

Image Processing Toolbox

For Use with MATLAB®

Computation

Visualization

Programming

User's Guide

Version 2.1

How to Contact The MathWorks:

	508-647-7000	Phone
	508-647-7001	Fax
	The MathWorks, Inc. 24 Prime Park Way Natick, MA 01760-1500	Mail
	http://www.mathworks.com ftp.mathworks.com comp.soft-sys.matlab	Web Anonymous FTP server Newsgroup
	support@mathworks.com suggest@mathworks.com bugs@mathworks.com doc@mathworks.com subscribe@mathworks.com service@mathworks.com info@mathworks.com	Technical support Product enhancement suggestions Bug reports Documentation error reports Subscribing user registration Order status, license renewals, passcodes Sales, pricing, and general information

Image Processing Toolbox User's Guide

© COPYRIGHT 1993 - 1998 by The MathWorks, Inc. All Rights Reserved.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

U.S. GOVERNMENT: If Licensee is acquiring the Programs on behalf of any unit or agency of the U.S. Government, the following shall apply: (a) For units of the Department of Defense: the Government shall have only the rights specified in the license under which the commercial computer software or commercial software documentation was obtained, as set forth in subparagraph (a) of the Rights in Commercial Computer Software or Commercial Software Documentation Clause at DFARS 227.7202-3, therefore the rights set forth herein shall apply; and (b) For any other unit or agency: NOTICE: Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of, the computer software and accompanying documentation, the rights of the Government regarding its use, reproduction, and disclosure are as set forth in Clause 52.227-19 (c)(2) of the FAR.

MATLAB, Simulink, Handle Graphics, and Real-Time Workshop are registered trademarks and Stateflow and Target Language Compiler are trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History: August 1993 First printing Version 1
 May 1997 Second printing Version 2
 January 1998 Revised for MATLAB 5.2 (online version)

Image Credits

moon	Copyright Michael Myers. Used with permission.
cameraman	Copyright Massachusetts Institute of Technology. Used with permission.
trees	<i>Trees with a View</i> , watercolor and ink on paper, copyright Susan Cohen. Used with permission.
forest	Photograph of Carmanah Ancient Forest, British Columbia, Canada, courtesy of Susan Cohen.
circuit	Micrograph of 16-bit A/D converter circuit, courtesy of Steve Decker and Shujaat Nadeem, MIT, 1993.
m83	M83 spiral galaxy astronomical image courtesy of Anglo-Australian Observatory, photography by David Malin.
al umgrns	Copyright J. C. Russ, <i>The Image Processing Handbook</i> , Second Edition, 1994, CRC Press, Boca Raton, ISBN 0-8493-2516-1. Used with permission.
bacteri a	
bl ood1	
bonemarr	
circl es	
circl esm	
debye1	
enamel	
fl owers	
i c	
ngc4024l	
ngc4024m	
ngc4024s	
ri ce	
saturn	
shot1	
testpat1	
testpat2	
text	
tire	

Before You Begin

What is the Image Processing Toolbox?	x
New Features in Version 2	x
Installing the Toolbox	xi
About This Manual	xii
Typographical Conventions	xiii

Introduction

1

Overview	1-2
Images in MATLAB and the Image Processing Toolbox	1-3
Data Types	1-3
Image Types in the Toolbox	1-5
Indexed Images	1-5
Intensity Images	1-6
Binary Images	1-7
RGB Images	1-8
Multiframe Image Arrays	1-9
Limitations	1-10
Working with Image Data	1-11
Reading and Writing Images	1-11
Converting Images to Other Types	1-11
Color Space Conversions	1-13
Working with uint8 Data	1-13
Converting Between Data Types	1-14
Turning the Logical Flag on or off	1-15

Coordinate Systems	1-16
Pixel Coordinates	1-16
Spatial Coordinates	1-17
Using a Nondefault Spatial Coordinate System	1-18

Displaying Images

2

Overview	2-2
Standard Display Techniques	2-3
Displaying Images with imshow	2-3
Preferences	2-3
The truesize Function	2-4
Displaying Indexed Images	2-5
Displaying Intensity Images	2-5
Displaying Binary Images	2-6
Changing the Display Colors	2-7
Displaying RGB Images	2-7
Displaying Images in Graphics Files	2-8
Displaying Nonimage Data as an Intensity Image	2-8
Special Display Techniques	2-10
Adding a Colorbar	2-10
Displaying Multiframe Image Arrays	2-11
Displaying Frames Individually	2-11
Displaying All Frames at Once	2-12
Converting the Array to a Movie	2-13
Displaying Multiple Images	2-14
Displaying Each Image in a Separate Figure	2-14
Displaying Multiple Images in the Same Figure	2-15
Zooming in on a Region of an Image	2-16
Texture Mapping	2-17
Printing Images	2-19

Geometric Operations

3

Overview	3-2
Interpolation	3-3
Image Types	3-4
Image Resizing	3-5
Image Rotation	3-6
Image Cropping	3-7

Neighborhood and Block Operations

4

Overview	4-2
Types of Block Processing Operations	4-2
Sliding Neighborhood Operations	4-4
Padding of Borders	4-5
Linear and Nonlinear Filtering	4-5
Distinct Block Operations	4-8
Overlap	4-9
Column Processing	4-11
Sliding Neighborhoods	4-11
Distinct Blocks	4-12
Restrictions	4-14

Linear Filtering and Filter Design

5

Overview	5-2
Linear Filtering	5-3
Convolution	5-3
Rotating the Convolution Kernel	5-4
Determining the Center Pixel	5-4
Applying the Computational Molecule	5-4
Padding of Borders	5-5
The filter2 Function	5-7
Separability	5-8
Determining Separability	5-9
Higher-Dimensional Convolution	5-9
Using Predefined Filter Types	5-9
Filter Design	5-13
FIR Filters	5-13
Frequency Transformation Method	5-14
Frequency Sampling Method	5-15
Windowing Method	5-16
Creating the Desired Frequency Response Matrix	5-18
Computing the Frequency Response of a Filter	5-19

Transforms

6

Overview	6-2
Fourier Transform	6-3
Definition	6-3
Example	6-4
The Discrete Fourier Transform	6-8
Relationship to the Fourier Transform	6-8
Example	6-9

Applications	6-11
Frequency Response of Linear Filters	6-11
Fast Convolution	6-12
Locating Image Features	6-13
Discrete Cosine Transform	6-15
The DCT Transform Matrix	6-16
The DCT and Image Compression	6-17
Radon Transform	6-19
Using the Radon Transform to Detect Lines	6-22
The Inverse Radon Transform	6-25

Analyzing and Enhancing Images

7

Overview	7-2
Pixel Values and Statistics	7-3
Pixel Selection	7-3
Intensity Profile	7-4
Image Contours	7-7
Image Histogram	7-8
Summary Statistics	7-9
Feature Measurement	7-9
Image Analysis	7-10
Edge Detection	7-10
Quadtree Decomposition	7-11

Image Enhancement	7-14
Intensity Adjustment	7-14
Gamma Correction	7-16
Histogram Equalization	7-18
Noise Removal	7-20
Linear Filtering	7-21
Median Filtering	7-21
Adaptive Filtering	7-23

Binary Image Operations

8

Overview	8-2
Neighborhoods	8-2
Padding of Borders	8-2
Displaying Binary Images	8-3
Morphological Operations	8-4
Dilation and Erosion	8-4
Related Operations	8-7
Predefined Operations	8-8
Object-Based Operations	8-10
4- and 8-Connected Neighborhoods	8-10
Perimeter Determination	8-12
Flood Fill	8-13
Connected-Components Labeling	8-15
Object Selection	8-16
Feature Measurement	8-18
Image Area	8-18
Euler Number	8-19
Lookup-Table Operations	8-20

Region-Based Processing

9

Overview	9-2
Specifying a Region of Interest	9-3
Selecting a Polygon	9-3
Other Selection Methods	9-5
Filtering a Region	9-6
Filling a Region	9-8

Color

10

Overview	10-2
Working with Different Color Depths	10-3
Reducing the Number of Colors in an Image	10-5
Using <code>rgb2ind</code>	10-5
Quantization	10-6
Colormap Mapping	10-7
Using <code>imapprox</code>	10-7
Dithering	10-8
Converting to Other Color Spaces	10-9
NTSC Format	10-9
YCbCr Format	10-10
HSV Format	10-10

Reference

11

Before You Begin

What is the Image Processing Toolbox?	x
New Features in Version 2.1	x
Installing the Toolbox	xi
About This Manual	xii
Typographical Conventions	xiii

What is the Image Processing Toolbox?

The Image Processing Toolbox is a collection of functions that extend the capability of the MATLAB® numeric computing environment. The toolbox supports a wide range of image processing operations, including:

- Geometric operations
- Neighborhood and block operations
- Linear filtering and filter design
- Transforms
- Image analysis and enhancement
- Binary image operations
- Region of interest operations

Many of the toolbox functions are MATLAB M-files, series of MATLAB statements that implement specialized image processing algorithms. You can view the MATLAB code for these functions using the statement:

```
type function_name
```

You can extend the capabilities of the Image Processing Toolbox by writing your own M-files, or by using the toolbox in combination with other toolboxes, such as the Signal Processing Toolbox and the Wavelet Toolbox.

New Features in Version 2.1

Version 2.1 of the Image Processing Toolbox offers a number of advances over versions 1 and 2. For information about the differences between versions 1 and 2, and versions 2 and 2.1, type this command at the MATLAB prompt:

```
helpwin images/Readme
```

For a list of all of the functions in the Image Processing Toolbox, type this command:

```
helpwin images/Contents
```

Installing the Toolbox

To determine if the Image Processing Toolbox is installed on your system, type this command at the MATLAB prompt:

```
ver
```

When you enter this command, MATLAB displays information about the version of MATLAB you are running, including a list of all toolboxes installed on your system and their version numbers.

For information about installing the toolbox, see the *MATLAB Installation Guide* for your platform.

About This Manual

This manual has three main parts:

- Chapters 1 and 2 discuss working with image data and displaying images in MATLAB and the Image Processing Toolbox.
- Chapters 3 to 10 provide in-depth discussion of the concepts behind the software. Each chapter covers a different topic in image processing. For example, Chapter 5 discusses linear filtering, and Chapter 9 discusses binary image operations. Each chapter provides numerous examples that apply the toolbox to representative image processing tasks.
- Chapter 11 gives a detailed reference description of each toolbox function. Reference descriptions include a synopsis of the function's syntax, as well as a complete explanation of options. Many reference descriptions also include examples, a description of the function's algorithm, and references to additional reading material.

All users should read Chapters 1 and 2. Less experienced users will find Chapters 3 to 10 a valuable introduction to image processing, while more experienced users may prefer to use Chapter 11.

Typographical Conventions

To Indicate...	This Manual Uses...	Example
Example code	Monospace type	To assign the value 5 to A, enter: A = 5
MATLAB output	Monospace type	MATLAB responds with: A = 5
Function names	Monospace type	The cos function finds the cosine of each array element.
New terms	<i>Italics</i>	An <i>array</i> is an ordered collection of information.
Keys	Boldface with an initial capital letter	Press the Return key.
Menu names, items, and GUI controls	Boldface with an initial capital letter	Choose the Print option from the File menu.
Mathematical expressions	Variables in <i>italics</i> . Functions, operators, and constants in standard type	This vector represents the polynomial: $p = x^2 + 2x + 3$

Introduction

Overview	1-2
Images in MATLAB and the Image Processing Toolbox	1-3
Data Types	1-3
Image Types in the Toolbox	1-5
Indexed Images	1-5
Intensity Images	1-6
Binary Images	1-7
RGB Images	1-8
Multiframe Image Arrays	1-9
Working with Image Data	1-11
Reading and Writing Images	1-11
Converting Images to Other Types	1-11
Working with uint8 Data	1-13
Coordinate Systems	1-16
Pixel Coordinates	1-16
Spatial Coordinates	1-17

Overview

This chapter introduces you to the fundamentals of image processing using MATLAB and the Image Processing Toolbox. It describes the types of images supported, and how MATLAB represents them. It also explains the basics of working with image data and coordinate systems.

Images in MATLAB and the Image Processing Toolbox

The basic data structure in MATLAB is the *array*, an ordered set of real or complex elements. This object is naturally suited to the representation of *images*, real-valued, ordered sets of color or intensity data. (MATLAB does not support complex-valued images.)

MATLAB stores most images as two-dimensional arrays (i.e., matrices), in which each element of the matrix corresponds to a single *pixel* in the displayed image. (Pixel is derived from *picture element* and usually denotes a single dot on a computer display.) For example, an image composed of 200 rows and 300 columns of different colored dots would be stored in MATLAB as a 200-by-300 matrix.

This convention makes working with images in MATLAB similar to working with any other type of matrix data, and makes the full power of MATLAB available for image processing applications. For example, you can select a single pixel from an image matrix using normal matrix subscripting:

```
I(2, 15)
```

This command returns the value of the pixel at row 2, column 15 of the image I.

Data Types

By default, MATLAB stores most data in arrays of class `double`. The data in these arrays is stored as double precision (64-bit) floating-point numbers. All of MATLAB's functions and capabilities work with these arrays.

For image processing, however, this data representation is not always ideal. The number of pixels in an image may be very large; for example, a 1000-by-1000 image has a million pixels. Since each pixel is represented by at least one array element, this image would require about 8 megabytes of memory.

In order to reduce memory requirements, MATLAB supports storing image data in arrays of class `uint8`. The data in these arrays is stored as 8-bit unsigned integers. Data stored in `uint8` arrays requires one eighth as much memory as data in `double` arrays.

Because the types of values that can be stored in `uint8` arrays and `double` arrays differ, the Image Processing Toolbox uses different conventions for interpreting the values in these arrays. (Noninteger values cannot be stored in

`uint8` arrays, for example, but they can be stored in `double` arrays.) The next section discusses how the toolbox interprets image data, depending on the class of the data array.

In addition to differences in the types of data values they store, `uint8` arrays and `double` arrays differ in the operations that MATLAB supports. See page 1-13 for information about the operations MATLAB supports for `uint8` arrays.

Image Types in the Toolbox

The Image Processing Toolbox supports four basic types of images:

- Indexed images
- Intensity images
- Binary images
- RGB images

This section discusses how MATLAB and the Image Processing Toolbox represent each of these image types.

Indexed Images

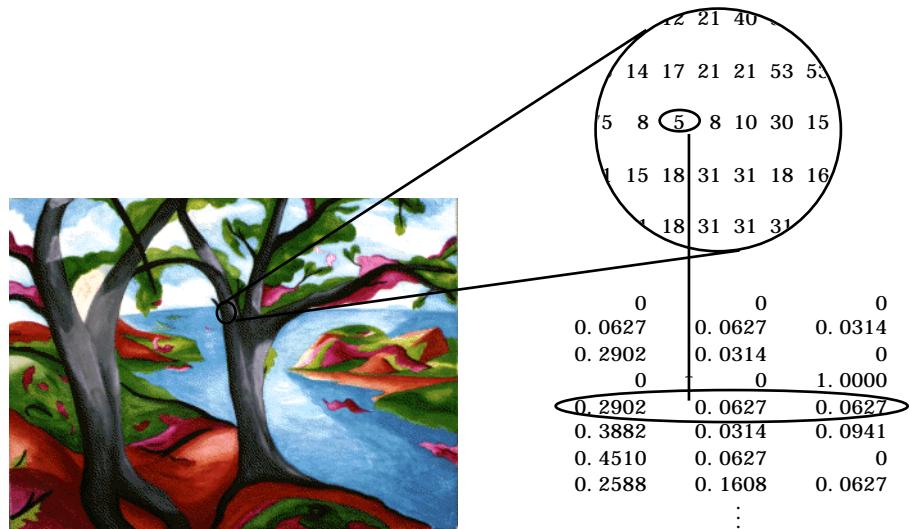
An indexed image consists of two arrays, an image matrix and a colormap. The colormap is an ordered set of values that represent the colors in the image. For each image pixel, the image matrix contains a value that is an index into the colormap.

The colormap is an m -by-3 matrix of class double. Each row of the colormap matrix specifies the red, green, and blue (RGB) values for a single color:

```
color = [R G B]
```

R, G, and B are real scalars that range from 0 (black) to 1.0 (full intensity).

The figure below illustrates the structure of an indexed image. The pixels in the image are represented by integers, which are pointers (indices) to color values stored in the colormap.

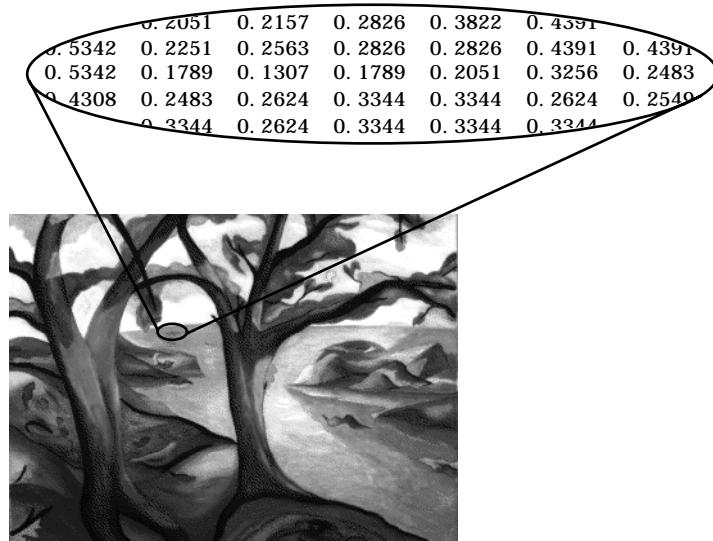


The relationship between the values in the image matrix and the colormap depends on whether the image matrix is of class `double` or `uint8`. If the image matrix is of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. If the image matrix is of class `uint8`, there is an offset; the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on. The `uint8` convention is also used in graphics file formats, and enables 8-bit indexed images to support up to 256 colors. In the image above, the image matrix is of class `double`, so there is no offset. For example, the value 5 points to the fifth row of the colormap.

Intensity Images

MATLAB stores an intensity image as a single matrix, with each element of the matrix corresponding to one image pixel. The matrix can be of class `double`, in which case it contains values in the range [0,1], or of class `uint8`, in which case the data range is [0,255]. The elements in the intensity matrix represent various intensities, or gray levels, where the intensity 0 represents black and the intensity 1 (or 255) represents full intensity, or white.

This figure depicts an intensity image of class `double`.



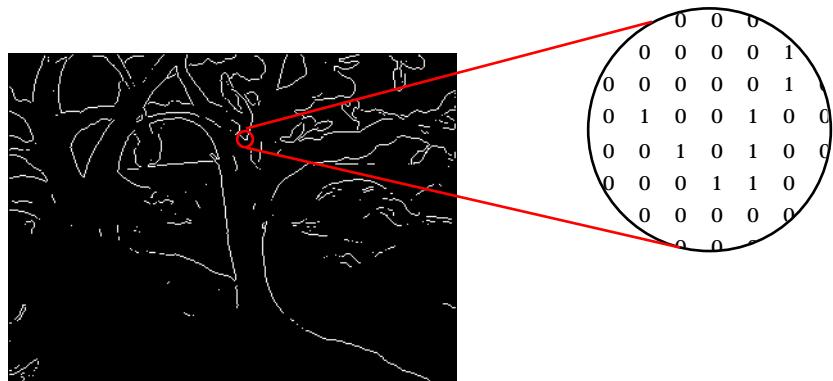
Binary Images

In a binary image, each pixel assumes one of only two discrete values. Essentially, these two values correspond to on and off. A binary image is stored as a two-dimensional matrix of 0's (off pixels) and 1's (on pixels).

A binary image can be considered a special kind of intensity image, containing only black and white. Other interpretations are possible, however; you can also think of a binary image as an indexed image with only two colors.

A binary image can be stored in an array of class `double` or `uint8`. However, a `uint8` array is preferable, because it uses far less memory. In the Image Processing Toolbox, any function that returns a binary image returns it as a `uint8` logical array. The toolbox uses the presence of the logical flag to signify that the data range is [0,1]. (If the logical flag is off, the toolbox assumes the data range is [0,255].)

This figure shows an example of a binary image.



RGB Images

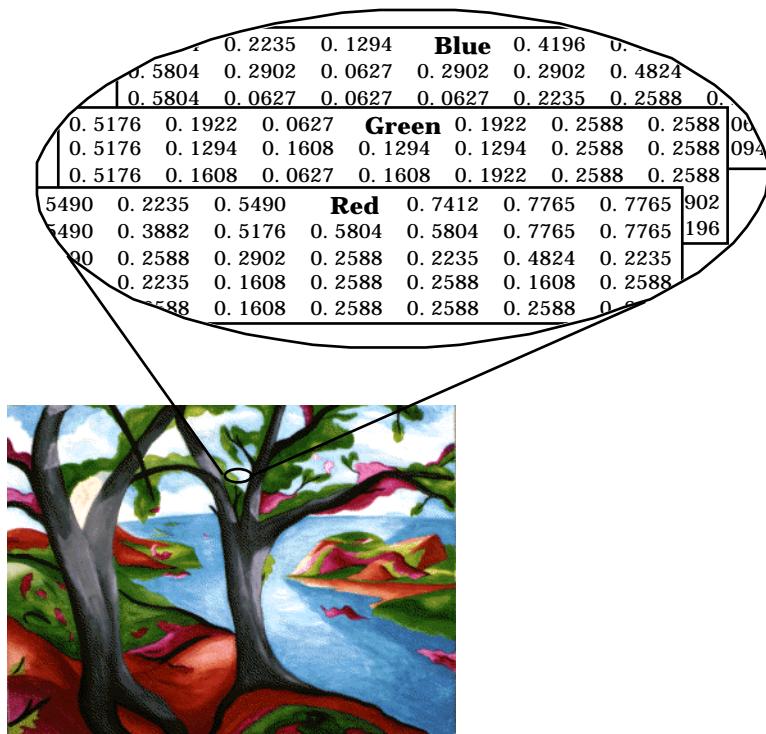
Like an indexed image, an RGB image represents each pixel color as a set of three values, representing the red, green, and blue intensities that make up the color. Unlike an indexed image, however, these intensity values are stored directly in the image array, not indirectly in a colormap.

In MATLAB, the red, green, and blue components of an RGB image reside in a single m -by- n -by-3 array. m and n are the numbers of rows and columns of pixels in the image, and the third dimension consists of three planes, containing red, green, and blue intensity values. For each pixel in the image, the red, green, and blue elements combine to create the pixel's actual color.

For example, to determine the color of the pixel (112,86), look at the RGB triplet stored in (112,86,1:3). Suppose (112,86,1) contains the value 0.1238, (112,86,2) contains 0.9874, and (112,86,3) contains 0.2543. The color for the pixel at (112,86) is:

0.1238 0.9874 0.2543

An RGB array can be of class `double`, in which case it contains values in the range [0,1], or of class `uint8`, in which case the data range is [0,255]. The figure below shows an RGB image of class `double`:



Multiframe Image Arrays

For some applications, you may need to work with collections of images related by time or view, such as magnetic resonance imaging (MRI) slices or movie frames.

The Image Processing Toolbox provides support for storing multiple images in the same array. Each separate image is called a *frame*. If an array holds multiple frames, they are concatenated along the fourth dimension. For example, an array with five 400-by-300 RGB images would be 400-by-300-by-3-by-5. A similar multiframe intensity or indexed image would be 400-by-300-by-1-by-5.

For example, if you have a group of images A1, A2, A3, A4, and A5, you can store them in a single array using:

```
A = cat(4, A1, A2, A3, A4, A5)
```

You can also extract frames from a multiframe image. For example, if you have a multiframe image MULTI, this command extracts the third frame:

```
FRM3 = MULTI (:, :, :, 3)
```

Note that in a multiframe image array, each image must be the same size and have the same number of planes. In a multiframe indexed image, each image must also use the same colormap.

Limitations

Many of the functions in the toolbox operate only on the first two or first three dimensions. You can still use four-dimensional arrays with these functions, but you must process each frame individually. For example, this call displays the seventh frame in the array MULTI:

```
imshow(MULTI (:, :, :, 7))
```

If you pass an array to a function and the array has more dimensions than the function is designed to operate on, your results may be unpredictable. In some cases, the function will simply process the first frame of the array, but in other cases the operation will not produce meaningful results.

See the reference entries in Chapter 11 for information about how individual functions work with the dimensions of an image array. For more information about displaying the images in a multiframe array, see Chapter 2.

Working with Image Data

This section discusses ways of working with the data arrays that represent images, including:

- Reading in image data from files, and writing image data out to files
- Converting images to other image types
- Working with `uint8` arrays in MATLAB and the Image Processing Toolbox

Reading and Writing Images

You can use the MATLAB `imread` function to read image data from files. `imread` can read these graphics file formats:

- BMP
- HDF
- JPEG
- PCX
- TIFF
- XWD

To write image data from MATLAB to a file, use the `imwrite` function. `imwrite` can write the same file formats that `imread` reads.

In addition, you can use the `imfinfo` function to return information about the image data in a file.

See the reference entries for `imread`, `imwrite`, and `imfinfo` in Chapter 11 for more information about these functions.

Converting Images to Other Types

For certain operations, it is helpful to convert an image to a different image type. For example, if you want to filter a color image that is stored as an indexed image, you should first convert it to RGB format. When you apply the filter to the RGB image, MATLAB filters the intensity values in the image, as is appropriate. If you attempt to filter the indexed image, MATLAB simply applies the filter to the indices in the indexed image matrix, and the results may not be meaningful.

The Image Processing Toolbox provides several functions that enable you to convert any image to another image type. These functions have mnemonic names; for example, `i nd2gray` converts an indexed image to a grayscale intensity format.

Note that when you convert an image from one format to another, the resulting image may look different from the original. For example, if you convert a color indexed image to an intensity image, the resulting image is grayscale, not color. For more information about how these functions work, see their reference entries in Chapter 11.

The table below summarizes these image conversion functions:

Function	Purpose
<code>di ther</code>	Create a binary image from a grayscale intensity image by dithering; create an indexed image from an RGB image by dithering
<code>gray2i nd</code>	Create an indexed image from a grayscale intensity image
<code>graysl i ce</code>	Create an indexed image from a grayscale intensity image by thresholding
<code>i m2bw</code>	Create a binary image from an intensity image, indexed image, or RGB image, based on a luminance threshold
<code>i nd2gray</code>	Create a grayscale intensity image from an indexed image
<code>i nd2rgb</code>	Create an RGB image from an indexed image
<code>mat2gray</code>	Create a grayscale intensity image from data in a matrix, by scaling the data
<code>rgb2gray</code>	Create a grayscale intensity image from an RGB image
<code>rgb2i nd</code>	Create an indexed image from an RGB image

You can also perform certain conversions just using MATLAB syntax. For example, you can convert an intensity image to RGB format by concatenating three copies of the original matrix along the third dimension:

```
RGB = cat(3, I, I, I);
```

The resulting RGB image has identical matrices for the red, green, and blue planes, so the image displays as shades of gray.

In addition to these standard conversion tools, there are some functions that return a different image type as part of the operation they perform. For example, the region of interest routines each return a binary image that you can use to mask an indexed or intensity image for filtering or for other operations.

Color Space Conversions

The Image Processing Toolbox represents colors as RGB values, either directly (in an RGB image) or indirectly (in an indexed image). However, there are other methods for representing colors. For example, a color can be represented by its hue, saturation, and value components (HSV). Different methods for representing colors are called *color spaces*.

The toolbox provides a set of routines for converting between RGB and other color spaces. The image processing functions themselves assume all color data is RGB, but you can process an image that uses a different color space by first converting it to RGB, and then converting the processed image back to the original color space. For more information about color space conversion routines, see Chapter 10.

Working with uint8 Data

MATLAB provides limited support for operations on `uint8` arrays. `iimread` reads images into MATLAB as `uint8` arrays, and the `image` and `imagesc` functions can display `uint8` images.

MATLAB also supports these operations on `uint8` arrays:

- Displaying data values
- Indexing into arrays using standard MATLAB subscripting
- Reshaping, reordering, and concatenating arrays, using functions such as `reshape`, `cat`, and `permute`

- Saving to and loading from MAT-files
- The `all` and `any` functions
- Logical operators and indexing
- Relational operators

In addition, MATLAB supports the `find` function for `uint8` arrays, but the returned array is of class `double`.

Aside from the ones listed above, MATLAB does not support most operations for `uint8` arrays. In particular, mathematical operations are not supported. If you attempt to perform an unsupported operation on a `uint8` array, you receive an error. For example:

```
BW3 = BW1 + BW2  
??? Function '+' not defined for variables of class 'uint8'.
```

Most of the functions in the Image Processing Toolbox accept `uint8` input. See the reference entries in Chapter 11 for information about the individual functions.

Converting Between Data Types

If you want to perform operations that are not supported for `uint8` arrays, convert the data to double precision using the `double` function. For example:

```
BW3 = double(BW1) + double(BW2);
```

Keep in mind that converting between data types changes the way MATLAB and the toolbox interpret the image data. If you want the resulting array to be interpreted properly as image data, you need to rescale or offset the data when you convert it.

The toolbox has two functions, `im2double` and `im2uint8`, for converting images from one data type to another. These functions automatically handle the rescaling and offsetting of the original data. For example, this command converts a double-precision RGB image with data in the range [0,1] to a `uint8` RGB image with data in the range [0,255]:

```
RGB2 = im2uint8(RGB1);
```

To convert an indexed image, the input to the conversion function is the image matrix, not the colormap. Colormaps are always of class `double` in MATLAB.

Also, an indexed image cannot be converted from double to uint8 if the image matrix contains values greater than 256.

For more information about `im2double` and `im2uint8`, see the reference entries for those functions in Chapter 11.

Turning the Logical Flag on or off

As discussed on page 1-7, a uint8 binary image must have its logical flag on. If you use `im2uint8` to convert a binary image of type double to uint8, this flag is turned on automatically. If you do the conversion manually, however, you must use the `logical` function to turn on the logical flag. For example:

```
B = logical(uint8(round(A)));
```

To turn the logical flag off, you can use the unary plus operator. For example, if A is a uint8 logical array:

```
B = +A;
```

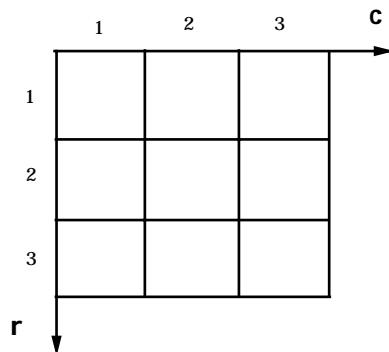
Coordinate Systems

Locations in an image can be expressed in various coordinate systems, depending on context. This section discusses the two main coordinate systems used in the Image Processing Toolbox, and the relationship between them. These systems are:

- The pixel coordinate system
- The spatial coordinate system

Pixel Coordinates

Generally, the most convenient method for expressing locations in an image is to use pixel coordinates. In this coordinate system, the image is treated as a grid of discrete elements, ordered from top to bottom and left to right. For example:



For pixel coordinates, the first component r (the row) increases downward, while the second component c (the column) increases to the right. Pixel coordinates are integer values and range between 1 and the length of the row or column.

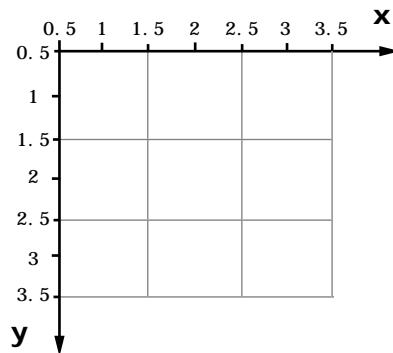
There is a one-to-one correspondence between pixel coordinates and the coordinates MATLAB uses for matrix subscripting. This correspondence makes the relationship between an image's data matrix and the way the image displays easy to understand. For example, the data for the pixel in the fifth row, second column is stored in the matrix element (5,2).

Spatial Coordinates

In the pixel coordinate system, a pixel is treated as a discrete unit, uniquely identified by a single coordinate pair, such as (5,2). From this perspective, a location such as (5.3,2.2) is not meaningful.

At times, however, it is useful to think of a pixel as a square patch, having area. From this perspective, a location such as (5.3,2.2) *is* meaningful, and is distinct from (5,2). In this spatial coordinate system, locations in an image are positions on a plane, and they are described in terms of x and y .

This figure illustrates the spatial coordinate system used for images. Notice that y increases downward:



This spatial coordinate system corresponds quite closely to the pixel coordinate system in many ways. For example, the spatial coordinates of the center point of any pixel are identical to the pixel coordinates for that pixel.

There are some important differences, however. In pixel coordinates, the upper-left corner of an image is (1,1), while in spatial coordinates, this location by default is (0.5,0.5). This difference is due to the pixel coordinate system being discrete, while the spatial coordinate system is continuous. Also, the upper-left corner is always (1,1) in pixel coordinates, but you can specify a nondefault origin for the spatial coordinate system. See “Using a Nondefault Spatial Coordinate System” on page 1-18 for more information.

Another potentially confusing difference is largely a matter of convention: the order of the horizontal and vertical components is reversed in the notation for these two systems. Pixel coordinates are expressed as (r,c), while spatial coordinates are expressed as (x,y). In the reference entries in Chapter 11, when

the syntax for a function uses r and c , it refers to the pixel coordinate system. When the syntax uses x and y , it refers to the spatial coordinate system.

Using a Nondefault Spatial Coordinate System

By default, the spatial coordinates of an image correspond with the pixel coordinates. For example, the center point of the pixel in row 5, column 3 has spatial coordinates $x=3$, $y=5$. (Remember, the order of the coordinates is reversed.) This correspondence simplifies many of the toolbox functions considerably. Several functions primarily work with spatial coordinates rather than pixel coordinates, but as long as you are using the default spatial coordinate system, you can specify locations in pixel coordinates.

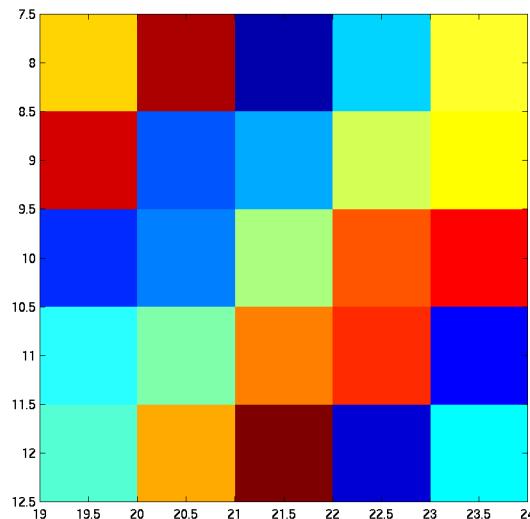
In some situations, however, you may want to use a nondefault spatial coordinate system. For example, you could specify that the upper-left corner of an image is the point $(19.0, 7.5)$, rather than $(0.5, 0.5)$. If you call a function that returns coordinates for this image, the coordinates returned will be values in this nondefault spatial coordinate system.

To establish a nondefault spatial coordinate system, you can specify the `XData` and `YData` image properties when you display the image. These properties are two-element vectors that control the range of coordinates spanned by the image. By default, for an image A , `XData` is `[1 size(A, 2)]`, and `YData` is `[1 size(A, 1)]`.

For example, if A is a 100 row by 200 column image, the default `XData` is `[1 200]`, and the default `YData` is `[1 100]`. The values in these vectors are actually the coordinates for the center points of the first and last pixels (not the pixel edges), so the actual coordinate range spanned is slightly larger; for instance, if `XData` is `[1 200]`, the x -axis range spanned by the image is `[0.5 200.5]`.

These commands display an image using nondefault XData and YData:

```
A = magic(5);  
x = [19.5 23.5];  
y = [8.0 12.0];  
image(A, 'XData', x, 'YData', y), axis image, colormap(jet(25))
```



See the reference entries in Chapter 11 for information about functions in the Image Processing Toolbox that can establish nondefault spatial coordinate systems.

Displaying Images

Overview	2-2
Standard Display Techniques	2-3
Displaying Images with imshow	2-3
Displaying Indexed Images	2-5
Displaying Intensity Images	2-5
Displaying Binary Images	2-6
Displaying RGB Images	2-7
Displaying Images in Graphics Files	2-8
Displaying Nonimage Data as an Intensity Image	2-8
Special Display Techniques	2-10
Adding a Colorbar	2-10
Displaying Multiframe Image Arrays	2-11
Displaying Multiple Images	2-14
Zooming in on a Region of an Image	2-16
Texture Mapping	2-17
Printing Images	2-19

Overview

The Image Processing Toolbox supports a number of image display techniques. For example, the function `imshow` displays any image type with a single function call. Other functions handle more specialized display needs. This chapter describes basic display techniques and discusses topics such as multiple image display and texture mapping. In addition, it includes information about printing images.

Standard Display Techniques

In MATLAB, the primary way to display images is by using the `image` function. This function creates a Handle Graphics® image object, and it includes syntax for setting the various properties of the object. MATLAB also includes the `imagesc` function, which is similar to `image` but which automatically scales the input data.

The Image Processing Toolbox includes an additional display routine, `imshow`. Like `image` and `imagesc`, this function creates a Handle Graphics image object. However, `imshow` also automatically sets various Handle Graphics properties and attributes of the image to optimize the display.

This section discusses displaying images using `imshow`. In general, `imshow` is preferable to `image` and `imagesc` for image processing applications. For information about `image` and `imagesc`, see the online MATLAB Function Reference. These functions are also described in more detail in the *Using MATLAB Graphics* manual.

Displaying Images with `imshow`

When you display an image using the `imshow` function, MATLAB sets figure, axes, and image properties that control the way image data is interpreted. These properties include the image `CData` and `CDataMapping` property, the axes `CLim` property, and the figure `ColorMap` property. In addition, depending on the arguments you specify and the settings of toolbox preferences, `imshow` may also:

- Set other figure and axes properties to tailor the display. For example, axes and tick marks may not display.
- Include or omit a border around the image.
- Call the `truesize` function to display the image without interpolation.

This section describes how you can control the behavior of `imshow`. The sections that follow it discuss how `imshow` displays specific image types.

Preferences

The Image Processing Toolbox includes a function, `iptsetpref`, that you can use to set preferences that affect the behavior of `imshow`:

- The `ImshowBorder` preference controls whether `imshow` leaves a border between the axes containing the image and the edges of the figure.
- The `ImshowAxesVisible` preference controls whether `imshow` displays images with the axes box and tick labels.
- The `ImshowTruesize` preference controls whether `imshow` calls the `truesize` function.

To determine the current value of a preference, use the `iptgetpref` function.

For more information about these and other toolbox preferences, and the values they accept, see the reference entries for `iptsetpref` and `iptgetpref` in Chapter 11.

The `truesize` Function

The `truesize` function assigns a single screen pixel to each image pixel. This is generally the preferred way to display an image.

If you display an image without calling `truesize`, the image displays at the default axis size; MATLAB must use interpolation to determine the values for display pixels that do not correspond to elements in the image matrix. (See Chapter 3 for a discussion of interpolation.)

In most situations, the `imshow` command calls `truesize` automatically. If you do not want `imshow` to call `truesize` (for example, if you display an image with a small number of pixels), you can set the `ImshowTruesize` preference to '`manual`'. For example:

```
iptsetpref('ImshowTruesize', 'manual')
```

You can also override the setting of the `ImshowTruesize` preference for a single `imshow` command. For example, this command prevents `imshow` from calling `truesize`, regardless of the preference setting:

```
imshow(X, map, 'notruesize')
```

For more information about the `imshow` and `truesize` functions, see the reference entries for those functions in Chapter 11.

Displaying Indexed Images

To display an indexed image with `imshow`, you specify both the image matrix and the colormap:

```
imshow(X, map)
```

For each pixel in `X`, `imshow` displays the color stored in the corresponding row of `map`. `imshow` sets the Handle Graphics properties that control how colors display:

- The image `CData` is set to the data in `X`.
- The image `CDataMapping` property is set to `direct`.
- The axes `CLim` property does not apply, because `CDataMapping` is set to `direct`.
- The figure `Colormap` property is set to the data in `map`.

The relationship between the values in the image matrix and the colormap depends on whether the image matrix is of class `double` or `uint8`. If the image matrix is of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. If the image matrix is of class `uint8`, there is an offset; the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on. The offset is handled automatically by the image object, and is not controlled through a Handle Graphics property.

Displaying Intensity Images

To display a grayscale intensity image, the syntax is:

```
imshow(I)
```

`imshow` displays the image by scaling the intensity values to serve as indices into a grayscale colormap. This is done as follows:

- The image `CData` is set to the data in the image matrix.
- The image `CDataMapping` property is set to `scaled`.
- The axes `CLim` property is set to `[0 1]` if the image matrix is of class `double`, or `[0 255]` if the image matrix is of class `uint8`.
- The figure `Colormap` property is set to a grayscale colormap whose values range from black to white.

By default, the number of levels of gray in the colormap is 256 on systems with 24-bit color, and 64 on other systems.

You can explicitly specify the number of gray levels with `imshow`. For example, you can display an image `I` with 32 gray levels using:

```
imshow(I, 32)
```

Note that this is *not* equivalent to:

```
imshow(I, gray(32))
```

If you use the latter syntax, `imshow` interprets the values in `I` as indices into a colormap and therefore does not set `CDataMapping` to `scaled`.

Because MATLAB scales intensity images to fill the colormap range, you can use any size colormap. Larger colormaps allow you to see more detail, but they also use up more color slots. For information about the number of color slots available on your system, see page 2-14.

Displaying Binary Images

To display a binary image, the syntax is:

```
imshow(BW)
```

If the image matrix is of class `double`, `imshow` treats a binary image as an intensity image, setting `CDataMapping` to `scaled`, `CLim` to `[0 1]`, and `Colormap` to a grayscale colormap. This means that the 0 values in the image matrix display as black, and the 1 values display as white.

If the image matrix is of class `uint8`, the behavior depends on whether or not the matrix has its logical flag on:

- If the logical flag is off, `imshow` treats the matrix as a `uint8` intensity image, setting `CLim` to `[0 255]`.
- If the logical flag is on, `imshow` sets `CLim` to `[0 1]`.

In general, this means that a `uint8` binary image must have its logical flag on to display properly. Therefore, all of the toolbox routines that return binary images return them as `uint8` logical matrices.

`imread` reads in an image as a `uint8` logical array of 0's and 1's only if the image is stored in the file with 1 bit per pixel. If a binary image is stored in a file with

4 or 8 bits per pixel, imread interprets this as a grayscale intensity image; the resulting array has values of 0 and 255, and the logical flag is off. You can convert this image to a uint8 logical array using the NOT EQUAL ($\sim=$) operator in MATLAB:

```
BW2 = BW ~= 0;
```

Changing the Display Colors

You may prefer to invert binary images when you display them, so that 0 values display as white and 1 values display as black. To do this, use the NOT (\sim) operator in MATLAB. For example:

```
imshow(~BW)
```

You can also display a binary image using a colormap. If the image is of class uint8, the 0 values display as the first color in the colormap, and the 1 values display as the second color. For example, the following command displays 0 as red and 1 as blue:

```
imshow(BW, [1 0 0; 0 0 1])
```

If the image is of class double, you need to add 1 to each value in the image matrix, because there is no offset in the colormap indexing:

```
imshow(BW + 1, [1 0 0; 0 0 1])
```

Displaying RGB Images

RGB images, also called *truecolor* images, represent color values directly, rather than through a colormap.

To display an RGB image, the syntax is:

```
imshow(RGB)
```

RGB is m-by-n-by-3 array. For each pixel (r, c) in RGB, imshow displays the color represented by the triplet (r, c, 1:3).

As with other image types, MATLAB sets the image CData to the data in the image array. However, in this case CData is a three-dimensional array, rather than a two-dimensional matrix. If CData is three-dimensional, MATLAB interprets the array as truecolor data, and ignores the values of the CDataMapping, CLim, and Colormap properties.

An RGB image array can be of class `double`, in which case it contains values in the range [0,1], or of class `uint8`, in which case the data range is [0,255].

Systems that use 24 bits per screen pixel can display truecolor images directly, because they allocate 8 bits (256 levels) each to the red, green, and blue color planes. On systems with fewer colors, MATLAB displays the image using a combination of color approximation and dithering. See Chapter 10 for more information.

Displaying Images in Graphics Files

Generally, when you display an image, the image data is stored as one or more variables in the MATLAB workspace. However, if you have an image stored in a graphics file that `imread` can read, you can display a file directly using this syntax:

```
imshow filename
```

The file must be in the current directory or on the MATLAB path.

For example, to display a file named `flowers.tif`:

```
imshow flowers.tif
```

This syntax is very useful for scanning through images stored in graphics files. Note, however, that when you use this syntax, the image data is not stored in the MATLAB workspace; if you want to process the image, you need to use `imread` to read in the data, or else use `getimage` to get the image data from the Handle Graphics image object. Also, if you use this syntax with a file containing more than one image, `imshow` displays only the first image.

Displaying Nonimage Data as an Intensity Image

In some cases, you may have data you want to display as an intensity image, even though the data is outside the appropriate range (i.e., [0,1] for `double` arrays or [0,255] for `uint8` arrays). For example, if you filter an intensity image, the output data may not fall in the range of the original data, but you may still want to display the results as an image.

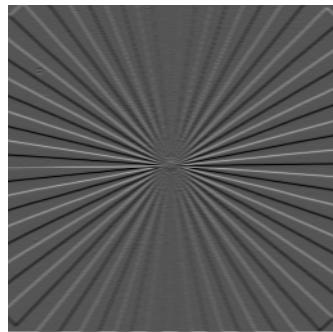
To display this data as an image, you can specify the data range directly, using:

```
imshow(I, [low high])
```

When you use this syntax, `imshow` sets the axes `CLim` property to `[low high]`. `CDataMapping` is always scaled for intensity images, so the image data is scaled so that `low` corresponds to the first row of the grayscale colormap and `high` corresponds to the last row.

If you use an empty matrix (`[]`) for the data range, `imshow` scales the data automatically, setting `low` and `high` to the minimum and maximum values in the array. For example:

```
I = imread('testpat1.tif');
J = filter2([1 2;-1 -2], I);
imshow(J, [])
```



Special Display Techniques

In addition to `imshow`, the toolbox includes functions that perform specialized display operations, or exercise more direct control over the display format. These functions, together with the MATLAB graphics functions, provide a range of image display options.

This section includes the following topics:

- Adding a colorbar to an image display
- Displaying multiframe image arrays
- Displaying multiple images
- Zooming in on a region of an image
- Texture mapping an image onto a surface

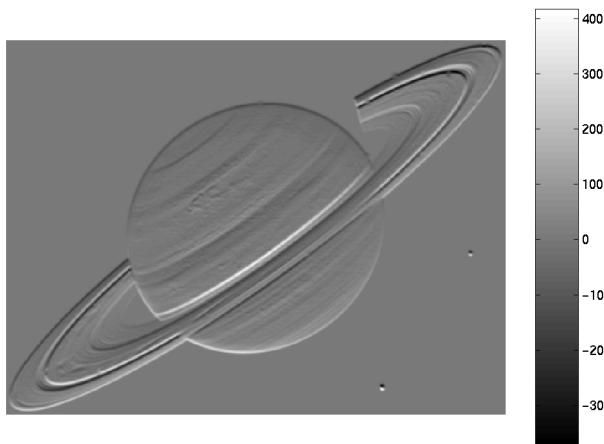
Adding a Colorbar

You can use the `colorbar` function to add a colorbar to an axes object. If you add a colorbar to an axes object that contains an image object, the colorbar indicates the data values that the different colors in the image correspond to.

Seeing the correspondence between data values and the colors displayed is especially useful if you are displaying nonimage data as an image, as described

on page 2-8. In the example below, you filter a grayscale image of class `uint8`, and the resulting data is no longer in the range [0,255]:

```
I = imread('saturn.tif');
h = [1 2 1; 0 0 0; -1 -2 -1];
I2 = filter2(h, I);
imshow(I2, []), colorbar
```



Displaying Multiframe Image Arrays

To view the images in a multiframe array, you can use any of these options:

- Display the frames individually, using the `imshow` function
- Display all of the frames at once, using the `montage` function
- Convert the array to a MATLAB movie, using the `immovie` function

Displaying Frames Individually

To view an individual frame, call `imshow` and specify the frame using standard MATLAB indexing notation. For example, to view the seventh frame in the intensity array `I`:

```
imshow(I (:,:, :, 7))
```

Displaying All Frames at Once

To view all of the frames in a multiframe array at one time, use the `montage` function. `montage` divides a figure into multiple display regions and displays each image in a separate region.

The syntax for `montage` is similar to the `imshow` syntax. To display a multiframe intensity image, the syntax is:

```
montage(I)
```

To display a multiframe indexed image, the syntax is:

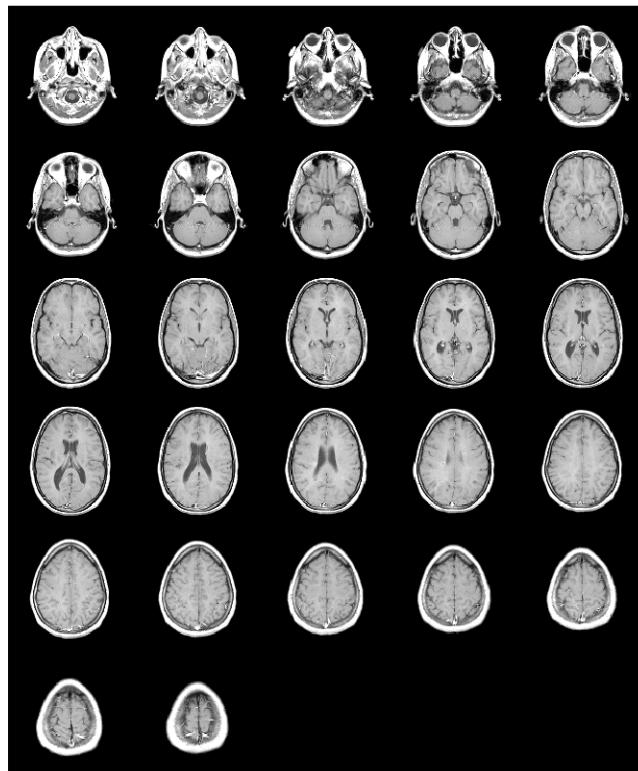
```
montage(X, map)
```

Note that all of the frames in a multiframe indexed array must use the same colormap.

The commands below display a multiframe image array. Notice that `montage` displays images in a row-wise manner. The first frame appears in the first

position of the first row, the next frame in the second position of the first row, and so on. `montage` arranges the frames so that they roughly form a square.

```
load mri  
montage(D, map)
```



Converting the Array to a Movie

To create a MATLAB movie from a multiframe image array, use the `immmovie` function. This function works only with indexed images; if you want to convert another type of image array to a movie, you must first convert it to an indexed image, using one of the conversion functions described in Chapter 1.

This call creates a movie from the multiframe indexed image shown above:

```
mov = immovie(D, map);
```

You can play the movie in MATLAB using the `movie` function:

```
movie(mov)
```

Note that when you play the movie, you need to supply the colormap used by the original image array. For example:

```
mov = imovie(D, map);  
colormap(map), movie(mov)
```

Displaying Multiple Images

MATLAB does not place any restrictions on the number of images you can display simultaneously. However, there are usually system limitations that are dependent on the computer hardware you are using.

The main limitation is the number of colors your system can display. This number depends primarily on the number of bits that are used to store the color information for each pixel. Most systems use either 8, 16, or 24 bits per pixel.

If you are using a system with 16 or 24 bits per pixel, you are unlikely to run into any problems, regardless of the number of images you display. However, if your system uses 8 bits per pixel, it can display a maximum of 256 different colors, so you can quickly run out of color slots if you display multiple images. (Actually, the total number of colors you can display is slightly fewer than 256, because some color slots are reserved for Handle Graphics objects. The operating system usually reserves a few colors as well.)

To determine the number of bits per pixel on your system, enter this command:

```
get(0, 'ScreenDepth')
```

See Chapter 10 for more information about working with different color depths.

This section discusses methods you can use to display multiple images at the same time, and includes information about working around system limitations. This section discusses:

- Displaying each image in a separate figure
- Displaying multiple images in the same figure

Displaying Each Image in a Separate Figure

The simplest way to display multiple images is to display them in different figure windows. `imshow` always displays in the current figure, so if you display

two images in succession, the second image replaces the first image. Therefore, if you want to display two or more images in separate windows, you must explicitly create a new empty figure before calling `imshow` for the second and subsequent images. For example:

```
imshow(I)
figure, imshow(I2)
```

When you use this approach, the figures you create are empty initially, so the images are displayed using `true size`.

If you have an 8-bit display, you must make sure that the total number of colormap entries does not exceed 256. For example, if you try to display three images, each having a different colormap with 128 entries, at least one of the images will display with the wrong colors. (If all three images use the same colormap, there will not be a problem, because only 128 color slots are used.) Note that intensity images are displayed using colormaps, so the color slots used by these images count toward the 256-color total.

One way to avoid these display problems is to manipulate the colormaps to use fewer colors. There are various ways to do this, such as using the `imapprox` function. See Chapter 10 for information about reducing the number of colors used by an image.

Another solution is to convert images to RGB (truecolor) format for display. The colors used by truecolor images do not count toward the 256-color total, because MATLAB automatically uses dithering and color approximation to display these images.

For example, to display an indexed image as an RGB image:

```
imshow(ind2rgb(X, map))
```

To display an intensity image as an RGB image:

```
imshow(cat(3, I, I, I))
```

Displaying Multiple Images in the Same Figure

You can display multiple images in a single figure window by using `imshow` in conjunction with the `subplot` function. This function divides a figure into multiple display regions. The syntax of `subplot` is:

```
subplot(m, n, p)
```

This call divides the figure into an m -by- n matrix of display regions and makes the p -th display region active. For example, you can display two images side by side using:

```
subplot(1, 2, 1), imshow(X1, map)
subplot(1, 2, 2), imshow(X2, map)
```

To display multiple images in the same figure with `imshow`, the images must use the same colormap, as in the example above. If the colormaps differ, the images will not display properly. This is because there is only one colormap for a given figure, and this figure colormap is replaced each time you use `imshow`. For example, if you display a second image in the figure, that image's colormap will become the figure colormap, and the first image will use it as well, even if it is not appropriate.

To avoid this problem, the toolbox provides a function, `subimage`, that you can use instead of `imshow` to display images in subplots. For example, to display two images in the same figure:

```
subplot(1, 2, 1), subimage(X1, map1)
subplot(1, 2, 2), subimage(X2, map2)
```

Both images display correctly, even if they have different colormaps. Generally, `subimage` converts the images to RGB format for display purposes.

Zooming in on a Region of an Image

The `zoom` command enables you to examine the details of an image by using the mouse to zoom in or out. When you zoom in, the figure remains the same size, but only a portion of the image is displayed, at a higher magnification. (`zoom` works by changing the axis limits; it does not change the image data in the figure.)

To turn `zoom` mode on, type:

```
zoom on
```

There are two ways to zoom in on an image:

- Click on a spot in the image by placing the cursor on the spot and pressing the left mouse button. The view changes automatically. The center of the new view is the spot where you clicked.
- Select a region by clicking on the image, holding down the mouse button, and dragging the mouse. A dotted rectangle appears. When you release the mouse button, the region enclosed by the rectangle is displayed.

To zoom out, click on the image with the right mouse button. (If you have a single-button mouse, hold down the **Shift** key and click.) To zoom out completely and restore the original view, enter:

```
zoom out
```

To turn zoom mode off:

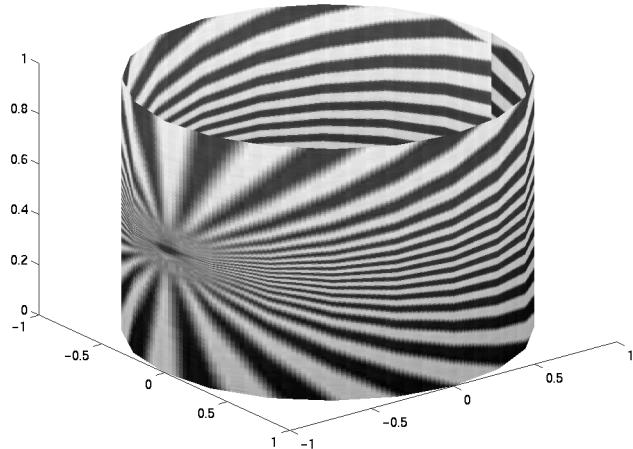
```
zoom off
```

Texture Mapping

When you use the `imshow` command, MATLAB displays the image in a two-dimensional view. However, it is also possible to display an image on a parametric surface, such as a sphere, or below a surface plot. The `warp` function creates these displays by *texture mapping* the image. Texture mapping is a process that maps an image onto a surface grid using bilinear interpolation.

This example texture maps an image of a test pattern onto a cylinder:

```
[x, y, z] = cylinder;  
I = imread('testpat1.tif');  
warp(x, y, z, I);
```



For more information about texture mapping, see the reference entry for the [warp function](#) in Chapter 11.

Printing Images

If you want to output a MATLAB image to use it in another application (such as a word-processing program or graphics editor), use `imwrite` to create a file in the appropriate format.

If you want to print the contents of a MATLAB figure (including nonimage elements such as labels), use the MATLAB `print` command, or choose the **Print** option from the **File** menu of the figure window. If you produce output this way, the results reflect the settings of various Handle Graphics properties. In some cases, you may need to change the settings of certain properties to get the results you want.

Here are some tips that may be helpful when you print images:

- Image colors print as shown on the screen. This means that images do not obey the `InvertHardcopy` property.
- To ensure that printed images have the proper size and aspect ratio, you should set the figure's `PaperPositionMode` property to `auto`. When `PaperPositionMode` is set to `auto`, the width and height of the printed figure are determined by the figure's dimensions on the screen. By default, the value of `PaperPositionMode` is `manual`. If you want the default value of `PaperPositionMode` to be `auto`, enter this line in your startup.m file:

```
set(0, 'DefaultFigurePaperPositionMode', 'auto')
```

For more information about printing and about Handle Graphics, see the *Using MATLAB Graphics* manual.

Geometric Operations

Overview	3-2
Interpolation	3-3
Image Types	3-4
Image Resizing	3-5
Image Rotation	3-6
Image Cropping	3-7

Overview

This chapter describes some basic image processing tools, the geometric functions. These functions modify the geometry of an image by resizing, rotating, or cropping the image. These functions support all image types.

The chapter begins with a discussion of *interpolation*, an operation common to most of the geometric functions. It then discusses each of the geometric functions separately, and shows how to apply them to sample images.

Interpolation

The `imresize` and `imrotate` geometric functions use two-dimensional interpolation as part of the operations they perform. (The `improfile` image analysis function also uses interpolation. See Chapter 7 for information about this function.) Interpolation is the process by which the software resamples the image data to determine values between defined pixels. For example, if you resize an image so it contains more pixels than it did originally, the software obtains values for the additional pixels through interpolation.

The Image Processing Toolbox provides three interpolation methods:

- Nearest neighbor interpolation
- Bilinear interpolation
- Bicubic interpolation

The interpolation methods all work in a fundamentally similar way. In each case, to determine the value for an interpolated pixel, you find the point in the input image that the output pixel corresponds to. You then assign a value to the output pixel by computing a weighted average of some set of pixels in the vicinity of the point. The weightings are based on the distance each pixel is from the point.

The methods differ in the set of pixels that are considered:

- For nearest neighbor interpolation, the output pixel is assigned the value of the pixel that the point falls within. No other pixels are considered.
- For bilinear interpolation, the output pixel value is a weighted average of pixels in the nearest 2-by-2 neighborhood.
- For bicubic interpolation, the output pixel value is a weighted average of pixels in the nearest 4-by-4 neighborhood.

