Examples |
|---|---|
| **Square Structuring Element** Use DIST to define the square array. `side = 3` `strucElem = DIST(side) LE side` Change *side* to alter the size of the structuring element. |  |
| **Vertical Rectangular Structuring Element** Use BYTARR to define the initial array. `strucElem = BYTARR(3,3)` `strucElem [0,*] = 1` Create a 2 x 3 structure by adding `strucElem[1,*] = 1`. |  |
| **Horizontal Rectangular Structuring Element** Use BYTARR to define the initial array. `strucElem = BYTARR(3,3)` `strucElem [*,0] = 1` Create a 3 x 2 structure by adding, `strucElem[*,1] = 1`. |  |
| **Diagonal Structuring Element** Use IDENTITY to create the initial array. `strucElem = BYTE(IDENTITY(3))` **Note -** BYTE is used to create a byte array, consistent with the other structuring elements. |  |
| **Irregular Structuring Elements** Define custom arrays to create irregular structuring elements or a series of rotations of a single structuring element. `strucElem = [[1,0,0,0,0,0,1], $` `[1,1,0,0,0,1,1], $` `[0,1,1,1,1,1,0], $` `[0,0,1,1,1,0,0], $` `[0,0,1,1,1,0,0], $` `[0,1,1,0,1,1,0], $` `[1,1,0,0,0,1,1], $` `[1,0,0,0,0,0,1]]` **Note -** Creating a series of rotations of a single structuring element is covered in "Thinning Image Objects" on page 527. |  |

*Table 11-2: Creating Various Structuring Elements Shapes with IDL*

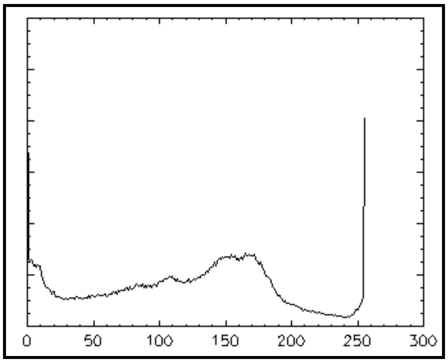
## Determining Intensity Values When Thresholding and Stretching Images

Thresholding and stretching images separate foreground pixels from background pixels and can be performed before or after applying a morphological operation to an image. While a threshold operation produces a binary image and a stretch operation produces a scaled, grayscale image, both operations rely upon the definition of an *intensity value*. This intensity value is compared to each pixel value within the image and an output pixel is generated based upon the conditions stated within the threshold or stretch statement.

Intensity histograms provide a means of determining useful intensity values as well as determining whether or not an image is a good candidate for thresholding or stretching. A histogram containing definitive peaks of intensities indicates that an image's foreground and background features can be successfully separated. A histogram containing connected, graduated ranges of intensities indicates the image is likely a poor candidate for thresholding or stretching.



Good Candidate



Poor Candidate

*Figure 11-1: Determining Appropriateness of Images for Thresholding or Stretching Using Intensity Histograms*

**Note**

To quickly view the intensity histogram of an image, create a window and use PLOT in conjunction with HISTOGRAM, entering `PLOT, HISTOGRAM(image)` where *image* denotes the image for which you wish to view a histogram.

## Thresholding an Image

Thresholding outputs a binary image as determined by a threshold intensity and one of the relational operators: EQ, NE, GE, GT, LE, or LT. In a statement containing a relational operator, thresholding compares each pixel in the original image to a threshold intensity. The output pixels (comprising the binary image) are assigned a value of 1 (white) when the relational statement is true and 0 (black) when the statement is false.

The following figure shows an intensity histogram of an image containing mineral crystals. The histogram indicates that the image can be successfully thresholded since there are definitive peaks of intensities. Also shown in the following figure, a statement such as `img LE 50` produces an image where all pixels less than the threshold intensity value of 50 are assigned a foreground pixel value of 1 (white). The statement, `img GE 50` produces a contrasting image where all original pixels values greater than 50 are assigned a foreground pixel value (white).

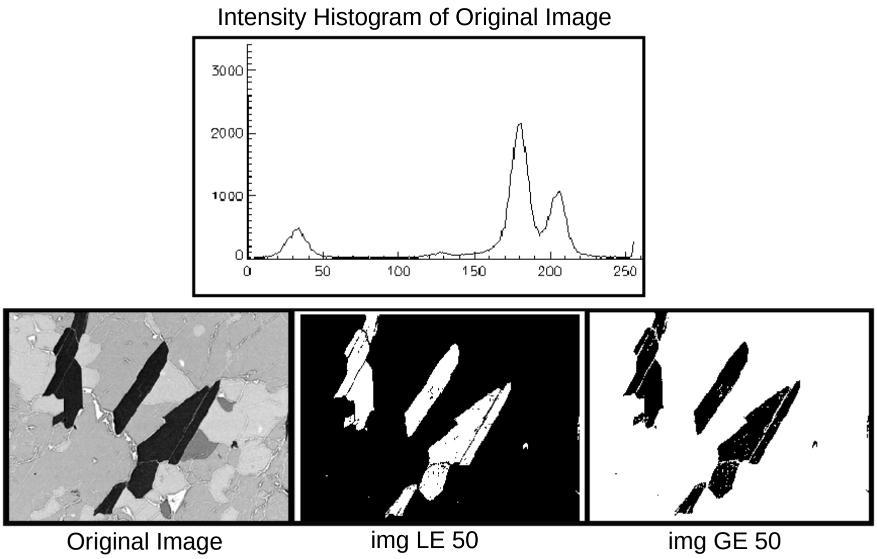


*Figure 11-2: Image Thresholding*

## Stretching an Image

Stretching an image (also know as scaling) creates a grayscale image, scaling a range of selected pixel values across all possible intensities. When using TVSCL or BYTSCL in conjunction with the > and < operators, a range of pixels defined by the intensity value and operator are scaled across the entire intensity range, (0 to 255).

The following figure shows the results of displaying each image stretching statement using `TVSCL, image`:

- `image = img < 50` — All pixel values greater than 50 are assigned a value of 50, now the maximum pixel value (white). Applying TVSCL or BYTSCL stretches the remaining pixel values across all possible intensities (0 to 255).
- `image = img < 190` — All pixel values greater than 190 are assigned a value of 190, now the maximum pixel value (white). Applying TVSCL or BYTSCL stretches the remaining pixel values across all possible intensities (0 to 255).
- `image = img > 150 < 190` — Using two intensity values, extract a single peak of values shown in the histogram, all values less than 150 are assigned a minimum pixel value (black) and all values greater than 190 are assigned a maximum pixel value (white). Applying TVSCL or BYTSCL stretches the remaining pixel values across all possible intensities (0 to 255).

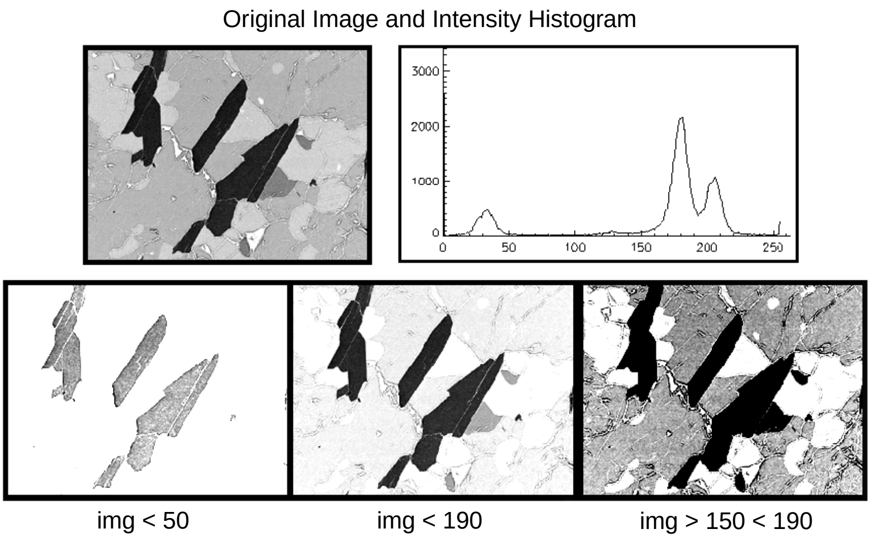


*Figure 11-3: Image Stretching*

# Eroding and Dilating Image Objects

The basic morphological operations, erosion and dilation, produce contrasting results when applied to either grayscale or binary images. Erosion shrinks image objects while dilation expands them. The specific actions of each operation are covered in the following sections.

## Characteristics of Erosion

- Erosion generally decreases the sizes of objects and removes small anomalies by subtracting objects with a radius smaller than the structuring element.
- With grayscale images, erosion reduces the brightness (and therefore the size) of bright objects on a dark background by taking the neighborhood minimum when passing the structuring element over the image.
- With binary images, erosion completely removes objects smaller than the structuring element and removes perimeter pixels from larger image objects.

## Characteristics of Dilation

- Dilation generally increases the sizes of objects, filling in holes and broken areas, and connecting areas that are separated by spaces smaller than the size of the structuring element.
- With grayscale images, dilation increases the brightness of objects by taking the neighborhood maximum when passing the structuring element over the image.
- With binary images, dilation connects areas that are separated by spaces smaller than the structuring element and adds pixels to the perimeter of each image object.

## Applying Erosion and Dilation

The following example applies erosion and dilation to grayscale and binary images. When using erosion or dilation, avoid the generation of indeterminate values for objects occurring along the edges of the image by padding the image, as shown in the following example. For code that you can copy and paste into an Editor window, see "Example Code: Eroding and Dilating Image Elements" on page 494 or complete the following steps for a detailed description of the process.

**Note**

This example uses a file from the `examples/demo/demodata` directory of your installation. If you have not already done so, you will need to install "IDL Demos" from your product CD-ROM to install the demo data file needed for this example.

1. Prepare the display device:

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

2. Load a grayscale color table:

```
LOADCT, 0
```

3. Select and read in the image file. Use the GRAYSCALE keyword to READ_JPEG to open the grayscale image:

```
file = FILEPATH('pollens.jpg', $
   SUBDIRECTORY = ['examples', 'demo', 'demodata'])
READ_JPEG, file, img, /GRAYSCALE
```

4. Get the size of the image:

```
dims = SIZE(img, /DIMENSION)
```

5. Define the structuring element. A radius of 2 results in a structuring element near the size of the specks of background noise. This radius also affects only the edges of the larger objects (whereas a larger radius would cause significant distortion of all image features):

```
radius = 2
```

6. Create a disk-shaped structuring element that corresponds to the shapes occurring within the image:

```
strucElem = SHIFT(DIST(2*radius+1), radius, radius) LE radius
```

**Tip**

Enter `PRINT, strucElem` to view the structure created by the previous statement.

7. Add a border to the image to avoid generating indeterminate values when passing the structuring element over objects along the edges of an image. If the starting origin of the structuring element is not specified in the call to ERODE, the origin defaults to one half the width of the structuring element. Therefore, creating a border equal to one half of the structuring element width (equal to the radius) is sufficient to avoid indeterminate values. Create padded images for both the erode operation (using the maximum array value for the border), and the dilate operation (using the minimum array value for the border) as follows:

   ```
   erodeImg = REPLICATE(MAX(img), dims[0]+2, dims[1]+2)
   erodeImg [1,1] = img

   dilateImg = REPLICATE(MIN(img), dims[0]+2, dims[1]+2)
   dilateImg [1,1] = img
   ```

**Note**

Padding is only necessary when accurate edge values are important. Adding a pad equal to more that one half the width of the structuring element does not negatively effect the morphological operation, but does minutely add to the processing time. The padding can be removed from the image after applying the morphological operation and before displaying the image if desired.

8. Get the size of either of the padded images, create a window and display the original image:

   ```
   padDims = SIZE(erodeImg, /DIMENSIONS)
   WINDOW, 0, XSIZE = 3*padDims[0], YSIZE = padDims[1], $
      TITLE = "Original, Eroded and Dilated Grayscale Images"
   TVSCL, img, 0
   ```

9. Apply the ERODE function to the grayscale image using the GRAY keyword and display the image:

   ```
   erodeImg = ERODE(erodeImg, strucElem, /GRAY)
   TVSCL, erodeImg, 1
   ```

10. For comparison, apply DILATE to the same image and display it:

    ```
    dilateImg = DILATE(dilateImg, strucElem, /GRAY)
    TVSCL, dilateImg, 2
    ```

The following image displays the effects of erosion (middle) and dilation (right). Erosion removes pixels from perimeters of objects, decreases the overall brightness of the grayscale image and removes objects smaller than the structuring element.

Dilation adds pixels to perimeters of objects, brightens the image, and fills in holes smaller than the structuring element as shown in the following figure.

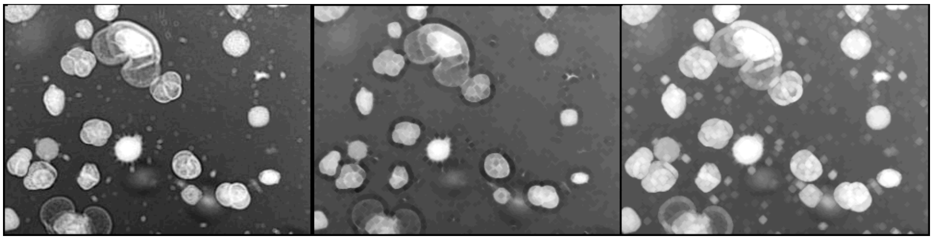


*Figure 11-4: Original (left), Eroded (center) and Dilated (right) Grayscale Images*

11. Create a window and use HISTOGRAM in conjunction with PLOT, displaying an intensity histogram to help determine the threshold intensity value:

```
WINDOW, 1, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(img)
```

**Note**

Using an intensity histogram as a guide for determining threshold values is described in the section, "Determining Intensity Values When Thresholding and Stretching Images" on page 486.

