IDL

Image Processing in IDL

RSI

IDL Version 6.2
July 2005 Edition

0705IDL62IP

Acknowledgments

IDL® is a registered trademark and ION™, ION Script™, ION Java™, are trademarks of ITT Industries, 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.

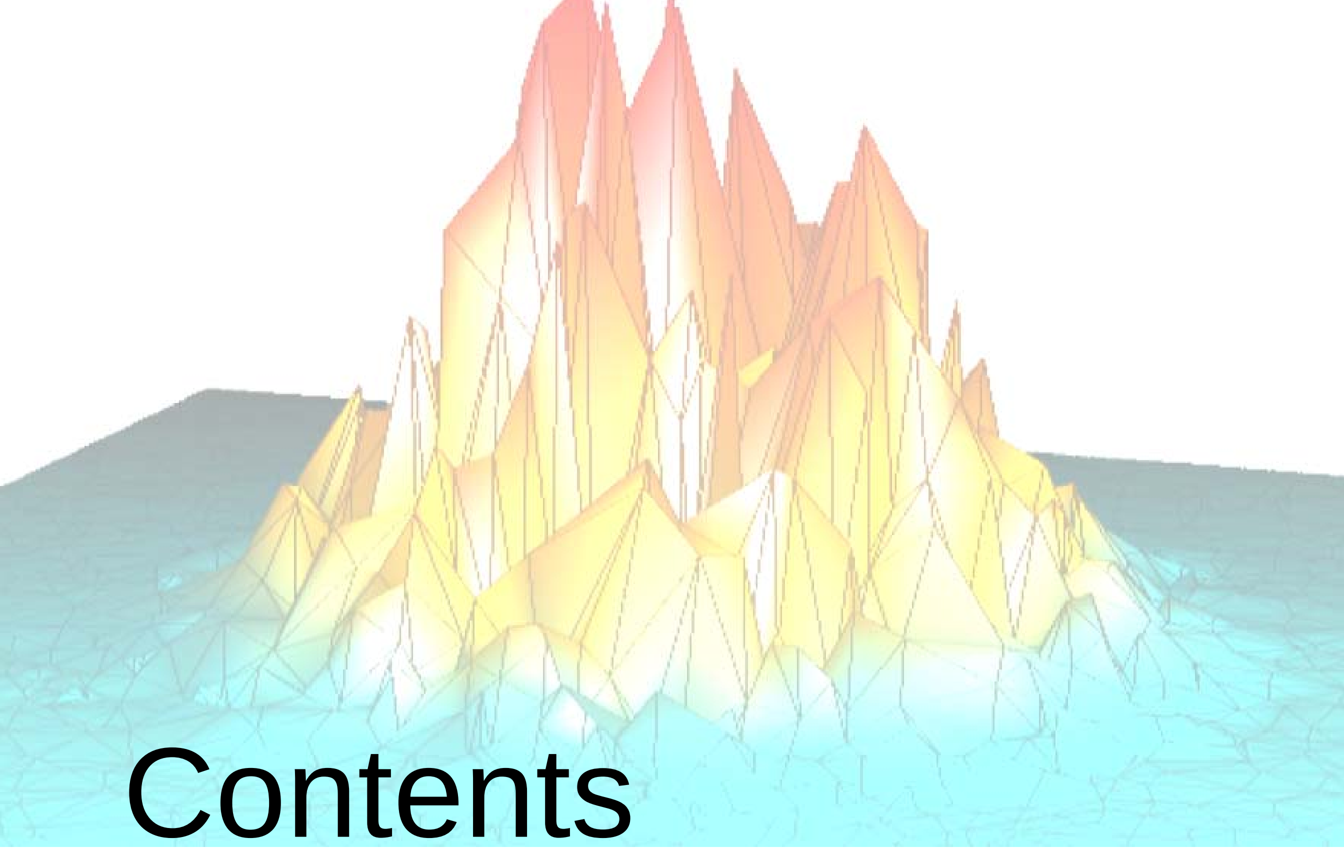
Portions of this computer program are copyright © 1995-1999 LizardTech, Inc. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

Portions of this software are copyrighted by Merge Technologies Incorporated.

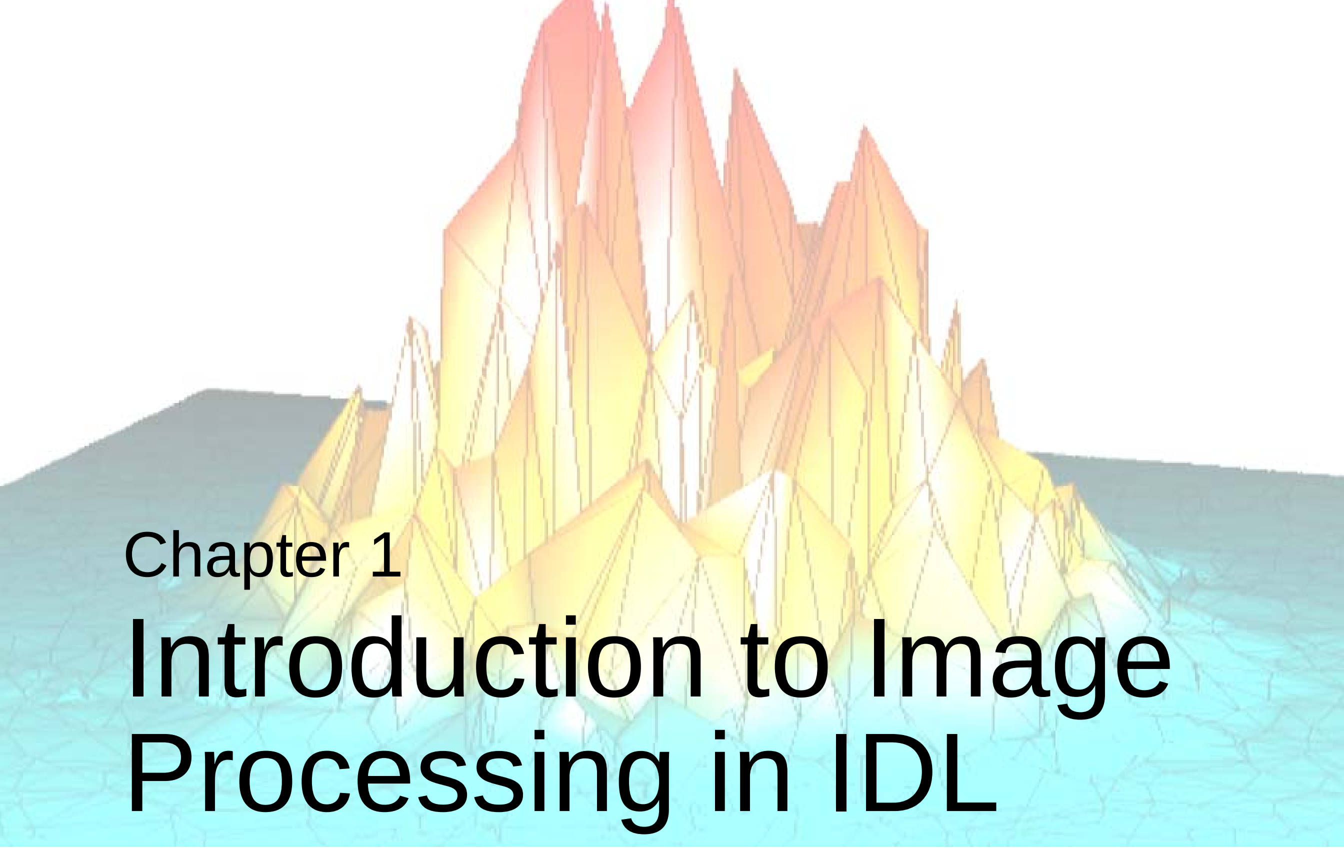
This product includes software developed by the Apache Software Foundation (http://www.apache.org/)

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:

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 2, "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 3, "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 4, “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 5, “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 6, “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 7, “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 8, “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 9, “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 12
- “Representing Image Data in IDL” on page 13
- “Accessing Images” on page 15

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 "Color Systems" in Chapter 8 of the *Using IDL* manual.

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 the examples section of “REFORM” in the *IDL Reference Guide* manual and “Calculating Image Statistics” in Chapter 4 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

Table 1-2: Some IDL Data Type Conversion Functions

Function	Description
FIX	Convert to 16-bit integer, or optionally other type
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 image file formats listed in "Supported File Formats" in Chapter 1 of the *Using IDL* manual.

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.

See "Importing and Writing Data into Variables" in Chapter 6 of the *Using IDL* manual for details on data access in IDL. This chapter and the *IDL Reference Guide* provide details on the file access routines used in examples in the following chapters.

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 information on using QUERY_IMAGE, see "Returning Image File Information" in Chapter 7 of the *Using IDL* manual.

References

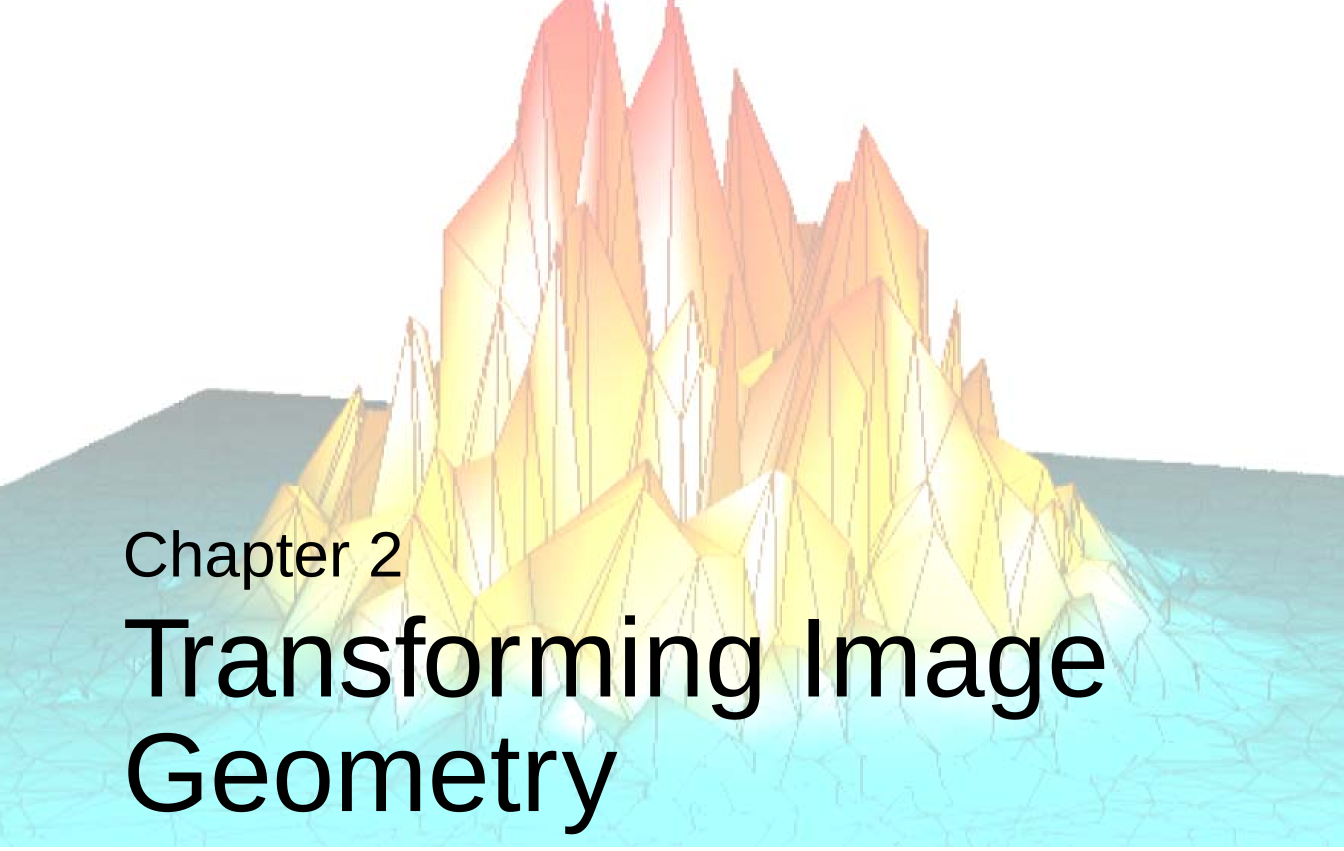
The following image processing sources were used in writing this book:

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
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 5, "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" in Chapter 8 of the *Using IDL* manual 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 20.	SIZE CURSOR	Focuses attention on important image features by creating a rectangular region of interest.
"Padding Images" on page 23.	SIZE	Creates a border around the perimeter of an image for presentation or advanced filtering purposes.
"Resizing Images" on page 26.	CONGRID REBIN	Enlarges or shrinks an image.

Table 2-1: Image Processing Tasks and Related Image Processing Routines

Task	Routine(s)	Description
"Shifting Images" on page 28.	SHIFT	Shifts image pixel values along any image dimension.
"Reversing Images" on page 30.	REVERSE	Reverses array elements to flip an image horizontally or vertically.
"Transposing Images" on page 32.	TRANSPOSE	Interchanges array dimensions, reflecting the image about a 45 degree line.
"Rotating Images" on page 34.	ROTATE ROT	Rotates an image to any orientation, using 90 degree or arbitrary increments.
"Planar Slicing of Volumetric Data" on page 38.	EXTRACT_SLICE SLICER3 XVOLUME	Displays a single slice or a series of planar slices in a single window or interactively extracts planar slices of volumetric data.

Table 2-1: Image Processing Tasks and Related Image Processing 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.

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 26.

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

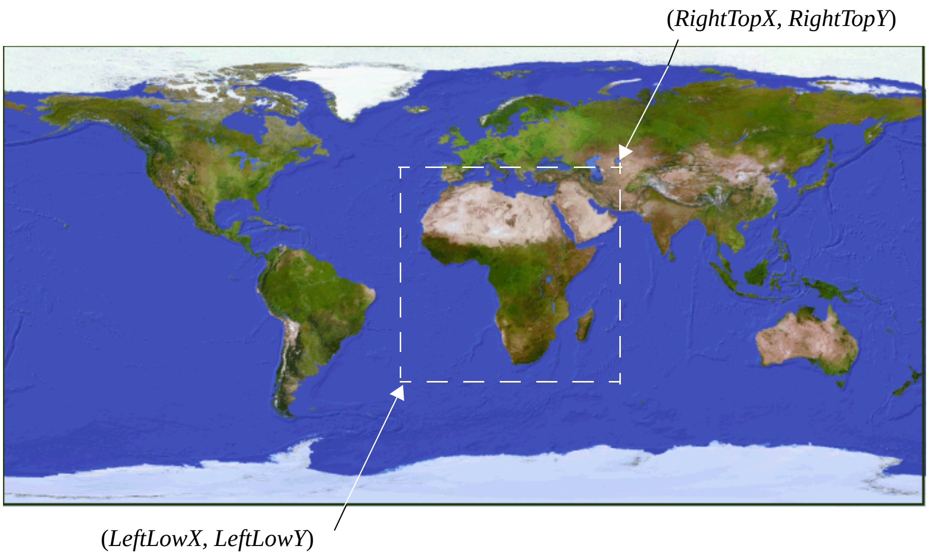


Figure 2-1: Defining the Boundaries of the Cropped Image Array

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 `CropWorld.pro` in the `examples/doc/image` subdirectory of the IDL installation directory 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.

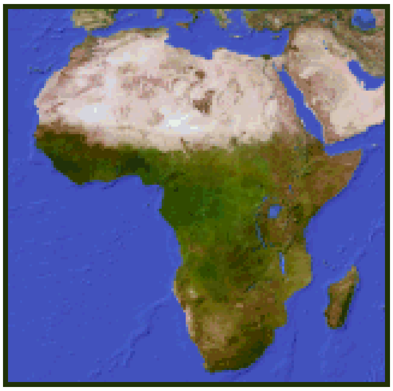


Figure 2-2: Result of the Cropped Image Example

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.

Example Code

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

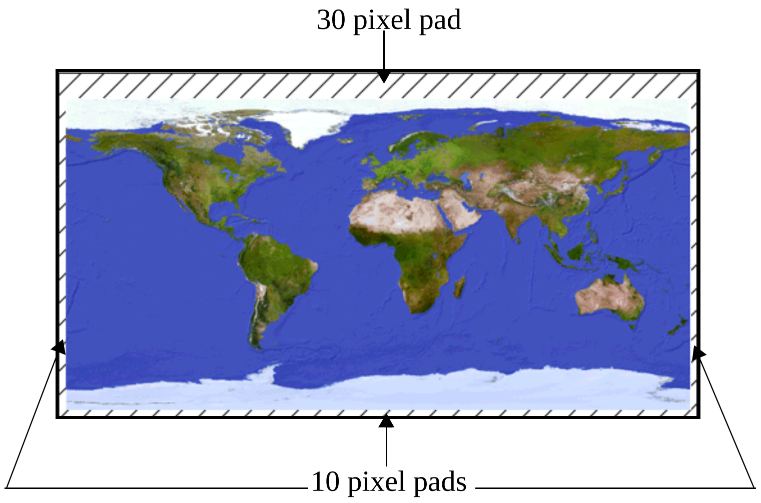


Figure 2-3: Diagonal Lines Indicate Padding

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 "Graphic Display Essentials" in Chapter 8 of the *Using IDL* manual 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.

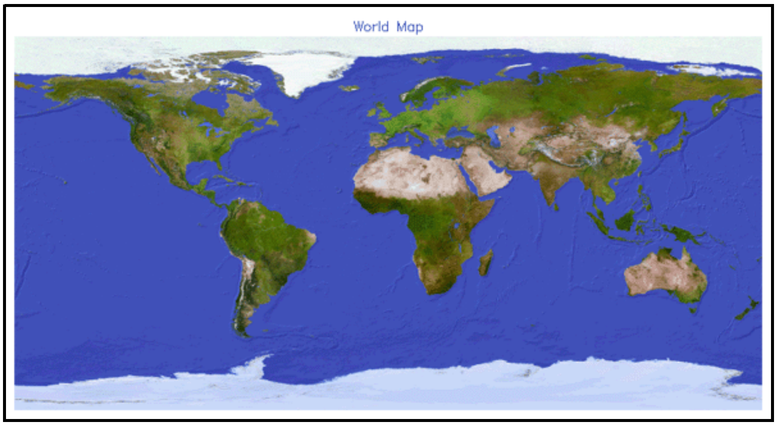


Figure 2-4: Resulting Padded Image

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.

Example Code

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

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).

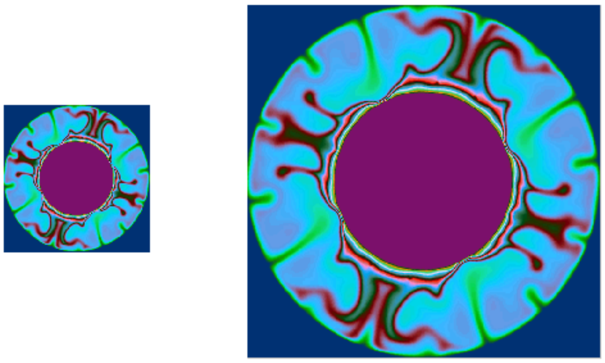


Figure 2-5: Original Image and Magnified Image

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.

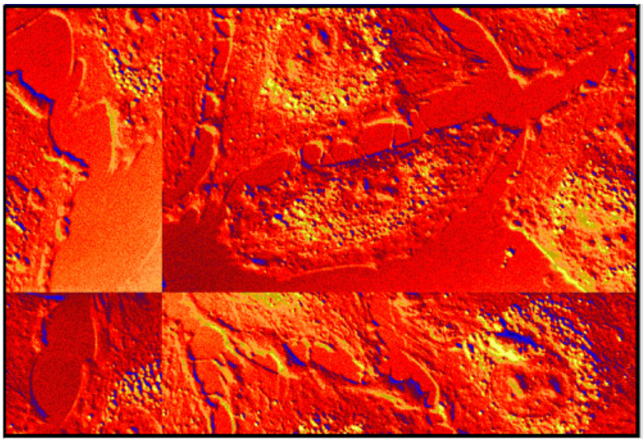


Figure 2-6: Example of Misaligned Image Array Elements

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.

Example Code

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

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.

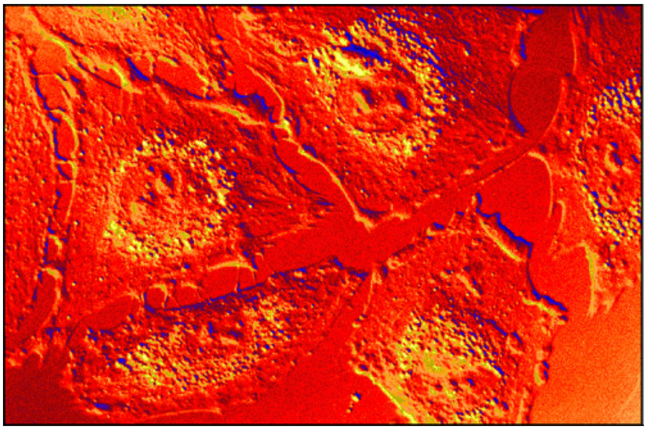


Figure 2-7: Resulting Shifted Array

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.

Example Code

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

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.

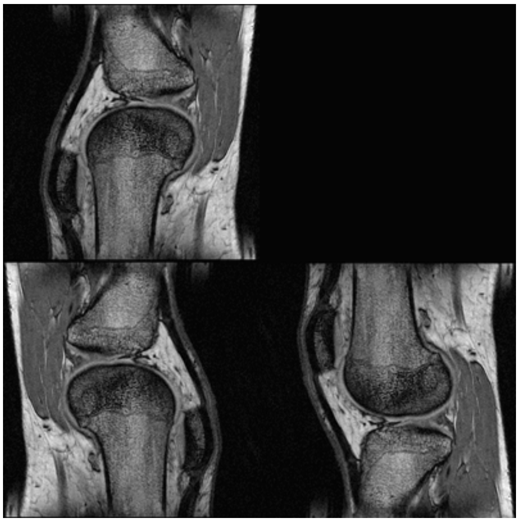


Figure 2-8: Original Image (Top); Reversed Dimension 1 (Bottom Left); and Reversed Dimension 2 (Bottom Right)

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.

Example Code

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

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).

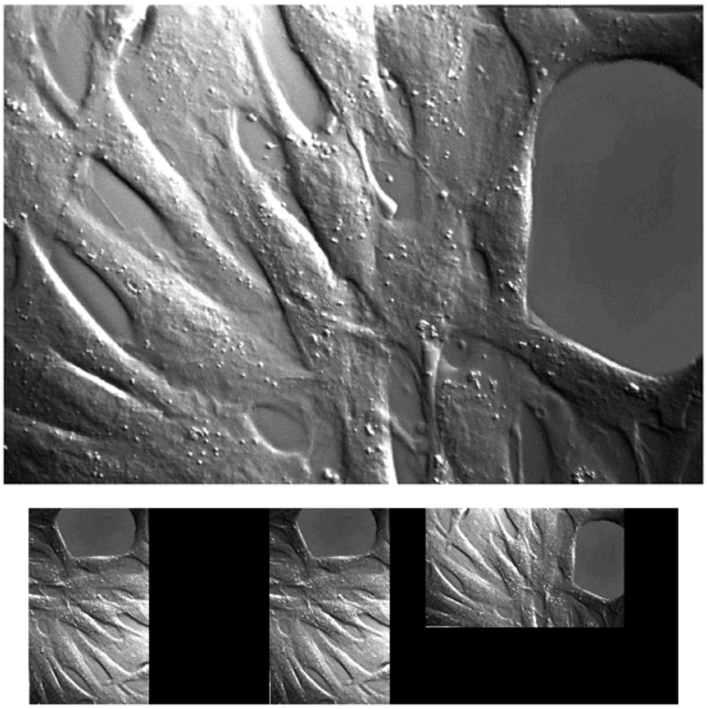


Figure 2-9: Original (Top) and Transposed Images (Bottom) from Left to Right, transposeImg1, transposeImg2, and transposeImg3

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 36.

Rotating an Image by 90 Degree Increments

The following example changes the orientation of an image by rotating it 270°.

Example Code

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

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 35 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).

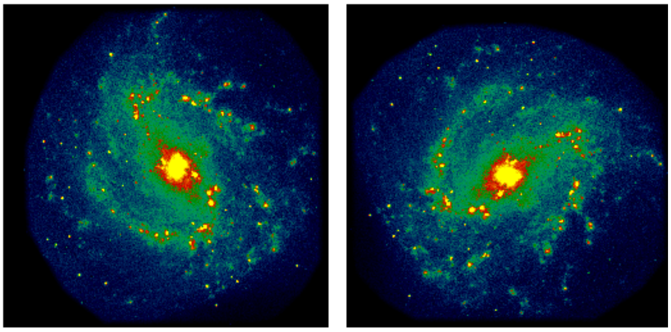


Figure 2-10: Using ROTATE to Alter Image Orientation

ROTATE *Direction* Argument Options

The following table describes the *Direction* options available with the ROTATE function syntax, *Result* = ROTATE (*Array*, *Direction*).

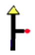
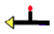
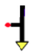
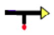
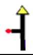
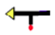
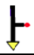
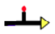
Direction	Transpose?	Rotation Counterclockwise	Sample Image
0	No	None	
1	No	90°	
2	No	180°	
3	No	270°	
4	Yes	None	
5	Yes	90°	
6	Yes	180°	
7	Yes	270°	

Table 2-2: Direction Options Available with ROTATE

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.

Example Code

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

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.