The number of pixels considered affects the complexity of the computation. Therefore the bilinear method takes longer than nearest neighbor interpolation, and the bicubic method takes longer than bilinear. However, the greater the number of pixels considered, the more accurate the effect is, so there is a tradeoff between processing time and quality.

Image Types

The functions that use interpolation take an argument that specifies the interpolation method. For these functions, the default method is nearest neighbor interpolation. This method produces acceptable results for all image types, and is the only method that is appropriate for indexed images. For intensity and RGB images, however, you should generally specify bilinear or bicubic interpolation, because these methods produce better results than nearest neighbor interpolation.

For RGB images, interpolation is performed on the red, green, and blue image planes individually.

For binary images, interpolation has effects that you should be aware of. If you use bilinear or bicubic interpolation, the computed values for the pixels in the output image will not all be 0 or 1. The effect on the resulting output image depends on the class of the input image:

- If the class of the input image is `double`, the output image is a grayscale image of class `double`. The output image is not binary, because it includes values other than 0 and 1.
- If the class of the input image is `uint8`, the output image is a binary image of class `uint8`. The interpolated pixel values are rounded off to 0 and 1 so the output image can be of class `uint8`.

If you use nearest neighbor interpolation, the result is always binary, because the values of the interpolated pixels are taken directly from pixels in the input image.

Image Resizing

The toolbox function `imresize` changes the size of an image using a specified interpolation method. If you do not specify an interpolation method, the function uses nearest neighbor interpolation.

You can use `imresize` to resize an image by a specific magnification factor. To enlarge an image, specify a factor greater than 1. For example, this command doubles the number of pixels in `X` in each direction:

```
Y = imresize(X, 2)
```

To reduce an image, specify a number between 0 and 1 as the magnification factor.

You can also specify the actual size of the output image. This command creates an output image of size 100-by-150:

```
Y = imresize(X, [100 150])
```

If the specified size does not produce the same aspect ratio as the input image has, the output image will be distorted.

If you reduce the image size and use bilinear or bicubic interpolation, `imresize` applies a lowpass filter to the image before interpolation. This reduces the effect of *Moiré patterns*, ripple patterns that result from aliasing during resampling. Note, however, that even with lowpass filtering, the resizing operation can introduce artifacts, because information is always lost when you reduce the size of an image.

`imresize` does not apply a lowpass filter if nearest neighbor interpolation is used, unless you explicitly specify the filter. This interpolation method is primarily used for indexed images, and lowpass filtering is not appropriate for these images.

For information about specifying a different filter, see the reference entry for `imresize` in Chapter 11.

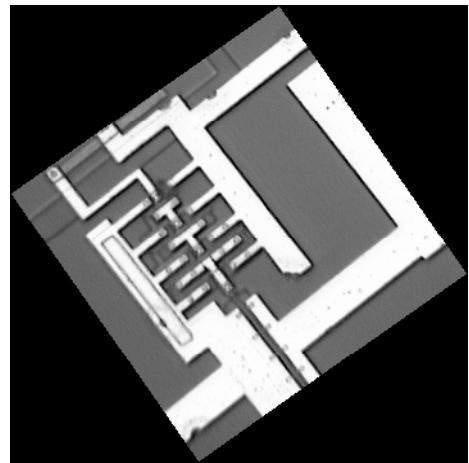
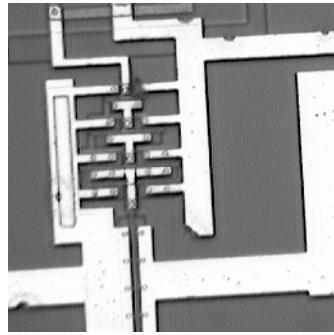
Image Rotation

The `imrotate` function rotates an image, using a specified interpolation method and rotation angle. If you do not specify an interpolation method, the function uses nearest neighbor interpolation.

You specify the rotation angle in degrees. If you specify a positive value, `imrotate` rotates the image counterclockwise; if you specify a negative value, `imrotate` rotates the image clockwise.

For example, these commands rotate an image 35° counterclockwise:

```
I = imread('ic.tif');
J = imrotate(I, 35, 'bilinear');
imshow(I)
figure, imshow(J)
```



Notice that the resulting image is larger than the input image, in order to include the entire original image. `imrotate` adds 0's to the image to fill the area outside the original image.

`imrotate` has an option for cropping the output image to the same size as the input image. See the reference entry for `imrotate` in Chapter 11 for more information.

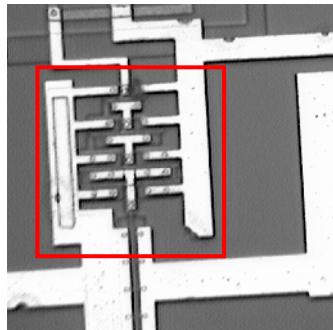
Image Cropping

The function `imcrop` extracts a rectangular portion of an image. You can specify the crop rectangle through input arguments, or select it with a mouse.

If you call `imcrop` without specifying the crop rectangle, the cursor changes to a cross hair when it is over the image. Click on one corner of the region you want to select, and while holding down the mouse button, drag across the image. `imcrop` draws a rectangle around the area you are selecting. When you release the mouse button, `imcrop` creates a new image from the selected region.

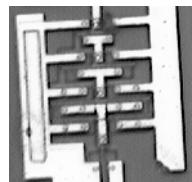
In this example, you display an image and call `imcrop`. The rectangle you select is shown in red.

```
imshow i_c.tif  
I = imcrop;
```



Now display the cropped image:

```
imshow(I)
```



If you do not provide any output arguments, `imcrop` displays the image in a new figure.

Neighborhood and Block Operations

Overview	4-2
Types of Block Processing Operations	4-2
Sliding Neighborhood Operations	4-4
Padding of Borders	4-5
Linear and Nonlinear Filtering	4-5
Distinct Block Operations	4-8
Overlap	4-9
Column Processing	4-11
Sliding Neighborhoods	4-11
Distinct Blocks	4-12

Overview

Certain image processing operations involve processing an image in sections called *blocks*, rather than processing the entire image at once. For example, many linear filtering operations and binary image operations work this way.

The Image Processing Toolbox provides several functions for specific operations that work with blocks; for example, the `dilate` function for binary image dilation. In addition, the toolbox provides more generic functions for processing an image in blocks. This chapter discusses these generic block processing functions.

To use one of the functions described in this chapter, you supply information about the size of the blocks, and specify a separate function to use to process the blocks. The block processing function does the work of breaking the input image into blocks, calling the specified function for each block, and reassembling the results into an output image.

Types of Block Processing Operations

Using these functions, you can perform various block processing operations, including *sliding neighborhood operations* and *distinct block operations*:

- In a sliding neighborhood operation, the input image is processed in a pixelwise fashion. That is, for each pixel in the input image, some operation is performed to determine the value of the corresponding pixel in the output image. The operation is based on the values of a block of neighboring pixels.
- In a distinct block operation, the input image is processed a block at a time. That is, the image is divided into rectangular blocks, and some operation is performed on each block individually to determine the values of the pixels in the corresponding block of the output image.

In addition, the toolbox provides functions for *column processing operations*. These operations are not actually distinct from block operations; instead, they are a way of speeding up block operations by rearranging blocks into matrix columns.

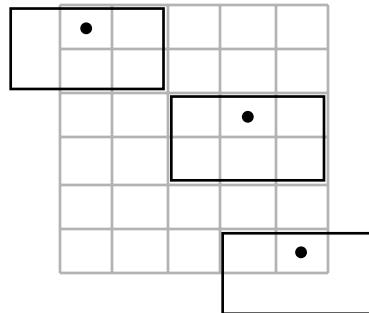
Note that even if you do not use the block processing functions described in this chapter, the information here may be useful to you, as it includes concepts fundamental to many areas of image processing. In particular, the discussion of sliding neighborhood operations is applicable to linear filtering and binary

morphological operations. See Chapter 5 and Chapter 8 for information about these applications.

Sliding Neighborhood Operations

A sliding neighborhood operation is an operation that is performed a pixel at a time, with the value of any given pixel in the output image being determined by applying some algorithm to the values of the corresponding input pixel's *neighborhood*. A pixel's neighborhood is some set of pixels, defined by their locations relative to that pixel, which is called the *center pixel*. The neighborhood is a rectangular block, and as you move from one element to the next in an image matrix, the neighborhood block slides in the same direction.

The figure below shows the neighborhood blocks for some of the elements in a 6-by-5 matrix with 2-by-3 sliding blocks. The center pixel for each neighborhood is marked with a dot.



The center pixel is the actual pixel in the input image being processed by the operation. If the neighborhood has an odd number of rows and columns, the center pixel is actually in the center of the neighborhood. If one of the dimensions has even length, the center pixel is just to the left of center or just above center. For example, in a 2-by-2 neighborhood, the center pixel is the upper left one.

For any m -by- n neighborhood, the center pixel is:

$$\text{floor}(([m\ n]+1)/2)$$

In the 2-by-3 block shown in the figure above, the center pixel is (1,2), the pixel in the second column of the top row of the neighborhood.

A sliding neighborhood operation proceeds like this:

- 1 Select a single pixel.
- 2 Determine the pixel's neighborhood.
- 3 Apply a function to the values of the pixels in the neighborhood. This function must return a scalar.
- 4 Find the pixel in the output image whose position corresponds to that of the center pixel in the input image. Set this output pixel to the value returned by the function.
- 5 Repeat steps 1 through 4 for each pixel in the input image.

For example, suppose the figure above represents an averaging operation. The function might sum the values of the six neighborhood pixels and then divide by 6. The result is the value of the output pixel.

Padding of Borders

As the figure on page 4-4 shows, some of the pixels in a neighborhood may be missing, especially if the center pixel is on the border of the image. Notice that in the figure, the upper left and bottom right neighborhoods include “pixels” that are not part of the image.

To process these neighborhoods, sliding neighborhood operations *pad* the borders of the image, usually with 0’s. In other words, these functions process the border pixels by assuming that the image is surrounded by additional rows and columns of 0’s. These rows and columns do not become part of the output image and are used only as parts of the neighborhoods of the actual pixels in the image.

Linear and Nonlinear Filtering

You can use sliding neighborhood operations to implement many kinds of filtering operations. One example of a sliding neighbor operation is convolution, which is used to implement linear filtering. MATLAB provides the `conv2` and `filter2` functions for performing convolution. See Chapter 5 for more information about these functions.

In addition to convolution, there are many other filtering operations you can implement through sliding neighborhoods. Many of these operations are nonlinear in nature. For example, you can implement a sliding neighborhood

operation where the value of an output pixel is equal to the standard deviation of the values of the pixels in the input pixel's neighborhood.

You can use the `nlfilter` function to implement a variety of sliding neighborhood operations. `nlfilter` takes as input arguments an image, a neighborhood size, and a function that returns a scalar, and returns an image of the same size as the input image. The value of each pixel in the output image is computed by passing the corresponding input pixel's neighborhood to the function. For example, this call computes each output pixel by taking the standard deviation of the values of the input pixel's 3-by-3 neighborhood (that is, the pixel itself and its eight contiguous neighbors):

```
I2 = nlfilter(I, [3 3], 'std2');
```

You can write an M-file to implement a specific function, and then use this function with `nlfilter`. For example, this command processes the matrix `I` in 2-by-3 neighborhoods with a function called `myfun.m`:

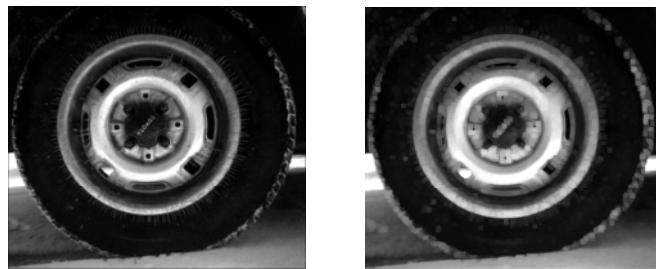
```
nlfilter(I, [2 3], 'myfun');
```

You can also use an inline function; in this case, the function name appears in the `nlfilter` call without quotation marks. For example:

```
f = inline('sqrt(min(x(:)))');
I2 = nlfilter(I, [2 2], f);
```

The example below uses `nlfilter` to set each pixel to the maximum value in its 3-by-3 neighborhood:

```
I = imread('tire.tif');
f = inline('max(x(:))');
I2 = nlfilter(I, [3 3], f);
imshow(I)
figure, imshow(I2)
```

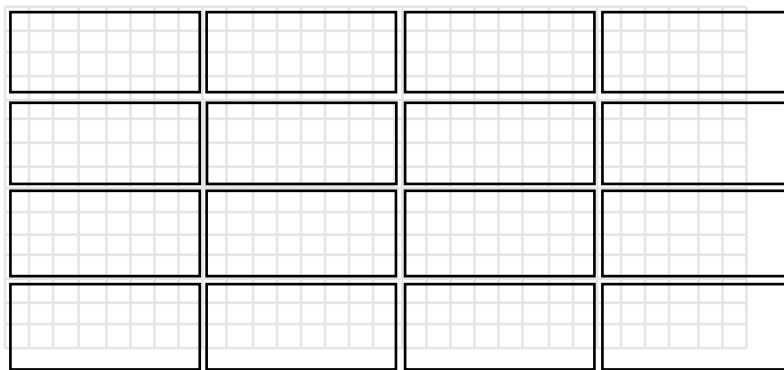


Many operations that `nlfilter` can implement run much faster if the computations are performed on matrix columns rather than rectangular neighborhoods. For information about this approach, see the discussion of the `colfilt` function on page 4-11.

Distinct Block Operations

Distinct blocks are rectangular partitions that divide a matrix into m -by- n sections. Distinct blocks overlay the image matrix starting in the upper-left corner, with no overlap. If the blocks don't fit exactly over the image, the toolbox adds zero padding so that they do.

The figure below shows a 15-by-30 matrix divided into 4-by-8 blocks.



The zero padding process adds 0's to the bottom and right of the image matrix, as needed. After zero padding, the matrix is size 16-by-32.

The function `blkproc` performs distinct block operations. `blkproc` extracts each distinct block from an image and passes it to a function you specify. `blkproc` assembles the returned blocks to create an output image.

For example, the command below processes the matrix `I` in 4-by-6 blocks with the function `myfun`.

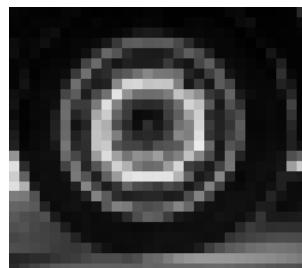
```
I2 = blkproc(I, [4 6], 'myfun');
```

You can specify the function as an inline function; in this case, the function name appears in the `blkproc` call without quotation marks. For example:

```
f = inline('mean2(x)*ones(size(x))');  
I2 = blkproc(I, [4 6], f);
```

The example below uses `blkproc` to set every pixel in each 8-by-8 block of an image matrix to the average of the elements in that block.

```
I = imread('tire.tif');
f = inline('uint8(round(mean2(x)*ones(size(x))))');
I2 = blkproc(I, [8 8], f);
imshow(I)
figure, imshow(I2);
```

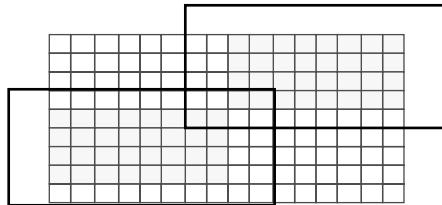


Notice that the inline function computes the mean of the block and then multiplies the result by a matrix of ones, so that the output block is the same size as the input block. As a result, the output image is the same size as the input image. `blkproc` does not require that the images be the same size; however, if this is the result you want, you must make sure that the function you specify returns blocks of the appropriate size.

Overlap

When you call `blkproc`, you can specify an overlap to the distinct blocks; that is, extra rows and columns of pixels outside the block whose values are taken into account when processing the block. When there is an overlap, `blkproc` passes the expanded block (including the overlap) to the specified function.

The figure below shows two 4-by-8 blocks with a 1-by-2 overlap. Notice that there is a 1-row overlap on *each* side of the block, and a 2-row overlap above and below the block.



To specify the overlap, you provide an additional input argument to bl kproc. To process the blocks in the figure above with the function myfun, the call is:

```
B = bl kproc(A, [4 8], [1 2], 'myfun')
```

Overlap often increases the amount of zero padding needed. For example, in the figure on page 4-8, the original 15-by-30 matrix becomes 16-by-32 with zero padding. If a 1-by-2 overlap is used, the padded matrix is 18-by-36.

Column Processing

The toolbox provides functions that you can use to process sliding neighborhoods or distinct blocks as columns. This approach is useful for operations that MATLAB performs columnwise; in many cases, column processing can reduce the execution time required to process an image.

For example, suppose the operation you are performing involves computing the mean of each block. This computation is much faster if you first rearrange the blocks into columns, because you can compute the mean of every column with a single call to the `mean` function, rather than calling `mean` for each block individually.

You can use the `colfilt` function to implement column processing. This function:

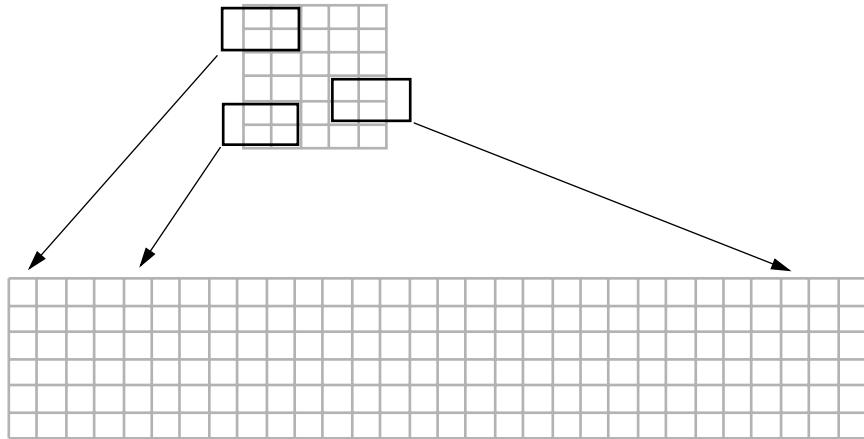
- 1 Reshapes each sliding or distinct block of an image matrix into a column in a temporary matrix
- 2 Passes the temporary matrix to a function you specify
- 3 Rearranges the resulting matrix back into the original shape

Sliding Neighborhoods

For a sliding neighborhood operation, `colfilt` creates a temporary matrix that has a separate column for each pixel in the original image. The column corresponding to a given pixel contains the values of that pixel's neighborhood from the original image.

The figure below illustrates this process. In this figure, a 6-by-5 image matrix is processed in 2-by-3 neighborhoods. `colfilt` creates one column for each pixel in the image, so there are a total of 30 columns in the temporary matrix. Each pixel's column contains the value of the pixels in its neighborhood, so there are six rows. `colfilt` zero pads the input image as necessary. For

example, the neighborhood of the upper left pixel in the figure has two zero-valued neighbors, due to zero padding.



The temporary matrix is passed to a function, which must return a single value for each column. (Many MATLAB functions work this way; for example, `mean`, `median`, `std`, `sum`, etc.) The resulting values are then assigned to the appropriate pixels in the output image.

`colfilt` can produce the same results as `nfilter` with faster execution time; however, it may use more memory. The example below sets each output pixel to the maximum value in the input pixel's neighborhood, producing the same result as the `nfilter` example on page 4-7. Notice that the function is `max(x)` rather than `max(x(:))`, because each neighborhood in the original image is a separate column in the temporary matrix.

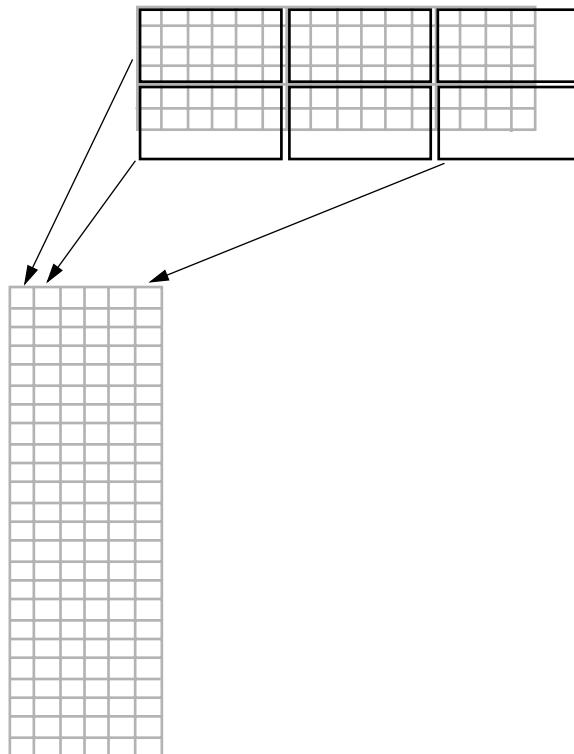
```
f = inline('max(x)');
I2 = colfilt(I, [3 3], 'sliding', f);
```

Distinct Blocks

For a distinct block operation, `colfilt` creates a temporary matrix by rearranging each block in the image into a column. `colfilt` pads the original image with 0's, if necessary, before creating the temporary matrix.

The figure below illustrates this process. In this figure, a 6-by-16 image matrix is processed in 4-by-6 blocks. `colfilt` first zero pads the image to make the size

8-by-18 (six 4-by-6 blocks), and then rearranges the blocks into 6 columns of 24 elements each.



After rearranging the image into a temporary matrix, `colfilt` passes this matrix to the function. The function must return a matrix of the same size as the temporary matrix. If the block size is m -by- n , and the image is mm -by- nn , the size of the temporary matrix is $(m*n)$ -by- $(ceil(mm/m) * ceil(nn/n))$. After the function processes the temporary matrix, the output is rearranged back into the shape of the original image matrix.

This example sets all the pixels in each 8-by-8 block of an image to the mean pixel value for the block, producing the same result as the `blkproc` example on page 4-9:

```
f = inline('ones(64, 1)*mean(x)');
I2 = colfilt(I, [8 8], 'distinct', f);
```

Notice that the inline function computes the mean of the block and then multiplies the result by a vector of ones, so that the output block is the same size as the input block. As a result, the output image is the same size as the input image.

Restrictions

You can use `colfilt` to implement many of the same distinct block operations that `blkproc` performs. However, `colfilt` has certain restrictions that `blkproc` does not:

- The output image must be the same size as the input image.
- The blocks cannot overlap.

For situations that do not satisfy these constraints, use `blkproc`.

Linear Filtering and Filter Design

Overview	5-2
Linear Filtering	5-3
Convolution	5-3
Padding of Borders	5-5
The filter2 Function	5-7
Separability	5-8
Higher-Dimensional Convolution	5-9
Using Predefined Filter Types	5-9
Filter Design	5-13
FIR Filters	5-13
Frequency Transformation Method	5-14
Frequency Sampling Method	5-15
Windowing Method	5-16
Creating the Desired Frequency Response Matrix	5-18
Computing the Frequency Response of a Filter	5-19

Overview

The Image Processing Toolbox provides a number of functions for designing and implementing two-dimensional linear filters for image data. This chapter describes these functions and how to use them effectively.

The material in this chapter is divided into two parts:

- The first part is an explanation of linear filtering and how it is implemented in the toolbox. This topic describes filtering in terms of the spatial domain, and is accessible to anyone doing image processing.
- The second part is a discussion about designing two-dimensional Finite Infinite Response (FIR) filters. This section assumes you are familiar with working in the frequency domain.

Linear Filtering

Filtering is a technique for modifying or enhancing an image. For example, you can filter an image to emphasize certain features or remove other features.

Filtering is a *neighborhood operation*, in which the value of any given pixel in the output image is determined by applying some algorithm to the values of the pixels in the neighborhood of the corresponding input pixel. A pixel's neighborhood is some set of pixels, defined by their locations relative to that pixel. (For a general discussion of neighborhood operations, see Chapter 4.)

Linear filtering is filtering in which the value of an output pixel is a linear combination of the values of the pixels in the input pixel's neighborhood. For example, an algorithm that computes a weighted average of the neighborhood pixels is one type of linear filtering operation.

This section discusses linear filtering in MATLAB and the Image Processing Toolbox. It includes:

- A description of how MATLAB performs linear filtering, using convolution
- A discussion about using predefined filter types

For information about how to design filters, see page 5-13.

Convolution

In MATLAB, linear filtering of images is implemented through two-dimensional *convolution*. In convolution, the value of an output pixel is computed by multiplying elements of two matrices and summing the results. One of these matrices represents the image itself, while the other matrix is the filter. For example, a filter might be:

$$\begin{bmatrix} 4 & -3 & 1 \\ 4 & 6 & 2 \end{bmatrix}$$

This filter representation is known as a *convolution kernel*. The MATLAB function conv2 implements image filtering by applying your convolution kernel to an image matrix. conv2 takes as arguments an input image and a filter, and returns an output image. For example, in this call, k is the convolution kernel, A is the input image, and B is the output image:

$$B = \text{conv2}(A, k);$$

`conv2` produces the output image by performing these steps:

- 1** Rotate the convolution kernel 180 degrees to produce a computational molecule.
- 2** Determine the center pixel of the computational molecule.
- 3** Apply the computational molecule to each pixel in the input image.

Each of these steps is explained below.

Rotating the Convolution Kernel

In two-dimensional convolution, the computations are performed using a *computational molecule*. This is simply the convolution kernel rotated 180 degrees, as in this call:

```
h = rot90(k, 2);
```

```
h =
```

```
2   6   4  
1  -3   4
```

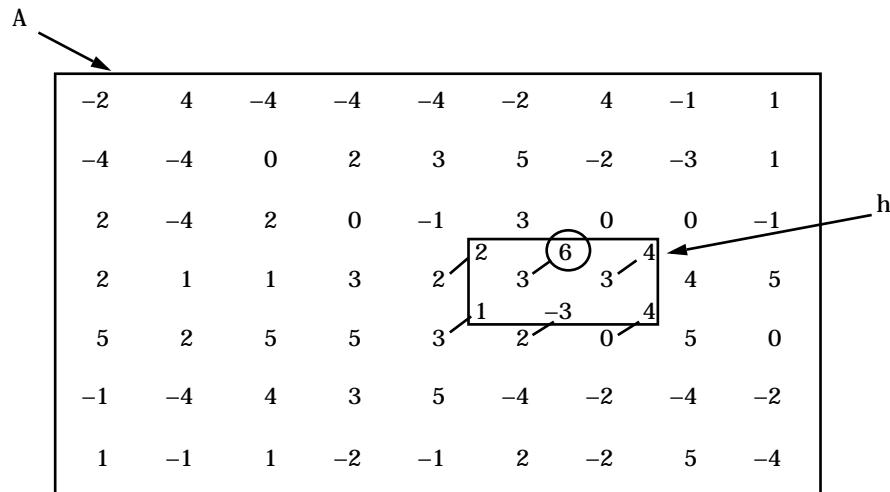
Determining the Center Pixel

To apply the computational molecule, you must first determine the *center pixel*. The center pixel is defined as $floor((size(h)+1)/2)$. For example, in a 5-by-5 molecule, the center pixel is (3,3). The molecule `h` shown above is 2-by-3, so the center pixel is (1,2).

Applying the Computational Molecule

The value of any given pixel in `B` is determined by applying the computational molecule `h` to the corresponding pixel in `A`. You can visualize this by overlaying `h` on `A`, with the center pixel of `h` over the pixel of interest in `A`. You then multiply each element of `h` by the corresponding pixel in `A`, and sum the results.

For example, to determine the value of the pixel (4,6) in B, overlay h on A, with the center pixel of h covering the pixel (4,6) in A. The center pixel is circled in this figure:



Now, look at the six pixels covered by h. For each of these pixels, multiply the value of the pixel by the value in h. Sum the results, and place this sum in B(4, 6) :

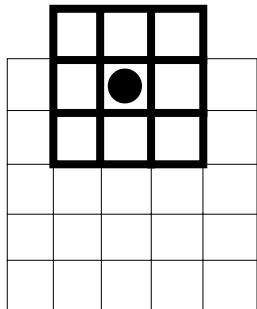
$$B(4, 6) = 2*2 + 3*6 + 3*4 + 3*1 + 2*-3 + 0*4 = 31$$

Perform this procedure for each pixel in A to determine the value of each corresponding pixel in B.

Padding of Borders

When you apply a filter to pixels on the borders of an image, some of the elements of the computational molecule may not overlap actual image pixels. For example, if the molecule is 3-by-3 and you are computing the result for a pixel on the top row of the image, some of the elements of the molecule are outside the border of the image.

This figure illustrates a 3-by-3 computational molecule being applied to the pixel (1,3) of a 5-by-5 matrix. The center pixel is indicated by a filled circle:



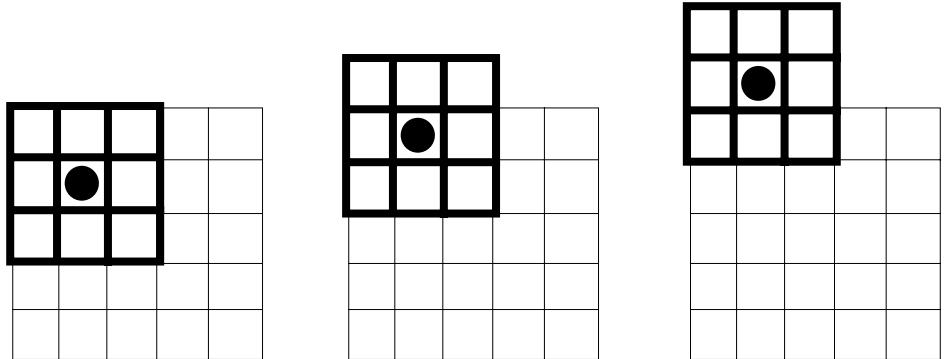
In order to compute output values for the border pixels, `conv2` pads the image matrix with zeroes. In other words, the output values are computed by assuming that the input image is surrounded by additional rows and columns of zeroes. In the figure shown above, the elements in the top row of the computational molecule are assumed to overlap zeroes.

Depending on what you are trying to accomplish, you may want to discard output pixels whose values depend on zero padding. To indicate what portion of the convolution to return, `conv2` takes a third input argument, called the `shape` parameter, whose value is one of these three strings:

- 'val' – returns only the pixels whose values can be computed without using zero padding of the input image. The resulting output image is smaller than the input image. In this example, the output image is 3-by-3.
- 'same' – returns the set of pixels that can be computed by applying the filter to all pixels that are actually part of the input image. Border pixels are computed using zero padding, but the center pixel of the computational kernel is applied only to pixels in the image. This results in an output image that is the same size as the input image.
- 'full' – returns the full convolution. This means `conv2` returns all pixels for which any of the pixels in the computational molecule overlap pixels in the image, even when the center pixel is outside the input image. The resulting output image is larger than the input image. In this example, the output image is 7-by-7.

`conv2` returns the full convolution by default.

The figure below illustrates applying a computational molecule to three different places in an image matrix:



The computational molecule overlaps only pixels that are in the original image. The result is included in the output matrix, regardless of the shape parameter.

The computational molecule overlaps pixels outside the original image, but the center pixel overlaps a pixel in the image. The result is included in the output matrix if the shape parameter is `same` or `full`.

The computational molecule overlaps pixels at the edges only. The center pixel is outside the image. The result is included in the output matrix if the shape parameter is `full`.

If you use the `full` option, then the order of the first two input arguments is interchangeable, because full convolution is commutative. In other words, it does not matter which matrix is considered the convolution kernel, because the result is the same in either case. If you use the `valid` or `same` option, the operation is not commutative, so the convolution kernel must be the second argument.

The `filter2` Function

In addition to the `conv2` function, MATLAB also provides the `filter2` function for two-dimensional linear filtering. `filter2` can produce the same results as `conv2`, and differs primarily in that it takes a computational molecule as an input argument, rather than a convolution kernel. (`filter2` operates by forming the convolution kernel from the computational molecule and then calling `conv2`.) The operation that `filter2` performs is called *correlation*.

If k is a convolution kernel, h is the corresponding computational molecule, and A is an image matrix, these calls produce identical results:

```
B = conv2(A, k, 'same');
```

and:

```
B = filter2(h, A, 'same');
```

The functions in the Image Processing Toolbox that produce filters (`fspecial`, `fsample`, etc.) all return computational molecules. You can use these filters directly with `filter2`, or you can rotate them 180 degrees and call `conv2`.

Separity

Before calling `conv2` to perform two-dimensional convolution, `filter2` first checks whether the filter is separable into two one-dimensional filters (one column vector and one row vector). If the filter is separable, `filter2` uses singular value decomposition to find the two vectors. `filter2` then calls `conv2` with this syntax:

```
conv2(A, kcol, krow);
```

`kcol` and `krow` are the column and row vectors that the two-dimensional convolution kernel k separates into (that is, $k = kcol * krow$).

`conv2` filters the columns with the column vector, and then, using the output of this operation, filters the rows using the row vector. The result is equivalent to two-dimensional convolution but is faster because it requires fewer computations.

Determining Separability

A filter is separable if its rank is 1. For example, this filter is separable:

```
k =
```

1	2	3
2	4	6
4	8	12

```
rank(k)
```

```
ans =
```

```
1
```

Higher-Dimensional Convolution

To perform two-dimensional convolution, you use `conv2` or `filter2`. To perform higher-dimensional convolution, you use the `convn` function. `convn` takes as arguments a data array and a convolution kernel, both of which can be of any dimension, and returns an array whose dimension is the higher of the two input arrays' dimensions. `convn` also takes a `shape` parameter argument that accepts the same values as in `conv2` and `filter2`, and which has analogous effects in higher dimensions.

One important application for the `convn` function is to filter image arrays that have multiple planes or frames. For example, suppose you have an array `A` containing five RGB images that you want to filter using a two-dimensional convolution kernel `k`. The image array is a four-dimensional array of size `m`-by-`n`-by-3-by-5. To filter this array with `conv2`, you would need to call the function 15 times, once for each combination of planes and frames, and assemble the results into a four-dimensional array. Using `convn`, you can filter the array in a single call:

```
B = convn(A, k);
```

For more information, see the reference entry for `convn` in Chapter 11.

Using Predefined Filter Types

The function `fspecial` produces several kinds of predefined filters, in the form of computational molecules. After creating a filter with `fspecial`, you can

apply it directly to your image data using `filter2`, or you can rotate it 180 degrees and use `conv2` or `convn`.

One simple filter `fspecial` can produce is an averaging filter. This type of filter computes the value of an output pixel by simply averaging the values of its neighboring pixels.

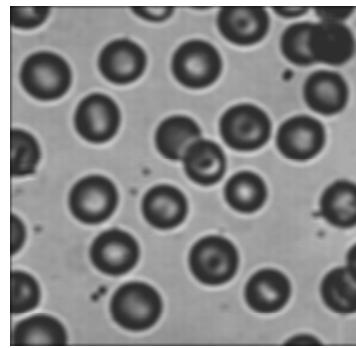
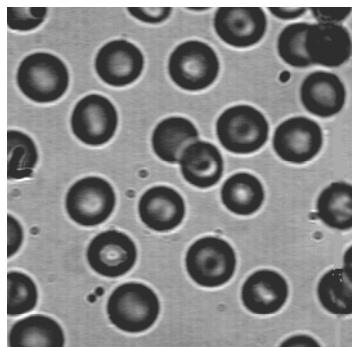
The default size of the averaging filter `fspecial` creates is 3-by-3, but you can specify a different size. The value of each element is $1/\text{length}(h(:))$. For example, a 5-by-5 averaging filter would be:

0. 0400	0. 0400	0. 0400	0. 0400	0. 0400
0. 0400	0. 0400	0. 0400	0. 0400	0. 0400
0. 0400	0. 0400	0. 0400	0. 0400	0. 0400
0. 0400	0. 0400	0. 0400	0. 0400	0. 0400
0. 0400	0. 0400	0. 0400	0. 0400	0. 0400

Applying this filter to a pixel is equivalent to adding up the values of that pixel's 5-by-5 neighborhood and dividing by 25. This has the effect of smoothing out local highlights and blurring edges in an image.

This example illustrates applying a 5-by-5 averaging filter to an intensity image:

```
I = imread('blood1.tif');
h = fspecial('average', 5);
I2 = uint8(round(filter2(h, I)));
imshow(I)
figure, imshow(I2)
```



Note that the output from `filter2` (and `conv2` and `convn`) is always of class `double`. In the example above, the input image is of class `uint8`, so the output from `filter2` consists of double-precision values in the range [0,255]. The call to the `uint8` function converts the output to `uint8`; the data is not in the proper range for an image of class `double`.

Another relatively simple filter `fspecial` can produce a 3-by-3 Sobel filter, which is effective at detecting the horizontal edges of objects in an image:

```
h = fspecial('sobel')
```

```
h =
```

1	2	1
0	0	0
-1	-2	-1

Unlike an averaging filter, the Sobel filter produces values outside the range of the input data. If the input image is of class `double`, the output array may include values outside the range [0,1]; if the input image is of class `uint8`, the output array may include values outside the range [0,255]. To display the output as an image, you can use `imshow` and specify the data range, or you can use the `mat2gray` function to convert the values to the range [0,1].

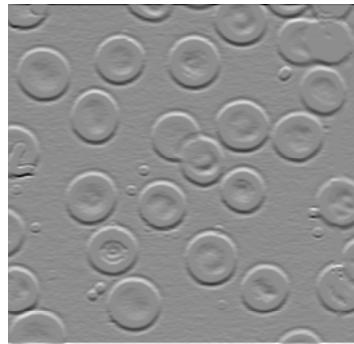
Note that if the input image is of class `uint8`, you should not simply convert the output to `uint8`; if the output contains values outside the range [0,255], these values cannot be represented as 8-bit integers, so the result of the conversion may be unpredictable. You can, however, rescale the output and then convert it. For example:

```
h = fspecial('sobel');
I2 = filter2(h, I);
J = uint8(round(mat2gray(I2)*255));
```

The following example creates a Sobel filter and uses `filter2` to apply the filter to the `blood1` image. Notice that in the call to `imshow`, the intensity range is specified as an empty matrix ([]). This instructs `imshow` to display the minimum value in `I2` as black, the maximum value as white, and values in

between as intermediate shades of gray, thus enabling you to display the `filter2` output without converting or rescaling it.

```
I = imread('blood1.tif');
h = fspecial('sobel');
I2 = filter2(h, I);
imshow(I2, [])
```



For a description of all the filter types `fspecial` provides, see the reference entry for `fspecial` in Chapter 11.

Filter Design

This section describes working in the frequency domain to design filters. Topics discussed include:

- Finite Impulse Response (FIR) filters, the class of linear filter that the toolbox supports
- The frequency transformation method, which transforms a one-dimensional FIR filter into a two-dimensional FIR filter
- The frequency sampling method, which creates a filter based on a desired frequency response
- The windowing method, which multiplies the ideal impulse response with a window function to generate the filter
- Creating the desired frequency response matrix
- Computing the frequency response of a filter

This section assumes you are familiar with working in the frequency domain. This topic is discussed in many signal processing and image processing textbooks.

NOTE Most of the design methods described in this section work by creating a two-dimensional filter from a one-dimensional filter or window created using functions from the Signal Processing Toolbox. Although this toolbox is not required, you may find it difficult to design filters in the Image Processing Toolbox if you do not have the Signal Processing Toolbox as well.

FIR Filters

The Image Processing Toolbox supports one class of linear filter, the two-dimensional Finite Impulse Response (FIR) filter. FIR filters have several characteristics that make them ideal for image processing in the MATLAB environment:

- FIR filters are easy to represent as matrices of coefficients.
- Two-dimensional FIR filters are natural extensions of one-dimensional FIR filters.

- There are several well-known, reliable methods for FIR filter design.
- FIR filters are easy to implement.
- FIR filters can be designed to have linear phase, which helps prevent distortion.

Another class of filter, the Infinite Impulse Response (IIR) filter, is not as suitable for image processing applications. It lacks the inherent stability and ease of design and implementation of the FIR filter. Therefore, this toolbox does not provide IIR filter support.

Frequency Transformation Method

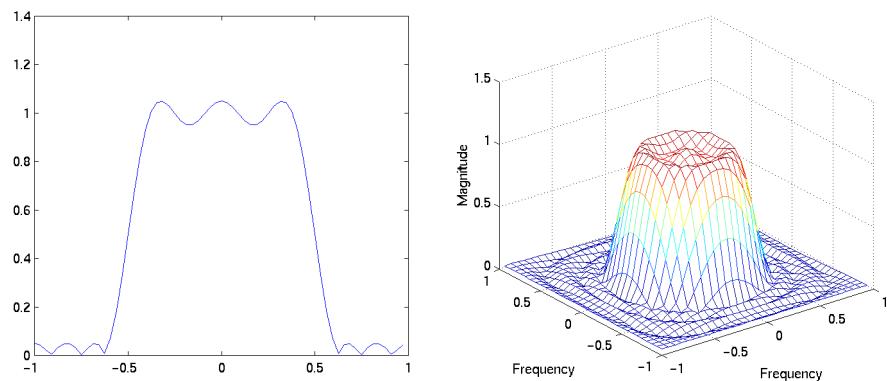
The frequency transformation method transforms a one-dimensional FIR filter into a two-dimensional FIR filter. The frequency transformation method preserves most of the characteristics of the one-dimensional filter, particularly the transition bandwidth and ripple characteristics. This method uses a *transformation matrix*, a set of elements that defines the frequency transformation.

The toolbox function `ftrans2` implements the frequency transformation method. This function's default transformation matrix produces filters with nearly circular symmetry. By defining your own transformation matrix, you can obtain different symmetries. (See Jae S. Lim, *Two-Dimensional Signal and Image Processing*, 1990, for details.)

The frequency transformation method generally produces very good results, as it is easier to design a one-dimensional filter with particular characteristics than a corresponding two-dimensional filter. For instance, the next example designs an optimal equiripple one-dimensional FIR filter and uses it to create a two-dimensional filter with similar characteristics. The shape of the

one-dimensional frequency response is clearly evident in the two-dimensional response.

```
b = remez(10, [0 0.4 0.6 1], [1 1 0 0]);
h = ftrans2(b);
[H, w] = freqz(b, 1, 64, 'whole');
colormap(jet(64))
plot(w/pi-1, fftshift(abs(H)))
figure, freqz2(h, [32 32])
```



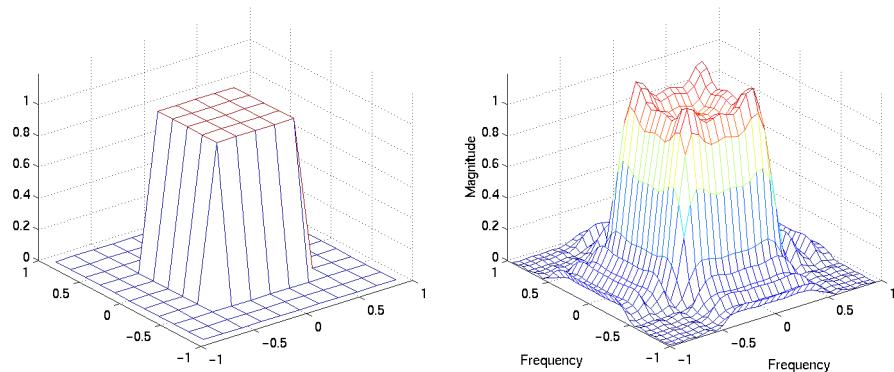
Frequency Sampling Method

The frequency sampling method creates a filter based on a desired frequency response. Given a matrix of points that defines the shape of the frequency response, this method creates a filter whose frequency response passes through those points. Frequency sampling places no constraints on the behavior of the frequency response between the given points; usually, the response ripples in these areas.

The toolbox function `fsamp2` implements frequency sampling design for two-dimensional FIR filters. `fsamp2` returns a filter `h` with a frequency response that passes through the points in the input matrix `Hd`. The example below creates an 11-by-11 filter using `fsamp2`, and plots the frequency response of the resulting filter. (The `freqz2` function in this example calculates the

two-dimensional frequency response of a filter. For more information, see page 5-19.)

```
Hd = zeros(11, 11); Hd(4:8, 4:8) = 1;
[f1, f2] = freqspace(11, 'meshgrid');
mesh(f1, f2, Hd), axis([-1 1 -1 1 0 1.2]), colormap(jet(64))
h = fsamp2(Hd);
figure, freqz2(h, [32 32]), axis([-1 1 -1 1 0 1.2])
```



Notice the ripples in the actual frequency response, compared to the desired frequency response. These ripples are a fundamental problem with the frequency sampling design method. They occur wherever there are sharp transitions in the desired response.

You can reduce the spatial extent of the ripples by using a larger filter. However, a larger filter does not reduce the height of the ripples, and requires more computation time for filtering. To achieve a smoother approximation to the desired frequency response, consider using the frequency transformation method or the windowing method.

Windowing Method

The windowing method involves multiplying the ideal impulse response with a window function to generate a corresponding filter. Like the frequency sampling method, the windowing method produces a filter whose frequency response approximates a desired frequency response. The windowing method, however, tends to produce better results than the frequency sampling method.

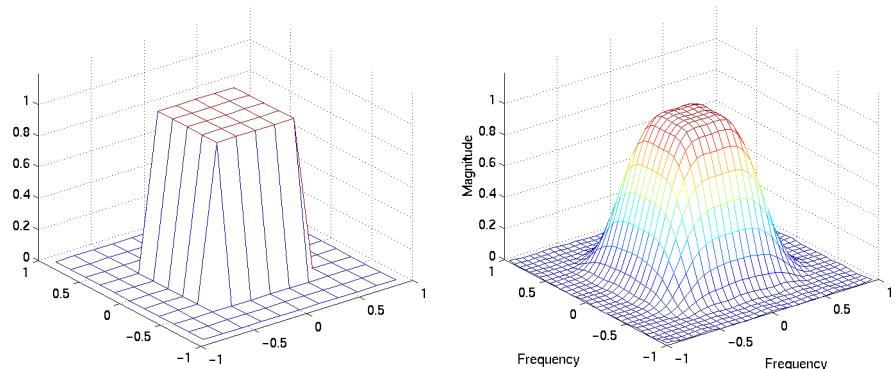
The toolbox provides two functions for window-based filter design, `fwi nd1` and `fwi nd2`. `fwi nd1` designs a two-dimensional filter by using a two-dimensional window that it creates from one or two one-dimensional windows that you specify. `fwi nd2` designs a two-dimensional filter by using a specified two-dimensional window directly.

`fwi nd1` supports two different methods for making the two-dimensional windows it uses:

- Transforming a single one-dimensional window to create a two-dimensional window that is nearly circularly symmetric, by using a process similar to rotation
- Creating a rectangular, separable window from two one-dimensional windows, by computing their outer product

The example below uses `fwi nd1` to create an 11-by-11 filter from the desired frequency response `Hd`. Here, the `hammi ng` function from the Signal Processing Toolbox is used to create a one-dimensional window, which `fwi nd1` then extends to a two-dimensional window.

```
Hd = zeros(11, 11); Hd(4:8, 4:8) = 1;
[f1, f2] = freqspace(11, 'meshgrid');
mesh(f1, f2, Hd), axis([-1 1 -1 1 0 1.2]), colormap(jet(64))
h = fwi nd1(Hd, hammi ng(11));
figure, freqz2(h, [32 32]), axis([-1 1 -1 1 0 1.2])
```

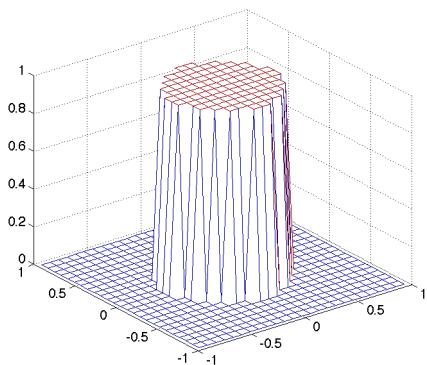


Creating the Desired Frequency Response Matrix

The filter design functions `fsamp2`, `fwind1`, and `fwind2` all create filters based on a desired frequency response magnitude matrix. You can create an appropriate desired frequency response matrix using the `freqspace` function. `freqspace` returns correct, evenly spaced frequency values for any size response. If you create a desired frequency response matrix using frequency points other than those returned by `freqspace`, you may get unexpected results, such as nonlinear phase.

For example, to create a circular ideal lowpass frequency response with cutoff at 0.5 use:

```
[f1, f2] = freqspace(25, 'meshgrid');
Hd = zeros(25, 25); d = sqrt(f1.^2 + f2.^2) < 0.5;
Hd(d) = 1;
mesh(f1, f2, Hd)
```



Note that for this frequency response, filters produced by `fsamp2`, `fwind1`, and `fwind2` are real. This result is desirable for most image processing applications. To achieve this in general, the desired frequency response should be symmetric about the frequency origin ($f_1 = 0$, $f_2 = 0$).

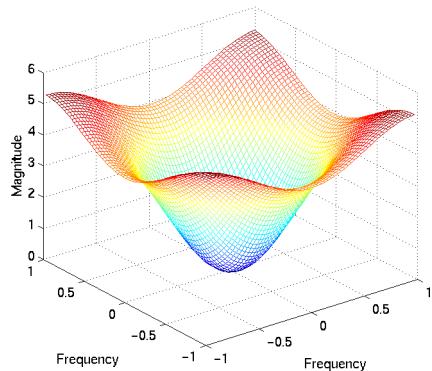
Computing the Frequency Response of a Filter

The `freqz2` function computes the frequency response for a two-dimensional filter. With no output arguments, `freqz2` creates a mesh plot of the frequency response. For example, consider this FIR filter:

```
h = [0.1667    0.6667    0.1667
      0.6667   -3.3333    0.6667
      0.1667    0.6667    0.1667];
```

This command computes and displays the 64-by-64 point frequency response of `h`:

```
freqz2(h)
```



To obtain the frequency response matrix `H` and the frequency point vectors `f1` and `f2`, use output arguments:

```
[H, f1, f2] = freqz2(h);
```

`freqz2` normalizes the frequencies `f1` and `f2` so that the value 1.0 corresponds to half the sampling frequency, or π radians.

For a simple m -by- n response, as shown above, `freqz2` uses the two-dimensional fast Fourier transform function `fft2`. You can also specify vectors of arbitrary frequency points, but in this case `freqz2` uses a slower algorithm.

For more information about the fast Fourier transform and its application to linear filtering and filter design, see Chapter 6.

Transforms

Overview	6-2
Fourier Transform	6-3
Definition	6-3
The Discrete Fourier Transform	6-8
Applications	6-11
Discrete Cosine Transform	6-15
The DCT Transform Matrix	6-16
The DCT and Image Compression	6-17
Radon Transform	6-19
Using the Radon Transform to Detect Lines	6-22
The Inverse Radon Transform	6-25

Overview

The usual mathematical representation of an image is a function of two spatial variables: $f(x, y)$. The value of the function at a particular location (x, y) represents the intensity of the image at that point. The term *transform* refers to an alternative mathematical representation of an image.

For example, the Fourier transform is a representation of an image as a sum of complex exponentials of varying magnitudes, frequencies, and phases. This representation is useful in a broad range of applications, including (but not limited to) image analysis, restoration, and filtering.

The discrete cosine transform (DCT) also represents an image as a sum of sinusoids of varying magnitudes and frequencies. The DCT is extremely useful for image compression; it is the basis of the widely used JPEG image compression algorithm.

The Radon transform represents an image as a collection of projections along various directions. It is used in areas ranging from seismology to computer vision.

This chapter defines each of these transforms, describes related toolbox functions, and shows examples of related image processing applications.

Fourier Transform

The Fourier transform plays a critical role in a broad range of image processing applications, including enhancement, analysis, restoration, and compression. This section defines the Fourier transform and the related discrete Fourier transform, discusses the fast Fourier transform, and presents some sample applications.

Definition

If $f(m, n)$ is a function of two discrete spatial variables m and n , then we define the *two-dimensional Fourier transform* of $f(m, n)$ by the relationship:

$$F(\omega_1, \omega_2) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m, n) e^{-j\omega_1 m} e^{-j\omega_2 n}$$

The variables ω_1 and ω_2 are frequency variables; their units are radians per sample. $F(\omega_1, \omega_2)$ is often called the *frequency-domain* representation of $f(m, n)$. $F(\omega_1, \omega_2)$ is a complex-valued function that is periodic both in ω_1 and ω_2 , with period 2π . Because of the periodicity, usually only the range $-\pi \leq \omega_1, \omega_2 \leq \pi$ is displayed. Note that $F(0, 0)$ is the sum of all the values of $f(m, n)$. For this reason, $F(0, 0)$ is often called the *constant component* or *DC component* of the Fourier transform. (DC stands for direct current; it is an electrical engineering term that refers to a constant-voltage power source, as opposed to a power source whose voltage varies sinusoidally.)

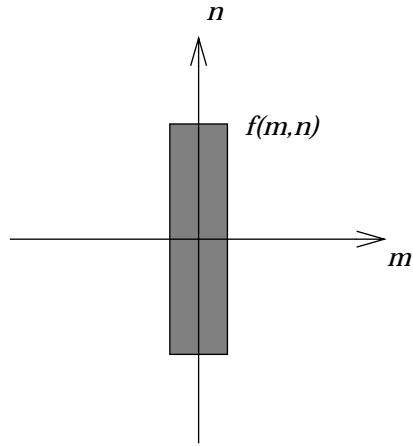
The inverse two-dimensional Fourier transform is given by:

$$f(m, n) = \frac{1}{2\pi} \int_{\omega_1 = -\pi}^{\pi} \int_{\omega_2 = -\pi}^{\pi} F(\omega_1, \omega_2) e^{j\omega_1 m} e^{j\omega_2 n} d\omega_1 d\omega_2$$

Roughly speaking, this equation means that $f(m, n)$ can be represented as a sum of an infinite number of complex exponentials (sinusoids) with different frequencies. The magnitude and phase of the contribution at the frequencies (ω_1, ω_2) are given by $F(\omega_1, \omega_2)$.

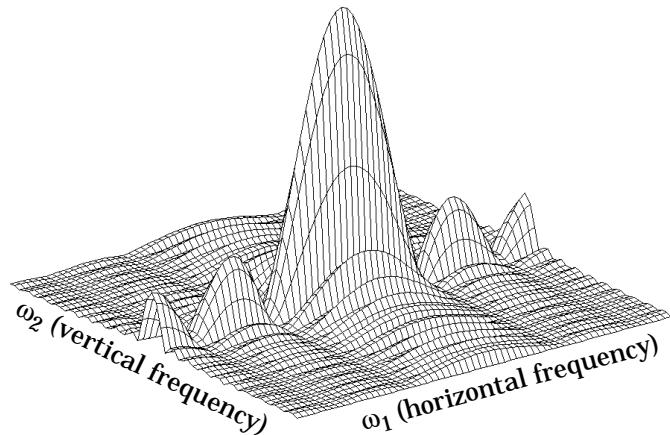
Example

Consider a function $f(m, n)$ that equals 1 within a rectangular region and 0 everywhere else.



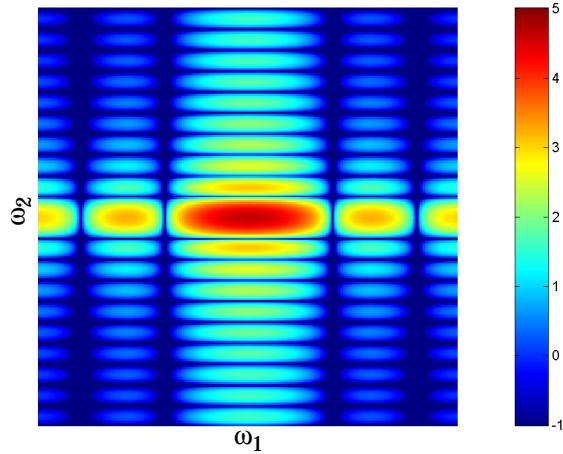
To simplify the diagram, $f(m, n)$ is shown as a continuous function, even though the variables m and n are discrete.

The magnitude of the Fourier transform, $|F(\omega_1, \omega_2)|$, is shown here as a mesh plot:



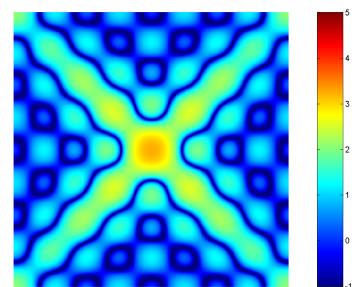
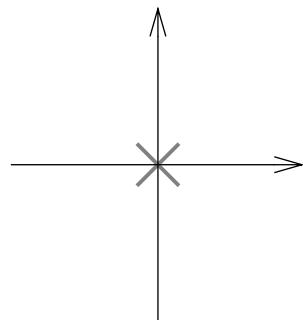
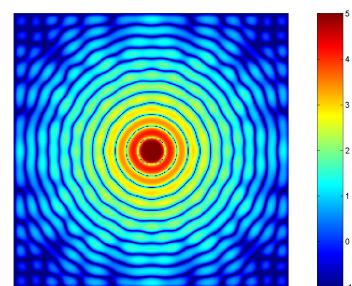
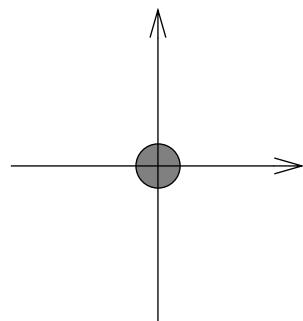
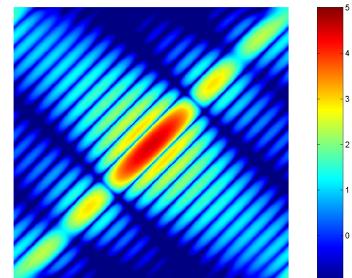
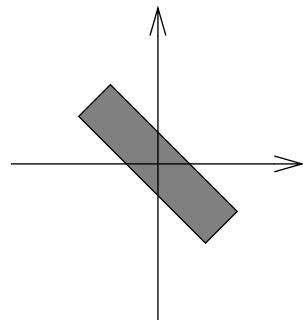
The peak at the center of the plot is $F(0, 0)$, which is the sum of all the values in $f(m, n)$. The plot also shows that $F(\omega_1, \omega_2)$ has more energy at high horizontal frequencies than at high vertical frequencies. This reflects the fact that horizontal cross sections of $f(m, n)$ are narrow pulses, while vertical cross sections are broad pulses. Narrow pulses have more high-frequency content than broad pulses.

Another common way to visualize the Fourier transform is to display $\log|F(\omega_1, \omega_2)|$ as an image, as in:



Using the logarithm helps to bring out details of the Fourier transform in regions where $F(\omega_1, \omega_2)$ is very close to 0.

Examples of the Fourier transform for other simple shapes are shown below.



The Discrete Fourier Transform

Working with the Fourier transform on a computer usually involves a form of the transform known as the discrete Fourier transform (DFT). There are two principal reasons for using this form:

- The input and output of the DFT are both discrete, which makes it convenient for computer manipulations.
- There is a fast algorithm for computing the DFT known as the fast Fourier transform (FFT).