12. To compare the effects of erosion and dilation on binary images, create a binary image, retaining pixels with values greater than or equal to 120:

```
img = img GE 120
```

13. Create padded binary images for the erode and dilation operations, using 1 as the maximum array value for the erosion image and 0 as the minimum value for the dilation image:

```
erodeImg = REPLICATE(1B, dims[0]+2, dims[1]+2)
erodeImg [1,1] = img

dilateImg = REPLICATE(0B, dims[0]+2, dims[1]+2)
dilateImg [1,1] = img
```

14. Get the dimensions of either image, create a second window and display the binary image:

```
dims = SIZE(erodeImg, /DIMENSIONS)
WINDOW, 2, XSIZE = 3*dims[0], YSIZE = dims[1], $
   TITLE = "Original, Eroded and Dilated Binary Images"
TVSCL, img, 0
```

15. Using the structuring element defined previously, apply the erosion and dilation operations to the binary images and display the results by entering the following lines:

```
erodeImg = ERODE(erodeImg, strucElem)
TVSCL, erodeImg, 1
dilateImg = DILATE(dilateImg, strucElem)
TVSCL, dilateImg, 2
```

The results are shown in the following figure.

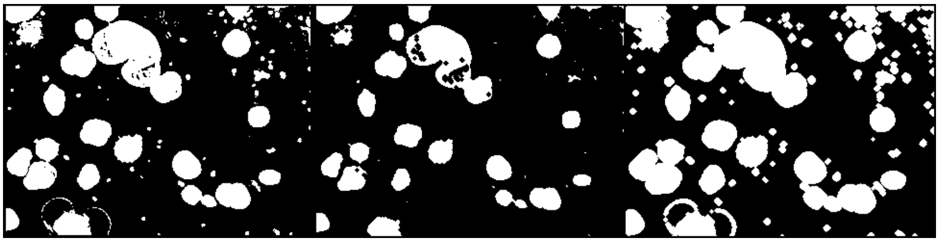


*Figure 11-5: Original, Eroded and Dilated Binary Images*

## Example Code: Eroding and Dilating Image Elements

Copy and paste the following text into the IDL Editor window. After saving the file as `MorphErodeDilate.pro`, compile and run the program to reproduce the previous example.

```
PRO MorphErodeDilate

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

; Load an image.
file = FILEPATH('pollens.jpg', $
   SUBDIRECTORY = ['examples', 'demo', 'demodata'])
READ_JPEG, file, img, /GRAYSCALE

; Get the image size.
dims = SIZE(img, /DIMENSIONS)

; Create the structuring element, a disk with a radius
; of 2.
radius = 2
strucElem = SHIFT(DIST(2*radius+1), $
   radius, radius) LE radius

; Print the structuring element in order to visualize
; the previous statement.
PRINT, strucElem

; To avoid indeterminate edge values, add padding equal
; to one half the size of the structuring element
; (equal to the radius). Pad image to be eroded with
; maximum array value, and image to be dilated with
; minimum array value.
erodeImg = REPLICATE(MAX(img), dims[0]+2, dims[1]+2)
erodeImg [1,1] = img
dilateImg = REPLICATE(MIN(img), dims[0]+2, dims[1]+2)
dilateImg [1,1] = img

; Get the size of either of the padded images,
;  create a window and display the original image.
padDims = SIZE(erodeImg, /DIMENSIONS)
WINDOW, 0, XSIZE = 3*padDims[0], YSIZE = padDims[1], $
   TITLE = "Original, Eroded and Dilated Grayscale Images"
TVSCL, img, 0

; Use the erosion operator on the image, applying the
; structuring element. Display the image.
erodeImg = ERODE(erodeImg, strucElem, /GRAY)
```

```
TVSCL, erodeImg, 1

; Apply the dilation operator to the image, and display
; it.
dilateImg = DILATE(dilateImg, strucElem, /GRAY)
TVSCL, dilateImg, 2

; Create a window and display a histogram to help
; determine the threshold intensity value.
WINDOW, 1, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(img)

; Create a binary image of the grayscale image.
img = img GE 120

; Create padded binary images for the erode
; and dilate operations.
erodeImg = REPLICATE(1B, dims[0]+2, dims[1]+2)
erodeImg [1,1] = img
dilateImg = REPLICATE(0B, dims[0]+2, dims[1]+2)
dilateImg [1,1] = img

; Get the dimensions, create a second window
; and display the binary image.
dims = SIZE(erodeImg, /DIMENSIONS)
WINDOW, 2, XSIZE = 3*dims[0], YSIZE = dims[1], $
   TITLE = "Original, Eroded and Dilated Binary Images"
TVSCL, img, 0

; Apply the erosion and dilation operators to the
; binary images and display the results.
erodeImg = ERODE(erodeImg, strucElem)
TVSCL, erodeImg, 1
dilateImg = DILATE(dilateImg, strucElem)
TVSCL, dilateImg, 2

END
```

## Smoothing with MORPH_OPEN

The MORPH_OPEN function applies the opening operation, which is erosion followed by dilation, to a binary or grayscale image. The opening operation removes noise from an image while maintaining the overall sizes of objects in the foreground. Opening is a useful process for smoothing contours, removing pixel noise, eliminating narrow extensions, and breaking thin links between features. After using an opening operation to darken small objects and remove noise, thresholding or other morphological processes can be applied to the image to further refine the display of the primary shapes within the image.

The following example applies the opening operation to an image of microscopic spherical organisms, *Rhinosporidium seeberi* protozoans. After applying the opening operation and thresholding the image, only the largest elements of the image are retained, the mature *R.seeberi* organisms.

For code that you can copy and paste into an Editor window, see "Example Code: Using MORPH_OPEN to Remove Noise" on page 498 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load grayscale color table:

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

2. Select and open the image file:

   ```
   file = FILEPATH('r_seeberi.jpg', $
       SUBDIRECTORY = ['examples', 'data'])
   READ_JPEG, file, image, /GRAYSCALE
   ```

3. Get the image dimensions, prepare a window and display the image:

   ```
   dims = SIZE(image, /DIMENSIONS)
   WINDOW, 0, XSIZE = 2*dims[0], YSIZE = 2*dims[1], $
       TITLE = 'Defining Shapes with Opening Operation'
   TVSCL, image, 0
   ```

4. Define the radius of the structuring element and create a disk-shaped element to extract circular features:

   ```
   radius = 7
   strucElem = SHIFT(DIST(2*radius+1), radius, radius) LE radius
   ```

   Compared to the previous example, a larger element is used in order to retain only the larger image elements, discarding all of the smaller background features. Further increases in the size of the structuring element would extract even larger image features.

**Tip**

Enter `PRINT, strucElem` to view the structure created by the previous statement.

5. Apply the MORPH_OPEN function to the image, specifying the GRAY keyword for the grayscale image:

```
morphImg = MORPH_OPEN(image, strucElem, /GRAY)
```

6. Display the image:

```
TVSCL, morphImg, 1
```

The following figure shows the original image (left) and the application of the opening operation to the original image (right). The opening operation has enhanced and maintained the sizes of the large bright objects within the image while blending the smaller background features.

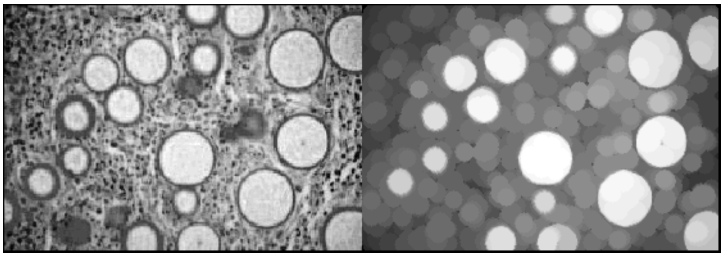


*Figure 11-6: Application of the Opening Operation to a Grayscale Image*

The following steps apply the opening operator to a binary image.

7. Create a window and use HISTOGRAM in conjunction with PLOT, displaying an intensity histogram to help determine the threshold intensity value:

```
WINDOW, 1, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(img)
```

**Note**

Using an intensity histogram as a guide for determining threshold values is described in the section, "Determining Intensity Values When Thresholding and Stretching Images" on page 486.

8. Using the histogram as a guide, create a binary image. To prepare to remove background noise, retain only areas of the image where pixel values are equal to or greater than 160:

```
threshImg = image GE 160
WSET, 0
TVSCL, threshImg, 2
```

9. Apply the opening operation to the binary image to remove noise and smooth contours, and then display the image:

```
morphThresh = MORPH_OPEN(threshImg, strucElem)
TVSCL, morphThresh, 3
```

The combination of thresholding and applying the opening operation has successfully extracted the primary foreground features as shown in the following figure.

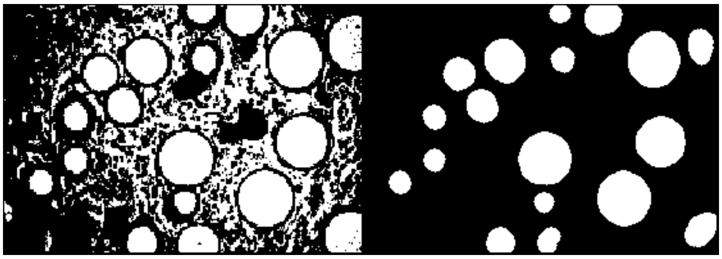


*Figure 11-7: Binary Image (left) and Application of the Opening Operator to the Binary Image (right)*

## Example Code: Using MORPH_OPEN to Remove Noise

Copy and paste the following text into the IDL Editor window. After saving the file as `MorphOpenExample.pro`, compile and run the program to reproduce the previous example.

```
PRO MorphOpenExample

; Prepare the display device and load grayscale color
; table.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 0

; Select and open the image file.
file = FILEPATH('r_seeberi.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, image, /GRAYSCALE
```

```
; Get the image dimensions, prepare a window and
; display the image.
dims = SIZE(image, /DIMENSIONS)
WINDOW, 0, XSIZE = 2*dims[0], YSIZE = 2*dims[1], $
   TITLE='Defining Shapes with the Opening Operator'
TVSCL, image, 0

; Define the radius of the structuring element and
; create the disk.
radius = 7
strucElem = SHIFT(DIST(2*radius+1), $
   radius, radius) LE radius

; Apply the opening operator to the image.
morphImg = MORPH_OPEN(image, strucElem, /GRAY)
TVSCL, morphImg, 1

; Create a window and display an intensity histogram
; to help determine the threshold intensity value.
WINDOW, 1, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(image)

; Threshold the image to prepare to remove background
; noise.
threshImg = image GE 160

; Display the thresholded image.
WSET, 0
TVSCL, threshImg, 2

; Apply the opening operator to the thresholded image.
morphThresh = MORPH_OPEN(threshImg, strucElem)

; Display the image.
TVSCL, morphThresh, 3

END
```

## Smoothing with MORPH_CLOSE

The morphological closing operation performs dilation followed by erosion, the opposite of the opening operation. The MORPH_CLOSE function smooths contours, links neighboring features, and fills small gaps or holes. The operation effectively brightens small objects in binary and grayscale images. Like the opening operation, primary objects retain their original shape.

The following example uses the closing operation and a square structuring element to extract the shapes of mineral crystals.

For code that you can copy and paste into an Editor window, see "Example Code: Using MORPH_CLOSE" on page 502 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load a grayscale color table:

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

2. Select the file, read the data and get the image dimensions:

```
file = FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])
img = READ_PNG(file)
dims = SIZE(img, /DIMENSIONS)
```

3. Using the dimensions of the image add a border for display purposes:

```
padImg = REPLICATE(0B, dims[0]+10, dims[1]+10)
padImg [5,5] = img
```

4. Get the padded image size, create a window and display the original image:

```
dims = SIZE(padImg, /DIMENSIONS)
WINDOW, 0, XSIZE=2*dims[0], YSIZE=2*dims[1], $
   TITLE='Defining Shapes with the Closing Operator'
TVSCL, padImg, 0
```

5. Using DIST, define a small square structuring element in order to retain the detail and angles of the image features:

```
side = 3
strucElem = DIST(side) LE side
```

**Tip**

Enter `PRINT, strucElem` to view the structure created by the previous statement.

6. Apply MORPH_CLOSE to the image and display the resulting image:

```
closeImg = MORPH_CLOSE(padImg, strucElem, /GRAY)
TVSCL, closeImg, 1
```

The following figure shows the original image (left) and the results of applying the closing operator (right). Notice that the closing operation has removed much of the small, dark noise from the background of the image, while maintaining the characteristics of the foreground features.

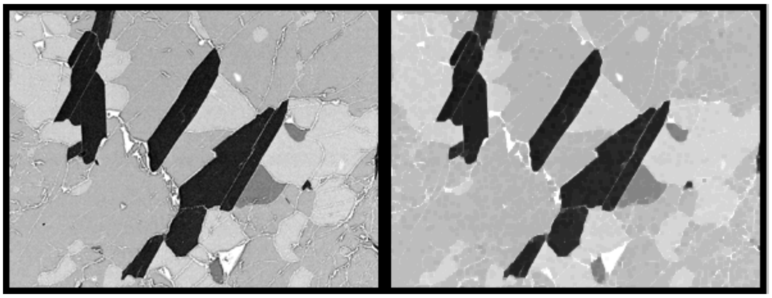


*Figure 11-8: Original (left) and Closed Image (right)*

7. Determine a threshold value, using an intensity histogram as a guide:

```
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(closeImg)
```

**Note**

Using an intensity histogram as a guide for determining threshold values is described in the section, "Determining Intensity Values When Thresholding and Stretching Images" on page 486.