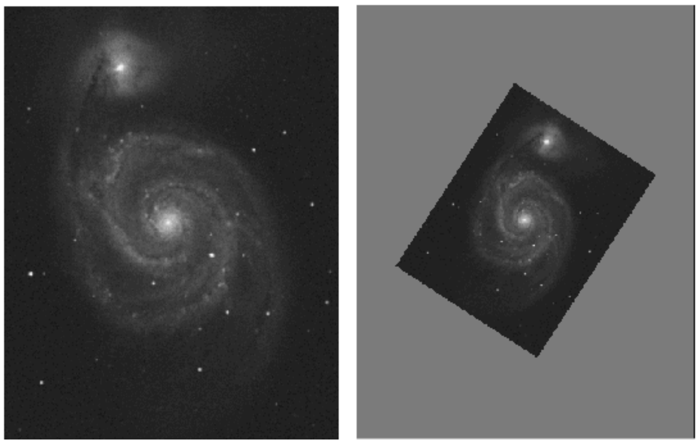


Figure 2-11: The Original Image (Left) and Modified Image (Right)

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.

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 40
- “Interactive Planar Slicing of Volumetric Data” on page 41

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.

Example Code

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

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.

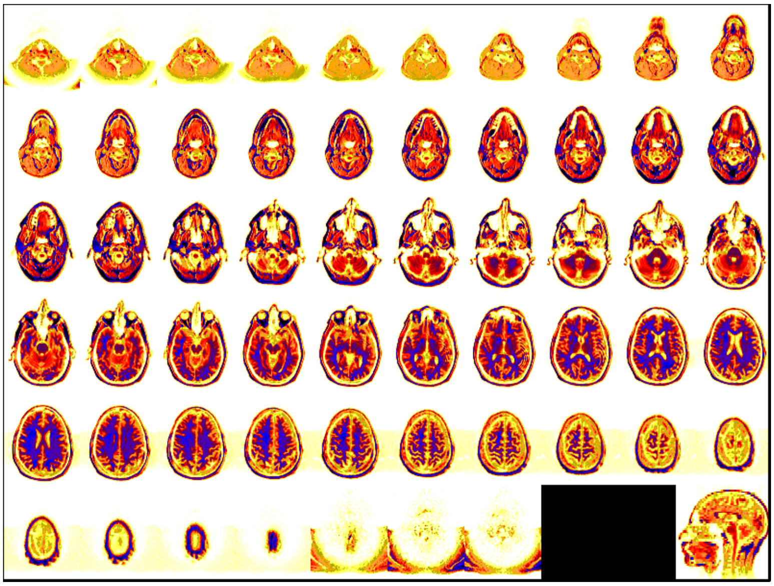


Figure 2-12: Planar Slices of a MRI Scan of a Human Head

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" below for more information.

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.

Example Code

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

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.

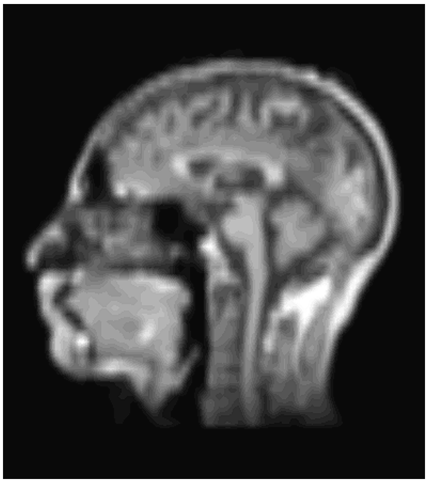


Figure 2-13: Example of Extracting a Slice of Data From a Volume

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 38, 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.

Tip

For more information and examples of displaying volumes and slicing volumetric data using XVOLUME, see "XVOLUME" in the *IDL Reference Guide* manual.

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.

Example Code

See `displayslicer3.pro` in the `examples/doc/image` subdirectory of the IDL installation directory for code that duplicates this example.

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.

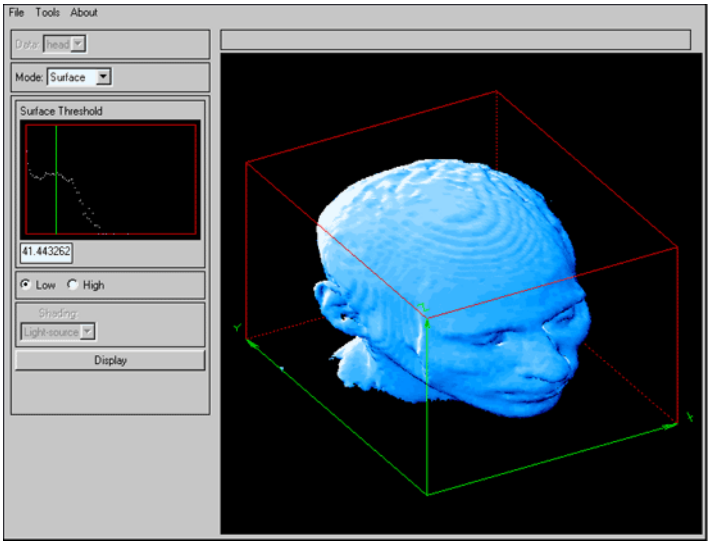
```
SLICER3, pData, DATA_NAMES ='head'
```

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.



41.443262

Figure 2-14: 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.

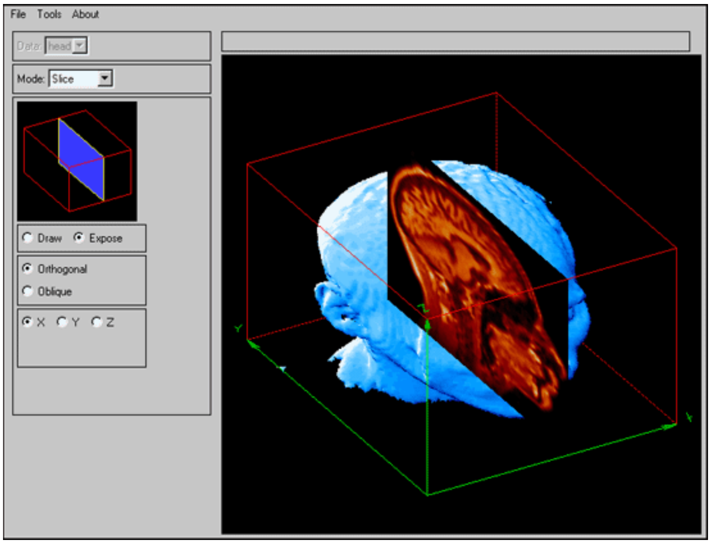


Figure 2-15: Visualizing a Slice of Volumetric Data

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.

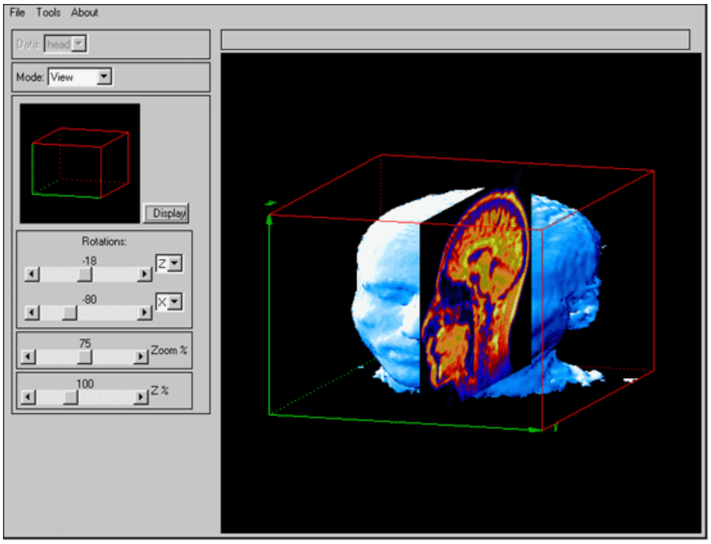
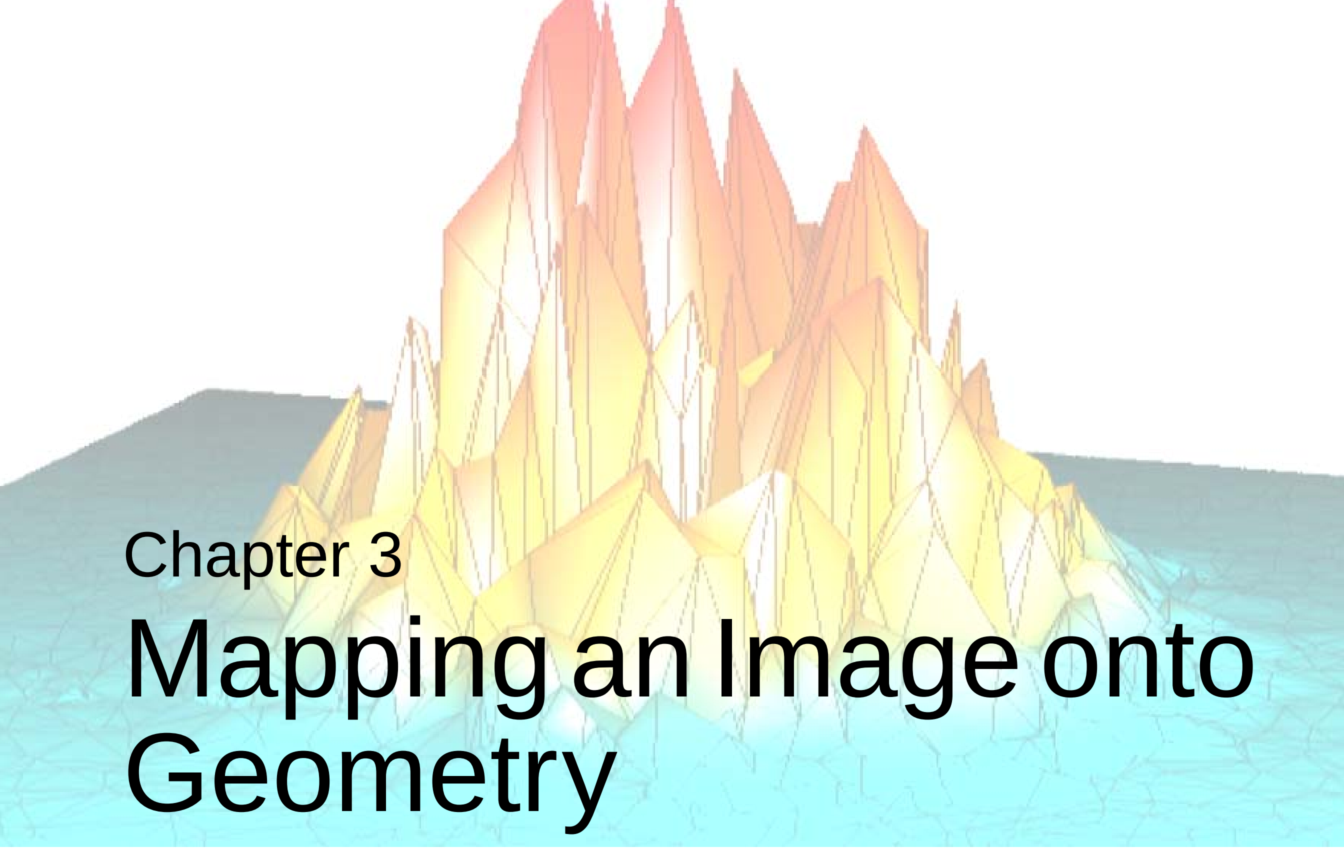


Figure 2-16: A Slice Overlaying an Isosurface

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`



Chapter 3

Mapping an Image onto Geometry

This chapter describes the following topics:

Mapping Images onto Surfaces Overview

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 50.	SHADE_SURF	Display the elevation data.
	IDLgrWindow::Init IDLgrView::Init 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 3-1: 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 57.	MESH_OBJ REPLICATE	Create a sphere.
	SCALE3	Specify system variables required for 3D viewing.
	SET_SHADING	Control the light source used by POLYSHADE.
	TVSCL 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 60.	MESH_OBJ REPLICATE	Create a sphere.
	IDLgrModel::Init IDLgrPalette::Init IDLgrImage::Init	Initialize model, palette and image objects.
	FINDGEN 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 3-1: 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 52
- "Displaying the Image and Geometric Surface Objects" on page 53

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.

Complete the following steps for a detailed description of the process.

Example Code

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

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
```

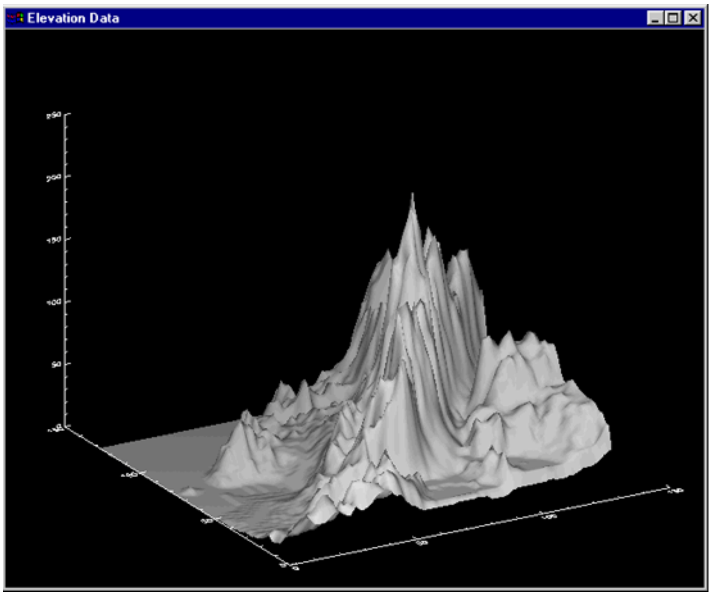


Figure 3-1: Visual Display of the Elevation Data

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.

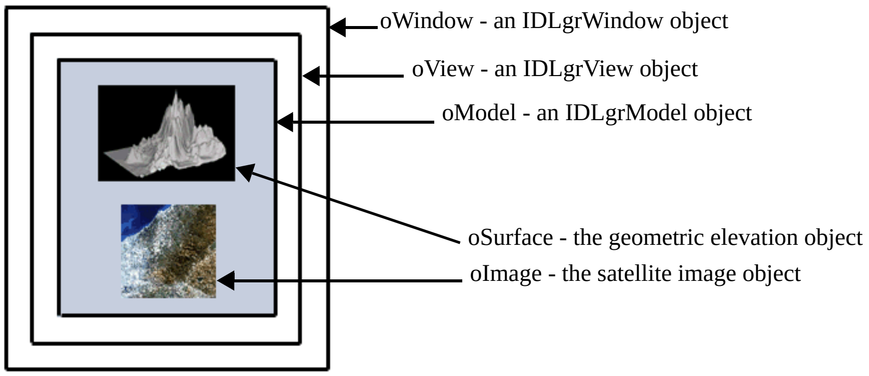

oWindow - an IDLgrWindow object
oView - an IDLgrView object
oModel - an IDLgrModel object
oSurface - the geometric elevation object
oImage - the satellite image object

Figure 3-2: Conceptualization of Object Graphics Display Example

Note

The XOBJVIEW utility (described in "Mapping an Image onto a Sphere Using Object Graphics" on page 60) 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
   ```

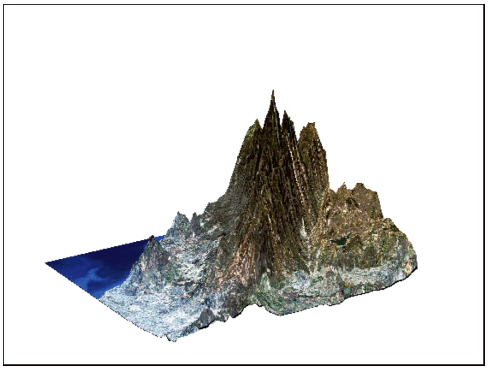


Figure 3-3: Image Mapped onto a Surface in an Object Graphics Window

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.

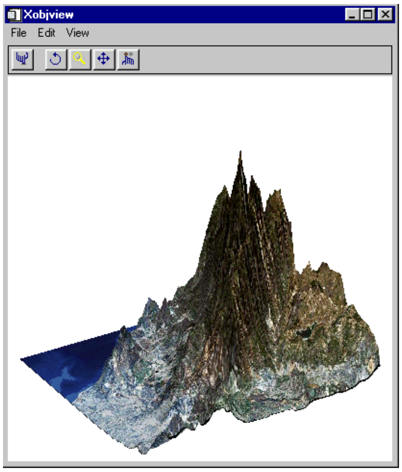


Figure 3-4: Displaying the Image Mapped onto the Surface in XOBJVIEW

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 57.

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 60

Mapping an Image onto a Sphere Using Direct Graphics

Complete the following steps for a detailed description of the process.

Example Code

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

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.

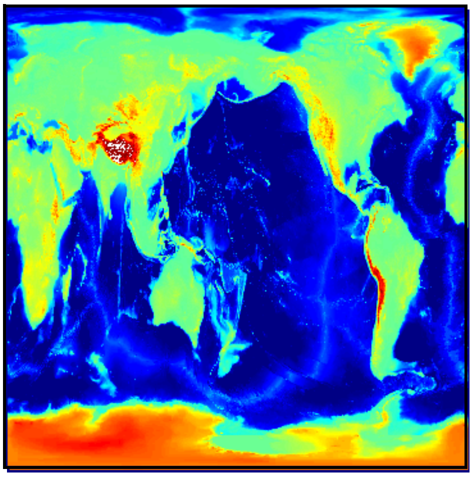


Figure 3-5: World Elevation Image

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.

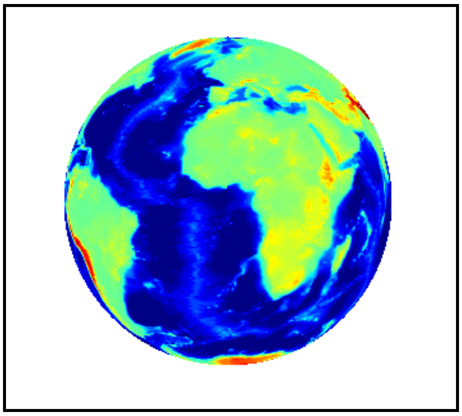


Figure 3-6: Direct Graphics Display of an Image Mapped onto a Sphere

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" below.

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 52. 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.

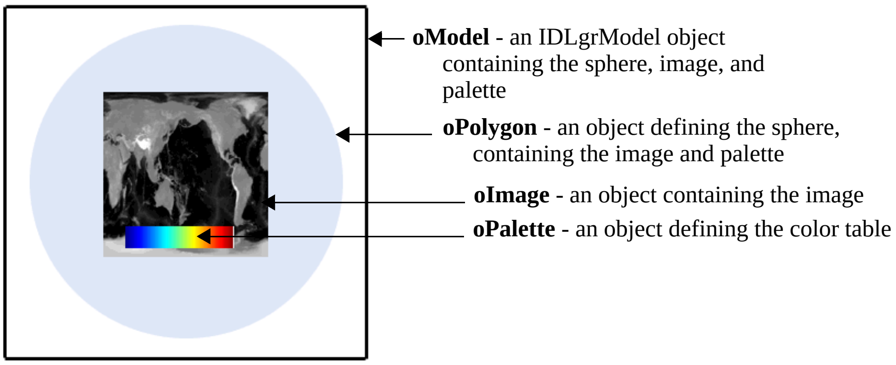

oModel - an IDLgrModel object containing the sphere, image, and palette
oPolygon - an object defining the sphere, containing the image and palette
oImage - an object containing the image
oPalette - an object defining the color table

Figure 3-7: Conceptualization of XOBJVIEW Object Graphics Example

Complete the following steps for a detailed description of the process.

Example Code

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

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.

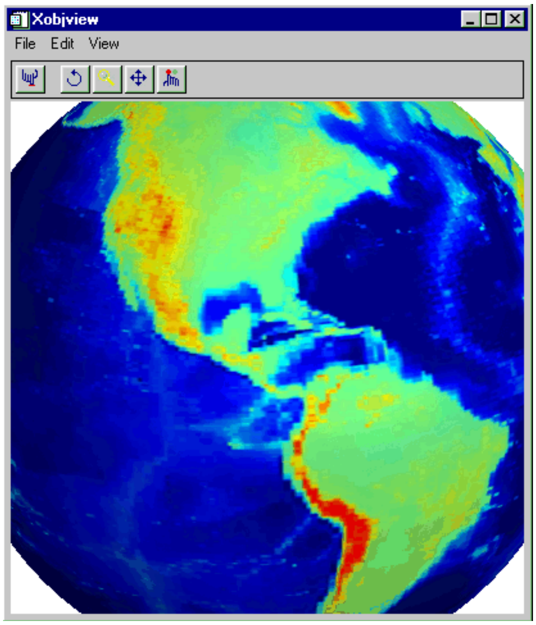
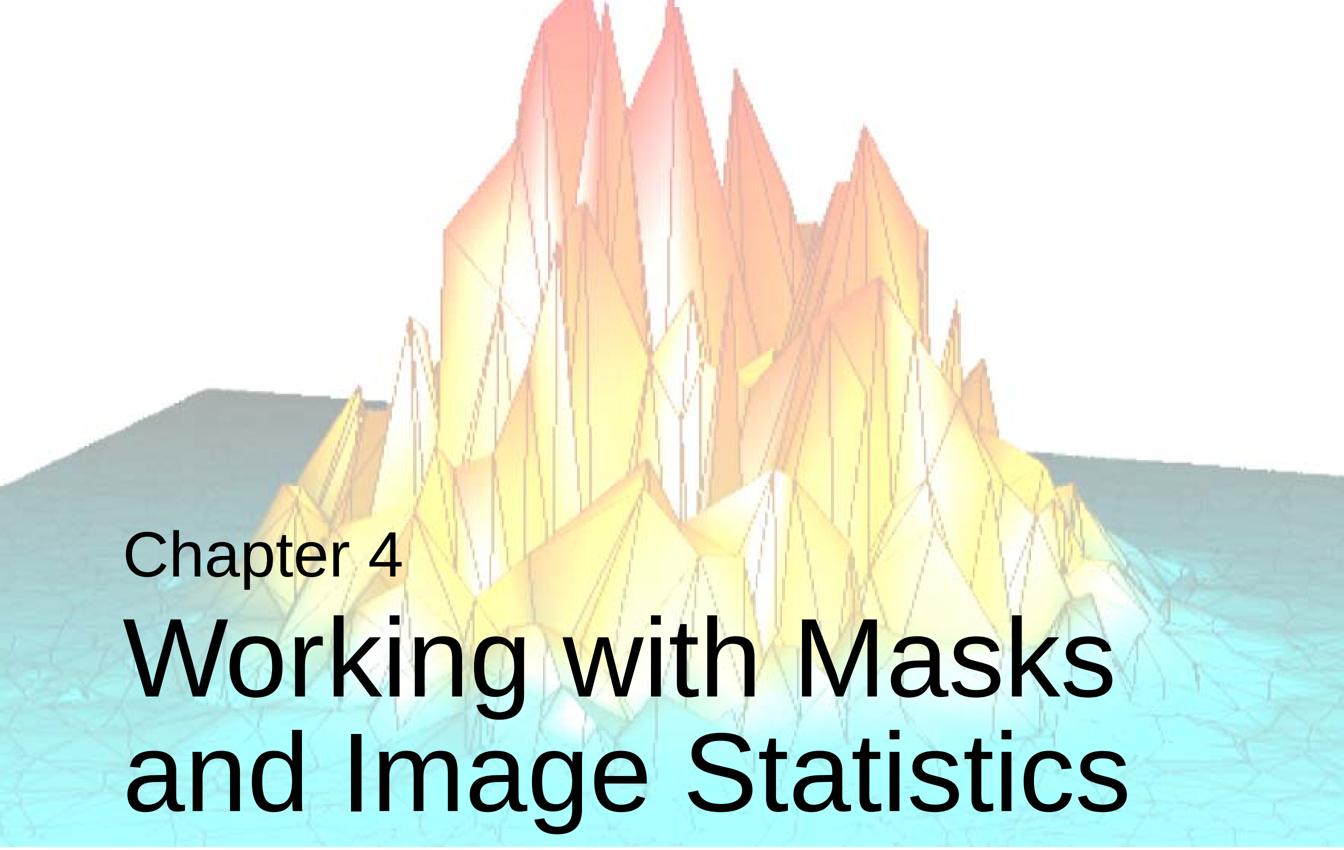


Figure 3-8: Magnified View of World Elevation Object

9. After closing the XOBJVIEW display, remove unneeded object references:

```
OBJ_DESTROY, [oModel, oImage, oPalette]
```



Chapter 4
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 68.

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 72.

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 76.

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 80.

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 68.	Relational Operators Mathematical Operators	Make masks and apply them to images.

Table 4-1: Image Math Tasks and Related Image Math Operators and Routines

Task	Operator(s) and Routine(s)	Description
"Clipping Images" on page 72.	Minimum and Maximum Operators Mathematical Operators	Clip the pixel values of an image to highlight specific features.
"Locating Pixel Values in an Image" on page 76.	WHERE Mathematical Operators	Locate specific pixel values within an image.
"Calculating Image Statistics" on page 80	Mathematical Operators IMAGE_STATISTICS	Calculate the sum, mean, standard deviation, and variance of the pixel values within an image.

Table 4-1: 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 4-2: 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

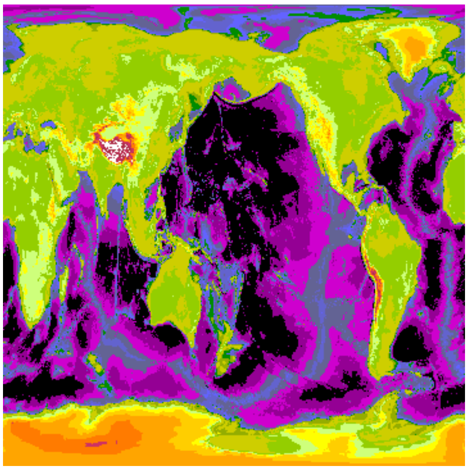


Figure 4-1: World Elevation Image

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.