The DFT is usually defined for a discrete function $f(m, n)$ that is nonzero only over the finite region $0 \leq m \leq M - 1$ and $0 \leq n \leq N - 1$. The two-dimensional M -by- N DFT and inverse M -by- N DFT relationships are given by:

$$F(p, q) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) e^{-j(2\pi/M)pm} e^{-j(2\pi/N)qn} \quad p = 0, 1, \dots, M-1 \\ q = 0, 1, \dots, N-1$$

$$f(m, n) = \frac{1}{MN} \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} F(p, q) e^{j(2\pi/M)pm} e^{j(2\pi/N)qn} \quad m = 0, 1, \dots, M-1 \\ n = 0, 1, \dots, N-1$$

The values $F(p, q)$ are called the DFT coefficients of $f(m, n)$. In particular, the value $F(0, 0)$ is called the DC coefficient. (Note that matrix indices in MATLAB always start at 1 rather than 0; therefore, the matrix elements $f(1, 1)$ and $F(1, 1)$ correspond to the mathematical quantities $f(0, 0)$ and $F(0, 0)$, respectively.)

The MATLAB functions `fft`, `fft2`, and `fftn` implement the fast Fourier transform algorithm for computing the one-dimensional DFT, two-dimensional DFT, and N-dimensional DFT, respectively. The functions `ifft`, `ifft2`, and `ifftn` compute the inverse DFT.

Relationship to the Fourier Transform

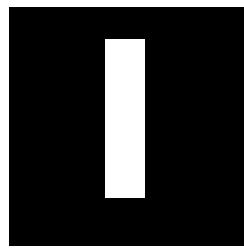
The DFT coefficients $F(p, q)$ are samples of the Fourier transform $F(\omega_1, \omega_2)$:

$$F(p, q) = F(\omega_1, \omega_2) \Big|_{\begin{array}{l} \omega_1 = 2\pi p/M \\ \omega_2 = 2\pi q/N \end{array}} \quad \begin{array}{l} p = 0, 1, \dots, M-1 \\ q = 0, 1, \dots, N-1 \end{array}$$

Example

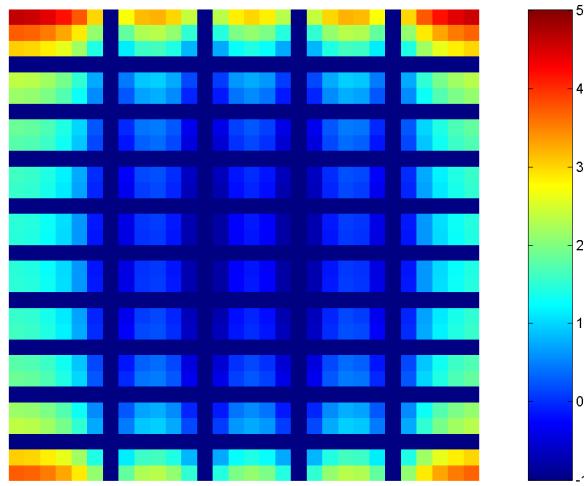
Let's construct a matrix f that is similar to the function $f(m,n)$ in the example on page 6-4.

```
f = zeros(30, 30);  
f(5:24, 13:17) = 1;  
imshow(f, 'notruesize')
```



Compute and visualize the 30-by-30 DFT of f with these commands:

```
F = fft2(f);  
F2 = log(abs(F));  
imshow(F2, [-1 5], 'notruesize'); colormap(jet); colorbar
```



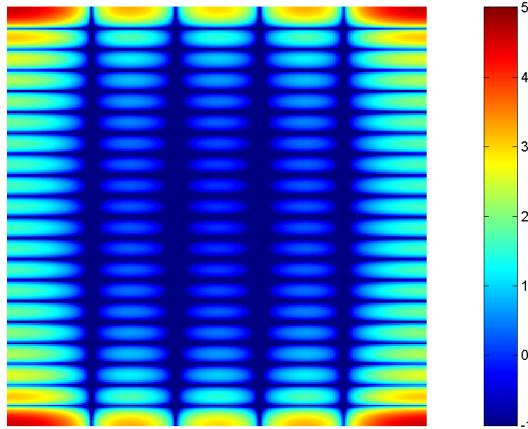
This plot differs from the Fourier transform displayed on page 6-6. First, the resolution is much lower. Second, the DC coefficient is displayed in the upper-left corner instead of the traditional location in the center.

The resolution can be increased by zero-padding f when computing its DFT. The zero-padding and DFT computation can be performed in a single step with this command:

```
F = fft2(f, 256, 256);
```

This command zero-pads f to be 256-by-256 before computing the DFT. The result is that the DCT coefficients F sample the Fourier transform much more finely:

```
imshow(log(abs(F)), [-1 5]); colormap(jet); colorbar
```



The DC coefficient, however, is still displayed in the upper-left corner rather than the center. You can fix this problem by using the function `fftshift`, which swaps the quadrants of F so that the DC coefficient is in the center.

```
F = fft2(f, 256, 256);
F2 = fftshift(F);
imshow(log(abs(F2)), [-1 5]); colormap(jet); colorbar
```

The resulting plot is identical to the one on page 6-6.

Applications

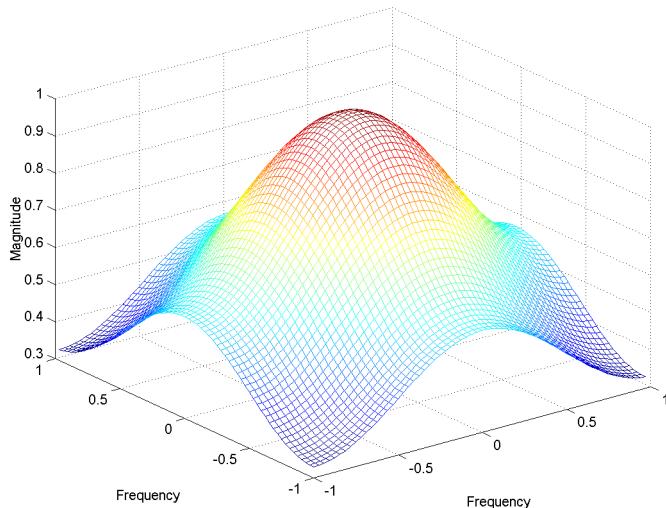
This section presents a few of the many image processing-related applications of the Fourier transform.

Frequency Response of Linear Filters

The Fourier transform of the impulse response of a linear filter gives the frequency response of the filter. The function `freqz2` computes and displays a filter's frequency response. The frequency response of the Gaussian

convolution kernel shows that this filter passes low frequencies and attenuates high frequencies.

```
h = fspecial('gaussian');
freqz2(h)
```



See Chapter 5 for more information about linear filtering, filter design, and frequency responses.

Fast Convolution

A key property of the Fourier transform is that the multiplication of two Fourier transforms corresponds to the convolution of the associated spatial functions. This property, together with the fast Fourier transform, forms the basis for a fast convolution algorithm.

Suppose that A is an M-by-N matrix and B is a P-by-Q matrix. The convolution of A and B can be computed using the following steps:

- 1 Zero-pad A and B so that they are at least $(M+P-1)\text{-by-}(N+Q-1)$. (Often A and B are zero-padded to a size that is a power of 2 because `fft2` is fastest for these sizes.)
- 2 Compute the two-dimensional DFT of A and B using `fft2`.

- 3** Multiply the two DFTs together.
- 4** Using `ifft2`, compute the inverse two-dimensional DFT of the result from step 3.

For example:

```
A = magic(3);
B = ones(3);
A(8, 8) = 0; % zero-pad A to be 8-by-8
B(8, 8) = 0; % zero-pad B to be 8-by-8
C = ifft2(fft2(A) .* fft2(B));
C = C(1: 5, 1: 5); % extract the nonzero portion
C = real(C) % remove imaginary part caused by roundoff error

C =

```

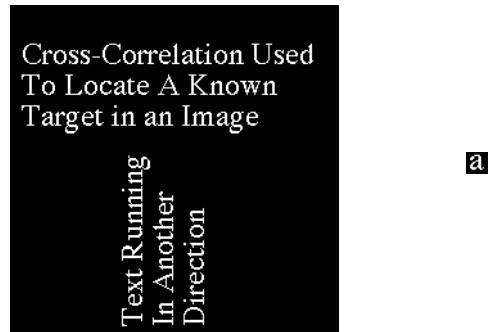
8.0000	9.0000	15.0000	7.0000	6.0000
11.0000	17.0000	30.0000	19.0000	13.0000
15.0000	30.0000	45.0000	30.0000	15.0000
7.0000	21.0000	30.0000	23.0000	9.0000
4.0000	13.0000	15.0000	11.0000	2.0000

The FFT-based convolution method is most often used for large inputs. For small inputs it is generally faster to use `filter2` or `conv2`.

Locating Image Features

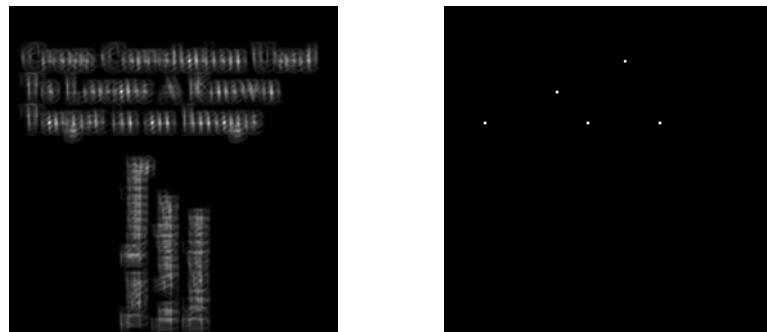
The Fourier transform can also be used to perform correlation, which is closely related to convolution. Correlation can be used to locate features within an image; in this context correlation is often called *template matching*. For instance, suppose you want to locate occurrences of the letter “a” in the image

text.tif. The figure below shows the image and the template you want to match:



a

The correlation of the image of the letter “a” with the larger image can be computed by first rotating the image of “a” by 180° and then using the FFT-based convolution technique described above. The left image below is the result of the correlation; bright peaks correspond to occurrences of the letter. The locations of these peaks are indicated by the white spots in the thresholded correlation image shown on the right.



Discrete Cosine Transform

The `dct2` function in the Image Processing Toolbox computes the two-dimensional discrete cosine transform (DCT) of an image. The DCT has the property that, for a typical image, most of the visually significant information about the image is concentrated in just a few coefficients of the DCT. For this reason, the DCT is often used in image compression applications. For example, the DCT is at the heart of the international standard lossy image compression algorithm known as JPEG. (The name comes from the working group that developed the standard: the Joint Photographic Experts Group.)

The two-dimensional DCT of an M-by-N matrix A is defined as follows:

$$B_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A_{mn} \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N}, \quad 0 \leq p \leq M-1, \quad 0 \leq q \leq N-1$$

$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \leq p \leq M-1 \end{cases} \quad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \leq q \leq N-1 \end{cases}$$

The values B_{pq} are called the *DCT coefficients* of A. (Note that matrix indices in MATLAB always start at 1 rather than 0; therefore, the MATLAB matrix elements A(1, 1) and B(1, 1) correspond to the mathematical quantities A_{00} and B_{00} , respectively.)

The DCT is an invertible transform, and its inverse is given by:

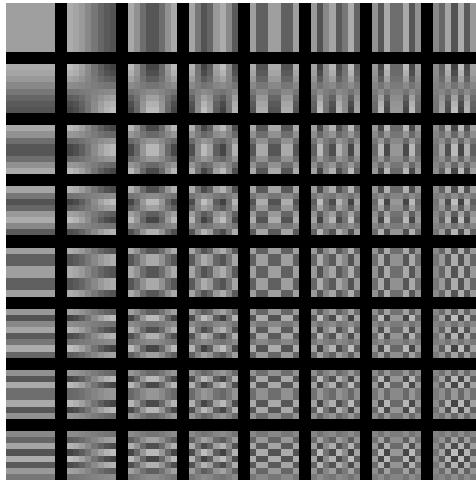
$$A_{mn} = \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} \alpha_p \alpha_q B_{pq} \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N}, \quad 0 \leq m \leq M-1, \quad 0 \leq n \leq N-1$$

$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \leq p \leq M-1 \end{cases} \quad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \leq q \leq N-1 \end{cases}$$

The inverse DCT equation can be interpreted as meaning that any M-by-N matrix A can be written as a sum of MN functions of the form:

$$\alpha_p \alpha_q \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N}, \quad 0 \leq p \leq M-1 \\ 0 \leq q \leq N-1$$

These functions are called the *basis functions* of the DCT. The DCT coefficients B_{pq} , then, can be regarded as the *weights* applied to each basis function. For 8-by-8 matrices, the 64 basis functions are illustrated by this image:



Horizontal frequencies increase from left to right, and vertical frequencies increase from top to bottom. The constant-valued basis function at the upper left is often called the *DC basis function*, and the corresponding DCT coefficient B_{00} is often called the *DC coefficient*.

The DCT Transform Matrix

The Image Processing Toolbox offers two different ways to compute the DCT. The first method is to use the function `dct2`. `dct2` uses an FFT-based algorithm for speedy computation with large inputs.

For small square inputs, such as 8-by-8 or 16-by-16, it may be more efficient to use the DCT *transform matrix*, which is returned by the function `dctmtx`. The M-by-M transform matrix T is given by:

$$T_{pq} = \begin{cases} \frac{1}{\sqrt{M}} & p = 0, \quad 0 \leq q \leq M-1 \\ \frac{\sqrt{2}}{\sqrt{M}} \cos \frac{\pi(2q+1)p}{2M} & 1 \leq p \leq M-1, \quad 0 \leq q \leq M-1 \end{cases}$$

For an M -by- M matrix A , T^*A is an M -by- M matrix whose columns contain the one-dimensional DCT of the columns of A . The two-dimensional DCT of A can be computed as $B=T^*A*T'$. Since T is a real orthonormal matrix, its inverse is the same as its transpose. Therefore, the inverse two-dimensional DCT of B is given by $T' *B*T$.

The DCT and Image Compression

In the JPEG image compression algorithm, the input image is divided into 8-by-8 or 16-by-16 blocks, and the two-dimensional DCT is computed for each block. The DCT coefficients are then quantized, coded, and transmitted. The JPEG receiver (or JPEG file reader) decodes the quantized DCT coefficients, computes the inverse two-dimensional DCT of each block, and then puts the blocks back together into a single image. For typical images, many of the DCT coefficients have values close to zero; these coefficients can be discarded without seriously affecting the quality of the reconstructed image.

The example code below computes the two-dimensional DCT of 8-by-8 blocks in the input image; discards (sets to zero) all but 10 of the 64 DCT coefficients in

each block; and then reconstructs the image using the two-dimensional inverse DCT of each block. The transform matrix computation method is used.

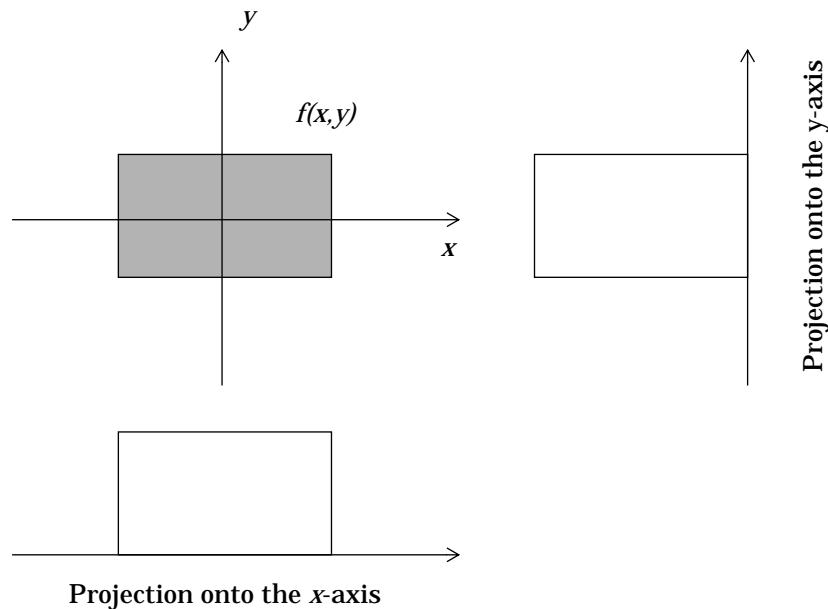
```
I = imread('cameraman.tif');
I = double(I)/255;
T = dctmtx(8);
B = blkproc(I, [8 8], 'P1*x*P2', T, T');
mask = [1 1 1 1 0 0 0 0
        1 1 1 0 0 0 0 0
        1 1 0 0 0 0 0 0
        1 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0];
B2 = blkproc(B, [8 8], 'P1.*x', mask);
I2 = blkproc(B2, [8 8], 'P1*x*P2', T', T);
imshow(I), figure, imshow(I2)
```



Although there is some loss of quality in the reconstructed image, it is clearly recognizable, even though almost 85% of the DCT coefficients were discarded. To experiment with discarding more or fewer coefficients, and to apply this technique to other images, try running the demo function `dctdemo`.

Radon Transform

The radon function in the Image Processing Toolbox computes *projections* of an image matrix along specified directions. A projection of a two-dimensional function $f(x,y)$ is a line integral in a certain direction. For example, the line integral of $f(x,y)$ in the vertical direction is the projection of $f(x,y)$ onto the x -axis; the line integral in the horizontal direction is the projection of $f(x,y)$ onto the y -axis. This figure shows horizontal and vertical projections for a simple two-dimensional function.



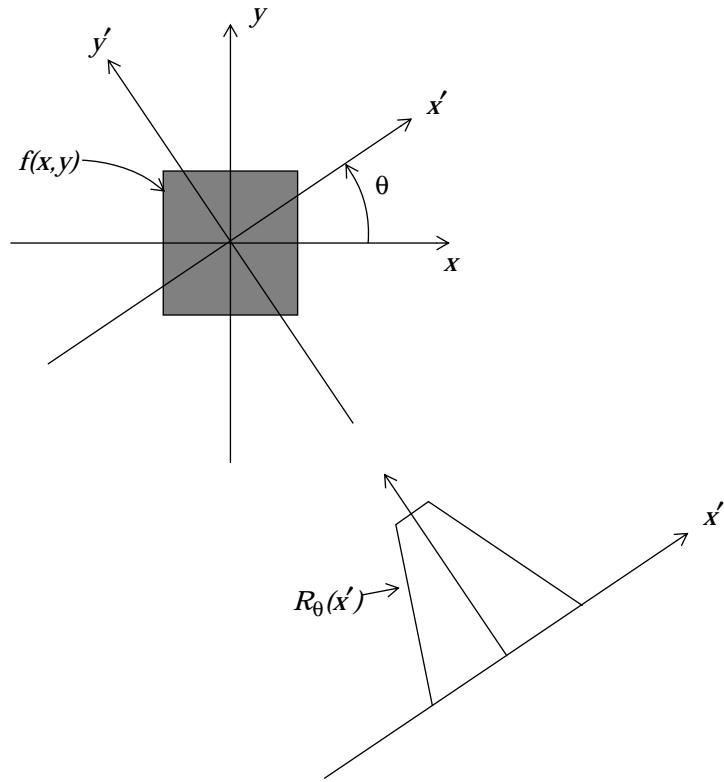
Projections can be computed along any angle θ . In general, the Radon transform of $f(x,y)$ is the line integral of f parallel to the y' axis:

$$R_\theta(x') = \int_{-\infty}^{\infty} f(x' \cos \theta - y' \sin \theta, x' \sin \theta + y' \cos \theta) dy'$$

where

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

This figure illustrates the geometry of the Radon transform.



This command computes the Radon transform of I for the angles specified in the vector theta:

```
[R, xp] = radon(I, theta);
```

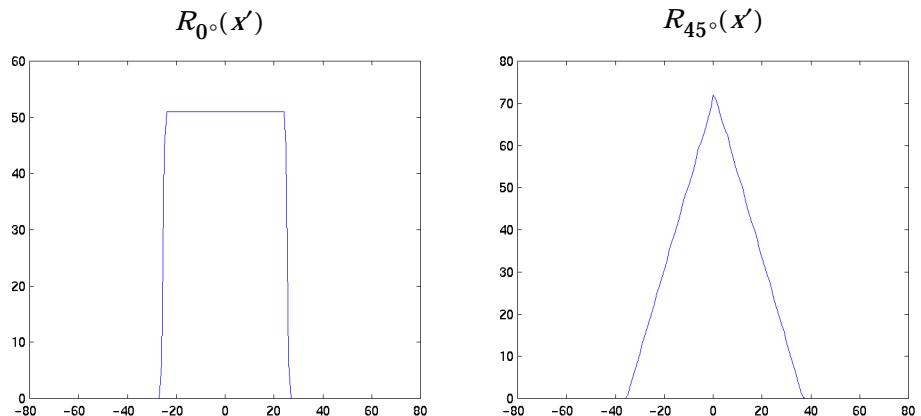
The columns of R contain the Radon transform for each angle in theta. xp contains the corresponding coordinates along the x' -axis. The “center pixel” of I is defined to be $\text{floor}((\text{size}(I)+1)/2)$; this is the pixel on the x' -axis corresponding to $x' = 0$.

The commands below compute and plot the Radon transform at 0° and 45° of an image containing a single square object.

```
I = zeros(100, 100);  
I(25:75, 25:75) = 1;  
imshow(I)
```

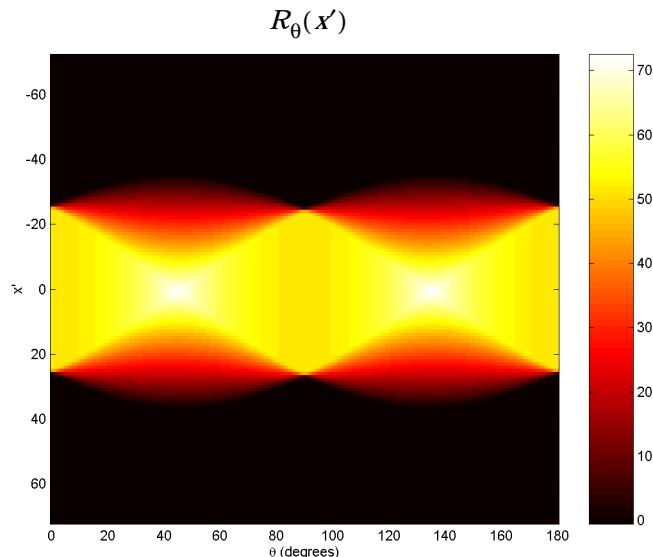


```
[R, xp] = radon(I, [0 45]);  
figure; plot(xp, R(:, 1)); title('R_{0^\circ} (x\prime)')  
figure; plot(xp, R(:, 2)); title('R_{45^\circ} (x\prime)')
```



The Radon transform for a large number of angles is often displayed as an image. In this example, the Radon transform for the square image is computed at angles from 0° to 180°, in 1° increments:

```
theta = 0: 180;
[R, xp] = radon(I, theta);
imagesc(theta, xp, R);
xlabel(' \theta (degrees)');
ylabel(' x' ');
colorbar
```

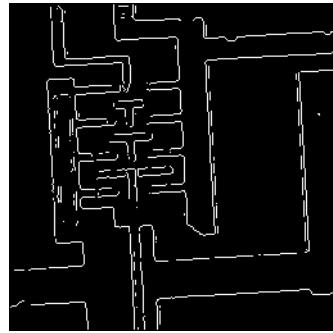
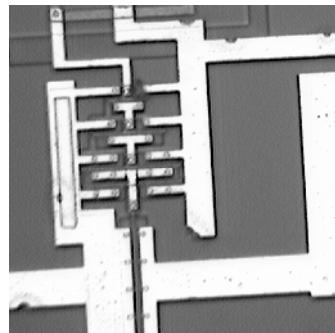


Using the Radon Transform to Detect Lines

The Radon transform is closely related to a common computer vision operation known as the Hough transform. You can use the `radon` function to implement a form of the Hough transform used to detect straight lines. The steps are:

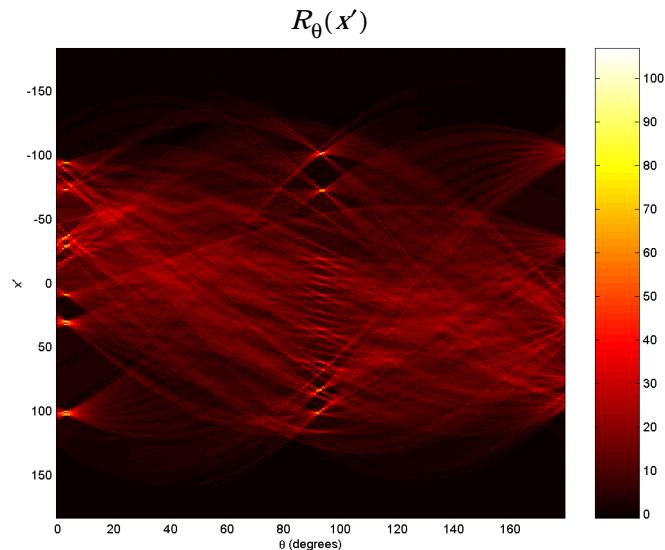
- 1 Compute a binary edge image using the edge function.

```
I = imread('ic.tif');
BW = edge(I);
imshow(I)
figure, imshow(BW)
```



- 2 Compute the Radon transform of the edge image.

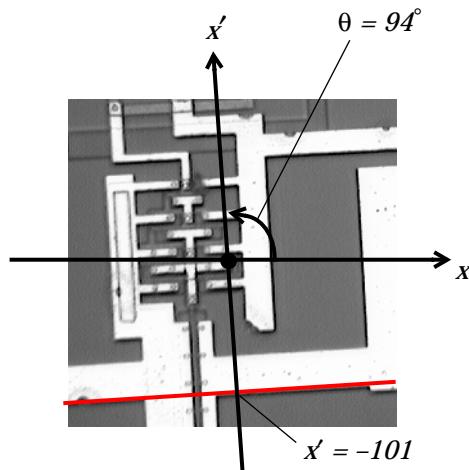
```
theta = 0: 179;  
[R, xp] = radon(BW, theta);  
imagesc(theta, xp, R); colormap(hot);  
xlabel ('\theta (degrees)'); ylabel ('x' );  
colorbar
```



- 3 Find the locations of strong peaks in the Radon transform matrix. The locations of these peaks correspond to the location of straight lines in the original image.

In this example, the strongest peak in R corresponds to $\theta = 94^\circ$ and $x' = -101$. The line perpendicular to that angle and located at $x' = -101$ is

shown below, superimposed in red on the original image. The Radon transform geometry is shown in black.



Notice that the other strong lines parallel to the red line also appear as peaks at $\theta = 94^\circ$ in the transform. Also, the lines perpendicular to this line appear as peaks at $\theta = 4^\circ$.

The Inverse Radon Transform

The `i radon` function in the toolbox uses the inverse Radon transform to reconstruct images from projection data, such as the output from the `radon` function. The inverse Radon transform is commonly used in tomography applications.

For more information about `i radon`, see the reference entry for this function in Chapter 11. Also, see the entry for the `phantom` function, which you can use to generate images for testing the Radon and inverse Radon transforms.

Analyzing and Enhancing Images

Overview	7-2
Pixel Values and Statistics	7-3
Pixel Selection	7-3
Intensity Profile	7-4
Image Contours	7-7
Image Histogram	7-8
Summary Statistics	7-9
Feature Measurement	7-9
Image Analysis	7-10
Edge Detection	7-10
Quadtree Decomposition	7-11
Image Enhancement	7-14
Intensity Adjustment	7-14
Noise Removal	7-20

Overview

The Image Processing Toolbox supports a range of standard image processing operations for analyzing and enhancing images. Its functions simplify several categories of tasks, including:

- Obtaining pixel values and statistics, which are numerical summaries of data in an image
- Analyzing images to extract information about their essential structure
- Enhancing images to make certain features easier to see or to reduce noise

This section describes specific operations within each category, and shows how to implement each kind of operation using toolbox functions.

Pixel Values and Statistics

The Image Processing Toolbox provides several functions that return information about the data values that make up an image. These functions return information about image data in various forms, including:

- The data values for selected pixels (`impixel`, `pixval`)
- The data values along a path in an image (`improfile`)
- A contour plot of the image data (`imcontour`)
- A histogram of the image data (`imhist`)
- Summary statistics for the image data (`mean2`, `std2`, `corr2`)
- Feature measurements for image regions (`imfeature`)

Pixel Selection

The `impixel` function returns the data values for a selected pixel or set of pixels. You can supply the coordinates of the pixels as input arguments, or you can select pixels using a mouse.

If you call `impixel` with no input arguments, the cursor changes to a cross hair when it is over the image. You can then click on the pixels of interest; `impixel` displays a small star over each pixel you select. When you are done selecting pixels, press **Return**. `impixel` returns the color values for the selected pixels, and removes the stars.

`impixel` works with indexed, intensity, and RGB images. `impixel` always returns pixel values as RGB triplets, regardless of the image type:

- For an RGB image, `impixel` returns the actual data for the pixel. The values are either `uint8` integers or `double` floating-point numbers, depending on the class of the image array.
- For an indexed image, `impixel` returns the RGB triplet stored in the row of the colormap that the pixel value points to. The values are `double` floating-point numbers.
- For an intensity image, `impixel` returns the intensity value as an RGB triplet, where R=G=B. The values are either `uint8` integers or `double` floating-point numbers, depending on the class of the image array.

In this example, you call `improfile` and click on three points in the displayed image, and then press **Return**.

```
imshow canoe.tif  
improfile
```



```
ans =  
  
0.1294    0.1294    0.1294  
0.5176        0        0  
0.7765    0.6118    0.4196
```

Notice that the second pixel, which is part of the canoe, is pure red; its green and blue values are both 0.

The toolbox also provides the `pixval` function, which interactively displays the data values for pixels as you move the cursor over the image. `pixval` can also display the Euclidean distance between two pixels. For more information about `pixval`, see the reference entry for this function in Chapter 11.

Intensity Profile

The `improfile` function calculates and plots the intensity values along a line segment or a multiline path in an image. You can supply the coordinates of the line segments as input arguments, or you can define the desired path using a mouse. In either case, `improfile` uses interpolation to determine the values of equally spaced points along the path. (By default, `improfile` uses nearest neighbor interpolation, but you can specify a different method. See Chapter 3

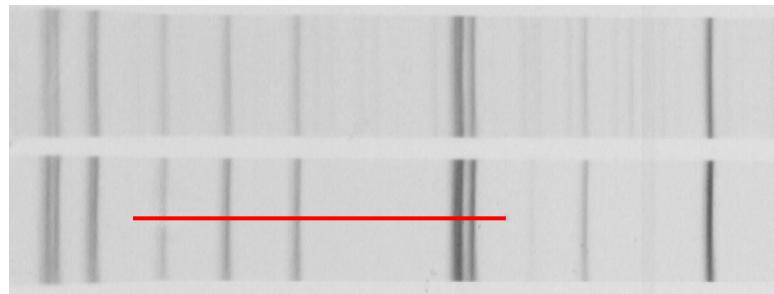
for a discussion of interpolation.) `improfile` works best with intensity and RGB images. (For indexed images, it displays the index values from the image matrix, not the intensity values from the colormap.)

For a single line segment, `improfile` plots the intensity values in a two-dimensional view. For a multiline path, `improfile` plots the intensity values in a three-dimensional view.

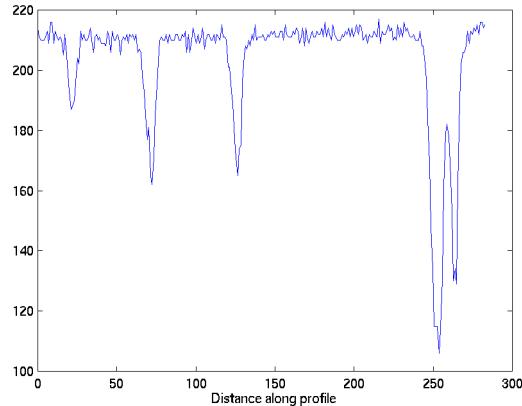
If you call `improfile` with no arguments, the cursor changes to a cross hair when it is over the image. You can then specify line segments by clicking on the endpoints; `improfile` draws a line between each two consecutive points you select. When you finish specifying the path, press **Return**. `improfile` displays the plot in a new figure.

In this example, you call `improfile` and specify a single line with the mouse. The line is shown in red, and is drawn from left to right.

```
imshow debye1.tif  
improfile
```



`improfile` displays a plot of the data along the line:



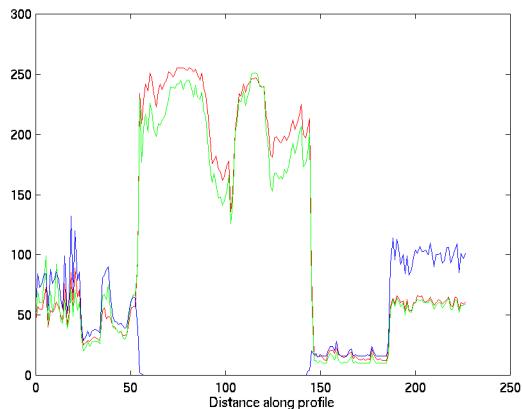
Notice the peaks and valleys and how they correspond to the light and dark bands in the image.

The example below shows how `improfile` works with an RGB image. The red line is drawn from top to bottom.

```
imshow flowers.tif  
improfile
```



`improfile` displays a plot with separate lines for the red, green, and blue intensities:



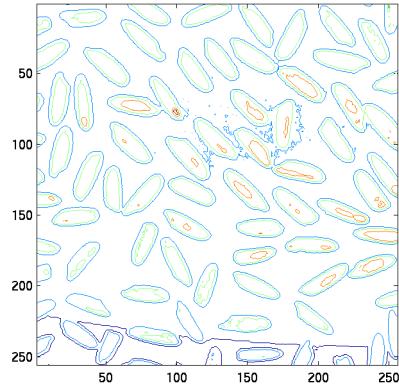
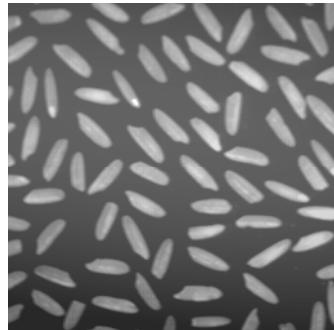
Notice how the lines correspond to the colors in the image. For example, the central region of the plot shows high intensities of green and red, while the blue intensity is 0. These are the values for the yellow flower.

Image Contours

You can use the toolbox function `imcontour` to display a contour plot of the data in an intensity image. This function is similar to the `contour` function in MATLAB, but it automatically sets up the axes so their orientation and aspect ratio match the image.

This example displays a grayscale image of grains of rice and a contour plot of the image data.

```
I = imread('rice.tif');
imshow(I)
figure, imcontour(I)
```



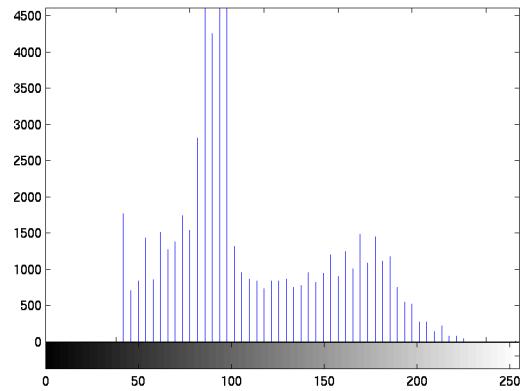
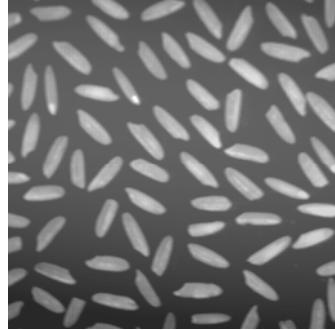
You can use the `clabel` function to label the levels of the contours. See the description of `clabel` in the online MATLAB Function Reference for details.

Image Histogram

An *image histogram* is a chart that shows the distribution of intensities in an indexed or intensity image. The image histogram function `i mhist` creates this plot by making n equally spaced bins, each representing a range of data values. It then calculates the number of pixels within each range. For example, the

commands below display an image of grains of rice, and a histogram based on 64 bins.

```
I = imread('rice.tif');
imshow(I)
figure, imhist(I, 64)
```



The histogram shows a peak at around 100, due to the dark background in the image.

For information about how to modify an image by changing the distribution of its histogram, see “Intensity Adjustment” on page 7-14.

Summary Statistics

You can compute standard image statistics using the `mean2`, `std2`, and `corr2` functions. `mean2` and `std2` compute the mean and standard deviation of the elements of a matrix. `corr2` computes the correlation coefficient between two matrices of the same size.

These functions are two-dimensional versions of the `mean`, `std`, and `corrcoef` functions described in the online MATLAB Function Reference.

Feature Measurement

You can use the `imfeature` function to compute feature measurements for image regions. For example, `imfeature` can measure such features as the area, center of mass, and bounding box for a region you specify. See the reference entry for `imfeature` in Chapter 11 for more information.

Image Analysis

Image analysis techniques return information about the structure of an image. This section describes toolbox functions that you can use for these image analysis techniques:

- Edge detection
- Quadtree decomposition

The functions described in this section work only with grayscale intensity images.

Edge Detection

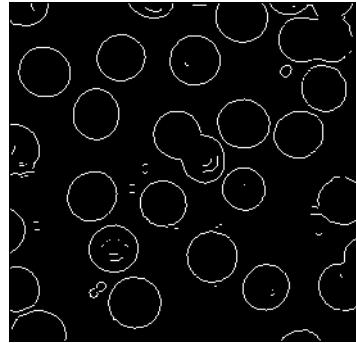
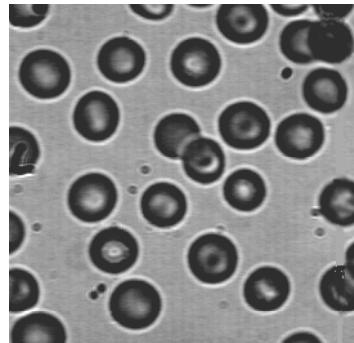
You can use the `edge` function to detect edges, which are those places in an image that correspond to object boundaries. To find edges, this function looks for places in the image where the intensity changes rapidly, using one of these two criteria:

- Places where the first derivative of the intensity is larger in magnitude than some threshold
- Places where the second derivative of the intensity has a zero crossing

`edge` provides a number of derivative estimators, each of which implements one of the definitions above. For some of these estimators, you can specify whether the operation should be sensitive to horizontal or vertical edges, or both. `edge` returns a binary image containing 1's where edges are found and 0's elsewhere.

The example below uses the Sobel method to detect the edges in an image of blood cells. By default, the operation is sensitive to both horizontal and vertical edges.

```
I = imread('blood1.tif');
BW = edge(I, 'sobel');
imshow(I)
figure, imshow(BW)
```



For an interactive demonstration of edge detection, try running `edgedemo`.

Quadtree Decomposition

Quadtree decomposition is an analysis technique that involves subdividing an image into blocks that are more homogeneous than the image itself. This technique reveals information about the structure of the image. It is also useful as the first step in adaptive compression algorithms.

You can perform quadtree decomposition using the `qtdecomp` function. This function works by dividing a square image into four equal-sized square blocks, and then testing each block to see if it meets some criterion of homogeneity (e.g., if all of the pixels in the block are within a specific dynamic range). If a block meets the criterion, it is not divided any further. If it does not meet the criterion, it is subdivided again into four blocks, and the test criterion is applied to those blocks. This process is repeated iteratively until each block meets the criterion. The result may have blocks of several different sizes.

For example, suppose you want to perform quadtree decomposition on a 128-by-128 grayscale image. The first step is to divide the image into four

64-by-64 blocks. You then apply the test criterion to each block; for example, the criterion might be:

$$\max(\text{block}(:)) - \min(\text{block}(:)) \leq 0.2$$

If one of the blocks meets this criterion, it is not divided any further; it is 64-by-64 in the final decomposition. If a block does not meet the criterion, it is then divided into four 32-by-32 blocks, and the test is then applied to each of these blocks. The blocks that fail to meet the criterion are then divided into four 16-by-16 blocks, and so on, until all blocks “pass.” Some of the blocks may be as small as 1-by-1, unless you specify otherwise.

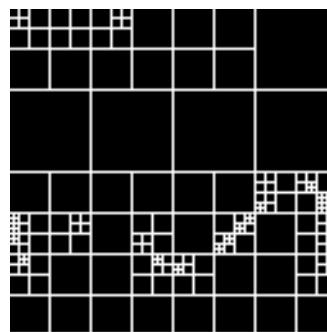
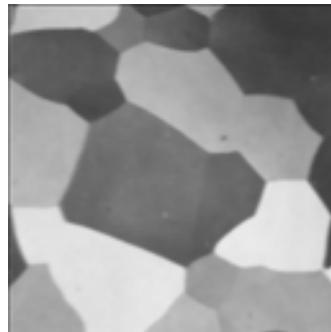
The call to `qtdecomp` for this example would be:

$$S = \text{qtdecomp}(I, 0.2)$$

`S` is returned as a sparse matrix whose nonzero elements represent the upper-left corners of the blocks; the value of each nonzero element indicates the block size. `S` is the same size as `I`.

Note that the threshold value is specified as a value between 0 and 1, even if `I` is of class `uint8`. If `I` is `uint8`, the threshold value you supply is multiplied by 255 to determine the actual threshold to use.

The example below shows an image and a representation of its quadtree decomposition. Each black square represents a homogeneous block, and the white lines represent the boundaries between blocks. Notice how the blocks are smaller in areas corresponding to large changes in intensity in the image.



You can also supply `qtdecomp` with a function (rather than a threshold value) for deciding whether to split blocks; for example, you might base the decision

on the variance of the block. See the reference entry for `qtdecomp` in Chapter 11 for more information.

For an interactive demonstration of quadtree decomposition, try running `qtdemo`.

Image Enhancement

Image enhancement techniques are used to improve an image, where “improve” is sometimes defined objectively (e.g., increase the signal-to-noise ratio), and sometimes subjectively (e.g., make certain features easier to see by modifying the colors or intensities).

This section discusses these image enhancement techniques:

- Intensity adjustment
- Noise removal

The functions described in this section apply primarily to grayscale intensity images. However, some of these functions can be applied to color images as well. For information about how these functions work with color images, see the reference entries for the individual functions in Chapter 11.

Intensity Adjustment

Intensity adjustment is a technique for mapping an image’s intensity values to a new range. For example, look at the rice image and its histogram on page 7-9. Notice that the image has rather low contrast, and that the histogram indicates no values below 40 or above 255. If you remap the data values to fill the entire intensity range [0,255], you can increase the contrast of the image.

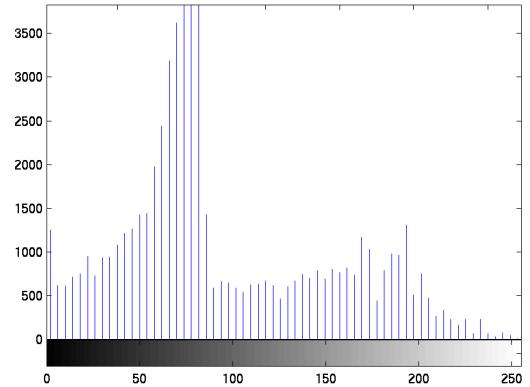
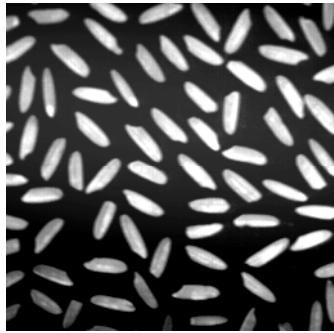
You can do this kind of adjustment with the `imadjust` function. For example, this code performs the adjustment described above:

```
I = imread('rice.tif');
J = imadjust(I, [0.15 0.9], [0 1]);
```

Notice that the intensities are specified as values between 0 and 1, even if `I` is of class `uint8`. If `I` is `uint8`, the values you supply are multiplied by 255 to determine the actual values to use.

Now display the adjusted image and its histogram:

```
imshow(J)
figure, imhist(J, 64)
```



Notice the increased contrast in the image, and that the histogram now fills the entire range.

Similarly, you can decrease the contrast of an image by narrowing the range of the data, as in this call:

```
J = imadjust(I, [0 1], [0.3 0.8]);
```

The general syntax is:

```
J = imadjust(I, [low high], [bottom top])
```

where `low` and `high` are the intensities in the input image, which are mapped to `bottom` and `top` in the output image.

In addition to increasing or decreasing contrast, you can perform a wide variety of other image enhancements with `imadjust`. In the example below, the man's coat is too dark to reveal any detail. The call to `imadjust` maps the range [0,51] in the original image to [128,255] in the output image. This brightens the

image considerably, and also widens the dynamic range of the dark portions of the original image, making it much easier to see the details in the coat.

```
I = imread('cameraman.tif');
J = imadjust(I, [0 0.2], [0.5 1]);
imshow(I)
figure, imshow(J)
```



Notice that this operation results in much of the image being washed out. This is because all values above 51 in the original image get mapped to 255 in the adjusted image.

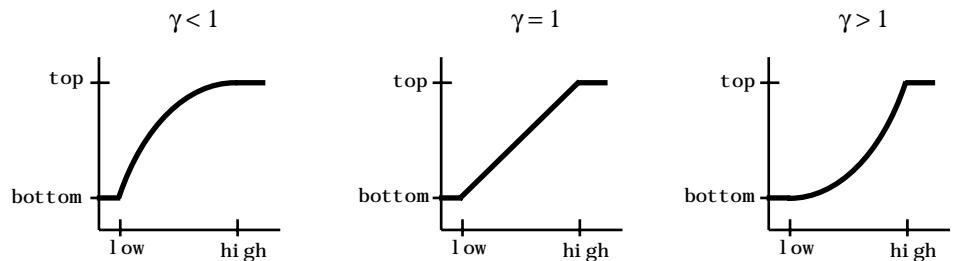
Gamma Correction

`imadjust` maps `low` to `bottom`, and `high` to `top`. By default, the values between `low` and `high` are mapped linearly to values between `bottom` and `top`. For example, the value halfway between `low` and `high` corresponds to the value halfway between `bottom` and `top`.

`imadjust` can accept an additional argument which specifies the *gamma correction* factor. Depending on the value of gamma, the mapping between values in the input and output images may be nonlinear. For example, the value halfway between `low` and `high` may map to a value either greater than or less than the value halfway between `bottom` and `top`.

Gamma can be any value between 0 and infinity. If gamma is 1 (the default), the mapping is linear. If gamma is less than 1, the mapping is weighted toward higher (brighter) output values. If gamma is greater than 1, the mapping is weighted toward lower (darker) output values.

The figure below illustrates this relationship. The three transformation curves show how values are mapped when gamma is less than, equal to, and greater than 1. (In each graph, the x-axis represents the intensity values in the input image, and the y-axis represents the intensity values in the output image.)



The example below illustrates gamma correction. Notice that in the call to `imadjust`, the data ranges of the input and output images are specified as empty matrices. When you specify an empty matrix, `imadjust` uses the default range of [0,1]. In the example, both ranges are left empty; this means that gamma correction is applied without any other adjustment of the data.

```
[X, map] = imread('forest.tif')
I = ind2gray(X, map);
J = imadjust(I, [], [], 0.5);
imshow(I)
figure, imshow(J)
```



Histogram Equalization

The process of adjusting intensity values can be done automatically by the `histeq` function. `histeq` performs *histogram equalization*, which involves transforming the intensity values so that the histogram of the output image approximately matches a specified histogram. (By default, `histeq` tries to match a flat histogram with 64 bins, but you can specify a different histogram instead; see the reference entry for `histeq` in Chapter 11.)

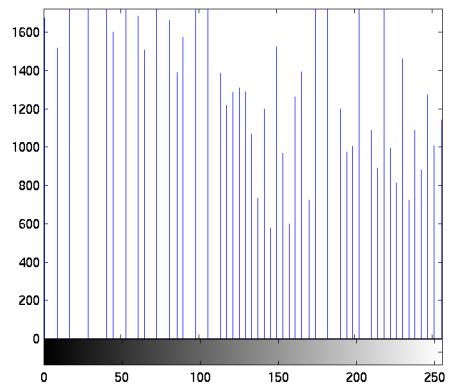
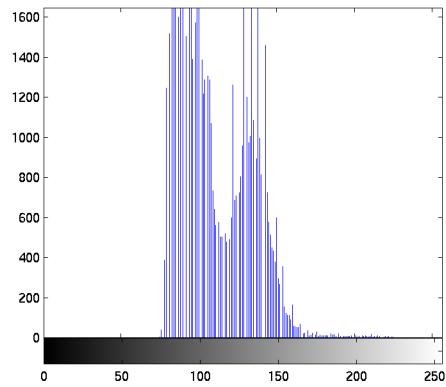
This example illustrates using `histeq` to adjust an intensity image. The original image has low contrast, with most values in the middle of the intensity range. `histeq` produces an output image having values evenly distributed throughout the range.

```
I = imread('pout.tif');
J = histeq(I);
imshow(I)
figure, imshow(J)
```



The example below shows the histograms for the two images:

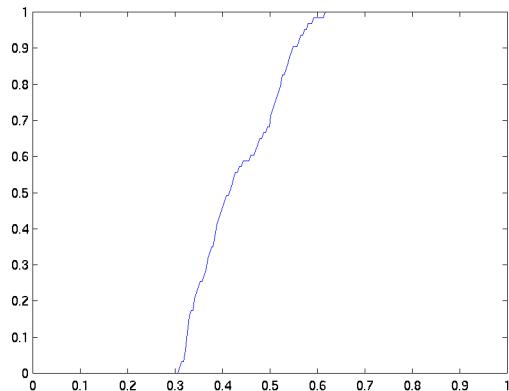
```
imhist(I)  
figure, imhist(J)
```



`histeq` can return an additional 1-by-256 vector that shows, for each possible input value, the resulting output value. (The values in this vector are in the

range [0,1], even if the image is of class uint8.) You can plot this data to get the transformation curve. For example:

```
I = imread('pout.tif');
[J, T] = histeq(I);
plot((0:255)/255, T);
```



Notice how this curve reflects the histograms in the previous figure, with the input values being mostly between 0.3 and 0.6, while the output values are distributed evenly between 0 and 1.

For an interactive demonstration of intensity adjustment, try running `iadj demo`.

Noise Removal

Digital images are prone to a variety of types of noise. There are several ways that noise can be introduced into an image, depending on how the image is created. For example:

- If the image is scanned from a photograph made on film, the film grain is a source of noise. Noise can also be the result of damage to the film, or be introduced by the scanner itself.
- If the image is acquired directly in a digital format, the mechanism for gathering the data (such as a CCD detector) can introduce noise.
- Electronic transmission of image data can introduce noise.

The toolbox provides a number of different ways to remove or reduce noise in an image. Different methods are better for different kinds of noise. The methods available include:

- Linear filtering
- Median filtering
- Adaptive filtering

Also, in order to simulate the effects of some of the problems listed above, the toolbox provides the `imnoise` function, which you can use to *add* various types of noise to an image. The examples in this section use this function.

Linear Filtering

You can use linear filtering to remove certain types of noise. Certain filters, such as averaging or Gaussian filters, are appropriate for this purpose. For example, an averaging filter is useful for removing grain noise from a photograph. Because each pixel gets set to the average of the pixels in its neighborhood, local variations caused by grain are reduced.

For more information about linear filtering, see Chapter 5.

Median Filtering

Median filtering is similar to using an averaging filter, in that each output pixel is set to an “average” of the pixel values in the neighborhood of the corresponding input pixel. However, with median filtering, the value of an output pixel is determined by the *median* of the neighborhood pixels, rather than the mean. The median is much less sensitive than the mean to extreme values (called *outliers*). Median filtering is therefore better able to remove these outliers without reducing the sharpness of the image.

The `medfilt2` function implements median filtering. The example below compares using an averaging filter and `medfilt2` to remove *salt and pepper* noise. This type of noise consists of random pixels being set to black or white (the extremes of the data range). In both cases the size of the neighborhood used for filtering is 3-by-3.

First, read in the image and add noise to it.

```
I = imread('eight.tif');
J = imnoise(I, 'salt & pepper', 0.02);
imshow(I)
figure, imshow(J)
```



Now filter the noisy image and display the results. Notice that `medfilt2` does a better job of removing noise, with less blurring of edges.

```
K = filter2(fspecial ('average', 3), J)/255;
L = medfilt2(J, [3 3]);
imshow(K)
figure, imshow(L)
```



Median filtering is a specific case of *order-statistic filtering*. For information about order-statistic filtering, see the reference entry for the `ordfilt2` function in Chapter 11.

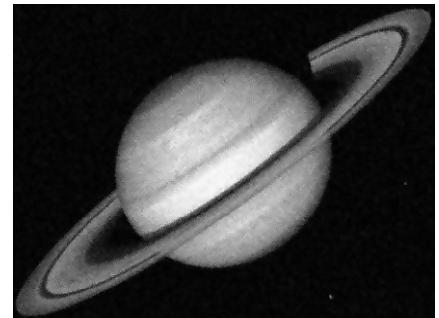
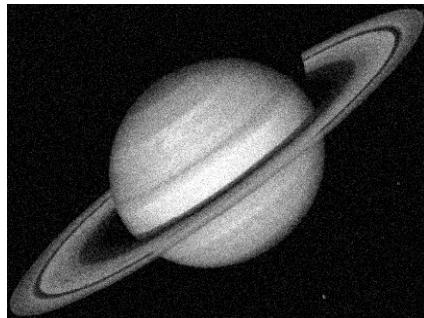
Adaptive Filtering

The `wiener2` function applies a Wiener filter (a type of linear filter) to an image *adaptively*, tailoring itself to the local image variance. Where the variance is large, `wiener2` performs little smoothing. Where the variance is small, `wiener2` performs more smoothing.

This approach often produces better results than linear filtering. The adaptive filter is more selective than a comparable linear filter, preserving edges and other high frequency parts of an image. In addition, there are no design tasks; the `wiener2` function handles all preliminary computations, and implements the filter for an input image. `wiener2`, however, does require more computation time than linear filtering.

`wiener2` work best when the noise is constant-power (“white”) additive noise, such as Gaussian noise. This example applies `wiener2` to an image of Saturn that has had Gaussian noise added:

```
I = imread('saturn.tif');
J = imnoise(I, 'gaussian', 0, 0.005);
K = wiener2(J, [5 5]);
imshow(J)
figure, imshow(K)
```



For an interactive demonstration of filtering to remove noise, try running `nrfiltdemo`.

Binary Image Operations

Overview	8-2
Neighborhoods	8-2
Padding of Borders	8-2
Displaying Binary Images	8-3
Morphological Operations	8-4
Dilation and Erosion	8-4
Related Operations	8-7
Object-Based Operations	8-10
4- and 8-Connected Neighborhoods	8-10
Perimeter Determination	8-12
Flood Fill	8-13
Connected-Components Labeling	8-15
Object Selection	8-16
Feature Measurement	8-18
Image Area	8-18
Euler Number	8-19
Lookup-Table Operations	8-20

Overview

A binary image is an image in which each pixel assumes one of only two discrete values. Essentially, these two values correspond to on and off. Looking at an image in this way makes it easier to distinguish structural features. For example, in a binary image, it is easy to distinguish objects from the background.

In the Image Processing Toolbox, a binary image is stored as a two-dimensional matrix of 0's (which represent off pixels) and 1's (which represent on pixels). The on pixels are the foreground of the image, and the off pixels are the background.

Binary image operations return information about the form or structure of binary images only. To perform these operations on another type of image, you must first convert it to binary (using, for example, the `im2bw` function).

Neighborhoods

Most binary image algorithms work with groups of pixels called *neighborhoods*. A pixel's neighborhood is some set of pixels, defined by their locations relative to that pixel. The neighborhood can include or omit the pixel itself, and the pixels included in the neighborhood are not necessarily adjacent to the pixel of interest. Different types of neighborhoods are used for different binary operations.

Padding of Borders

If a pixel is near the border of an image, some of the pixels in its neighborhood may be missing. For example, if the neighborhood is defined to include the pixel directly above the pixel of interest, then a pixel in the top row of an image will be missing this neighbor.

In order to determine how to process these pixels, the binary image functions *pad* the borders of the image, usually with 0's. In other words, these functions process the border pixels by assuming that the image is surrounded by additional rows and columns of 0's. These rows and columns do not become part of the output image and are used only as parts of the neighborhoods of the actual pixels in the image. However, the padding can in some cases produce *border effects*, in which the regions near the borders of the output image do not appear to be homogeneous with the rest of the image.

Displaying Binary Images

When you display a binary image with `imshow`, by default the foreground (i.e., the on pixels) is white and the background is black. You may prefer to invert these images when you display or print them, or else display them using a colormap. See Chapter 2 for more information about displaying binary images.

The remainder of this chapter describes the functions in the Image Processing Toolbox that perform various types of binary image operations. These operations include:

- Morphological operations
- Object-based operations
- Feature measurement
- Lookup table operations

Morphological Operations

Morphological operations are methods for processing binary images based on shapes. These operations take a binary image as input, and return a binary image as output. The value of each pixel in the output image is based on the corresponding input pixel and its neighbors. By choosing the neighborhood shape appropriately, you can construct a morphological operation that is sensitive to specific shapes in the input image.

Dilation and Erosion

The main morphological operations are *dilation* and *erosion*. Dilation and erosion are related operations, although they produce very different results. Dilation adds pixels to the boundaries of objects (i.e., changes them from off to on), while erosion removes pixels on object boundaries (changes them from on to off).

Each dilation or erosion operation uses a specified neighborhood. The state of any given pixel in the output image is determined by applying a rule to the neighborhood of the corresponding pixel in the input image. The rule used defines the operation as a dilation or an erosion:

- For dilation, if *any* pixel in the input pixel's neighborhood is on, the output pixel is on. Otherwise, the output pixel is off.
- For erosion, if *every* pixel in the input pixel's neighborhood is on, the output pixel is on. Otherwise, the output pixel is off.

The neighborhood for a dilation or erosion operation can be of arbitrary shape and size. The neighborhood is represented by a *structuring element*, which is a matrix consisting of only 0's and 1's. The *center pixel* in the structuring element represents the pixel of interest, while the elements in the matrix that are on (i.e., = 1) define the neighborhood.

The center pixel is defined as $\text{floor}((\text{size}(SE)+1)/2)$, where SE is the structuring element. For example, in a 4-by-7 structuring element, the center pixel is (2,4). When you construct the structuring element, you should make sure that the pixel of interest is actually the center pixel. You can do this by adding rows or columns of 0's, if necessary. For example, suppose you want the neighborhood to consist of a 3-by-3 block of pixels, with the pixel of interest in the upper-left corner of the block. The structuring element would not be

ones(3), because this matrix has the wrong center pixel. Rather, you could use this matrix as the structuring element:

$$\begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{matrix}$$

For erosion, the neighborhood consists of the on pixels in the structuring element. For dilation, the neighborhood consists of the on pixels in the structuring element rotated 180 degrees. (The center pixel is still selected before the rotation.)

Suppose you want to perform an erosion operation. The figure below shows a sample neighborhood you might use. Each neighborhood pixel is indicated by an x, and the center pixel is the one with a circle:

x				
	x			
		(x)		
			x	
				x

The structuring element is therefore:

$$\begin{matrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{matrix}$$

The state (i.e., on or off) of any given pixel in the output image is determined by applying the erosion rule to the neighborhood pixels for the corresponding pixel in the input image. For example, to determine the state of the pixel (4,6) in the output image:

- Overlay the structuring element on the input image, with the center pixel of the structuring element covering the pixel (4,6).
- Look at the pixels in the neighborhood of the input pixel. These are the five pixels covered by 1's in the structuring element. In this case the pixels are: (2,4), (3,5), (4,6), (5,7), (6,8). If all of these pixels are on, then set the pixel in the output image (4,6) to on. If any of these pixels is off, then set the pixel (4,6) in the output image to off.