8. Threshold the original image and display the resulting binary image:

```
binaryImg = padImg LE 160
WSET, 0
TVSCL, binaryImg, 2
```

9. Now display a binary version of the closed image:

```
binaryClose = closeImg LE 160
TVSCL, binaryClose, 3
```

The results of thresholding the original and closed image using the same intensity value clearly display the actions of the closing operator. The dark background noise has been removed, much as if a dilation operation had been applied, yet the sizes of the foreground features have been maintained.

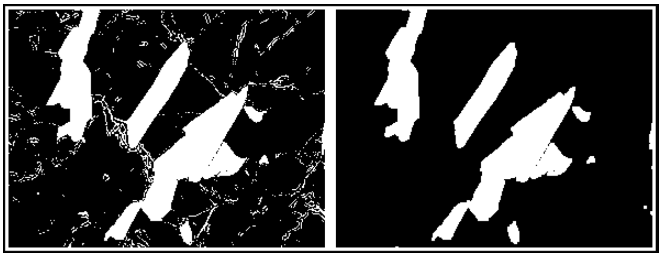


*Figure 11-9: Threshold of Original Image (left) and Closed Image (right)*

## Example Code: Using MORPH_CLOSE

Copy and paste the following text into the IDL Editor window. After saving the file as `MorphCloseExample.pro`, compile and run the program to reproduce the previous example.

```
PRO MorphCloseExample

; Prepare the display device and load grayscale color
; table.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 0

; Select and open the image file.
file = FILEPATH('mineral.png', $
   SUBDIRECTORY=['examples', 'data'])
img = READ_PNG(file)

; Get the image dimensions, prepare a window and
; display the image.
dims = SIZE(img, /DIMENSIONS)

; Pad the image and get the new dimensions.
padImg = REPLICATE(0B, dims[0]+10, dims[1]+10)
padImg [5, 5] = img
dims = SIZE(padImg, /DIMENSIONS)

; Display the padded image.
WINDOW, 0, XSIZE = 2*dims[0], YSIZE = 2*dims[1], $
   TITLE = 'Extracting Shapes with the Closing Operator'
TVSCL, padImg, 0
```

```
; Define the size of the structuring element
;  and create the square.
side = 3
strucElem = DIST(side) LE side
PRINT, strucElem

; Apply the closing operator to the image and display
; it.
closeImg = MORPH_CLOSE(padImg, strucElem, /GRAY)
TVSCL, closeImg, 1

; Create a window and display an intensity histogram
; to help determine the threshold intensity value.
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(closeImg)

; Display a binary version of the original image.
binaryImg = padImg LE 160
WSET, 0
TVSCL, binaryImg, 2

; Display a binary version of the closed image for
; for comparison with the original.
binaryClose = closeImg LE 160
TVSCL, binaryClose, 3

END
```

## Detecting Peaks of Brightness

The morphological top-hat operation, MORPH_TOPHAT, is also known as a peak detector. This operator extracts only the brightest pixels from the original grayscale image by first applying an opening operation to the image and then subtracting the result from the original image. The top-hat operation is especially useful when identifying small image features with high levels of brightness.

The following example applies the top-hat operation to an image of a mature *Rhinosporidium seeberi* sporangium (spore case) with endospores. The circular endospores will be extracted using a small disk-shaped structuring element. The top-hat morphological operation effectively highlights the small bright endospores within the image.

For code that you can copy and paste into an Editor window, see "Example Code: Detecting Bright Peaks with MORPH_TOPHAT" on page 506 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load a grayscale color table:

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

2. Select and open the image file as a grayscale image:

```
file = FILEPATH('r_seeberi_spore.jpg', $
    SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, img, /GRAYSCALE
```

3. Get the image dimensions, and add a border for display purposes:

```
dims = SIZE(img, /DIMENSIONS)
padImg = REPLICATE(0B, dims[0]+10, dims[1]+10)
padImg [5,5] = img
```

4. Get the new dimensions, create a window and display the original image:

```
dims = SIZE(padImg, /DIMENSIONS)
WINDOW, 1, XSIZE = 2*dims[0], YSIZE = 2*dims[1], $
    TITLE = 'Detecting Small Features with MORPH_TOPHAT'
TVSCL, padImg, 0
```

5. After examining the structures you want to extract from the image (the small bright specks), define a circular structuring element with a small radius:

```
radius = 3
strucElem = SHIFT(DIST(2*radius+1), radius, radius) LE radius
```

**Tip**

Enter `PRINT, strucElem` to view the structure created by the previous statement.

6. Apply MORPH_TOPHAT to the image and display the results:

```
tophatImg = MORPH_TOPHAT(padImg, strucElem)
TVSCL, tophatImg, 1
```

The following figure shows the original image (left) and the peaks of brightness that were detected after the top-hat operation subtracted an opened image from the original image (right).

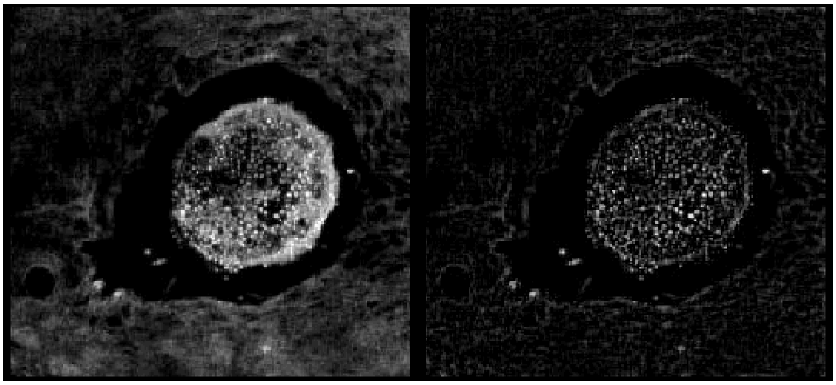


*Figure 11-10: Original (left) and Top-hat Image (right)*

7. Determine an intensity value with which to stretch the image using an intensity histogram as a guide:

```
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(padImg)
```

**Note**

Using an intensity histogram as a guide for determining intensity values is described in the section, "Determining Intensity Values When Thresholding and Stretching Images" on page 486.

8. Highlight the brighter image features by displaying a stretched version of the image:

   ```
   stretchImg = tophatImg < 70
   WSET, 0
   TVSCL, stretchImg, 2
   ```

   Pixels with values greater than 70 are assigned the maximum pixel value (white) and the remaining pixels are scaled across the full range of intensities.

9. Create a binary mask of the image to display only the brightest pixels:

   ```
   threshImg = tophatImg GE 60
   TVSCL, threshImg, 3
   ```

   The stretched top-hat image (left) and the image after applying a binary mask (right) are shown in the following figure. The endospores within the image have been successfully highlighted and extracted using the MORPH_TOPHAT function.

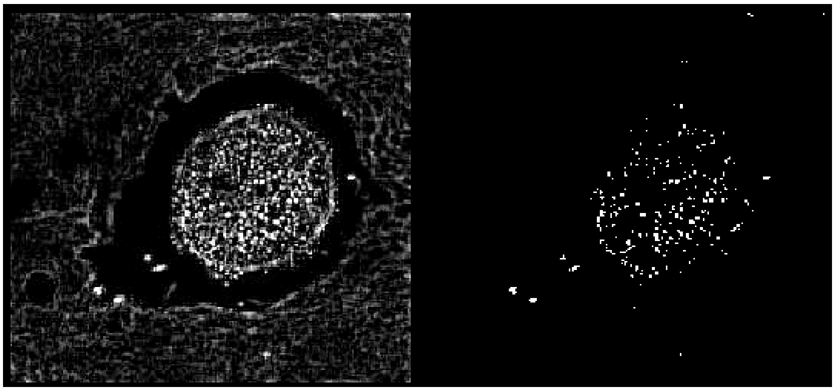


*Figure 11-11: Stretched Top-hat Image (left) and Binary Mask (right)*

## Example Code: Detecting Bright Peaks with MORPH_TOPHAT

Copy and paste the following text into the IDL Editor window. After saving the file as `MorphTophatExample.pro`, compile and run the program to reproduce the previous example.

```
PRO MorphTophatExample

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

```
; Select and open the image file.
file = FILEPATH('r_seeberi_spore.jpg',$
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, img, /GRAYSCALE

; Get the image dimensions, create a window and
; display image.
dims = SIZE(img, /DIMENSIONS)

; Pad the image.
padImg = REPLICATE(0B, dims[0]+10, dims[1]+10)
padImg [5,5] = img

; Get the new dimensions, create a window and display
; the image.
dims = SIZE(padImg, /DIMENSIONS)
WINDOW, 0, XSIZE = 2*dims[0], YSIZE = 2*dims[1], $
   TITLE='Detecting Small Features with MORPH_TOPHAT'
TVSCL, padImg, 0

; Define and create the structuring element.
radius = 3
strucElem = SHIFT(DIST(2*radius+1), $
   radius, radius) LE radius

; Apply the top-hat operator to the image and display
; it.
tophatImg = MORPH_TOPHAT(padImg, strucElem)
TVSCL, tophatImg , 1

; Create a window and display an intensity histogram
; to help determine the threshold intensity value.
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(padImg)

; Stretch and redisplay the image.
WSET, 0
stretchImg = tophatImg < 70
TVSCL, stretchImg, 2

; Threshold and display the binary image.
threshImg = tophatImg GE 60
TVSCL, threshImg, 3

END
```

## Creating Image Object Boundaries

The WATERSHED function applies the watershed operation to grayscale images. This operation creates boundaries in an image by detecting borders between poorly distinguished image areas that contain similar pixel values.

To understand the watershed operation, imagine translating the brightness of the image pixels into height. The brightest pixels become tall peaks and the darkest pixels become basins or depressions. Now imagine flooding the image. The watershed operation detects boundaries among areas with nearly the same value or height by noting the points where single pixels separate two similar areas. The points where these areas meet are then translated into boundaries.

**Note**

Images are usually smoothed before applying the watershed operation. This removes noise and small, unimportant fluctuations in the original image that can produce oversegmentation and a lack of meaningful boundaries.

The following example combines an image containing the boundaries defined by the watershed operation and the original image, a 1982 Landsat satellite image of the Barringer Meteor Crater in Arizona. For code that you can copy and paste into an Editor window, see “Example Code: Detecting Boundaries with WATERSHED” on page 512 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load the grayscale color table:

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

2. Select and open the image of Barringer Meteor Crater, AZ:

```
file = FILEPATH('meteor_crater.jpg', $
    SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, img, /GRAYSCALE
```

3. Get the image size and create a window:

```
dims = SIZE(img, /DIMENSIONS)
WINDOW, 0, XSIZE = 3*dims[0], YSIZE = 2*dims[1]
```

4. Display the original image, annotating it using the XYOUTS procedure:

```
TVSCL, img, 0
XYOUTS, 50, 444, 'Original Image', Alignment = .5, $
    /DEVICE, COLOR = 255
```

5. Using /EDGE_TRUNCATE to avoid spikes along the edges, smooth the image to avoid oversegmentation and display the smoothed image:

```
smoothImg = smooth(7, /EDGE_TRUNCATE)
TVSCL, smoothImg, 1
XYOUTS, (60 + dims[0]), 444, 'Smoothed Image', $
    Alignment = .5, /DEVICE, COLOR = 255
```

   The following figure shows that the smoothing operation retains the major features within the image.

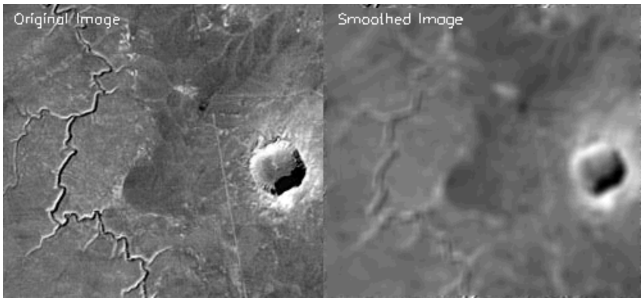


*Figure 11-12: Smoothing the Original Image*

6. Define the radius of the structuring element and create the disk:

```
radius = 3
strucElem = SHIFT(DIST(2*radius+1), radius, radius) LE radius
```

**Tip**

Enter `PRINT, strucElem` to view the structure created by the previous statement.

7. Use the top-hat operation before using watershed to highlight the bright areas within the image.