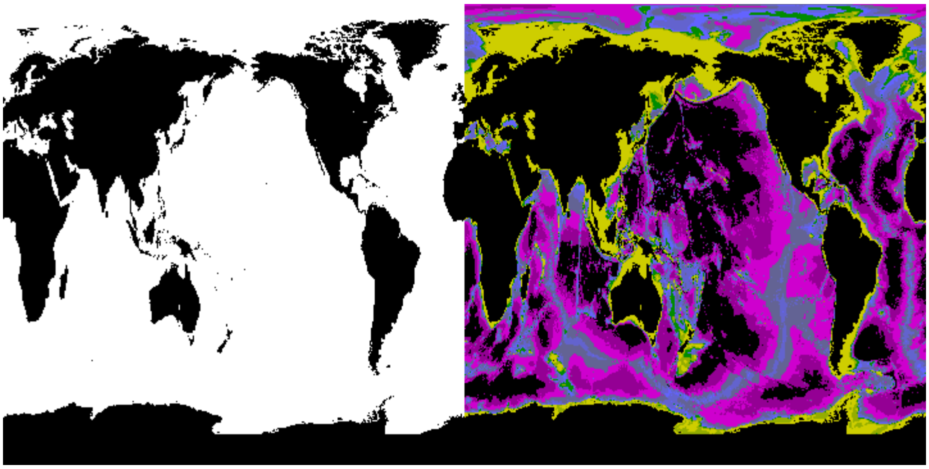


Figure 4-2: Oceans Mask (left) and the Resulting Image (right)

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.

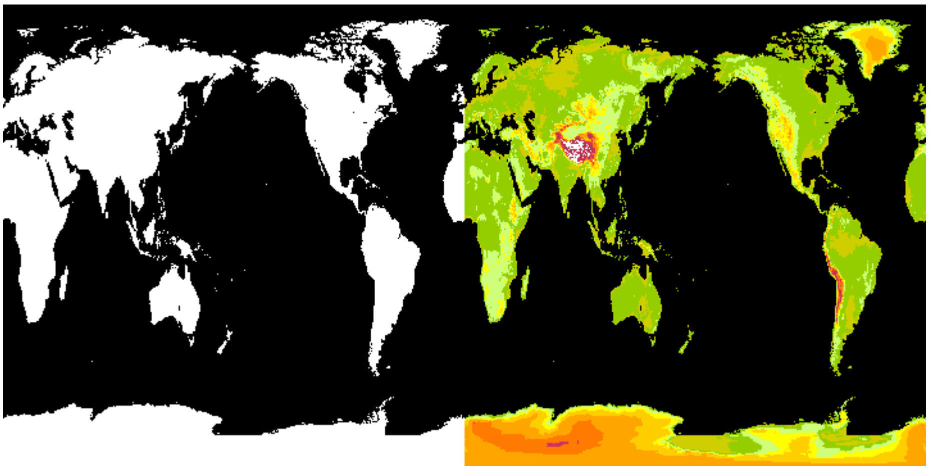


Figure 4-3: Land Mask (left) and the Resulting Image (right)

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 4-3: 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 for Threshold and Stretch" in Chapter 9 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

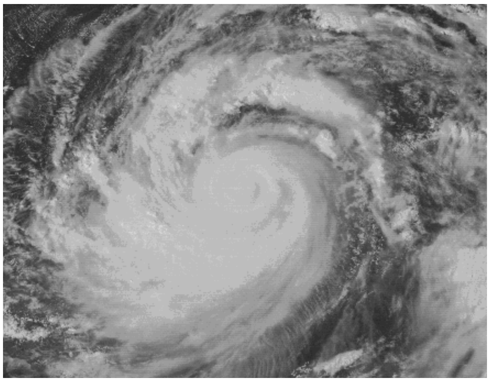


Figure 4-4: Image of Hurricane Gilbert

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.

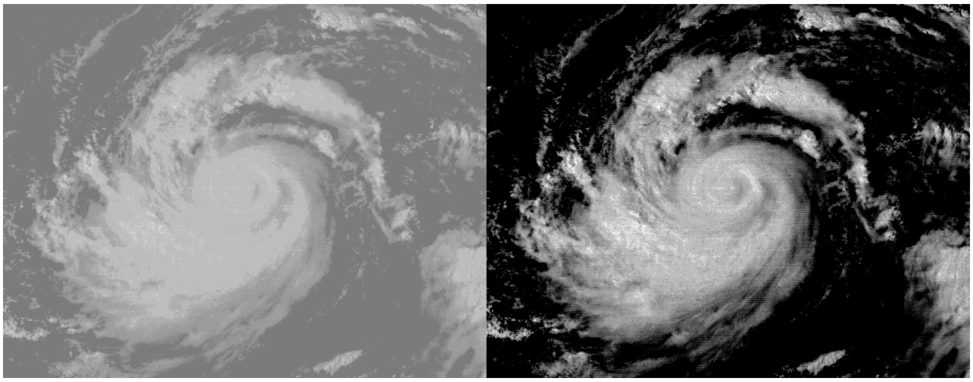


Figure 4-5: Pixel Values Greater Than 125, TV (left) and TVSCL (right)

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.

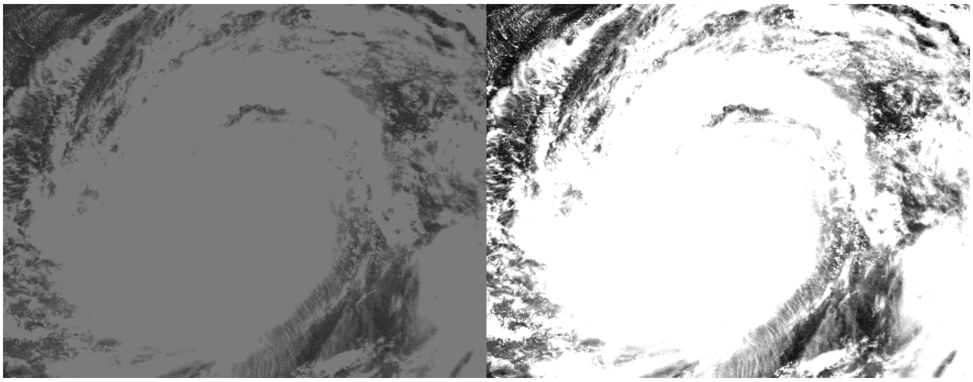


Figure 4-6: Pixel Values Less Than 125, TV (left) and TVSCL (right)

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 68 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

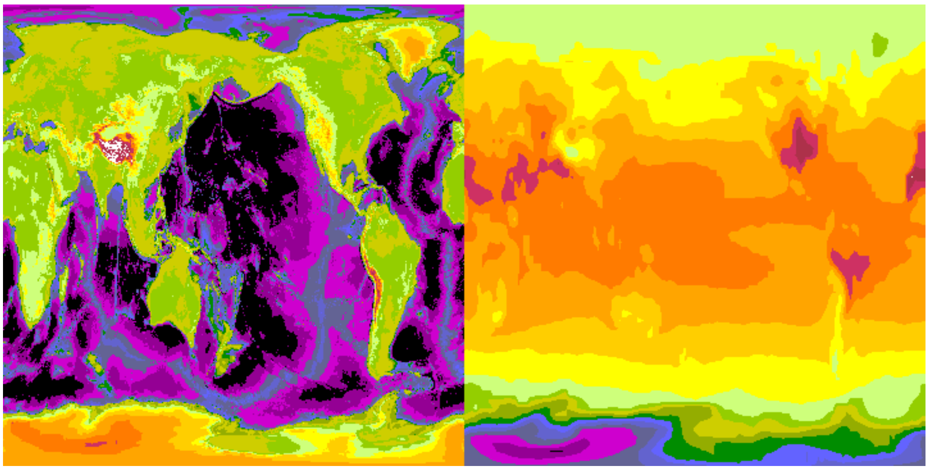


Figure 4-7: World Elevation (left) and Temperature (right)

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.

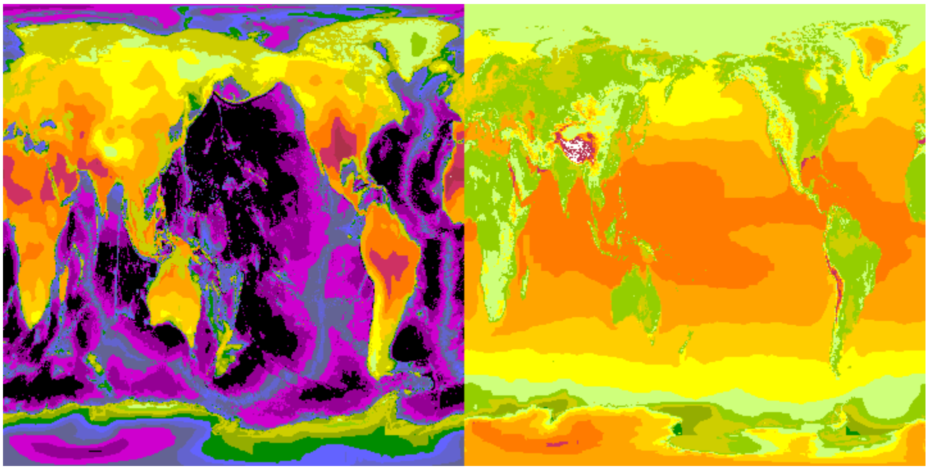


Figure 4-8: Temperature Over Land (left) and Over Oceans (right)

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.

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

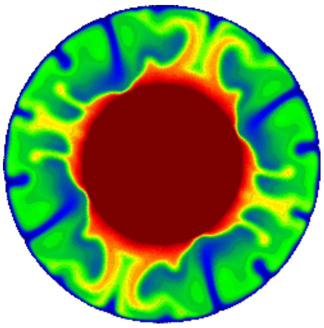


Figure 4-9: Earth Mantle Convection

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.

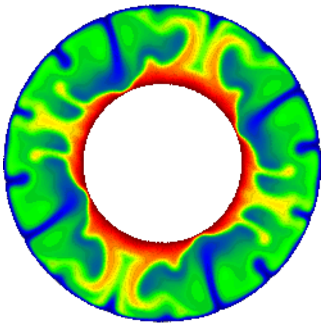


Figure 4-10: The Difference of the Original Image and the Core

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
```

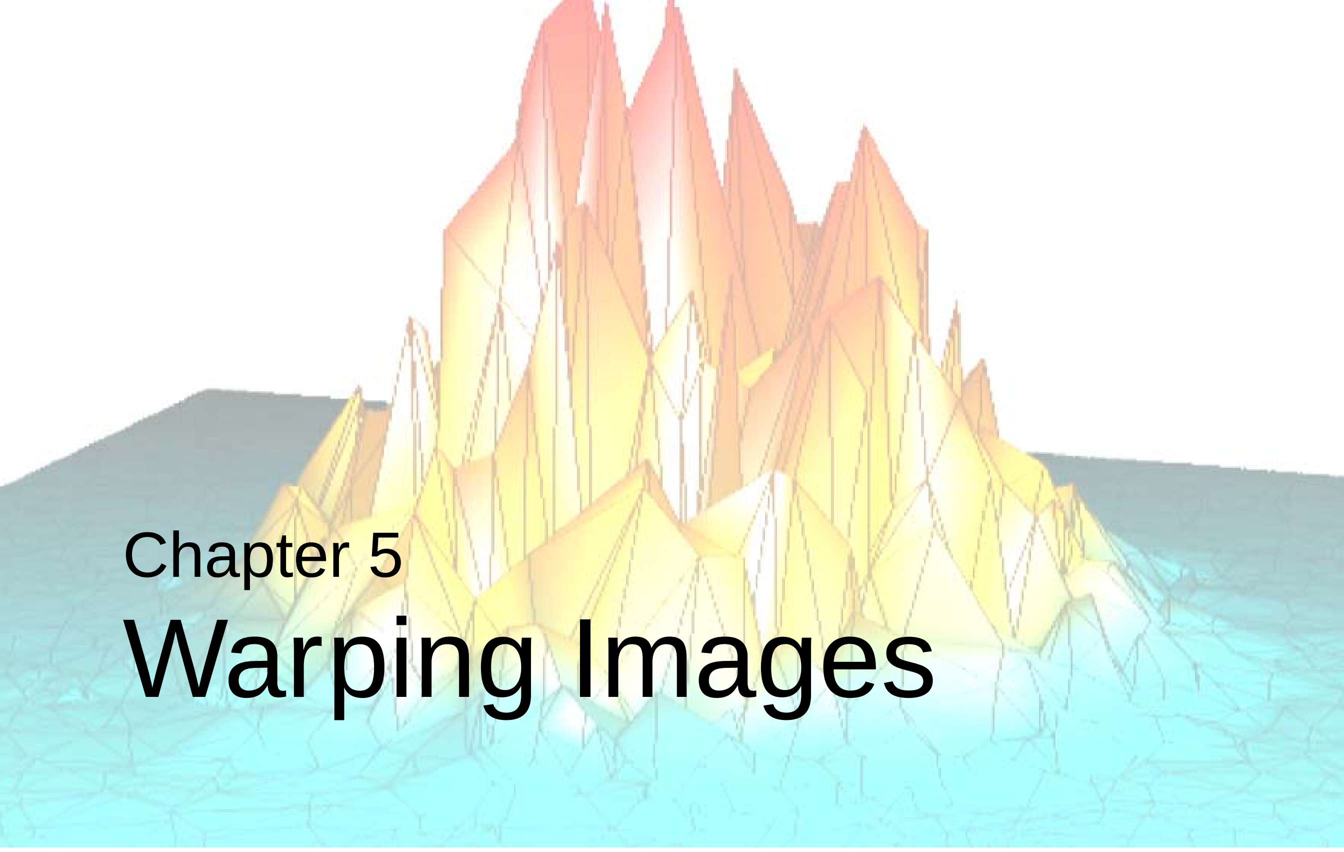
13. Print out the resulting statistics:

```
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.



Chapter 5

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 87.

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 88.

The following table introduces the tasks and routines covered in this chapter.

Task	Routine	Description
Creating a Direct Graphics Display of Image Warping See "Warping Images Using Direct Graphics" on page 89.	WSET 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 5-1: Image Warping Tasks and Routines

Task	Routine	Description
Creating an Object Graphics Display of Image Warping See “Warping Image Objects” in Chapter 4 of the *Object Programming* manual.	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 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 IDLgrView::Init IDLgrModel::Init	Initialize the objects necessary for an Object Graphics display.

Table 5-1: 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 89.

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 object 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. For an example, see "Defining Transparency in Image Objects" and "Warping Image Objects" in Chapter 4 of the *Object Programming* manual.

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

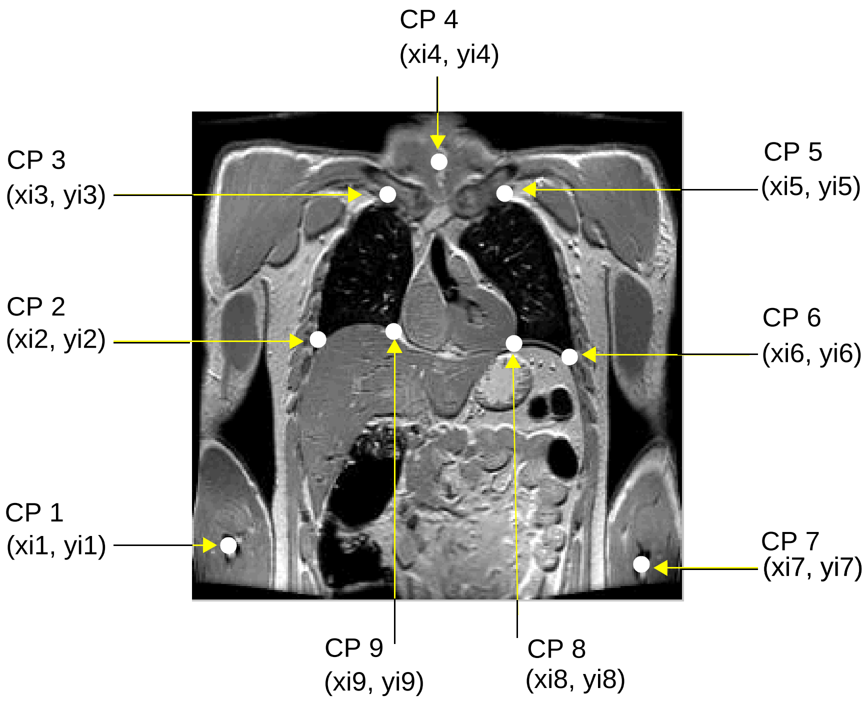


Figure 5-1: Control Points (CP) Selection in the Input Image

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.

Note

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.

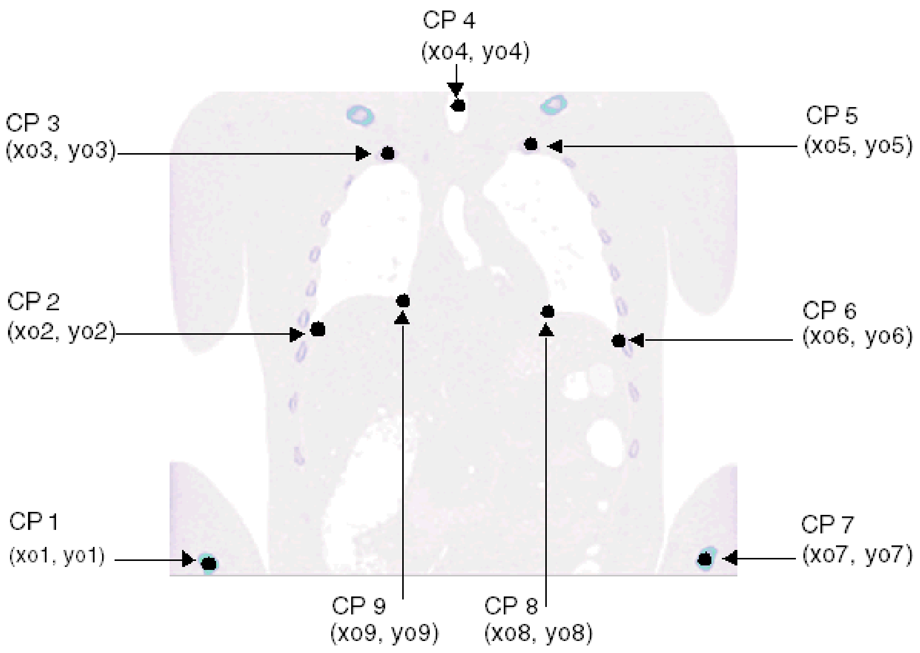


Figure 5-2: Control Point (CP) Selection in the Reference Image

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.

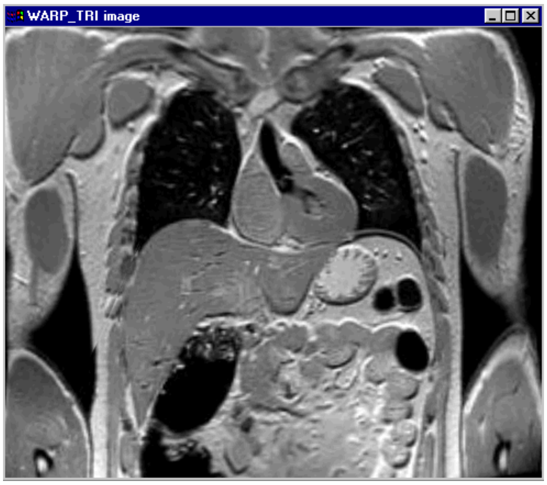


Figure 5-3: Warped Image Produced with WARP_TRI

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.

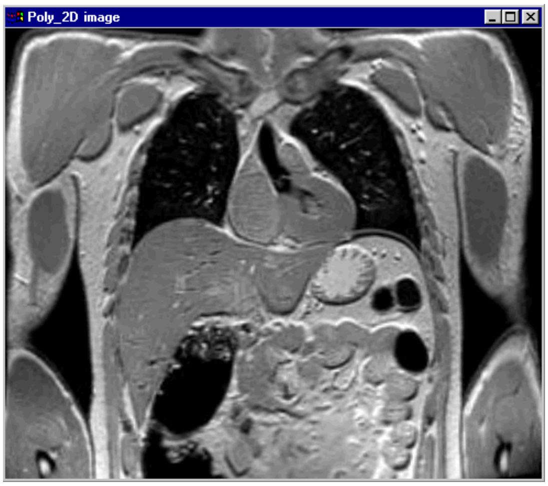


Figure 5-4: Warped Image Produced with POLY_2D

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.

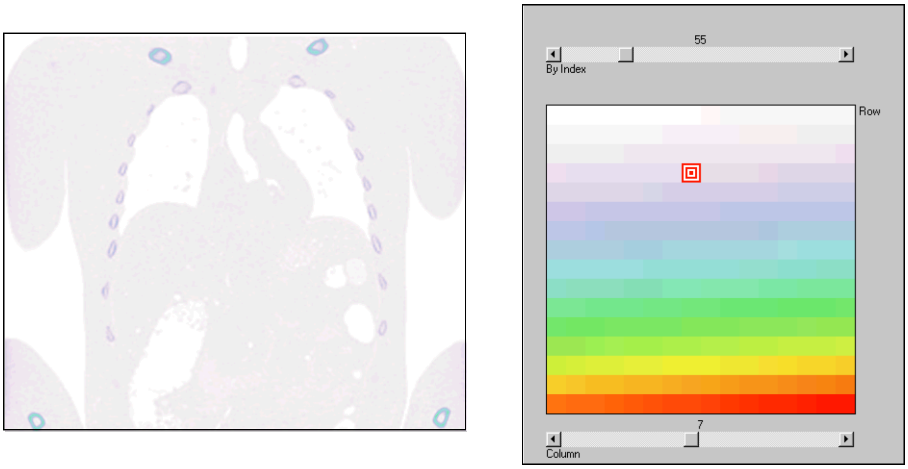


Figure 5-5: Using XPALETTE to Identify Mask Values

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.

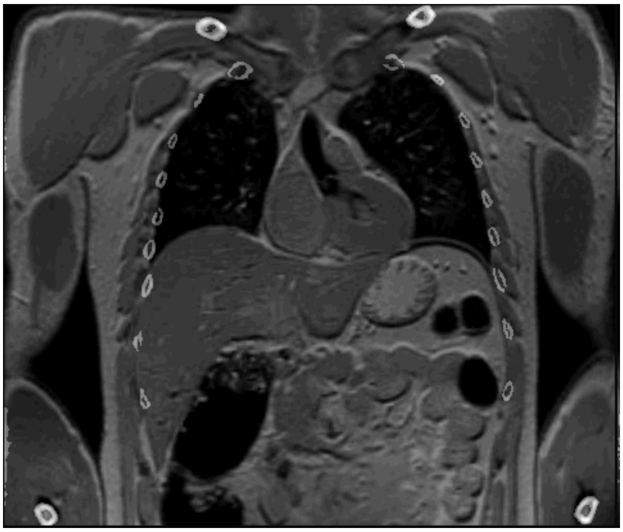
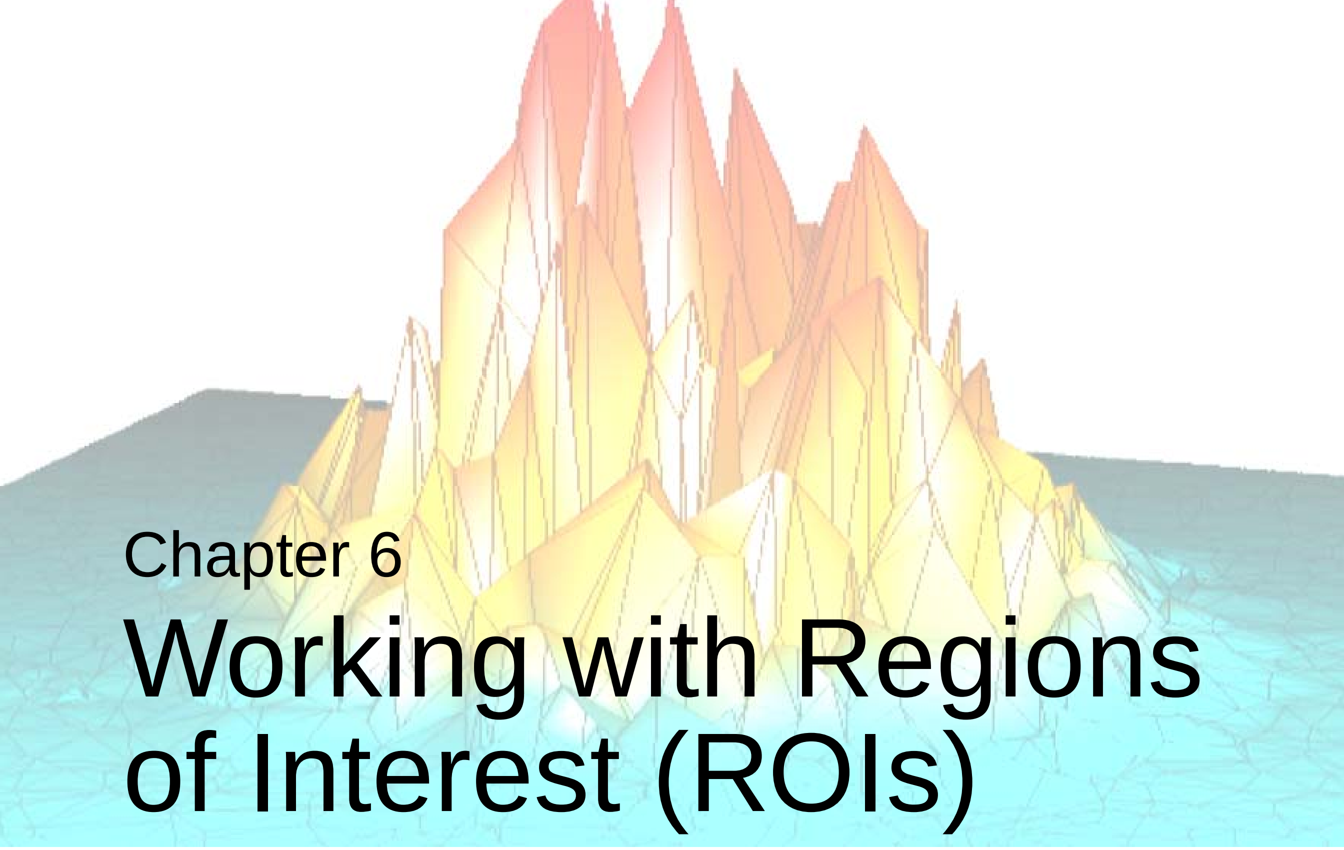


Figure 5-6: Direct Graphics Display of a Transparent Blended Image

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 Image Objects" in Chapter 4 of the *Object Programming* manual.