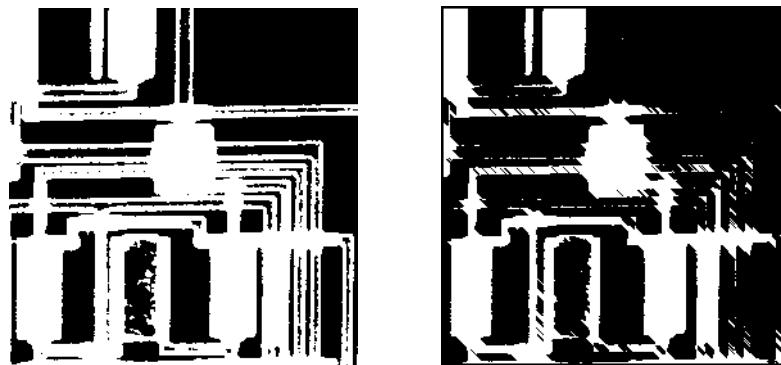
You perform this procedure for each pixel in the input image to determine the state of each corresponding pixel in the output image.

Note that for pixels on borders of the image, some of the 1's in the structuring element are actually outside the image. These elements are assumed to cover off pixels. As a result, the output image will usually have a black border, as in the example below.

The Image Processing Toolbox performs dilation through the `dilate` function, and erosion through the `erode` function. Each of these functions takes an input image and a structuring element as input, and returns an output image.

This example illustrates the erosion operation described above:

```
BW1 = imread('circbw.tif');
SE = eye(5);
BW2 = erode(BW1, SE);
imshow(BW1)
figure, imshow(BW2)
```



Notice the diagonal streaks in the output image (on the right). These are due to the shape of the structuring element.

Related Operations

There are many other types of morphological operations in addition to dilation and erosion. However, many of these operations are just modified dilations or erosions, or combinations of dilation and erosion. For example, *closure* consists of a dilation operation followed by erosion with the same structuring element. A related operation, *opening*, is the reverse of closure; it consists of erosion followed by dilation.

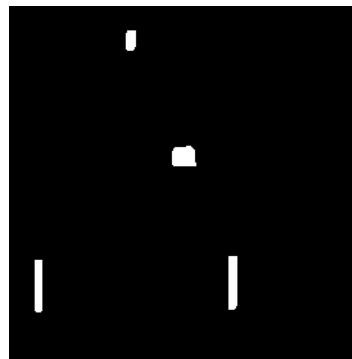
For example, suppose you want to remove all the circuit lines from the original circuit image, leaving only the rectangular outlines of microchips. You can accomplish this through opening.

To perform the opening, you begin by choosing a structuring element. This structuring element should be large enough to remove the lines when you erode the image, but not large enough to remove the rectangles. It should consist of all 1's, so it removes everything but large continuous patches of foreground pixels. Therefore, you create the structuring element like this:

```
SE = ones(40, 30);
```

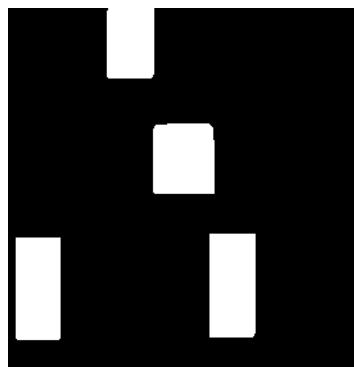
Next, you perform the erosion. This removes all of the lines, but also shrinks the rectangles:

```
BW2 = erode(BW1, SE);  
imshow(BW2)
```



Finally, you perform dilation, using the same structuring element, to restore the rectangles to their original sizes:

```
BW3 = dilate(BW2, SE);  
imshow(BW3)
```

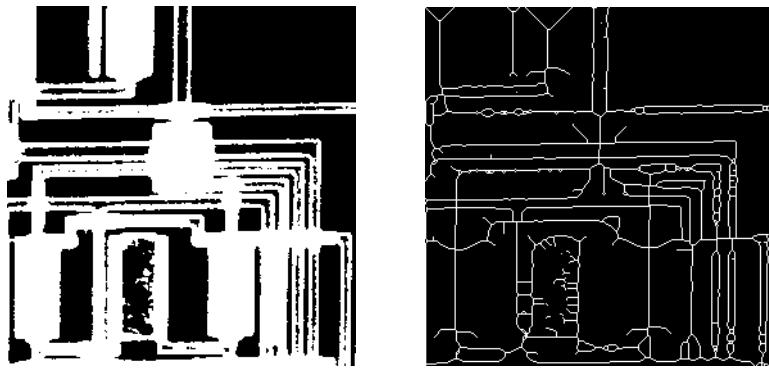


Predefined Operations

You can use `dilate` and `erode` to implement any morphological operation that can be defined as a set of dilations and erosions. However, there are certain operations that are so common that the toolbox provides them as predefined procedures. These operations are available through the `bwmorph` function. `bwmorph` provides eighteen predefined operations, including opening and closure.

For example, suppose you want to reduce all objects in the circuit image to lines, without changing the essential structure (topology) of the image. This process is known as *skeletonization*. You can use bwmorph to do this:

```
BW1 = imread('circ_rcbw.tif');
BW2 = bwmorph(BW1, 'skel', Inf);
imshow(BW1)
figure, imshow(BW2)
```



The third argument to bwmorph indicates the number of times to perform the operation. For example, if this value is 2, the operation is performed twice, with the result of the first operation being used as the input for the second operation. In the example above, the value is Inf. In this case bwmorph performs the operation repeatedly until it no longer changes.

For more information about the predefined operations available, see the reference entry for bwmorph in Chapter 11.

Object-Based Operations

In a binary image, an *object* is any set of connected on pixels. For example, this matrix represents a binary image containing a single object, a 3-by-3 square. The rest of the image is background:

0	0	0	0	0	0
0	0	0	0	0	0
0	1	1	1	0	0
0	1	1	1	0	0
0	1	1	1	0	0
0	0	0	0	0	0

This section discusses the types of neighborhoods used for object-based operations, and describes how to use toolbox functions to perform:

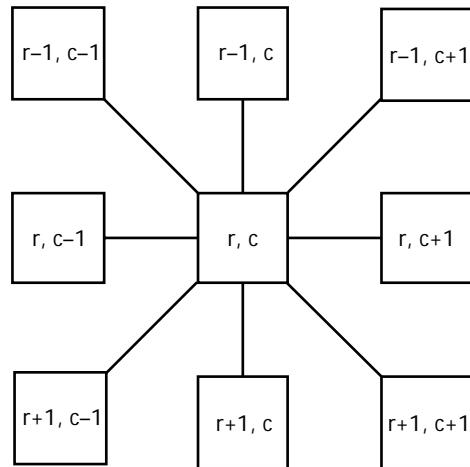
- Perimeter determination
- Binary flood fill
- Connected-components labeling
- Object selection

4- and 8-Connected Neighborhoods

For many operations, distinguishing objects depends on the convention used to decide whether pixels are connected. There are two different conventions typically used: 4-connected or 8-connected neighborhoods.

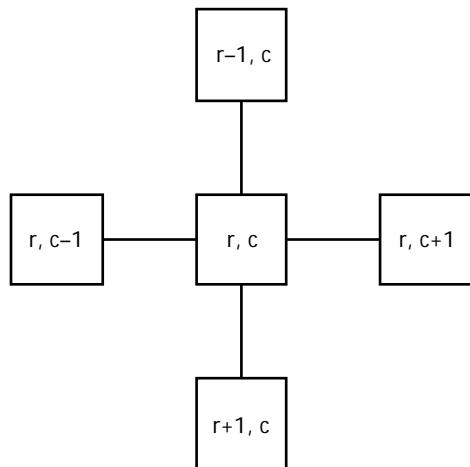
In an 8-connected neighborhood, all of the pixels that touch the pixel of interest are considered, including those on the diagonals. This means that if two adjoining pixels are on, they are part of the same object, regardless of whether they are connected along the horizontal, vertical, or diagonal direction.

This figure illustrates an 8-connected neighborhood:



In a 4-connected neighborhood, the pixels along the diagonals are not considered. This means that two adjoining pixels are part of the same object only if they are both on and are connected along the horizontal or vertical direction.

This figure illustrates a 4-connected neighborhood:



The type of neighborhood you choose affects the number of objects there are in an image and the boundaries of those objects. Therefore, the results of the object-based operations often differ for the two types of neighborhoods.

For example, this matrix represents a binary image that has one 8-connected object or two 4-connected objects:

0	0	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	1	1	0
0	0	0	1	1	0
0	0	0	0	0	0

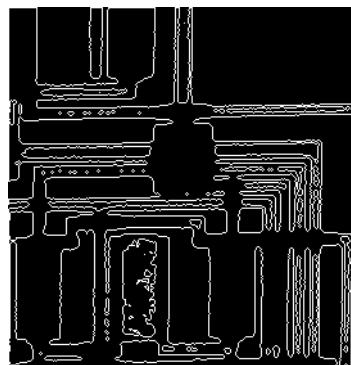
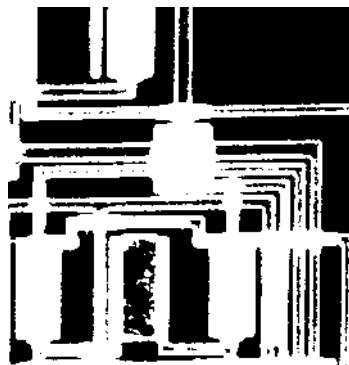
Perimeter Determination

The `bwperim` function determines the perimeter pixels of the objects in a binary image. You can use either a 4- or 8-connected neighborhood for perimeter determination. A pixel is considered a perimeter pixel if it satisfies both of these criteria:

- It is an on pixel.
- One (or more) of the pixels in its neighborhood is off.

This example finds the perimeter pixels in the circuit image:

```
BW1 = imread('circuit.tif');
BW2 = bwperim(BW1);
imshow(BW1)
figure, imshow(BW2)
```



Flood Fill

The `bwfill` function performs a *flood-fill* operation on a binary image. You specify a background pixel as a starting point, and `bwfill` changes connected background pixels (0's) to foreground pixels (1's), stopping when it reaches object boundaries. The boundaries are determined based on the type of neighborhood you specify.

This operation can be useful in removing irrelevant artifacts from images. For example, suppose you have a binary image, derived from a photograph, in which the foreground objects represent spheres. In the binary image, these objects should appear as circles, but instead are donut shaped because of reflections in the original photograph. Before doing any further processing of the image, you may want to first fill in the "donut holes" using `bwfill`.

`bwfill` differs from the other object-based operations in that it operates on background pixels, rather than the foreground. If the foreground is 8-connected, the background is 4-connected, and vice versa. Note, however, that as with the other object-based functions, you specify the connectedness of the foreground when you call `bwfill`.

The implications of 4- vs. 8-connected foreground can be illustrated with this matrix:

`BW1 =`

0	0	0	0	0	0	0	0
0	1	1	1	1	1	0	0
0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	0
0	1	1	1	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Regardless of whether the foreground is 4-connected or 8-connected, this image contains a single object. However, the topology of the object differs depending on the type of neighborhood. If the foreground is 8-connected, the object is a closed contour, and there are two separate background elements (the part inside the loop and the part outside). If the foreground is 4-connected, the contour is open, and there is only one background element.

Suppose you call `bwfill`, specifying the pixel $BW1(4, 3)$ as the starting point:

```
bwfill(BW1, 4, 3)
```

```
ans =
```

0	0	0	0	0	0	0	0
0	1	1	1	1	1	0	0
0	1	1	1	1	1	0	0
0	1	1	1	1	1	0	0
0	1	1	1	1	1	0	0
0	1	1	1	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

`bwfill` fills in just the inside of the loop, because `bwfill` uses an 8-connected foreground by default. If you specify a 4-connected foreground instead, `bwfill` fills in the entire image, because the entire background is a single 8-connected element:

```
bwfill(BW1, 4, 3, 4)
```

```
ans =
```

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

Note that unlike other binary image operations, `bwfill` pads with 1's rather than 0's along the image border. This prevents the fill operation from wrapping around the border.

You can also use `bwfill` interactively, selecting starting pixels with a mouse. See the reference entry for `bwfill` in Chapter 11 for more information about this function.

Connected-Components Labeling

You can use the `bwlabel` function to perform *connected-components labeling*, which is a method for indicating each discrete object in a binary image. You specify the input binary image and the type of neighborhood, and `bwlabel` returns a matrix of the same size as the input image. The different objects in the input image are distinguished by different integer values in the output matrix.

For example, suppose you have this binary image:

`BW1 =`

```
0 0 0 0 0 0 0 0
0 1 1 0 0 1 1 1
0 1 1 0 0 0 1 1
0 1 1 0 0 0 0 0
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0
```

You call `bwlabel`, specifying 4-connected neighborhoods:

`bwlabel(BW1, 4)`

`ans =`

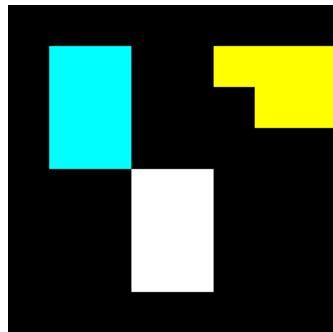
```
0 0 0 0 0 0 0 0
0 1 1 0 0 3 3 3
0 1 1 0 0 0 3 3
0 1 1 0 0 0 0 0
0 0 0 2 2 0 0 0
0 0 0 2 2 0 0 0
0 0 0 2 2 0 0 0
0 0 0 0 0 0 0 0
```

In the output matrix, the 1's represent one object, the 2's a second object, and the 3's a third. Notice that if you had used 8-connected neighborhoods (the default), there would be only two objects, because the first and second objects would be a single object, connected along the diagonal.

The output matrix is not a binary image, and its class is double. A useful approach to viewing it is to display it as a pseudocolor indexed image, first adding 1 to each element, so that the data is in the proper range. Each object displays in a different color, so the objects are easier to distinguish than in the original image.

The example below illustrates this technique. Notice that this example uses a colormap in which the first color (the background) is black and the other colors are easily distinguished:

```
X = bwlabel(BW1, 4);  
map = [0 0 0; jet(3)];  
imshow(X+1, map, 'notruesize')
```



Object Selection

You can use the `bwselect` function to select individual objects in a binary image. You specify pixels in the input image, and `bwselect` returns a binary image that includes only those objects from the input image that contain one of the specified pixels.

You can specify the pixels either noninteractively or with a mouse. For example, suppose you want to select 8-connected objects in the image displayed in the current axes. You type:

```
BW2 = bwselect(8);
```

The cursor changes to a cross-hair when it is over the image. Click on the objects you want to select; `bwselect` displays a small star over each pixel you click on. When you are done, press **Return**. `bwselect` returns a binary image consisting of the objects you selected, and removes the stars.

See the reference entry for `bwselect` in Chapter 11 for more information about this function.

Feature Measurement

When you process an image, you may want to obtain information about how certain features of the image change. For example, when you perform dilation, you may want to determine how many pixels are changed from off to on by the operation, or to see if the number of objects in the image changes. This section describes two functions, `bwarea` and `bweuler`, that return two common measures of binary images: the image area and the Euler number.

In addition, you can use the `imfeature` function, in combination with `bwlabel`, to compute measurements of various features in a binary image. See the reference entry for `imfeature` in Chapter 11 for more information.

Image Area

The `bwarea` function returns the area of a binary image. The area is a measure of the size of the foreground of the image. Roughly speaking, the area is the number of on pixels in the image.

`bwarea` does not simply count the number of on pixels, however. Rather, `bwarea` weights different pixel patterns unequally when computing the area. This weighting compensates for the distortion that is inherent in representing a continuous image with discrete pixels. For example, a diagonal line of 50 pixels is longer than a horizontal line of 50 pixels. As a result of the weighting `bwarea` uses, the horizontal line has area of 50, but the diagonal line has area of 62.5.

This example uses `bwarea` to determine how much the area of the circuit image increases after a dilation operation:

```
BW1 = imread('circbw.tif');
SE = ones(5);
BW2 = dilate(BW1, SE);
increase = (bwarea(BW2) - bwarea(BW1)) / bwarea(BW1);

increase =
0.3456
```

The dilation increases the area by about 35 percent.

For more information about the weighting pattern used by `bwarea`, see the reference entry for `bwarea` in Chapter 11.

Euler Number

The `bweuler` function returns the Euler number for a binary image. The Euler number is a measure of the topology of an image. It is defined as the total number of objects in the image minus the number of holes in those objects. You can use either 4- or 8-connected neighborhoods.

This example computes the Euler number for the circuit image, using 8-connected neighborhoods:

```
BW1 = imread('circbw.tif');
eul = bweuler(BW1, 8)

eul =
-85
```

In this example, the Euler number is negative, indicating that the number of holes is greater than the number of objects.

Lookup-Table Operations

Certain binary image operations can be implemented most easily through lookup tables. A lookup table is a column vector in which each element represents the value to return for one possible combination of pixels in a neighborhood.

You can use the `makelut` function to create lookup tables for various operations. `makelut` creates lookup tables for 2-by-2 and 3-by-3 neighborhoods. This figure illustrates these types of neighborhoods. Each neighborhood pixel is indicated by an `x`, and the center pixel is the one with a circle:

<code>(X)</code>	<code>x</code>
<code>x</code>	<code>x</code>

2-by-2 neighborhood

<code>x</code>	<code>x</code>	<code>x</code>
<code>x</code>	<code>(X)</code>	<code>x</code>
<code>x</code>	<code>x</code>	<code>x</code>

3-by-3 neighborhood

For a 2-by-2 neighborhood, there are 16 possible permutations of the pixels in the neighborhood. Therefore, the lookup table for this operation is a 16-element vector. For a 3-by-3 neighborhood, there are 512 permutations, so the lookup table is a 512-element vector.

Once you create a lookup table, you can use it to perform the desired operation by using the `applylut` function.

The example below illustrates using lookup-table operations to modify an image containing text. You begin by writing a function that returns 1 if three or more pixels in the 3-by-3 neighborhood are 1, and 0 otherwise. You then call `makelut`, passing in this function as the first argument, and using the second argument to specify a 3-by-3 lookup table:

```
f = inline('sum(x(:)) >= 3');
lut = makelut(f, 3);
```

`lut` is returned as a 512-element vector of 1's and 0's. Each value is the output from the function for one of the 512 possible permutations.

You then perform the operation using `applylut`:

```
BW1 = imread('text.tif');
BW2 = applylut(BW1, lut);
imshow(BW1)
figure, imshow(BW2)
```

Cross-Correlation Used
To Locate A Known
Target in an Image

Text Running
In Another
Direction

Cross-Correlation Used
To Locate A Known
Target in an Image

Text Running
In Another
Direction

For information about how `applylut` maps pixel combinations in the image to entries in the lookup table, see the reference entry for `applylut` in Chapter 11.

Note that you cannot use `makelut` and `applylut` for neighborhoods of sizes other than 2-by-2 or 3-by-3. These functions support only 2-by-2 and 3-by-3 neighborhoods, because lookup tables are not practical for neighborhoods larger than 3-by-3. For example, a lookup table for a 4-by-4 neighborhood would have 65,536 entries.

Region-Based Processing

Overview	9-2
Specifying a Region of Interest	9-3
Selecting a Polygon	9-3
Other Selection Methods	9-5
Filtering a Region	9-6
Filling a Region	9-8

Overview

This chapter describes operations that you can perform on a selected region of an image. It discusses these topics:

- Specifying a region
- Filtering a region
- Filling a region

For an interactive demonstration of region-based processing, try running `roi demo`.

Specifying a Region of Interest

A *region of interest* is a portion of an image that you want to filter or perform some other operation on. You define a region of interest by creating a *binary mask*, which is a binary image of the same size as the image you want to process. The mask contains 1's for all pixels that are part of the region of interest, and 0's everywhere else.

This section discusses methods for creating binary masks.

Selecting a Polygon

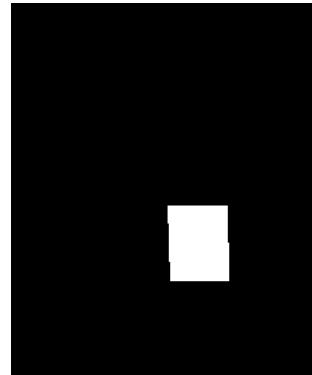
You can use the `roi poly` function to specify a polygonal region of interest. If you call `roi poly` with no input arguments, the cursor changes to a cross hair when it is over the image displayed in the current axes. You can then specify the vertices of the polygon by clicking on points in the image with the mouse. When you are done selecting vertices, press **Return**; `roi poly` returns a binary image, of the same size as the input image, containing 1's inside the specified polygon, and 0's everywhere else.

The example below illustrates using `roi poly` to create a binary mask. The border of the selected region is shown in red on the original image.

```
I = imread('pout.tif');
imshow(I)
BW = roi poly;
```



```
imshow(BW)
```



You can also use `roi poly` noninteractively. See the reference entry for `roi poly` in Chapter 11 for more information.

Other Selection Methods

`roi poly` provides an easy way to create a binary mask. However, you can use *any* binary image as a mask, provided that the binary image is the same size as the image being filtered.

For example, suppose you want to filter the intensity image I , but only filter those pixels whose values are greater than 0.5. You could create the appropriate mask with this command:

```
BW = (I > 0.5);
```

You can also use the `roi col or` function to define the region of interest based on a color or intensity range. For more information, see the reference entry for this function in Chapter 11.

Filtering a Region

You can use the `roi fil t2` function to process a region of interest. When you call `roi fil t2`, you specify an intensity image, a binary mask, and a filter. `roi fil t2` filters the input image, and returns an image that consists of filtered values for pixels where the binary mask contains 1's, and unfiltered values for pixels where the binary mask contains 0's. This type of operation is called *masked filtering*.

This example uses the mask created on page 9-4 to increase the contrast of the logo on the girl's coat.

```
h = fspecial('unsharp');  
I2 = roi fil t2(h, I, BW);  
imshow(I)  
figure, imshow(I2)
```



`roi fil t2` also has a form that you can use to specify a function to operate on the region of interest. In the example below, the `imadjust` function is used to

lighten parts of an image. The mask in the example is a binary image containing text. The resulting image has the text imprinted on it.

```
BW = imread('text.tif');
I = imread('cameraman.tif');
f = inline('imadjust(x,[],[],0.3)');
I2 = roifilt2(I, BW, f);
imshow(I2)
```



Note that `roifilt2` is best suited to operations that return data in the same range as in the original image, because the output image takes some of its data directly from the input image. Certain filtering operations can result in values outside the normal image data range (i.e., [0,1] for images of class `double`, [0,255] for images of class `uint8`). For more information about `roifilt2`, see the reference entry in Chapter 11.

Filling a Region

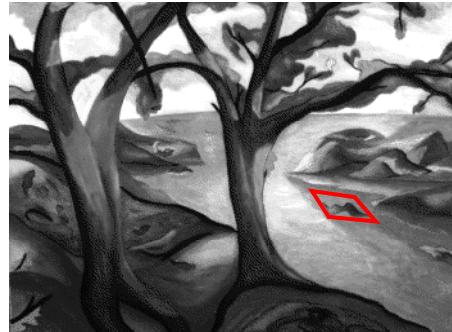
You can use the `roi fill` function to fill a region of interest, interpolating from the borders of the region. This function is useful for image editing, to remove extraneous details or artifacts.

`roi fill` performs the fill using an interpolation method based on Laplace's equation. This method results in the smoothest possible fill, given the values on the boundary of the region.

As with `roi poly`, you select the region of interest with the mouse. When you complete the selection, `roi fill` returns an image with the selected region filled in.

This example uses `roi fill` to modify the `trees` image. The border of the selected region is shown in red on the original image.

```
load trees
I = ind2gray(X, map);
imshow(I)
I2 = roi fill;
```



```
imshow(I2)
```



Color

Overview	10-2
Working with Different Color Depths	10-3
Reducing the Number of Colors in an Image	10-5
Using <code>rgb2ind</code>	10-5
Using <code>imapprox</code>	10-7
Dithering	10-8
Converting to Other Color Spaces	10-9
NTSC Format	10-9
YCbCr Format	10-10
HSV Format	10-10

Overview

This chapter describes toolbox functions that help you work with color image data. It discusses these topics:

- Working on systems with different color depths
- Reducing the number of colors in an image
- Converting an image to a different color space

Note that “color” includes shades of gray, so much of the discussion in this chapter applies to grayscale images as well as color images.

For additional information about how MATLAB handles color, see the *Using MATLAB Graphics* manual.

Working with Different Color Depths

Most computer displays use either 8, 16, or 24 bits per screen pixel. The number of bits per pixel determines how many different colors can be displayed.

24-bit displays provide optimal color resolution. A 24-bit display uses 8 bits for each of the three color components; therefore there are 256 levels each of red, green, and blue. This makes it possible to support 16,777,216 (i.e., 2^{24}) distinct colors. Of these colors, 256 are shades of gray. (These are the colors where R=G=B.)

16-bit displays also provide a large number of colors. A 16-bit display uses 5 bits for each color component, resulting in 32 shades each of red, green, and blue. (Some 16-bit displays use the remaining bit to increase the number of shades of green to 64.) 16-bit displays support 32,768 (i.e., 2^{15}) distinct colors (or twice this number if all 16 bits are used). Of these colors, 32 are shades of gray.

8-bit displays support a much more limited number of colors. An 8-bit display can produce any of the colors available on a 24-bit display, but only 256 distinct colors can appear at one time. There are 256 shades of gray available, but using all of them takes up all of the available color slots.

To determine your system's color depth, enter this MATLAB command:

```
get(0, 'ScreenDepth')
```

MATLAB returns an integer representing the number of bits per pixel. Note that on some systems, MATLAB may return 32. These systems produce 24-bit color; the remaining 8 bits are for an alpha (transparency) channel that MATLAB does not currently use.

Depending on your system, you may be able to choose the color depth you want to use. (There may be tradeoffs between color depth and screen resolution.) In general, 24-bit display mode produces the best results. If you need to use a lower color depth, 16-bit is generally preferable to 8-bit. However, keep in mind that 16-bit color has certain limitations:

- An image may have finer gradations of color than a 16-bit display can represent. If a color is unavailable, MATLAB uses the closest approximation.
- There are only 32 shades of gray available. If you are working primarily with grayscale images, you may get better display results using 8-bit display mode.

Regardless of the number of colors your system can display, MATLAB can store and process images with up to 2^{24} colors (or even more, if the image data is of class `double`). These images display best on systems with 24-bit color, and usually look fine on 16-bit systems as well. (On a 16-bit system, if an image uses colors that the system is not capable of displaying, MATLAB uses color approximation to display the image.)

If your display supports only 8-bit color, you may run into problems due to the limited number of color slots. See Chapter 2 for information about working around these problem. Also, see the next section for information about reducing the number of colors used by an image.

Reducing the Number of Colors in an Image

On systems with 24-bit color, RGB (truecolor) images can display up to 2^{24} (i.e., 16,777,216) colors. On systems with lower color depths, RGB images display reasonably well, because MATLAB automatically uses color approximation and dithering if needed. MATLAB can also read and write these images using `imread` and `imwrite`, without any loss of color resolution.

Indexed images, however, may cause problems if they have a large number of colors. In general, you should limit indexed images to 256 colors, because:

- On systems with 8-bit color, indexed images with more than 256 colors do not display properly.
- On some platforms, colormaps cannot exceed 256 entries.
- If an indexed image has more than 256 colors, MATLAB cannot store the image data in a `uint8` array, and must use an array of class `double` instead.
- Most image file formats limit indexed images to 256 colors. `imwrite` produces an error if you try to write an indexed image with more colors than this to one of these formats.

This section describes how to reduce the number of colors in an indexed or RGB image:

- The `rgb2ind` function converts an RGB image to an indexed image, and provides a number of methods for approximating the original image with fewer colors.
- The `imapprox` function takes an indexed image and returns a new indexed image that uses fewer colors, using the same color-approximation methods that `rgb2ind` uses.

In addition, this section includes a discussion of dithering, which is used by these functions to increase the apparent color resolution of an image.

Using `rgb2ind`

`rgb2ind` converts an RGB image to an indexed image, reducing the number of colors in the process. This function provides these methods for approximating the colors in the original image:

- Quantization
- Colormap mapping

The quality of the resulting image depends on the approximation method you use, the range of colors in the input image, and whether or not you use dithering. Note that different methods work better for different images. See page 10-8 for information about dithering.

Quantization

In general, the most effective approach to reducing the number of colors in an image is *quantization*. This approach involves cutting the RGB color cube into a number of smaller regions, and then mapping all colors that fall within a given region to the value at the center of that region. Quantization ensures that the colors in the output image do not deviate too much from those in the input image, and also enables you to control the maximum number of output colors.

`rgb2ind` supports two quantization methods: *uniform quantization* and *minimum variance quantization*. These methods differ in the approach used to divide up the RGB cube. With uniform quantization, the color cube is cut up into equal-sized cubes. With minimum variance quantization, the color cube is cut up into smaller boxes (not necessarily cubes) of different sizes; the sizes of the boxes depend on how the colors are distributed in the image.

For a given number of colors, minimum variance quantization produces better results than uniform quantization, because it takes into account the actual data. Minimum variance quantization allocates more of the colormap entries to colors that appear frequently in the input image, and fewer to colors that appear infrequently. For example, if the input image has many shades of green and few shades of red, there will be many more greens than reds in the output colormap. The resulting color resolution is higher than with uniform quantization. The computation takes longer, however.

To perform uniform quantization, you call `rgb2ind` and specify a tolerance. The tolerance determines the size of the cubes that the RGB color cube is cut into. For example, if you specify a tolerance of 0.1, the edges of the cubes are one-tenth the length of the RGB cube. The total number of small cubes is:

$$n = \text{floor}(1/\text{tol}) + 1$$

Each cube represents a single color in the output image. Therefore, the maximum length of the colormap is `n`. `rgb2ind` removes any colors that do not appear in the input image, so the actual colormap may be much smaller than `n`.

To perform minimum variance quantization, you call `rgb2ind` and specify the maximum number of colors in the output image's colormap. The number you specify determines how many cubes the RGB color cube is divided into. The resulting colormap usually has the number of entries you specify, because the color cube is divided so that each region contains at least one color that appears in the input image. If the input image actually uses fewer colors than the number you specify, the output colormap is also smaller. In this case, the output image has all of the colors of the input image, and there is no loss of color resolution in the conversion.

These commands use minimum variance quantization to create an indexed image with 185 colors:

```
RGB = imread('flowers.tif');
[X, map] = rgb2ind(RGB, 185);
```

Colormap Mapping

If you specify the actual colormap to use, `rgb2ind` uses *colormap mapping* to find the colors in the specified colormap that best match the colors in the RGB image. This method is useful if you need to create images that use a fixed colormap. For example, if you want to display multiple indexed images on an 8-bit display, you can avoid color problems by having them all use the same colormap. Colormap mapping produces a good approximation if the specified colormap has similar colors to those in the RGB image. If the colormap does not have similar colors to those in the RGB image, this method produces poor results.

This example illustrates mapping two images to the same colormap. The colormap specified includes colors all throughout the RGB cube, so the output images are able to reasonably approximate the input images.

```
RGB1 = imread('autumn.tif');
RGB2 = imread('flowers.tif');
X1 = rgb2ind(RGB1, colordcube(128));
X2 = rgb2ind(RGB2, colordcube(128));
```

Using `i mapprox`

`i mapprox` is based on `rgb2ind`, and provides the same approximation methods. Use `i mapprox` when you need to reduce the number of colors in an indexed image. Essentially, `i mapprox` first calls `i nd2rgb` to convert the image to RGB

format, and then calls `rgb2ind` to return a new indexed image with fewer colors.

For example, these commands create a version of the `trees` image with 64 colors, rather than the original 128:

```
load trees  
[Y, newmap] = imapprox(X, map, 64);
```

Dithering

When you use `rgb2ind` or `imapprox` to reduce the number of colors in an image, the resulting image may look inferior to the original, because some of the colors are lost. `rgb2ind` and `imapprox` both perform *dithering* to increase the apparent color resolution of the output image. Dithering changes the colors of pixels in a neighborhood, so that the average color around each pixel approximates the original RGB color. Dithering provides this extra color resolution at the expense of spatial resolution, and it adds high frequency noise, but it typically results in a more visually pleasing approximation.

If you do not want the output image to be dithered, you can specify this when you call `rgb2ind` or `imapprox`. For example:

```
[X, map] = rgb2ind(RGB, 128, 'nodither');
```

Converting to Other Color Spaces

The Image Processing Toolbox represents colors as RGB values, either directly (in an RGB image) or indirectly (in an indexed image). However, there are other models besides RGB for representing colors numerically. For example, a color can be represented by its hue, saturation, and value components (HSV) instead. The various models for color data are called *color spaces*.

The functions in the Image Processing Toolbox that work with color assume that images use the RGB color space. The toolbox provides support for other color spaces through a set of conversion functions. You can use these functions to convert between RGB and these color spaces:

- National Television Systems Committee (NTSC) format
- YCbCr format
- Hue, saturation, value (HSV) format

This section describes these color spaces and the conversion routines for working with them.

NTSC Format

The NTSC format is used in televisions in the United States. One of the main advantages of this format is that grayscale information is separated from color data, so the same signal can be used for both color and black and white sets. In this NTSC format, image data consists of three components: luminance (Y), hue (I), and saturation (Q). The first component, luminance, represents grayscale information, while the last two components make up chrominance (color information).

The function `rgb2ntsc` converts colormaps or RGB images to the NTSC color space. `ntsc2rgb` performs the reverse operation.

For example, these commands convert the `f1owers` image to NTSC format:

```
RGB = imread('f1owers.tif');
YIQ = rgb2ntsc(RGB);
```

Because luminance is one of the components of the NTSC format, the RGB to NTSC conversion is also useful for isolating the gray level information in an image. The toolbox functions `rgb2gray` and `ind2gray` use the `rgb2ntsc` function to extract the grayscale information from a color image.

For example, these commands are equivalent to calling `rgb2gray`:

```
YIQ = rgb2ntsc(RGB);  
I = YIQ(:,:,1);
```

YCbCr Format

The YCbCr format is widely used for digital video. In this format, luminance information is stored as a single component (Y), and chrominance information is stored as two color difference components (Cb and Cr). Cb represents the difference between the blue component and a reference value, and Cr represents the difference between the red component and a reference value.

YCbCr data can be double precision, but the color space is particularly well suited to `uint8` data. For `uint8` images, the data range for Y is [16,235], and the range for Cb and Cr is [16,240]. YCbCr leaves room at the top and bottom of the full `uint8` range so that additional (non-image) information can be included in a video stream.

The function `rgb2ycbcr` converts colormaps or RGB images to the YCbCr color space. `ycbcr2rgb` performs the reverse operation.

For example, these commands convert the `flowers` image to YCbCr format:

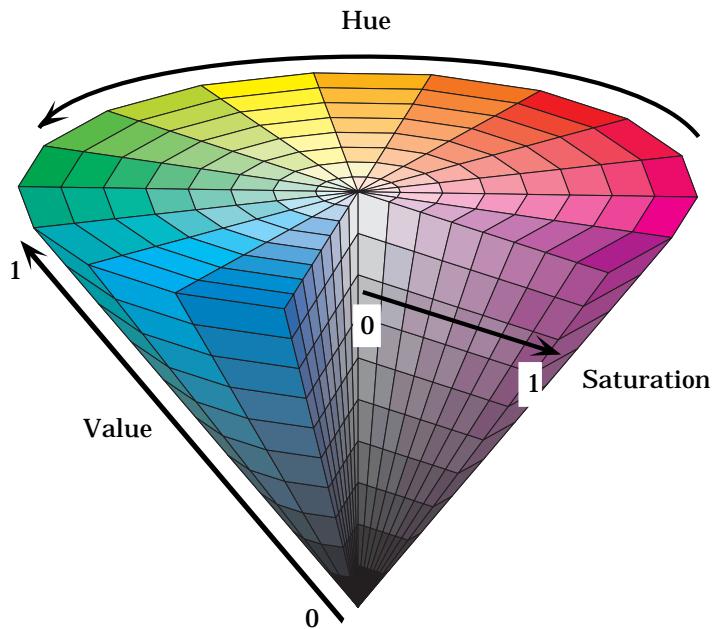
```
RGB = imread('flowers.tif');  
YCBCR = rgb2ycbcr(RGB);
```

HSV Format

The functions `rgb2HSV` and `HSV2RGB` convert images between the RGB and hue, saturation, value (HSV) color spaces. The HSV color space is often used for picking colors (e.g., of paints or inks) from a color wheel or palette, because it corresponds better to how people experience color than the RGB color space does.

As hue varies from 0 to 1.0, the corresponding colors vary from red, through yellow, green, cyan, blue, and magenta, back to red. As saturation varies from 0 to 1.0, the corresponding colors vary from unsaturated (shades of gray) to fully saturated (no white component). As value, or brightness, varies from 0 to 1.0, the corresponding colors become increasingly brighter.

The figure below illustrates the HSV color space:



These commands convert an RGB image to HSV color space:

```
RGB = imread('flowers.tif');
HSV = rgb2hsv(RGB);
```

10 Color

Reference

This chapter provides detailed descriptions of the functions in the Image Processing Toolbox. It begins with a list of functions grouped by subject area and continues with the reference entries in alphabetical order.

The list of functions includes all functions in the Image Processing Toolbox itself, plus a few functions in MATLAB that are especially useful for image processing. All of the functions listed have reference entries, with these exceptions:

- The MATLAB functions `image`, `imagesc`, and `interp2`. For information about these functions, see the online MATLAB Function Reference.
- The Image Processing Toolbox demo functions and slideshow functions. For information about any of these functions, run the demo or slideshow and click the **Info** button.

Image Display

<code>colorbar</code>	Display colorbar (MATLAB)
<code>getimage</code>	Get image data from axes
<code>image</code>	Create and display image object (MATLAB)
<code>imagesc</code>	Scale data and display as image (MATLAB)
<code>immovie</code>	Make movie from multiframe indexed image
<code>imshow</code>	Display image
<code>montage</code>	Display multiple image frames as rectangular montage
<code>subimage</code>	Display multiple images in single figure
<code>truesize</code>	Adjust display size of image
<code>warp</code>	Display image as texture-mapped surface
<code>zoom</code>	Zoom in and out of image or 2-D plot (MATLAB)

Image File I/O

i mfi nfo	Return information about image file (MATLAB)
i mread	Read image file (MATLAB)
i mwri te	Write image file (MATLAB)

Geometric Operations

i mcrop	Crop image
i mresi ze	Resize image
i mrotate	Rotate image
i nterp2	2-D data interpolation (MATLAB)

Pixel Values and Statistics

corr2	Compute 2-D correlation coefficient
i mcontour	Create contour plot of image data
i mf eature	Compute feature measurements for image regions
i nhist	Display histogram of image data
i mpi xel	Determine pixel color values
i mprof il e	Compute pixel-value cross-sections along line segments
mean2	Compute mean of matrix elements
pi xval	Display information about image pixels
std2	Compute standard deviation of matrix elements

Image Analysis

<code>edge</code>	Find edges in intensity image
<code>qtdecomp</code>	Perform quadtree decomposition
<code>qtgetblk</code>	Get block values in quadtree decomposition
<code>qtsetblk</code>	Set block values in quadtree decomposition

Image Enhancement

<code>histeq</code>	Enhance contrast using histogram equalization
<code>imadjust</code>	Adjust image intensity values or colormap
<code>imnoise</code>	Add noise to an image
<code>medfilt2</code>	Perform 2-D median filtering
<code>ordfilt2</code>	Perform 2-D order-statistic filtering
<code>winer2</code>	Perform 2-D adaptive noise-removal filtering

Linear Filtering

<code>conv2</code>	Perform 2-D convolution (MATLAB)
<code>convmtx2</code>	Compute 2-D convolution matrix
<code>convn</code>	Perform N-D convolution (MATLAB)
<code>filter2</code>	Perform 2-D filtering (MATLAB)
<code>fspecial</code>	Create predefined filters

Linear 2-D Filter Design

<code>freqspace</code>	Determine 2-D frequency response spacing (MATLAB)
<code>freqz2</code>	Compute 2-D frequency response
<code>fsamp2</code>	Design 2-D FIR filter using frequency sampling
<code>ftrans2</code>	Design 2-D FIR filter using frequency transformation
<code>fwind1</code>	Design 2-D FIR filter using 1-D window method
<code>fwind2</code>	Design 2-D FIR filter using 2-D window method

Image Transforms

<code>dct2</code>	Compute 2-D discrete cosine transform
<code>dctmtx</code>	Compute discrete cosine transform matrix
<code>fft2</code>	Compute 2-D fast Fourier transform (MATLAB)
<code>fftn</code>	Compute N-D fast Fourier transform (MATLAB)
<code>fftshift</code>	Reverse quadrants of output of FFT (MATLAB)
<code>idct2</code>	Compute 2-D inverse discrete cosine transform
<code>ifft2</code>	Compute 2-D inverse fast Fourier transform (MATLAB)
<code>ifftn</code>	Compute N-D inverse fast Fourier transform (MATLAB)
<code>iradon</code>	Compute inverse Radon transform
<code>phantom</code>	Generate a head phantom image
<code>radon</code>	Compute Radon transform

Neighborhood and Block Processing

bestblk	Choose block size for block processing
blkproc	Implement distinct block processing for image
col2im	Rearrange matrix columns into blocks
colfilt	Perform neighborhood operations using columnwise functions
im2col	Rearrange image blocks into columns
nlfilter	Perform general sliding-neighborhood operations

Binary Image Operations

applylut	Perform neighborhood operations using lookup tables
bwarea	Compute area of objects in binary image
bweuler	Compute Euler number of binary image
bwfill	Fill background regions in binary image
bwlabel	Label connected components in binary image
bwmorph	Perform morphological operations on binary image
bwperim	Determine perimeter of objects in binary image
bwsel ect	Select objects in binary image
dilate	Perform dilation on binary image
erode	Perform erosion on binary image
makelut	Construct lookup table for use with applylut

Region-Based Processing

roi col or	Select region of interest, based on color
roi fil1	Smoothly interpolate within arbitrary region
roi fil2	Filter a region of interest
roi pol y	Select polygonal region of interest

Colormap Manipulation

brighten	Brighten or darken colormap (MATLAB)
cmpermute	Rearrange colors in colormap
cmunique	Find unique colormap colors and corresponding image
colormap	Set or get color lookup table (MATLAB)
imapprox	Approximate indexed image by one with fewer colors
rgbplot	Plot RGB colormap components (MATLAB)

Color Space Conversions

hsv2rgb	Convert HSV values to RGB color space (MATLAB)
ntsc2rgb	Convert NTSC values to RGB color space
rgb2hsv	Convert RGB values to HSV color space (MATLAB)
rgb2ntsc	Convert RGB values to NTSC color space
rgb2ycbcr	Convert RGB values to YCbCr color space
ycbcr2rgb	Convert YCbCr values to RGB color space

Image Types and Type Conversions

<code>di ther</code>	Convert image using dithering
<code>gray2i nd</code>	Convert intensity image to indexed image
<code>graysl i ce</code>	Create indexed image from intensity image by thresholding
<code>i m2bw</code>	Convert image to binary image by thresholding
<code>i m2doubl e</code>	Convert image array to double precision
<code>i m2ui nt8</code>	Convert image array to 8-bit unsigned integers
<code>i nd2gray</code>	Convert indexed image to intensity image
<code>i nd2rgb</code>	Convert indexed image to RGB image
<code>i sbw</code>	Return true for binary image
<code>i sgray</code>	Return true for intensity image
<code>i si nd</code>	Return true for indexed image
<code>i srgb</code>	Return true for RGB image
<code>mat2gray</code>	Convert matrix to intensity image
<code>rgb2gray</code>	Convert RGB image or colormap to grayscale
<code>rgb2i nd</code>	Convert RGB image to indexed image

Toolbox Preferences

<code>i ptgetpref</code>	Get value of Image Processing Toolbox preference
<code>i ptsetpref</code>	Set value of Image Processing Toolbox preference

Demos

dct demo	2-D DCT image compression demo
edgedemo	Edge detection demo
fir demo	2-D FIR filtering and filter design demo
i madj demo	Intensity adjustment and histogram equalization demo
nrfilt demo	Noise reduction filtering demo
qt demo	Quadtree decomposition demo
roi demo	Region-of-interest processing demo

Slide Shows

i pss001	Region labeling of steel grains
i pss002	Feature-based logic
i pss003	Correction of nonuniform illumination

applylut

Purpose	Perform neighborhood operations on binary images, using lookup tables
Syntax	$A = \text{applylut}(BW, lut)$
Description	$A = \text{applylut}(BW, lut)$ performs a 2-by-2 or 3-by-3 neighborhood operation on binary image BW by using a lookup table (lut). lut is either a 16-element or 512-element vector returned by <code>makelut</code> . The vector consists of the output values for all possible 2-by-2 or 3-by-3 neighborhoods.
	The values returned in A depend on the values in lut . For example, if lut consists of all 1's and 0's, A will be a binary image.
Class Support	BW and lut can be of class <code>uint8</code> or <code>double</code> . If the class of lut is <code>uint8</code> , or if the elements of lut are all integers between 0 and 255, then the class of A is <code>uint8</code> ; otherwise, the class of A is <code>double</code> .
Algorithm	<code>applylut</code> performs a neighborhood operation on a binary image by producing a matrix of indices into lut , and then replacing the indices with the actual values in lut . The specific algorithm used depends on whether you use 2-by-2 or 3-by-3 neighborhoods.
	2-by-2 neighborhoods
	For 2-by-2 neighborhoods, <code>length(lut)</code> is 16. There are 4 pixels in each neighborhood, and 2 possible states for each pixel, so the total number of permutations is $2^4 = 16$.
	To produce the matrix of indices, <code>applylut</code> convolves the binary image BW with this matrix:
	$\begin{matrix} 8 & 2 \\ 4 & 1 \end{matrix}$
	The resulting convolution contains integer values in the range [0,15]. <code>applylut</code> uses the central part of the convolution, of the same size as BW , and adds 1 to each value to shift the range to [1,16]. It then constructs A by replacing the values in the cells of the index matrix with the values in lut that the indices point to.

3-by-3 neighborhoods

For 3-by-3 neighborhoods, `length(lut)` is 512. There are 9 pixels in each neighborhood, and 2 possible states for each pixel, so the total number of permutations is $2^9 = 512$.

To produce the matrix of indices, `applylut` convolves the binary image `BW` with this matrix:

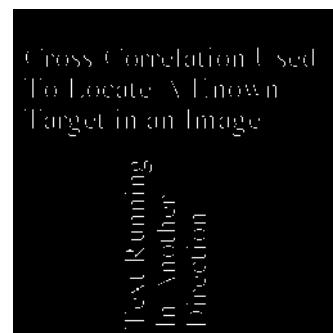
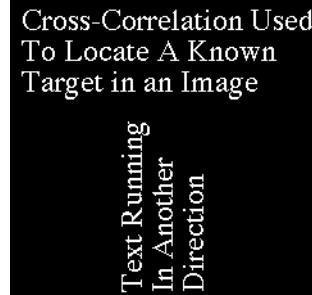
256	32	4
128	16	2
64	8	1

The resulting convolution contains integer values in the range [0,511]. `applylut` uses the central part of the convolution, of the same size as `BW`, and adds 1 to each value to shift the range to [1,512]. It then constructs `A` by replacing the values in the cells of the index matrix with the values in `lut` that the indices point to.

Example

In this example, you perform erosion using a 2-by-2 neighborhood. An output pixel is on only if all four of the input pixel's neighborhood pixels are on.

```
lut = makelut('sum(x(:)) == 4', 2);
BW1 = imread('text.tif');
BW2 = applylut(BW1, lut);
imshow(BW1)
figure, imshow(BW2)
```



See Also

`makelut`

bestblk

Purpose	Determine block size for block processing
Syntax	<pre>sz = bestblk([m n], k) [mb, nb] = bestblk([m n], k)</pre>
Description	<p><code>sz = bestblk([m n], k)</code> returns, for an m-by-n image, the optimal block size for block processing. k is a scalar specifying the maximum row and column dimensions for the block; if the argument is omitted, it defaults to 100. <code>sz</code> is a 1-by-2 vector containing the row and column dimensions for the block.</p> <p><code>[mb, nb] = bestblk([m n], k)</code> returns the row and column dimensions for the block in <code>mb</code> and <code>nb</code>, respectively.</p>
Algorithm	<p><code>bestblk</code> returns the optimal block size given m, n, and k. The algorithm for determining <code>sz</code> is:</p> <ul style="list-style-type: none">• If m is less than or equal to k, return m.• If m is greater than k, consider all values between $\min(m/10, k/2)$ and k. Return the value that minimizes the padding required. <p>The same algorithm is then repeated for n.</p>
Example	<pre>sz = bestblk([640 800], 72) sz =</pre> <pre>64 50</pre>
See Also	<code>blkproc</code>

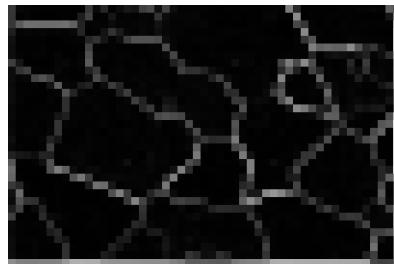
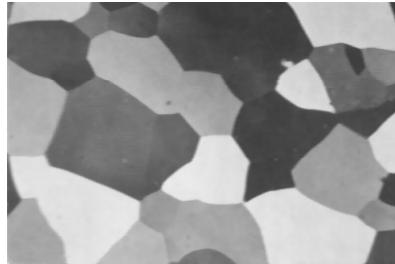
Purpose	Implement distinct block processing for an image
Syntax	<pre>B = blkproc(A, [m n], fun) B = blkproc(A, [m n], fun, P1, P2, ...) B = blkproc(A, [m n], [mborder nborder], fun, ...) B = blkproc(A, 'indexed', ...)</pre>
Description	<p><code>B = blkproc(A, [m n], fun)</code> processes the image A by applying the function fun to each distinct m-by-n block of A, padding A with zeros if necessary. fun can be an inline function, a text string containing the name of a function, or a string containing an expression. fun should operate on an m-by-n block, x, and return a matrix, vector, or scalar y:</p> $y = \text{fun}(x)$ <p><code>blkproc</code> does not require that y be the same size as x. However, B is the same size as A only if y is the same size as x.</p> <p><code>B = blkproc(A, [m n], fun, P1, P2, ...)</code> passes the additional parameters P1, P2, ..., to fun.</p> <p><code>B = blkproc(A, [m n], [mborder nborder], fun, ...)</code> defines an overlapping border around the blocks. <code>blkproc</code> extends the original m-by-n blocks by mborder on the top and bottom, and nborder on the left and right, resulting in blocks of size $(m+2*mborder)$-by-$(n+2*nborder)$. <code>blkproc</code> pads the border with zeros, if necessary, on the edges of A. fun should operate on the extended block.</p> <p>The line below processes an image matrix as 4-by-6 blocks, each having a row border of 2 and a column border of 3. Because each 4-by-6 block has this 2-by-3 border, fun actually operates on blocks of size 8-by-12.</p> <pre>B = blkproc(A, [4 6], [2 3], fun, ...)</pre> <p><code>B = blkproc(A, 'indexed', ...)</code> processes A as an indexed image, padding with zeros if the class of A is <code>uint8</code>, or ones if the class of A is <code>double</code>.</p>
Class Support	The input image A can be of any class supported by fun. The class of B depends on the class of the output from fun.

blkproc

Example

This example uses blkproc to set the pixels in each 8-by-8 block to the standard deviation of the elements in that block.

```
I = imread('alumgrns.tif');
I2 = blkproc(I, [8 8], 'std2(x)*ones(size(x))');
imshow(I)
figure, imshow(I2, []);
```



See Also

[colfilt](#), [nlfilter](#)

Purpose	Brighten or darken a colormap
Syntax	<code>brighten(beta)</code> <code>newmap = brighten(beta)</code> <code>newmap = brighten(map, beta)</code> <code>brighten(fig, beta)</code>
Description	<code>brighten(beta)</code> replaces the current colormap with a brighter or darker map that has essentially the same colors. The map is brighter if $0 < \beta \leq 1$ and darker if $-1 \leq \beta < 0$. <code>brighten(beta)</code> followed by <code>brighten(-beta)</code> restores the original map. <code>newmap = brighten(beta)</code> returns a brighter or darker version of the current colormap without changing the display. <code>newmap = brighten(map, beta)</code> returns a brighter or darker version of the specified colormap without changing the display. <code>brighten(fig, beta)</code> brightens all of the objects in the figure <code>fig</code> .
Remarks	<code>brighten</code> is a function in MATLAB.
See Also	<code>imadjust</code> , <code>rgbplot</code> <code>colormap</code> in the online MATLAB Function Reference

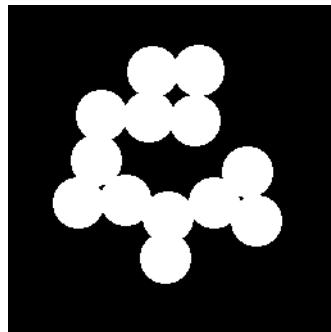
bwarea

Purpose	Compute the area of the objects in a binary image
Syntax	<code>total = bwarea(BW)</code>
Description	<code>total = bwarea(BW)</code> estimates the area of the objects in binary image <code>BW</code> . <code>total</code> is a scalar whose value corresponds roughly to the total number of on pixels in the image, but may not be exactly the same because different patterns of pixels are weighted differently.
Class Support	<code>BW</code> can be of class <code>uint8</code> or <code>double</code> . <code>total</code> is of class <code>double</code> .
Algorithm	<code>bwarea</code> estimates the area of all of the on pixels in an image by summing the areas of each pixel in the image. The area of an individual pixel is determined by looking at its 2-by-2 neighborhood. There are six different patterns distinguished, each representing a different area: <ul style="list-style-type: none">• Patterns with zero on pixels (area = 0)• Patterns with one on pixel (area = 1/4)• Patterns with two adjacent on pixels (area = 1/2)• Patterns with two diagonal on pixels (area = 3/4)• Patterns with three on pixels (area = 7/8)• Patterns with all four on pixels (area = 1) Keep in mind that each pixel is part of four different 2-by-2 neighborhoods. This means, for example, that a single on pixel surrounded by off pixels has a total area of 1.

Example

This example computes the area in the objects of a 256-by-256 binary image.

```
BW = imread('circles.tif');  
imshow(BW);
```



```
bwarea(BW)
```

```
ans =
```

```
15799
```

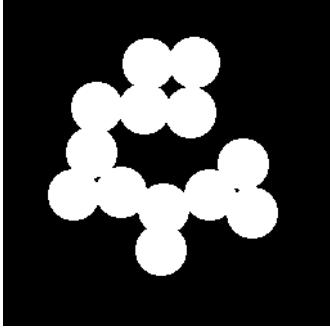
See Also

bweuler, bwperim

References

Pratt, William K. *Digital Image Processing*. New York: John Wiley & Sons, Inc., 1991. p. 634.

bweuler

Purpose	Compute the Euler number of a binary image
Syntax	<code>eul = bweuler(BW, n)</code>
Description	<code>eul = bweuler(BW, n)</code> returns the Euler number for the binary image <code>BW</code> . <code>eul</code> is a scalar whose value is the total number of objects in the image minus the total number of holes in those objects. <code>n</code> can have a value of either 4 or 8, where 4 specifies 4-connected objects and 8 specifies 8-connected objects; if the argument is omitted, it defaults to 8.
Class Support	<code>BW</code> can be of class <code>uint8</code> or <code>double</code> . <code>eul</code> is of class <code>double</code> .
Example	<pre>BW = imread('circles.tif'); imshow(BW);</pre>  <code>bweuler(BW)</code> <pre>ans = -2</pre>
Algorithm	<code>bweuler</code> computes the Euler number by considering patterns of convexity and concavity in local 2-by-2 neighborhoods. See Pratt, p. 633, for a discussion of the algorithm used.
See Also	<code>bwmorph</code> , <code>bwperim</code>

References

- Pratt, William K. *Digital Image Processing*. New York: John Wiley & Sons, Inc., 1991. p. 633.
- Horn, Berthold P. K., *Robot Vision*. New York: McGraw-Hill, 1986. pp. 73-77.

bwfill

Purpose	Fill background regions in a binary image
Syntax	<pre>BW2 = bwfill(BW1, c, r, n) BW2 = bwfill(BW1, n) [BW2, idx] = bwfill(...)</pre> <pre>BW2 = bwfill(x, y, BW1, xi, yi, n) [x, y, BW2, idx, xi, yi] = bwfill(...)</pre> <pre>BW2 = bwfill(BW1, 'holes', n) [BW2, idx] = bwfill(BW1, 'holes', n)</pre>
Description	<p><code>BW2 = bwfill(BW1, c, r, n)</code> performs a flood-fill operation on the input binary image <code>BW1</code>, starting from the pixel <code>(r,c)</code>. If <code>r</code> and <code>c</code> are equal-length vectors, the fill is performed in parallel from the starting pixels <code>(r(k),c(k))</code>. <code>n</code> can have a value of either 4 or 8 (the default), where 4 specifies 4-connected foreground and 8 specifies 8-connected foreground. The foreground of <code>BW1</code> comprises the on pixels (i.e., having value of 1).</p> <p><code>BW2 = bwfill(BW1, n)</code> displays the image <code>BW1</code> on the screen and lets you select the starting points using the mouse. If you omit <code>BW1</code>, <code>bwfill</code> operates on the image in the current axes. Use normal button clicks to add points. Press Backspace or Delete to remove the previously selected point. A shift-click, right-click, or double-click selects a final point and then starts the fill; pressing Return finishes the selection without adding a point.</p> <p><code>[BW2, idx] = bwfill(...)</code> returns the linear indices of all pixels filled by <code>bwfill</code>.</p> <p><code>BW2 = bwfill(x, y, BW1, xi, yi, n)</code> uses the vectors <code>x</code> and <code>y</code> to establish a nondefault spatial coordinate system for <code>BW1</code>. <code>xi</code> and <code>yi</code> are scalars or equal-length vectors that specify locations in this coordinate system.</p> <p><code>[x, y, BW2, idx, xi, yi] = bwfill(...)</code> returns the XData and YData in <code>x</code> and <code>y</code>; the output image in <code>BW2</code>; linear indices of all filled pixels in <code>idx</code>; and the fill starting points in <code>xi</code> and <code>yi</code>.</p> <p><code>BW2 = bwfill(BW1, 'holes', n)</code> fills the holes in the binary image <code>BW1</code>. <code>bwfill</code> automatically determines which pixels are in object holes, and then changes the value of those pixels from 0 to 1. <code>n</code> defaults to 8 if you omit the argument.</p>

[BW2, idx] = bwfill(BW1, 'holes', n) returns the linear indices of all pixels filled in by bwfill.

If bwfill is used with no output arguments, the resulting image is displayed in a new figure.

Remarks

bwfill differs from many other binary image operations in that it operates on background pixels, rather than foreground pixels. If the foreground is 8-connected, the background is 4-connected, and vice versa. Note, however, that you specify the connectedness of the *foreground* when you call bwfill.