```
tophatImg = MORPH_TOPHAT(smoothImg, strucElem)
```

8. Display the image:

```
TVSCL, tophatImg, 2
XYOUTS, (60 + 2*dims[0]), 444, 'Top-hat Image', $
   Alignment = .5, /DEVICE, COLOR = 255
```

9. Determine an intensity value with which to stretch the image using an intensity histogram as a guide:

```
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(smoothImg)
```

An intensity histogram of the smoothed image is used instead of the top-hat image since it was empirically determined that the top-hat histogram did not provide the required information.

**Note**

Using an intensity histogram as a guide for determining intensity values is described in the section, "Determining Intensity Values When Thresholding and Stretching Images" on page 486.

10. Stretch the image to set all pixels with a value greater than 70 to the maximum pixel value (white) and display the results:

```
WSET, 0
tophatImg = tophatImg < 70
TVSCL, tophatImg
XYOUTS, 75, 210, 'Stretched Top-hat Image', $
    Alignment = .5, /DEVICE, COLOR = 255
```

The original top-hat image (left) and the results of stretching the image (right) are shown in the following figure.

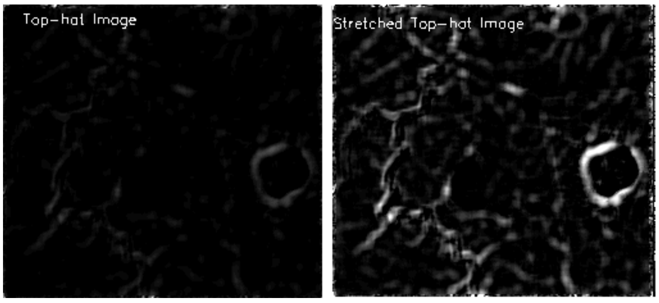


*Figure 11-13: Original (left) and Stretched Top-hat Image (right)*

11. Apply the WATERSHED function to the stretched top-hat image. Specify 8-neighbor connectivity to survey the eight closest pixels to the given pixel, resulting in fewer enclosed regions, and display the results:

```
watershedImg = WATERSHED(tophatImg, CONNECTIVITY = 8)
TVSCL, watershedImg, 4
XYOUTS, (70 + dims[0]), 210, 'Watershed Image', $
    Alignment = .5, /DEVICE, COLOR = 255
```

12. Combine the watershed image with the original image and display the result:

```
img [WHERE (watershedImg EQ 0)]= 0
TVSCL, img, 5
XYOUTS, (70 + 2*dims[0]), 210, 'Watershed Overlay', $
    Alignment = .5, /DEVICE, COLOR = 255
```

The following display shows all images created in the previous example. The final image, shown in the lower right-hand corner of the following figure, shows the original image with an overlay of the boundaries defined by the watershed operation.

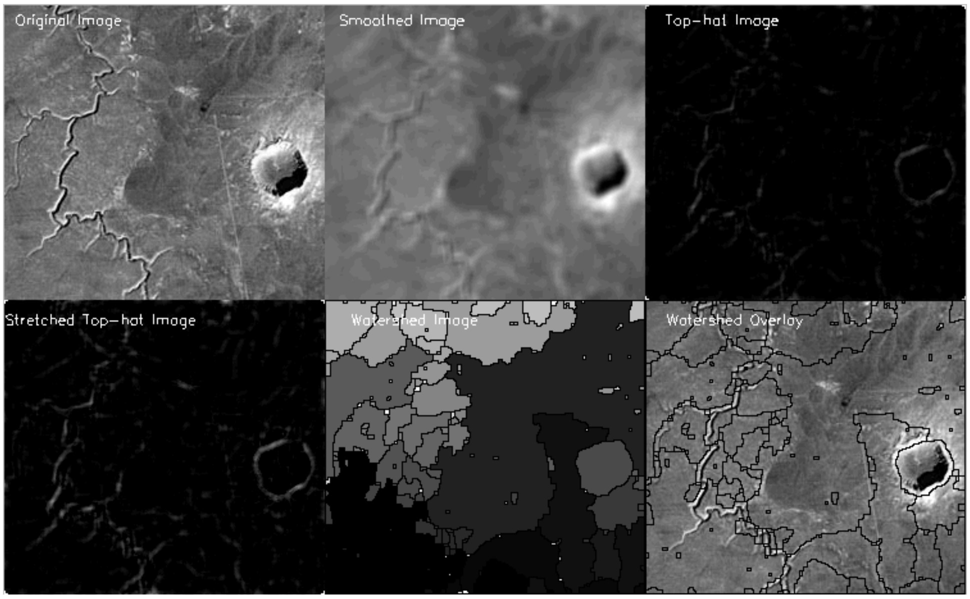


*Figure 11-14: Boundaries Defined by the Watershed Operation*

## Example Code: Detecting Boundaries with WATERSHED

Copy and paste the following text into the IDL Editor window. After saving the file as `WatershedExample.pro`, compile and run the program to reproduce the previous example.

```
PRO WatershedExample

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

; Select and open image of Barrington Meteor Crater,
; AZ.
file = FILEPATH('meteor_crater.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, img, /GRAYSCALE

; Get the image size, create a window and display the
; image.
dims = SIZE(img, /DIMENSIONS)
WINDOW, 0, XSIZE = 3*dims[0], YSIZE = 2*dims[1], $
   TITLE = 'Defining Boundaries with WATERSHED'

; Display the original image.
TVSCL, img, 0
XYOUTS, 50, 444, 'Original Image', Alignment = .5, $
   /DEVICE, COLOR = 255

; Smooth the image and display it.
smoothImg = SMOOTH(img, 7, /EDGE_TRUNCATE)
TVSCL, smoothImg, 1
XYOUTS, (60 + dims[0]), 444, 'Smoothed Image', $
   ALIGNMENT = .5, /DEVICE, COLOR = 255

; Define the radius and create the structuring element.
radius = 3
strucElem = SHIFT(DIST(2*radius+1), $
   radius, radius) LE radius

; Use the top-hat operator before using watershed to
; highlight bright areas within the image.
tophatImg = MORPH_TOPHAT(smoothImg, strucElem)

; Display the image.
TVSCL, tophatImg, 2
XYOUTS, (60 + 2*dims[0]), 444, 'Top-hat Image', $
   ALIGNMENT = .5, /DEVICE, COLOR = 255
```

```
; Determine the intensity value using a histogram as a
; guide. Stretch the image.
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(smoothImg)
tophatImg = tophatImg < 70

; Display the stretched image.
WSET, 0
TVSCL, tophatImg
XYOUTS, 75, 210, 'Stretched Top-hat Image', $
   ALIGNMENT = .5, /DEVICE, COLOR = 255

; Use the WATERSHED operator to create boundaries
; and display the results.
watershedImg = WATERSHED(tophatImg, CONNECTIVITY = 8)
TVSCL, watershedImg, 4
XYOUTS, (70 + dims[0]), 210, 'Watershed Image', $
   ALIGNMENT = .5, /DEVICE, COLOR = 255

; Overlay the boundaries defined by watershed onto
; the original image.
img [WHERE (watershedImg EQ 0)] = 0
TVSCL, img, 5
XYOUTS, (70 + 2*dims[0]), 210, 'Watershed Overlay', $
   ALIGNMENT = .5, /DEVICE, COLOR = 255

END
```

## Selecting Specific Image Objects

The hit-or-miss morphological operation is used primarily for identifying specific shapes within binary images. The MORPH_HITORMISS function uses two structuring elements; a “hit” structure and a “miss” structure. The operation first applies an erosion operation with the hit structure to the original image. The operation then applies an erosion operator with the miss structure to an inverse of the original image. The matching image elements entirely *contain the hit structure* and are entirely and solely *contained by the miss structure*.

**Note**

An image must be padded with a border equal to one half the size of the structuring element if you want the hit-or-miss operation to be applied to image elements occurring along the edges of the image.

The hit-or-miss operation is very sensitive to the shape, size and rotation of the two structuring elements. Hit and miss structuring elements must be specifically designed to extract the desired geometric shapes from each individual image. When dealing with complicated images, extracting specific image regions may require multiple applications of hit and miss structures, using a range of sizes or several rotations of the structuring elements.

The following example uses the image of the *Rhinosporidium seeberi* parasitic protozoans, containing simple circular shapes. After specifying distinct hit and miss structures, the elements of the image that meet the hit and miss conditions are identified and overlaid on the original image.

For code that you can copy and paste into an Editor window, see “Example Code: Identifying Objects with MORPH_HITORMISS” on page 518 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load a grayscale color table:

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

2. Select and open the image file:

```
file = FILEPATH('r_seeberi.jpg', $
    SUBDIRECTORY = ['examples','data'])
READ_JPEG, file, img, /GRAYSCALE
```

3. Pad the image so that objects at the edges of the image are not discounted:

   ```
   dims = SIZE(img, /DIMENSIONS)
   padImg = REPLICATE(0B, dims[0]+10, dims[1]+10)
   padImg [5,5] = img
   ```

   Failing to pad an image causes all objects occurring at the edges of the image to fail the hit and miss conditions.

4. Get the image dimensions, create a window and display the padded image:

   ```
   dims = SIZE(padImg, /DIMENSIONS)
   WINDOW, 0, XSIZE = 3*dims[0], YSIZE = 2*dims[1], $
       TITLE='Displaying Hit-or-Miss Matches'
   TVSCL, padImg, 0
   ```

5. Define the radius of the structuring element and create a large, disk-shaped element to extract the large, circular image objects:

   ```
   radstr = 7
   strucElem = SHIFT(DIST(2*radstr+1), radstr, radstr) LE radstr
   ```

**Tip**

Enter `PRINT, strucElem` to view the structure created by the previous statement.

6. Apply MORPH_OPEN for a smoothing effect and display the image:

   ```
   openImg = MORPH_OPEN(padImg, strucElem, /GRAY)
   TVSCL, openImg, 1
   ```

7. Since the hit-or-miss operation requires a binary image, display an intensity histogram as a guide for determining a threshold value:

   ```
   WINDOW, 2, XSIZE = 400, YSIZE = 300
   PLOT, HISTOGRAM(openImg)
   ```

   **Note**

   Using an intensity histogram as a guide for determining threshold values is described in the section, "Determining Intensity Values When Thresholding and Stretching Images" on page 486.

8. Create a binary image by retaining only those image elements with pixel values greater than or equal to 150 (the bright foreground objects):

   ```
   threshImg = openImg GE 150
   WSET, 0
   TVSCL, threshImg, 2
   ```

The results of opening (left) and thresholding (right) are shown in the following figure.

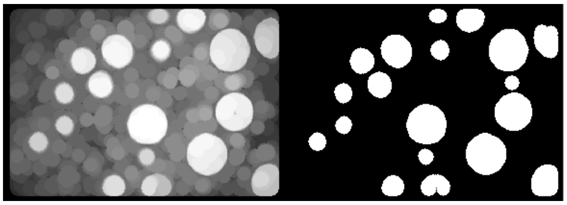


*Figure 11-15: Results of Opening (left) and Thresholding (right)*

9. Create the structuring elements for the hit-or-miss operation:

```
radhit = 7
radmiss = 23
hit = SHIFT(DIST(2*radhit+1), radhit, radhit) LE radhit
miss = SHIFT(DIST(2*radmiss+1), radmiss, radmiss) GE radmiss
```

While the shapes of the structuring elements are purposefully circular, the sizes were chosen after empirically testing, seeking elements suitable for this example.

**Tip**

Enter `PRINT, hit` or `PRINT, miss` to view the structures.

The following figures shows the hit and miss structuring elements and the binary image. Knowing that the region must enclose the hit structure and be surrounded by a background area at least as large as the miss structure, can you predict which regions will be "matches?"

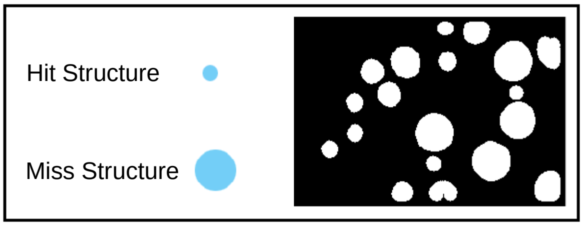


*Figure 11-16: Applying the Hit and Miss Structuring Elements to a Binary Image*

10. Apply the MORPH_HITORMISS function to the binary image. Image regions matching the hit and miss conditions are designated at *matches*:

    ```
    matches = MORPH_HITORMISS(threshImg, hit, miss)
    ```

11. Display the elements matching the hit and miss conditions, dilating the elements to the radius of a *hit*:

    ```
    dmatches = DILATE(matches, hit)
    TVSCL, dmatches, 3
    ```

12. Display the original image overlaid with the matching elements:

    ```
    padImg [WHERE (dmatches EQ 1)] = 1
    TVSCL, padImg, 4
    ```

The following figure shows the elements of the image which matched the hit and miss conditions, having a radius of at least 7 (the hit structure), yet fitting entirely inside a structure with a radius of 23 (the miss structure).

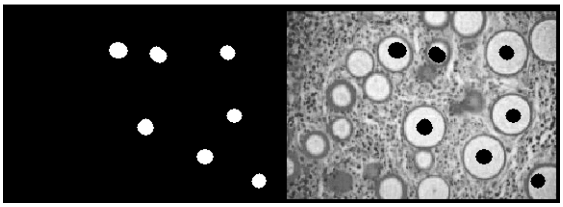


*Figure 11-17: Image Elements Matching Hit and Miss Conditions*

Initially, it may appear that more regions should have been “matches” since they met the hit condition of having a radius of 7 or more. However, as the following figure shows, many such regions failed the miss condition since neighboring regions impinged upon the miss structure. Such a region appears on the left in the following figure.

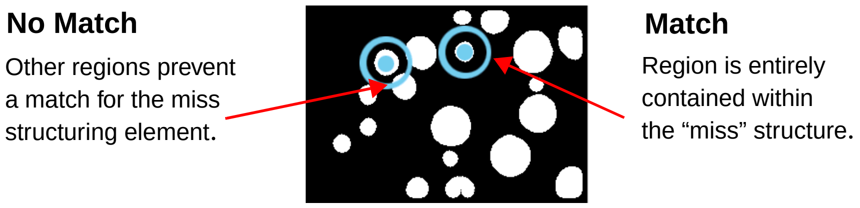

No Match
Other regions prevent a match for the miss structuring element.
Match
Region is entirely contained within the “miss” structure.

*Figure 11-18: Example of Hit and Miss Relationship*

Considering the simplicity of the previous image, it is understandable that selecting hit and miss structures for more complex images can require significant empirical testing. It is to your advantage to keep in mind how sensitive the hit-or-miss operation is to the shapes, sizes and rotations of the hit and miss structures.

## Example Code: Identifying Objects with MORPH_HITORMISS

Copy and paste the following text into the IDL Editor window. After saving the file as `MorphHitorMissExample.pro`, compile and run the program to reproduce the previous example.