Chapter 6

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, 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 101 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 103.	XROI	Create an ROI interactively, prior to analysis or display.
"Displaying ROI Objects in a Direct Graphics Window" on page 105.	DRAW_ROI	Display ROI objects in a Direct Graphics window.

Table 6-1: Tasks and Routines Associated with Regions of Interest

Task	Routine(s)/Object(s)	Description
"Programmatically Defining ROIs" on page 109.	CONTOUR DRAW_ROI IDLanROI::ComputeMask IMAGE_STATISTICS IDLanROI::ComputeGeometry	Define ROIs using CONTOUR and display them using DRAW_ROI. Return various statistics for each ROI.
"Growing a Region" on page 113.	REGION_GROW	Expand an original region to include all connected, neighboring pixels which meet specified constraints.
"Creating and Displaying an ROI Mask" on page 118.	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 122.	IDLanROI::ContainsPoints	Determine whether a point lies within the boundary of a region.
"Creating a Surface Mesh of an ROI Group" on page 125.	IDLanROIGroup::Add IDLanROIGroup::ComputeMesh XOBJVIEW	Add ROIs to an ROI group object, triangulate a surface mesh and display the group object using XOBJVIEW.

Table 6-1: 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:

$1.5 \le x < 2.5$ and $1.5 \le y < 2.5$

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.

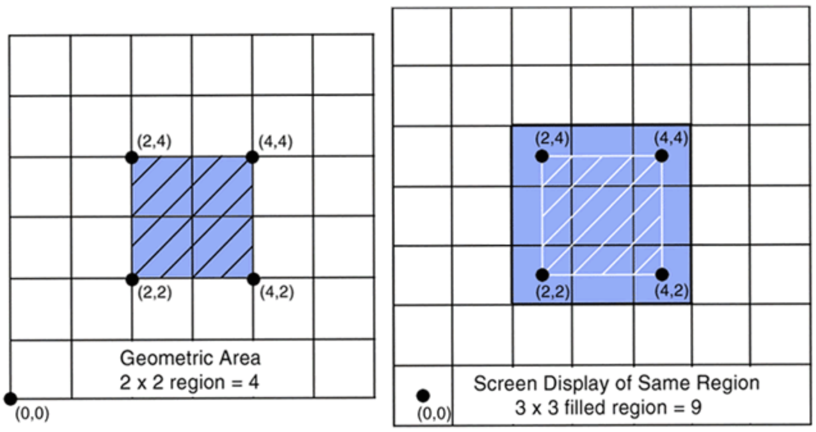


Figure 6-1: A Region's Undisplayed Area (left) vs. Displayed Area (right)

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 109 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 109 and "Growing a Region" on page 113), 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:

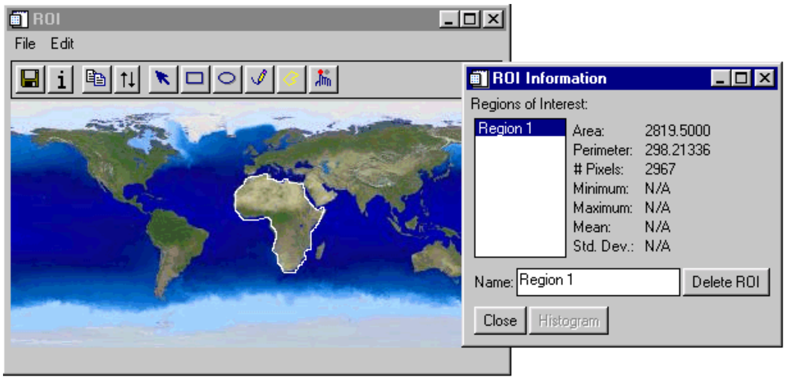
1. Open XROI by typing the following at the command line:

   ```
   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.

   See "Using XROI" under "XROI" in the *IDL Reference Guide* manual for details on the interface elements.Flip the image vertically to display it right-side-up by clicking the **Flip** button.

3. 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

Figure 6-2: Defining an ROI of Africa and Showing the ROI Information Dialog

4. 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. Complete the following steps for a detailed description of the process.

Example Code

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

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*.

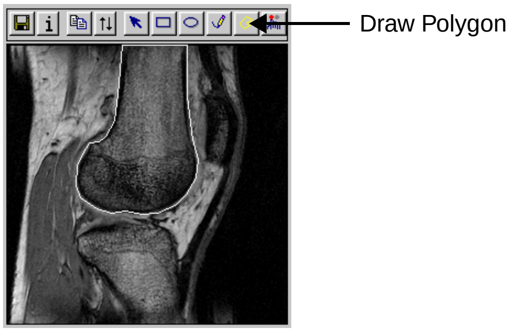


Figure 6-3: Defining the Femur ROI

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.

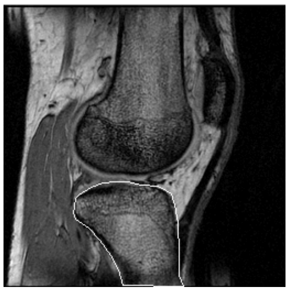


Figure 6-4: Defining the Tibia ROI

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.

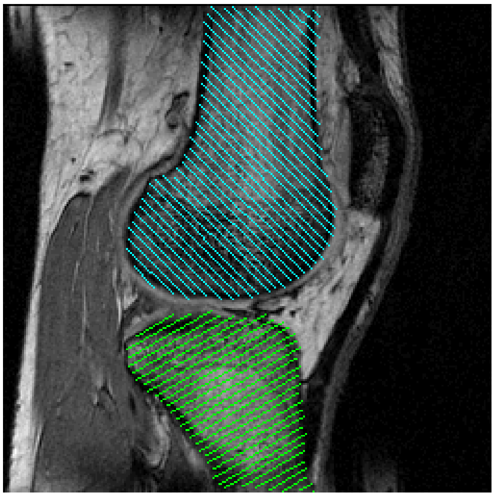


Figure 6-5: Defined Region Objects Overlaid onto Original Image

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 101.

9. Clean up object references that are not destroyed by the window manager when you close the Object Graphics displays:

```
OBJ_DESTROY, [femurROIout, tibiaROIout]
```

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

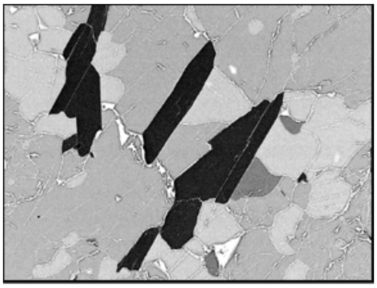


Figure 6-6: Initial Image

4. Create a mask that identifies the darkest pixels, whose values are less than 50:

```
threshImg = (img LT 50)
```

Note

See "Determining Intensity Values for Threshold and Stretch" on page 243 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 `ProgramDefineROI.pro` in the `examples/doc/image` subdirectory of the IDL installation directory.

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.

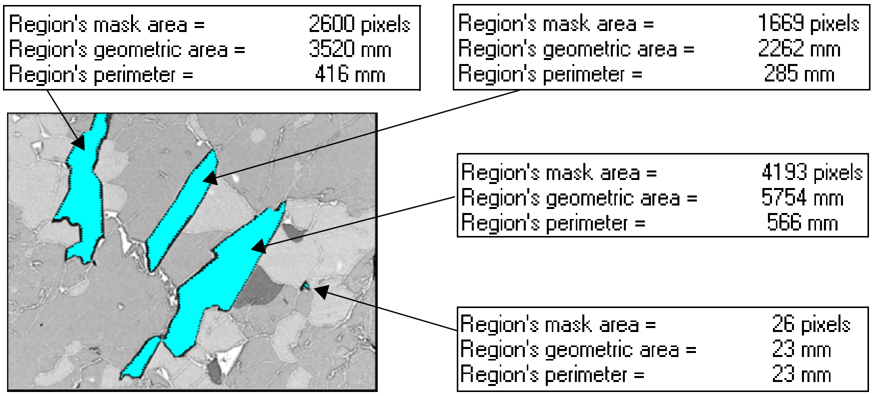

Region's mask area = 2600 pixels
Region's geometric area = 3520 mm
Region's perimeter = 416 mm
Region's mask area = 1669 pixels
Region's geometric area = 2262 mm
Region's perimeter = 285 mm
Region's mask area = 4193 pixels
Region's geometric area = 5754 mm
Region's perimeter = 566 mm
Region's mask area = 26 pixels
Region's geometric area = 23 mm
Region's perimeter = 23 mm

Figure 6-7: Display of Programmatically Defined Regions

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

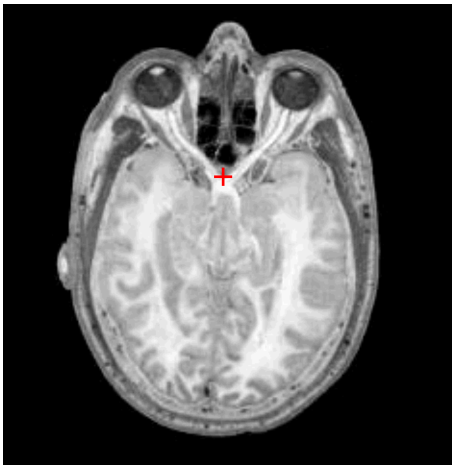


Figure 6-8: Original Image Showing Region to be Selected

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.

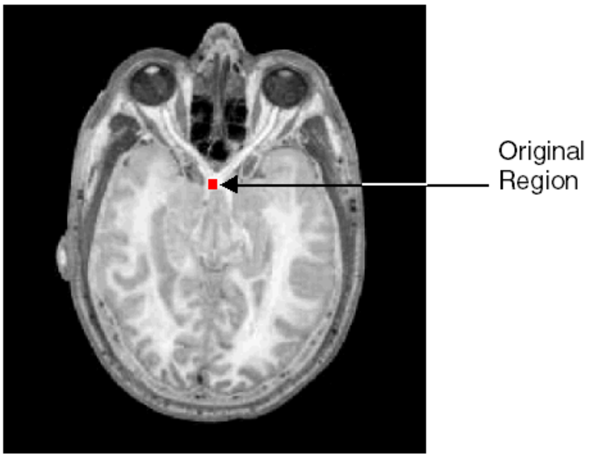


Figure 6-9: Square ROI at Selected Coordinates

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).

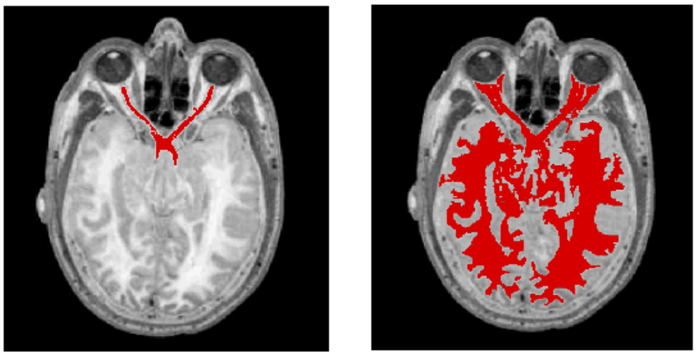


Figure 6-10: Regions Expanded Using REGION_GROW

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.

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. Complete the following steps for a detailed description of the process.

Example Code

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

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:

```
XROI, img, REGIONS_OUT = ROIout, /BLOCK
```

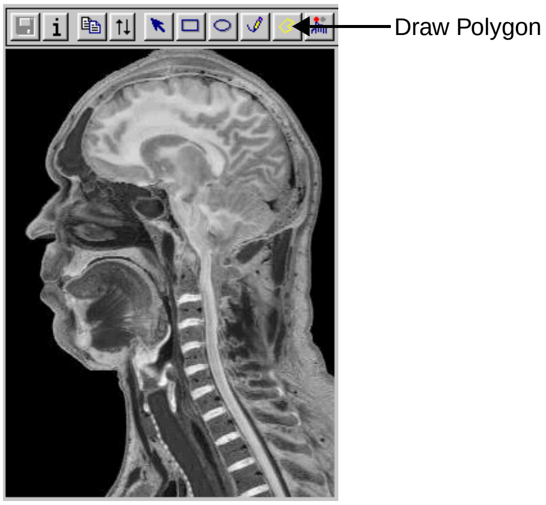


Figure 6-11: ROI Definition in XROI

Close the XROI window to save the region object data in the variable, *ROIout*.

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.

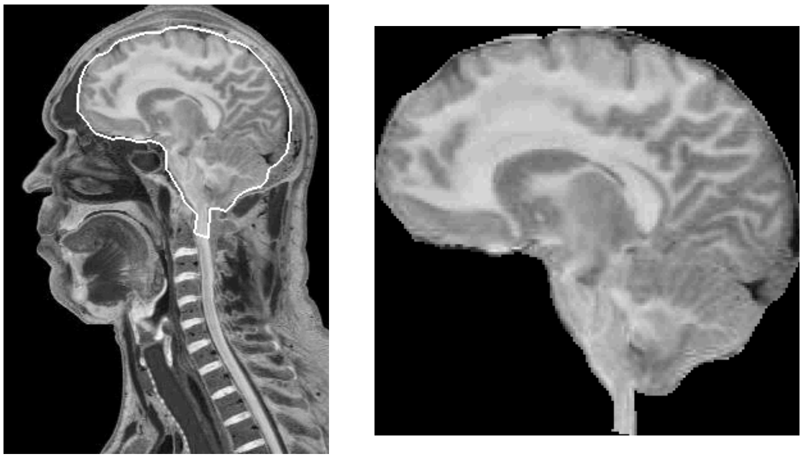


Figure 6-12: Original and Magnified View of the ROI

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]
```

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

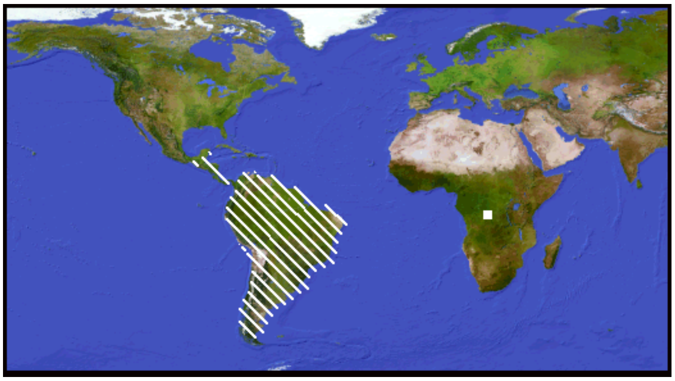


Figure 6-13: Detail of Point Containment Test

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. Complete the following steps for a detailed description of the process.

Example Code

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

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 `GroupROIMesh.pro` in the `examples/doc/image` subdirectory of the IDL installation directory.

The following image shows samples of the ROIs to be defined.

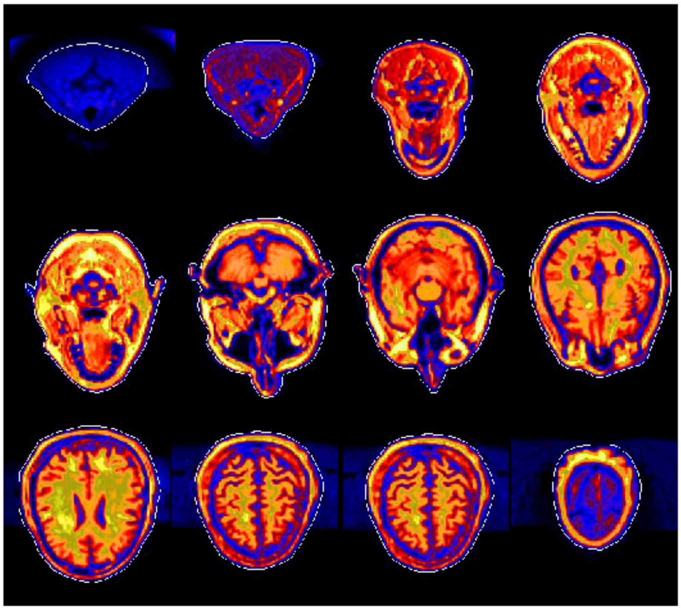


Figure 6-14: ROIs to be Defined

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.

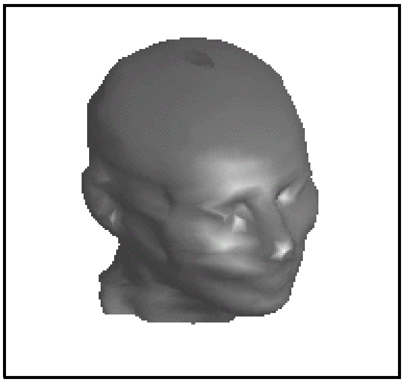
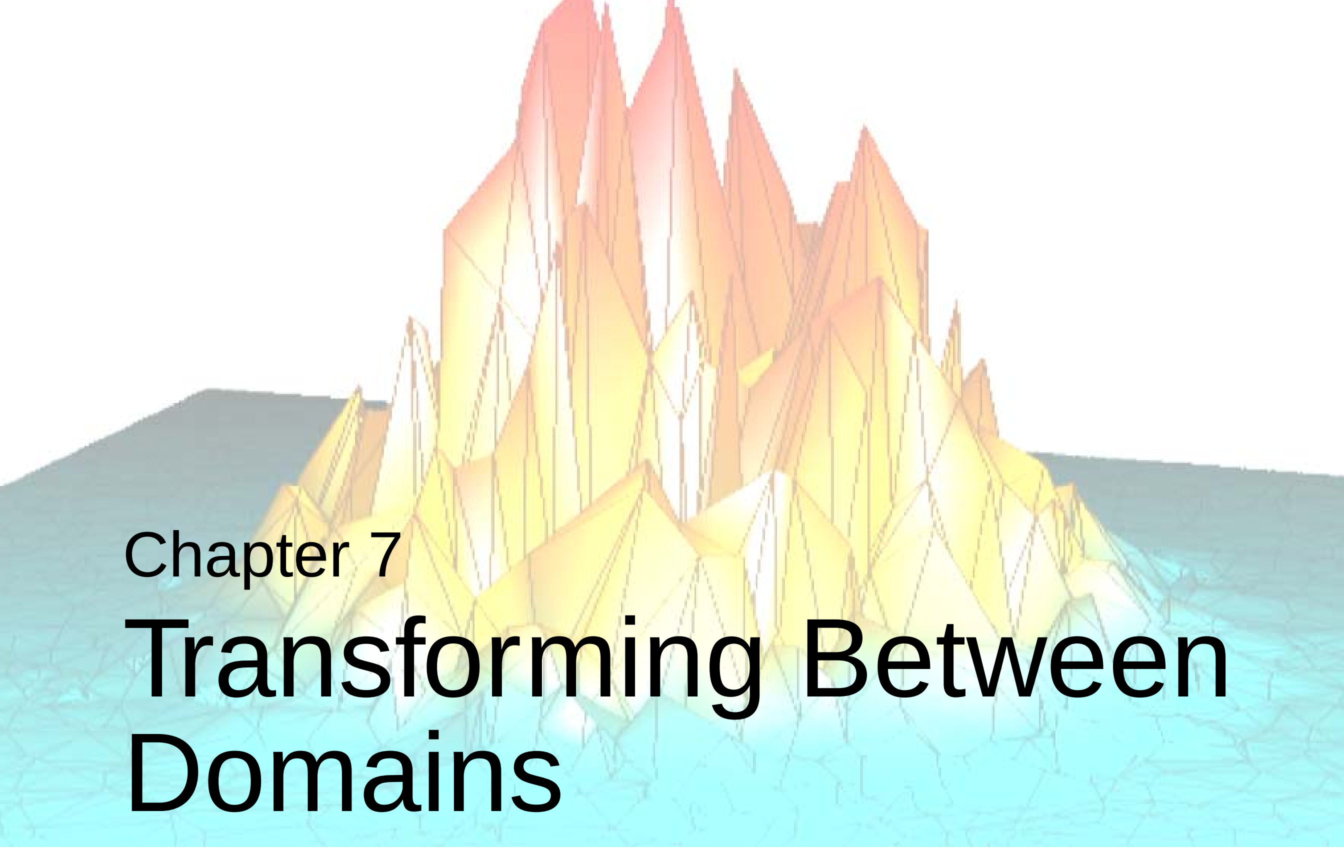


Figure 6-15: Result of Creating a Mesh from a Group of ROIs



Chapter 7

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 Between Domains with FFT" on page 132 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 Between Domains with Wavelets" on page 148 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 161 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 161 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 Between Domains with FFT" on page 132	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 Between Domains with Wavelets" on page 148	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.
"Transforming to and from the Hough and Radon Domains" on page 161	HOUGH 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 7-1: Image Transformation 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.

Transforming Between Domains 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 136
- "Transforming from the Frequency Domain" on page 140

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 143 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 136 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

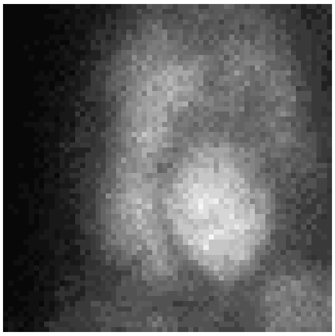


Figure 7-1: Original Gated Blood Pool Image

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.

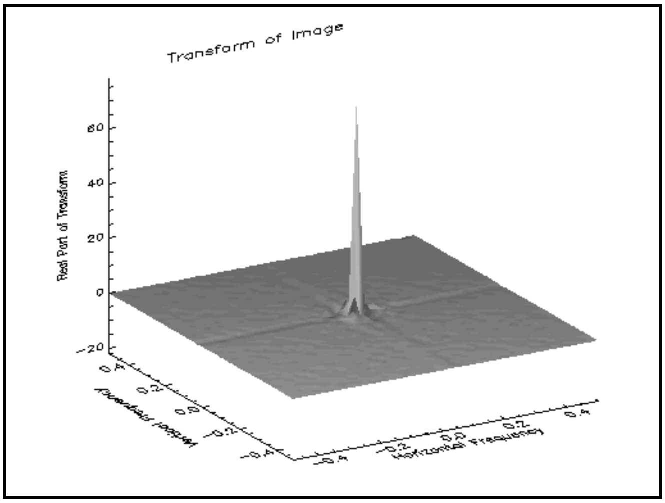


Figure 7-2: FFT of the Gated Blood Pool Image

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.

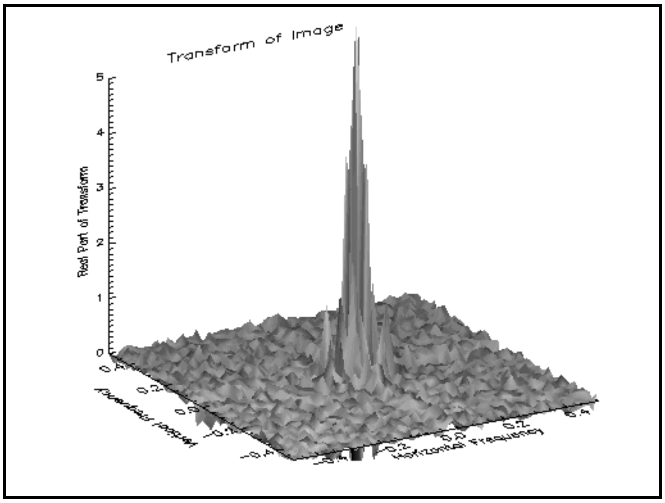


Figure 7-3: FFT of the Gated Blood Pool Image Scaled Between 0 and 5

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

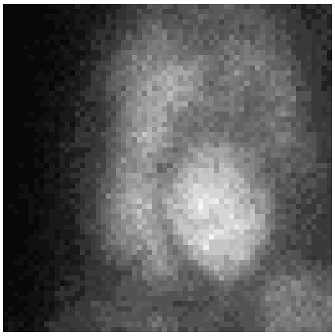


Figure 7-4: Original Gated Blood Pool Image

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.

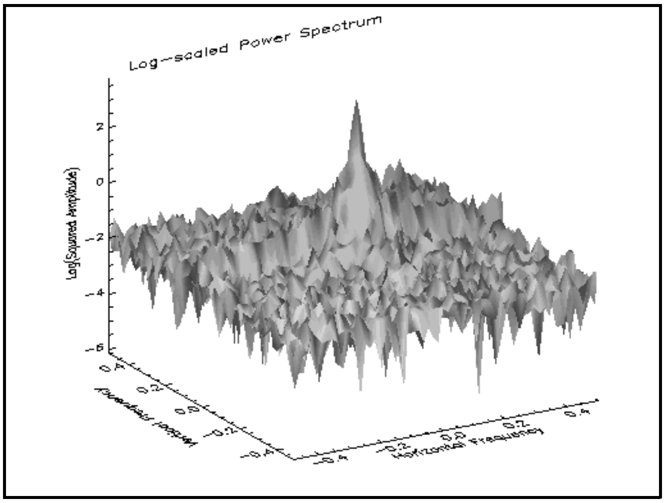


Figure 7-5: Log-scaled FFT Power Spectrum of Image (as a surface)

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.