Class Support

The input image BW1 can be of class double or uint8. The output image BW2 is of class uint8.

bwfill

Examples

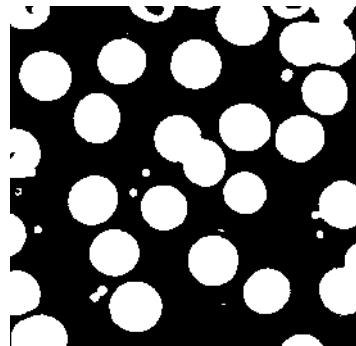
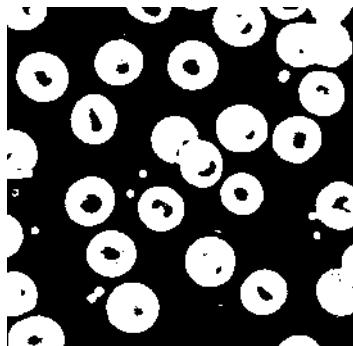
```
BW1 = [ 1     0     0     0     0     0     0     0     0  
        1     1     1     1     1     0     0     0     0  
        1     0     0     0     0     1     0     1     0  
        1     0     0     0     0     1     1     1     0  
        1     1     1     1     0     1     1     1     1  
        1     0     0     1     1     0     1     0     0  
        1     0     0     0     1     0     0     1     0  
        1     0     0     0     1     1     1     1     0 ]
```

```
BW2 = bwfill(BW1, 3, 3, 8)
```

```
BW2 =
```

```
1     0     0     0     0     0     0     0     0  
1     1     1     1     1     0     0     0     0  
1     1     1     1     1     0     1     0     0  
1     1     1     1     1     1     1     1     0  
1     1     1     1     0     1     1     1     1  
1     0     0     1     1     0     1     1     1  
1     0     0     0     1     0     0     1     0  
1     0     0     0     1     1     1     1     0 ]
```

```
I = imread('blood1.tif');  
BW3 = ~im2bw(I);  
BW4 = bwfill(BW3, 'holes');  
imshow(BW3)  
figure, imshow(BW4)
```



See Also

[bwselect](#), [roifill](#)

Purpose	Label connected components in a binary image
Syntax	$L = \text{bwlabel}(BW, n)$ $[L, num] = \text{bwlabel}(BW, n)$
Description	$L = \text{bwlabel}(BW, n)$ returns a matrix L , of the same size as BW , containing labels for the connected objects in BW . n can have a value of either 4 or 8, where 4 specifies 4-connected objects and 8 specifies 8-connected objects; if the argument is omitted, it defaults to 8. The elements of L are integer values greater than or equal to 0. The pixels labeled 0 are the background. The pixels labeled 1 make up one object, the pixels labeled 2 make up a second object, and so on.
	$[L, num] = \text{bwlabel}(BW, n)$ returns in num the number of connected objects found in BW .
Class Support	The input image BW can be of class <code>double</code> or <code>uint8</code> . The output matrix L is of class <code>double</code> .
Remarks	You can use the MATLAB <code>find</code> function in conjunction with <code>bwlabel</code> to return vectors of indices for the pixels that make up a specific object. For example, to return the coordinates for the pixels in object 2: $[r, c] = \text{find}(\text{bwlabel}(BW) == 2)$ You can display the output matrix as a pseudocolor indexed image. Each object appears in a different color, so the objects are easier to distinguish than in the original image. To do this, you must first add 1 to each element in the output matrix, so that the data is in the proper range. Also, it is good idea to use a colormap in which the first few colors are very distinct.

bwlabel

Example

This example illustrates using 4-connected objects. Notice objects 2 and 3; with 8-connected labeling, bwlabel would consider these a single object rather than two separate objects.

```
BW = [ 1      1      1      0      0      0      0      0
       1      1      1      0      1      1      0      0
       1      1      1      0      1      1      0      0
       1      1      1      0      0      0      1      0
       1      1      1      0      0      0      1      0
       1      1      1      0      0      0      1      0
       1      1      1      0      0      0      1      0
       1      1      1      0      0      1      1      0
       1      1      1      0      0      0      0      0]
```

```
L = bwlabel(BW, 4)
```

```
L =
```

```
1      1      1      0      0      0      0      0
1      1      1      0      2      2      0      0
1      1      1      0      2      2      0      0
1      1      1      0      0      0      3      0
1      1      1      0      0      0      3      0
1      1      1      0      0      0      3      0
1      1      1      0      0      0      3      0
1      1      1      0      0      0      0      0
```

```
[r, c] = find(L==2);
rc = [r c]
```

```
rc =
```

```
2      5
3      5
2      6
3      6
```

See Also

bweuler, bwselect

Reference

Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I*. Addison-Wesley, 1992. pp. 28-48.

Purpose	Perform morphological operations on binary images
Syntax	<code>BW2 = bwmorph(BW1, operation)</code> <code>BW2 = bwmorph(BW1, operation, n)</code>
Description	<p><code>BW2 = bwmorph(BW1, operation)</code> applies a specific morphological operation to the binary image <code>BW1</code>.</p> <p><code>BW2 = bwmorph(BW1, operation, n)</code> applies the operation <code>n</code> times. <code>n</code> can be <code>Inf</code>, in which case the operation is repeated until the image no longer changes.</p> <p><code>operation</code> is a string that can have one of these values:</p>

' bothat'	' erode'	' shrink'
' bridge'	' fill'	' skel'
' clean'	' hbreak'	' spur'
' close'	' majority'	' thicken'
' diag'	' open'	' thin'
' dilate'	' remove'	' tophat'

' bothat' ("bottom hat") performs binary closure (dilation followed by erosion) and subtracts the original image.

' bridge' bridges previously unconnected pixels. For example:

$$\begin{array}{ccc} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{array} \quad \text{becomes} \quad \begin{array}{ccc} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{array}$$

' clean' removes isolated pixels (individual 1's that are surrounded by 0's), such as the center pixel in this pattern:

$$\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}$$

' close' performs binary closure (dilation followed by erosion).

' di ag' uses diagonal fill to eliminate 8-connectivity of the background. For example:

0 1 0	0 1 0	
1 0 0	becomes	1 1 0
0 0 0		0 0 0

' dil ate' performs dilation using the structuring element ones(3).

' erode' performs dilation using the structuring element ones(3).

' fill' fills isolated interior pixels (individual 0's that are surrounded by 1's), such as the center pixel in this pattern:

1 1 1
1 0 1
1 1 1

' hbreak' removes H-connected pixels. For example:

1 1 1	1 1 1	
0 1 0	becomes	0 0 0
1 1 1		1 1 1

' maj ority' sets a pixel to 1 if five or more pixels in its 3-by-3 neighborhood are 1's; otherwise, it sets the pixel to 0.

' open' implements binary opening (erosion followed by dilation).

' remove' removes interior pixels. This option sets a pixel to 0 if all of its 4-connected neighbors are 1, thus leaving only the boundary pixels on.

' shri nk' , with n = Inf, shrinks objects to points. It removes pixels so that objects without holes shrink to a point, and objects with holes shrink to a connected ring halfway between each hole and the outer boundary. This option preserves the Euler number.

' skel ' , with n = Inf, removes pixels on the boundaries of objects but does not allow objects to break apart. The pixels remaining make up the image skeleton. This option preserves the Euler number.

'spur' removes spur pixels. For example:

0 0 0 0	0 0 0 0	
0 0 0 0	0 0 0 0	
0 0 1 0	becomes	0 0 0 0
0 1 0 0		0 1 0 0
1 1 0 0		1 1 0 0

'thicken', with $n = \text{Inf}$, thickens objects by adding pixels to the exterior of objects until doing so would result in previously unconnected objects being 8-connected. This option preserves the Euler number.

'thin', with $n = \text{Inf}$, thins objects to lines. It removes pixels so that an object without holes shrinks to a minimally connected stroke, and an object with holes shrinks to a connected ring halfway between each hole and the outer boundary. This option preserves the Euler number.

'tophat' ("top hat") returns the image minus the binary opening of the image.

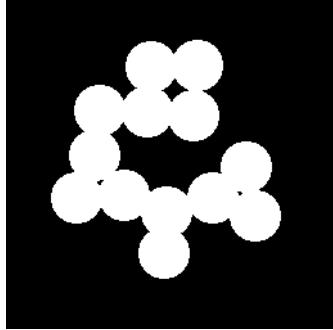
Class Support

The input image `BW1` can be of class `double` or `uint8`. The output image `BW2` is of class `uint8`.

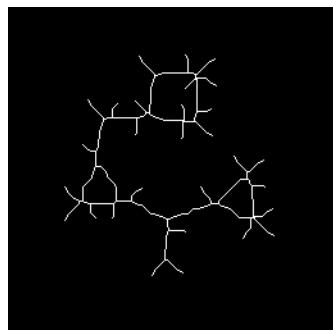
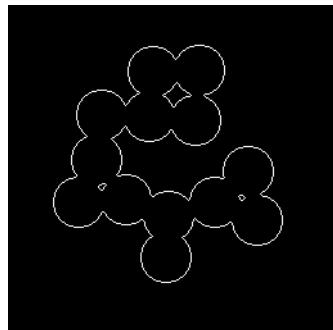
bwmorph

Examples

```
BW1 = imread('circles.tif');  
imshow(BW1);
```



```
BW2 = bwmorph(BW1, 'remove');  
BW3 = bwmorph(BW1, 'skel', Inf);  
imshow(BW2)  
figure, imshow(BW3)
```



See Also

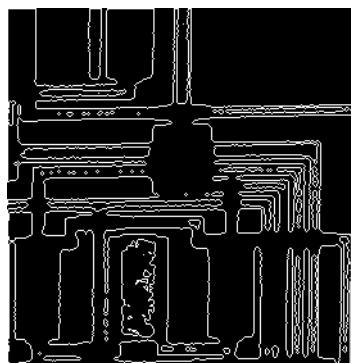
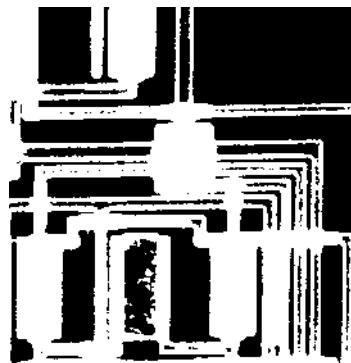
[bweuler](#), [bwperim](#), [dilate](#), [erode](#)

References

Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I*. Addison-Wesley, 1992.

Pratt, William K. *Digital Image Processing*. John Wiley & Sons, Inc., 1991.

Purpose	Determine the perimeter of the objects in a binary image
Syntax	$BW2 = \text{bwperim}(BW1, n)$
Description	$BW2 = \text{bwperim}(BW1, n)$ returns a binary image containing only the perimeter pixels of objects in the input image $BW1$. A pixel is part of the perimeter if its value is 1 and there is at least one zero-valued pixel in its neighborhood. n can have a value of either 4 or 8, where 4 specifies 4-connected neighborhoods and 8 specifies 8-connected neighborhoods; if the argument is omitted, it defaults to 8.
Class Support	The input image $BW1$ can be of class <code>double</code> or <code>uint8</code> . The output image $BW2$ is of class <code>uint8</code> .
Example	<pre>BW1 = imread('circbw.tif'); BW2 = bwperim(BW1, 8); imshow(BW1) figure, imshow(BW2)</pre>



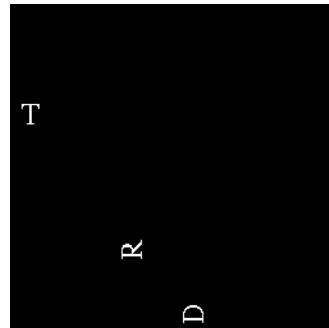
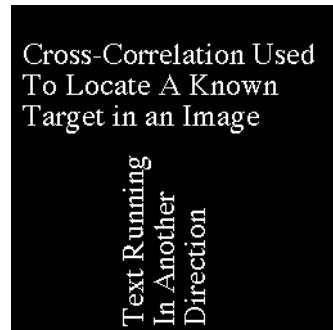
See Also `bwarea`, `bweuler`, `bwfill`

bwselect

Purpose	Select objects in a binary image
Syntax	<pre>BW2 = bwselect(BW1, c, r, n) BW2 = bwselect(BW1, n) [BW2, idx] = bwselect(...)</pre>
	<pre>BW2 = bwselect(x, y, BW1, xi, yi, n) [x, y, BW2, idx, xi, yi] = bwselect(...)</pre>
Description	<p><code>BW2 = bwselect(BW1, c, r, n)</code> returns a binary image containing the objects that overlap the pixel (r, c). r and c can be scalars or equal-length vectors. If r and c are vectors, $BW2$ contains the sets of objects overlapping with any of the pixels $(r(k), c(k))$. n can have a value of either 4 or 8 (the default), where 4 specifies 4-connected objects and 8 specifies 8-connected objects. Objects are connected sets of on pixels (i.e., pixels having a value of 1).</p> <p><code>BW2 = bwselect(BW1, n)</code> displays the image $BW1$ on the screen and lets you select the (r, c) coordinates using the mouse. If you omit $BW1$, <code>bwselect</code> operates on the image in the current axes. Use normal button clicks to add points. Pressing Backspace or Delete removes the previously selected point. A shift-click, right-click, or double-click selects the final point; pressing Return finishes the selection without adding a point.</p> <p><code>[BW2, idx] = bwselect(...)</code> returns the linear indices of the pixels belonging to the selected objects.</p> <p><code>BW2 = bwselect(x, y, BW1, xi, yi, n)</code> uses the vectors x and y to establish a nondefault spatial coordinate system for $BW1$. xi and yi are scalars or equal-length vectors that specify locations in this coordinate system.</p> <p><code>[x, y, BW2, idx, xi, yi] = bwselect(...)</code> returns the XData and YData in x and y; the output image in $BW2$; linear indices of all the pixels belonging to the selected objects in idx; and the specified spatial coordinates in xi and yi.</p> <p>If <code>bwselect</code> is called with no output arguments, the resulting image is displayed in a new figure.</p>

Example

```
BW1 = imread('text.tif');
c = [16 90 144];
r = [85 197 247];
BW2 = bwselect(BW1, c, r, 4);
imshow(BW1)
figure, imshow(BW2)
```

**Class Support**

The input image BW1 can be of class double or uint8. The output image BW2 is of class uint8.

See Also

[bwfill](#), [bwlabel](#), [impixel](#), [roipoly](#), [roifill](#)

cmpermute

Purpose	Rearrange the colors in a colormap
Syntax	<pre>[Y, newmap] = cmpermute(X, map) [Y, newmap] = cmpermute(X, map, index)</pre>
Description	<p><code>[Y, newmap] = cmpermute(X, map)</code> randomly reorders the colors in <code>map</code> to produce a new colormap <code>newmap</code>. <code>cmpermute</code> also modifies the values in <code>X</code> to maintain correspondence between the indices and the colormap, and returns the result in <code>Y</code>. The image <code>Y</code> and associated colormap <code>newmap</code> produce the same image as <code>X</code> and <code>map</code>.</p> <p><code>[Y, newmap] = cmpermute(X, map, index)</code> uses an ordering matrix (such as the second output of <code>sort</code>) to define the order of colors in the new colormap.</p>
Class Support	The input image <code>X</code> can be of class <code>uint8</code> or <code>double</code> . <code>Y</code> is returned as an array of the same class as <code>X</code> .
Example	To order a colormap by luminance, use:
	<pre>ntsc = rgb2ntsc(map); [dum, index] = sort(ntsc(:, 1)); [Y, newmap] = cmpermute(X, map, index);</pre>
See Also	<code>randperm</code> , <code>sort</code> in the online MATLAB Function Reference

Purpose	Find unique colormap colors and the corresponding image
Syntax	<pre>[Y, newmap] = cmuni que(X, map) [Y, newmap] = cmuni que(RGB) [Y, newmap] = cmuni que(I)</pre>
Description	<p>[Y, newmap] = cmuni que(X, map) returns the indexed image Y and associated colormap newmap that produce the same image as (X, map) but with the smallest possible colormap. cmuni que removes duplicate rows from the colormap and adjusts the indices in the image matrix accordingly.</p> <p>[Y, newmap] = cmuni que(RGB) converts the truecolor image RGB to the indexed image Y and its associated colormap newmap. newmap is the smallest possible colormap for the image, containing one entry for each unique color in RGB. (Note that newmap may be very large, because the number of entries can be as many as the number of pixels in RGB.)</p> <p>[Y, newmap] = cmuni que(I) converts the intensity image I to an indexed image Y and its associated colormap newmap. newmap is the smallest possible colormap for the image, containing one entry for each unique intensity level in I.</p>
Class Support	The input image can be of class <code>uint8</code> or <code>double</code> . The class of the output image Y is <code>uint8</code> if the length of newmap is less than or equal to 256. If the length of newmap is greater than 256, Y is of class <code>double</code> .
See Also	<code>gray2ind</code> , <code>rgb2ind</code>

col2im

Purpose	Rearrange matrix columns into blocks
Syntax	<pre>A = col2im(B, [m n], [mm nn], block_type) A = col2im(B, [m n], [mm nn])</pre>
Description	<p>col2im rearranges matrix columns into blocks. block_type is a string with one of these values:</p> <ul style="list-style-type: none">• 'distinct' for m-by-n distinct blocks• 'sliding' for m-by-n sliding blocks (default) <p><code>A = col2im(B, [m n], [mm nn], 'distinct')</code> rearranges each column of B into a distinct m-by-n block to create the matrix A of size mm-by-nn. If $B = [A_{11}(:) \ A_{12}(:) \ A_{21}(:) \ A_{22}(:)]$, where each column has length m*n, then $A = [A_{11} \ A_{12}; A_{21} \ A_{22}]$ where each A_{ij} is m-by-n.</p> <p><code>A = col2im(B, [m n], [mm nn], 'sliding')</code> rearranges the row vector B into a matrix of size $(mm-m+1)$-by-$(nn-n+1)$. B must be a vector of size 1-by-$(mm-m+1) * (nn-n+1)$. B is usually the result of processing the output of <code>im2col(..., 'sliding')</code> using a column compression function (such as sum).</p> <p><code>A = col2im(B, [m n], [mm nn])</code> uses the default block_type of 'sliding'.</p>
Class Support	B can be of class uint8 or double. A is of the same class as B.
See Also	<code>blkproc</code> , <code>colfilt</code> , <code>im2col</code> , <code>nlfilter</code>

Purpose	Perform neighborhood operations using columnwise functions
Syntax	<pre>B = colfilt(A, [m n], block_type, fun) B = colfilt(A, [m n], block_type, fun, P1, P2, ...) B = colfilt(A, [m n], [mblock nblock], block_type, fun, ...) B = colfilt(A, 'indexed', ...)</pre>
Description	<p>colfilt processes distinct or sliding blocks as columns. colfilt can perform similar operations to blkproc and nlfilter, but often executes much faster.</p> <p><code>B = colfilt(A, [m n], block_type, fun)</code> processes the image A by rearranging each m-by-n block of A into a column of a temporary matrix, and then applying the function fun to this matrix. fun can be an inline function, a text string containing the name of a function, or a string containing an expression. colfilt zero pads A, if necessary.</p> <p>Before calling fun, colfilt calls im2col to create the temporary matrix. After calling fun, colfilt rearranges the columns of the matrix back into m-by-n blocks using col2im.</p> <p>block_type is a string with one of these values:</p> <ul style="list-style-type: none"> • 'distinct' for m-by-n distinct blocks • 'sliding' for m-by-n sliding neighborhoods <p><code>B = colfilt(A, [m n], 'distinct', fun)</code> rearranges each m-by-n distinct block of A into a column in a temporary matrix, and then applies the function fun to this matrix. fun must return a matrix of the same size as the temporary matrix. colfilt then rearranges the columns of the matrix returned by fun into m-by-n distinct blocks.</p> <p><code>B = colfilt(A, [m n], 'sliding', fun)</code> rearranges each m-by-n sliding neighborhood of A into a column in a temporary matrix, and then applies the function fun to this matrix. fun must return a row vector containing a single value for each column in the temporary matrix. (Column compression functions such as sum return the appropriate type of output.) colfilt then rearranges the vector returned by fun into a matrix of the same size as A.</p>

colfilt

`B = colfilt(A, [m n],block_type,fun,P1,P2,...)` passes the additional parameters `P1, P2, ...` to `fun`. `colfilt` calls `fun` using:

```
y = fun(x,P1,P2,...)
```

where `x` is the temporary matrix before processing, and `y` is the temporary matrix after processing.

`B = colfilt(A, [m n], [mblock nblock], block_type, fun, ...)` processes the matrix `A` as above, but in blocks of size `mblock`-by-`nblock` to save memory. Note that using the `[mblock nblock]` argument does not change the result of the operation.

`B = colfilt(A, 'indexed', ...)` processes `A` as an indexed image, padding with zeros if the class of `A` is `uint8`, or ones if the class of `A` is `double`.

Class Support The input image `A` can be of any class supported by `fun`. The class of `B` depends on the class of the output from `fun`.

Example This example sets each output pixel to the mean value of the input pixel's 5-by-5 neighborhood:

```
I2 = colfilt(I,[5 5],'sliding','mean');
```

See Also `blkproc`, `col2im`, `im2col`, `nlfilter`

Purpose Display a colorbar

Syntax

```
colorbar('vert')
colorbar('horiz')
colorbar(h)
colorbar
h = colorbar(...)
```

Description

`colorbar('vert')` appends a vertical colorbar to the current axes, resizing the axes to make room for the colorbar. `colorbar` works with both two-dimensional and three-dimensional plots.

`colorbar('horiz')` appends a horizontal colorbar to the current axes.

`colorbar(h)` places the colorbar in the axes `h`. The colorbar is horizontal if the width of the axes is greater than its height.

`colorbar` with no arguments adds a new vertical colorbar or updates an existing one.

`h = colorbar(...)` returns a handle to the colorbar axes.

Remarks

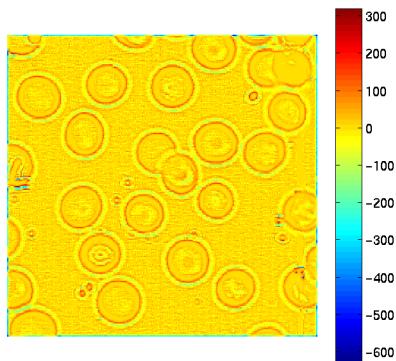
`colorbar` is a function in MATLAB.

colorbar

Examples

Display a colorbar to view values for a filtered image.

```
I = imread('blood1.tif');
h = fspecial('log');
I2 = filter2(h, I);
imshow(I2, []), colormap(jet(64)), colorbar
```



Purpose	Perform two-dimensional convolution
Syntax	$C = \text{conv2}(A, B)$ $C = \text{conv2}(\text{hcol}, \text{hrow}, A)$ $C = \text{conv2}(\dots, \text{shape})$
Description	<p>$C = \text{conv2}(A, B)$ performs the two-dimensional convolution of matrices A and B, returning the result in the output matrix C. The size in each dimension of C is equal to the sum of the corresponding dimensions of the input matrices minus one. That is, if the size of A is $[m_a, m_b]$ and the size of B is $[m_b, n_b]$, then the size of C is $[m_a+m_b-1, n_a+n_b-1]$.</p> <p>$C = \text{conv2}(\text{hcol}, \text{hrow}, A)$ convolves A separably with hcol in the column direction and hrow in the row direction. hcol and hrow are both vectors.</p> <p>$C = \text{conv2}(\dots, \text{shape})$ returns a subsection of the two-dimensional convolution, as specified by the shape parameter. shape is a string with one of these values:</p> <ul style="list-style-type: none"> • '<code>full</code>' (the default) returns the full two-dimensional convolution. • '<code>same</code>' returns the central part of the convolution of the same size as A. • '<code>valid</code>' returns only those parts of the convolution that are computed without the zero-padded edges. Using this option, $\text{size}(C) = [m_a-m_b+1, n_a-n_b+1]$ when $\text{size}(A) > \text{size}(B)$. <p>For image filtering, A should be the image matrix and B should be the filter (convolution kernel) if the shape parameter is '<code>same</code>' or '<code>valid</code>'. If the shape parameter is '<code>full</code>', the order does not matter, because full convolution is commutative.</p>
Class Support	All vector and matrix inputs to <code>conv2</code> can be of class <code>uint8</code> or <code>double</code> . The output matrix C is of class <code>double</code> .
Remarks	<code>conv2</code> is a function in MATLAB.

conv2

Example

```
A = magic(5)
```

```
A =
```

```
17    24    1    8    15  
23     5    7   14    16  
 4     6   13   20    22  
10    12   19   21     3  
11    18   25     2     9
```

```
B = [1 2 1; 0 2 0; 3 1 3]
```

```
B =
```

```
1     2     1  
0     2     0  
3     1     3
```

```
C = conv2(A, B)
```

```
C =
```

```
17    58    66    34    32    38    15  
23    85    88    35    67    76    16  
55   149   117   163   159   135    67  
79    78   160   161   187   129    51  
23    82   153   199   205   108    75  
30    68   135   168    91    84     9  
33    65   126    85   104    15    27
```

See Also

[filter2](#)

[xcorr, xcorr2](#) in the *Signal Processing Toolbox User's Guide*

[conv, deconv](#) in the online MATLAB Function Reference

Purpose Compute two-dimensional convolution matrix

Syntax
`T = convmtx2(H, m, n)`
`T = convmtx2(H, [m n])`

Description `T = convmtx2(H, m, n)` or `T = convmtx2(H, [m n])` returns the convolution matrix `T` for the matrix `H`. If `X` is an `m`-by-`n` matrix, then `reshape(T*X(:, :), size(H)+[m n]-1)` is the same as `conv2(X, H)`.

`T` is returned as a sparse matrix. The number of nonzero elements in `T` is no larger than `prod(size(H)) * m * n`.

See Also `conv2`
`convmtx` in the *Signal Processing Toolbox User's Guide*

convn

Purpose	Perform N-dimensional convolution
Syntax	$C = \text{convn}(A, B)$ $C = \text{convn}(A, B, \text{shape})$
Description	$C = \text{convn}(A, B)$ computes the N-dimensional convolution of matrices A and B. $C = \text{convn}(A, B, \text{shape})$ returns a subsection of the N-dimensional convolution, as specified by the shape parameter. shape is a string with one of these values: <ul style="list-style-type: none">'full' (the default) returns the full convolution.'same' returns the central part of the convolution of the same size as A.'valid' returns only those parts of the convolution that are computed without zero-padded edges.
Class Support	The input matrices A and B can be of class <code>uint8</code> or <code>double</code> . The output matrix C is of class <code>double</code> .
Remarks	<code>convn</code> is a function in MATLAB.
See Also	<code>conv2</code>

Purpose	Compute the two-dimensional correlation coefficient between two matrices
Syntax	<code>r = corr2(A, B)</code>
Description	<code>r = corr2(A, B)</code> computes the correlation coefficient between A and B, where A and B are matrices or vectors of the same size.
Class Support	A and B can be of class <code>uint8</code> or <code>double</code> . r is a scalar of class <code>double</code> .
Algorithm	<code>corr2</code> computes the correlation coefficient using
	$r = \frac{\sum_{m} \sum_{n} (A_{mn} - \bar{A})(B_{mn} - \bar{B})}{\sqrt{\left(\sum_{m} \sum_{n} (A_{mn} - \bar{A})^2\right) \left(\sum_{m} \sum_{n} (B_{mn} - \bar{B})^2\right)}}$
	where $\bar{A} = \text{mean2}(A)$, and $\bar{B} = \text{mean2}(B)$.
See Also	<code>std2</code> <code>corrcoef</code> in the online MATLAB Function Reference

dct2

Purpose	Compute two-dimensional discrete cosine transform
Syntax	$B = \text{dct2}(A)$ $B = \text{dct2}(A, m, n)$ $B = \text{dct2}(A, [m n])$
Description	$B = \text{dct2}(A)$ returns the two-dimensional discrete cosine transform of A . The matrix B is the same size as A and contains the discrete cosine transform coefficients $B(k_1, k_2)$. $B = \text{dct2}(A, m, n)$ or $B = \text{dct2}(A, [m n])$ pads the matrix A with zeros to size m -by- n before transforming. If m or n is smaller than the corresponding dimension of A , dct2 truncates A .
Class Support	A can be of class <code>uint8</code> or <code>double</code> . The returned matrix B is of class <code>double</code> .
Algorithm	The discrete cosine transform (DCT) is closely related to the discrete Fourier transform. It is a separable, linear transformation; that is, the two-dimensional transform is equivalent to a one-dimensional DCT performed along a single dimension followed by a one-dimensional DCT in the other dimension. The definition of the two-dimensional DCT for an input image A and output image B is

$$B_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A_{mn} \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N}, \quad 0 \leq p \leq M-1, \quad 0 \leq q \leq N-1$$
$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \leq p \leq M-1 \end{cases} \quad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \leq q \leq N-1 \end{cases}$$

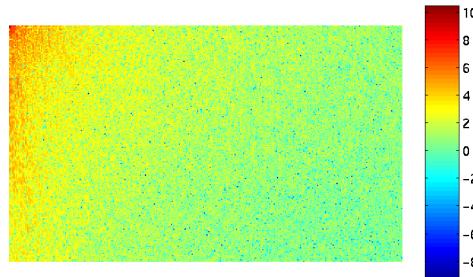
where M and N are the row and column size of A , respectively. If you apply the DCT to real data, the result is also real. The DCT tends to concentrate information, making it useful for image compression applications.

This transform can be inverted using `i dct2`.

Example

The commands below compute the discrete cosine transform for the `autumn` image. Notice that most of the energy is in the upper-left corner.

```
RGB = imread('autumn.tif');
I = rgb2gray(RGB);
J = dct2(I);
imshow(log(abs(J)), []), colormap(jet(64)), colorbar
```



Now set values less than magnitude 10 in the DCT matrix to zero, and then reconstruct the image using the inverse DCT function `i dct2`.

```
J(abs(J) < 10) = 0;
K = idct2(J)/255;
imshow(K)
```

**See Also**

`fft2`, `i dct2`, `i fft2`

References

Jain, Anil K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. pp. 150-153.

Pennebaker, William B., and Joan L. Mitchell. *JPEG: Still Image Data Compression Standard*. Van Nostrand Reinhold, 1993.

dctmtx

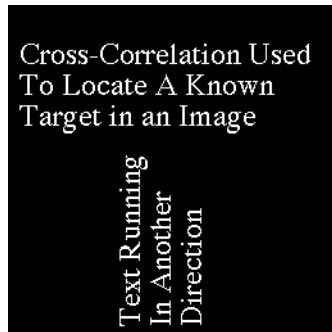
Purpose	Compute discrete cosine transform matrix
Syntax	<code>D = dctmtx(n)</code>
Description	<code>D = dctmtx(n)</code> returns the n-by-n DCT transform matrix. D^*A is the DCT of the columns of A and D'^*A is the inverse DCT of the columns of A (when A is n-by-n).
Class Support	n is a scalar of class double. D is returned as a matrix of class double.
Remarks	If A is square, the two-dimensional DCT of A can be computed as D^*A^*D' . This computation is sometimes faster than using <code>dct2</code> , especially if you are computing a large number of small DCTs, because D needs to be determined only once. For example, in JPEG compression, the DCT of each 8-by-8 block is computed. To perform this computation, use <code>dctmtx</code> to determine D , and then calculate each DCT using D^*A^*D' (where A is each 8-by-8 block). This is faster than calling <code>dct2</code> for each individual block.
See Also	<code>dct2</code>

Purpose	Perform dilation on a binary image
Syntax	$BW2 = \text{dilate}(BW1, SE)$ $BW2 = \text{dilate}(BW1, SE, alg)$ $BW2 = \text{dilate}(BW1, SE, \dots, n)$
Description	$BW2 = \text{dilate}(BW1, SE)$ performs dilation on the binary image $BW1$, using the binary structuring element SE . SE is a matrix containing only 1's and 0's. $BW2 = \text{dilate}(BW1, SE, alg)$ performs dilation using the specified algorithm. alg is a string that can have one of these values: <ul style="list-style-type: none">• 'spatial' (default) – processes the image in the spatial domain.• 'frequency' – processes the image in the frequency domain. Both algorithms produce the same result, but they make different tradeoffs between speed and memory use. The frequency algorithm is faster for large images and structuring elements than the spatial algorithm, but uses much more memory. $BW2 = \text{dilate}(BW1, SE, \dots, n)$ performs the dilation operation n times.
Class Support	The input image $BW1$ can be of class <code>double</code> or <code>uint8</code> . The output image $BW2$ is of class <code>uint8</code> .
Remarks	You should use the frequency algorithm only if you have a large amount of memory on your system. If you use this algorithm with insufficient memory, it may actually be <i>slower</i> than the spatial algorithm, due to virtual memory paging. If the frequency algorithm slows down your system excessively, or if you receive "out of memory" messages, use the spatial algorithm instead.

dilate

Example

```
BW1 = imread('text.tif');  
SE = ones(6, 2);  
BW2 = dilate(BW1, SE);  
imshow(BW1)  
figure, imshow(BW2)
```



See Also

bwmorph, erode

References

- Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I*. Addison-Wesley, 1992. p. 158.
- Gonzalez, Rafael C., and Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992. p. 518.

Purpose	Convert an image, increasing apparent color resolution by dithering
Syntax	$X = \text{dither}(\text{RGB}, \text{map})$ $BW = \text{dither}(I)$
Description	<p>$X = \text{dither}(\text{RGB}, \text{map})$ creates an indexed image approximation of the RGB image in the array RGB by dithering the colors in colormap map.</p> <p>$X = \text{dither}(\text{RGB}, \text{map}, Qm, Qe)$ creates an indexed image from RGB, specifying the parameters Qm and Qe. Qm specifies the number of quantization bits to use along each color axis for the inverse color map, and Qe specifies the number of quantization bits to use for the color space error calculations. If $Qe < Qm$, dithering cannot be performed and an undithered indexed image is returned in X. If you omit these parameters, dither uses the default values $Qm = 5$, $Qe = 8$.</p> <p>$BW = \text{dither}(I)$ converts the intensity image in the matrix I to the binary (black and white) image BW by dithering.</p>
Class Support	The input image (RGB or I) can be of class <code>double</code> or <code>uint8</code> . All other input arguments must be of class <code>double</code> . The output image (X or BW) is of class <code>uint8</code> if it is a binary image or if it is an indexed image with 256 or fewer colors; otherwise its class is <code>double</code> .
Algorithm	<code>dither</code> increases the apparent color resolution of an image by applying Floyd-Steinberg's error diffusion dither algorithm.
Reference	Floyd, R. W. and L. Steinberg. "An Adaptive Algorithm for Spatial Gray Scale." <i>International Symposium Digest of Technical Papers</i> . Society for Information Displays, 1975. p. 36. Lim, Jae S. <i>Two-Dimensional Signal and Image Processing</i> . Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 469-476.
See Also	<code>rgb2ind</code>

double

Purpose	Convert data to double precision
Syntax	<code>B = double(A)</code>
Description	<code>B = double(A)</code> creates a double-precision array <code>B</code> from the array <code>A</code> . If <code>A</code> is a <code>double</code> array, <code>B</code> is identical to <code>A</code> .
	<code>double</code> is useful if you have a <code>uint8</code> image array that you want to perform arithmetic operations on, because MATLAB does not support these operations on <code>uint8</code> data.
Remarks	<code>double</code> is a MATLAB built-in function.
Example	<pre>A = imread('saturn.tif'); B = sqrt(double(A));</pre>
See Also	<code>im2double</code> , <code>im2uint</code> , <code>uint8</code>

Purpose	Find edges in an intensity image
Syntax	<pre>BW = edge(I, 'sobel') BW = edge(I, 'sobel', thresh) BW = edge(I, 'sobel', thresh, direction) [BW, thresh] = edge(I, 'sobel', ...)</pre> <pre>BW = edge(I, 'prewitt') BW = edge(I, 'prewitt', thresh) BW = edge(I, 'prewitt', thresh, direction) [BW, thresh] = edge(I, 'prewitt', ...)</pre> <pre>BW = edge(I, 'roberts') BW = edge(I, 'roberts', thresh) [BW, thresh] = edge(I, 'roberts', ...)</pre> <pre>BW = edge(I, 'log') BW = edge(I, 'log', thresh) BW = edge(I, 'log', thresh, sigma) [BW, threshold] = edge(I, 'log', ...)</pre> <pre>BW = edge(I, 'zerocross', thresh, h) [BW, thresh] = edge(I, 'zerocross', ...)</pre> <pre>BW = edge(I, 'canny') BW = edge(I, 'canny', thresh) BW = edge(I, 'canny', thresh, sigma) [BW, threshold] = edge(I, 'canny', ...)</pre>
Description	edge takes an intensity image I as its input, and returns a binary image BW of the same size as I, with 1's where the function finds edges in I and 0's elsewhere.

edge supports six different edge-finding methods:

- The Sobel method finds edges using the Sobel approximation to the derivative. It returns edges at those points where the gradient of I is maximum.
- The Prewitt method finds edges using the Prewitt approximation to the derivative. It returns edges at those points where the gradient of I is maximum.
- The Roberts method finds edges using the Roberts approximation to the derivative. It returns edges at those points where the gradient of I is maximum.
- The Laplacian of Gaussian method finds edges by looking for zero crossings after filtering I with a Laplacian of Gaussian filter.
- The zero-cross method finds edges by looking for zero crossings after filtering I with a filter you specify.
- The Canny method finds edges by looking for local maxima of the gradient of I. The gradient is calculated using the derivative of a Gaussian filter. The method uses two thresholds, to detect strong and weak edges, and includes the weak edges in the output only if they are connected to strong edges. This method is therefore less likely than the others to be “fooled” by noise, and more likely to detect true weak edges.

The parameters you can supply differ depending on the method you specify. If you do not specify a method, edge uses the Sobel method.

Sobel Method

`BW = edge(I, 'sobel')` specifies the Sobel method.

`BW = edge(I, 'sobel', thresh)` specifies the sensitivity threshold for the Sobel method. edge ignores all edges that are not stronger than thresh. If you do not specify thresh, or if thresh is empty ([]), edge chooses the value automatically.

`BW = edge(I, 'sobel', thresh, direction)` specifies directionality for the Sobel method. direction is a string specifying whether to look for 'horizontal' or 'vertical' edges, or 'both' (the default).

`[BW, thresh] = edge(I, 'sobel', ...)` returns the threshold value.

Prewitt Method

`BW = edge(I, 'prewitt')` specifies the Prewitt method.

`BW = edge(I, 'prewitt', thresh)` specifies the sensitivity threshold for the Prewitt method. `edge` ignores all edges that are not stronger than `thresh`. If you do not specify `thresh`, or if `thresh` is empty ([]), `edge` chooses the value automatically.

`BW = edge(I, 'prewitt', thresh, direction)` specifies directionality for the Prewitt method. `direction` is a string specifying whether to look for 'horizontal' or 'vertical' edges, or 'both' (the default).

`[BW, thresh] = edge(I, 'prewitt', ...)` returns the threshold value.

Roberts Method

`BW = edge(I, method)` specifies the Roberts method.

`BW = edge(I, method, thresh)` specifies the sensitivity threshold for the Roberts method. `edge` ignores all edges that are not stronger than `thresh`. If you do not specify `thresh`, or if `thresh` is empty ([]), `edge` chooses the value automatically.

`[BW, thresh] = edge(I, method, ...)` returns the threshold value.

Laplacian of Gaussian Method

`BW = edge(I, 'log')` specifies the Laplacian of Gaussian method.

`BW = edge(I, 'log', thresh)` specifies the sensitivity threshold for the Laplacian of Gaussian method. `edge` ignores all edges that are not stronger than `thresh`. If you do not specify `thresh`, or if `thresh` is empty ([]), `edge` chooses the value automatically.

`BW = edge(I, 'log', thresh, sigma)` specifies the Laplacian of Gaussian method, using `sigma` as the standard deviation of the LoG filter. The default `sigma` is 2; the size of the filter is $n \times n$, where $n = \text{ceil}(\sigma * 3) * 2 + 1$.

`[BW, thresh] = edge(I, 'log', ...)` returns the threshold value.

Zero-cross Method

`BW = edge(I, 'zerocross', thresh, h)` specifies the zero-cross method, using the filter `h`. `thresh` is the sensitivity threshold; if the argument is empty ([]), `edge` chooses the sensitivity threshold automatically.

`[BW, thresh] = edge(I, 'zerocross', ...)` returns the threshold value.

Canny Method

`BW = edge(I, 'canny')` specifies the Canny method.

`BW = edge(I, 'canny', thresh)` specifies sensitivity thresholds for the Canny method. `thresh` is a two-element vector in which the first element is the low threshold, and the second element is the high threshold. If you specify a scalar for `thresh`, this value is used for the high threshold and $0.4 * \text{thresh}$ is used for the low threshold. If you do not specify `thresh`, or if `thresh` is empty ([]), `edge` chooses low and high values automatically.

`BW = edge(I, 'canny', thresh, sigma)` specifies the Canny method, using `sigma` as the standard deviation of the Gaussian filter. The default `sigma` is 1; the size of the filter is chosen automatically, based on `sigma`.

`[BW, thresh] = edge(I, 'canny', ...)` returns the threshold values as a two-element vector.

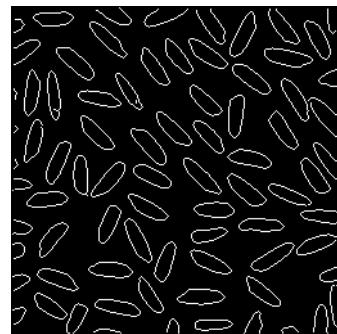
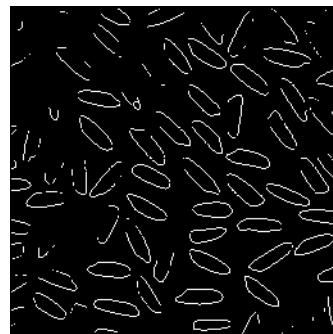
Class Support `I` can be of class `uint8` or `double`. `BW` is of class `uint8`.

Remarks For the '`log`' and '`zerocross`' methods, if you specify a threshold of 0, the output image has closed contours, because it includes all of the zero crossings in the input image.

Example

Find the edges of the rice.tif image using the Prewitt and Canny methods:

```
I = imread('rice.tif');
BW1 = edge(I, 'prewitt');
BW2 = edge(I, 'canny');
imshow(BW1)
figure, imshow(BW2)
```

**References**

Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 478-488.

Parker, James R. *Algorithms for Image Processing and Computer Vision*. New York: John Wiley & Sons, Inc., 1997. pp. 23-29.

Canny, John. "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1986. Vol. PAMI-8, No. 6, pp. 679-698.

erode

Purpose	Perform erosion on a binary image
Syntax	<pre>BW2 = erode(BW1, SE) BW2 = erode(BW1, SE, alg) BW2 = erode(BW1, SE, . . . , n)</pre>
Description	<p><code>BW2 = erode(BW1, SE)</code> performs erosion on the binary image <code>BW1</code>, using the binary structuring element <code>SE</code>. <code>SE</code> is a matrix containing only 1's and 0's.</p> <p><code>BW2 = erode(BW1, SE, alg)</code> performs erosion using the specified algorithm. <code>alg</code> is a string that can have one of these values:</p> <ul style="list-style-type: none">• '<code>'spatial'</code>' (default) – processes the image in the spatial domain.• '<code>'frequency'</code>' – processes the image in the frequency domain. <p>Both algorithms produce the same result, but they make different tradeoffs between speed and memory use. The frequency algorithm is faster for large images and structuring elements than the spatial algorithm, but uses much more memory.</p> <p><code>BW2 = erode(BW1, SE, . . . , n)</code> performs the erosion operation <code>n</code> times.</p>
Class Support	The input image <code>BW1</code> can be of class <code>double</code> or <code>uint8</code> . The output image <code>BW2</code> is of class <code>uint8</code> .
Remarks	You should use the frequency algorithm only if you have a large amount of memory on your system. If you use this algorithm with insufficient memory, it may actually be <i>slower</i> than the spatial algorithm, due to virtual memory paging. If the frequency algorithm slows down your system excessively, or if you receive "out of memory" messages, use the spatial algorithm instead.

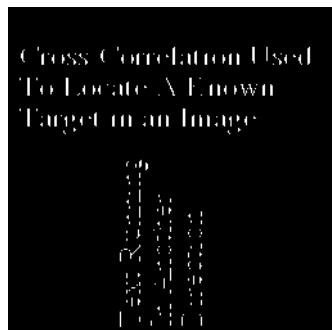
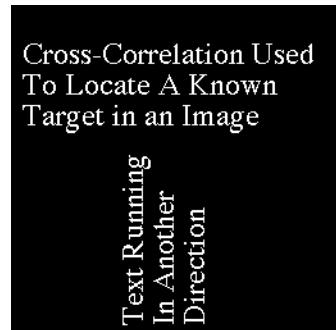
Example

```
BW1 = imread('text.tif')  
SE = ones(3, 1)
```

```
SE =
```

```
1  
1  
1
```

```
BW2 = erode(BW1, SE);  
imshow(BW1)  
figure, imshow(BW2)
```

**See Also**

bwmorph, dilate

References

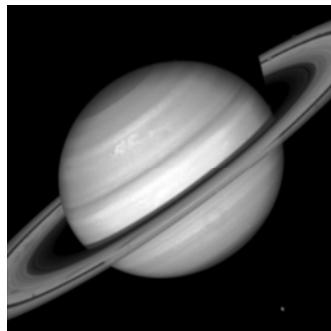
- Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I*. Addison-Wesley, 1992. p. 158.
- Gonzalez, Rafael C., and Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992. p. 518.

fft2

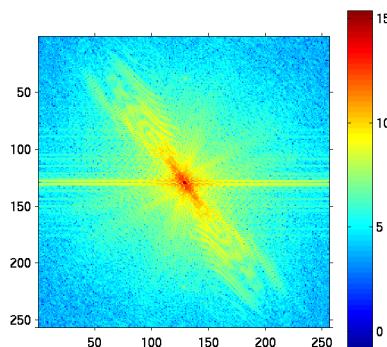
Purpose	Compute two-dimensional fast Fourier transform (FFT)
Syntax	$B = \text{fft2}(A)$ $B = \text{fft2}(A, m, n)$
Description	$B = \text{fft2}(A)$ performs a two-dimensional fast Fourier transform, returning the result in B . B is the same size as A ; if A is a vector, B has the same orientation as A . $B = \text{fft2}(A, m, n)$ truncates or zero pads A , if necessary, to create an m -by- n matrix before performing the FFT. The result B is also m -by- n .
Class Support	The input matrix A can be of class <code>uint8</code> or <code>double</code> . The output matrix B is of class <code>double</code> .
Remarks	<code>fft2</code> is a function in MATLAB.

Example

```
load imdemos saturn2  
imshow(saturn2)
```



```
B = fftshift(fft2(saturn2));  
imshow(log(abs(B)), [], 'colormap(jet(64))', 'colorbar')
```

**Algorithm**

fft2(A) is simply
$$\text{fft}(\text{fft}(A) \cdot ') \cdot '$$

This computes the one-dimensional fft of each column A, then of each row of the result. The time required to compute fft2(A) depends on the number of prime factors of m and n. fft2 is fastest when m and n are powers of 2.

See Also

dct2, fftshift, idct2, ifft2
fft, ifft in the online MATLAB Function Reference

fftn

Purpose	Compute N-dimensional fast Fourier transform
Syntax	$B = \text{fftn}(A)$ $B = \text{fftn}(A, \text{size})$
Description	$B = \text{fftn}(A)$ performs the N-dimensional fast Fourier transform. The result B is the same size as A . $B = \text{fftn}(A, \text{size})$ pads A with zeros (or truncates A) to create an N-dimensional array of size size before doing the transform. The size of the result is size .
Class Support	The input matrix A can be of class <code>uint8</code> or <code>double</code> . The output matrix B is of class <code>double</code> .
Remarks	<code>fftn</code> is a function in MATLAB.
Algorithm	$\text{fftn}(A)$ is equivalent to:
	<pre>B = A; for p = 1:length(size(A)) B = fft(B, [], p); end</pre>
	This code computes the one-dimensional fast Fourier transform along each dimension of A . The time required to compute $\text{fftn}(A)$ depends strongly on the number of prime factors of the dimensions of A . It is fastest when all of the dimensions are powers of 2.
See Also	<code>fft2</code> , <code>ifftn</code> <code>fft</code> in the online MATLAB Function Reference

Purpose	Shift DC component of fast Fourier transform to center of spectrum
Syntax	<code>B = fftshift(A)</code>
Description	<code>B = fftshift(A)</code> rearranges the outputs of <code>fft</code> , <code>fft2</code> , and <code>fftn</code> by moving the zero frequency component to the center of the array. For vectors, <code>fftshift(A)</code> swaps the left and right halves of <code>A</code> . For matrices, <code>fftshift(A)</code> swaps quadrants one and three of <code>A</code> with quadrants two and four. For higher-dimensional arrays, <code>fftshift(A)</code> swaps “half-spaces” of <code>A</code> along each dimension.
Remarks	<code>fftshift</code> is a function in MATLAB.
Example	<pre>B = fftn(A); C = fftshift(B);</pre>
See Also	<code>fft2</code> , <code>fftn</code> fft in the online MATLAB Function Reference

filter2

Purpose	Perform two-dimensional linear filtering
Syntax	$B = \text{filter2}(h, A)$ $B = \text{filter2}(h, A, \text{shape})$
Description	$B = \text{filter2}(h, A)$ filters the data in A with the two-dimensional FIR filter in the matrix h . It computes the result, B , using two-dimensional correlation, and returns the central part of the correlation that is the same size as A . $B = \text{filter2}(h, A, \text{shape})$ returns the part of B specified by the shape parameter. shape is a string with one of these values: <ul style="list-style-type: none">'full' returns the full two-dimensional correlation. In this case, B is larger than A.'same' (the default) returns the central part of the correlation. In this case, B is the same size as A.'valid' returns only those parts of the correlation that are computed without zero-padded edges. In this case, B is smaller than A.
Class Support	The matrix inputs to <code>filter2</code> can be of class <code>uint8</code> or <code>double</code> . The output matrix B is of class <code>double</code> .
Remarks	Two-dimensional correlation is equivalent to two-dimensional convolution with the filter matrix rotated 180 degrees. See the Algorithm section for more information about how <code>filter2</code> performs linear filtering. <code>filter2</code> is a function in MATLAB.

Example

```
A = magic(6)
```

```
A =
```

35	1	6	26	19	24
3	32	7	21	23	25
31	9	2	22	27	20
8	28	33	17	10	15
30	5	34	12	14	16
4	36	29	13	18	11

```
h = fspecial('sobel')
```

```
h =
```

1	2	1
0	0	0
-1	-2	-1

```
B = filter2(h, A, 'valid')
```

```
B =
```

-8	4	4	-8
-23	-44	-5	40
-23	-50	1	40
-8	4	4	-8

Algorithm

Given an image A and a two-dimensional FIR filter h, filter2 rotates your filter matrix (the computational molecule) 180 degrees to create a convolution kernel. It then calls conv2, the two-dimensional convolution function, to implement the filtering operation.

filter2 uses conv2 to compute the full two-dimensional convolution of the FIR filter with the input matrix. By default, filter2 then extracts the central part of the convolution that is the same size as the input matrix, and returns this as the result. If the shape parameter specifies an alternate part of the convolution for the result, filter2 returns the appropriate part.

See Also

conv2, roi fil t2

freqspace

Purpose	Determine frequency spacing for two-dimensional frequency response
Syntax	<pre>[f1, f2] = freqspace(n) [f1, f2] = freqspace([m n]) [x1, y1] = freqspace(..., 'meshgrid') f = freqspace(N) f = freqspace(N, 'whole')</pre>
Description	<p>freqspace returns the implied frequency range for equally spaced frequency responses. freqspace is useful when creating desired frequency responses for fsamp2, fwild1, and fwild2, as well as for various one-dimensional applications.</p> <p>[f1, f2] = freqspace(n) returns the two-dimensional frequency vectors f1 and f2 for an n-by-n matrix.</p> <p>For n odd, both f1 and f2 are [-n+1: 2: n-1]/n.</p> <p>For n even, both f1 and f2 are [-n: 2: n-2]/n.</p> <p>[f1, f2] = freqspace([m n]) returns the two-dimensional frequency vectors f1 and f2 for an m-by-n matrix.</p> <p>[x1, y1] = freqspace(..., 'meshgrid') is equivalent to</p> <pre>[f1, f2] = freqspace(...); [x1, y1] = meshgrid(f1, f2);</pre> <p>f = freqspace(N) returns the one-dimensional frequency vector f assuming N evenly spaced points around the unit circle. For N even or odd, f is (0: 2/N: 1). For N even, freqspace therefore returns (N+2)/2 points. For N odd, it returns (N+1)/2 points.</p> <p>f = freqspace(N, 'whole') returns N evenly spaced points around the whole unit circle. In this case, f is 0: 2/N: 2*(N-1)/N.</p>
Remarks	freqspace is a function in MATLAB.
See Also	fsamp2, fwild1, fwild2 meshgrid in the online MATLAB Function Reference

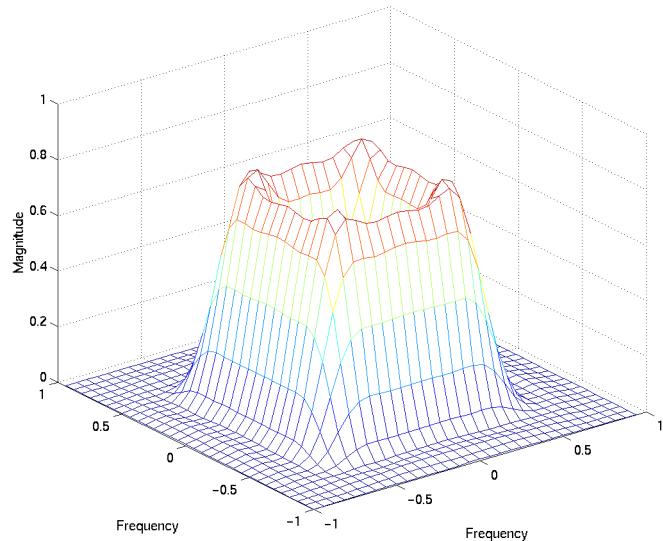
Purpose	Compute two-dimensional frequency response
Syntax	$[H, f1, f2] = \text{freqz2}(h, n1, n2)$ $[H, f1, f2] = \text{freqz2}(h, [n2 n1])$ $[H, f1, f2] = \text{freqz2}(h, f1, f2)$ $[H, f1, f2] = \text{freqz2}(h)$ $[...] = \text{freqz2}(h, [...], [dx dy])$ $[...] = \text{freqz2}(h, [...], dx)$ $\text{freqz2}(...)$
Description	<p>$[H, f1, f2] = \text{freqz2}(h, n1, n2)$ returns H, the $n2$-by-$n1$ frequency response of h, and the frequency vectors $f1$ (of length $n1$) and $f2$ (of length $n2$). h is a two-dimensional FIR filter, in the form of a computational molecule. $f1$ and $f2$ are returned as normalized frequencies in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or π radians.</p> <p>$[H, f1, f2] = \text{freqz2}(h, [n2 n1])$ returns the same result returned by $[H, f1, f2] = \text{freqz2}(h, n1, n2)$.</p> <p>$[H, f1, f2] = \text{freqz2}(h)$ uses $[n2 n1] = [64 64]$.</p> <p>$[H, f1, f2] = \text{freqz2}(h, f1, f2)$ returns the frequency response for the FIR filter h at frequency values in $f1$ and $f2$. These frequency values must be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or π radians.</p> <p>$[...] = \text{freqz2}(h, [...], [dx dy])$ uses $[dx dy]$ to override the intersample spacing in h. dx determines the spacing for the x-dimension and dy determines the spacing for the y-dimension. The default spacing is 0.5, which corresponds to a sampling frequency of 2.0.</p> <p>$[...] = \text{freqz2}(h, [...], dx)$ uses dx to determine the intersample spacing in both dimensions.</p> <p>With no output arguments, $\text{freqz2}(...)$ produces a mesh plot of the two-dimensional magnitude frequency response.</p>
Class Support	The input matrix h can be of class <code>uint8</code> or <code>double</code> . All other inputs to <code>freqz2</code> must be of class <code>double</code> . All outputs are of class <code>double</code> .

freqz2

Example

Use the window method to create a 16-by-16 filter, then view its frequency response using freqz2.

```
Hd = zeros(16, 16);  
Hd(5:12, 5:12) = 1;  
Hd(7:10, 7:10) = 0;  
h = window(Hd, bartlett(16));  
colormap(jet(64))  
freqz2(h, [32 32]); axis([-1 1 -1 1 0 1])
```



See Also

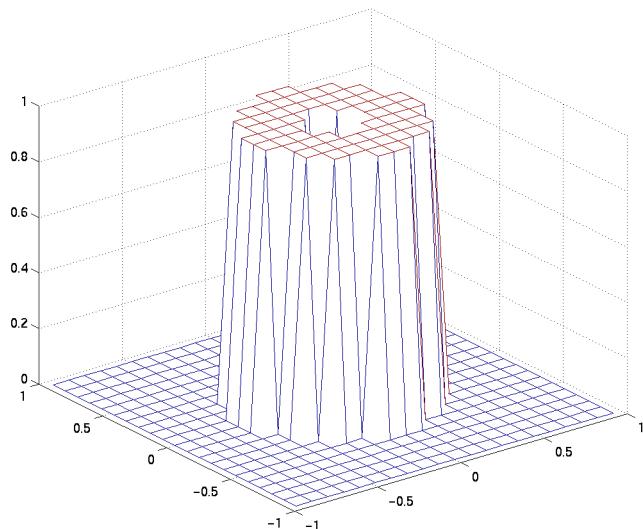
[freqz](#) in the Signal Processing Toolbox

Purpose	Design two-dimensional FIR filter using frequency sampling
Syntax	<pre>h = fsamp2(Hd) h = fsamp2(f1, f2, Hd, [m n])</pre>
Description	<p><code>fsamp2</code> designs two-dimensional FIR filters based on a desired two-dimensional frequency response sampled at points on the Cartesian plane.</p> <p><code>h = fsamp2(Hd)</code> designs a two-dimensional FIR filter with frequency response <code>Hd</code>, and returns the filter coefficients in matrix <code>h</code>. (<code>fsamp2</code> returns <code>h</code> as a computational molecule, which is the appropriate form to use with <code>filter2</code>.) The filter <code>h</code> has a frequency response that passes through points in <code>Hd</code>. If <code>Hd</code> is <code>m</code>-by-<code>n</code>, then <code>h</code> is also <code>m</code>-by-<code>n</code>.</p> <p><code>Hd</code> is a matrix containing the desired frequency response sampled at equally spaced points between -1.0 and 1.0 along the <i>x</i> and <i>y</i> frequency axes, where 1.0 corresponds to half the sampling frequency, or π radians.</p> $H_d(f_1, f_2) = H_d(\omega_1, \omega_2) \Big _{\omega_1 = \pi f_1, \omega_2 = \pi f_2}$ <p>For accurate results, use frequency points returned by <code>freqspace</code> to create <code>Hd</code>. (See the entry for <code>freqspace</code> for more information.)</p> <p><code>h = fsamp2(f1, f2, Hd, [m n])</code> produces an <code>m</code>-by-<code>n</code> FIR filter by matching the filter response at the points in the vectors <code>f1</code> and <code>f2</code>. The frequency vectors <code>f1</code> and <code>f2</code> are in normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians. The resulting filter fits the desired response as closely as possible in the least squares sense. For best results, there must be at least <code>m*n</code> desired frequency points. <code>fsamp2</code> issues a warning if you specify fewer than <code>m*n</code> points.</p>
Class Support	The input matrix <code>Hd</code> can be of class <code>uint8</code> or <code>double</code> . All other inputs to <code>fsamp2</code> must be of class <code>double</code> . All outputs are of class <code>double</code> .
Example	Use <code>fsamp2</code> to design an approximately symmetric two-dimensional bandpass filter with passband between 0.1 and 0.5 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians).

fsamp2

- 1 Create a matrix Hd that contains the desired bandpass response. Use freqspace to create the frequency range vectors f1 and f2.