```
PRO MorphHitorMissExample

; Prepare the display device and load a grayscale color
; table.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 0

; Select and open an image of the parasitic protozoa.
file = FILEPATH('r_seeberi.jpg', $
   SUBDIRECTORY=['examples','data'])
READ_JPEG, file, img, /GRAYSCALE

; Pad the image to avoid discounting edge objects.
dims = SIZE(img, /DIMENSIONS)
padImg = REPLICATE(0B, dims[0]+10, dims[1]+10)
padImg[5, 5] = img

; Get the image dimensions.
dims = SIZE(padImg, /DIMENSIONS)

; Prepare a window and display the image.
WINDOW, 0, XSIZE=3*dims[0], YSIZE=2*dims[1], $
   TITLE='Displaying Hit-or-Miss Matches'
TVSCL, padImg, 0

; Define and create a structuring element for the
; opening operator.
radstr = 7
strucElem = SHIFT(DIST(2*radstr+1), $
   radstr, radstr) LE radstr

; Apply the opening operator for a smoothing effect.
openImg = MORPH_OPEN(padImg, strucElem, /GRAY)
TVSCL, openImg, 1

; Use an intensity histogram as a guide for
; thresholding.
```

```
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(openImg)

; Threshold the image.
threshImg = openImg GE 150
WSET, 0
TVSCL, threshImg, 2

; Create the structuring elements for the hit-or-miss
; operator.
radhit = 7
radmiss = 23
hit = SHIFT(DIST(2*radhit+1), radhit, radhit) LE radhit
miss = SHIFT(DIST(2*radmiss+1), $
   radmiss, radmiss) GE radmiss

; Using structuring elements, define matching regions.
matches = MORPH_HITORMISS(threshImg, hit, miss)

; Display the regions matching hit and miss conditions.
; Dilate the matches to the radius of a 'hit'.
dmatches = DILATE(matches, hit)
TVSCL, dmatches, 3

; Display the original image overlaid with the matching
; regions.
padImg [WHERE (dmatches EQ 1)] = 1
TVSCL, padImg, 4

END
```

## Detecting Edges of Image Objects

The MORPH_GRADIENT function applies the gradient operation to a grayscale image. This operation highlights object edges by subtracting an eroded version of the original image from a dilated version. Repeatedly applying the gradient operator or increasing the size of the structuring element results in wider edges.

The following example extracts image features by applying the morphological gradient operation to an image of the Mars globe. For code that you can copy and paste into an Editor window, see "Example Code: Displaying Edges with MORPH_GRADIENT" on page 522 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load the grayscale color table:

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

2. Select and read in the file:

```
file = FILEPATH('marsglobe.jpg', $
   SUBDIRECTORY=['examples', 'data'])
READ_JPEG, file, image, /GRAYSCALE
```

3. Get the image size, create a window and display the smoothed image:

```
dims = SIZE(image, /DIMENSIONS)
WINDOW, 0, XSIZE =2*dims[0], YSIZE = 2*dims[1], $
   TITLE = 'Original and MORPH_GRADIENT Images'
```

The original image is shown in the following figure.

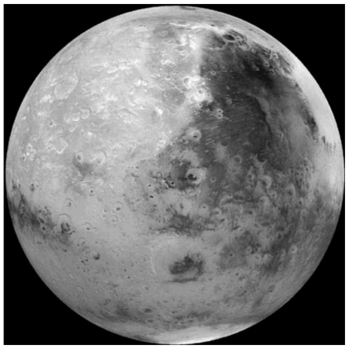


*Figure 11-19: Image of Mars Globe*

4. Preserve the greatest amount of detail within the image by defining a structuring element with a radius of 1, avoiding excessively thick edge lines:

```
radius = 1
strucElem = SHIFT(DIST(2*radius+1), radius, radius) LE radius
```

**Tip**

Enter `PRINT, strucElem` to view the structure created by the previous statement.

5. Apply the MORPH_GRADIENT function to the image and display the result:

```
morphImg = MORPH_GRADIENT(image, strucElem)
TVSCL, morphImg, 2
```

6. To more easily distinguish features within the dark image, prepare to stretch the image by displaying an intensity histogram:

```
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(1-image)
```

The previous line returns a histogram of an inverse of the original image since the final display will also be an inverse display for showing the greatest detail.

7. Stretch the image and display its inverse:

```
WSET, 0
TVSCL, 1-(morphImg < 87 ), 3
```

The following figure displays the initial and stretched gradient images.

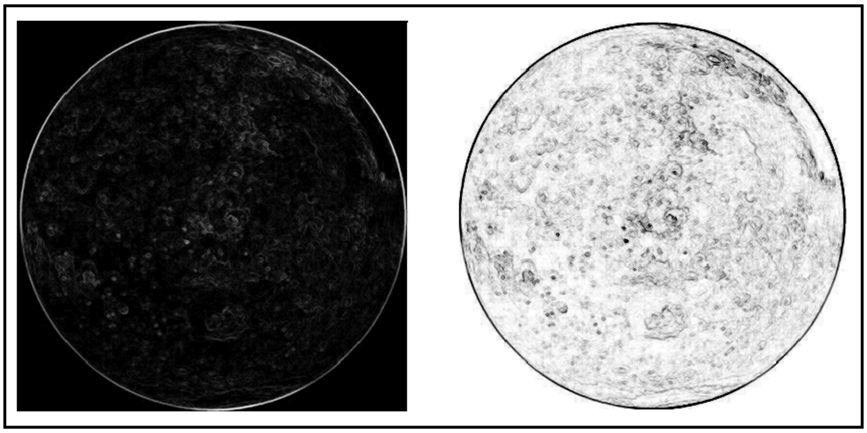


*Figure 11-20: Initial and Stretched Results of the Gradient Operation*

## Example Code: Displaying Edges with MORPH_GRADIENT

Copy and paste the following text into the IDL Editor window. After saving the file as `MorphGradientEx.pro`, compile and run the program to reproduce the previous example.

```
PRO MorphGradientEx

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

; Select and read in the file.
file = FILEPATH('marsglobe.jpg', $
   SUBDIRECTORY = ['examples', 'data'] )
READ_JPEG, file, image, /GRAYSCALE

; Get the image size, create a window and display the
; image.
dims = SIZE(image, /DIMENSIONS)
WINDOW, 0, XSIZE =2*dims[0], YSIZE = 2*dims[1], $
   TITLE = 'Original and MORPH_GRADIENT Images'
TVSCL, image, 0

; Define the structuring element, apply the
; morphological operator and display the image.
radius = 1
strucElem = SHIFT(DIST(2*radius+1), $
   radius, radius) LE radius
morphImg = MORPH_GRADIENT(image, strucElem)
TVSCL, morphImg, 2

; Display an inverse intesity histogram to determine
; stretch intensity value.
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(1 - image)

; Display inverse of stretched gradient image.
WSET, 0
TVSCL, 1 - (morphImg < 87 ), 3

END
```

## Creating Distance Maps

The MORPH_DISTANCE function computes a grayscale, *N*-dimensional distance map from a binary image. The map shows, for each foreground pixel, the distance to the nearest background pixel using a given norm. The norm simply defines how neighboring pixels are sampled. See the MORPH_DISTANCE description *in the IDL Reference Guide* for full details. The resulting values in the grayscale image denote the distance from the surveyed pixel to the nearest background pixel. The brighter the pixel, the farther it is from the background.

The following example applies the distance transformation to a grayscale image of a cultured sample of *Neocosmospora vasinfecta*, a common fungal plant pathogen. For code that you can copy and paste into an Editor window, see "Example Code: Displaying Distances with MORPH_DISTANCE" on page 525 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load a grayscale color table:

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

2. Select and load an image:

```
file = FILEPATH('n_vasinfecta.jpg', $
    SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, img, /GRAYSCALE
```

3. Get the size of the image and create a border for display purposes:

```
dims = SIZE(img, /DIMENSIONS)
padImg = REPLICATE(0B, dims[0]+10, dims[1]+10)
padImg[5,5] = img
```

4. Get the dimensions of the padded image, create a window and display the original image:

```
dims = SIZE(padImg, /DIMENSIONS)
WINDOW, 0, XSIZE = 2*dims[0], YSIZE = 2*dims[1], $
    TITLE='Distance Map and Overlay of Binary Image'
TVSCL, padImg, 0
```

5. Use an intensity histogram as a guide for creating a binary image:

```
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(padImg)
```

**Note**

Using an intensity histogram as a guide for determining intensity values is described in the section, "Determining Intensity Values When Thresholding and Stretching Images" on page 486.

6. Before using the distance transform, the grayscale image must be translated into a binary image. Create and display a binary image containing the dark tubules. Threshold the image, masking out pixels with values greater than 120:

```
binaryImg = stretchImg LT 120
WSET, 0
TVSCL, binaryImg, 1
```

The original image (left) and binary image (right) appear in the following figure.

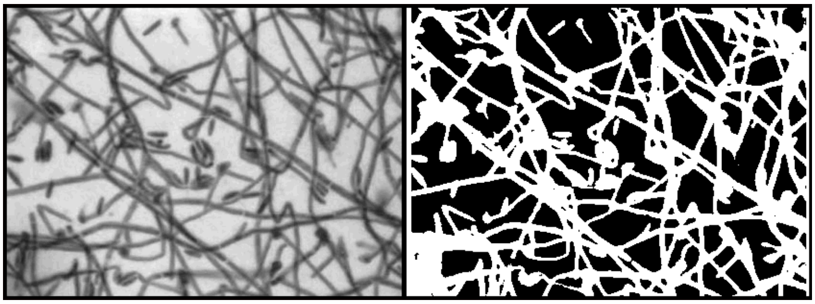


*Figure 11-21: Original Image (left) and Binary Image (right)*

7. Compute the distance map using MORPH_DISTANCE, specifying "chessboard" neighbor sampling, which surveys each horizontal, vertical and diagonal pixel touching the pixel being surveyed, and display the result:

```
distanceImg = MORPH_DISTANCE(binaryImg, NEIGHBOR_SAMPLING =
1)
TVSCL, distanceImg, 2
```

8. Display a combined image of the distance map and the binary image. Black areas within the binary image (having a value of 0) are assigned the maximum pixel value occurring in the distance image:

```
distanceImg [WHERE (binaryImg EQ 0)] = MAX(distanceImg)
TVSCL, distanceImg, 3
```

The distance map (left) and resulting blended image (right) show the distance of each image element pixel from the background.

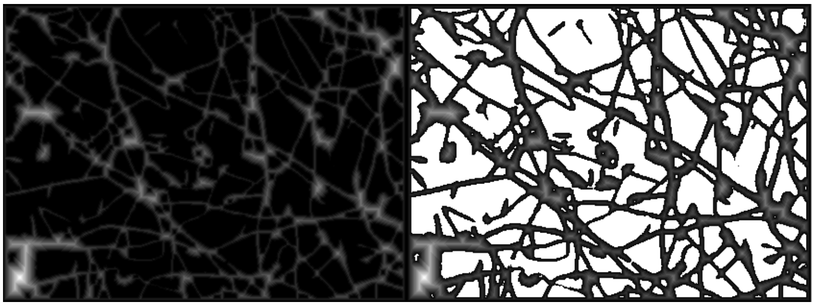


*Figure 11-22: Distance Map (left) and Merged Map and Binary Image (right)*

## Example Code: Displaying Distances with MORPH_DISTANCE

Copy and paste the following text into the IDL Editor window. After saving the file as `MorphDistanceExample.pro`, compile and run the program to reproduce the previous example.

```
PRO MorphDistanceExample

; Prepare the display device and load grayscale color
; table.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 0

; Select and load an image.
file  = FILEPATH('n_vasinfecta.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, img, /GRAYSCALE
dims = SIZE(img, /DIMENSIONS)

; Pad the image for display purposes.
padImg = REPLICATE(0B, dims[0] + 10, dims[1] + 10)
padImg[5, 5] = img

; Get the size of the padded image.
dims = SIZE(padImg, /DIMENSIONS)
WINDOW, 0, XSIZE = 2*dims[0], YSIZE = 2*dims[1], $
   TITLE = 'Distance Map and Overlay of Thresholded Image'
TVSCL, padImg, 0

; Use an intensity histogram to help determine
; threshold intensity value.
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(padImg)

; Create a binary image.
binaryImg = padImg LT 120
WSET, 0
TVSCL, binaryImg, 1

; Compute distance map using "chessboard" neighbor
; sampling.
distanceImg = MORPH_DISTANCE(binaryImg, $
   NEIGHBOR_SAMPLING = 1)
TVSCL, distanceImg, 2

; Overlay the distance map onto the binary image. Black
; areas within the binary image are assigned the maximum
; pixel brightness within the distance image.
distanceImg[WHERE(binaryImg EQ 0)] = MAX(distanceImg)
TVSCL, distanceImg, 3

END
```

## Thinning Image Objects

The MORPH_THIN function performs a thinning operation on binary images. After designating “hit” and “miss” structures, the thinning operation applies the hit-or-miss operator to the original image and then subtracts the result from the original image.

The thinning operation is typically applied repeatedly, leaving only pixel-wide linear representations of the image objects. The thinning operation halts when no more pixels can be removed from the image. This occurs when the thinning operation (applying the hit and miss structures and subtracting the result) produces no change in the input image. At this point, the thinned image is identical to the input image.

When repeatedly applying the thinning operation, each successive iteration uses hit and miss structures that have had the individual elements of the structures rotated one position clockwise. For example, the following 3-by-3 arrays show the initial structure (left) and the structure after rotating the elements one position clockwise around the central value (right).