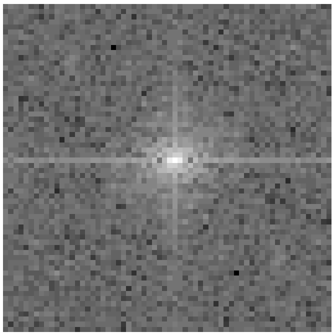


Figure 7-6: Log-scaled FFT Power Spectrum of Image (as an image)

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 143. Complete the following steps for a detailed description of the process.

Example Code

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

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.

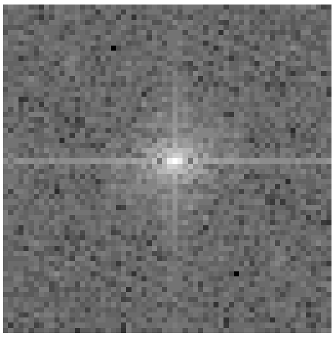


Figure 7-7: Log-scaled FFT Power Spectrum of the Gated Blood Pool Image

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.

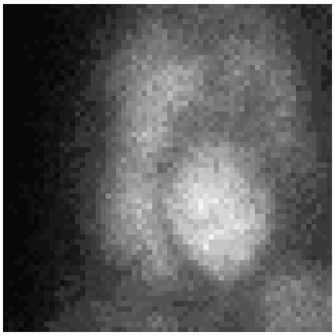


Figure 7-8: Inverse FFT of the Gated Blood Pool Image

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

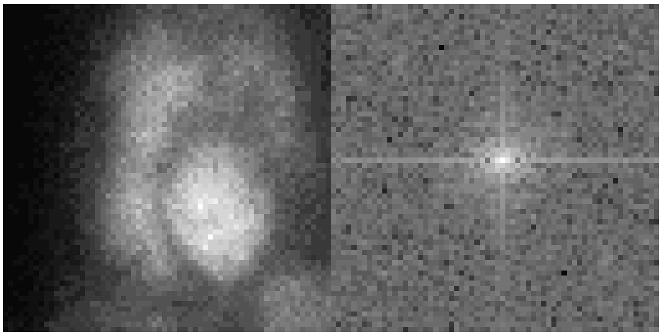


Figure 7-9: Original Image and Its FFT Power Spectrum

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.

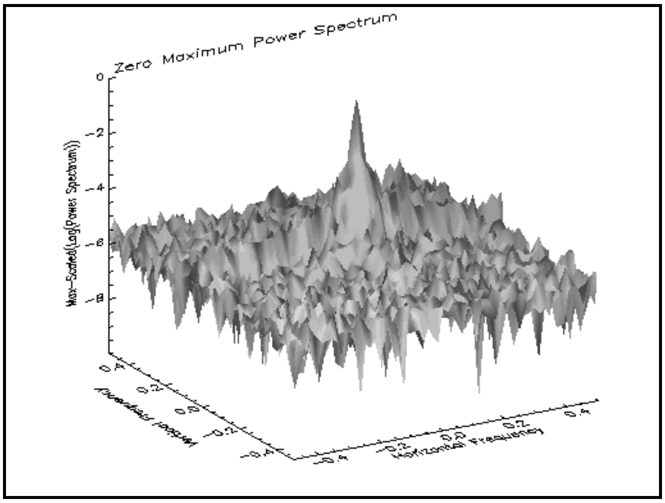


Figure 7-10: FFT Power Spectrum of the Image Scaled to a Zero Maximum

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.

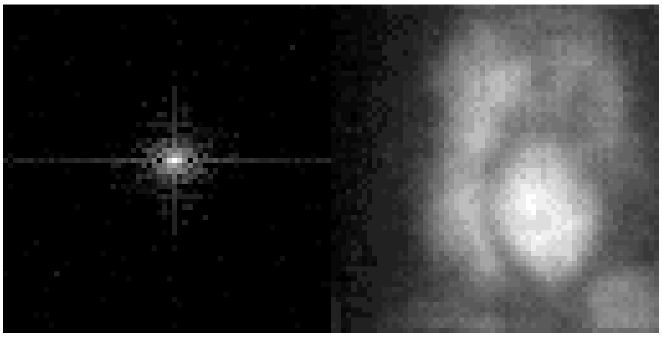


Figure 7-11: Masked FFT Power Spectrum and Resulting Inverse Transform

Transforming Between Domains 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 152
- “Transforming from the Time-Frequency Domain” on page 155

The wavelet transformation process is the basis for many image compression algorithms. See “Removing Noise with the Wavelet Transform” on page 158 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 152 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

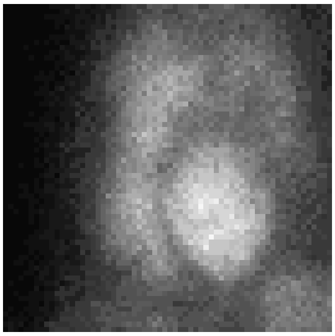


Figure 7-12: Original Gated Blood Pool Image

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.

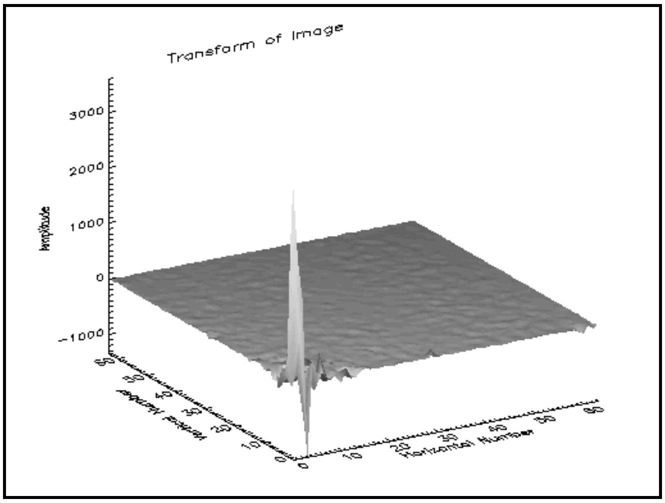


Figure 7-13: Wavelet Transform of Gated Blood Pool Image

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.

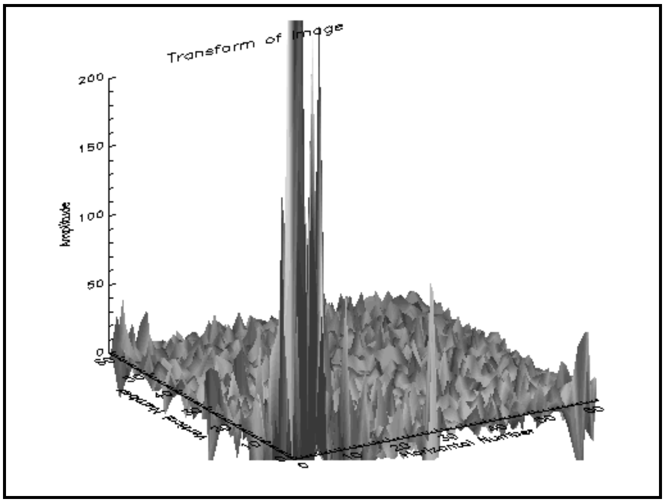


Figure 7-14: Wavelet Transform of Image Scaled Between 0 and 200

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

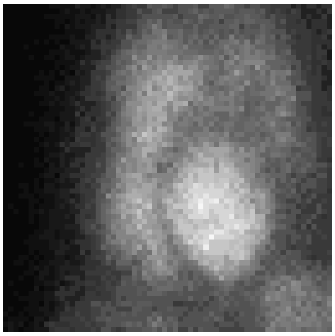


Figure 7-15: Original Gated Blood Pool Image

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.

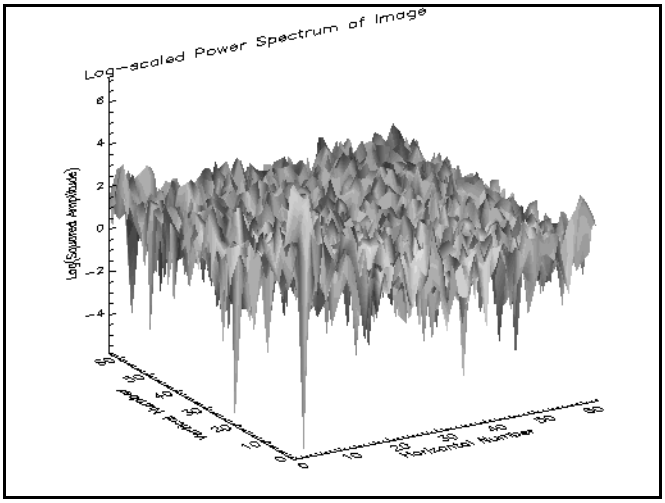


Figure 7-16: Log-scaled Wavelet Power Spectrum of Image (as a surface)

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.

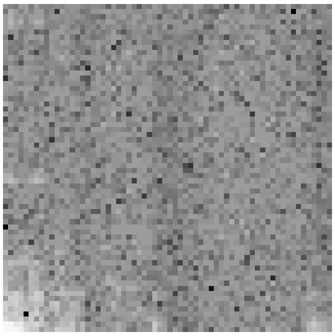


Figure 7-17: Log-scaled Wavelet Power Spectrum of Image (as am image)

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 158. Complete the following steps for a detailed description of the process.

Example Code

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

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.

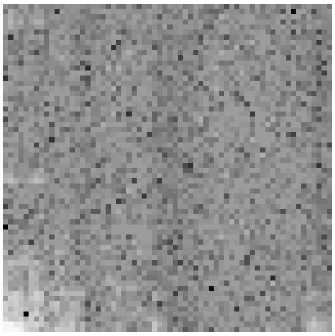


Figure 7-18: Log-scaled Wavelet Power Spectrum of Image

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. No image data is lost when transforming an image to and from the time-frequency domain.

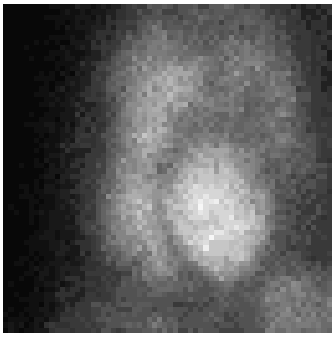


Figure 7-19: Inverse of the Wavelet Transform of the Gated Blood Pool Image

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

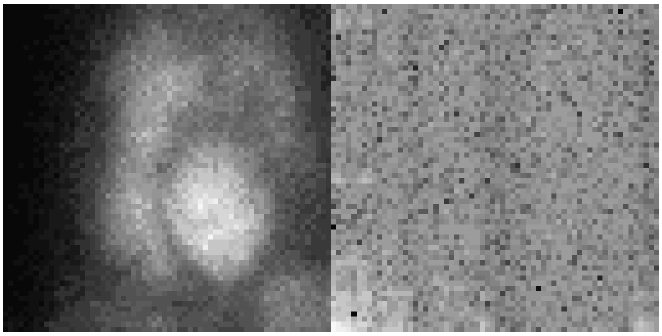


Figure 7-20: Gated Blood Pool Image and Its Wavelet Power Spectrum

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.

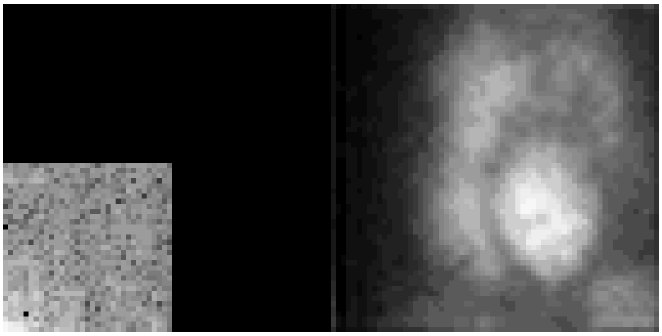


Figure 7-21: Masked Wavelet Power Spectrum and Its Resulting Inverse Transform

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 162
- "Transforming from the Hough and Radon Domains (Backprojecting)" on page 165

The Hough transformation process is used to find straight lines within an image. See "Finding Straight Lines with the Hough Transform" on page 168 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 174 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

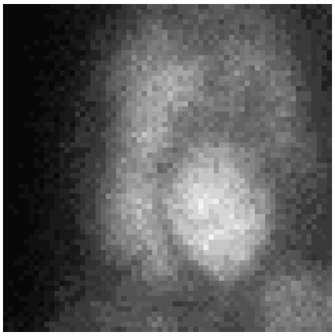


Figure 7-22: Original Gated Blood Pool Image

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.

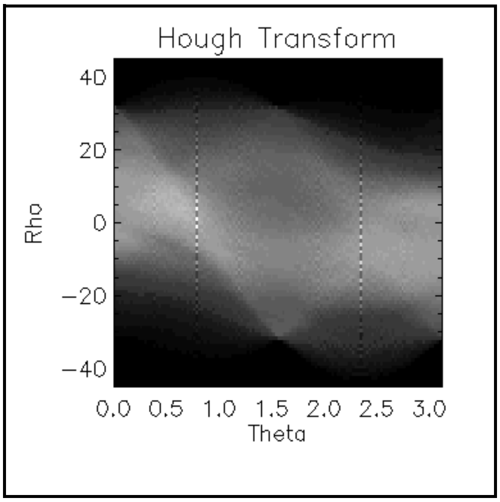


Figure 7-23: Hough Transform of the Gated Blood Pool Image

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.

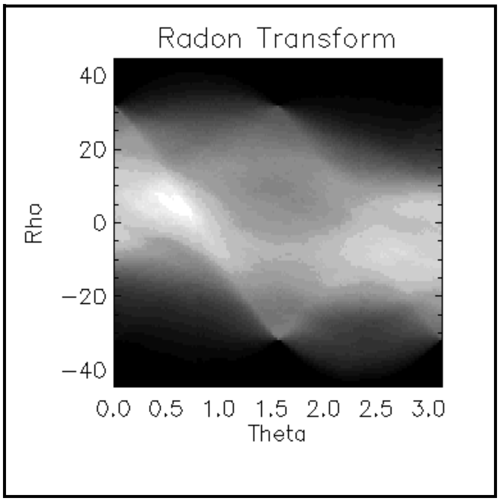


Figure 7-24: Radon Transform of the Gated Blood Pool Image

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

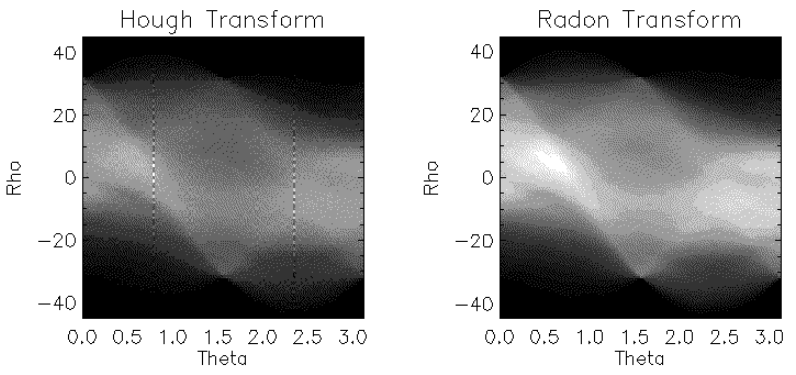


Figure 7-25: Hough (left) and Radon (right) Transforms of Image

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.

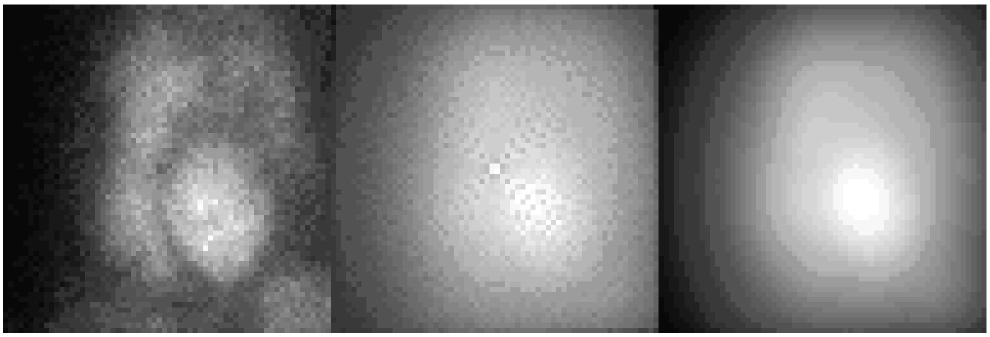


Figure 7-26: Original Gated Blood Pool Image (left), Hough Backprojection (center), and Radon Backprojection (right)

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

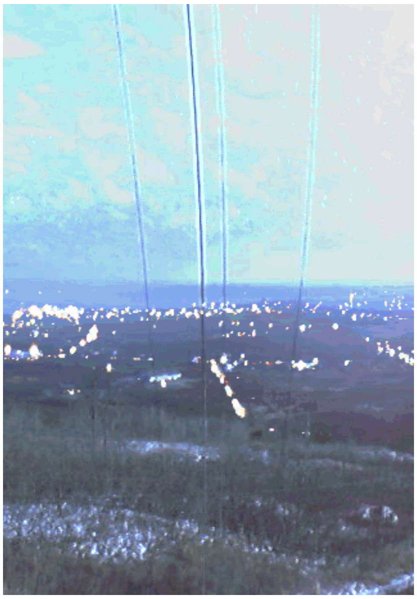


Figure 7-27: Image of Rockland, Maine

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.

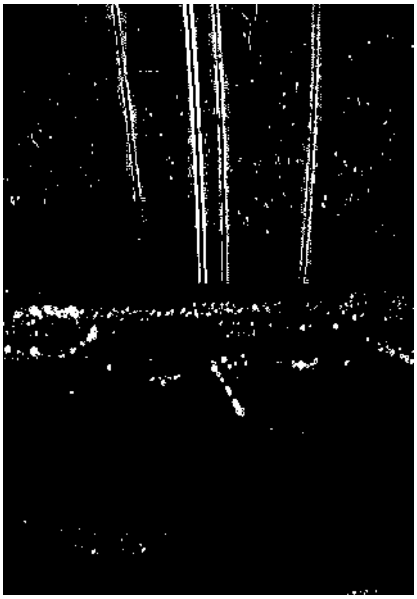


Figure 7-28: Mask of Rockland Image

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.

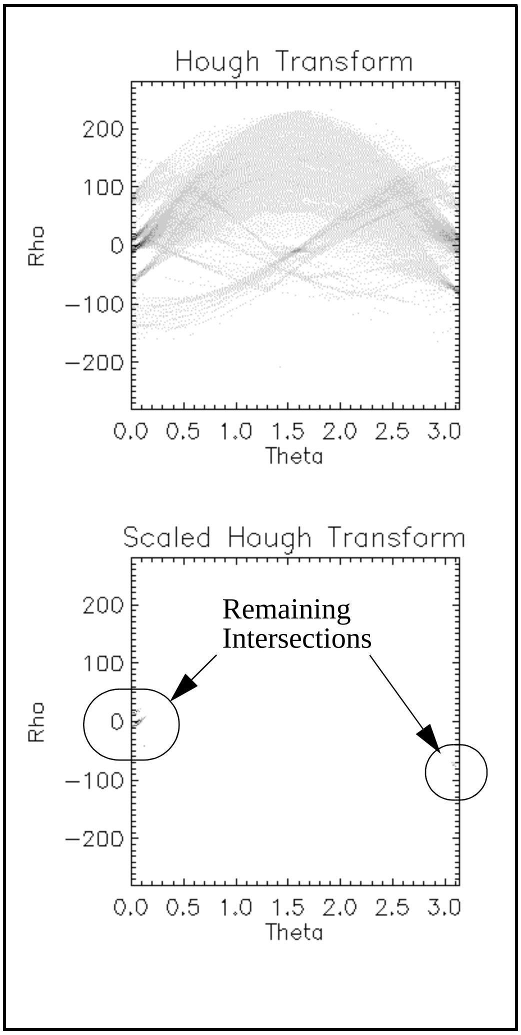


Figure 7-29: The Hough Transform (top) and the Scaled Transform (bottom) of the Masked Intensity Image

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.

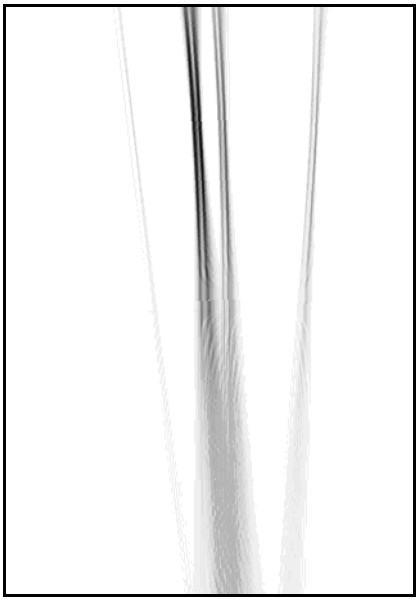


Figure 7-30: The Resulting Backprojection of the Scaled Hough Transform

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.

Example Code

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

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 and display it:

```
image = ROBERTS(endocellImage)
TVSCL, image, 1
```

The following figure shows the results of the edge detection filter.

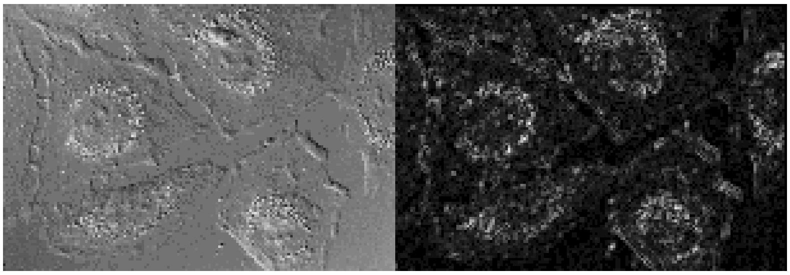


Figure 7-31: Original Image (left) and the Resulting Edge-Filtered Image (right)

7. Transform the filtered image:

```
transform = RADON(image, RHO = rho, THETA = theta)
```

8. Define the size and offset parameters for the transform displays:

```
displaySize = [256, 256]
offset = displaySize/3
```

9. 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
```

10. Scale the transform to include only the density values above the mean of the transform:

```
scaledTransform = transform > MEAN(transform)
```

11. 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.

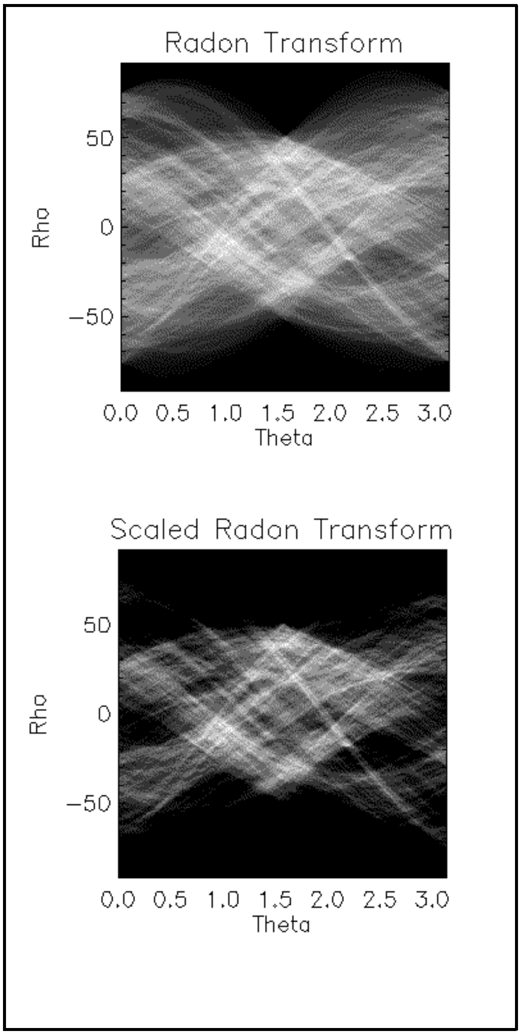


Figure 7-32: Radon Transform (top) and Scaled Transform (bottom) of the Edge-Filtered Image

12. Backproject the scaled transform:

```
backprojection = RADON(scaledTransform, /BACKPROJECT, $
   RHO = rho, THETA=theta, NX = imageSize[0], $
   NY = imageSize[1])
```

13. 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
```

14. Use the backprojection as a mask to provide a color density contrast of the original image:

```
constrastingImage = endocellImage*backprojection
```

15. 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.