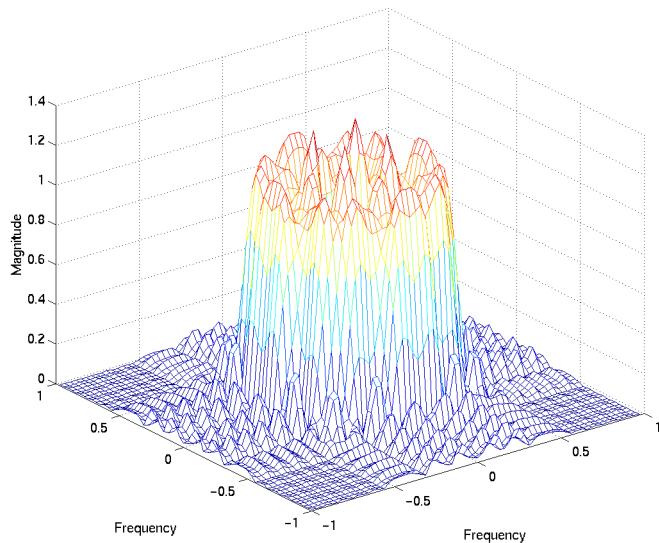
```
[f1, f2] = freqspace(21, 'meshgrid');
Hd = ones(21);
r = sqrt(f1.^2 + f2.^2);
Hd((r<0.1) | (r>0.5)) = 0;
colormap(jet(64))
mesh(f1, f2, Hd)
```



- 2 Design the filter that passes through this response.

```
h = fsamp2(Hd);
```

freqz2(h)

**Algorithm**

fsamp2 computes the filter h by taking the inverse discrete Fourier transform of the desired frequency response. If the desired frequency response is real and symmetric (zero phase), the resulting filter is also zero phase.

See Also

conv2, filter2, freqspace, ftrans2, fwild1, fwild2

Reference

Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 213-217.

fspecial

Purpose	Create predefined filters
Syntax	<pre>h = fspecial(type) h = fspecial(type, parameters)</pre>
Description	<p><code>h = fspecial(type)</code> creates a two-dimensional filter <code>h</code> of the specified type. (<code>fspecial</code> returns <code>h</code> as a computational molecule, which is the appropriate form to use with <code>filter2</code>.) <code>type</code> is a string having one of these values:</p> <ul style="list-style-type: none">• '<code>gaussian</code>' for a Gaussian lowpass filter• '<code>sobel</code>' for a Sobel horizontal edge-emphasizing filter• '<code>prewitt</code>' for a Prewitt horizontal edge-emphasizing filter• '<code>laplacian</code>' for a filter approximating the two-dimensional Laplacian operator• '<code>log</code>' for a Laplacian of Gaussian filter• '<code>average</code>' for an averaging filter• '<code>unsharp</code>' for an unsharp contrast enhancement filter <p>Depending on <code>type</code>, <code>fspecial</code> may take additional parameters which you can supply. These parameters all have default values.</p> <p><code>h = fspecial('gaussian', n, sigma)</code> returns a rotationally symmetric Gaussian lowpass filter with standard deviation <code>sigma</code> (in pixels). <code>n</code> is a 1-by-2 vector specifying the number of rows and columns in <code>h</code>. (<code>n</code> can also be a scalar, in which case <code>h</code> is <code>n</code>-by-<code>n</code>.) If you do not specify the parameters, <code>fspecial</code> uses the default values of [3 3] for <code>n</code> and 0.5 for <code>sigma</code>.</p> <p><code>h = fspecial('sobel')</code> returns this 3-by-3 horizontal edge-finding and y-derivative approximation filter:</p> <pre>[1 2 1 0 0 0 -1 -2 -1]</pre> <p>To find vertical edges, or for x-derivatives, use <code>-h'</code>.</p>

`h = fspecial('prewitt')` returns this 3-by-3 horizontal edge-finding and y -derivative approximation filter:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

To find vertical edges, or for x -derivatives, use `-h'`.

`h = fspecial('laplacian', al pha)` returns a 3-by-3 filter approximating the two-dimensional Laplacian operator. The parameter `al pha` controls the shape of the Laplacian and must be in the range 0 to 1.0. `fspecial` uses the default value of 0.2 if you do not specify `al pha`.

`h = fspecial('log', n, si gma)` returns a rotationally symmetric Laplacian of Gaussian filter with standard deviation `si gma` (in pixels). `n` is a 1-by-2 vector specifying the number of rows and columns in `h`. (`n` can also be a scalar, in which case `h` is `n`-by-`n`.) If you do not specify the parameters, `fspecial` uses the default values of [5 5] for `n` and 0.5 for `si gma`.

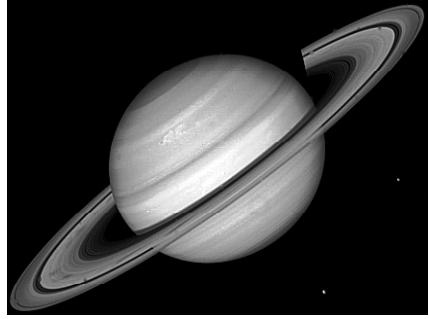
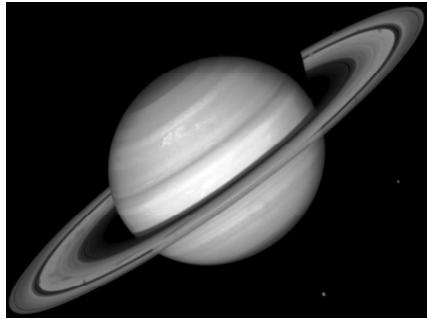
`h = fspecial('average', n)` returns an averaging filter. `n` is a 1-by-2 vector specifying the number of rows and columns in `h`. (`n` can also be a scalar, in which case `h` is `n`-by-`n`.) If you do not specify `n`, `fspecial` uses the default value of [3 3].

`h = fspecial('unsharp', al pha)` returns a 3-by-3 unsharp contrast enhancement filter. `fspecial` creates the unsharp filter from the negative of the Laplacian filter with parameter `al pha`. `al pha` controls the shape of the Laplacian and must be in the range 0 to 1.0. `fspecial` uses the default value of 0.2 if you do not specify `al pha`.

fspecial

Example

```
I = imread('saturn.tif');
h = fspecial('unsharp', 0.5);
I2 = filter2(h, I)/255;
imshow(I)
figure, imshow(I2)
```



Algorithms

`fspecial` creates Gaussian filters using:

$$h_g(n_1, n_2) = e^{-(n_1^2 + n_2^2)/(2\sigma^2)}$$

$$h(n_1, n_2) = \frac{h_g(n_1, n_2)}{\sum_{n_1} \sum_{n_2} h_g}$$

`fspecial` creates Laplacian filters using:

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

$$\nabla^2 \approx \frac{4}{(\alpha + 1)} \begin{bmatrix} \frac{\alpha}{4} & \frac{1-\alpha}{4} & \frac{\alpha}{4} \\ \frac{1-\alpha}{4} & -1 & \frac{1-\alpha}{4} \\ \frac{\alpha}{4} & \frac{1-\alpha}{4} & \frac{\alpha}{4} \end{bmatrix}$$

`fspecial` creates Laplacian of Gaussian (LoG) filters using:

$$h_g(n_1, n_2) = e^{-(n_1^2 + n_2^2)/(2\sigma^2)}$$
$$h(n_1, n_2) = \frac{(n_1^2 + n_2^2 - 2\sigma^2)h_g(n_1, n_2)}{2\pi\sigma^6 \sum_{n_1} \sum_{n_2} h_g}$$

`fspecial` creates averaging filters using:

```
ones(n(1), n(2))/(n(1)*n(2))
```

`fspecial` creates unsharp filters using:

$$\frac{1}{(\alpha+1)} \begin{bmatrix} -\alpha & \alpha-1 & -\alpha \\ \alpha-1 & \alpha+5 & \alpha-1 \\ -\alpha & \alpha-1 & -\alpha \end{bmatrix}$$

See Also

`conv2`, `edge`, `filter2`, `fsamp2`, `fwind1`, `fwind2`

[del2](#) in the online MATLAB Function Reference

ftrans2

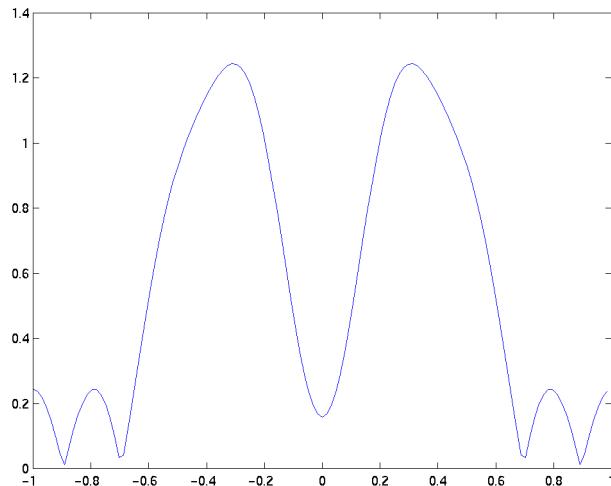
Purpose	Design two-dimensional FIR filter using frequency transformation
Syntax	$h = \text{ftrans2}(b, t)$ $h = \text{ftrans2}(b)$
Description	$h = \text{ftrans2}(b, t)$ produces the two-dimensional FIR filter h that corresponds to the one-dimensional FIR filter b using the transform t . (ftrans2 returns h as a computational molecule, which is the appropriate form to use with filter2 .) b must be a one-dimensional, odd-length (Type I) FIR filter such as can be returned by fir1 , fir2 , or remez in the Signal Processing Toolbox. The transform matrix t contains coefficients that define the frequency transformation to use. If t is m -by- n and b has length Q , then h is size $((m-1)*(Q-1)/2+1) \times ((n-1)*(Q-1)/2+1)$. $h = \text{ftrans2}(b)$ uses the McClellan transform matrix t : $t = [1 \ 2 \ 1; \ 2 \ -4 \ 2; \ 1 \ 2 \ 1]/8;$
Remarks	The transformation below defines the frequency response of the two-dimensional filter returned by ftrans2 : $H(\omega_1, \omega_2) = B(\omega) _{\cos \omega = T(\omega_1, \omega_2)}$ where $B(\omega)$ is the Fourier transform of the one-dimensional filter b : $B(\omega) = \sum_{n=-N}^N b(n) e^{-j\omega n}$ and $T(\omega_1, \omega_2)$ is the Fourier transform of the transformation matrix t : $T(\omega_1, \omega_2) = \sum_{n_2} \sum_{n_1} t(n_1, n_2) e^{-j\omega_1 n_1} e^{-j\omega_2 n_2}$ The returned filter h is the inverse Fourier transform of $H(\omega_1, \omega_2)$: $h(n_1, n_2) = \frac{1}{(2\pi)^2} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} H(\omega_1, \omega_2) e^{j\omega_1 n_1} e^{j\omega_2 n_2} d\omega_1 d\omega_2$

Example

Use ftrans2 to design an approximately circularly symmetric two-dimensional bandpass filter with passband between 0.1 and 0.6 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians).

- 1 Since ftrans2 transforms a one-dimensional FIR filter to create a two-dimensional filter, first design a one-dimensional FIR bandpass filter using the Signal Processing Toolbox function remez:

```
colormap(jet(64))
b = remez(10, [0 0.05 0.15 0.55 0.65 1], [0 0 1 1 0 0]);
[H, w] = freqz(b, 1, 128, 'whole');
plot(w/pi-1, fftshift(abs(H)))
```

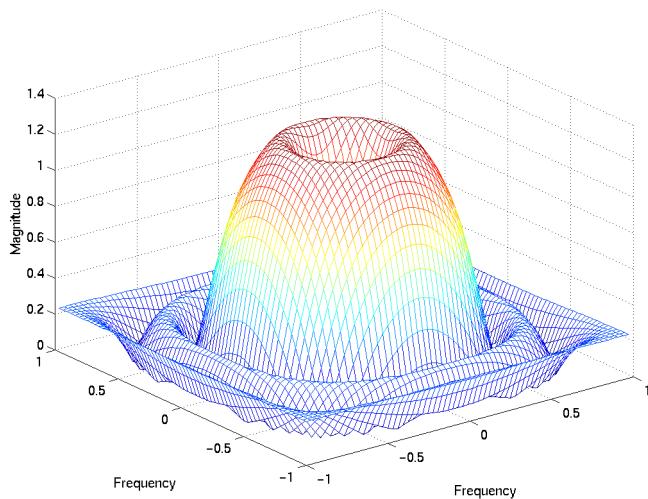


- 2 Use ftrans2 with the default McClellan transformation to create the desired approximately circularly symmetric filter

```
h = ftrans2(b);
```

ftrans2

freqz2(h)



See Also

conv2, filter2, fsamp2, fwind1, fwind2

Reference

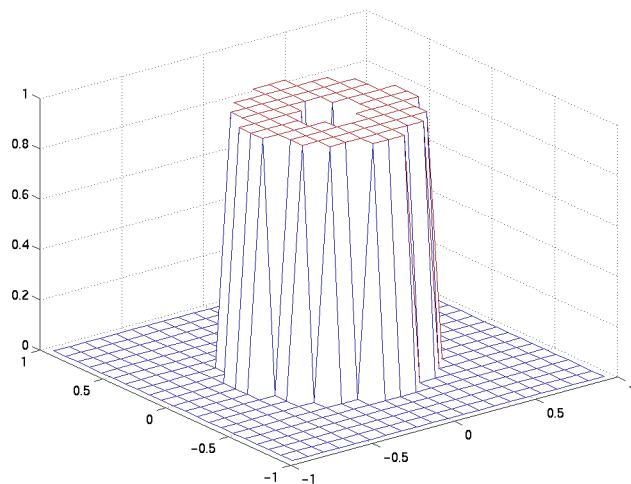
Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 218-237.

Purpose	Design two-dimensional FIR filter using one-dimensional window method
Syntax	$h = \text{fwind1}(Hd, wIn)$ $h = \text{fwind1}(Hd, wIn1, wIn2)$ $h = \text{fwind1}(f1, f2, Hd, \dots)$
Description	<p><code>fwind1</code> designs two-dimensional FIR filters using the window method. <code>fwind1</code> uses a one-dimensional window specification to design a two-dimensional FIR filter based on the desired frequency response <code>Hd</code>. <code>fwind1</code> works with one-dimensional windows only; use <code>fwind2</code> to work with two-dimensional windows.</p> <p><code>h = fwind1(Hd, wIn)</code> designs a two-dimensional FIR filter <code>h</code> with frequency response <code>Hd</code>. (<code>fwind1</code> returns <code>h</code> as a computational molecule, which is the appropriate form to use with <code>filter2</code>.) <code>fwind1</code> uses the one-dimensional window <code>wIn</code> to form an approximately circularly symmetric two-dimensional window using Huang's method. You can specify <code>wIn</code> using windows from the Signal Processing Toolbox, such as <code>boxcar</code>, <code>hamming</code>, <code>hann</code>, <code>bartlett</code>, <code>blackman</code>, <code>kaiser</code>, or <code>chebwin</code>. If <code>length(wIn)</code> is <code>n</code>, then <code>h</code> is <code>n</code>-by-<code>n</code>.</p> <p><code>Hd</code> is a matrix containing the desired frequency response sampled at equally spaced points between <code>-1.0</code> and <code>1.0</code> (in normalized frequency, where <code>1.0</code> corresponds to half the sampling frequency, or π radians) along the <code>x</code> and <code>y</code> frequency axes. For accurate results, use frequency points returned by <code>freqspace</code> to create <code>Hd</code>. (See the entry for <code>freqspace</code> for more information.)</p> <p><code>h = fwind1(Hd, wIn1, wIn2)</code> uses the two one-dimensional windows <code>wIn1</code> and <code>wIn2</code> to create a separable two-dimensional window. If <code>length(wIn1)</code> is <code>n</code> and <code>length(wIn2)</code> is <code>m</code>, then <code>h</code> is <code>m</code>-by-<code>n</code>.</p> <p><code>h = fwind1(f1, f2, Hd, ...)</code> lets you specify the desired frequency response <code>Hd</code> at arbitrary frequencies (<code>f1</code> and <code>f2</code>) along the <code>x</code> and <code>y</code> axes. The frequency vectors <code>f1</code> and <code>f2</code> should be in the range <code>-1.0</code> to <code>1.0</code>, where <code>1.0</code> corresponds to half the sampling frequency, or π radians. The length of the window(s) controls the size of the resulting filter, as above.</p>
Example	Use <code>fwind1</code> to design an approximately circularly symmetric two-dimensional bandpass filter with passband between <code>0.1</code> and <code>0.5</code> (normalized frequency, where <code>1.0</code> corresponds to half the sampling frequency, or π radians).

fwind1

- 1 Create a matrix Hd that contains the desired bandpass response. Use freqspace to create the frequency range vectors f1 and f2.

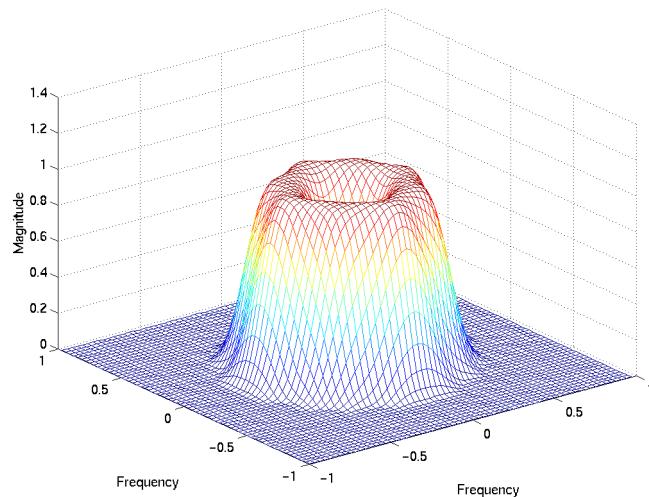
```
[f1, f2] = freqspace(21, 'meshgrid');
Hd = ones(21);
r = sqrt(f1.^2 + f2.^2);
Hd((r<0.1) | (r>0.5)) = 0;
colormap(jet(64))
mesh(f1, f2, Hd)
```



- 2 Design the filter using a one-dimensional Hamming window.

```
h = fwind1(Hd, hamming(21));
```

freqz2(h)



Algorithm

fwind1 takes a one-dimensional window specification and forms an approximately circularly symmetric two-dimensional window using Huang's method

$$w(n_1, n_2) = w(t) \Big|_{t = \sqrt{n_1^2 + n_2^2}}$$

where $w(t)$ is the one-dimensional window and $w(n_1, n_2)$ is the resulting two-dimensional window.

Given two windows, fwind1 forms a separable two-dimensional window,

$$w(n_1, n_2) = w_1(n_1)w_2(n_2)$$

fwind1 calls fwind2 with Hd and the two-dimensional window. fwind2 computes h using an inverse Fourier transform and multiplication by the two-dimensional window:

$$h_d(n_1, n_2) = \frac{1}{(2\pi)^2} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} H_d(\omega_1, \omega_2) e^{j\omega_1 n_1} e^{j\omega_2 n_2} d\omega_1 d\omega_2$$

$$h(n_1, n_2) = h_d(n_1, n_2)w(n_1, n_2)$$

fwind1

See Also conv2, filter2, fsamp2, freqspace, ftrans2, fwind2

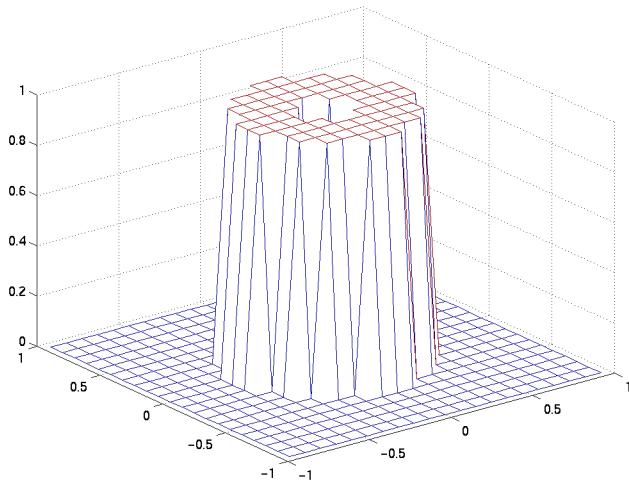
Reference Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990.

Purpose	Design two-dimensional FIR filter using two-dimensional window method
Syntax	$h = \text{fwind2}(Hd, win)$ $h = \text{fwind2}(f1, f2, Hd, win)$
Description	Use <code>fwind2</code> to design two-dimensional FIR filters using the window method. <code>fwind2</code> uses a two-dimensional window specification to design a two-dimensional FIR filter based on the desired frequency response <code>Hd</code> . <code>fwind2</code> works with two-dimensional windows; use <code>fwind1</code> to work with one-dimensional windows. <code>h = fwind2(Hd, win)</code> produces the two-dimensional FIR filter <code>h</code> using an inverse Fourier transform of the desired frequency response <code>Hd</code> and multiplication by the window <code>win</code> . <code>Hd</code> is a matrix containing the desired frequency response at equally spaced points in the Cartesian plane. <code>fwind2</code> returns <code>h</code> as a computational molecule, which is the appropriate form to use with <code>filter2</code> . <code>h</code> is the same size as <code>win</code> . For accurate results, use frequency points returned by <code>freqspace</code> to create <code>Hd</code> . (See the entry for <code>freqspace</code> for more information.) <code>h = fwind2(f1, f2, Hd, win)</code> lets you specify the desired frequency response <code>Hd</code> at arbitrary frequencies (<code>f1</code> and <code>f2</code>) along the <code>x</code> and <code>y</code> axes. The frequency vectors <code>f1</code> and <code>f2</code> should be in the range -1.0 to 1.0 , where 1.0 corresponds to half the sampling frequency, or π radians. <code>h</code> is the same size as <code>win</code> .
Example	Use <code>fwind2</code> to design an approximately circularly symmetric two-dimensional bandpass filter with passband between 0.1 and 0.5 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians).

fwind2

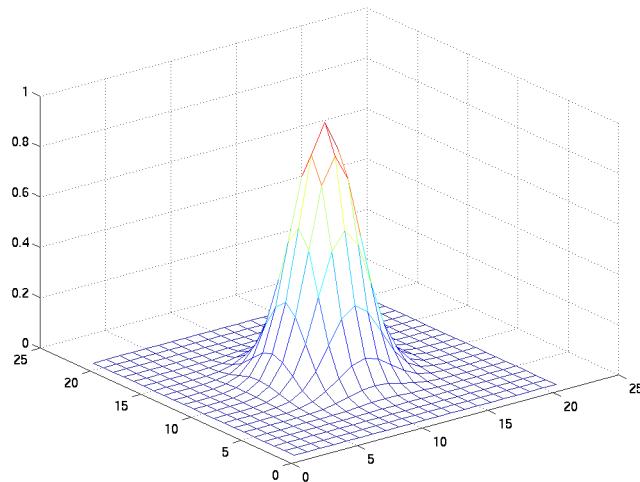
- 1 Create a matrix Hd that contains the desired bandpass response. Use freqspace to create the frequency range vectors f1 and f2.

```
[f1, f2] = freqspace(21, 'meshgrid');
Hd = ones(21);
r = sqrt(f1.^2 + f2.^2);
Hd((r<0.1) | (r>0.5)) = 0;
colormap(jet(64))
mesh(f1, f2, Hd)
```



2 Create a two-dimensional Gaussian window using fspecial.

```
win = fspecial('gaussian', 21, 2);  
win = win ./ max(win(:)); % Make the maximum window value be 1.  
mesh(win)
```

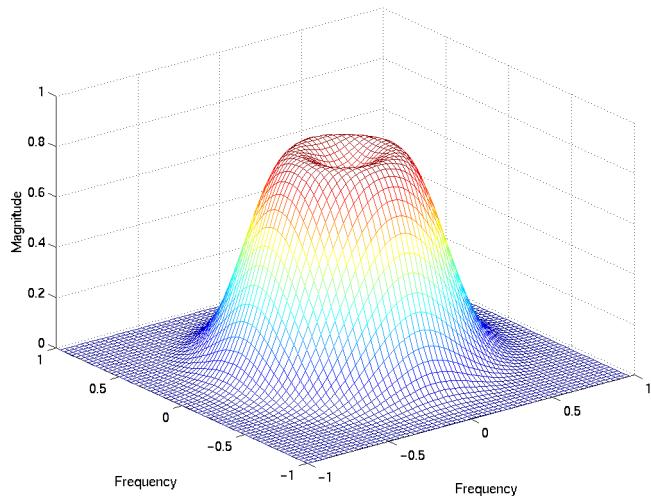


3 Design the filter using the window from step 2.

```
h = fwind2(Hd, win);
```

fwind2

freqz2(h)



Algorithm

fwind2 computes h using an inverse Fourier transform and multiplication by the two-dimensional window win:

$$h_d(n_1, n_2) = \frac{1}{(2\pi)^2} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} H_d(\omega_1, \omega_2) e^{j\omega_1 n_1} e^{j\omega_2 n_2} d\omega_1 d\omega_2$$
$$h(n_1, n_2) = h_d(n_1, n_2) w(n_1, n_2)$$

See Also

conv2, filter2, fsamp2, freqspace, ftrans2, fwind1

Reference

Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 202-213.

Purpose Get image data from axes

Syntax

```
A = getimage(h)
[x, y, A] = getimage(h)
[..., A, flag] = getimage(h)
[...] = getimage
```

Description

`A = getimage(h)` returns the first image data contained in the Handle Graphics object `h`. `h` can be a figure, axes, image, or texture-mapped surface. `A` is identical to the image CData; it contains the same values and is of the same class (`uint8` or `double`) as the image CData. If `h` is not an image or does not contain an image or texture-mapped surface, `A` is empty.

`[x, y, A] = getimage(h)` returns the image XData in `x` and the YData in `y`. XData and YData are two-element vectors that indicate the range of the *x*-axis and *y*-axis.

`[..., A, flag] = getimage(h)` returns an integer flag that indicates the type of image `h` contains. This table summarizes the possible values for `flag`:

Flag	Type of image
0	Not an image; <code>A</code> is returned as an empty matrix
1	Intensity image with values in standard range ([0,1] for <code>double</code> arrays, [0,255] for <code>uint8</code> arrays)
2	Indexed image
3	Intensity data, but not in standard range
4	RGB image

`[...] = getimage` returns information for the current axes. It is equivalent to
`[...] = getimage(gca)`.

Class Support

The output array `A` is of the same class as the image CData. All other inputs and outputs are of class `double`.

getimage

Example

This example illustrates obtaining the image data from an image displayed directly from a file.

```
imshow rice.tif  
I = getimage;
```

Purpose	Convert an intensity image to an indexed image
Syntax	$[X, \text{map}] = \text{gray2ind}(I, n)$
Description	<code>gray2ind</code> scales, then rounds, an intensity image to produce an equivalent indexed image.
	$[X, \text{map}] = \text{gray2ind}(I, n)$ converts the intensity image I to an indexed image X with colormap $\text{gray}(n)$. If n is omitted, it defaults to 64.
Class Support	The input image I can be of class <code>uint8</code> or <code>double</code> . The class of the output image X is <code>uint8</code> if the colormap length is less than or equal to 256. If the colormap length is greater than 256, X is of class <code>double</code> .
See Also	<code>ind2gray</code>

grayslice

Purpose Create indexed image from intensity image, using multilevel thresholding

Syntax

```
X = grayslice(I, n)
X = grayslice(I, v)
```

Description

$X = \text{grayslice}(I, n)$ thresholds the intensity image I using cutoff values $\frac{1}{n}, \frac{2}{n}, \dots, \frac{n-1}{n}$, returning an indexed image in X .

$X = \text{grayslice}(I, v)$, where v is a vector of values between 0 and 1, thresholds I using the values of v , returning an indexed image in X .

You can view the thresholded image using `imshow(X, map)` with a colormap of appropriate length.

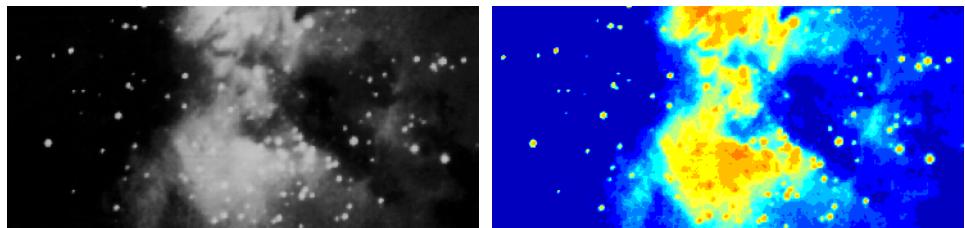
Class Support

The input image I can be of class `uint8` or `double`. Note that the threshold values are always between 0 and 1, even if I is of class `uint8`. In this case, each threshold value is multiplied by 255 to determine the actual threshold to use.

The class of the output image X depends on the number of threshold values, as specified by n or `length(v)`. If the number of threshold values is less than 256, then X is of class `uint8`, and the values in X range from 0 to n or `length(v)`. If the number of threshold values is 256 or greater, X is of class `double`, and the values in X range from 1 to $n+1$ or `length(v)+1`.

Example

```
I = imread('ngc4024m.tif');
X = grayslice(I, 16);
imshow(I)
figure, imshow(X,jet(16))
```



See Also `gray2ind`

Purpose	Enhance contrast using histogram equalization
Syntax	$J = \text{histeq}(I, \text{hgram})$ $J = \text{histeq}(I, n)$ $[J, T] = \text{histeq}(I, \dots)$ $\text{newmap} = \text{histeq}(X, \text{map}, \text{hgram})$ $\text{newmap} = \text{histeq}(X, \text{map})$ $[\text{newmap}, T] = \text{histeq}(X, \dots)$
Description	<p><code>histeq</code> enhances the contrast of images by transforming the values in an intensity image, or the values in the colormap of an indexed image, so that the histogram of the output image approximately matches a specified histogram.</p> <p><code>J = histeq(I, hgram)</code> transforms the intensity image <code>I</code> so that the histogram of the output intensity image <code>J</code> with <code>length(hgram)</code> bins approximately matches <code>hgram</code>. The vector <code>hgram</code> should contain integer counts for equally spaced bins with intensity values from 0 to 1.0. <code>histeq</code> automatically scales <code>hgram</code> so that <code>sum(hgram) = prod(size(I))</code>. The histogram of <code>J</code> will better match <code>hgram</code> when <code>length(hgram)</code> is much smaller than the number of discrete levels in <code>I</code>.</p> <p><code>J = histeq(I, n)</code> transforms the intensity image <code>I</code>, returning in <code>J</code> an intensity image with <code>n</code> discrete gray levels. A roughly equal number of pixels is mapped to each of the <code>n</code> levels in <code>J</code>, so that the histogram of <code>J</code> is approximately flat. (The histogram of <code>J</code> is flatter when <code>n</code> is much smaller than the number of discrete levels in <code>I</code>.) The default value for <code>n</code> is 64.</p> <p><code>[J, T] = histeq(I, ...)</code> returns the gray scale transformation that maps gray levels in the intensity image <code>I</code> to gray levels in <code>J</code>.</p> <p><code>newmap = histeq(X, map, hgram)</code> transforms the colormap associated with the indexed image <code>X</code> so that the histogram of the gray component of the indexed image (<code>X, newmap</code>) approximately matches <code>hgram</code>. <code>histeq</code> returns the transformed colormap in <code>newmap</code>. <code>length(hgram)</code> must be the same as <code>size(map, 1)</code>.</p> <p><code>newmap = histeq(X, map)</code> transforms the values in the colormap so that the histogram of the gray component of the indexed image <code>X</code> is approximately flat. It returns the transformed colormap in <code>newmap</code>.</p>

histeq

[newmap, T] = histeq(X, ...) returns the grayscale transformation T that maps the gray component of map to the gray component of newmap.

Class Support

For syntaxes that include an intensity image I as input, I can be of class uint8 or double, and the output image J has the same class as I. For syntaxes that include an indexed image X as input, X can be of class uint8 or double; the output colormap is always of class double. Also, the optional output T (the gray level transform) is always of class double.

Examples

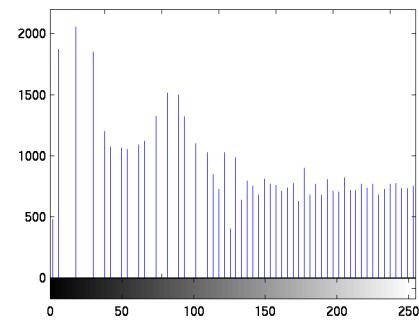
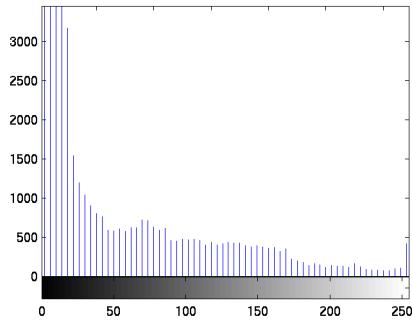
Enhance the contrast of an intensity image using histogram equalization.

```
I = imread('tire.tif');
J = histeq(I);
imshow(I)
figure, imshow(J)
```



Display the resulting histograms.

```
imhist(I, 64)
figure; imhist(J, 64)
```



Algorithm

When you supply a desired histogram `hgram`, `histeq` chooses the grayscale transformation T to minimize

$$|c_1(T(k)) - c_0(k)|$$

where c_0 is the cumulative histogram of A , c_1 is the cumulative sum of `hgram` for all intensities k . This minimization is subject to the constraints that T must be monotonic and $c_1(T(a))$ cannot overshoot $c_0(a)$ by more than half the distance between the histogram counts at a . `histeq` uses this transformation to map the gray levels in X (or the colormap) to their new values

$$b = T(a)$$

If you do not specify `hgram`, `histeq` creates a flat `hgram`

```
hgram = ones(1, n) * prod(size(A)) / n;
```

and then applies the previous algorithm.

See Also

`brighten`, `imadjust`, `imhist`

hsv2rgb

Purpose	Convert hue-saturation-value (HSV) values to RGB color space
Syntax	<pre>rgbmap = hsv2rgb(hsvmap) RGB = hsv2rgb(HSV)</pre>
Description	<p><code>rgbmap = hsv2rgb(hsvmap)</code> converts the HSV values in <code>hsvmap</code> to RGB color space. <code>hsvmap</code> is an m-by-3 matrix that contains hue, saturation, and value components as its three columns, and <code>rgbmap</code> is returned as an m-by-3 matrix that represents the same set of colors as red, green, and blue values. Both <code>rgbmap</code> and <code>hsvmap</code> contain values in the range 0 to 1.0.</p> <p><code>RGB = hsv2rgb(HSV)</code> converts the HSV image to the equivalent RGB image. HSV is an m-by-n-by-3 image array whose three planes contain the hue, saturation, and value components for the image. RGB is returned as an m-by-n-by-3 image array whose three planes contain the red, green, and blue components for the image.</p>
Class Support	The input array to <code>hsv2rgb</code> must be of class <code>double</code> . The output array is of class <code>double</code> .
Remarks	<code>hsv2rgb</code> is a function in MATLAB.
See Also	<code>rgb2hsv</code> , <code>rgbplot</code> <code>colormap</code> in the online MATLAB Function Reference

Purpose	Compute two-dimensional inverse discrete cosine transform
Syntax	$B = \text{idct2}(A)$ $B = \text{idct2}(A, m, n)$ $B = \text{idct2}(A, [m n])$
Description	<p>$B = \text{idct2}(A)$ returns the two-dimensional inverse discrete cosine transform of A.</p> <p>$B = \text{idct2}(A, m, n)$ or $B = \text{idct2}(A, [m n])$ pads A with zeros to size m-by-n before transforming. If $[m n] < \text{size}(A)$, idct2 crops A before transforming.</p> <p>For any A, $\text{idct2}(\text{dct2}(A))$ equals A to within roundoff error.</p>
Class Support	The input matrix A can be of class <code>uint8</code> or <code>double</code> . The output matrix B is of class <code>double</code> .
Algorithm	<code>i dct2</code> computes the two-dimensional inverse DCT using
	$A_{mn} = \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} \alpha_p \alpha_q B_{pq} \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N}, \quad 0 \leq m \leq M-1$ $0 \leq n \leq N-1$ $\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \leq p \leq M-1 \end{cases} \quad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \leq q \leq N-1 \end{cases}$
See Also	<code>dct2</code> , <code>dctmtx</code> , <code>fft2</code> , <code>ifft2</code>
References	<p>Jain, Anil K. <i>Fundamentals of Digital Image Processing</i>. Englewood Cliffs, NJ: Prentice Hall, 1989. pp. 150-153.</p> <p>Pennebaker, William B., and Joan L. Mitchell. <i>JPEG: Still Image Data Compression Standard</i>. New York: Van Nostrand Reinhold, 1993.</p>

ifft2

Purpose	Compute two-dimensional inverse fast Fourier transform
Syntax	$B = \text{ifft2}(A)$ $B = \text{ifft2}(A, m, n)$
Description	$B = \text{ifft2}(A)$ returns the two-dimensional inverse fast Fourier transform of matrix A. If A is a vector, B has the same orientation as A. $B = \text{ifft2}(A, m, n)$ pads matrix A with zeros to size m-by-n. If $[m n] < \text{size}(A)$, i fft2 crops A before transforming. For any A, $\text{ifft2}(\text{fft2}(A))$ equals A to within roundoff error. If A is real, $\text{ifft2}(\text{fft2}(A))$ may have small imaginary parts.
Class Support	The input matrix A can be of class <code>uint8</code> or <code>double</code> . The output matrix B is of class <code>double</code> .
Remarks	<code>ifft2</code> is a function in MATLAB.
Algorithm	The algorithm for <code>ifft2(A)</code> is the same as the algorithm for <code>fft2(A)</code> , except for a sign change and scale factors of $[m n] = \text{size}(A)$. Like <code>fft2</code> , the execution time is fastest when m and n are powers of 2 and slowest when they are large prime numbers.
See Also	<code>fft2</code> , <code>fftshift</code> , <code>idct2</code> <code>dftmtx</code> , <code>filter</code> , <code>freqz</code> , <code>specplot</code> , <code>spectrum</code> in the <i>Signal Processing Toolbox User's Guide</i> <code>fft</code> , <code>ifft</code> in the online MATLAB Function Reference

Purpose	Compute N-dimensional inverse fast Fourier transform
Syntax	$B = \text{ifftn}(A)$ $B = \text{ifftn}(A, \text{size})$
Description	$B = \text{ifftn}(A)$ performs the N-dimensional inverse fast Fourier transform. The result B is the same size as A . $B = \text{ifftn}(A, \text{size})$ pads A with zeros (or truncates A) to create an N-dimensional array of size size before doing the inverse transform. The size of the result is size . For any A , $\text{ifftn}(\text{fft}(A))$ equals A within roundoff error. If A is real, $\text{ifftn}(\text{fft}(A))$ may have small imaginary parts.
Class Support	The input matrix A can be of class <code>uint8</code> or <code>double</code> . The output matrix B is of class <code>double</code> .
Remarks	<code>ifftn</code> is a function in MATLAB.
Algorithm	$\text{ifftn}(A)$ is equivalent to <pre>B = A; for p = 1:length(size(A)) B = ifft(B, [], p); end</pre> This code computes the one-dimensional inverse fast Fourier transform along each dimension of A . The time required to compute $\text{ifftn}(A)$ depends strongly on the number of prime factors of the dimensions of A . It is fastest when all of the dimensions are powers of 2.
See Also	<code>fft2</code> , <code>fftn</code> , <code>ifft2</code>

im2bw

Purpose	Convert an image to a binary image, based on threshold
Syntax	$BW = \text{im2bw}(I, \text{level})$ $BW = \text{im2bw}(X, \text{map}, \text{level})$ $BW = \text{im2bw}(\text{RGB}, \text{level})$
Description	<code>im2bw</code> produces binary images from indexed, intensity, or RGB images. To do this, it converts the input image to grayscale format (if it is not already an intensity image), and then converts this grayscale image to binary by thresholding. The output binary image <code>BW</code> has values of 0 (black) for all pixels in the input image with luminance less than <code>level</code> and 1 (white) for all other pixels. (Note that you specify <code>level</code> in the range [0,1], regardless of the class of the input image.)
	<code>BW = im2bw(I, level)</code> converts the intensity image <code>I</code> to black and white.
	<code>BW = im2bw(X, map, level)</code> converts the indexed image <code>X</code> with colormap <code>map</code> to black and white.
	<code>BW = im2bw(RGB, level)</code> converts the RGB image <code>RGB</code> to black and white.
Class Support	The input image can be of class <code>uint8</code> or <code>double</code> . The output image <code>BW</code> is of class <code>uint8</code> .
Example	<pre>load trees BW = im2bw(X, map, 0.4); imshow(X, map) figure, imshow(BW)</pre>
	 
See Also	<code>ind2gray</code> , <code>rgb2gray</code>

Purpose	Rearrange image blocks into columns
Syntax	<pre>B = im2col(A, [m n], block_type) B = im2col(A, [m n]) B = im2col(A, 'indexed', ...)</pre>
Description	<p><code>im2col</code> rearranges image blocks into columns. <code>block_type</code> is a string that can have one of these values:</p> <ul style="list-style-type: none"> • 'distinct' for m-by-n distinct blocks • 'sliding' for m-by-n sliding blocks (default) <p><code>B = im2col(A, [m n], 'distinct')</code> rearranges each distinct m-by-n block in the image <code>A</code> into a column of <code>B</code>. <code>im2col</code> pads <code>A</code> with zeros, if necessary, so its size is an integer multiple of m-by-n. If $A = [A_{11} \ A_{12}; A_{21} \ A_{22}]$, where each A_{ij} is m-by-n, then $B = [A_{11}(:) \ A_{12}(:) \ A_{21}(:) \ A_{22}(:)]$.</p> <p><code>B = im2col(A, [m n], 'sliding')</code> converts each sliding m-by-n block of <code>A</code> into a column of <code>B</code>, with no zero padding. <code>B</code> has m^*n rows and will contain as many columns as there are m-by-n neighborhoods of <code>A</code>. If the size of <code>A</code> is $[mm \ nn]$, then the size of <code>B</code> is (m^*n)-by-$((mm-m+1)*(nn-n+1))$.</p> <p><code>B = im2col(A, [m n])</code> uses the default <code>block_type</code> of 'sliding'.</p> <p>For the sliding block case, each column of <code>B</code> contains the neighborhoods of <code>A</code> reshaped as <code>nhood(:)</code> where <code>nhood</code> is a matrix containing an m-by-n neighborhood of <code>A</code>. <code>im2col</code> orders the columns of <code>B</code> so that they can be reshaped to form a matrix in the normal way. For example, suppose you use a function, such as <code>sum(B)</code>, that returns a scalar for each column of <code>B</code>. You can directly store the result in a matrix of size $(mm-m+1)$-by-$(nn-n+1)$, using these calls:</p> <pre>B = im2col(A, [m n], 'sliding'); C = reshape(sum(B), mm-m+1, nn-n+1);</pre> <p><code>B = im2col(A, 'indexed', ...)</code> processes <code>A</code> as an indexed image, padding with zeros if the class of <code>A</code> is <code>uint8</code>, or ones if the class of <code>A</code> is <code>double</code>.</p>
Class Support	The input image <code>A</code> can be of class <code>uint8</code> or <code>double</code> . The output matrix <code>B</code> is of the same class as the input image.
See Also	<code>blkproc</code> , <code>col2im</code> , <code>colfilt</code> , <code>nlfilter</code>

im2double

Purpose	Convert image array to double precision
Syntax	<code>I2 = im2double(I1)</code> <code>RGB2 = im2double(RGB1)</code> <code>BW2 = im2double(BW1)</code> <code>X2 = im2double(X1, 'indexed')</code>
Description	<code>im2double</code> takes an image as input, and returns an image of class <code>double</code> . If the input image is of class <code>double</code> , the output image is identical to it. If the input image is of class <code>uint8</code> , <code>im2double</code> returns the equivalent image of class <code>double</code> , rescaling or offsetting the data as necessary. <code>I2 = im2double(I1)</code> converts the intensity image <code>I1</code> to double precision, rescaling the data if necessary. <code>RGB2 = im2double(RGB1)</code> converts the truecolor image <code>RGB1</code> to double precision, rescaling the data if necessary. <code>BW2 = im2double(BW1)</code> converts the binary image <code>BW1</code> to double precision. <code>X2 = im2double(X1, 'indexed')</code> converts the indexed image <code>X1</code> to double precision, offsetting the data if necessary.
See Also	<code>double</code> , <code>im2uint8</code> , <code>uint8</code>

Purpose	Convert image array to eight-bit unsigned integers
Syntax	<code>I2 = im2uint8(I1)</code> <code>RGB2 = im2uint8(RGB1)</code> <code>BW2 = im2uint8(BW1)</code> <code>X2 = im2uint8(X1, 'indexed')</code>
Description	<code>im2uint8</code> takes an image as input, and returns an image of class <code>uint8</code> . If the input image is of class <code>uint8</code> , the output image is identical to it. If the input image is of class <code>double</code> , <code>im2uint8</code> returns the equivalent image of class <code>uint8</code> , rescaling or offsetting the data as necessary. <code>I2 = im2uint8(I1)</code> converts the intensity image <code>I1</code> to <code>uint8</code> , rescaling the data if necessary. <code>RGB2 = im2uint8(RGB1)</code> converts the truecolor image <code>RGB1</code> to <code>uint8</code> , rescaling the data if necessary. <code>BW2 = im2uint8(BW1)</code> converts the binary image <code>BW1</code> to <code>uint8</code> . <code>X2 = im2uint8(X1, 'indexed')</code> converts the indexed image <code>X1</code> to <code>uint8</code> , offsetting the data if necessary. Note that <code>max(X1(:))</code> must be 256 or less, or else <code>X1</code> cannot be converted to <code>uint8</code> .
See Also	<code>double</code> , <code>im2double</code> , <code>uint8</code>

imadjust

Purpose	Adjust image intensity values or colormap
Syntax	<pre>J = imadjust(I, [low high], [bottom top], gamma) newmap = imadjust(map, [low high], [bottom top], gamma)</pre>
Description	<p><code>J = imadjust(I, [low high], [bottom top], gamma)</code> transforms the values in the intensity image <code>I</code> to values in <code>J</code> by mapping values between <code>low</code> and <code>high</code> to values between <code>bottom</code> and <code>top</code>. Values below <code>low</code> and above <code>high</code> are clipped; that is, values below <code>low</code> map to <code>bottom</code>, and those above <code>high</code> map to <code>top</code>. You can use an empty matrix (<code>[]</code>) for <code>[low high]</code> or for <code>[bottom top]</code> to specify the default of <code>[0 1]</code>. <code>gamma</code> specifies the shape of the curve describing the relationship between the values in <code>I</code> and <code>J</code>. If <code>gamma</code> is less than 1, the mapping is weighted toward higher (brighter) output values. If <code>gamma</code> is greater than than 1, the mapping is weighted toward lower (darker) output values. If you omit the argument, <code>gamma</code> defaults to 1 (linear mapping).</p> <p><code>newmap = imadjust(map, [low high], [bottom top], gamma)</code> transforms the colormap associated with an indexed image. If <code>[low high]</code> and <code>[bottom top]</code> are both 2-by-3, and <code>gamma</code> is a 1-by-3 vector, <code>imadjust</code> rescales the red, green, and blue components separately. The rescaled colormap, <code>newmap</code>, is the same size as <code>map</code>.</p> <p><code>RGB2 = imadjust(RGB1, ...)</code> performs the adjustment on each image plane (red, green, and blue) of the RGB image <code>RGB1</code>. As with the colormap adjustment, you can use different parameter values for each plane by specifying <code>[low high]</code> and <code>[bottom top]</code> as 2-by-3 matrices, and <code>gamma</code> as a 1-by-3 vector.</p>
Class Support	For syntaxes that include an input image (rather than a colormap), the input image can be of class <code>uint8</code> or <code>double</code> . The output image has the same class as the input image.
Remarks	If <code>top < bottom</code> , the output image is reversed (i.e., as in a negative).

Example

```
I = imread('pout.tif');
J = imadjust(I, [0.3 0.7], []);
imshow(I)
figure, imshow(J)
```

**See Also**

brighten, histeq

imapprox

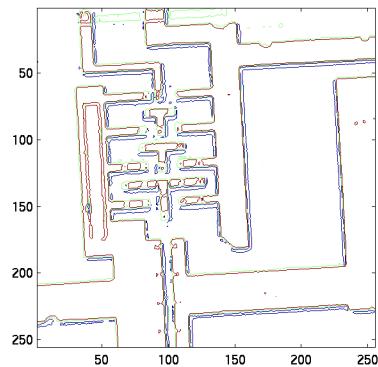
Purpose	Approximate indexed image by one with fewer colors
Syntax	<pre>[Y, newmap] = imapprox(X, map, n) [Y, newmap] = imapprox(X, map, tol) Y = imapprox(X, map, newmap) [...] = imapprox(..., dither_option)</pre>
Description	<p><code>[Y, newmap] = imapprox(X, map, n)</code> approximates the colors in the indexed image <code>X</code> and associated colormap <code>map</code> by using minimum variance quantization. <code>imapprox</code> returns indexed image <code>Y</code> with colormap <code>newmap</code>, which has at most <code>n</code> colors.</p> <p><code>[Y, newmap] = imapprox(X, map, tol)</code> approximates the colors in <code>X</code> and <code>map</code> through uniform quantization. <code>newmap</code> contains at most $(\text{floor}(1/\text{tol})+1)^3$ colors. <code>tol</code> must be between 0 and 1.0.</p> <p><code>Y = imapprox(X, map, newmap)</code> approximates the colors in <code>map</code> by using colormap mapping to find the colors in <code>newmap</code> that best match the colors in <code>map</code>.</p> <p><code>Y = imapprox(..., dither_option)</code> enables or disables dithering. <code>dither_option</code> is a string that can have one of these values:</p> <ul style="list-style-type: none">'dither' dithers, if necessary, to achieve better color resolution at the expense of spatial resolution (default).'nodiither' maps each color in the original image to the closest color in the new map. No dithering is performed.
Class Support	The input image <code>X</code> can be of class <code>uint8</code> or <code>double</code> . The output image <code>Y</code> is of class <code>uint8</code> if the length of <code>newmap</code> is less than or equal to 256. If the length of <code>newmap</code> is greater than 256, <code>X</code> is of class <code>double</code> .
Algorithm	<code>imapprox</code> uses <code>rgb2ind</code> to create a new colormap that uses fewer colors.
See Also	<code>cmuniquue</code> , <code>dither</code> , <code>rgb2ind</code>

Purpose	Create a contour plot of image data
Syntax	<code>imcontour(I, n)</code> <code>imcontour(I, v)</code> <code>imcontour(x, y, ...)</code> <code>imcontour(..., LineSpec)</code> <code>[C, h] = imcontour(...)</code>
Description	<code>imcontour(I, n)</code> draws a contour plot of the intensity image <code>I</code> , automatically setting up the axes so their orientation and aspect ratio match the image. <code>n</code> is the number of equally spaced contour levels in the plot; if you omit the argument, the number of levels and the values of the levels are chosen automatically.
	<code>imcontour(I, v)</code> draws a contour plot of <code>I</code> with contour lines at the data values specified in vector <code>v</code> . The number of contour levels is equal to <code>length(v)</code> .
	<code>imcontour(x, y, ...)</code> uses the vectors <code>x</code> and <code>y</code> to specify the x- and y-axis limits.
	<code>imcontour(..., LineSpec)</code> draws the contours using the line type and color specified by <code>LineSpec</code> . Marker symbols are ignored.
	<code>[C, h] = imcontour(...)</code> returns the contour matrix <code>C</code> and a vector of handles to the objects in the plot. (The objects are actually patches, and the lines are the edges of the patches.) You can use the <code>clabel</code> function with the contour matrix <code>C</code> to add contour labels to the plot.
Class Support	The input image can be of class <code>uint8</code> or <code>double</code> .

imcontour

Example

```
I = imread('i.c.tif');
imcontour(I, 3)
```



See Also

[clabel](#), [contour](#), [LineSpec](#) in the online MATLAB Function Reference

Purpose	Crop an image
Syntax	<pre>I2 = imcrop(I) X2 = imcrop(X, map) RGB2 = imcrop(RGB)</pre>
	<pre>I2 = imcrop(I, rect) X2 = imcrop(X, map, rect) RGB2 = imcrop(RGB, rect)</pre>
	<pre>[...] = imcrop(x, y, ...) [A, rect] = imcrop(...) [x, y, A, rect] = imcrop(...)</pre>
Description	<p><code>imcrop</code> crops an image to a specified rectangle. In the syntaxes below, <code>imcrop</code> displays the input image and waits for you to specify the crop rectangle with the mouse:</p> <pre>I2 = imcrop(I) X2 = imcrop(X, map) RGB2 = imcrop(RGB)</pre> <p>If you omit the input arguments, <code>imcrop</code> operates on the image in the current axes.</p> <p>To specify the rectangle:</p> <ul style="list-style-type: none">• For a single-button mouse, press the mouse button and drag to define the crop rectangle. Finish by releasing the mouse button.• For a 2- or 3-button mouse, press the left mouse button and drag to define the crop rectangle. Finish by releasing the mouse button. <p>If you hold down the Shift key while dragging, or if you press the right mouse button on a 2- or 3-button mouse, <code>imcrop</code> constrains the bounding rectangle to be a square.</p> <p>When you release the mouse button, <code>imcrop</code> returns the cropped image in the supplied output argument. If you do not supply an output argument, <code>imcrop</code> displays the output image in a new figure.</p>

You can also specify the cropping rectangle noninteractively, using these syntaxes:

```
I2 = imcrop(I, rect)
X2 = imcrop(X, map, rect)
RGB2 = imcrop(RGB, rect)
```

rect is a 4-element vector with the form [x_{min} y_{min} width height]; these values are specified in spatial coordinates.

To specify a nondefault spatial coordinate system for the input image, precede the other input arguments with two 2-element vectors specifying the XData and YData:

```
[...] = imcrop(x, y, ...)
```

If you supply additional output arguments, imcrop returns information about the selected rectangle and the coordinate system of the input image:

```
[A, rect] = imcrop(...)
[x, y, A, rect] = imcrop(...)
```

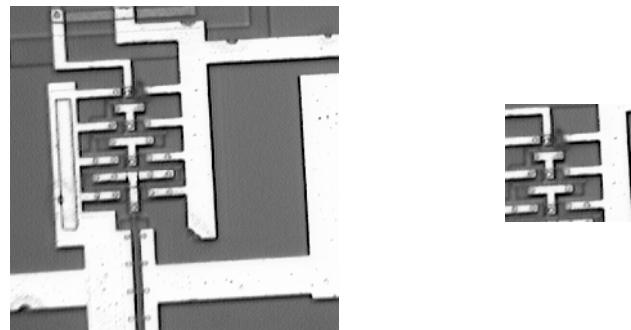
A is the output image. x and y are the XData and YData of the input image.

Class Support	The input image A can be of class uint8 or double. The output image B is of the same class as A. rect is always of class double.
----------------------	--

Remarks	Because rect is specified in terms of spatial coordinates, the width and height elements of rect do not always correspond exactly with the size of the output image. For example, suppose rect is [20 20 40 30], using the default spatial coordinate system. The upper-left corner of the specified rectangle is the center of the pixel (20,20) and the lower-right corner is the center of the pixel (50,60). The resulting output image is 31-by-41, not 30-by-40, because the output image includes all pixels in the input image that are completely <i>or partially</i> enclosed by the rectangle.
----------------	---

Example

```
I = imread('ic.tif');
I2 = imcrop(I, [60 40 100 90]);
imshow(I)
figure, imshow(I2)
```

**See Also**

zoom

imfeature

Purpose	Compute feature measurements for image regions	
Syntax	<pre>stats = imfeature(L, <i>measurements</i>) stats = imfeature(L, <i>measurements</i>, n)</pre>	
Description	<p><code>stats = imfeature(L, <i>measurements</i>)</code> computes a set of measurements for each labeled region in the label matrix <code>L</code>. Positive integer elements of <code>L</code> correspond to different regions. For example, the set of elements of <code>L</code> equal to 1 corresponds to region 1; the set of elements of <code>L</code> equal to 2 corresponds to region 2; and so on. <code>stats</code> is a structure array of length <code>max(L(:))</code>. The fields of the structure array denote different measurements for each region, as specified by <i>measurements</i>.</p> <p><i>measurements</i> can be a comma-separated list of strings, a cell array containing strings, the single string '<code>'all'</code>', or the single string '<code>'basic'</code>'. The set of valid measurement strings includes:</p>	
<hr/>		
'Area'	'Image'	'EulerNumber'
'Centroid'	'FilledImage'	'Extrema'
'BoundingBox'	'FilledArea'	'EquivalentDiameter'
'MajorAxisLength'	'ConvexHull'	'Solidity'
'MinorAxisLength'	'ConvexImage'	'Extent'
'Eccentricity'	'ConvexArea'	'PixelList'
'Orientation'		

Measurement strings are case insensitive and can be abbreviated.

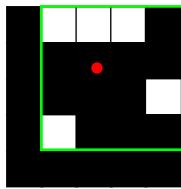
If *measurements* is the string '`'all'`', then all of the above measurements are computed. If *measurements* is not specified or if it is the string '`'basic'`', then these measurements are computed: '`Area`', '`Centroid`', and '`BoundingBox`'.

`stats = imfeature(L, measurements, n)` specifies the type of connectivity used in computing the '`FilledImage`', '`FilledArea`', and '`EulerNumber`' measurements. `n` can have a value of either 4 or 8, where 4 specifies 4-connected objects and 8 specifies 8-connected objects; if the argument is omitted, it defaults to 8.

Definitions

- 'Area' – Scalar; the actual number of pixels in the region. (This value may differ slightly from the value returned by bwarea, which weights different patterns of pixels differently.)
- 'Centroid' – 1-by-2 vector; the x - and y -coordinates of the center of mass of the region.
- 'BoundingBox' – 1-by-4 vector; the smallest rectangle that can contain the region. The format of the vector is [x y width height], where x and y are the x - and y -coordinates of the upper-left corner of the rectangle, and width and height are the width and height of the rectangle. Note that x and y are always noninteger values, because they are the spatial coordinates for the upper-left corner of a pixel in the image; for example, if this pixel is the third pixel in the fifth row of the image, then $x = 2.5$ and $y = 4.5$.

This figure illustrates the centroid and bounding box. The region consists of the white pixels; the green box is the bounding box, and the red dot is the centroid:



- 'MajorAxisLength' – Scalar; the length (in pixels) of the major axis of the ellipse that has the same second-moments as the region.
- 'MinorAxisLength' – Scalar; the length (in pixels) of the minor axis of the ellipse that has the same second-moments as the region.
- 'Eccentricity' – Scalar; the eccentricity of the ellipse that has the same second-moments as the region. The eccentricity is the ratio of the distance between the foci of the ellipse and its major axis length. The value is between 0 and 1. (0 and 1 are degenerate cases; an ellipse whose eccentricity is 0 is actually a circle, while an ellipse whose eccentricity is 1 is a line segment.)
- 'Orientation' – Scalar; the angle (in degrees) between the x -axis and the major axis of the ellipse that has the same second-moments as the region.