```
h0 =  [[0,0,0],              h1 = [[0,0,0],
        [0,1,0],                    [1,1,0],
        [1,1,1]]                    [1,1,0]]
```

The following example uses eight rotations of each of the original hit and miss structuring elements. The repeated application of the thinning operation results in an image containing only pixel-wide lines indicating the original grains of pollen. This example displays the results of each successive thinning operation.

**Note**

This example uses a file from the `examples/demo/demodata` directory of your installation. If you have not already done so, you will need to install “IDL Demos” from your product CD-ROM to install the demo data file needed for this example.

For code that you can copy and paste into an Editor window, see “Example Code: Thinning Image Objects” on page 531 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load a grayscale color table:

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

2. Select and open the image file:

```
file = FILEPATH('pollens.jpg', $
    SUBDIRECTORY = ['examples','demo','demodata'])
READ_JPEG, file, img, /GRAYSCALE
```

3. Get the image dimensions, create a window and display the original image:

```
dims = SIZE(img, /DIMENSIONS)
WINDOW, 0, XSIZE = 2*dims[0], YSIZE = 2*dims[1], $
    TITLE='Original, Binary and Thinned Images'
TVSCL, img, 0
```

4. The thinning operation requires a binary image. Create a binary image, retaining pixels with values greater than or equal to 140, and display the image:

```
binaryImg = img GE 140
TVSCL, binaryImg, 1
```

**Note**

The following lines were used to determine the threshold value:

```
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(img)
```

See "Determining Intensity Values When Thresholding and Stretching Images" on page 486 for details about using a histogram to determine intensity values.

5. Prepare hit and miss structures for thinning. Rotate the outer elements of each successive hit and miss structure one position clockwise:

**Note**

For a version of these structures that is easy to copy and paste into an Editor window, see "Example Code: Thinning Image Objects" on page 531.

```
h0 = [[0b,0,0], $
      [0,1,0], $
      [1,1,1]]
m0 = [[1b,1,1], $
      [0,0,0], $
      [0,0,0]]
h1 = [[0b,0,0], $
      [1,1,0], $
      [1,1,0]]
m1 = [[0b,1,1], $
      [0,0,1], $
      [0,0,0]]
h2 = [[1b,0,0], $
      [1,1,0], $
      [1,0,0]]
m2 = [[0b,0,1], $
      [0,0,1], $
      [0,0,1]]
```

```
h3 = [[1b,1,0], $
      [1,1,0], $
      [0,0,0]]
m3 = [[0b,0,0], $
      [0,0,1], $
      [0,1,1]]
h4 = [[1b,1,1], $
      [0,1,0], $
      [0,0,0]]
m4 = [[0b,0,0], $
      [0,0,0], $
      [1,1,1]]
h5 = [[0b,1,1], $
      [0,1,1], $
      [0,0,0]]
m5 = [[0b,0,0], $
      [1,0,0], $
      [1,1,0]]
h6 = [[0b,0,1], $
      [0,1,1], $
      [0,0,1]]
m6 = [[1b,0,0], $
      [1,0,0], $
      [1,0,0]]
h7 = [[0b,0,0], $
      [0,1,1], $
      [0,1,1]]
m7 = [[1b,1,0], $
      [1,0,0], $
      [0,0,0]]
```

6. Define the iteration variables for the WHILE loop and prepare to pass in the binary image:

```
bCont = 1b
iIter = 1
thinImg = binaryImg
```

7. Enter the following WHILE loop statements into the Editor window. The loop specifies that the image will continue to be thinned with MORPH_THIN until the thinned image is equal to the image input into the loop. Since *thinImg* equals *inputImg*, the loop is exited when a complete iteration produces no changes in the image. In this case, the condition, `bCont eq 1` fails and the loop is exited.

```
WHILE bCont EQ 1b DO BEGIN & $
   PRINT,'Iteration: ', iIter & $
   inputImg = thinImg & $
   thinImg = MORPH_THIN(inputImg, h0, m0) & $
   thinImg = MORPH_THIN(thinImg, h1, m1) & $
   thinImg = MORPH_THIN(thinImg, h2, m2) & $
   thinImg = MORPH_THIN(thinImg, h3, m3) & $
   thinImg = MORPH_THIN(thinImg, h4, m4) & $
   thinImg = MORPH_THIN(thinImg, h5, m5) & $
   thinImg = MORPH_THIN(thinImg, h6, m6) & $
   thinImg = MORPH_THIN(thinImg, h7, m7) & $
   TVSCL, thinImg, 2 & $
   WAIT, 1 & $
   bCont = MAX(inputImg - thinImg) & $
   iIter = iIter + 1 & $
ENDWHILE
```

**Note**

The & after BEGIN and the $ allow you to use the WHILE/DO loop at the IDL command line. These & and $ symbols are not required when the WHILE/DO loop in placed in an IDL program as shown in "Example Code: Thinning Image Objects" on page 531.

8. Display an inverse of the final result:

```
TVSCL, 1 - thinImg, 3
```

The following figure displays the results of the thinning operation, reducing the original objects to a single pixel wide lines.

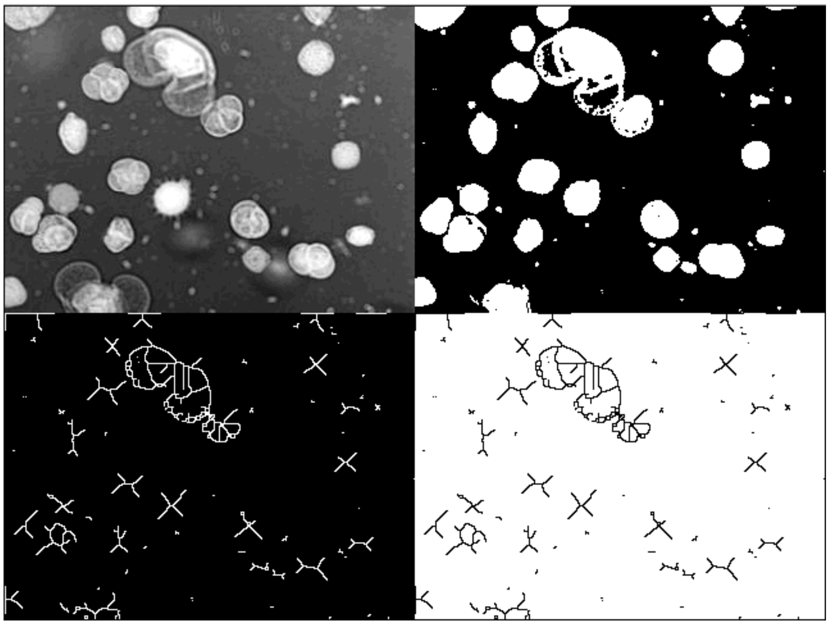


*Figure 11-23: Original Image (top left), Binary Image (top right), Thinned Image (bottom left) and Inverse Thinned Image (bottom right)*

Each successive thinning iteration removed pixels marked by the results of the hit-or-miss operation as long as the removal of the pixels would not destroy the connectivity of the line.

## Example Code: Thinning Image Objects

Copy and paste the following text into the IDL Editor window. After saving the file as `MorphThinExample.pro`, compile and run the program to reproduce the previous example.

**Note**

The following code displays the eight pairs of hit and miss structuring elements on individual lines so that the code can be easily copied into an Editor window. Although it is less visible, the elements of each successive structure are rotated as described in the beginning of this section, "Thinning Image Objects" on page 527.

```
PRO MorphThinExample

; Prepare the display device and load grayscale color
; table.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 0

; Load an image.
file = FILEPATH('pollens.jpg', $
   SUBDIRECTORY = ['examples', 'demo', 'demodata'])
READ_JPEG, file, img, /GRAYSCALE

; Get the image size, prepare a display window and
; display the image.
dims = SIZE(img, /DIMENSIONS)
WINDOW, 0, XSIZE = 2*dims[0], YSIZE = 2*dims[1], $
   TITLE = 'Original, Binary and Thinned Images'
TVSCL, img, 0

; Generate a binary image by thresholding.
binaryImg = img GE 140
TVSCL, binaryImg, 1

; Prepare hit and miss structures for thinning.
h0 = [[0b, 0, 0], [0, 1, 0], [1, 1, 1]]
m0 = [[1b, 1, 1], [0, 0, 0], [0, 0, 0]]
h1 = [[0b, 0, 0], [1, 1, 0], [1, 1, 0]]
m1 = [[0b, 1, 1], [0, 0, 1], [0, 0, 0]]
h2 = [[1b, 0, 0], [1, 1, 0], [1, 0, 0]]
m2 = [[0b, 0, 1], [0, 0, 1], [0, 0, 1]]
h3 = [[1b, 1, 0], [1, 1, 0], [0, 0, 0]]
m3 = [[0b, 0, 0], [0, 0, 1], [0, 1, 1]]
h4 = [[1b, 1, 1], [0, 1, 0], [0, 0, 0]]
m4 = [[0b, 0, 0], [0, 0, 0], [1, 1, 1]]
h5 = [[0b, 1, 1], [0, 1, 1], [0, 0, 0]]
m5 = [[0b, 0, 0], [1, 0, 0], [1, 1, 0]]
h6 = [[0b, 0, 1], [0, 1, 1], [0, 0, 1]]
m6 = [[1b, 0, 0], [1, 0, 0], [1, 0, 0]]
h7 = [[0b, 0, 0], [0, 1, 1], [0, 1, 1]]
m7 = [[1b, 1, 0], [1, 0, 0], [0, 0, 0]]

; Iterate until the thinned image is identical to
; the input image for a given iteration.
bCont = 1b
iIter = 1
thinImg = binaryImg
WHILE bCont EQ 1b DO BEGIN
   PRINT,'Iteration: ', iIter
   inputImg = thinImg
   ; Perform the thinning using the first pair
   ; of structure elements.
   thinImg = MORPH_THIN(inputImg, h0, m0)

   ; Perform the thinning operation using the
   ; remaining structural element pairs.
   thinImg = MORPH_THIN(thinImg, h1, m1)
   thinImg = MORPH_THIN(thinImg, h2, m2)
   thinImg = MORPH_THIN(thinImg, h3, m3)
   thinImg = MORPH_THIN(thinImg, h4, m4)
   thinImg = MORPH_THIN(thinImg, h5, m5)
   thinImg = MORPH_THIN(thinImg, h6, m6)
   thinImg = MORPH_THIN(thinImg, h7, m7)

   ; Display the results of thinning and wait a second for
   ; display purposes.
   TVSCL, thinImg, 2
   WAIT, 1

   ; Test the condition and increment the loop.
   bCont = MAX(inputImg - thinImg)
   iIter = iIter + 1

; End WHILE loop statements.
ENDWHILE

; Show inverse of final result.
TVSCL, 1 - thinImg, 3

END
```

## Combining Morphological Operations

The following example uses a variety of morphological operations to remove bridges from a satellite image of New York waterways. For code that you can copy and paste into an Editor window, see “Example Code: Combining Morphological Operations in Feature Extraction” on page 538 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load a color table:

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

2. Specify the known dimensions and use READ_BINARY to load the image:

```
xsize = 768
ysize = 512
img = READ_BINARY(FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data']), $
    DATA_DIMS = [xsize, ysize])
```

3. Increase the image's contrast and display the image:

```
img = BYTSCL(img)
WINDOW, 1, TITLE = 'Original Image'
TVSCL, img
```

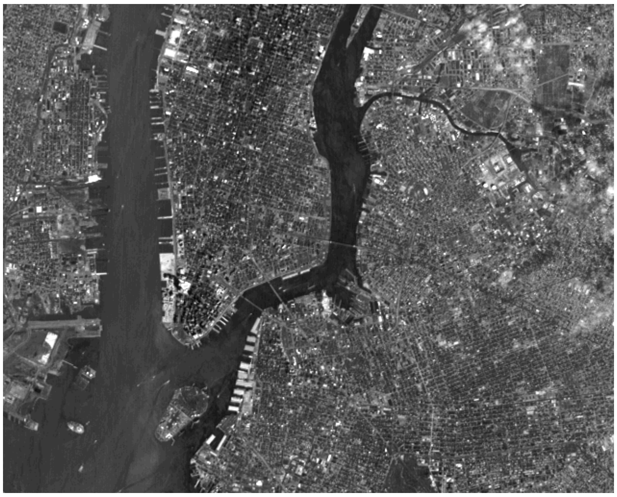


*Figure 11-24: Original Image*

4. Prepare to threshold the image, using an intensity histogram as a guide for determining the intensity value:

   ```
   WINDOW, 4, XSIZE = 400, YSIZE = 300
   PLOT, HISTOGRAM(img)
   ```

**Note**

Using an intensity histogram as a guide for determining threshold values is described in the section, "Determining Intensity Values When Thresholding and Stretching Images" on page 486.

5. Create a mask of the darker pixels that have values less than 70:

   ```
   maskImg = img LT 70
   ```

6. Define and create a small square structuring element, which has a shape similar to the bridges which will be masked out:

   ```
   side = 3
   strucElem = DIST(side) LE side
   ```

7. Remove details in the binary mask's shape by applying the opening operation:

   ```
   maskImg = MORPH_OPEN(maskImg, strucElem)
   ```

8. Fuse gaps in the mask's shape by applying the closing operation and display the image:

   ```
   maskImg = MORPH_CLOSE(maskImg, strucElem)
   WINDOW, 1, title='Mask After Opening and Closing'
   TVSCL, maskImg
   ```

   This results in the following figure:

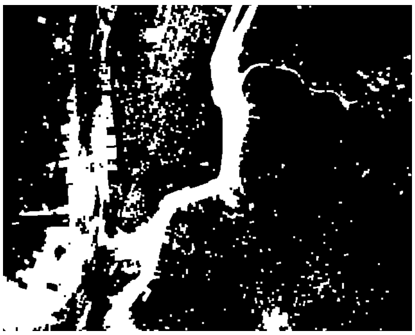


*Figure 11-25: Image Mask After Opening and Closing Operations*

9. Prepare to remove all but the largest region in the mask by labeling the regions:

   ```
   labelImg = LABEL_REGION(maskImg)
   ```

10. Discard the black background by keeping only the white areas of the previous figure:

    ```
    regions = labelImg[WHERE(labelImg NE 0)]
    ```

11. Define *mainRegion* as the area where the population of the *labelImg* region matches the region with the largest population:

    ```
    mainRegion = WHERE(HISTOGRAM(labelImg) EQ $
       MAX(HISTOGRAM(regions)))
    ```

12. Define *maskImg* as the area of *labelImg* equal to the largest region of *mainRegion*, having an index number of 0 and display the image:

    ```
    maskImg = labelImg EQ mainRegion[0]
    Window, 3, TITLE = 'Final Masked Image'
    TVSCL, maskImg
    ```

    This results in a mask of the largest region, the waterways, as shown in the following figure.

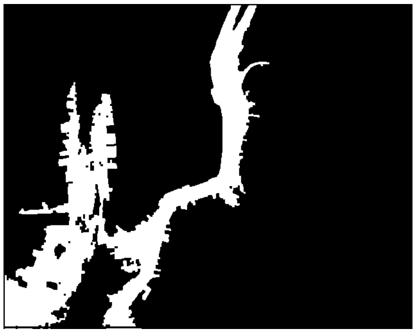


*Figure 11-26: Final Image Mask*

13. Remove noise and smooth contours in the original image:

    ```
    newImg = MORPH_OPEN(img, strucElem, /GRAY)
    ```

14. Replace the new image with the original image, where it's not masked:

    ```
    newImg[WHERE(maskImg EQ 0)] = img[WHERE(maskImg EQ 0)]
    ```

15. View the results using FLICK to alternate the display between the original image and the new image containing the masked areas:

    ```
    WINDOW, 0, XSIZE = xsize, YSIZE = ysize
    FLICK, img, newImg
    ```

    Hit any key to stop the image from flickering. Details of the two images are shown in the following figure.

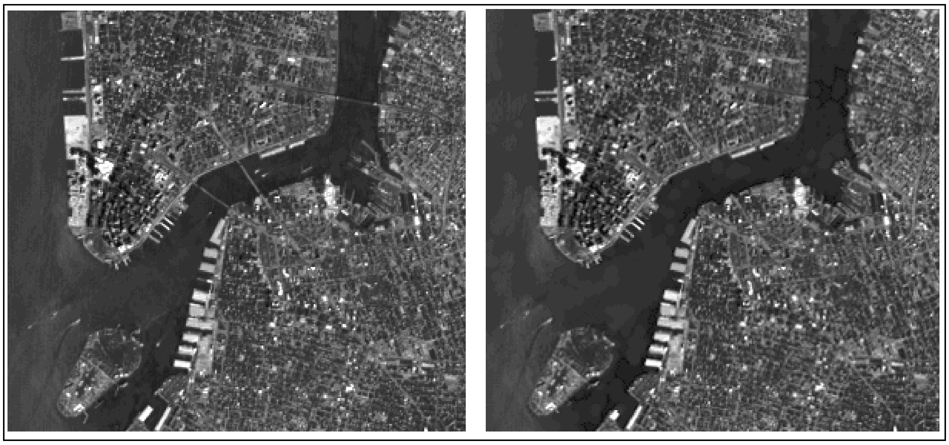


*Figure 11-27: Details of Original (left) and Resulting Image of New York (right)*

## Example Code: Combining Morphological Operations in Feature Extraction

To reproduce the previous example, copy and paste the code into an Editor window. After saving the file as `RemoveBridges.pro`, compile and run the program.

```
PRO RemoveBridges

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

; Read an image of New York using known dimensions.
xsize = 768
ysize = 512
img = READ_BINARY(FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data']), $
   DATA_DIMS = [xsize, ysize])

; Increase image's contrast and display it.
img = BYTSCL(img)
WINDOW, 0
TVSCL, img

; Use a histogram to determine threshold value.
WINDOW, 4, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(img)

; Create an image mask from thresholded image.
maskImg = img LT 70

; Make a square-shaped structuring element.
side = 3
strucElem = DIST(side) LE side

; Remove details in the mask's shape.
maskImg = MORPH_OPEN(maskImg, strucElem)

; Fuse gaps in the mask's shape and display.
maskImg = MORPH_CLOSE(maskImg, strucElem)
WINDOW, 1, title='Mask After Opening and Closing'
TVSCL, maskImg

; Label regions to prepare to remove all but
; the largest region in the mask.
labelImg = LABEL_REGION(maskImg)

; Remove background and all but the largest region.
```

```
regions = labelImg[WHERE(labelImg NE 0)]
mainRegion = WHERE(HISTOGRAM(labelImg) EQ $
   MAX(HISTOGRAM(regions)))
maskImg = labelImg EQ mainRegion[0]

; Display the resulting mask.
Window, 3, TITLE = 'Final Masked Image'
TVSCL, maskImg

; Remove noise and smooth contours in  the original
; image.
newImg = MORPH_OPEN(img, strucElem, /GRAY)

; Replace new image with original image, where not
; masked.
newImg[WHERE(maskImg EQ 0)] = img[WHERE(maskImg EQ 0)]

; View result, comparing the new image with the
; original.
PRINT, 'Hit any key to end program.'
WINDOW, 2, XSIZE = xsize, YSIZE = ysize, $
   TITLE = 'Hit Any Key to End Program'

; Flicker between original and new image.
FLICK, img, newImg

END
```

## Analyzing Image Shapes

After using a morphological operation to expose the basic elements within an image, it is often useful to then extract and analyze specific information about those image elements. The following examples use the LABEL_REGION function and the CONTOUR procedure to identify and extract information about specific image objects.

The LABEL_REGION function labels all of the regions within a binary image, giving each region a unique index number. Use this function in conjunction with the HISTOGRAM function to view the population of each region. See "Using LABEL_REGION to Extract Image Object Information" in the following section for an example.

The CONTOUR procedure draws a contour plot from image data, and allows the selection of image objects occurring at a specific contour level. Further processing using PATH_* keywords returns the location and coordinates of polygons that define a specific contour level. See "Using CONTOUR to Extract Image Object Information" on page 546 for an example.

### Using LABEL_REGION to Extract Image Object Information

The following example identifies unique regions within the image of the *Rhinosporidium seeberi* parasitic protozoans and prints out region populations. For code that you can copy and paste into an Editor window, see "Example Code: Displaying Regions with LABEL_REGION" on page 544 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load a grayscale color table:

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

2. Select and open the image file:

```
file = FILEPATH('r_seeberi.jpg', $
    SUBDIRECTORY = ['examples','data'])
READ_JPEG, file, image, /GRAYSCALE
```

3. Get the image dimensions and add a border (for display purposes only):

```
dims = SIZE(image, /DIMENSIONS)
padImg = REPLICATE(0B, dims[0]+20, dims[1]+20)
padImg[10,10] = image
```

4. Get the dimensions of the padded image, create a window and display the original image:

```
dims = SIZE(padImg, /DIMENSIONS)
WINDOW, 0, XSIZE = 2*dims[0], YSIZE = 2*dims[1], $
    TITLE = 'Opened, Thresholded and Labeled Region Images'
TVSCL, padImg, 0
```

5. Create a large, circular structuring element to extract the large circular foreground features. Define the radius of the structuring element and create the disk:

```
radius = 5
strucElem = SHIFT(DIST(2*radius+1), radius, radius) LE radius
```

**Tip**

Enter `PRINT, strucElem` to view the structure created by the previous statement.

6. Apply the opening operation to the image to remove background noise and display the image:

```
openImg = MORPH_OPEN(padImg, strucElem, /GRAY)
TVSCL, openImg, 1
```

This original image (left) and opened image (right) appear in the following figure.

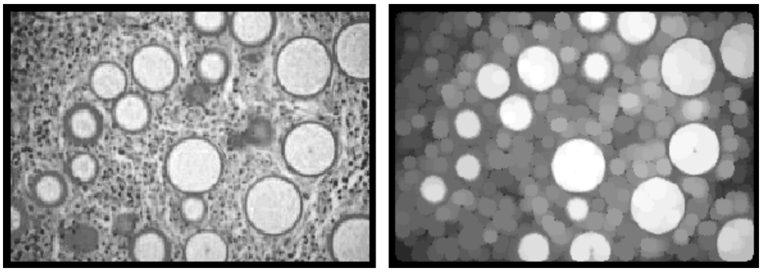


*Figure 11-28: Original Image (left) and Application of Opening Operator (right)*

7. Display an intensity histogram to use as a guide when thresholding:

```
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(openImg)
```

**Note**

Using an intensity histogram as a guide for determining threshold values is described in the section, "Determining Intensity Values When Thresholding and Stretching Images" on page 486.

8. Retain only the brighter, foreground pixels by setting the threshold intensity at 170 and display the binary image:

```
threshImg = openImg GE 170
WSET, 0
TVSCL, threshImg, 2
```

9. Identify unique regions using the LABEL_REGION function:

```
regions = LABEL_REGION(threshImg)
```

10. Use the HISTOGRAM function to calculate the number of elements in each region:

```
hist = HISTOGRAM(regions)
```

11. Create a FOR loop that will return the population and percentage of each foreground region based on the results returned by the HISTOGRAM function:

```
FOR i=1, N_ELEMENTS (hist) - 1 DO PRINT, 'Region', i, $
   ', Pixel Popluation = ', hist(i), $
   '   Percent = ', 100.*FLOAT(hist[i])/(dims[0]*dims[1])
```

12. Load a color table and display the regions. For this example, use the sixteen level color table to more easily distinguish individual regions:

```
LOADCT, 12
TVSCL, regions, 3
```

In the following figure, the image containing the labeled regions (right) shows 19 distinct foreground regions.

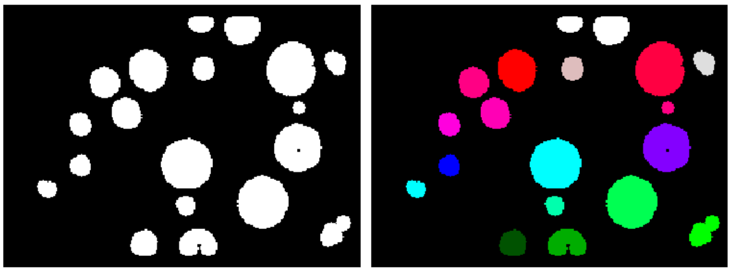


*Figure 11-29: Binary Image (left) and Image of Unique Regions (right)*

**Tip**

Display the color table by entering XLOADCT at the command line. By viewing the color table, you can see that region index values start in the lower-left corner of the image. Realizing this makes it easier to relate the region populations printed in the Output Log with the regions shown in the image.

13. Create a new window and display the individual region populations by graphing the values of *hist* using the SURFACE procedure:

```
WINDOW, 1, $
   TITLE = 'Surface Representation of Region Populations'
FOR i = 1, N_ELEMENTS(hist)-1 DO $
   regions[WHERE(regions EQ i)] = hist[i]
SURFACE, regions
```

The previous command results in the following display of the region populations.

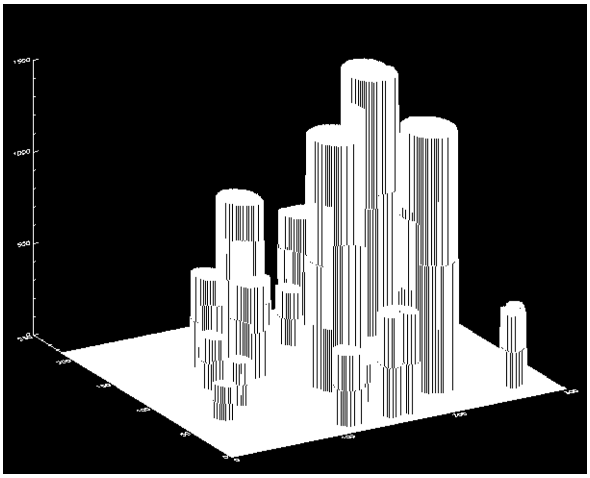


*Figure 11-30: Surface Representation of Region Populations*

## Example Code: Displaying Regions with LABEL_REGION

Copy and paste the following text into the IDL Editor window. After saving the file as `LabelRegionExample.pro`, compile and run the program to reproduce the previous example.