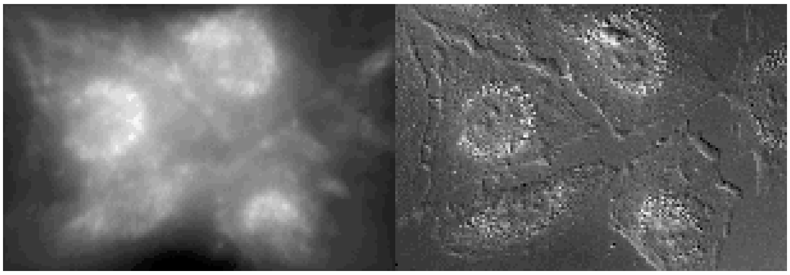
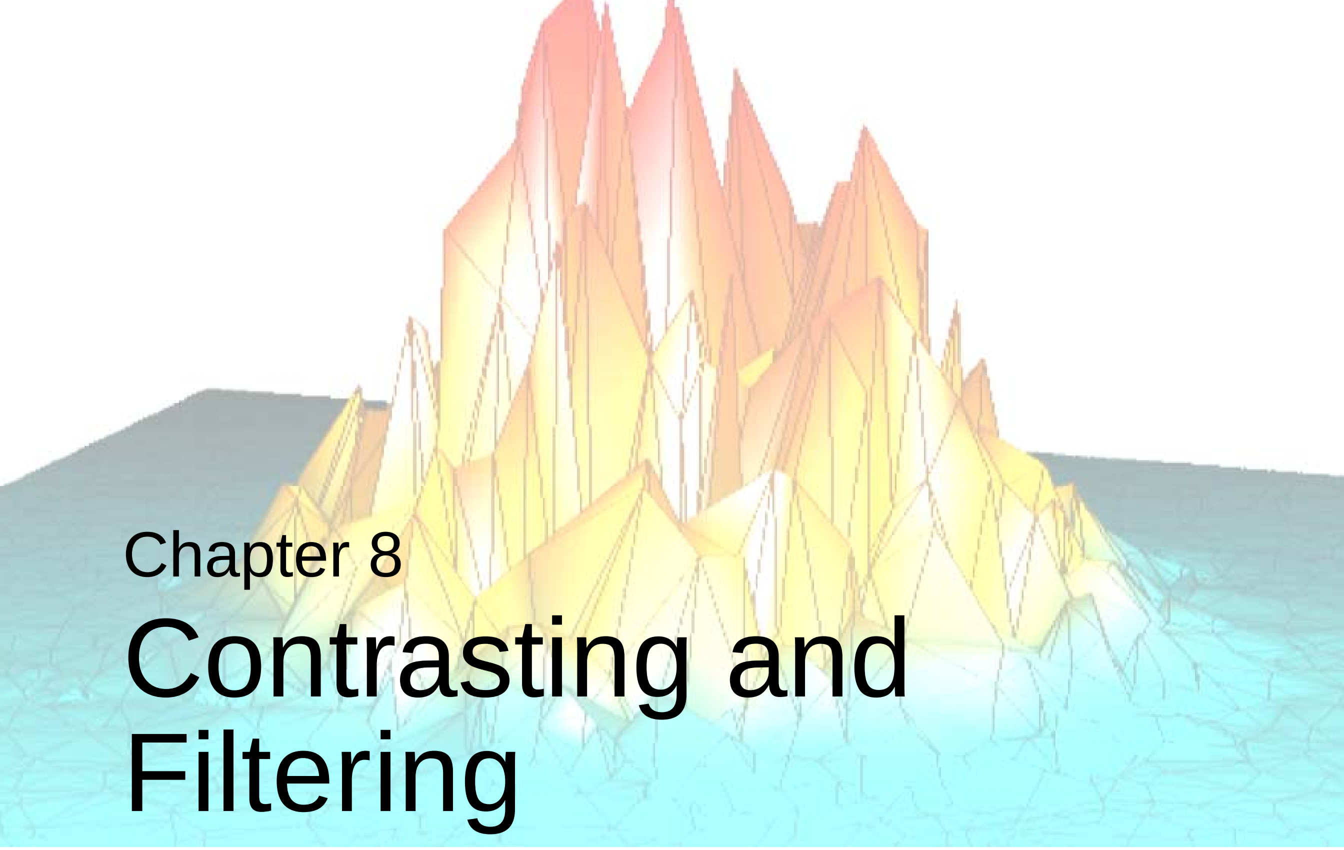


Figure 7-33: The Backprojection of the Radon Transform and the Resulting Contrast Image



Chapter 8
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 183 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 186 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 195 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 211 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 220 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 224 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 229 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 183	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 186	HIST_EQUAL ADAPT_HIST_EQUAL	Use histogram equalization to show minor variations in uniform areas.
"Filtering an Image" on page 195	CONVOL	Enhance contrast by applying some basic filters (low pass, high pass, directional, and Laplacian) to images.

Table 8-1: Image Contrasting and Filtering Tasks and Related Routines

Type of Contrasts or Filters	Routines	Description
"Smoothing an Image" on page 211	SMOOTH MEDIAN	Smooth high variations within an image.
"Sharpening an Image" on page 220	CONVOL	Sharpen an image by decreasing too bright pixels and increasing too dark pixels.
"Detecting Edges" on page 224	ROBERTS SOBEL	Use the contrast within an image to detect the possible edges of shapes.
"Removing Noise" on page 229	HANNING LEEFILT	Remove noise from an image by either windowing or using an adaptive filter.

Table 8-1: 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

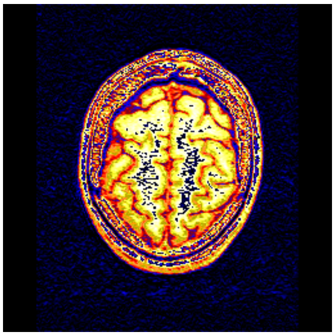


Figure 8-1: Magnetic Resonance Image (MRI) of a Human Brain

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.

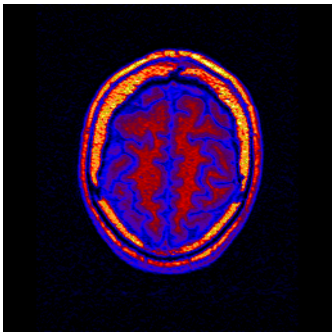


Figure 8-2: Byte-Scaled MRI

Working with Histograms

The histogram of an image shows the number of pixels for each pixel 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 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.

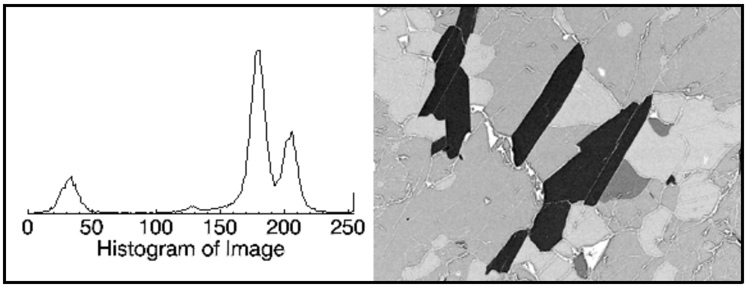


Figure 8-3: Example of a Histogram (left) and Its Related Image (right)

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 "H_EQ_CT" in the *IDL Reference Guide* manual 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 190

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

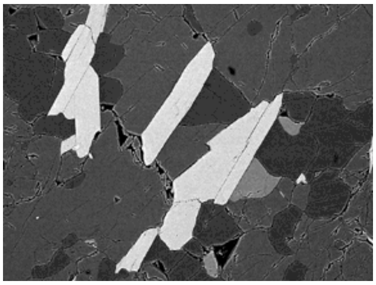


Figure 8-4: The Mineral Image and Its Related Color Table

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.

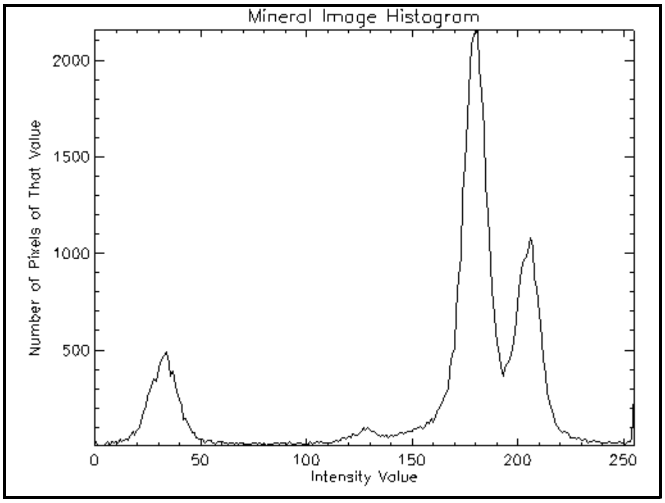


Figure 8-5: Histogram of the Original Image

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.

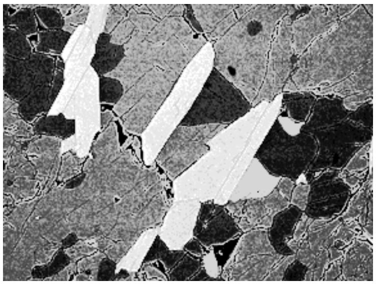


Figure 8-6: Equalized Mineral Image

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.

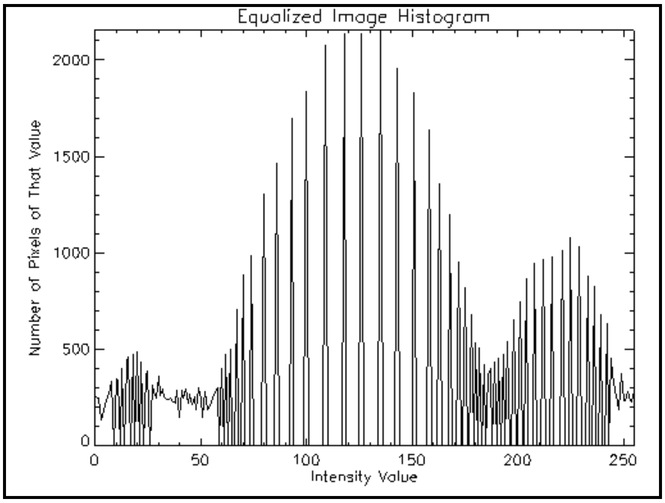


Figure 8-7: Histogram of the Equalized Image

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

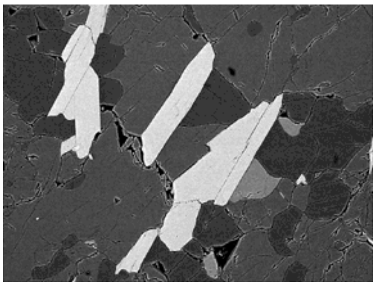


Figure 8-8: The Mineral Image and Its Related Color Table

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.

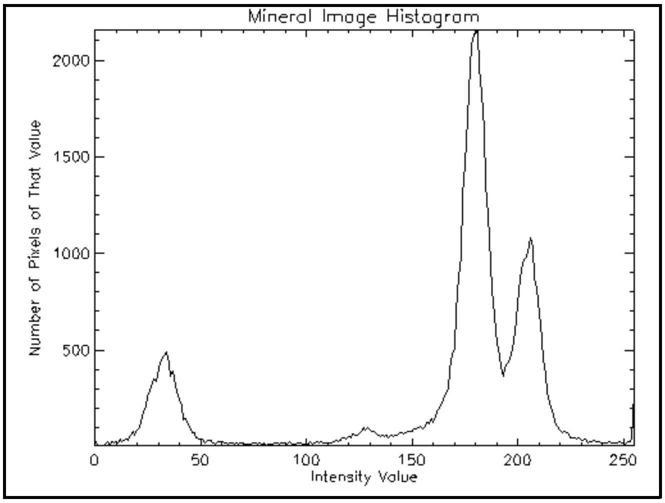


Figure 8-9: Histogram of the Original Image

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.

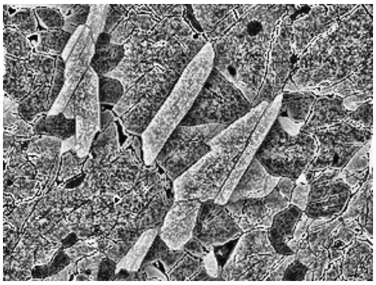


Figure 8-10: Adaptive Equalized Mineral Image

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.

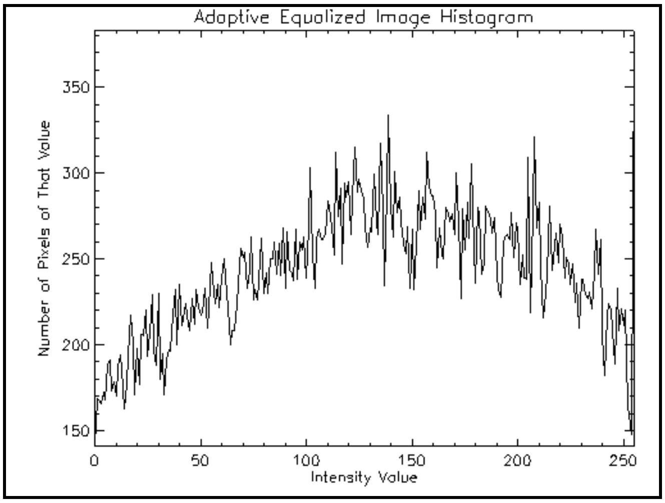


Figure 8-11: Histogram of the Adaptive Equalized Image

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 7, "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 229 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 196
- "High Pass Filtering" on page 199
- "Directional Filtering" on page 203
- "Laplacian Filtering" on page 206

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 211 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

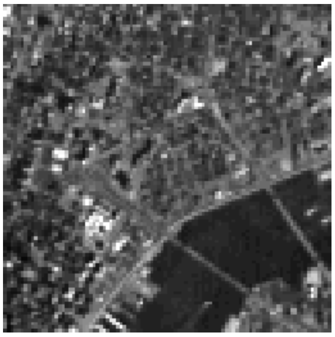


Figure 8-12: Cropped New York Image

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.

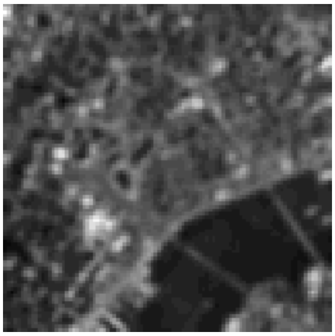


Figure 8-13: Low Pass Filtered New York Image

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.

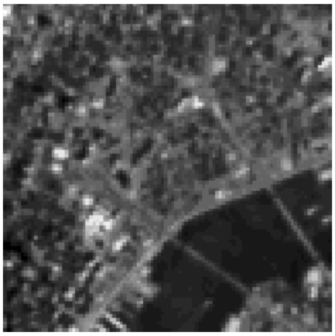


Figure 8-14: Low Pass Combined New York Image

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 220 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

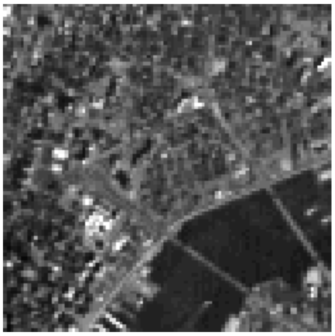


Figure 8-15: Cropped New York Image

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.

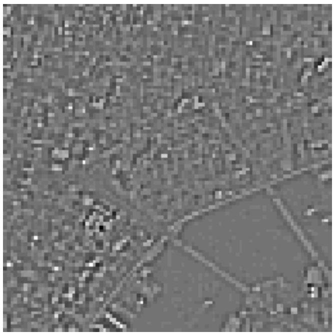


Figure 8-16: High Pass Filtered New York Image

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.

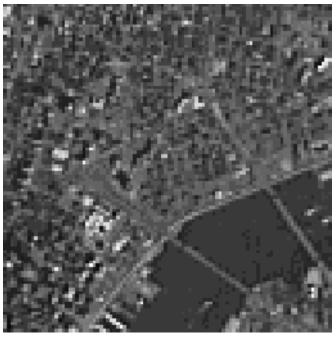


Figure 8-17: High Pass Combined New York Image

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 224 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

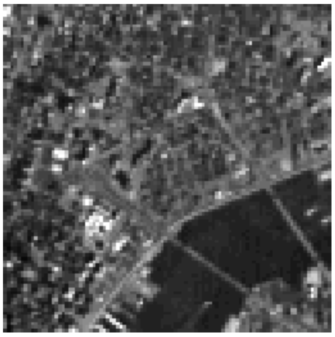


Figure 8-18: Cropped New York Image

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.

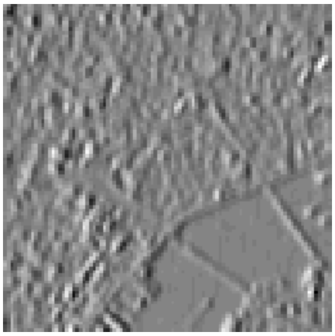


Figure 8-19: Direction Filtered New York Image

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.

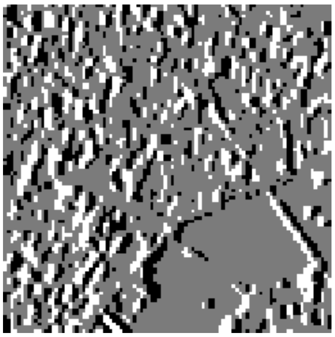


Figure 8-20: Slopes of Direction Filtered New York Image

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 224 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

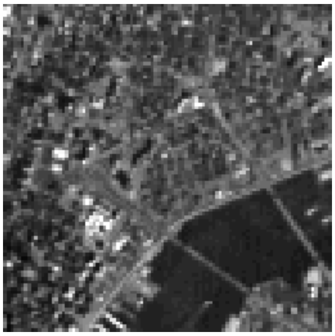


Figure 8-21: Cropped New York Image

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.

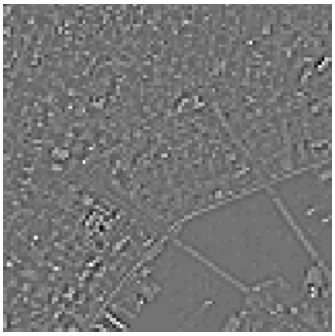


Figure 8-22: Laplacian Filtered New York Image

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.

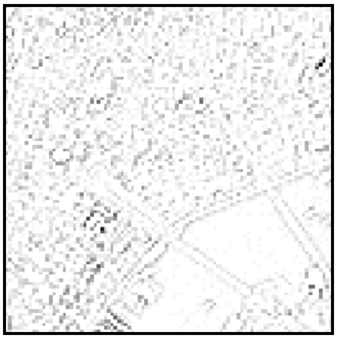


Figure 8-23: Negative Values of Laplacian Filtered New York Image

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 196 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 215

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

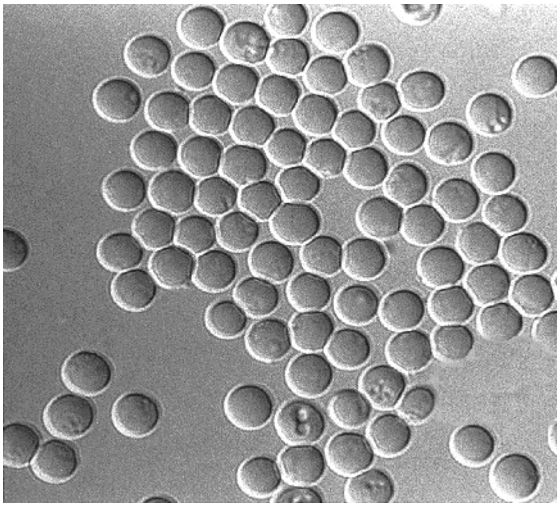


Figure 8-24: Original Red Blood Cells Image

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.

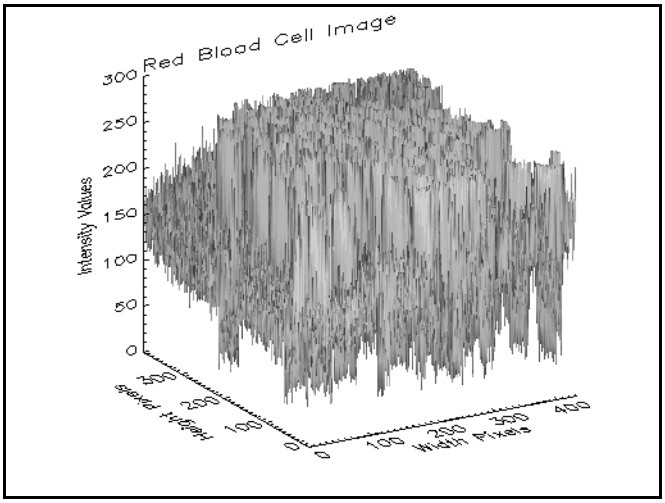


Figure 8-25: Surface of Original Red Blood Cells Image

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.

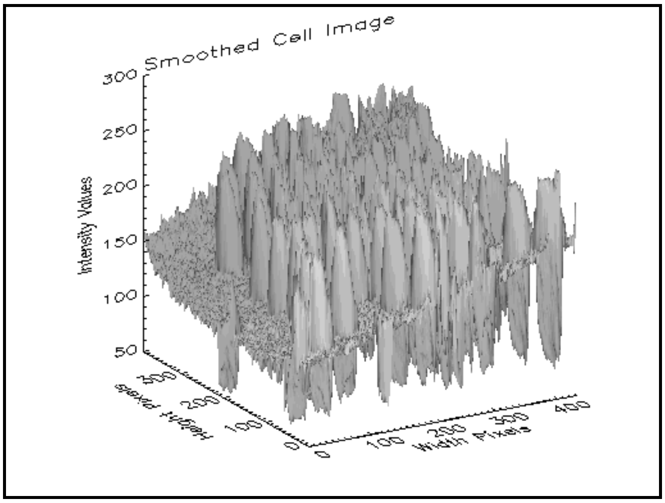


Figure 8-26: Surface of Average-Smoothed Red Blood Cells Image

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.

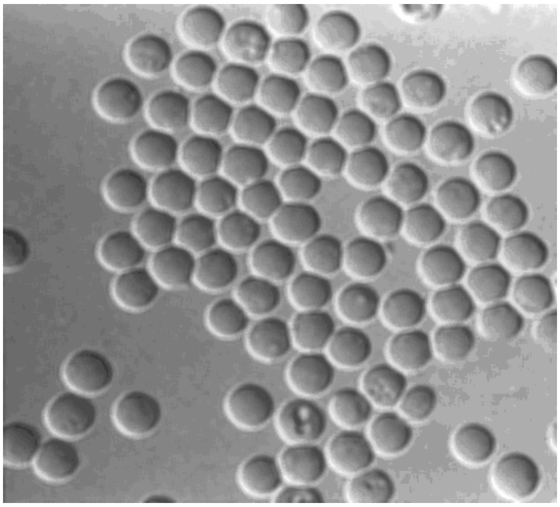


Figure 8-27: Average-Smoothed Red Blood Cells Image

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

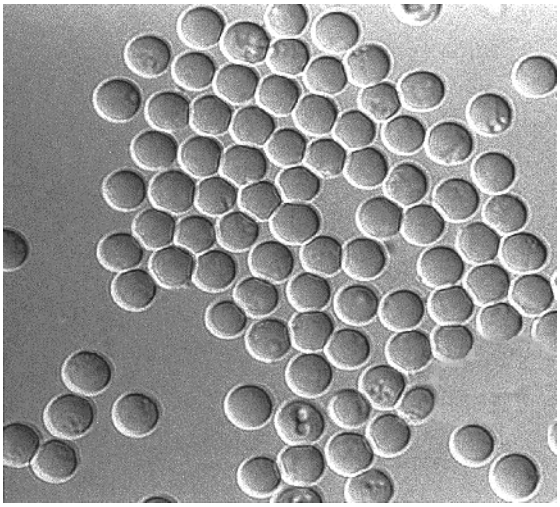


Figure 8-28: Original Red Blood Cells Image

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.

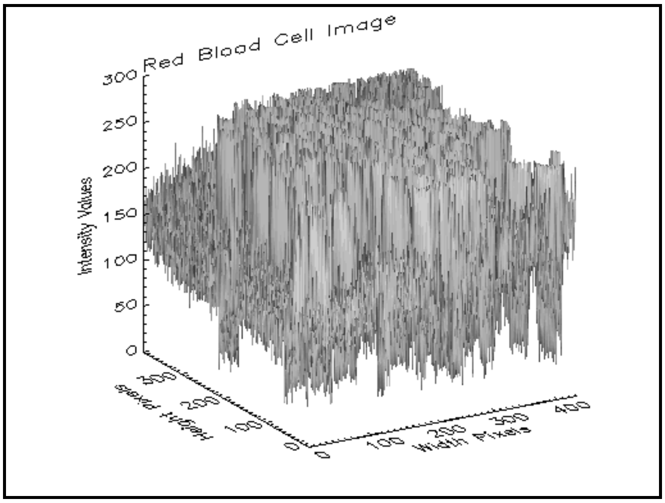


Figure 8-29: Surface of Original Red Blood Cells Image

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.

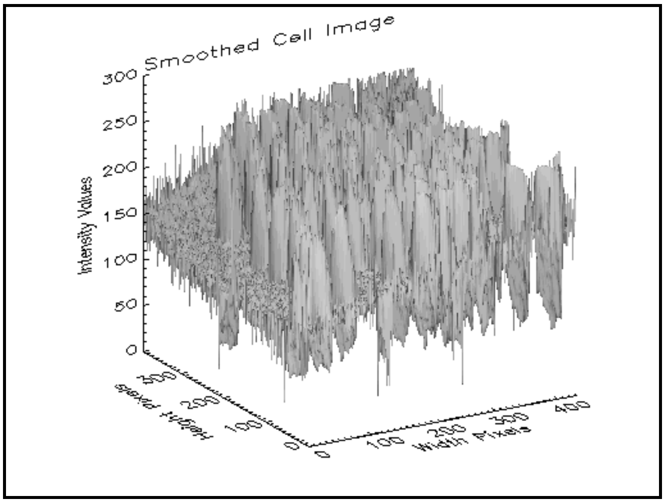


Figure 8-30: Surface of Middle-Smoothed Red Blood Cells Image

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.