This figure illustrates the axes and orientation of the ellipse. The left side of the figure shows an image region and its corresponding ellipse. The right side

shows the same ellipse, with features indicated graphically; the solid blue lines are the axes, the red dots are the foci, and the orientation is the angle between the horizontal dotted line and the major axis:

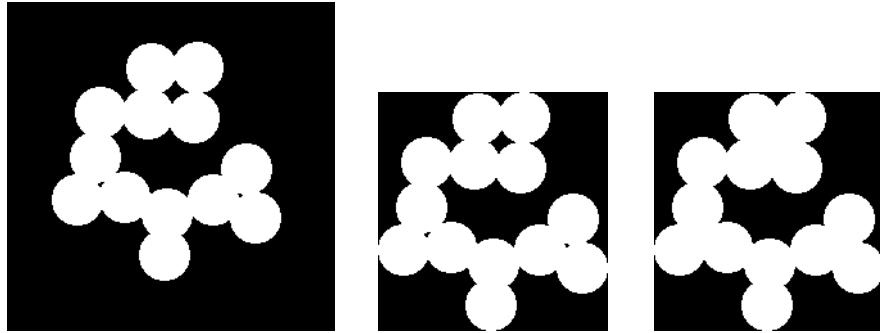


'Image' – Binary image (uint8) of the same size as the bounding box of the region; the on pixels correspond to the region, and all other pixels are off.

'FilledImage' – Binary image (uint8) of the same size as the bounding box of the region; the on pixels correspond to the region, with all holes filled in.

'FilledArea' – Scalar; the number of on pixels in FilledImage.

This figure illustrates 'Image' and 'FilledImage' :



Original image, containing a single region

'Image'

'FilledImage'

'ConvexHull' – p-by-2 matrix; the smallest convex polygon that can contain the region. Each row of the matrix contains the x- and y-coordinates of one vertex of the polygon.

'ConvexImage' – Binary image (uint8); the convex hull, with all pixels within the hull filled in (i.e., set to on). (For pixels that the boundary of the hull passes

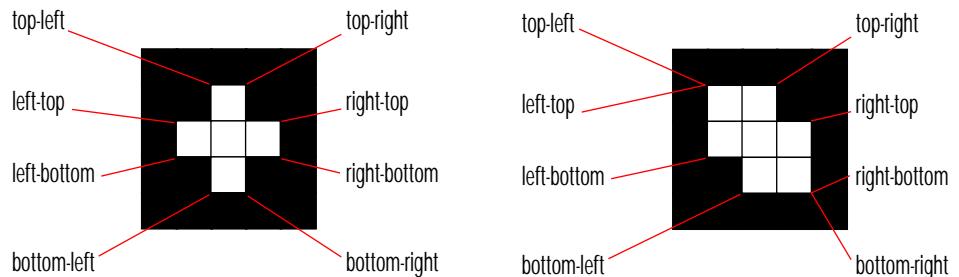
through, `imfeature` uses the same logic as `roi poly` to determine whether the pixel is inside or outside the hull.) The image is the size of the bounding box of the region.

'ConvexArea' – Scalar; the number of pixels in 'ConvexImage' .

'EulerNumber' – Scalar; equal to the number of objects in the region minus the number of holes in those objects.

'Extrema' – 8-by-2 matrix; the extremal points in the region. Each row of the matrix contains the *x*- and *y*-coordinates of one of the points; the format of the vector is [top-left top-right right-top right-bottom bottom-right bottom-left left-bottom left-top].

This figure illustrates the extrema of two different regions. In the region on the left, each extremal point is distinct; in the region on the right, certain extremal points (e.g., top-left and left-top) are identical:



'EquivalentDiameter' – Scalar; the diameter of a circle with the same area as the region. Computed as $\sqrt{4 * \text{Area}/\pi}$.

'Solidity' – Scalar; the proportion of the pixels in the convex hull that are also in the region. Computed as $\text{Area}/\text{ConvexArea}$.

'Extent' – Scalar; the proportion of the pixels in the bounding box that are also in the region. Computed as the Area divided by area of the bounding box.

'PixelList' – p-by-2 matrix; the actual pixels in the region. Each row of the matrix contains the *x*- and *y*-coordinates of one pixel in the region.

Class Support

The input label matrix *L* can be of class `uint8` or `double`.

Remarks

The comma-separated list syntax for structure arrays is very useful when working with the output of `imfeature`. For example, for a field that contains a scalar, you can use a this syntax to create a vector containing the value of this field for each region in the image.

For instance, if `stats` is a structure array with field `Area`, then these two expressions are equivalent:

```
stats(1).Area, stats(2).Area, ..., stats(end).Area
```

and:

```
stats.Area
```

Therefore, you can use these calls to create a vector containing the area of each region in the image:

```
stats = imfeature(L, 'Area');  
allArea = [stats.Area];
```

`allArea` is a vector of the same length as the structure array `stats`.

The function `ismember` is useful in conjunction with `imfeature` for selecting regions based on certain criteria. For example, these commands create a binary image containing only the regions in `text.tif` whose area is greater than 80:

```
idx = find([stats.Area] > 80);  
BW2 = ismember(L, idx);
```

Most of the measurements take very little time to compute. The exceptions are these, which may take significantly longer, depending on the number of regions in `L`:

- 'ConvexHull'
- 'ConvexImage'
- 'ConvexArea'
- 'FilledImage'

Note that computing certain groups of measurements takes about the same amount of time as computing just one of them, because `imfeature` takes advantage of intermediate computations used in both computations. Therefore, it is fastest to compute all of the desired measurements in a single call to `imfeature`.

Example

```
BW = imread('text.tif');
L = bwlabel(BW);
stats = imfeature(L, 'all');
stats(23)

ans =

    Area: 89
    Centroid: [95.6742 192.9775]
    BoundingBox: [87.5000 184.5000 16 15]
    MajorAxisLength: 19.9127
    MinorAxisLength: 14.2953
    Eccentricity: 0.6961
    Orientation: 9.0845
    ConvexHull: [28x2 double]
    ConvexImage: [15x16 uint8 ]
    ConvexArea: 205
    Image: [15x16 uint8 ]
    FilledImage: [15x16 uint8 ]
    FilledArea: 122
    EulerNumber: 0
    Extrema: [ 8x2 double]
    EquivalentDiameter: 10.6451
    Solidity: 0.4341
    Extent: 0.3708
    PixelList: [89x2 double]
```

See Also

bwlabel

[ismember](#) in the online MATLAB Function Reference

imfinfo

Purpose	Return information about a graphics file
Syntax	<code>info = imfinfo(filename, fmt)</code> <code>info = imfinfo(filename)</code>
Description	<code>info = imfinfo(filename, fmt)</code> returns a structure whose fields contain information about an image in a graphics file. <code>filename</code> is a string that specifies the name of the graphics file, and <code>fmt</code> is a string that specifies the format of the file. The file must be in the current directory or in a directory on the MATLAB path. If <code>imfinfo</code> cannot find a file named <code>filename</code> , it looks for a file named <code>filename.</code> <code>fmt</code> .

This table lists the possible values for `fmt`:

Format	File type
'bmp'	Windows Bitmap (BMP)
'hdf'	Hierarchical Data Format (HDF)
'jpg' or 'jpeg'	Joint Photographic Experts Group (JPEG)
'pcx'	Windows Paintbrush (PCX)
'tif' or 'tiff'	Tagged Image File Format (TIFF)
'xwd'	X Windows Dump (XWD)

If `filename` is a TIFF or HDF file containing more than one image, `info` is a structure array with one element (i.e., an individual structure) for each image in the file. For example, `info(3)` would contain information about the third image in the file.

The set of fields in `info` depends on the individual file and its format. However, the first nine fields are always the same. This table lists these fields and describes their values:

Field	Value
<code>filename</code>	A string containing the name of the file; if the file is not in the current directory, the string contains the full pathname of the file
<code>fileModDate</code>	A string containing the date when the file was last modified
<code>fileSize</code>	An integer indicating the size of the file in bytes
<code>format</code>	A string containing the file format, as specified by <code>fmt</code> ; for JPEG and TIFF files, the three-letter variant is returned
<code>formatVersion</code>	A string or number describing the version of the format
<code>width</code>	An integer indicating the width of the image in pixels
<code>height</code>	An integer indicating the height of the image in pixels
<code>bitDepth</code>	An integer indicating the number of bits per pixel
<code>colorType</code>	A string indicating the type of image; either ' <code>truecolor</code> ' for a truecolor RGB image, ' <code>grayscale</code> ' for a grayscale intensity image, or ' <code>indexed</code> ' for an indexed image

`info = imfinfo(filename)` attempts to infer the format of the file from its contents.

Remarks

`imfinfo` is a function in MATLAB.

imfinfo

Example

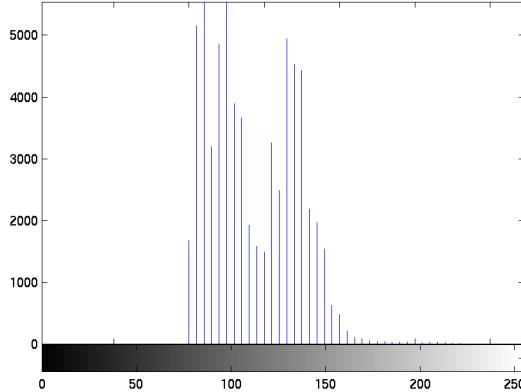
```
info = imfinfo('canoe.tif')

info =

    Filename: 'canoe.tif'
    FileModDate: '25-Oct-1996 22:10:39'
    FileSize: 69708
    Format: 'tif'
    FormatVersion: []
    Width: 346
    Height: 207
    BitDepth: 8
    ColorType: 'indexed'
    FormatSignature: [73 73 42 0]
    ByteOrder: 'little-endian'
    NewSubfileType: 0
    BitsPerSample: 8
    Compression: 'PackBits'
    PhotometricInterpretation: 'RGB_Palette'
    StripOffsets: [ 9x1 double]
    SamplesPerPixel: 1
    RowsPerStrip: 23
    StripByteCounts: [ 9x1 double]
    XResolution: 72
    YResolution: 72
    ResolutionUnit: 'Inch'
    Colormap: [256x3 double]
    PlanarConfiguration: 'Chunky'
    TileWidth: []
    TileLength: []
    TileOffsets: []
    TileByteCounts: []
    Orientation: 1
    FillOrder: 1
    GrayResponseUnit: 0.0100
    MaxSampleValue: 255
    MinSampleValue: 0
    Thresholding: 1
```

See Also

[imread](#), [imwrite](#)

Purpose	Display a histogram of image data
Syntax	<code>imhist(I, n)</code> <code>imhist(X, map)</code> <code>[counts, x] = imhist(...)</code>
Description	<p><code>imhist(I, n)</code> displays a histogram with n bins for the intensity image I above a grayscale colorbar of length n. If you omit the argument, <code>imhist</code> uses a default value of $n = 256$ if I is a grayscale image, or $n = 2$ if I is a binary image.</p> <p><code>imhist(X, map)</code> displays a histogram for the indexed image X. This histogram shows the distribution of pixel values above a colorbar of the colormap map. The colormap must be at least as long as the largest index in X. The histogram has one bin for each entry in the colormap.</p> <p><code>[counts, x] = imhist(...)</code> returns the histogram counts in <code>counts</code> and the bin locations in <code>x</code> so that <code>stem(x, counts)</code> shows the histogram. For indexed images, it returns the histogram counts for each colormap entry; the length of <code>counts</code> is the same as the length of the colormap.</p>
Class Support	The input image can be of class <code>uint8</code> or <code>double</code> .
Example	<pre>I = imread('pout.tif'); imhist(I)</pre> 
See Also	histeq hist in the online MATLAB Function Reference

immmovie

Purpose	Make a movie of a multiframe indexed image
Syntax	<code>mov = immovie(X, map)</code>
Description	<code>mov = immovie(X, map)</code> returns the movie matrix <code>mov</code> from the images in the multiframe indexed image <code>X</code> . As it creates the movie matrix, it displays the movie frames on the screen. You can play the movie using the MATLAB <code>movie</code> function.
	<code>X</code> comprises multiple indexed images, all having the same size and all using the colormap <code>map</code> . <code>X</code> is an <code>m</code> -by- <code>n</code> -by- <code>1</code> -by- <code>k</code> array, where <code>k</code> is the number of images.
Class Support	<code>X</code> can be of class <code>uint8</code> or <code>double</code> . <code>mov</code> is of class <code>double</code> .
Example	<pre>load mri mov = immovie(D, map);</pre>
See Also	<code>montage</code> <code>getframe</code> , <code>movie</code> in the online MATLAB Function Reference

Purpose	Add noise to an image
Syntax	<code>J = imnoise(I, type)</code> <code>J = imnoise(I, type, parameters)</code>
Description	<code>J = imnoise(I, type)</code> adds noise of type <code>type</code> to the intensity image <code>I</code> . <code>type</code> is a string that can have one of these values: <ul style="list-style-type: none"> • '<code>gaussian</code>' for Gaussian white noise • '<code>salt & pepper</code>' for "on and off" pixels • '<code>speckle</code>' for multiplicative noise The <code>parameters</code> you supply depend on the type of noise. If you omit these arguments, <code>imnoise</code> uses default values for the parameters.
	<code>J = imnoise(I, 'gaussian', m, v)</code> adds Gaussian white noise of mean <code>m</code> and variance <code>v</code> to the image <code>I</code> . The default is zero mean noise with 0.01 variance.
	<code>J = imnoise(I, 'salt & pepper', d)</code> adds salt and pepper noise to the image <code>I</code> , where <code>d</code> is the noise density. This affects approximately <code>d*prod(size(I))</code> pixels. The default is 0.05 noise density.
	<code>J = imnoise(I, 'speckle', v)</code> adds multiplicative noise to the image <code>I</code> , using the equation <code>J = I + n*I</code> , where <code>n</code> is uniformly distributed random noise with mean 0 and variance <code>v</code> . The default for <code>v</code> is 0.04.
Class Support	The input image <code>I</code> can be of class <code>uint8</code> or <code>double</code> . The output image <code>J</code> is of the same class as <code>I</code> .

imnoise

Example

```
I = imread('eighth.tif');
J = imnoise(I, 'salt & pepper', 0.02);
imshow(I)
figure, imshow(J)
```



See Also

[rand](#), [randn](#) in the online MATLAB Function Reference

Purpose Determine pixel color values

Syntax

```
P = impixel(I)
P = impixel(X, map)
P = impixel(RGB)

P = impixel(I, c, r)
P = impixel(X, map, c, r)
P = impixel(RGB, c, r)
[c, r, P] = impixel(...)

P = impixel(x, y, I, xi, yi)
P = impixel(x, y, X, map, xi, yi)
P = impixel(x, y, RGB, xi, yi)
[xi, yi, P] = impixel(x, y, ...)
```

Description

`impixel` returns the red, green, and blue color values of specified image pixels. In the syntaxes below, `impixel` displays the input image and waits for you to specify the pixels with the mouse:

```
P = impixel(I)
P = impixel(X, map)
P = impixel(RGB)
```

If you omit the input arguments, `impixel` operates on the image in the current axes.

Use normal button clicks to select pixels. Press **Backspace** or **Delete** to remove the previously selected pixel. A shift-click, right-click, or double-click adds a final pixel and ends the selection; pressing **Return** finishes the selection without adding a pixel.

When you finish selecting pixels, `impixel` returns an m -by-3 matrix of RGB values in the supplied output argument. If you do not supply an output argument, `impixel` returns the matrix in `ans`.

You can also specify the pixels noninteractively, using these syntaxes:

```
P = impixel(I, c, r)
P = impixel(X, map, c, r)
P = impixel(RGB, c, r)
```

r and c are equal-length vectors specifying the coordinates of the pixels whose RGB values are returned in P. The k-th row of P contains the RGB values for the pixel (r(k), c(k)).

If you supply three output arguments, `impixel` returns the coordinates of the selected pixels:

```
[c, r, P] = impixel(...)
```

To specify a nondefault spatial coordinate system for the input image, use these syntaxes:

```
P = impixel(x, y, I, xi, yi)
P = impixel(x, y, X, map, xi, yi)
P = impixel(x, y, RGB, xi, yi)
```

x and y are 2-element vectors specifying the image XData and YData. xi and yi are equal-length vectors specifying the spatial coordinates of the pixels whose RGB values are returned in P. If you supply three output arguments, `impixel` returns the coordinates of the selected pixels:

```
[xi, yi, P] = impixel(x, y, ...)
```

Class Support The input image can be of class `uint8` or `double`. All other inputs and outputs are of class `double`.

Example

```
RGB = imread('flowers.tif');
c = [12 146 410];
r = [104 156 129];
pixels = impixel(RGB, c, r)
```

```
pixels =
```

```
   61    59    101
  253    240      0
  237     37    44
```

See Also `improfile`, `pixval`

Purpose Compute pixel-value cross-sections along line segments

Syntax

```
c = improfile
```

```
c = improfile(n)
```

```
c = improfile(I, xi, yi)
```

```
c = improfile(I, xi, yi, n)
```

```
[cx, cy, c] = improfile(...)
```

```
[cx, cy, c, xi, yi] = improfile(...)
```

```
[...] = improfile(x, y, I, xi, yi)
```

```
[...] = improfile(x, y, I, xi, yi, n)
```

```
[...] = improfile(..., method)
```

Description

`improfile` computes the intensity values along a line or a multiline path in an image. `improfile` selects equally spaced points along the path you specify, and then uses interpolation to find the intensity value for each point. `improfile` works with grayscale intensity images and RGB images.

If you call `improfile` with one of these syntaxes, it operates interactively on the image in the current axes:

```
c = improfile
```

```
c = improfile(n)
```

`n` specifies the number of points to compute the intensity value for. If you do not provide this argument, `improfile` chooses a value for `n`, roughly equal to the number of pixels the path traverses.

You specify the line or path using the mouse, by clicking on points in the image. Press **Backspace** or **Delete** to remove the previously selected point. A shift-click, right-click, or double-click adds a final point and ends the selection; pressing **Return** finishes the selection without adding a point. When you finish selecting points, `improfile` returns the interpolated data values in `c`. `c` is an n -by-1 vector if the input is a grayscale intensity image, or an n -by-1-by-3 array if the input is an RGB image.

improfile

If you omit the output argument, `improfile` displays a plot of the computed intensity values. If the specified path consists of a single line segment, `improfile` creates a two-dimensional plot of intensity values versus the distance along the line segment; if the path consists of two or more line segments, `improfile` creates a three-dimensional plot of the intensity values versus their x- and y-coordinates.

You can also specify the path noninteractively, using these syntaxes:

```
c = improfile(I, xi, yi)  
c = improfile(I, xi, yi, n)
```

`xi` and `yi` are equal-length vectors specifying the spatial coordinates of the endpoints of the line segments.

You can use these syntaxes to return additional information:

```
[cx, cy, c] = improfile(...)  
[cx, cy, c, xi, yi] = improfile(...)
```

`cx` and `cy` are vectors of length `n`, containing the spatial coordinates of the points at which the intensity values are computed.

To specify a nondefault spatial coordinate system for the input image, use these syntaxes:

```
[...] = improfile(x, y, I, xi, yi)  
[...] = improfile(x, y, I, xi, yi, n)
```

`x` and `y` are 2-element vectors specifying the image `XData` and `YData`.

`[..., method]` uses the specified interpolation method. `method` is a string that can have one of these values:

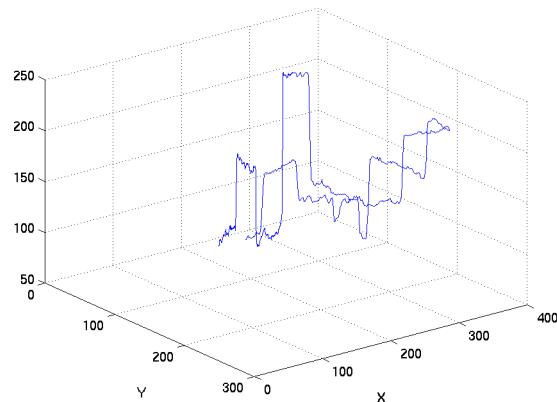
- '`'nearest'` (default) uses nearest neighbor interpolation.
- '`'bilinear'` uses bilinear interpolation.
- '`'bicubic'` uses bicubic interpolation.

If you omit the `method` argument, `improfile` uses the default method of '`'nearest'`.

Class Support	The input image can be of class <code>uint8</code> or <code>double</code> . All other inputs and outputs are of class <code>double</code> .
----------------------	---

Example

```
I = imread(' alumgrns.tif');
x = [35 338 346 103];
y = [253 250 17 148];
improfile(I, x, y), grid on
```

**See Also**

[improfile](#), [pixval](#)

[interp2](#) in the online MATLAB Function Reference

imread

Purpose	Read images from graphics files														
Syntax	<pre>A = imread(filename, fmt) [X, map] = imread(filename, fmt) [...] = imread(filename) [...] = imread(..., idx) (TIFF only) [...] = imread(..., ref) (HDF only)</pre>														
Description	<p><code>A = imread(filename, fmt)</code> reads the image in <code>filename</code> into <code>A</code>, whose class is <code>uint8</code>. If the file contains a grayscale intensity image, <code>A</code> is a two-dimensional array. If the file contains a truecolor (RGB) image, <code>A</code> is a three-dimensional (<code>m</code>-by-<code>n</code>-by-3) array. <code>filename</code> is a string that specifies the name of the graphics file, and <code>fmt</code> is a string that specifies the format of the file. The file must be in the current directory or in a directory in the MATLAB path, or you can specify the full pathname for a file located anywhere on your system. If <code>imread</code> cannot find a file named <code>filename</code>, it looks for a file named <code>filename</code>.<code>fmt</code>.</p> <p>This table lists the possible values for <code>fmt</code>:</p> <hr/> <table><thead><tr><th>Format</th><th>File type</th></tr></thead><tbody><tr><td>'bmp'</td><td>Windows Bitmap (BMP)</td></tr><tr><td>'hdf'</td><td>Hierarchical Data Format (HDF)</td></tr><tr><td>'jpg' or 'jpeg'</td><td>Joint Photographic Experts Group (JPEG)</td></tr><tr><td>'pcx'</td><td>Windows Paintbrush (PCX)</td></tr><tr><td>'tif' or 'tiff'</td><td>Tagged Image File Format (TIFF)</td></tr><tr><td>'xwd'</td><td>X Windows Dump (XWD)</td></tr></tbody></table> <hr/>	Format	File type	'bmp'	Windows Bitmap (BMP)	'hdf'	Hierarchical Data Format (HDF)	'jpg' or 'jpeg'	Joint Photographic Experts Group (JPEG)	'pcx'	Windows Paintbrush (PCX)	'tif' or 'tiff'	Tagged Image File Format (TIFF)	'xwd'	X Windows Dump (XWD)
Format	File type														
'bmp'	Windows Bitmap (BMP)														
'hdf'	Hierarchical Data Format (HDF)														
'jpg' or 'jpeg'	Joint Photographic Experts Group (JPEG)														
'pcx'	Windows Paintbrush (PCX)														
'tif' or 'tiff'	Tagged Image File Format (TIFF)														
'xwd'	X Windows Dump (XWD)														

`[X, map] = imread(filename, fmt)` reads the indexed image in `filename` into `X` and its associated colormap into `map`. `X` is of class `uint8`, and `map` is of class `double`. The colormap values are rescaled to the range [0,1].

`[...] = imread(filename)` attempts to infer the format of the file from its content.

`[...] = imread(..., idx)` reads in one image from a multi-image TIFF file. `idx` is an integer value that specifies the order in which the image appears in the file. For example, if `idx` is 3, `imread` reads the third image in the file. If you omit this argument, `imread` reads the first image in the file.

`[...] = imread(..., ref)` reads in one image from a multi-image HDF file. `ref` is an integer value that specifies the reference number used to identify the image. For example, if `ref` is 12, `imread` reads the image whose reference number is 12. (Note that in an HDF file the reference numbers do not necessarily correspond to the order of the images in the file.) If you omit this argument, `imread` reads the first image in the file.

This table summarizes the types of images that `imread` can read:

Format	Variants
BMP	1-bit, 4-bit, 8-bit, and 24-bit uncompressed images; 4-bit and 8-bit run-length encoded (RLE) images
HDF	8-bit raster image datasets, with or without associated colormap; 24-bit raster image datasets
JPEG	Any baseline JPEG image; JPEG images with some commonly used extensions
PCX	1-bit, 8-bit, and 24-bit images
TIFF	Any baseline TIFF image, including 1-bit, 8-bit, and 24-bit uncompressed images; 1-bit, 8-bit, and 24-bit images with packbit compression; 1-bit images with CCITT compression
XWD	1-bit and 8-bit ZPixmaps; XYBitmaps; 1-bit XYPixmaps

Remarks

`imread` is a function in MATLAB.

Examples

This example reads the sixth image in a TIFF file:

```
[X, map] = imread('flowers.tif', 6);
```

imread

This example reads the fourth image in an HDF file:

```
info = imfinfo('skull.hdf');  
[X, map] = imread('skull.hdf', info(4).Reference);
```

See Also

`imfinfo`, `imwrite`

`fread` in the online MATLAB Function Reference

Purpose	Resize an image
Syntax	$B = \text{imresize}(A, m, \text{method})$ $B = \text{imresize}(A, [mrows\ ncols], \text{method})$ $B = \text{imresize}(\dots, \text{method}, n)$ $B = \text{imresize}(\dots, \text{method}, h)$
Description	<p><code>imresize</code> resizes an image of any type using the specified interpolation method. <code>method</code> is a string that can have one of these values:</p> <ul style="list-style-type: none"> • 'nearest' (default) uses nearest neighbor interpolation. • 'bilinear' uses bilinear interpolation. • 'bicubic' uses bicubic interpolation. <p>If you omit the <code>method</code> argument, <code>imresize</code> uses the default method of 'nearest'.</p> <p><code>B = imresize(A, m, method)</code> returns an image that is <code>m</code> times the size of <code>A</code>. If <code>m</code> is between 0 and 1.0, <code>B</code> is smaller than <code>A</code>. If <code>m</code> is greater than 1.0, <code>B</code> is larger than <code>A</code>.</p> <p><code>B = imresize(A, [mrows\ ncols], method)</code> returns an image of size <code>[mrows\ ncols]</code>. If the specified size does not produce the same aspect ratio as the input image has, the output image is distorted.</p> <p>When the specified output size is smaller than the size of the input image, and <code>method</code> is 'bilinear' or 'bicubic', <code>imresize</code> applies a lowpass filter before interpolation to reduce aliasing. The default filter size is 11-by-11.</p> <p>You can specify a different order for the default filter using:</p> $[\dots] = \text{imresize}(\dots, \text{method}, n)$ <p><code>n</code> is an integer scalar specifying the size of the filter, which is <code>n</code>-by-<code>n</code>. If <code>n</code> is 0 (zero), <code>imresize</code> omits the filtering step.</p> <p>You can also specify your own filter <code>h</code> using:</p> $[\dots] = \text{imresize}(\dots, \text{method}, h)$ <p><code>h</code> is any two-dimensional FIR filter (such as those returned by <code>ftrans2</code>, <code>fwind1</code>, <code>fwind2</code>, or <code>fsamp2</code>).</p>

imresize

Class Support The input image can be of class `uint8` or `double`. The output image is of the same class as the input image.

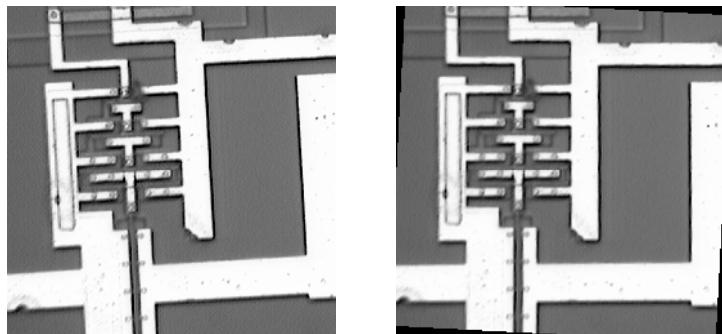
See Also `interp2` in the online MATLAB Function Reference

Purpose	Rotate an image
Syntax	$B = \text{imrotate}(A, \text{angle}, \text{method})$ $B = \text{imrotate}(A, \text{angle}, \text{method}, 'crop')$
Description	$B = \text{imrotate}(A, \text{angle}, \text{method})$ rotates the image A by angle degrees in a counter-clockwise direction, using the specified interpolation method. method is a string that can have one of these values: <ul style="list-style-type: none">• 'nearest' (default) uses nearest neighbor interpolation.• 'bilinear' uses bilinear interpolation.• 'bicubic' uses bicubic interpolation. If you omit the method argument, <code>imrotate</code> uses the default method of 'nearest'. The returned image matrix B is, in general, larger than A to include the whole rotated image. <code>imrotate</code> sets invalid values on the periphery of B to 0. $B = \text{imrotate}(A, \text{angle}, \text{method}, 'crop')$ rotates the image A through angle degrees and returns the central portion which is the same size as A.
Class Support	The input image can be of class <code>uint8</code> or <code>double</code> . The output image is of the same class as the input image.
Remarks	To rotate the image clockwise, specify a negative angle.

imrotate

Example

```
I = imread('i.c.tif');
J = imrotate(I,-4,'bilinear','crop');
imshow(I)
figure, imshow(J)
```



See Also

[imcrop](#), [imresize](#)

Purpose	Display an image
Syntax	<pre>imshow(I, n) imshow(I, [low high]) imshow(BW) imshow(X, map) imshow(RGB) imshow(..., display_option) imshow(x, y, A, ...) imshow filename h = imshow(...)</pre>
Description	<p><code>imshow(I, n)</code> displays the intensity image <code>I</code> with <code>n</code> discrete levels of gray. If you omit <code>n</code>, <code>imshow</code> uses 256 gray levels on 24-bit displays, or 64 gray levels on other systems.</p> <p><code>imshow(I, [low high])</code> displays <code>I</code> as a grayscale intensity image, specifying the data range for <code>I</code>. The value <code>low</code> (and any value less than <code>low</code>) displays as black, the value <code>high</code> (and any value greater than <code>high</code>) displays as white, and values in between display as intermediate shades of gray. <code>imshow</code> uses the default number of gray levels. If you use an empty matrix (<code>[]</code>) for <code>[low high]</code>, <code>imshow</code> uses <code>[min(I(:)) max(I(:))]</code>; the minimum value in <code>I</code> displays as black, and the maximum value displays as white.</p> <p><code>imshow(BW)</code> displays the binary image <code>BW</code>. Values of 0 display as black, and values of 1 display as white.</p> <p><code>imshow(X, map)</code> displays the indexed image <code>X</code> with the colormap <code>map</code>.</p> <p><code>imshow(RGB)</code> displays the truecolor image <code>RGB</code>.</p> <p><code>imshow(..., display_option)</code> displays the image, calling <code>truekiye</code> if <code>display_option</code> is '<code>truekiye</code>', or suppressing the call to <code>truekiye</code> if <code>display_option</code> is '<code>notruekiye</code>'. Either option string can be abbreviated. If you do not supply this argument, <code>imshow</code> determines whether to call <code>truekiye</code> based on the setting of the '<code>ImshowTruesize</code>' preference.</p> <p><code>imshow(x, y, A, ...)</code> uses the 2-element vectors <code>x</code> and <code>y</code> to establish a nondefault spatial coordinate system, by specifying the image <code>XData</code> and <code>YData</code>.</p>

imshow

`imshow filename` displays the image stored in the graphics file *filename*. `imshow` calls `imread` to read the image from the file, but the image data is not stored in the MATLAB workspace. The file must be in the current directory or on the MATLAB path.

`h = imshow(...)` returns the handle to the image object created by `imshow`.

Class Support The input image can be of class `uint8` or `double`.

Remarks You can use the `iptsetpref` function to set several toolbox preferences that modify the behavior of `imshow`:

- '`ImshowBorder`' controls whether `imshow` displays the image with a border around it.
- '`ImshowAxesVisible`' controls whether `imshow` displays the image with the axes box and tick labels.
- '`ImshowTrueSize`' controls whether `imshow` calls the `trueSize` function.

Note that the `display_opti`n argument to `imshow` enables you to override the '`ImshowTrueSize`' preference.

For more information about these preferences, see the reference entry for `iptsetpref`.

See Also `getimage`, `imread`, `iptgetpref`, `iptsetpref`, `subimage`, `trueSize`, `warp`
`image`, `imagesc` in the online MATLAB Function Reference

Purpose	Write an image to a graphics file
Syntax	<code>imwrite(A, filename, fmt)</code> <code>imwrite(X, map, filename, fmt)</code> <code>imwrite(..., filename)</code> <code>imwrite(..., Parameter, Value, ...)</code>
Description	<code>imwrite(A, filename, fmt)</code> writes the image in <code>A</code> to <code>filename</code> . <code>filename</code> is a string that specifies the name of the output file, and <code>fmt</code> is a string that specifies the format of the file. If <code>A</code> is a grayscale intensity image or a truecolor (RGB) image of class <code>uint8</code> , <code>imwrite</code> writes the actual values in the array to the file. If <code>A</code> is of class <code>double</code> , <code>imwrite</code> rescales the values in the array before writing, using <code>uint8(round(255*A))</code> . This operation converts the floating-point numbers in the range [0,1] to 8-bit integers in the range [0,255].

This table lists the possible values for `fmt`:

Format	File type
'bmp'	Windows Bitmap (BMP)
'hdf'	Hierarchical Data Format (HDF)
'jpg' or 'jpeg'	Joint Photographers Expert Group (JPEG)
'pcx'	Windows Paintbrush (PCX)
'tif' or 'tiff'	Tagged Image File Format (TIFF)
'xwd'	X Windows Dump (XWD)

`imwrite(X, map, filename, fmt)` writes the indexed image in `X`, and its associated colormap `map`, to `filename`. If `X` is of class `uint8`, `imwrite` writes the actual values in the array to the file. If `X` is of class `double`, `imwrite` offsets the values in the array before writing, using `uint8(X-1)`. `map` must be of class `double`; `imwrite` rescales the values in `map` using `uint8(round(255*map))`.

`imwrite(..., filename)` writes the image to `filename`, inferring the format to use from the filename's extension. The extension must be one of the legal values for `fmt`.

imwrite

`imwrite(..., Parameter, Value, ...)` specifies parameters that control various characteristics of the output file. Parameters are currently supported for HDF, JPEG, and TIFF files.

This table describes the available parameters for HDF files:

Parameter	Values	Default
'Compressi on'	One of these strings: 'none', 'rle', 'jpeg'	'rle'
'Qual i ty'	A number between 0 and 100; parameter applies only if 'Compressi on' is 'jpeg'; higher numbers mean quality is better (less image degradation due to compression), but the resulting file size is larger	75
'WriteMode'	One of these strings: 'overwrite', 'append'	'overwrite'

This table describes the available parameters for JPEG files:

Parameter	Values	Default
'Qual i ty'	A number between 0 and 100; higher numbers mean quality is better (less image degradation due to compression), but the resulting file size is larger	75

This table describes the available parameters for TIFF files:

Parameter	Values	Default
'Compression'	One of these strings: 'none', 'packbits', 'ccitt'; 'ccitt' is valid for binary images only	'ccitt' for binary images; 'packbits' for all other images
'Description'	Any string; fills in the ImageDescription field returned by <code>imfinfo</code>	empty
'Resolution'	A scalar value that is used for the XResolution and YResolution tags in the output file	72

This table summarizes the types of images that `imwrite` can write:

Format	Variants
BMP	8-bit uncompressed images with associated colormap; 24-bit uncompressed images
HDF	8-bit raster image datasets, with or without associated colormap; 24-bit raster image datasets
JPEG	Baseline JPEG images ¹
PCX	8-bit images
TIFF	Baseline TIFF images, including 1-bit, 8-bit, and 24-bit uncompressed images; 1-bit, 8-bit, and 24-bit images with packbit compression; 1-bit images with CCITT compression
XWD	8-bit ZPixmaps

1. Indexed images are converted to RGB before writing out JPEG files, because the JPEG format does not support indexed images.

imwrite

Remarks `imwrite` is a function in MATLAB.

Example `imwrite(X, map, 'flowers.hdf', 'Compressi on', 'none', ...
'WriteMode', 'append')`

See Also `imfinfo`, `imread`
`fwrite` in the online MATLAB Function Reference

Purpose	Convert an indexed image to an intensity image
Syntax	<code>I = ind2gray(X, map)</code>
Description	<code>I = ind2gray(X, map)</code> converts the image <code>X</code> with colormap <code>map</code> to an intensity image <code>I</code> . <code>ind2gray</code> removes the hue and saturation information from the input image while retaining the luminance.
Class Support	<code>X</code> can be of class <code>uint8</code> or <code>double</code> . <code>I</code> is of class <code>double</code> .
Example	<pre>load trees I = ind2gray(X, map); imshow(X, map) figure, imshow(I)</pre>
Algorithm	<code>ind2gray</code> converts the colormap to NTSC coordinates using <code>rgb2ntsc</code> , and sets the hue and saturation components (I and Q) to zero, creating a gray colormap. <code>ind2gray</code> then replaces the indices in the image <code>X</code> with the corresponding grayscale intensity values in the gray colormap.
See Also	<code>gray2ind</code> , <code>imshow</code> , <code>rgb2ntsc</code>



ind2rgb

Purpose	Convert an indexed image to an RGB image
Syntax	<code>RGB = ind2rgb(X, map)</code>
Description	<code>RGB = ind2rgb(X, map)</code> converts the matrix <code>X</code> and corresponding colormap <code>map</code> to RGB (truecolor) format.
Class Support	<code>X</code> can be of class <code>uint8</code> or <code>double</code> . <code>RGB</code> is an <code>m</code> -by- <code>n</code> -by- <code>3</code> array of class <code>double</code> .
See Also	<code>ind2gray</code> , <code>rgb2ind</code>

Purpose	Get Image Processing Toolbox preferences
Syntax	<code>val ue = iptgetpref(prefname)</code>
Description	<code>val ue = iptgetpref(prefname)</code> returns the value of the Image Processing Toolbox preference specified by the string <code>prefname</code> . Preference names are case insensitive and can be abbreviated. <code>iptgetpref</code> without an input argument displays the current setting of all Image Processing Toolbox preferences.
Example	<pre>val ue = iptgetpref('ImshowAxesVisible') val ue = off</pre>
See Also	<code>imshow</code> , <code>iptsetpref</code>

iptsetpref

Purpose	Set Image Processing Toolbox preferences
Syntax	<code>iptsetpref(prefname, value)</code>
Description	<code>iptsetpref(prefname, value)</code> sets the Image Processing Toolbox preference specified by the string <code>prefname</code> to <code>value</code> . The setting persists until the end of the current MATLAB session, or until you change the setting. (To make the value persist between sessions, put the command in your <code>startup.m</code> file.)
This table describes the available preferences. Note that the preference names are case insensitive and can be abbreviated:	
Preference Name	Values
' <code>ImshowBorder</code> '	' <code>loose</code> ' (default) or ' <code>tight</code> '
If ' <code>ImshowBorder</code> ' is ' <code>loose</code> ', <code>imshow</code> displays the image with a border between the image and the edges of the figure window, thus leaving room for axes labels, titles, etc. If ' <code>ImshowBorder</code> ' is ' <code>tight</code> ', <code>imshow</code> adjusts the figure size so that the image entirely fills the figure. (However, there may still be a border if the image is very small, or if there are other objects besides the image and its axes in the figure.)	

Preference Name	Values
'ImshowAxesVisible'	'on' or 'off' (default)
	If 'ImshowAxesVisible' is 'on', imshow displays the image with the axes box and tick labels. If 'ImshowAxesVisible' is 'off', imshow displays the image without the axes box and tick labels.
'ImshowTruesize'	'auto' (default) or 'manual'
	If 'ImshowTruesize' is 'manual', imshow does not call truesize. If 'ImshowTruesize' is 'auto', imshow automatically decides whether to call truesize. (imshow calls truesize if there will be no other objects in the resulting figure besides the image and its axes.) You can override this setting for an individual display by specifying the display_option argument to imshow, or you can call truesize manually after displaying the image.
'TruesizeWarning'	'on' (default) or 'off'
	If 'TruesizeWarning' is 'on', the truesize function displays a warning if the image is too large to fit on the screen. (The entire image is still displayed, but at less than true size.) If 'TruesizeWarning' is 'off', truesize does not display the warning. Note that this preference applies even when you call truesize indirectly, such as through imshow.

iptsetpref(prefname) displays the valid values for prefname.

Example

```
iptsetpref('ImshowBorder', 'tight')
```

See Also

imshow, iptgetpref, truesize

axes in the online MATLAB Function Reference

iradon

Purpose	Compute inverse Radon transform
Syntax	<pre>I = iradon(P, theta) I = iradon(P, theta, interp, filter, d, n)</pre>
Description	<p><code>I = iradon(P, theta)</code> reconstructs the image <code>I</code> from projection data in the two-dimensional array <code>P</code>. The columns of <code>P</code> are parallel beam projection data. <code>i radon</code> assumes that the center of rotation is the center point of the projections, which is defined as <code>ceil(size(P, 1)/2)</code>.</p> <p><code>theta</code> describes the angles (in degrees) at which the projections were taken. It can be either a vector containing the angles or a scalar specifying <code>D_theta</code>, the incremental angle between projections. If <code>theta</code> is a vector, it must contain angles with equal spacing between them. If <code>theta</code> is a scalar specifying <code>D_theta</code>, the projections are taken at angles <code>theta = m*D_theta</code>, where <code>m = 0, 1, 2, ..., size(P, 2)-1</code>. If the input is the empty matrix ([]), <code>D_theta</code> defaults to <code>180/size(P, 2)</code>.</p> <p><code>I = iradon(P, theta, interp, filter, d, n)</code> specifies parameters to use in the inverse Radon transform. You can specify any combination of the last four arguments. <code>i radon</code> uses default values for any of these arguments that you omit.</p> <p><code>interp</code> specifies the type of interpolation to use in the backprojection. The available options are listed in order of increasing accuracy and computational complexity:</p> <ul style="list-style-type: none">• 'nearest' – nearest neighbor interpolation• 'linear' – linear interpolation (default)• 'spline' – spline interpolation <p><code>filter</code> specifies the filter to use for frequency domain filtering. <code>filter</code> is a string that specifies any of the following standard filters:</p> <ul style="list-style-type: none">• 'Ram-Lak' – The cropped Ram-Lak or ramp filter (default). The frequency response of this filter is f. Because this filter is sensitive to noise in the projections, one of the filters listed below may be preferable. These filters

multiply the Ram-Lak filter by a window that de-emphasizes high frequencies.

- 'Shepp-Logan' – The Shepp-Logan filter multiplies the Ram-Lak filter by a sinc function.
- 'Cosine' – The cosine filter multiplies the Ram-Lak filter by a cosine function.
- 'Hamming' – The Hamming filter multiplies the Ram-Lak filter by a Hamming window.
- 'Hann' – The Hann filter multiplies the Ram-Lak filter by a Hann window.

d is a scalar in the range (0,1] that modifies the filter by rescaling its frequency axis. The default is 1. If d is less than 1, the filter is compressed to fit into the frequency range [0,d], in normalized frequencies; all frequencies above d are set to 0.

n is a scalar that specifies the number of rows and columns in the reconstructed image. If n is not specified, the size is determined from the length of the projections:

$$n = \text{2*floor(size}(P, 1) / (2*sqrt(2)))$$

If you specify n, iradon reconstructs a smaller or larger portion of the image, but does not change the scaling of the data. If the projections were calculated with the radon function, the reconstructed image may not be the same size as the original image.

[I, h] = iradon(. . .) returns the frequency response of the filter in the vector h.

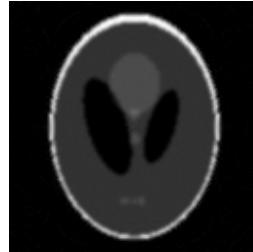
Class Support

All input arguments must be of class double. The output arguments are of class double.

iradon

Example

```
P = phantom(128);  
R = radon(P, 0: 179);  
I = iradon(R, 0: 179, 'nearest', 'Hann');  
imshow(P)  
figure, imshow(I)
```



Algorithm

`i radon` uses the filtered backprojection algorithm to perform the inverse Radon transform. The filter is designed directly in the frequency domain and then multiplied by the FFT of the projections. The projections are zero-padded to a power of 2 before filtering to prevent spatial domain aliasing and to speed up the FFT.

See Also

`radon`, `phantom`

References

Kak, Avinash C., and Malcolm Slaney, *Principles of Computerized Tomographic Imaging*. New York: IEEE Press.

Purpose	Return true for a binary image
Syntax	<code>flag = isbw(A)</code>
Description	<code>flag = isbw(A)</code> returns 1 if A is a binary image and 0 otherwise. <code>isbw</code> uses these criteria to decide if A is a binary image: <ul style="list-style-type: none">• If A is of class <code>double</code>, all values must be either 0 or 1, and the number of dimensions of A must be 2.• If A is of class <code>uint8</code>, its logical flag must be on, and the number of dimensions of A must be 2. Note that a four-dimensional array that contains multiple binary images returns 0, not 1.
Class Support	A can be of class <code>uint8</code> or <code>double</code> .
See Also	<code>isind</code> , <code>isgray</code> , <code>isrgb</code>

isgray

Purpose	Return true for intensity image
Syntax	<code>flag = isgray(A)</code>
Description	<code>flag = isgray(A)</code> returns 1 if A is a grayscale intensity image and 0 otherwise. <code>isgray</code> uses these criteria to decide if A is an intensity image: <ul style="list-style-type: none">• If A is of class <code>double</code>, all values must be in the range [0,1], and the number of dimensions of A must be 2.• If A is of class <code>uint8</code>, the number of dimensions of A must be 2. Note that a four-dimensional array that contains multiple intensity images returns 0, not 1.
Class Support	A can be of class <code>uint8</code> or <code>double</code> .
See Also	<code>isbw</code> , <code>isind</code> , <code>isrgb</code>

Purpose	Return true for an indexed image
Syntax	<code>flag = isind(A)</code>
Description	<code>flag = isind(A)</code> returns 1 if A is an indexed image and 0 otherwise. <code>isind</code> uses these criteria to determine if A is an indexed image: <ul style="list-style-type: none">• If A is of class <code>double</code>, all values in A must be integers greater than or equal to 1, and the number of dimensions of A must be 2.• If A is of class <code>uint8</code>, its logical flag must be off, and the number of dimensions of A must be 2. Note that a four-dimensional array that contains multiple indexed images returns 0, not 1.
Class Support	A can be of class <code>uint8</code> or <code>double</code> .
See Also	<code>isbw</code> , <code>isgray</code> , <code>isrgb</code>

isrgb

Purpose	Return true for an RGB image
Syntax	<code>flag = isrgb(A)</code>
Description	<code>flag = isrgb(A)</code> returns 1 if A is an RGB truecolor image and 0 otherwise. <code>isrgb</code> uses these criteria to determine if A is an RGB image: <ul style="list-style-type: none">• If A is of class <code>double</code>, all values must be in the range [0,1], and A must be <code>m-by-n-by-3</code>.• If A is of class <code>uint8</code>, A must be <code>m-by-n-by-3</code>. Note that a four-dimensional array that contains multiple RGB images returns 0, not 1.
Class Support	A can be of class <code>uint8</code> or <code>double</code> .
See Also	<code>isbw</code> , <code>isgray</code> , <code>isind</code>

Purpose Construct a lookup table for use with `applylut`

Syntax

```
lut = makelut(fun, n)
lut = makelut(fun, n, P1, P2, ...)
```

Description `lut = makelut(fun, n)` returns a lookup table for use with `applylut`. `fun` is either a string containing the name of a function or an inline function object. The function should take a 2-by-2 or 3-by-3 matrix of 1's and 0's as input and return a scalar. `n` is either 2 or 3, indicating the size of the input to `fun`. `makelut` creates `lut` by passing all possible 2-by-2 or 3-by-3 neighborhoods to `fun`, one at a time, and constructing either a 16-element vector (for 2-by-2 neighborhoods) or a 512-element vector (for 3-by-3 neighborhoods). The vector consists of the output from `fun` for each possible neighborhood.

```
lut = makelut(fun, n, P1, P2, ...)
```

 passes the additional parameters `P1, P2, ...`, to `fun`.

Class Support `lut` is returned as a vector of class `double`.

makelut

Example

In this example, the function returns 1 (true) if the number of 1's in the neighborhood is 2 or greater, and returns 0 (false) otherwise. `makelut` then uses the function to construct a lookup table for 2-by-2 neighborhoods.

```
f = inline('sum(x(:)) >= 2');
lut = makelut(f, 2)
```

```
lut =
```

```
0
0
0
1
0
1
1
1
0
1
1
1
1
1
1
1
```

See Also

`applylut`

Purpose

Convert a matrix to a grayscale intensity image

Syntax

```
I = mat2gray(A, [amin amax])  
I = mat2gray(A)
```

Description

`I = mat2gray(A, [amin amax])` converts the matrix A to the intensity image I. The returned matrix I contains values in the range 0 (black) to 1.0 (full intensity or white). `amin` and `amax` are the values in A that correspond to 0 and 1.0 in I.

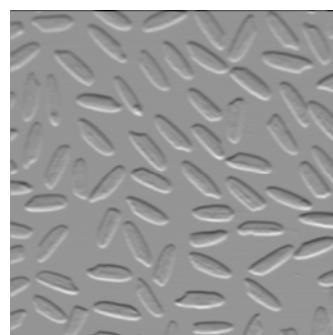
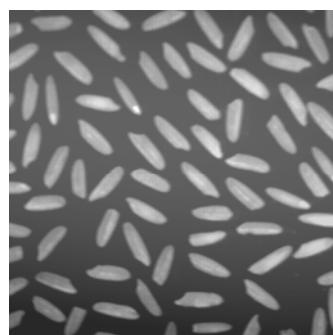
`I = mat2gray(A)` sets the values of `amin` and `amax` to the minimum and maximum values in A.

Class Support

The input array A and the output image I are of class double.

Example

```
I = imread('rice.tif');  
J = filter2(fspecial('sobel'), I);  
K = mat2gray(J);  
imshow(I)  
figure, imshow(K)
```

**See Also**

`gray2ind`

mean2

Purpose	Compute the mean of the elements of a matrix
Syntax	<code>b = mean2(A)</code>
Description	<code>b = mean2(A)</code> computes the mean of the values in <code>A</code> .
Class Support	<code>A</code> is an array of class <code>uint8</code> or <code>double</code> . <code>b</code> is a scalar of class <code>double</code> .
Algorithm	<code>mean2</code> computes the mean of an array <code>A</code> using <code>mean(A(:))</code> .
See Also	<code>std2</code> <code>mean</code> , <code>std</code> in the online MATLAB Function Reference

Purpose	Perform two-dimensional median filtering
Syntax	<pre>B = medfilt2(A, [m n]) B = medfilt2(A) B = medfilt2(A, 'indexed', ...)</pre>
Description	<p>Median filtering is a nonlinear operation often used in image processing to reduce “salt and pepper” noise. Median filtering is more effective than convolution when the goal is to simultaneously reduce noise and preserve edges.</p> <p><code>B = medfilt2(A, [m n])</code> performs median filtering of the matrix A in two dimensions. Each output pixel contains the median value in the m-by-n neighborhood around the corresponding pixel in the input image. <code>medfilt2</code> pads the image with zeros on the edges, so the median values for the points within $[m \ n]/2$ of the edges may appear distorted.</p> <p><code>B = medfilt2(A)</code> performs median filtering of the matrix A using the default 3-by-3 neighborhood.</p> <p><code>B = medfilt2(A, 'indexed', ...)</code> processes A as an indexed image, padding with zeros if the class of A is <code>uint8</code>, or ones if the class of A is <code>double</code>.</p>
Class Support	The input image A can be of class <code>uint8</code> or <code>double</code> . The output image B is of the same class as A.
Remarks	<p>If the input image A is of class <code>uint8</code>, all of the output values are returned as <code>uint8</code> integers. If the number of pixels in the neighborhood (i.e., $m*n$) is even, some of the median values may not be integers. In these cases, the fractional parts are discarded.</p> <p>For example, suppose you call <code>medfilt2</code> using 2-by-2 neighborhoods, and the input image is a <code>uint8</code> array that includes this neighborhood:</p> <pre>1 5 4 8</pre> <p><code>medfilt2</code> returns an output value of 4 for this neighborhood, although the true median is 4.5.</p>

medfilt2

Example

This example adds salt and pepper noise to an image, then restores the image using `medfilt2`.

```
I = imread('eight.tif');
J = imnoise(I, 'salt & pepper', 0.02);
K = medfilt2(J);
imshow(J)
figure, imshow(K)
```



Algorithm

`medfilt2` uses `ordfilt2` to perform the filtering.

See Also

`filter2`, `ordfilt2`, `wiener2`

Reference

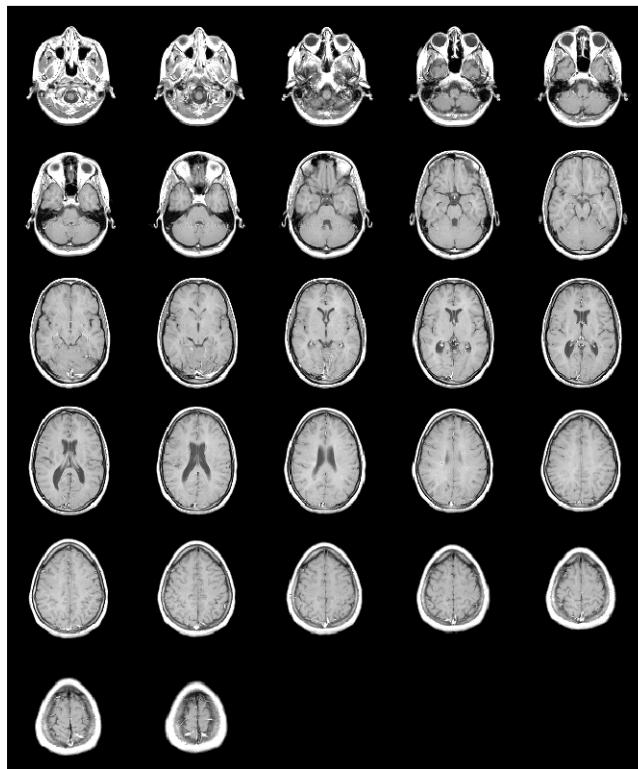
Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 469-476.

Purpose	Display multiple image frames as a rectangular montage
Syntax	<code>montage(I)</code> <code>montage(BW)</code> <code>montage(X, map)</code> <code>montage(RGB)</code> <code>h = montage(...)</code>
Description	<code>montage</code> displays all of the frames of a multiframe image array in a single image object, arranging the frames so that they roughly form a square. <code>montage(I)</code> displays the k frames of the intensity image array I. I is m-by-n-by-1-by-k. <code>montage(BW)</code> displays the k frames of the binary image array BW. BW is m-by-n-by-1-by-k. <code>montage(X, map)</code> displays the k frames of the indexed image array X, using the colormap map for all frames. X is m-by-n-by-1-by-k. <code>montage(RGB)</code> displays the k frames of the truecolor image array RGB. RGB is m-by-n-by-3-by-k. <code>h = montage(...)</code> returns the handle to the image object.
Class Support	The input image can be of class <code>uint8</code> or <code>double</code> .

montage

Example

```
load mri  
montage(D, map)
```



See Also

[immovie](#)

Purpose	Perform general sliding-neighborhood operations
Syntax	<pre>B = nlfilter(A, [m n], fun) B = nlfilter(A, [m n], fun, P1, P2, . . .) B = nlfilter(A, 'indexed', . . .)</pre>
Description	<p><code>B = nlfilter(A, [m n], fun)</code> applies the function <code>fun</code> to each m-by-n sliding block of <code>A</code>. <code>fun</code> can be a string containing the name of a function, a string containing an expression, or an inline function object. <code>fun</code> should accept an m-by-n block as input, and return a scalar result:</p> $c = \text{fun}(x)$ <p><code>c</code> is the output value for the center pixel in the m-by-n block <code>x</code>. <code>nlfilter</code> calls <code>fun</code> for each pixel in <code>A</code>. <code>nlfilter</code> zero pads the m-by-n block at the edges, if necessary.</p> <p><code>B = nlfilter(A, [m n], fun, P1, P2, . . .)</code> passes the additional parameters <code>P1, P2, . . . ,</code> to <code>fun</code>.</p> <p><code>B = nlfilter(A, 'indexed', . . .)</code> processes <code>A</code> as an indexed image, padding with ones if <code>A</code> is of class <code>double</code> and zeros if <code>A</code> is of class <code>uint8</code>.</p>
Class Support	The input image <code>A</code> can be of any class supported by <code>fun</code> . The class of <code>B</code> depends on the class of the output from <code>fun</code> .
Remarks	<code>nlfilter</code> can take a long time to process large images. In some cases, the <code>colfilt</code> function can perform the same operation much faster.
Example	This call produces the same result as calling <code>medfilt2</code> with a 3-by-3 neighborhood:
	<pre>B = nlfilter(A, [3 3], 'median(x(:))');</pre>
See Also	<code>bblkproc</code> , <code>colfilt</code>

ntsc2rgb

Purpose	Convert NTSC values to RGB color space
Syntax	<pre>rgbmap = ntsc2rgb(yi qmap) RGB = ntsc2rgb(YI Q)</pre>
Description	<p><code>rgbmap = ntsc2rgb(yi qmap)</code> converts the m-by-3 NTSC (television) color values in <code>yi qmap</code> to RGB color space. If <code>yi qmap</code> is m-by-3 and contains the NTSC luminance (Y) and chrominance (I and Q) color components as columns, then <code>rgbmap</code> is an m-by-3 matrix that contains the red, green, and blue values equivalent to those colors. Both <code>rgbmap</code> and <code>yi qmap</code> contain intensities in the range 0 to 1.0. The intensity 0 corresponds to the absence of the component, while the intensity 1.0 corresponds to full saturation of the component.</p> <p><code>RGB = ntsc2rgb(YI Q)</code> converts the NTSC image <code>YI Q</code> to the equivalent truecolor image <code>RGB</code>.</p> <p><code>ntsc2rgb</code> computes the RGB values from the NTSC components using:</p> $\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.621 \\ 1.000 & -0.272 & -0.647 \\ 1.000 & -1.106 & 1.703 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$
Class Support	The input image or colormap must be of class <code>double</code> . The output is of class <code>double</code> .
See Also	<code>rgb2ntsc</code> , <code>rgb2ind</code> , <code>ind2rgb</code> , <code>ind2gray</code>

Purpose	Perform two-dimensional order-statistic filtering
Syntax	$Y = \text{ordfilt2}(X, \text{order}, \text{domain})$ $Y = \text{ordfilt2}(X, \text{order}, \text{domain}, S)$
Description	$Y = \text{ordfilt2}(X, \text{order}, \text{domain})$ replaces each element in X by the order-th element in the sorted set of neighbors specified by the nonzero elements in domain . $Y = \text{ordfilt2}(X, \text{order}, \text{domain}, S)$, where S is the same size as domain , uses the values of S corresponding to the nonzero values of domain as additive offsets.
Class Support	The class of X may be double or uint8. The class of Y is the same as the class of X , unless the additive offset form of ordfilt2 is used, in which case the class of Y is double.
Remarks	domain is equivalent to the structuring element used for binary image operations. It is a matrix containing only 1's and 0's; the 1's define the neighborhood for the filtering operation. For example, $Y = \text{ordfilt2}(X, 5, \text{ones}(3, 3))$ implements a 3-by-3 median filter; $Y = \text{ordfilt2}(X, 1, \text{ones}(3, 3))$ implements a 3-by-3 minimum filter; and $Y = \text{ordfilt2}(X, 9, \text{ones}(3, 3))$ implements a 3-by-3 maximum filter. $Y = \text{ordfilt2}(X, 1, [0 1 0; 1 0 1; 0 1 0])$ replaces each element in X by the minimum of its north, east, south, and west neighbors. The syntax that includes S (the matrix of additive offsets) can be used to implement grayscale morphological operations, including grayscale dilation and erosion.
See Also	medfilt2
Reference	Haralick, Robert M., and Linda G. Shapiro. <i>Computer and Robot Vision, Volume I</i> . Addison-Wesley, 1992.

phantom

Purpose	Generate a head phantom image
Syntax	<pre>P = phantom(def, n) P = phantom(E, n) [p, E] = phantom(...)</pre>
Description	<p><code>P = phantom(def, n)</code> generates an image of a head phantom that can be used to test the numerical accuracy of <code>radon</code> and <code>iRadon</code> or other two-dimensional reconstruction algorithms. <code>P</code> is a grayscale intensity image that consists of one large ellipse (representing the brain) containing several smaller ellipses (representing features in the brain).</p> <p><code>def</code> is a string that specifies the type of head phantom to generate. Valid values are:</p> <ul style="list-style-type: none">• 'Shepp-Logan' – a test image used widely by researchers in tomography.• 'Modif ied Shepp-Logan' (default) – a variant of the Shepp-Logan phantom in which the contrast is improved for better visual perception. <p><code>n</code> is a scalar that specifies the number of rows and columns in <code>P</code>. If you omit the argument, <code>n</code> defaults to 256.</p> <p><code>P = phantom(E, n)</code> generates a user-defined phantom, where each row of the matrix <code>E</code> specifies an ellipse in the image. <code>E</code> has 6 columns, with each column</p>

containing a different parameter for the ellipses. This table describes the columns of the matrix:

Column	Parameter	Meaning
Column 1	A	Additive intensity value of the ellipse
Column 2	a	Length of the horizontal semi-axis of the ellipse
Column 3	b	Length of the vertical semi-axis of the ellipse
Column 4	x0	x -coordinate of the center of the ellipse
Column 5	y0	y -coordinate of the center of the ellipse
Column 6	phi	Angle (in degrees) between the horizontal semi-axis of the ellipse and the x -axis of the image

For purposes of generating the phantom, the domains for the x - and y -axes span $[-1,1]$. Columns 2 through 5 must be specified in terms of this range.