```
PRO LabelRegionExample

; Prepare the display device and load grayscale color
; table.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 0

; Select and open the image file.
file = FILEPATH('r_seeberi.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
READ_JPEG, file, image, /GRAYSCALE

; Get the image dimensions and add a border to the
; image.
dims = SIZE(image, /DIMENSIONS)
padImg = REPLICATE(0B, dims[0]+20, dims[1]+20)
padImg [10,10] = image

; Get the size of the padded image and display it.
dims = SIZE(padImg, /DIMENSIONS)
WINDOW, 0, XSIZE = 2*dims[0], YSIZE = 2*dims[1], $
   TITLE = 'Opened, Thresholded and Labeled Region Images'
TVSCL, padImg, 0

; Define the radius of the structuring element and
; create the disk.
radius = 5
strucElem = SHIFT(DIST(2*radius+1), $
   radius, radius) LE radius

; Apply the opening operator to the image.
openImg = MORPH_OPEN(padImg, strucElem, /GRAY)
TVSCL, openImg, 1

; Determine threshold value using histogram as a guide.
WINDOW, 2, XSIZE = 400, YSIZE = 300
PLOT, HISTOGRAM(openImg)

; Threshold the image to prepare to remove background
; noise.
threshImg = openImg GE 170

; Display the image.
```

```
WSET, 0
TVSCL, threshImg, 2

; Identify regions and print each region's pixel
; population and percentage.
regions = LABEL_REGION(threshImg)
hist = HISTOGRAM(regions)
FOR i=1, N_ELEMENTS (hist) - 1 DO PRINT, 'Region', i, $
   ', Pixel Popluation = ', hist(i), '    Percent = ', $
   100.*FLOAT(hist[i])/(dims[0]*dims[1])

; Load a color table and display the regions.
LOADCT, 12
TVSCL, regions, 3

; Display the pixel population of the regions.
WINDOW, 1, $
   TITLE='Surface Representation of Region Populations'
FOR i=1, N_ELEMENTS(hist)-1 DO $
   regions[WHERE(regions EQ i)] = hist [i]
SURFACE, regions

END
```

## Using CONTOUR to Extract Image Object Information

It is possible to extract information about an image feature using the CONTOUR procedure. The following example illustrates how to select an image feature and return the area of that feature, in this case, calculating the size of a gas pocket in a CT scan of the thoracic cavity.

**Note**

For more information on computing statistics for defined image objects see Chapter 8, “Working with Regions of Interest (ROIs)”

For code that you can copy and paste into an Editor window, see “Example Code: Extracting the Area of a Contour” on page 548 or complete the following steps for a detailed description of the process.

1. Prepare the display device and load a color table:

```
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 5
```

2. Determine the path to the file:

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

3. Initialize the size parameters:

```
dims = [256, 256]
```

4. Import the image from the file:

```
image = READ_BINARY(file, DATA_DIMS = dims)
```

5. Create a window and display the image:

```
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1]
TVSCL, image
```

6. Create another window and use CONTOUR to display a filled contour of the image, specifying 255 contour levels which correspond to the number of values occurring in byte data:

```
WINDOW, 2
CONTOUR, image, /XSTYLE, /YSTYLE, NLEVELS = 255, $
   /FILL
```

**Note**

Replace `NLEVELS = 255` with `NLEVELS = MAX(image)` if your display uses less than 256 colors.

7. Use the PATH_* keywords to obtain information about the contours occurring at level 40:

   ```
   CONTOUR, image, /XSTYLE, /YSTYLE, LEVELS = 40, $
      PATH_INFO = info, PATH_XY = xy, /PATH_DATA_COORDS
   ```

   The PATH_INFO variable, *info*, contains information about the paths of the contours, which when used in conjunction with PATH_XY, traces closed contour paths. Specify PATH_DATA_COORDS when using PATH_XY if you want the contour positions to be measured in data units instead of the default normalized units.

8. Using the coordinate information obtained in the previous step, use the PLOTS procedure to draw the contours of image objects occurring at level 40, using a different line style for each contour:

   ```
   FOR i = 0, (N_ELEMENTS(info) - 1) DO PLOTS, $
      xy[*, info[i].offset:(info[i].offset + info[i].n - 1)], $
      LINESTYLE = (i < 5), /DATA
   ```

9. The specified contour is drawn with a dashed line or LINESTYLE number 2 (determined by looking at “Graphics Keywords” in Appendix B of the *IDL Reference Guide*). Use REFORM to create vectors containing the x and y boundary coordinates of the contour:

   ```
   x = REFORM(xy[0, info[2].offset:(info[2].offset + $
      info[2].n - 1)])
   y = REFORM(xy[1, info[2].offset:(info[2].offset + $
      info[2].n - 1)])
   ```

10. Set the last element of the coordinate vectors equal to the first element to ensure that the contour area is completely enclosed:

    ```
    x = [x, x[0]]
    y = [y, y[0]]
    ```

11. This example obtains information about the left-most gas pocket. For display purposes only, draw an arrow pointing to the region of interest:

    ```
    ARROW, 10, 10, (MIN(x) + MAX(x))/2, COLOR = 180, $
       (MIN(y) + MAX(y))/2, THICK = 2, /DATA
    ```

The gas pocket is indicated with an arrow as shown in the following figure.

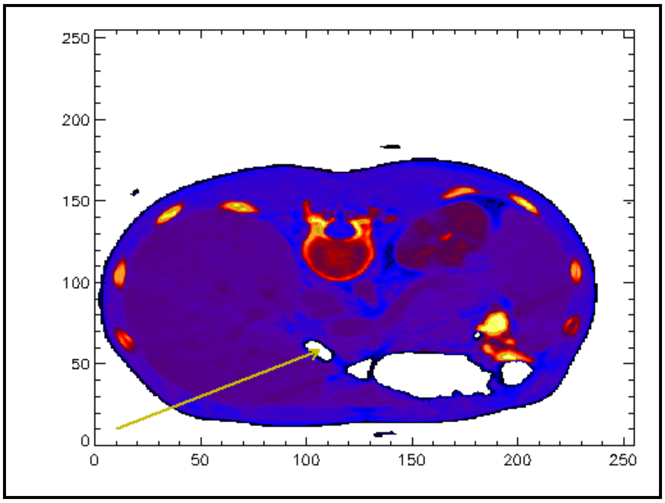


*Figure 11-31: Gas Pocket Indicated in CT Scan of Thoracic Cavity*

12. Output the resulting coordinate vectors, using TRANSPOSE to print vertical lists of the coordinates:

```
PRINT, ''
PRINT, '       x        ,       y'
PRINT, [TRANSPOSE(x), TRANSPOSE(y)], FORMAT = '(2F15.6)'
```

The FORMAT statement tells IDL to format two 15 character floating point values that have 6 characters following the decimal of each value.

13. Use the POLY_AREA function to compute the area of the polygon created by the x and y coordinates and print the result:

```
area = POLY_AREA(x, y)
PRINT, 'area = ', ROUND(area), '  square pixels'
```

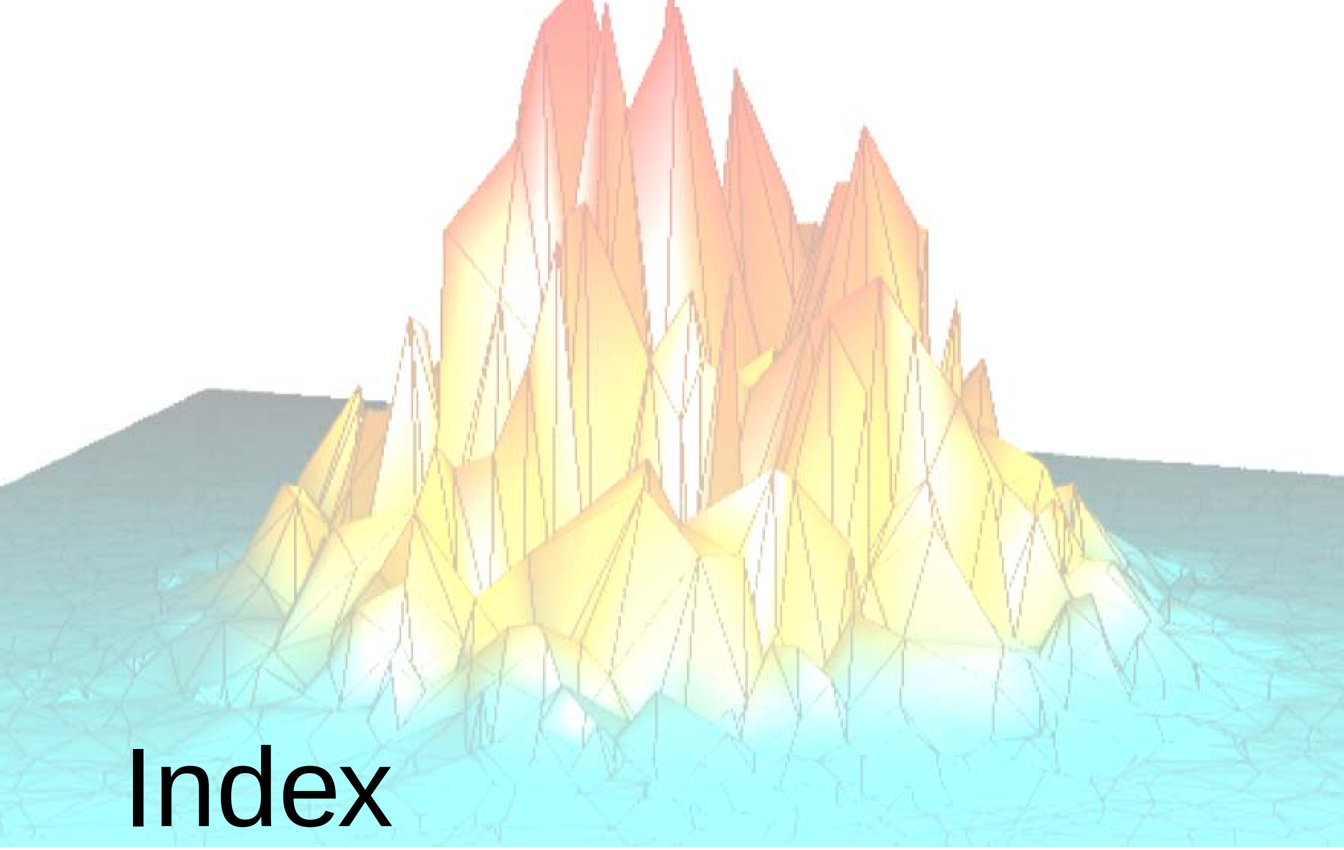
The result, 121 square pixels, appears in the Output Log.

## Example Code: Extracting the Area of a Contour

Copy and paste the following text into the IDL Editor window. After saving the file as `ExtractContourInfo.pro`, compile and run the program to reproduce the previous example.

```
PRO ExtractContourInfo

; Prepare the display device and load a color table.
DEVICE, DECOMPOSED = 0, RETAIN = 2
LOADCT, 5

; Determine the path to the file.
file = FILEPATH('ctscan.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize size parameters.
dims = [256, 256]

; Import the image from the file.
image = READ_BINARY(file, DATA_DIMS = dims)

; Create a window and display the image.
WINDOW, 0, XSIZE = dims[0], YSIZE = dims[1]
TVSCL, image

; Display the filled contour in another window.
WINDOW, 2, TITLE = 'Contour of CT Scan'
CONTOUR, image, /XSTYLE, /YSTYLE, NLEVELS = 255, $
   /FILL

; Use the PATH_* keywords to obtain the vertices (and
; related information) of contour areas occurring at
; level 40.
CONTOUR, image, /XSTYLE, /YSTYLE, LEVELS = 40, $
   PATH_INFO = info, PATH_XY = xy, /PATH_DATA_COORDS

; Plot the level 40 contours over the filled contour
; display.  Use different linestyles for each closed
; contour at level 40.
FOR i = 0, (N_ELEMENTS(info) - 1) DO PLOTS, $
   xy[*, info[i].offset:(info[i].offset + info[i].n - 1)], $
   LINESTYLE = (i < 5), /DATA

; From the previous display, we determined the gas
; pocket we are interested in is the third closed
; contour at level 40, with the number 2, dashed line
; style. Obtain the x and y coordinates for this closed
; contour.
x = REFORM(xy[0, info[2].offset:(info[2].offset + $
   info[2].n - 1)])
y = REFORM(xy[1, info[2].offset:(info[2].offset + $
   info[2].n - 1)])
HELP, (xy[0, info[2].offset:(info[2].offset +$
   info[2].n - 1)])
```

```
PRINT, (xy[1, info[2].offset:(info[2].offset + $
   info[2].n - 1)])

; Set the last element of the coordinate vectors to the
; first element to ensure that the contour area is
; completely enclosed.
x = [x, x[0]]
y = [y, y[0]]

; Draw an arrow pointing to the region of interest for
; display purposes only.
ARROW, 10, 10, (MIN(x) + MAX(x))/2, COLOR = 180, $
   (MIN(y) + MAX(y))/2, THICK = 2, /DATA

; Output the resulting vectors.
PRINT, ''
PRINT, '       x       ,       y'
PRINT, [TRANSPOSE(x), TRANSPOSE(y)], FORMAT = '(2F15.6)'

; Compute area of gas pocket and output results.
area = POLY_AREA(x, y)
PRINT, 'area = ', ROUND(area), '  square pixels'

END
```



# Index