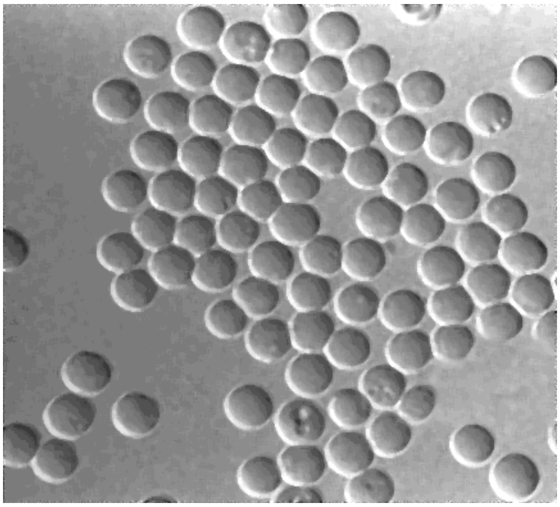


Figure 8-31: Middle-Smoothed Red Blood Cells Image

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 199 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

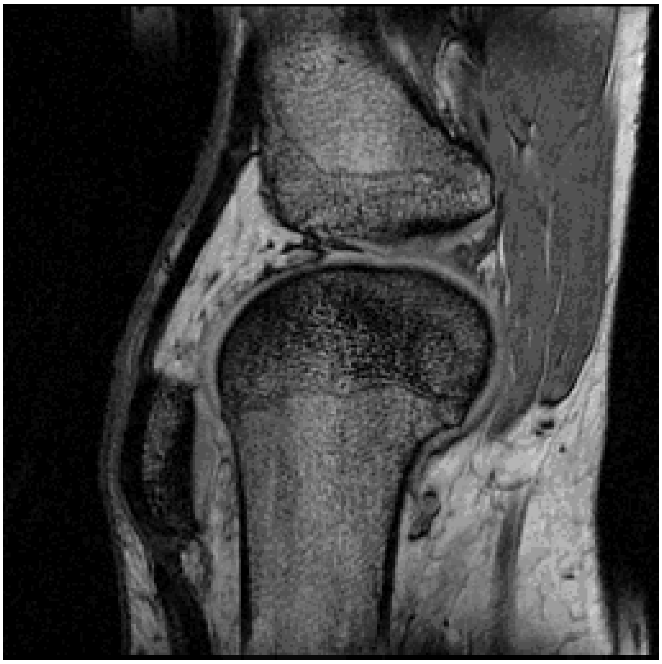


Figure 8-32: Original Knee MRI

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.

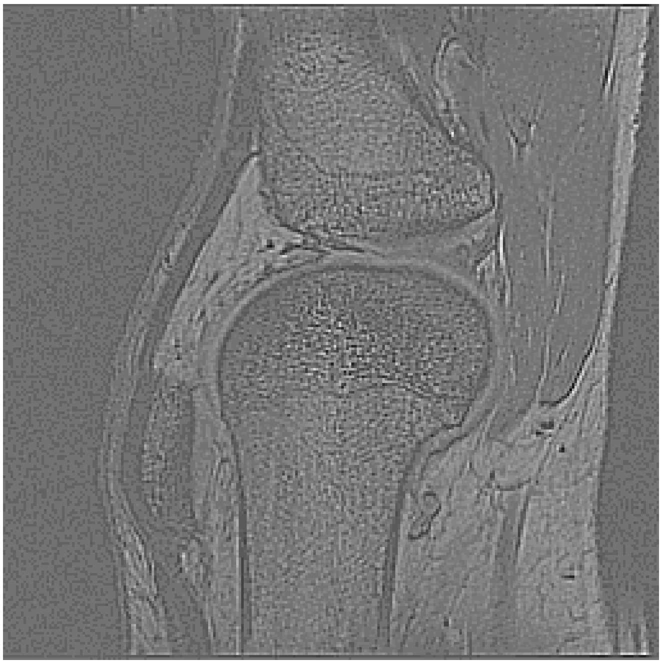


Figure 8-33: Sharpen FIltered Knee MRI

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.

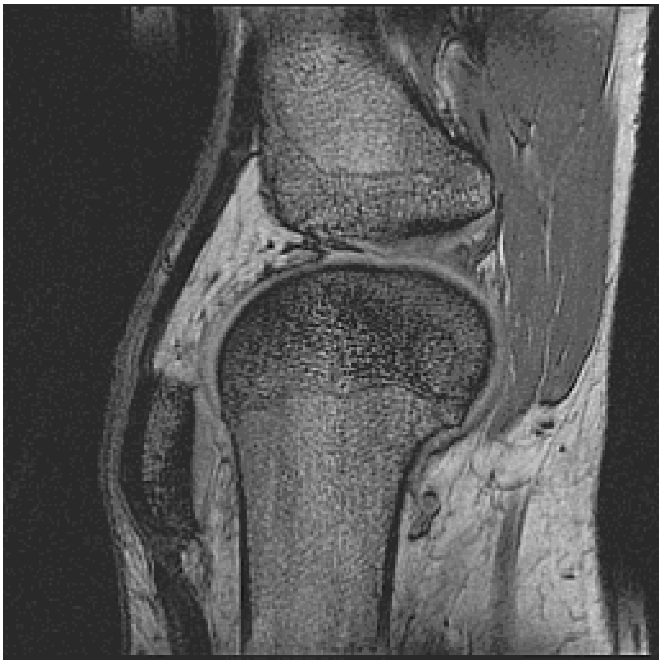


Figure 8-34: Sharpened Knee MRI

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 203 and “Laplacian Filtering” on page 206 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 9 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 226

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 4 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

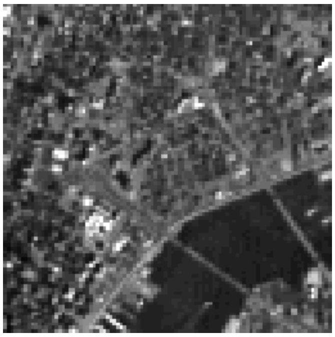


Figure 8-35: Cropped New York Image

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.

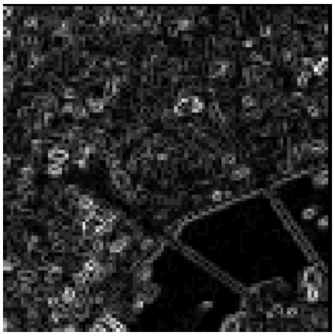


Figure 8-36: Roberts Filter Applied to the New York Image

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

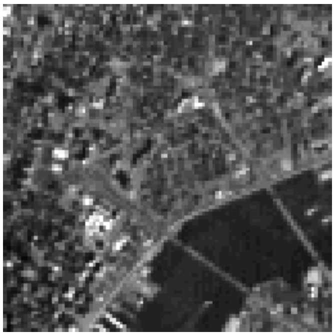


Figure 8-37: Cropped New York Image

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.

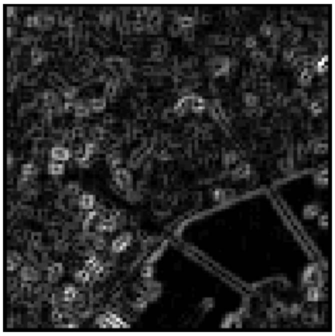


Figure 8-38: Sobel Filter Applied to the New York Image

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 233

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 7, "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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

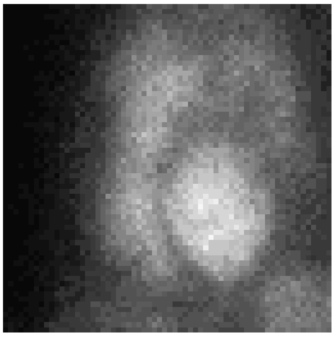


Figure 8-39: Original Gated Blood Pool Image

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.

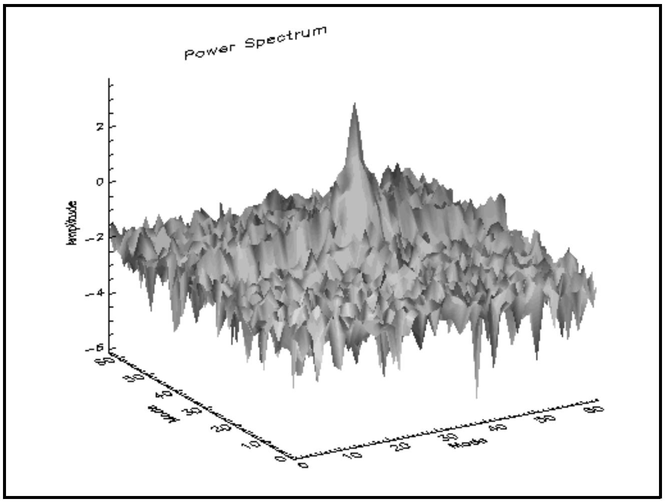


Figure 8-40: Power Spectrum of the Gated Blood Pool Image

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.

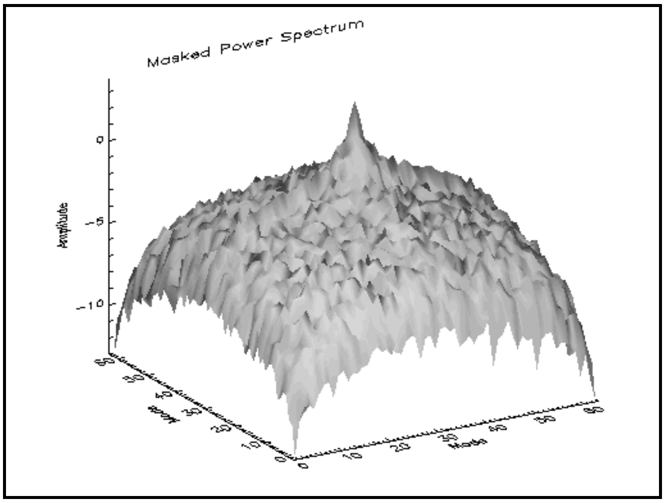


Figure 8-41: Masked Power Spectrum of the Gated Blood Pool Image

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.

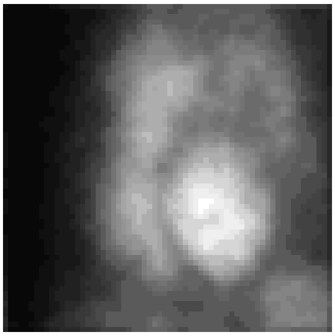


Figure 8-42: Resulting Hanning Filtered Image

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

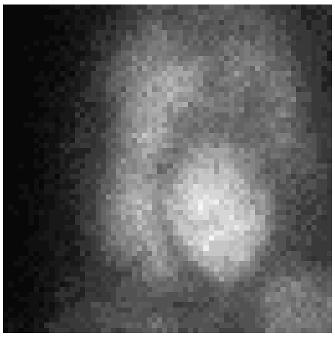


Figure 8-43: Original Gated Blood Pool Image

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.

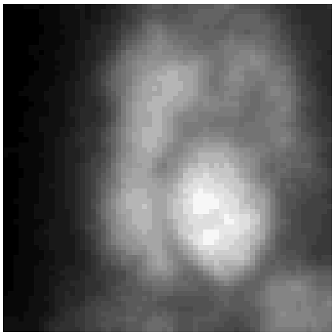
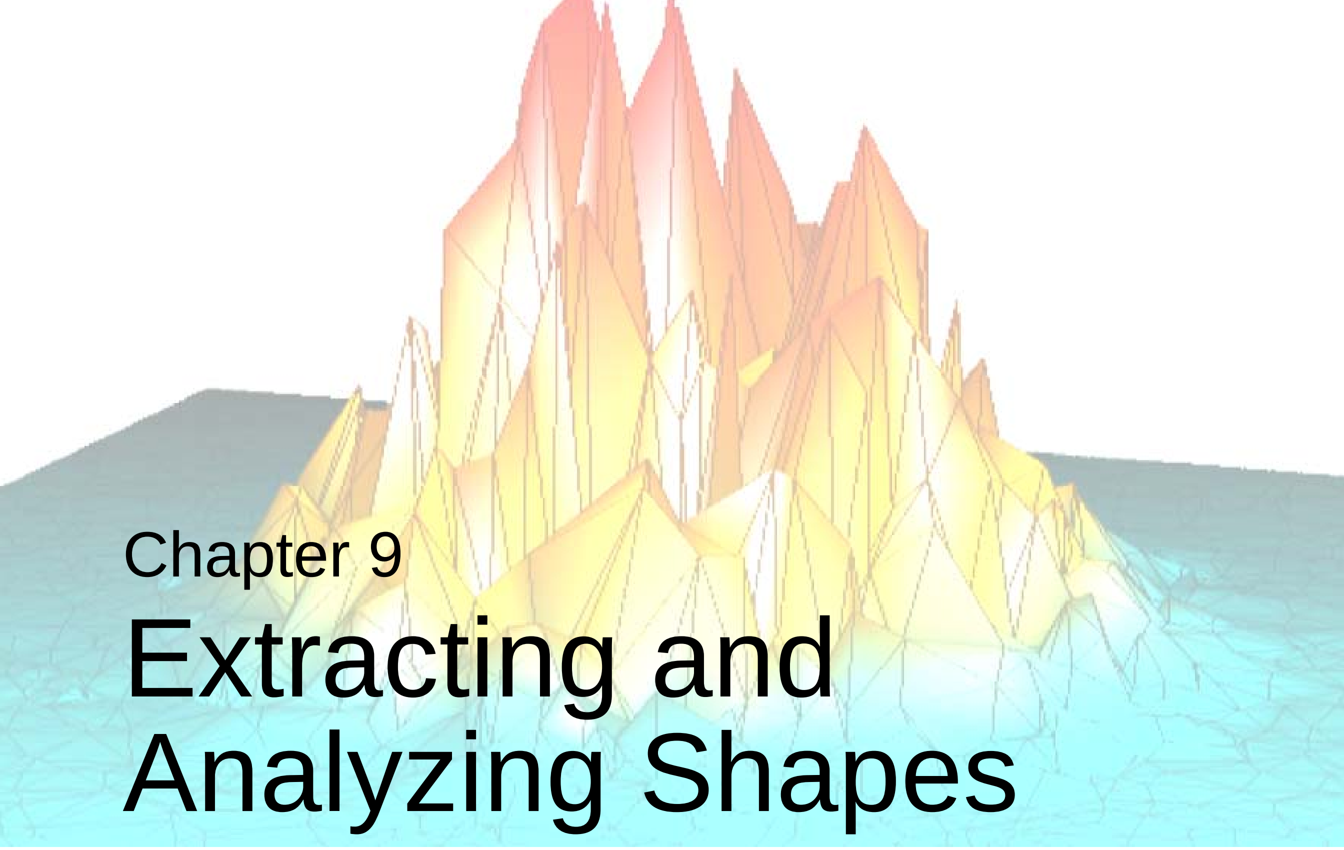


Figure 8-44: Lee Filtered Gated Blood Pool Image



Chapter 9

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 for Threshold and Stretch" on page 243 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 285 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 "Determining Structuring Element Shapes and Sizes" on page 241.

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 246.	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 251.	MORPH_OPEN	Apply an erosion operation followed by a dilation operation to a binary or grayscale image.
"Smoothing with MORPH_CLOSE" on page 254.	MORPH_CLOSE	Apply a dilation operation followed by an erosion operation to a binary or grayscale image.
"Detecting Peaks of Brightness" on page 257.	MORPH_TOPHAT	Retain only the brightest pixels within a grayscale image.
"Creating Image Object Boundaries" on page 260.	WATERSHED	Detect boundaries between similar regions in a grayscale image.
"Selecting Specific Image Objects" on page 264.	MORPH_HITORMISS	Use "hit" and "miss" structures to identify image elements that meet the specified conditions.

Table 9-1: Shape Extraction and Analysis Tasks and Routines

Task	Routine(s)	Description
"Detecting Edges of Image Objects" on page 269.	MORPH_GRADIENT	Subtract an eroded version of a grayscale image from a dilated version of the image, highlighting edges.
"Creating Distance Maps" on page 272.	MORPH_DISTANCE	Estimate for each binary foreground pixel the distance to the nearest background pixel, using a given norm.
"Thinning Image Objects" on page 275.	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 285.	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 9-1: 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 280.

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

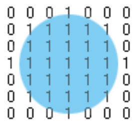
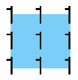
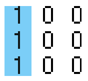
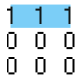
Disk-Shaped Structuring Element
Use SHIFT in conjunction with DIST to create the disk shape.
`radius = 3`
`strucElem = SHIFT(DIST(2*radius+1), radius, $`
`     radius) LE radius`
Change *radius* to alter the size of the structuring element.
	 0 0 0 1 0 0 0
	0 1 1 1 1 1 0
	0 1 1 1 1 1 0
	1 1 1 1 1 1 1
	0 1 1 1 1 1 0
	0 1 1 1 1 1 0
	0 0 0 1 0 0 0

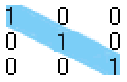
Table 9-2: Creating Various Structuring Elements Shapes with IDL

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.	

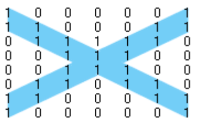
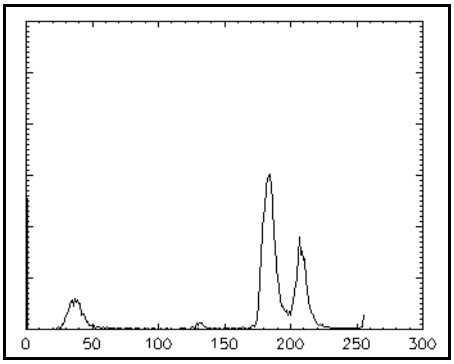
Irregular Structuring Elements
Define custom arrays to create irregular structuring elements or a series of rotations of a single structuring element.
`strucElem = [[1,0,0,0,0,0,1], $`
`             [1,1,0,0,0,1,1], $`
`             [0,1,1,1,1,1,0], $`
`             [0,0,1,1,1,0,0], $`
`             [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 275.
	

Table 9-2: Creating Various Structuring Elements Shapes with IDL

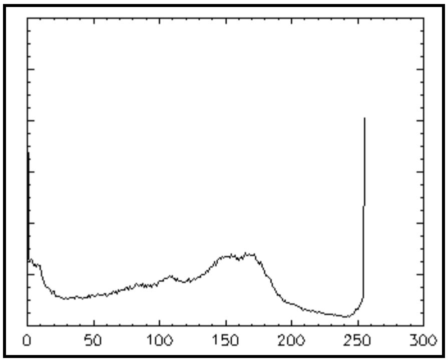
Determining Intensity Values for Threshold and Stretch

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 9-1: 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).

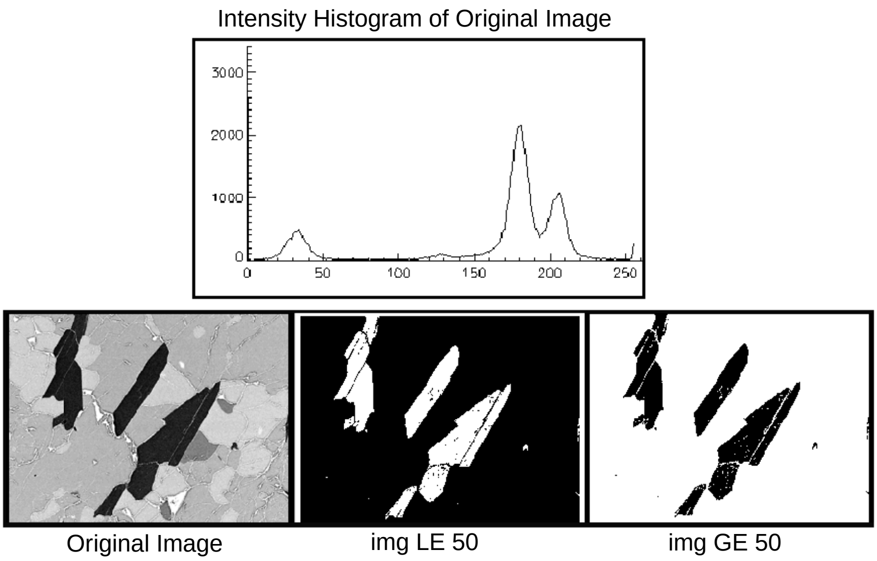


Figure 9-2: Image Thresholding

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).

- `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).

The following figure shows the results of displaying each image stretching statement using `TVSCL, image`:

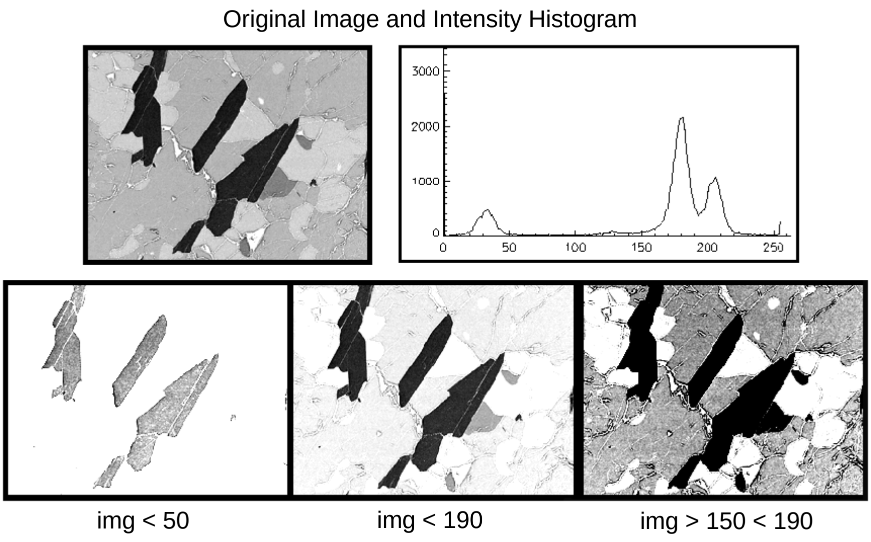


Figure 9-3: Image Stretching

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

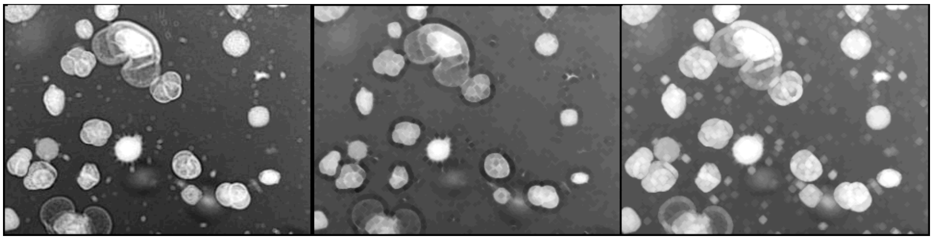


Figure 9-4: Original (left), Eroded (center) and Dilated (right) Grayscale Images

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 for Threshold and Stretch" on page 243.

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.

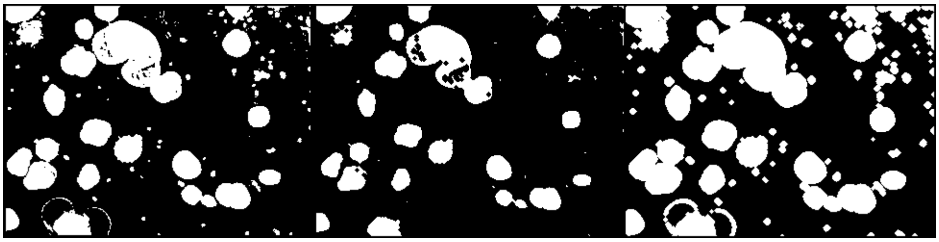


Figure 9-5: Original, Eroded and Dilated Binary Images

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

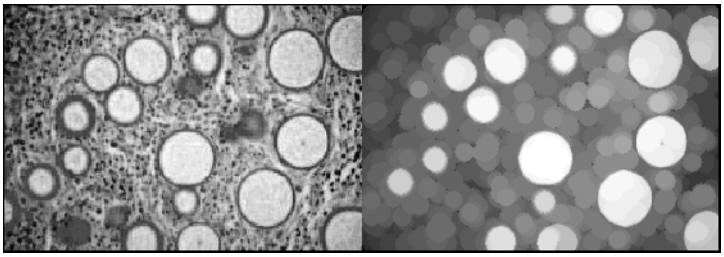


Figure 9-6: Application of the Opening Operation to a Grayscale Image

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 for Threshold and Stretch" on page 243.

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.

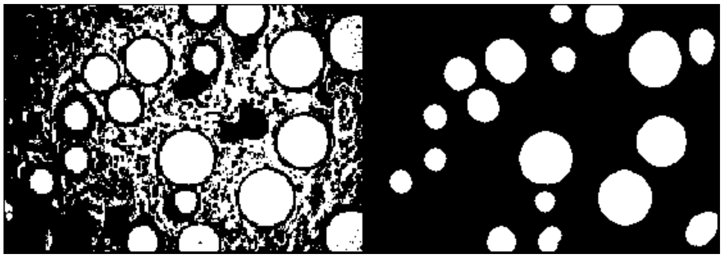


Figure 9-7: Binary Image (left) and Application of the Opening Operator to the Binary Image (right)

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

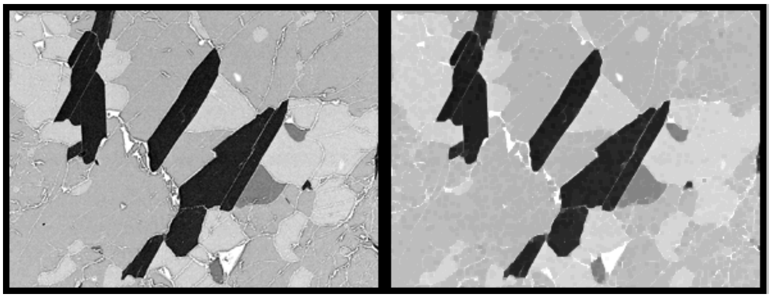


Figure 9-8: Original (left) and Closed Image (right)

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 for Threshold and Stretch" on page 243.

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.

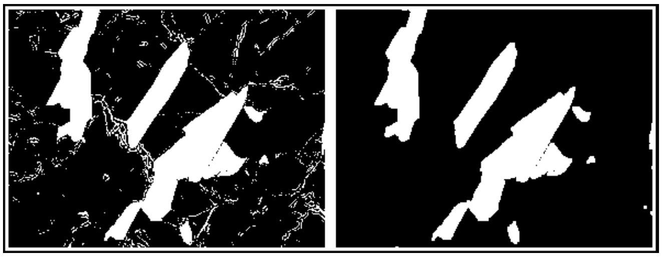


Figure 9-9: Threshold of Original Image (left) and Closed Image (right)

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. Complete the following steps for a detailed description of the process.

Example Code

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

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).

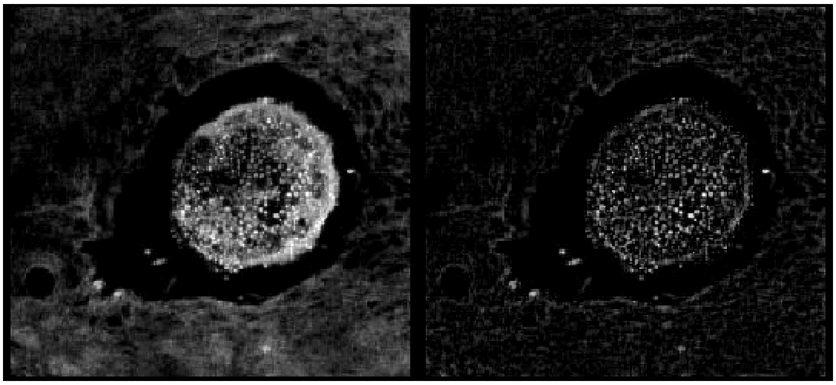


Figure 9-10: Original (left) and Top-hat Image (right)

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 for Threshold and Stretch" on page 243.

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.

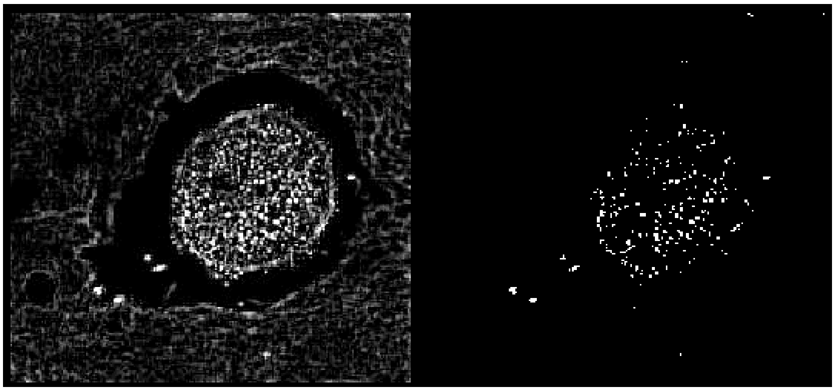


Figure 9-11: Stretched Top-hat Image (left) and Binary Mask (right)

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

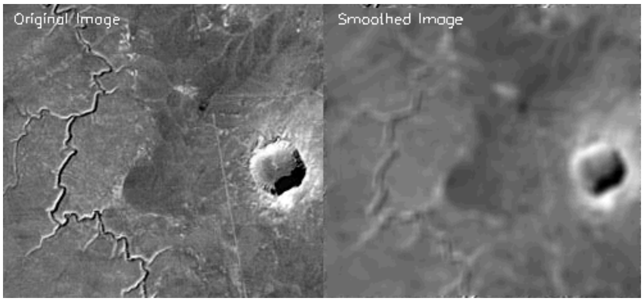


Figure 9-12: Smoothing the Original Image

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 for Threshold and Stretch" on page 243.

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.

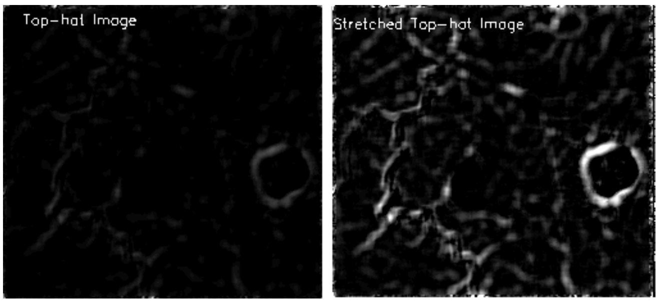


Figure 9-13: Original (left) and Stretched Top-hat Image (right)

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.

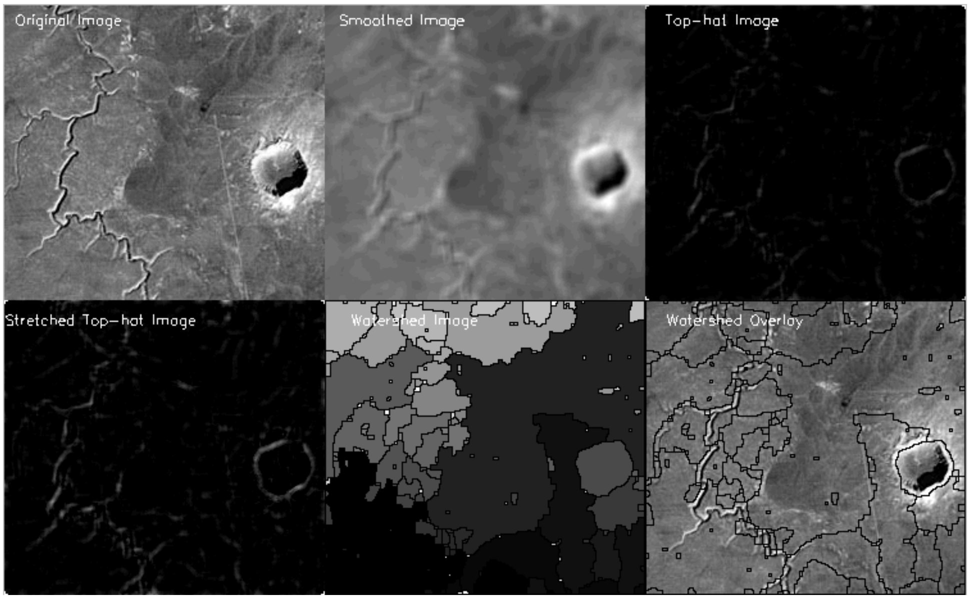


Figure 9-14: Boundaries Defined by the Watershed Operation

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. Complete the following steps for a detailed description of the process.

Example Code

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

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 for Threshold and Stretch" on page 243.

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.

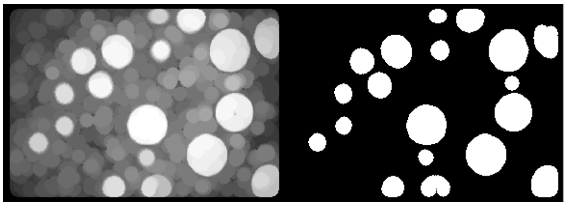


Figure 9-15: Results of Opening (left) and Thresholding (right)

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?"

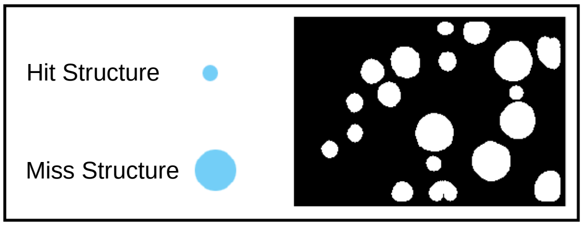


Figure 9-16: Applying the Hit and Miss Structuring Elements to a Binary Image

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).

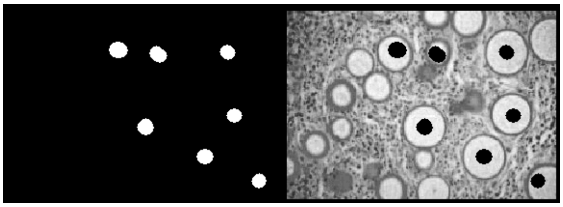


Figure 9-17: Image Elements Matching Hit and Miss Conditions

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.

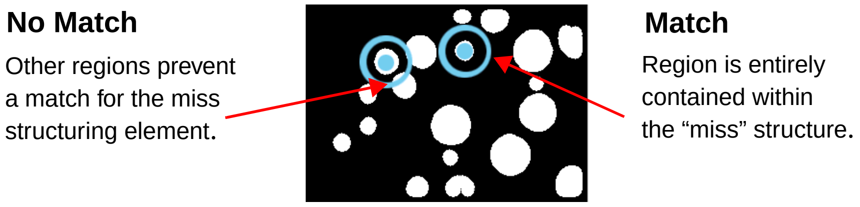

No Match
Other regions prevent a match for the miss structuring element.
Match
Region is entirely contained within the "miss" structure.

Figure 9-18: 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.

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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

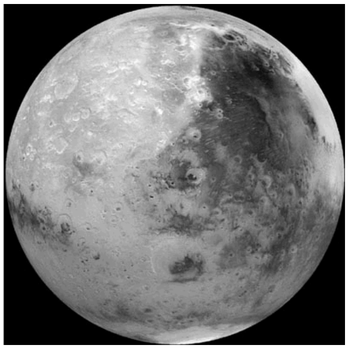


Figure 9-19: Image of Mars Globe

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.

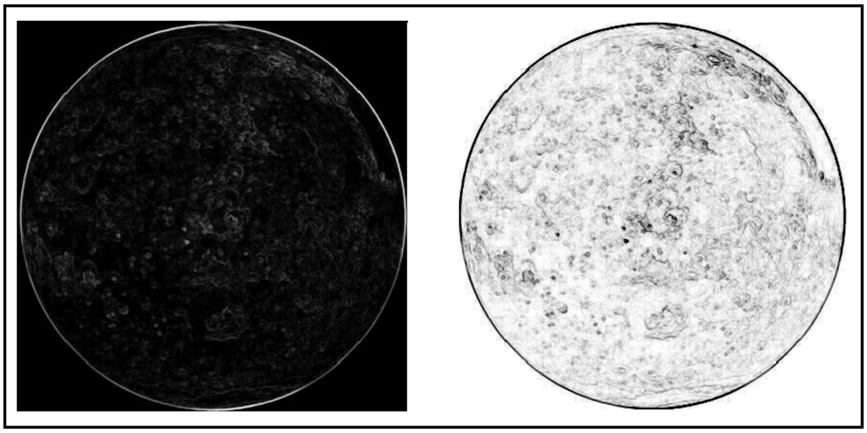


Figure 9-20: Initial and Stretched Results of the Gradient Operation

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. Complete the following steps for a detailed description of the process.

Example Code

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

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 for Threshold and Stretch" on page 243.

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 = padImg LT 120
   WSET, 0
   TVSCL, binaryImg, 1
   ```

   The original image (left) and binary image (right) appear in the following figure.

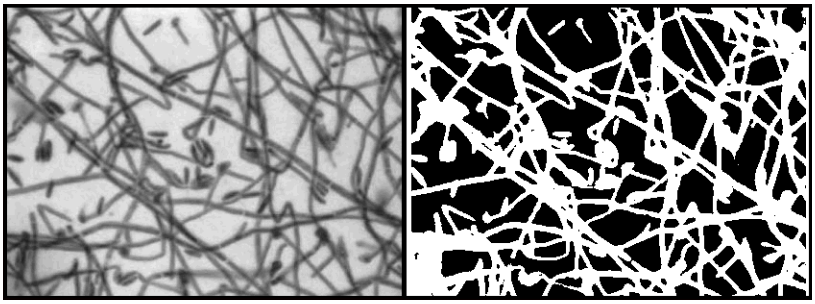


Figure 9-21: Original Image (left) and Binary Image (right)

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.

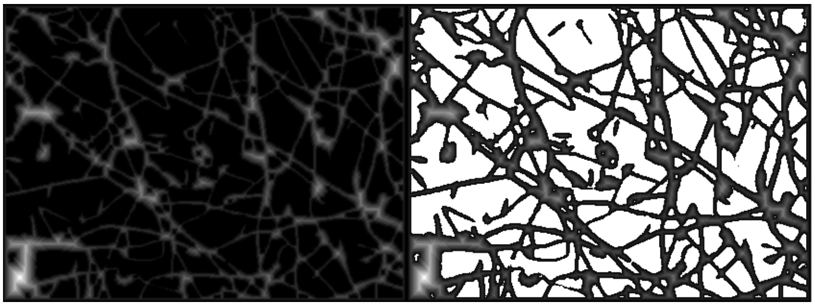


Figure 9-22: Distance Map (left) and Merged Map and Binary Image (right)

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. Complete the following steps for a detailed description of the process.

Example Code

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

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 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 for Threshold and Stretch" on page 243 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 `MorphThinExample.pro` in the `examples/doc/image` subdirectory of the IDL installation directory. This 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 shown below and as described in the beginning of this section, "Thinning Image Objects" on page 275.

```
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 `MorphThinExample.pro` in the `examples/doc/image` subdirectory of the IDL installation directory.

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.

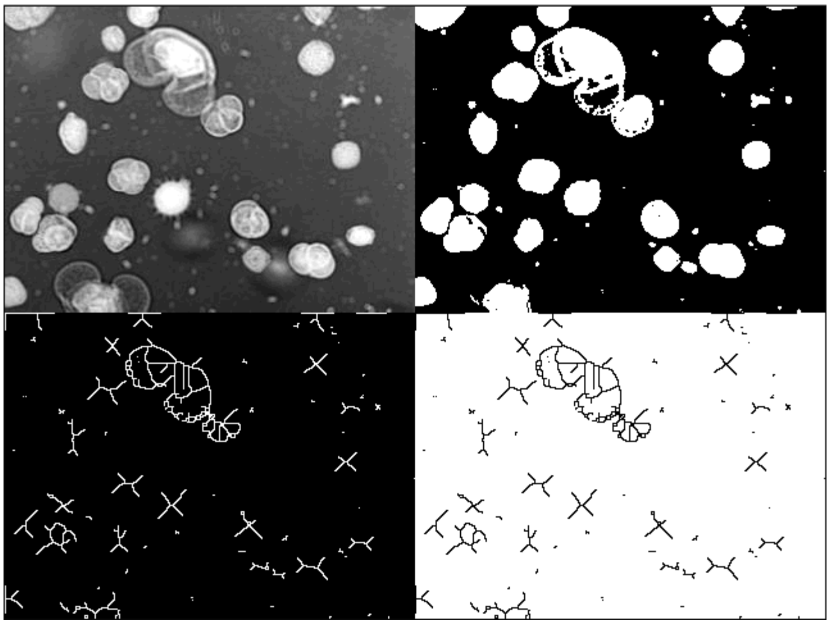


Figure 9-23: Original Image (top left), Binary Image (top right), Thinned Image (bottom left) and Inverse Thinned Image (bottom right)

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.

Combining Morphological Operations

The following example uses a variety of morphological operations to remove bridges from a satellite image of New York waterways. Complete the following steps for a detailed description of the process.

Example Code

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

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
```

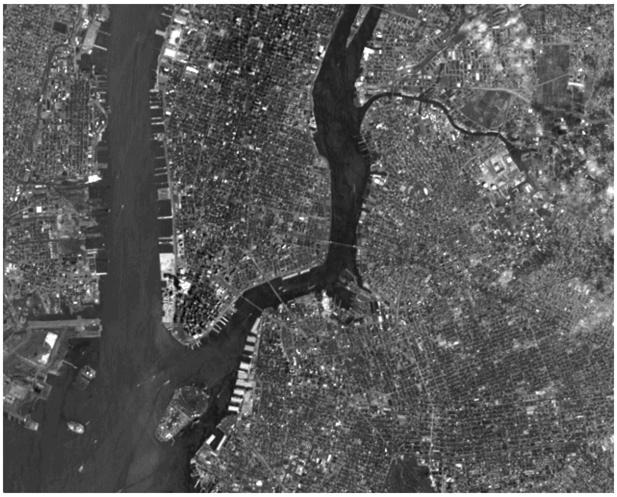


Figure 9-24: Original Image

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 for Threshold and Stretch" on page 243.

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:

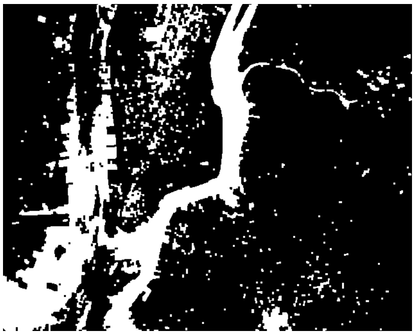


Figure 9-25: Image Mask After Opening and Closing Operations

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.

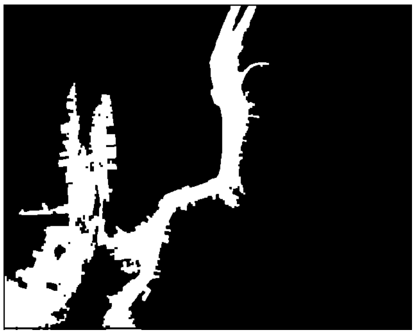


Figure 9-26: Final Image Mask

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.

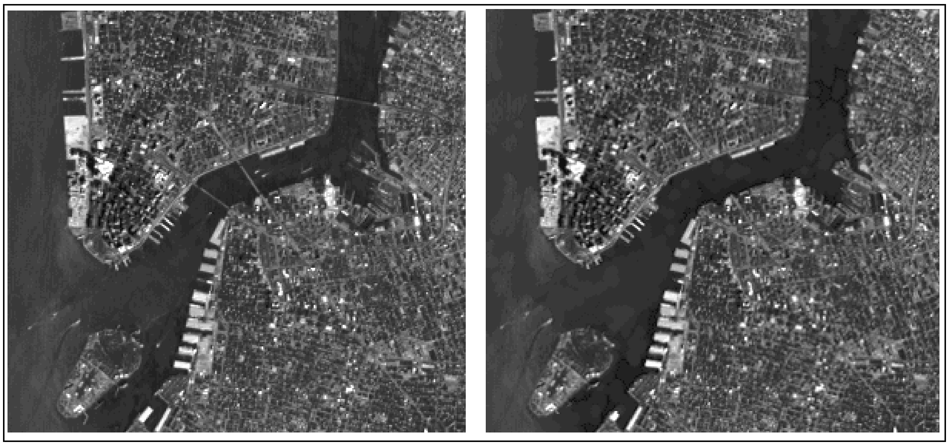


Figure 9-27: Details of Original (left) and Resulting Image of New York (right)

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 289 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. Complete the following steps for a detailed description of the process.

Example Code

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

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.

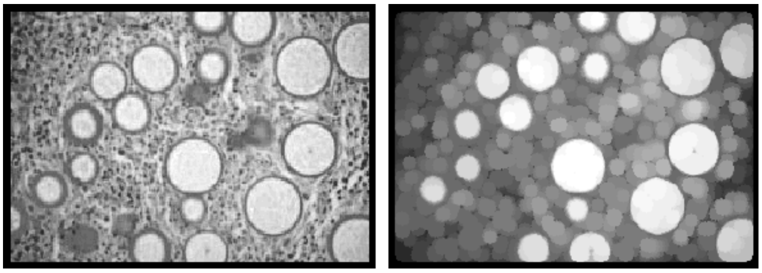


Figure 9-28: Original Image (left) and Application of Opening Operator (right)

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 for Threshold and Stretch" on page 243.

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.

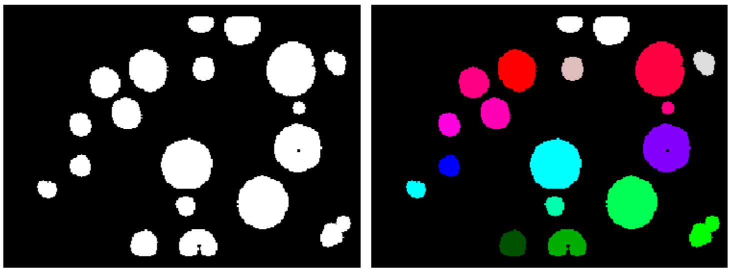


Figure 9-29: Binary Image (left) and Image of Unique Regions (right)

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.

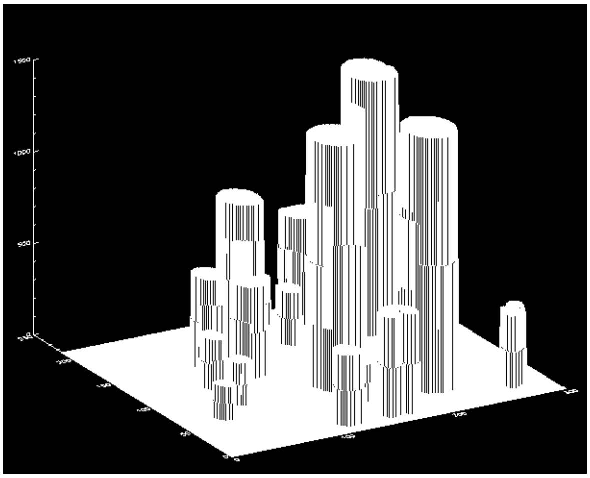


Figure 9-30: Surface Representation of Region Populations

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. Complete the following steps for a detailed description of the process.

Example Code

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

Note

For more information on computing statistics for defined image objects see Chapter 6, "Working with Regions of Interest (ROIs)"

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.

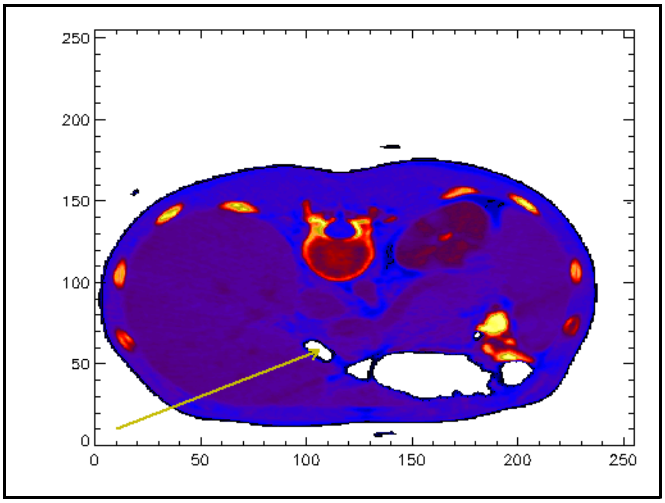


Figure 9-31: Gas Pocket Indicated in CT Scan of Thoracic Cavity

12. Output the resulting coordinate vectors, using TRANSPOSE to print vertical lists of the coordinates:

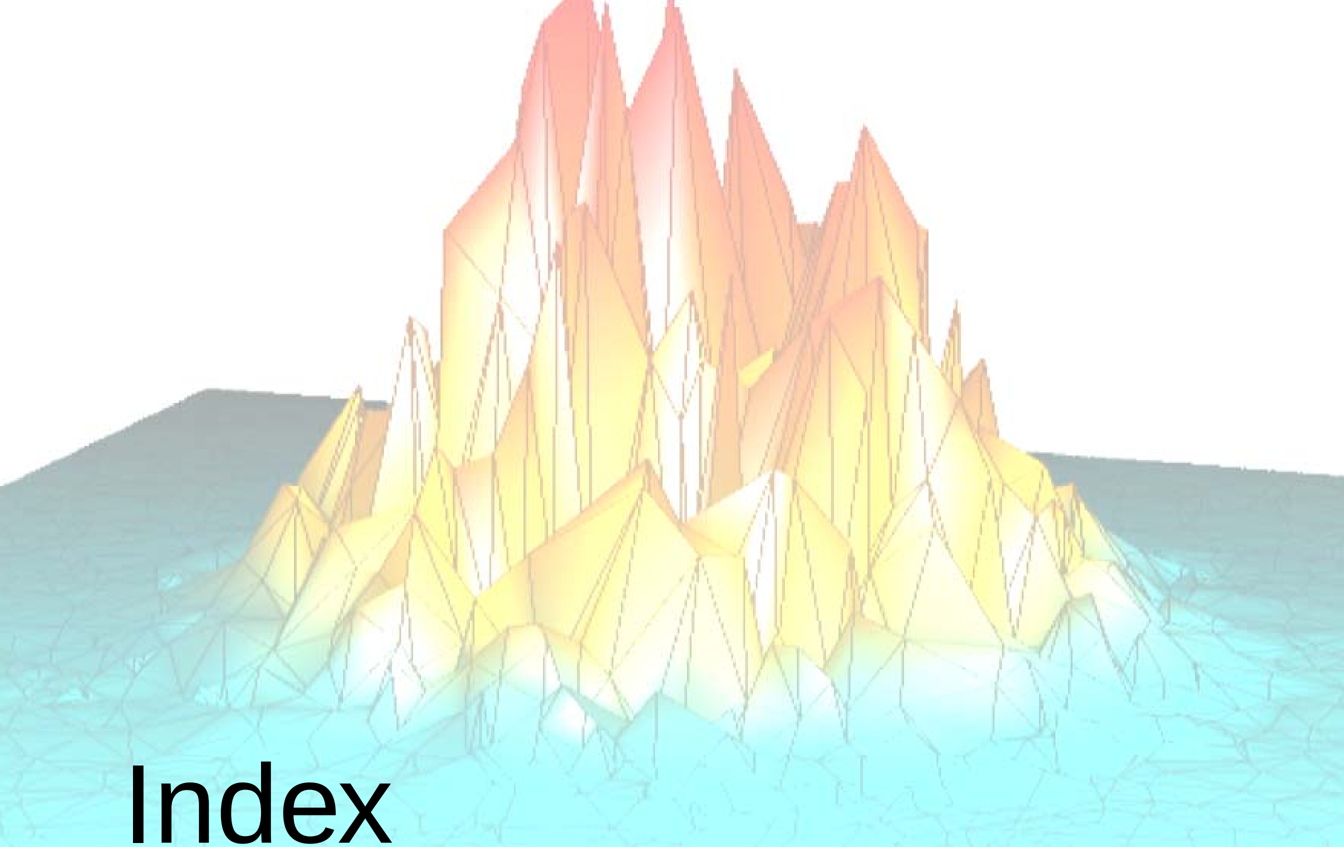
    ```
    PRINT, ''
    PRINT, '       x       ,       y'
    PRINT, [TRANSPOSE(x), TRANSPOSE(y)], FORMAT = '(2F15.6)'
    ```

    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.



Index