`[P, E] = phantom(...)` returns the matrix E used to generate the phantom.

Class Support

All inputs must be of class `double`. All outputs are of class `double`.

Remarks

For any given pixel in the output image, the pixel's value is equal to the sum of the additive intensity values of all ellipses that the pixel is a part of. If a pixel is not part of any ellipse, its value is 0.

The additive intensity value A for an ellipse can be positive or negative; if it is negative, the ellipse will be darker than the surrounding pixels. Note that, depending on the values of A, some pixels may have values outside the range $[0,1]$.

phantom

Example

```
P = phantom('Modified Shepp-Logan', 200);  
imshow(P)
```



References

Jain, Anil K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. p. 439.

See Also

radon, i radon

Purpose	Display information about image pixels
Syntax	<code>pixval on</code> <code>pixval off</code> <code>pixval</code> <code>pixval (fig, option)</code>
Purpose	<code>pixval on</code> turns on interactive display of information about image pixels in the current figure. <code>pixval</code> installs a black bar at the bottom of the figure, which displays the row-column coordinates for whatever pixel the cursor is currently over, and the color information for that pixel. If the image is binary or intensity, the color information is a single intensity value. If the image is indexed or RGB, the color information is an RGB triplet. The values displayed are the actual data values, regardless of the class of the image array, or whether the data is in normal image range. If you click on the image and hold down the mouse button while you move the cursor, <code>pixval</code> also displays the Euclidean distance between the point you clicked on and the current cursor location. <code>pixval</code> draws a line between these points to indicate the distance being measured. When you release the mouse button, the line and the distance display disappear. You can move the display bar by clicking on it and dragging it to another place in the figure.
	<code>pixval off</code> turns interactive display off in the current figure. You can also turn off the display by clicking the button on the right side of the display bar. <code>pixval toggle</code> toggles interactive display on or off in the current figure.
	<code>pixval (fig, option)</code> applies the <code>pixval</code> command to the figure specified by <code>fig</code> . <code>option</code> is string containing 'on' or 'off'.
Remarks	<code>pixval</code> conflicts with certain other functions, such as <code>zoom</code> and <code>rotate3d</code> , that enable you to use a mouse to interact with the figure. Use only one of these functions at a time.
See Also	<code>impixel</code> , <code>improfile</code>

qtdecomp

Purpose	Perform quadtree decomposition
Syntax	<pre>S = qtdecomp(I) S = qtdecomp(I, threshol d) S = qtdecomp(I, threshol d, mi ndi m) S = qtdecomp(I, threshol d, [mi ndi m maxdi m])</pre> <pre>S = qtdecomp(I, fun) S = qtdecomp(I, fun, P1, P2, . . .)</pre>
Description	<p>qtdecomp divides a square image into four equal-sized square blocks, and then tests each block to see if it meets some criterion of homogeneity. If a block meets the criterion, it is not divided any further. If it does not meet the criterion, it is subdivided again into four blocks, and the test criterion is applied to those blocks. This process is repeated iteratively until each block meets the criterion. The result may have blocks of several different sizes.</p> <p><code>S = qtdecomp(I)</code> performs a quadtree decomposition on the intensity image <code>I</code>, and returns the quadtree structure in the sparse matrix <code>S</code>. If <code>S(k, m)</code> is nonzero, then <code>(k, m)</code> is the upper-left corner of a block in the decomposition, and the size of the block is given by <code>S(k, m)</code>. By default, <code>qtdecomp</code> splits a block unless all elements in the block are equal.</p> <p><code>S = qtdecomp(I, threshol d)</code> splits a block if the maximum value of the block elements minus the minimum value of the block elements is greater than <code>threshol d</code>. <code>threshol d</code> is specified as a value between 0 and 1, even if <code>I</code> is of class <code>uint8</code>. If <code>I</code> is <code>uint8</code>, the threshold value you supply is multiplied by 255 to determine the actual threshold to use.</p> <p><code>S = qtdecomp(I, threshol d, mi ndi m)</code> will not produce blocks smaller than <code>mi ndi m</code>, even if the resulting blocks do not meet the threshold condition.</p> <p><code>S = qtdecomp(I, threshol d, [mi ndi m maxdi m])</code> will not produce blocks smaller than <code>mi ndi m</code> or larger than <code>maxdi m</code>. Blocks larger than <code>maxdi m</code> are split even if they meet the threshold condition. <code>maxdi m/mi ndi m</code> must be a power of 2.</p> <p><code>S = qtdecomp(I, fun)</code> uses the function <code>fun</code> to determine whether to split a block. <code>qtdecomp</code> calls <code>fun</code> with all the current blocks of size <code>m</code>-by-<code>m</code> stacked into an <code>m</code>-by-<code>m</code>-by-<code>k</code> array, where <code>k</code> is the number of <code>m</code>-by-<code>m</code> blocks. <code>fun</code> should return a <code>k</code>-element vector, containing only 1's and 0's, where 1 indicates that the</p>

corresponding block should be split, and 0 indicates it should not be split. (For example, if $k(3)$ is 0, the third m -by- m block should not be split.) fun can be a string containing the name of a function, a string containing an expression, or an inline function.

$S = \text{qtdecomp}(I, \text{fun}, P1, P2, \dots)$ passes $P1, P2, \dots$, as additional arguments to fun .

Class Support

For the syntaxes that do not include a function, the input image can be of class `uint8` or `double`. For the syntaxes that include a function, the input image can be of any class supported by the function. The output matrix is always of class `sparse`.

Remarks

`qtdecomp` is appropriate primarily for square images whose dimensions are a power of 2, such as 128-by-128 or 512-by-512. These images can be divided until the blocks are as small as 1-by-1. If you use `qtdecomp` with an image whose dimensions are not a power of 2, at some point the blocks cannot be divided further. For example, if an image is 96-by-96, it can be divided into blocks of size 48-by-48, then 24-by-24, 12-by-12, 6-by-6, and finally 3-by-3. No further division beyond 3-by-3 is possible. To process this image, you must set `mindim` to 3 (or to 3 times a power of 2); if you are using the syntax that includes a function, the function must return 0 at the point when the block cannot be divided further.

qtdecomp

Example

```
I = [ 1     1     1     1     2     3     6     6
      1     1     2     1     4     5     6     8
      1     1     1     1    10    15     7     7
      1     1     1     1    20    25     7     7
     20    22    20    22     1     2     3     4
     20    22    22    20     5     6     7     8
     20    22    20    20     9    10    11    12
     22    22    20    20    13    14    15    16];
```

```
S = qtdecomp(I, 5);
```

```
full(S)
```

```
ans =
```

```
4     0     0     0     2     0     2     0
0     0     0     0     0     0     0     0
0     0     0     0     1     1     2     0
0     0     0     0     1     1     0     0
4     0     0     0     2     0     2     0
0     0     0     0     0     0     0     0
0     0     0     0     2     0     2     0
0     0     0     0     0     0     0     0
```

See Also

[qtgetblk](#), [qtsetblk](#)

Purpose	Get block values in quadtree decomposition
Syntax	$[val\ s, r, c] = qtgetblk(I, S, dim)$ $[val\ s, idx] = qtgetblk(I, S, dim)$
Description	$[val\ s, r, c] = qtgetblk(I, S, dim)$ returns in $val\ s$ an array containing the dim -by- dim blocks in the quadtree decomposition of I . S is the sparse matrix returned by $qtdecomp$; it contains the quadtree structure. $val\ s$ is a dim -by- dim -by- k array, where k is the number of dim -by- dim blocks in the quadtree decomposition; if there are no blocks of the specified size, all outputs are returned as empty matrices. r and c are vectors containing the row and column coordinates of the upper-left corners of the blocks. $[val\ s, idx] = qtgetblk(I, S, dim)$ returns in idx a vector containing the linear indices of the upper-left corners of the blocks.
Class Support	I can be of class <code>uint8</code> or <code>double</code> . S is of class <code>sparse</code> .
Remarks	The ordering of the blocks in $val\ s$ matches the columnwise order of the blocks in I . For example, if $val\ s$ is 4-by-4-by-2, $val\ s(:,:,1)$ contains the values from the first 4-by-4 block in I , and $val\ s(:,:,2)$ contains the values from the second 4-by-4 block.

qtgetblk

Example

This example continues the qtdecomp example.

```
[ val s, r, c] = qtgetblk(I, S, 4)
```

```
val s(:,:,1) =
```

1	1	1	1
1	1	2	1
1	1	1	1
1	1	1	1

```
val s(:,:,2) =
```

20	22	20	22
20	22	22	20
20	22	20	20
22	22	20	20

```
r =
```

1
5

```
c =
```

1
1

See Also

[qtdecomp](#), [qtsetblk](#)

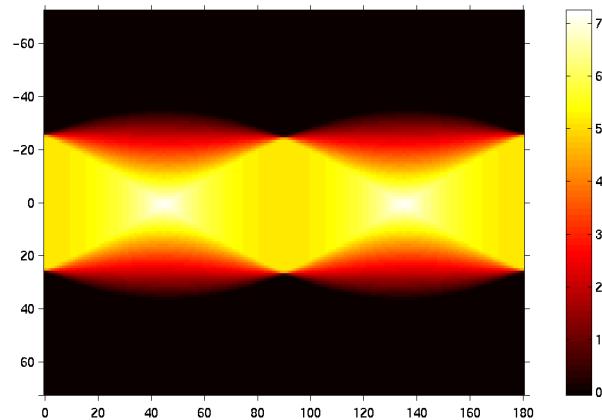
Purpose	Set block values in quadtree decomposition																																																																
Syntax	<code>J = qtsetblk(I, S, dim, vals)</code>																																																																
Description	<code>J = qtsetblk(I, S, dim, vals)</code> replaces each $dim \times dim$ block in the quadtree decomposition of <code>I</code> with the corresponding $dim \times dim$ block in <code>vals</code> . <code>S</code> is the sparse matrix returned by <code>qtdecomp</code> ; it contains the quadtree structure. <code>vals</code> is a $dim \times dim \times k$ array, where <code>k</code> is the number of $dim \times dim$ blocks in the quadtree decomposition.																																																																
Class Support	<code>I</code> can be of class <code>uint8</code> or <code>double</code> . <code>S</code> is of class <code>sparse</code> .																																																																
Remarks	The ordering of the blocks in <code>vals</code> must match the columnwise order of the blocks in <code>I</code> . For example, if <code>vals</code> is 4-by-4-by-2, <code>vals(:,:,1)</code> contains the values used to replace the first 4-by-4 block in <code>I</code> , and <code>vals(:,:,2)</code> contains the values for the second 4-by-4 block.																																																																
Example	<p>This example continues the <code>qtgetblk</code> example.</p> <pre>newvals = cat(3, zeros(4), ones(4)); J = qtsetblk(I, S, 4, newvals) J =</pre> <table style="margin-left: 200px; border-collapse: collapse;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>2</td><td>3</td><td>6</td><td>6</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>4</td><td>5</td><td>6</td><td>8</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>10</td><td>15</td><td>7</td><td>7</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>20</td><td>25</td><td>7</td><td>7</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>5</td><td>6</td><td>7</td><td>8</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>9</td><td>10</td><td>11</td><td>12</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>13</td><td>14</td><td>15</td><td>16</td></tr> </table>	0	0	0	0	2	3	6	6	0	0	0	0	4	5	6	8	0	0	0	0	10	15	7	7	0	0	0	0	20	25	7	7	1	1	1	1	1	2	3	4	1	1	1	1	5	6	7	8	1	1	1	1	9	10	11	12	1	1	1	1	13	14	15	16
0	0	0	0	2	3	6	6																																																										
0	0	0	0	4	5	6	8																																																										
0	0	0	0	10	15	7	7																																																										
0	0	0	0	20	25	7	7																																																										
1	1	1	1	1	2	3	4																																																										
1	1	1	1	5	6	7	8																																																										
1	1	1	1	9	10	11	12																																																										
1	1	1	1	13	14	15	16																																																										
See Also	<code>qtdecomp</code> , <code>qtgetblk</code>																																																																

radon

Purpose	Compute Radon transform
Syntax	<pre>R = radon(I, theta) R = radon(I, theta, n) [R, xp] = radon(...)</pre>
Description	<p>The <code>radon</code> function computes the Radon transform, which is the projection of the image intensity along a radial line oriented at a specified angle.</p> <p><code>R = radon(I, theta)</code> returns the Radon transform of the intensity image <code>I</code> for the angle <code>theta</code> degrees. If <code>theta</code> is a scalar, the result <code>R</code> is a column vector containing the Radon transform for <code>theta</code> degrees. If <code>theta</code> is a vector, then <code>R</code> is a matrix in which each column is the Radon transform for one of the angles in <code>theta</code>. If you omit <code>theta</code>, it defaults to 0:179.</p> <p><code>R = radon(I, theta, n)</code> returns a Radon transform with the projection computed at <code>n</code> points. <code>R</code> has <code>n</code> rows. If you do not specify <code>n</code>, the number of points at which the projection is computed is:</p> <pre>2 * ceil(norm(size(I) - floor((size(I)-1)/2))) + 3</pre> <p>This number is sufficient to compute the projection at unit intervals, even along the diagonal.</p> <p><code>[R, xp] = radon(...)</code> returns a vector <code>xp</code> containing the radial coordinates corresponding to each row of <code>R</code>.</p>
Class Support	<code>I</code> can be of class <code>uint8</code> or <code>double</code> . All other inputs and outputs are of class <code>double</code> .
Remarks	<p>The radial coordinates returned in <code>xp</code> are the values along the x'-axis, which is oriented at <code>theta</code> degrees counterclockwise from the x-axis. The origin of both axes is the center pixel of the image, which is defined as:</p> <pre>floor((size(I)+1)/2)</pre> <p>For example, in a 20-by-30 image, the center pixel is (10,15).</p>

Example

```
iptsetpref('ImshowAxesVisible', 'on')
I = zeros(100, 100);
I(25:75, 25:75) = 1;
theta = 0:180;
[R, xp] = radon(I, theta);
imshow(theta, xp, R, []), colormap(hot), colorbar
```

**See Also**

[i](#) [radon](#), [phantom](#)

References

Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 42-45.

Bracewell, Ronald N. *Two-Dimensional Imaging*. Englewood Cliffs, NJ: Prentice Hall, 1995. pp. 505-537.

rgb2gray

Purpose	Convert an RGB image or colormap to grayscale
Syntax	<pre>I = rgb2gray(RGB) newmap = rgb2gray(map)</pre>
Description	<p><code>rgb2gray</code> converts RGB images to grayscale by eliminating the hue and saturation information while retaining the luminance.</p> <p><code>I = rgb2gray(RGB)</code> converts the truecolor image <code>RGB</code> to the grayscale intensity image <code>I</code>.</p> <p><code>newmap = rgb2gray(map)</code> returns a grayscale colormap equivalent to <code>map</code>.</p>
Class Support	If the input is an RGB image, it can be of class <code>uint8</code> or <code>double</code> ; the output image <code>I</code> is of the same class as the input image. If the input is a colormap, the input and output colormaps are both of class <code>double</code> .
Algorithm	<code>rgb2gray</code> converts the RGB values to NTSC coordinates, sets the hue and saturation components to zero, and then converts back to RGB color space.
See Also	<code>ind2gray</code> , <code>ntsc2rgb</code> , <code>rgb2ind</code> , <code>rgb2ntsc</code>

Purpose	Convert RGB values to hue-saturation-value (HSV) color space
Syntax	<pre>hsvmap = rgb2hsv(rgbmap) HSV = rgb2hsv(RGB)</pre>
Description	<p><code>hsvmap = rgb2hsv(rgbmap)</code> converts the m-by-3 RGB values in RGB to HSV color space. <code>hsvmap</code> is an m-by-3 matrix that contains the hue, saturation, and value components as columns that are equivalent to the colors in the RGB colormap. Both <code>rgbmap</code> and <code>hsvmap</code> are of class <code>double</code> and contain values in the range 0 to 1.0.</p> <p><code>HSV = rgb2hsv(RGB)</code> converts the truecolor image <code>RGB</code> to the equivalent HSV image <code>HSV</code>.</p>
Class Support	If the input is an RGB image, it can be of class <code>uint8</code> or <code>double</code> ; the output image is of class <code>double</code> . If the input is a colormap, the input and output colormaps are both of class <code>double</code> .
Remarks	<code>rgb2hsv</code> is a function in MATLAB.
See Also	<code>hsv2rgb</code> , <code>rgbplot</code> <code>colormap</code> in the online MATLAB Function Reference

rgb2ind

Purpose	Convert an RGB image to an indexed image
Syntax	<pre>[X, map] = rgb2ind(RGB) [X, map] = rgb2ind(RGB, tol) [X, map] = rgb2ind(RGB, n) X = rgb2ind(RGB, map) [...] = rgb2ind(..., dither_option)</pre>
Description	<p><code>rgb2ind</code> converts RGB images to indexed images using one of four different methods: direct translation, uniform quantization, minimum variance quantization, and colormap mapping. For all methods except direct translation, <code>rgb2ind</code> dithers the image unless you specify '<code>'nodither'</code>' for <code>dither_option</code>.</p> <p><code>[X, map] = rgb2ind(RGB)</code> converts the RGB image in the array <code>RGB</code> to an indexed image <code>X</code> with colormap <code>map</code> using direct translation. The resulting colormap may be very long, as it has one entry for each pixel in <code>RGB</code>. Do not set <code>dither_option</code> if you use this method.</p> <p><code>[X, map] = rgb2ind(RGB, tol)</code> converts the RGB image to an indexed image <code>X</code> using uniform quantization. <code>map</code> contains at most $(\text{floor}(1/tol) + 1)^3$ colors. <code>tol</code> must be between 0 and 1.0.</p> <p><code>[X, map] = rgb2ind(RGB, n)</code> converts the RGB image to an indexed image <code>X</code> using minimum variance quantization. <code>map</code> contains at most <code>n</code> colors.</p> <p><code>X = rgb2ind(RGB, map)</code> converts the RGB image to an indexed image <code>X</code> with colormap <code>map</code> by matching colors in <code>RGB</code> with the nearest color in the colormap <code>map</code>.</p> <p><code>[...] = rgb2ind(..., dither_option)</code> enables or disables dithering. <code>dither_option</code> is a string that can have one of these values:</p> <ul style="list-style-type: none">• '<code>'dither'</code>' (default) dithers, if necessary, to achieve better color resolution at the expense of spatial resolution.• '<code>'nodither'</code>' maps each color in the original image to the closest color in the new map. No dithering is performed.
Class Support	The input image can be of class <code>uint8</code> or <code>double</code> . The output image is of class <code>uint8</code> if the length of <code>map</code> is less than or equal to 256, or <code>double</code> otherwise.

Remarks

If you specify `tol`, `rgb2ind` uses uniform quantization to convert the image. This method involves cutting the RGB color cube into smaller cubes of length `tol`. For example, if you specify a `tol` of 0.1, the edges of the cubes are one-tenth the length of the RGB cube. The total number of small cubes is:

$$n = (\text{floor}(1/tol) + 1)^3$$

Each cube represents a single color in the output image. Therefore, the maximum length of the colormap is `n`. `rgb2ind` removes any colors that don't appear in the input image, so the actual colormap may be much smaller than `n`.

If you specify `n`, `rgb2ind` uses minimum variance quantization. This method involves cutting the RGB color cube into smaller boxes (not necessarily cubes) of different sizes, depending on how the colors are distributed in the image. If the input image actually uses fewer colors than the number you specify, the output colormap is also smaller.

If you specify `map`, `rgb2ind` uses colormap mapping, which involves finding the colors in `map` that best match the colors in the RGB image.

Example

```
RGB = imread('flowers.tif');
[X, map] = rgb2ind(RGB, 128);
imshow(X, map)
```

**See Also**

`cmuniquedither`, `imapprox`, `ind2rgb`, `rgb2gray`

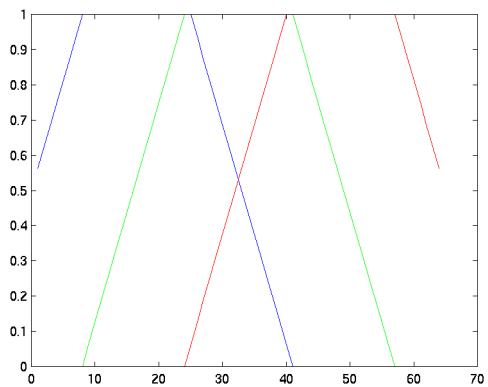
rgb2ntsc

Purpose	Convert RGB values to NTSC color space
Syntax	$YI\ Qmap = \text{rgb2ntsc}(rgbmap)$ $YI\ Q = \text{rgb2ntsc}(RGB)$
Description	$YI\ Qmap = \text{rgb2ntsc}(rgbmap)$ converts the m -by-3 RGB values in $rgbmap$ to NTSC color space. $YI\ Qmap$ is an m -by-3 matrix that contains the NTSC luminance (Y) and chrominance (I and Q) color components as columns that are equivalent to the colors in the RGB colormap. $YI\ Q = \text{rgb2ntsc}(RGB)$ converts the truecolor image RGB to the equivalent NTSC image $YI\ Q$.
	<code>rgb2ntsc</code> defines the NTSC components using:
	$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$
Class Support	If the input is an RGB image, it can be of class <code>uint8</code> or <code>double</code> ; the output image is of class <code>double</code> . If the input is a colormap, the input and output colormaps are both of class <code>double</code> .
Remarks	In the NTSC color space, the luminance is the grayscale signal used to display pictures on monochrome (black and white) televisions. The other components carry the hue and saturation information.
See Also	<code>ntsc2rgb</code> , <code>rgb2ind</code> , <code>ind2rgb</code> , <code>ind2gray</code>

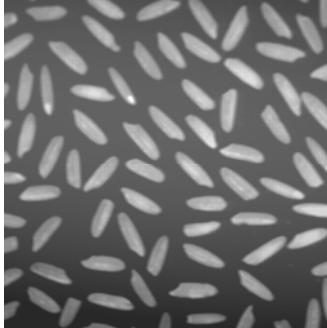
Purpose	Convert RGB values to YCbCr color space
Syntax	<code>ycbcrmap = rgb2ycbcr(rgbmap)</code> <code>YCBCR = rgb2ycbcr(RGB)</code>
Description	<code>ycbcrmap = rgb2ycbcr(rgbmap)</code> converts the RGB values in <code>rgbmap</code> to the YCbCr color space. <code>ycbcrmap</code> is an m -by-3 matrix that contains the YCbCr luminance (Y) and chrominance (Cb and Cr) color components as columns. Each row represents the equivalent color to the corresponding row in the RGB colormap. <code>YCBCR = rgb2ycbcr(RGB)</code> converts the truecolor image <code>RGB</code> to the equivalent image in the YCbCr color space.
Class Support	If the input is an RGB image, it can be of class <code>uint8</code> or <code>double</code> ; the output image is of the same class as the input image. If the input is a colormap, the input and output colormaps are both of class <code>double</code> .
See Also	<code>ntsc2rgb</code> , <code>rgb2ntsc</code> , <code>ycbcr2rgb</code>

rgbplot

Purpose	Plot colormap
Syntax	<code>rgbplot(map)</code>
Description	<code>rgbplot(map)</code> plots the three columns of <code>map</code> , where <code>map</code> is an m -by-3 colormap matrix. <code>rgbplot</code> draws the first column in red, the second in green, and the third in blue.
Example	<code>rgbplot(jet)</code>



See Also [colormap](#) in the online MATLAB Function Reference

Purpose	Select region of interest, based on color
Syntax	$BW = \text{roi col or}(A, \text{low}, \text{high})$ $BW = \text{roi col or}(A, v)$
Description	<code>roi col or</code> selects a region of interest within an indexed or intensity image and returns a binary image. (You can use the returned image as a mask for masked filtering using <code>roi filt2</code> .)
	$BW = \text{roi col or}(A, \text{low}, \text{high})$ returns a region of interest selected as those pixels that lie within the colormap range $[\text{low} \text{ } \text{high}]$:
	$BW = (A \geq \text{low}) \& (A \leq \text{high})$
	BW is a binary image with 0's outside the region of interest and 1's inside.
	$BW = \text{roi col or}(A, v)$ returns a region of interest selected as those pixels in A that match the values in vector v . BW is a binary image with 1's where the values of A match the values of v .
Class Support	The input image A can be of class <code>uint8</code> or <code>double</code> . The output image BW is of class <code>uint8</code> .
Example	<pre>I = imread('rice.tif'); BW = roi col or(I, 128, 255); imshow(I); figure, imshow(BW)</pre>  
See Also	<code>roi filt2</code> , <code>roi poly</code>

roi fill

Purpose	Smoothly interpolate within an arbitrary image region
Syntax	<pre>J = roi fill(I, c, r) J = roi fill(I) J = roi fill(I, BW) [J, BW] = roi fill(...)</pre>
	<pre>J = roi fill(x, y, I, xi, yi) [x, y, J, BW, xi, yi] = roi fill(...)</pre>
Description	<p><code>roi fill</code> fills in a specified polygon in an intensity image. It smoothly interpolates inward from the pixel values on the boundary of the polygon by solving Laplace's equation. <code>roi fill</code> can be used, for example, to "erase" small objects in an image.</p> <p><code>J = roi fill(I, c, r)</code> fills in the polygon specified by <code>c</code> and <code>r</code>, which are equal-length vectors containing the row-column coordinates of the pixels on vertices of the polygon. The k-th vertex is the pixel $(r(k), c(k))$.</p> <p><code>J = roi fill(I)</code> displays the image <code>I</code> on the screen and lets you specify the polygon using the mouse. If you omit <code>I</code>, <code>roi fill</code> operates on the image in the current axes. Use normal button clicks to add vertices to the polygon. Pressing Backspace or Delete removes the previously selected vertex. A shift-click, right-click, or double-click adds a final vertex to the selection and then starts the fill; pressing Return finishes the selection without adding a vertex.</p> <p><code>J = roi fill(I, BW)</code> uses <code>BW</code> (a binary image the same size as <code>I</code>) as a mask. <code>roi fill</code> fills in the regions in <code>I</code> corresponding to the nonzero pixels in <code>BW</code>. If there are multiple regions, <code>roi fill</code> performs the interpolation on each region independently.</p> <p><code>[J, BW] = roi fill(...)</code> returns the binary mask used to determine which pixels in <code>I</code> get filled. <code>BW</code> is a binary image the same size as <code>I</code> with 1's for pixels corresponding to the interpolated region of <code>I</code> and 0's elsewhere.</p> <p><code>J = roi fill(x, y, I, xi, yi)</code> uses the vectors <code>x</code> and <code>y</code> to establish a nondefault spatial coordinate system. <code>xi</code> and <code>yi</code> are equal-length vectors that specify polygon vertices as locations in this coordinate system.</p>

[x, y, J, BW, xi, yi] = roi fill(. . .) returns the XData and YData in x and y; the output image in J; the mask image in BW; and the polygon coordinates in xi and yi. xi and yi are empty if the roi fill(I, BW) form is used.

If roi fill is called with no output arguments, the resulting image is displayed in a new figure.

Class Support

The input image I and the binary mask BW can be of class double or uint8. The output image J is of the same class as I. All other inputs and outputs are of class double.

Example

```
I = imread('eighth.tif');
c = [222 272 300 270 221 194];
r = [21 21 75 121 121 75];
J = roi fill(I, c, r);
imshow(I)
figure, imshow(J)
```



See Also

roi fill2, roi poly

roifilt2

Purpose	Filter a region of interest
Syntax	<pre>J = roifilt2(h, I, BW) J = roifilt2(I, BW, fun) J = roifilt2(I, BW, fun, P1, P2, ...)</pre>
Description	<p><code>J = roifilt2(h, I, BW)</code> filters the data in <code>I</code> with the two-dimensional linear filter <code>h</code>. <code>BW</code> is a binary image the same size as <code>I</code> that is used as a mask for filtering. <code>roifilt2</code> returns an image that consists of filtered values for pixels in locations where <code>BW</code> contains 1's, and unfiltered values for pixels in locations where <code>BW</code> contains 0's. For this syntax, <code>roifilt2</code> calls <code>filter2</code> to implement the filter.</p> <p><code>J = roifilt2(I, BW, fun)</code> processes the data in <code>I</code> using the function <code>fun</code>. The result <code>J</code> contains computed values for pixels in locations where <code>BW</code> contains 1's, and the actual values in <code>I</code> for pixels in locations where <code>BW</code> contains 0's.</p> <p><code>fun</code> can be a string containing the name of a function, a string containing an expression, or an inline function object. <code>fun</code> should take a matrix as a single argument and return a matrix of the same size:</p> <pre>y = fun(x)</pre> <p><code>J = roifilt2(I, BW, fun, P1, P2, ...)</code> passes the additional parameters <code>P1, P2, ...,</code> to <code>fun</code>.</p>
Class Support	For the syntax that includes a filter <code>h</code> , the input image <code>I</code> can be of class <code>double</code> or <code>uint8</code> , and the output array <code>J</code> is of class <code>double</code> . For the syntax that includes a function, <code>I</code> can be of any class supported by <code>fun</code> , and the class of <code>J</code> depends on the class of the output from <code>fun</code> .

Example

This example continues the roi poly example.

```
h = fspecial('unsharp');  
J = roifilt2(h, I, BW);  
imshow(J)
```

**See Also**

[filter2](#), [roi poly](#)

roi poly

Purpose	Select a polygonal region of interest
Syntax	$BW = \text{roi poly}(I, c, r)$ $BW = \text{roi poly}(I)$ $BW = \text{roi poly}(x, y, I, xi, yi)$ $[BW, xi, yi] = \text{roi poly}(\dots)$ $[x, y, BW, xi, yi] = \text{roi poly}(\dots)$
Description	<p>Use <code>roi poly</code> to select a polygonal region of interest within an image. <code>roi poly</code> returns a binary image that you can use as a mask for masked filtering.</p> <p><code>BW = roi poly(I, c, r)</code> returns the region of interest selected by the polygon described by vectors <code>c</code> and <code>r</code>. <code>BW</code> is a binary image the same size as <code>I</code> with 0's outside the region of interest and 1's inside.</p> <p><code>BW = roi poly(I)</code> displays the image <code>I</code> on the screen and lets you specify the polygon using the mouse. If you omit <code>I</code>, <code>roi poly</code> operates on the image in the current axes. Use normal button clicks to add vertices to the polygon. Pressing Backspace or Delete removes the previously selected vertex. A shift-click, right-click, or double-click adds a final vertex to the selection and then starts the fill; pressing Return finishes the selection without adding a vertex.</p> <p><code>BW = roi poly(x, y, I, xi, yi)</code> uses the vectors <code>x</code> and <code>y</code> to establish a nondefault spatial coordinate system. <code>xi</code> and <code>yi</code> are equal-length vectors that specify polygon vertices as locations in this coordinate system.</p> <p><code>[BW, xi, yi] = roi poly(\dots)</code> returns the polygon coordinates in <code>xi</code> and <code>yi</code>. Note that <code>roi poly</code> always produces a closed polygon. If the points specified describe a closed polygon (i.e., if the last pair of coordinates is identical to the first pair), the length of <code>xi</code> and <code>yi</code> is equal to the number of points specified. If the points specified do not describe a closed polygon, <code>roi poly</code> adds a final point having the same coordinates as the first point. (In this case the length of <code>xi</code> and <code>yi</code> is one greater than the number of points specified.)</p> <p><code>[x, y, BW, xi, yi] = roi poly(\dots)</code> returns the XData and YData in <code>x</code> and <code>y</code>; the mask image in <code>BW</code>; and the polygon coordinates in <code>xi</code> and <code>yi</code>.</p> <p>If <code>roi poly</code> is called with no output arguments, the resulting image is displayed in a new figure.</p>

Class Support The input image I can be of class double or uint8. The output image BW is of class uint8. All other inputs and outputs are of class double.

Remarks For any of the roi poly syntaxes, you can replace the input image I with two arguments, m and n, that specify the row and column dimensions of an arbitrary image. For example, these commands create a 100-by-200 binary mask:

```
c = [112 112 79 79];
r = [37 66 66 37];
BW = roi poly(100, 200, c, r);
```

If you specify m and n with an interactive form of roi poly, an m-by-n black image is displayed, and you use the mouse to specify a polygon within this image.

Example

```
I = imread('eighth.tif');
c = [222 272 300 270 221 194];
r = [21 21 75 121 121 75];
BW = roi poly(I, c, r);
imshow(I)
figure, imshow(BW)
```



See Also [roi filter](#), [roi color](#), [roi fill](#)

std2

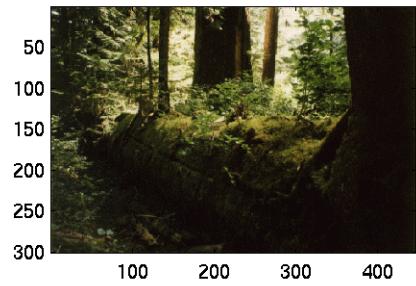
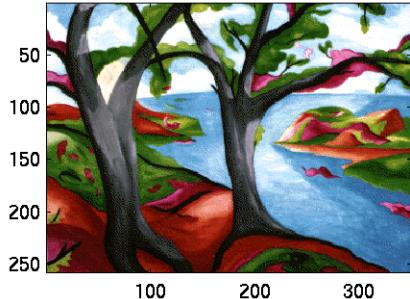
Purpose	Compute the standard deviation of the elements of a matrix
Syntax	<code>b = std2(A)</code>
Description	<code>b = std2(A)</code> computes the standard deviation of the values in <code>A</code> .
Class Support	<code>A</code> is an array of class <code>uint8</code> or <code>double</code> . <code>b</code> is a scalar of class <code>double</code> .
Algorithm	<code>std2</code> computes the standard deviation of the array <code>A</code> using <code>std(A(:))</code> .
See Also	<code>corr2</code> , <code>mean2</code> <code>std</code> , <code>mean</code> in the online MATLAB Function Reference

Purpose	Display multiple images in the same figure
Syntax	<code>subi mage(X, map)</code> <code>subi mage(I)</code> <code>subi mage(BW)</code> <code>subi mage(RGB)</code> <code>subi mage(x, y, . . .)</code> <code>h = subi mage(. . .)</code>
Description	You can use <code>subi mage</code> in conjunction with <code>subplot</code> to create figures with multiple images, even if the images have different colormaps. <code>subi mage</code> works by converting images to truecolor for display purposes, thus avoiding colormap conflicts. <code>subi mage(X, map)</code> displays the indexed image <code>X</code> with colormap <code>map</code> in the current axes. <code>subi mage(I)</code> displays the intensity image <code>I</code> in the current axes. <code>subi mage(BW)</code> displays the binary image <code>BW</code> in the current axes. <code>subi mage(RGB)</code> displays the truecolor image <code>RGB</code> in the current axes. <code>subi mage(x, y, . . .)</code> displays an image using a nondefault spatial coordinate system. <code>h = subi mage(. . .)</code> returns a handle to an image object.
Class Support	The input image can be of class <code>uint8</code> or <code>double</code> .

subimage

Example

```
load trees  
[X2, map2] = imread('forest.tif');  
subplot(1, 2, 1), subimage(X, map)  
subplot(1, 2, 2), subimage(X2, map2)
```



See Also

[imshow](#)

[subplot](#) in the online MATLAB Function Reference

Purpose	Adjust display size of an image
Syntax	<code>truesize(fig, [mrows ncols])</code> <code>truesize(fig)</code>
Description	<code>truesize(fig, [mrows ncols])</code> adjusts the display size of an image. <code>fig</code> is a figure containing a single image or a single image with a colorbar. <code>[mrows ncols]</code> is a 1-by-2 vector that specifies the requested screen area in pixels that the image should occupy. <code>truesize(fig)</code> uses the image height and width for <code>[mrows ncols]</code> . This results in the display having one screen pixel for each image pixel. If you omit the figure argument, <code>truesize</code> works on the current figure.
Remarks	If the 'TruesizeWarning' toolbox preference is 'on', <code>truesize</code> displays a warning if the image is too large to fit on the screen. (The entire image is still displayed, but at less than true size.) If 'TruesizeWarning' is 'off', <code>truesize</code> does not display the warning. Note that this preference applies even when you call <code>truesize</code> indirectly, such as through <code>imshow</code> .
See Also	<code>imshow</code> , <code>iptsetpref</code> , <code>iptgetpref</code>

uint8

Purpose	Convert data to unsigned 8-bit integers
Syntax	<code>B = uint8(A)</code>
Description	<code>B = uint8(A)</code> creates the unsigned 8-bit integer array <code>B</code> from the array <code>A</code> . If <code>A</code> is a <code>uint8</code> array, <code>B</code> is identical to <code>A</code> . The elements of a <code>uint8</code> array can range from 0 to 255. For any value in <code>A</code> outside this range, the resulting value in <code>B</code> is not defined (and may vary from platform to platform). The fractional part of each value in <code>A</code> is discarded on conversion. This means, for example, that <code>uint8(102.99)</code> is 102, not 103. Therefore, it is often a good idea to round off the values in <code>A</code> before converting to <code>uint8</code> . For example:
	<code>B = uint8(round(A))</code>
	MATLAB supports these operations on <code>uint8</code> arrays: <ul style="list-style-type: none">• Displaying data values• Indexing into arrays using standard MATLAB subscripting• Reshaping, reordering, and concatenating arrays, using functions such as <code>reshape</code>, <code>cat</code>, and <code>permute</code>• Saving to and loading from MAT-files• The <code>all</code> and <code>any</code> functions• Logical operators and indexing• Relational operators
	MATLAB also supports the <code>find</code> function for <code>uint8</code> arrays, but the returned array is of class <code>double</code> .
	Most of the functions in the Image Processing Toolbox accept <code>uint8</code> input. See the individual Reference entries for information about <code>uint8</code> support.
Remarks	<code>uint8</code> is a MATLAB built-in function.
See Also	<code>double</code> , <code>im2double</code> , <code>im2uint8</code>

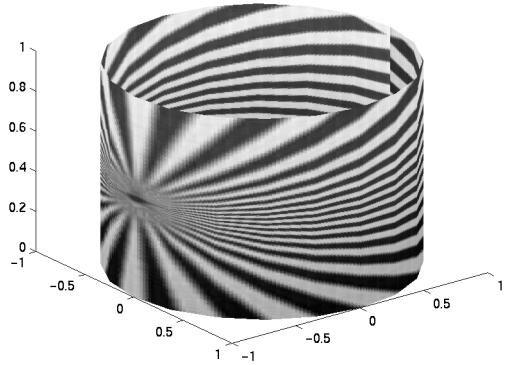
Purpose	Display an image as a texture-mapped surface
Syntax	<code>warp(X, map)</code> <code>warp(I, n)</code> <code>warp(BW)</code> <code>warp(RGB)</code> <code>warp(z, ...)</code> <code>warp(x, y, z, ...)</code> <code>h = warp(...)</code>
Description	<p><code>warp(X, map)</code> displays the indexed image <code>X</code> with colormap <code>map</code> as a texture map on a simple rectangular surface.</p> <p><code>warp(I, n)</code> displays the intensity image <code>I</code> with gray scale colormap of length <code>n</code> as a texture map on a simple rectangular surface.</p> <p><code>warp(BW)</code> displays the binary image <code>BW</code> as a texture map on a simple rectangular surface.</p> <p><code>warp(RGB)</code> displays the RGB image in the array <code>RGB</code> as a texture map on a simple rectangular surface.</p> <p><code>warp(z, ...)</code> displays the image on the surface <code>z</code>.</p> <p><code>warp(x, y, z, ...)</code> displays the image on the surface <code>(x, y, z)</code>.</p> <p><code>h = warp(...)</code> returns a handle to a texture mapped surface.</p>
Class Support	The input image can be of class <code>uint8</code> or <code>double</code> .
Remarks	Texture-mapped surfaces render more slowly than images.

warp

Example

This example texture maps an image of a test pattern onto a cylinder:

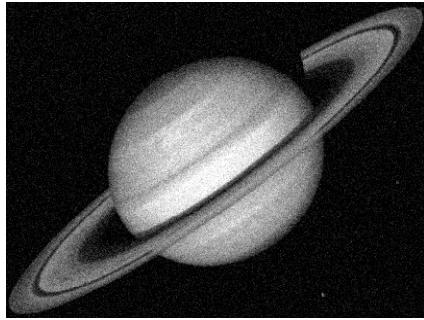
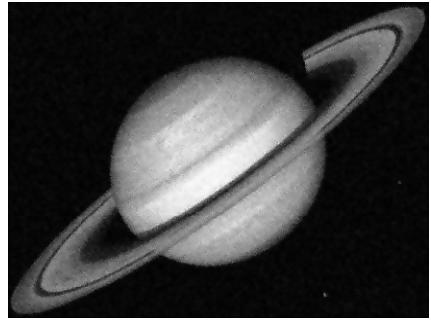
```
[x, y, z] = cylinder;
I = imread('testpat1.tif');
warp(x, y, z, I);
```



See Also

[imshow](#)

[image](#), [imagesc](#), [surf](#) in the online MATLAB Function Reference

Purpose	Perform two-dimensional adaptive noise-removal filtering
Syntax	<code>J = wiener2(I, [m n], noise)</code> <code>[J, noise] = wiener2(I, [m n])</code>
Description	wiener2 lowpass filters an intensity image that has been degraded by constant power additive noise. wiener2 uses a pixel-wise adaptive Wiener method based on statistics estimated from a local neighborhood of each pixel.
	<code>J = wiener2(I, [m n], noise)</code> filters the image I using pixel-wise adaptive Wiener filtering, using neighborhoods of size m-by-n to estimate the local image mean and standard deviation. If you omit the [m n] argument, m and n default to 3. The additive noise (Gaussian white noise) power is assumed to be noise.
	<code>[J, noise] = wiener2(I, [m n])</code> also estimates the additive noise power before doing the filtering. wiener2 returns this estimate in noise.
Class Support	The input image I can be of class uint8 or double. The output image J is of the same class as I.
Example	Degraded and restored images of the planet Saturn.
	<pre>I = imread('saturn.tif'); J = imnoise(I, 'gaussian', 0, 0.005); K = wiener2(J, [5 5]); imshow(J) figure, imshow(K)</pre>  
Algorithm	wiener2 estimates the local mean and variance around each pixel

wiener2

$$\mu = \frac{1}{NM} \sum_{n_1, n_2 \in \eta} a(n_1, n_2)$$
$$\sigma^2 = \frac{1}{NM} \sum_{n_1, n_2 \in \eta} a^2(n_1, n_2) - \mu^2$$

where η is the N -by- M local neighborhood of each pixel in the image A . `wiener2` then creates a pixel-wise Wiener filter using these estimates

$$b(n_1, n_2) = \mu + \frac{\sigma^2 - v^2}{\sigma^2} (a(n_1, n_2) - \mu)$$

where v^2 is the noise variance. If the noise variance is not given, `wiener2` uses the average of all the local estimated variances.

See Also

`filter2`, `medfilt2`

Reference

Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 536-540.

Purpose	Convert YCbCr values to RGB color space
Syntax	<code>rgbmap = ycbcr2rgb(ycbcrmap)</code> <code>RGB = ycbcr2rgb(YCBCR)</code>
Description	<code>rgbmap = ycbcr2rgb(ycbcrmap)</code> converts the YCbCr values in the colormap <code>ycbcrmap</code> to the RGB color space. If <code>ycbcrmap</code> is m -by-3 and contains the YCbCr luminance (Y) and chrominance (Cb and Cr) color components as its columns, then <code>rgbmap</code> is returned as an m -by-3 matrix that contains the red, green, and blue values equivalent to those colors. <code>RGB = ycbcr2rgb(YCBCR)</code> converts the YCbCr image <code>YCBCR</code> to the equivalent truecolor image <code>RGB</code> .
Class Support	If the input is a YCbCr image, it can be of class <code>uint8</code> or <code>double</code> ; the output image is of the same class as the input image. If the input is a colormap, the input and output colormaps are both of class <code>double</code> .
See Also	<code>ntsc2rgb</code> , <code>rgb2ntsc</code> , <code>rgb2ycbcr</code>

zoom

Purpose	Zoom in and out of an image
Syntax	<code>zoom on</code> <code>zoom off</code> <code>zoom out</code> <code>zoom reset</code> <code>zoom</code> <code>zoom xon</code> <code>zoom yon</code> <code>zoom(factor)</code> <code>zoom(fig, option)</code>
Description	<p><code>zoom on</code> turns on interactive zooming for the current figure. When zooming is enabled, clicking the mouse on a point within an axes changes the axes limits by a factor of 2, to either zoom in or out from the point:</p> <ul style="list-style-type: none">• For a single-button mouse, zoom in by clicking the mouse button and zoom out by shift-clicking.• For a two- or three-button mouse, zoom in by clicking the left mouse button and zoom out by clicking the right mouse button. <p>Clicking and dragging over an axes when interactive zooming is enabled draws a rubber-band box. When the mouse button is released, the axes zoom in to the region enclosed by the rubber-band box.</p> <p>Double-clicking within an axes returns the axes to its initial zoom setting.</p> <p><code>zoom out</code> returns the plot to its initial zoom setting.</p> <p><code>zoom reset</code> remembers the current <code>zoom</code> setting as the initial <code>zoom</code> setting. Later calls to <code>zoom out</code>, or double-clicks when interactive zoom mode is enabled, return to this zoom level.</p> <p><code>zoom</code> toggles the interactive zoom status.</p> <p><code>zoom xon</code> and <code>zoom yon</code> sets <code>zoom on</code> for the <i>x</i>- and <i>y</i>-axis, respectively.</p> <p><code>zoom(factor)</code> zooms in by the specified factor, without affecting the interactive zoom mode. By default, factor is 2. A factor between 0 and 1 specifies zooming out by $1/\text{factor}$.</p>

`zoom(fig, option)` applies the `zoom` command to the figure specified by `fig`. `option` is a string containing any of the above arguments. If you do not specify a figure, `zoom` works on the current figure.

See Also

`imcrop`

zoom

A

adaptive filtering 7-23, 11-195
aliasing 3-5
analyzing images
 contour plots 7-7
 edge detection 7-10, 11-52
 histograms 7-8, 11-117
 intensity profiles 7-4, 11-123
 pixel values 7-3, 11-121
 quadtree decomposition 7-11, 11-166
 summary statistics 7-9
anti-aliasing 3-5, 11-129
applylut 8-20, 11-10
area of binary images 8-18, 11-16
arrays
 logical 1-7, 1-15
 storing images 1-3
averaging filter 5-10, 11-70

B

bestblk 11-12
bicubic interpolation 3-3
bilinear interpolation 3-3
binary image operations 8-2
 connected-components labeling 8-15, 11-23
 feature measurement 8-18
 flood fill 8-13, 11-20
 lookup-table operations 8-20, 11-10, 11-151
 morphological operations 8-4, 11-25
 neighborhoods 8-2, 11-10
 object-based operations 8-10
 padding borders 8-2
binary images 1-7, 11-147
 4-connected neighborhoods 8-10
 8-connected neighborhoods 8-10
 converting from other types 11-96

displaying 2-6, 8-3
Euler number 8-19, 11-18
image area 8-18, 11-16
object selection 8-16, 11-30
perimeter determination 8-12, 11-29
processing 8-2
binary masks 9-3
blkproc 4-8, 11-13
block processing 4-2
 block size 11-12
 column processing 4-11
 distinct blocks 4-8, 11-13
 padding borders 4-5
 sliding neighborhoods 4-4, 11-159
BMP files 1-11, 11-114, 11-126, 11-135
borders
 padding 4-5, 5-5, 8-2
brighten 11-15
bwarea 8-18, 11-16
bweuler 8-19, 11-18
bwfill 8-13, 11-20
bwlabel 8-15, 11-23
bwmorph 8-8, 11-25
bwperim 8-12, 11-29
bwslect 8-16, 11-30

C

Canny edge detector 11-52
center pixel
 linear filtering 5-4
 morphological operations 8-4
closure 8-7, 11-25
cmpermute 11-32
cmunique 11-33
col2im 11-34

- colfilt 4-11, 11-35
 color 10-2
 approximation 10-6, 11-49, 11-102, 11-177
 dithering 10-8, 11-49
 quantization 10-6, 11-177
 reducing number of colors 10-5
 color depth 10-3
 color spaces
 converting between 1-13, 10-9, 11-92, 11-160, 11-175, 11-178, 11-179, 11-197
 HSV 10-10, 11-92
 NTSC 10-9, 11-160, 11-178
 RGB 10-9
 YCbCr 10-10, 11-179, 11-197
 colobar 2-10, 11-37
 colormap mapping 10-7
 colormaps
 brightening 11-15
 darkening 11-15
 plotting RGB values 11-180
 rearranging colors 11-32
 reducing number of colors 10-7
 removing duplicate entries 11-33
 RGB components 1-5
 column processing 4-11, 11-35
 reshaping blocks into columns 11-97
 reshaping columns into blocks 11-34
 computational molecule 5-4
 connected-components labeling 8-15, 11-23
 contour plots 7-7, 11-103
 conv2 5-3, 11-39
 conversions between image types 1-12
 convmtx2 11-41
 convn 5-9, 11-42
 convolution
 convolution matrix 11-41
 Fourier transform 6-12
 higher-dimensional 5-9, 11-42
 separability 5-8
 two-dimensional 5-3, 11-39, 11-62
 convolution kernel 5-3
 center pixel 5-4
 coordinate systems
 pixel coordinates 1-16
 spatial coordinates 1-17
 corr2 7-9, 11-43
 correlation 5-7, 11-62
 Fourier transform 6-13
 correlation coefficient 11-43
 cropping an image 3-7, 11-105

D

- data types
 8-bit integers 1-3, 1-13, 11-192
 converting between 1-14, 11-50, 11-192
 double-precision 1-3, 11-50
 dct2 6-15, 11-44
 dctmtx 6-16, 11-46
 dilate 8-6, 11-47
 dilation 8-4, 11-26, 11-47
 discrete cosine transform 6-15, 11-44
 image compression 6-17
 inverse 11-93
 transform matrix 6-16, 11-46
 discrete Fourier transform 6-8
 display techniques 2-2, 11-133
 adding a colorbar 2-10, 11-37
 displaying at true size 2-4, 11-191
 multiple images 2-14, 11-189
 preferences 2-3
 texture mapping 2-17, 11-193
 zooming 2-16, 11-198

distinct block operations 4-8

- overlap 4-9, 11-13

- zero padding 4-8

dither 1-12, 11-49

dithering 10-8, 11-49, 11-176

doubl e 1-14, 11-50

E

edge 7-10, 11-51

edge detection 7-10

- methods 11-52

enhancing images

- intensity adjustment 7-14, 11-100

- noise removal 7-20

erode 8-6, 11-56

erosion 8-4, 11-26, 11-56

Euclidean distance 7-4, 11-165

Euler number 8-19, 11-18

F

fast Fourier transform 6-8

- higher-dimensional 11-60

- higher-dimensional inverse 11-95

- two-dimensional 11-58

- two-dimensional inverse 11-94

feature measurement 7-9, 11-108

- binary images 8-18

fft 6-8

fft2 6-8, 11-58

fftn 6-8, 11-60

fftshift 11-61

file formats 1-11, 11-114, 11-126, 11-135

files

- displaying images in 2-8

- reading image data from 11-126

reading image information from 11-114

writing image data to 11-135

filling a region 9-8

filter design 5-13

- frequency sampling method 5-15, 11-67

- frequency transformation method 5-14, 11-74

- windowing method 5-16, 11-77, 11-81

filter2 5-7, 11-62

filters

- adaptive 7-23, 11-195

- averaging 5-10, 11-70

- binary masks 9-6

- designing 5-13

- Finite Impulse Response (FIR) 5-13

- frequency response 5-19, 6-11

- Gaussian 11-70

- Infinite Impulse Response (FIR) 5-14

- Laplacian 11-70

- Laplacian of Gaussian 11-70

- linear 5-3, 7-21, 11-62

- median 7-21, 11-155

- order-statistic 11-161

- predefined types 5-9, 11-70

- Prewitt 11-70

- Sobel 5-11, 11-70

- unsharp 11-70

FIR filters 5-13

flood-fill operation 8-13, 11-20

Fourier transform 6-3

- computing frequency response 6-11

- convolution 6-12

- correlation 6-13

- higher-dimensional 11-60

- higher-dimensional inverse 11-95

- rearranging output 11-61

- two-dimensional 11-58

- two-dimensional inverse 11-94

f
 freqspace 5-18, 11-64
frequency response
 computing 5-19, 6-11, 11-65
 desired response matrix 5-18, 11-64
frequency sampling method (filter design) 5-15,
 11-67
frequency transformation method (filter design)
 5-14, 11-74
freqz2 5-19, 6-11, 11-65
fsamp2 5-15, 11-67
fspeci al 5-9, 11-70
ftrans2 5-14, 11-74
fwind1 5-17, 11-77
fwind2 5-17, 11-81

G
 gamma correction 7-16
 Gaussian filter 11-70
 Gaussian noise 7-23
 geometric functions 3-2
 cropping 3-7, 11-105
 interpolation 3-3
 resizing 3-5, 11-129
 rotation 3-6, 11-131
get **i**mage 11-85
getting preference values 11-142
gray2i nd 1-12, 11-87
gayscale morphological operations 11-161
grayscale 1-12, 11-88

H
 HDF files 1-11, 11-114, 11-126, 11-135
histeq 7-18, 11-89
 histogram equalization 7-18, 11-89
 histograms 7-8, 11-117

Hough transform
 detecting lines 6-22
HSV color space 10-10, 11-92, 11-175
hsv2rgb 10-10, 11-92

I
idct2 11-93
ifft 6-8
ifft2 6-8, 11-94
ifftn 6-8, 11-95
 IIR filters 5-14
im2bw 1-12, 11-96
im2col 11-97
im2doubl e 1-14, 11-98
im2ui nt8 1-14, 11-99
imadj ust 7-14, 11-100
 image area (binary images) 8-18, 11-16
 image types
 binary 1-7, 8-2
 converting between 1-11
 indexed 1-5
 intensity 1-6
 multiframe images 1-9
 RGB (truecolor) 1-8
 images
 analyzing 7-3
 color 10-2
 color depth 10-3
 converting to binary 11-96
 data types 1-3, 11-50, 11-192
 displaying 2-2, 11-133
 displaying multiple images 2-14, 11-189
 enhancing 7-14
 file formats 1-11, 11-114, 11-126, 11-135
 getting data from axes 11-85
 mean value 11-154

- reading data from files 11-126
 reading information from files 11-114
 reducing number of colors 10-5, 11-102
 standard deviation 11-188
 storing in MATLAB 1-3
 writing to files 11-135
- i**
mapprox 10-7, 11-102
mcontour 7-7, 11-103
mcrop 3-7, 11-105
mfeature 7-9, 8-18, 11-108
mfinfo 1-11, 11-114
mhist 7-8, 11-117
mmovie 2-13, 11-118
mnoise 7-21, 11-119
mpiexel 7-3, 11-121
iprofile 7-4, 11-123
imread 1-11, 11-126
imresize 3-5, 11-129
imrotate 3-6, 11-131
imshow 2-3, 11-133
imwrite 1-11, 11-135
ind2gray 1-12, 11-139
ind2rgb 1-12, 11-140
indexed images 1-5, 11-149
 converting from intensity 11-87
 converting from RGB 11-176
 converting to intensity 11-139
 converting to RGB 11-140
 displaying 2-5
 reducing number of colors 10-5
intensity adjustment 7-14, 11-100
 gamma correction 7-16
 histogram equalization 7-18
intensity images 1-6, 11-148
 converting from indexed 11-139
 converting from matrices 11-153
 converting from RGB 11-174
 converting to indexed 11-87
 displaying 2-5
 number of gray levels displayed 2-6
intensity profiles 7-4, 11-123
interpolation 3-3
 bicubic 3-3
 bilinear 3-3
 intensity profiles 7-4
 nearest neighbor 3-3
iptgetpref 2-4, 11-141
iptsetpref 2-3, 11-142
iradon 6-25, 11-144
isbw 11-147
isgray 11-148
isind 11-149
isrgb 11-150
- J**
 JPEG files 1-11, 11-114, 11-126, 11-135
 JPEG image compression 6-17
- L**
 Laplacian filter 11-70
 Laplacian of Gaussian edge detector 11-52
 Laplacian of Gaussian filter 11-70
linear filtering 4-5, 5-3, 11-62
 averaging filter 5-10
 center pixel 5-4
 computational molecule 5-4
convolution 5-3
 convolution kernel 5-3
correlation 5-7
 filter design 5-13
FIR filters 5-13
IIR filters 5-14

noise removal 7-21
 predefined filter types 5-9
 Sobel filter 5-11
 logical arrays 1-7, 1-15
 lookup-table operations 8-20, 11-151

M

`makelut` 8-20, 11-151
 masked filtering 9-6, 11-184
`mat2gray` 1-12, 11-153
 matrices
 converting to intensity images 11-153
 storing images 1-3
`McClellan transform` 11-74
`mean2` 7-9, 11-154
`medfilt2` 7-21, 11-155
 median filtering 7-21, 11-155
 minimum variance quantization 10-6, 11-177
 Moiré patterns 3-5
`montage` 2-12, 11-157
 morphological operations 8-4, 11-25
 center pixel 8-4
 closure 8-7, 11-25
 diagonal fill 11-26
 dilation 8-4, 11-26, 11-47
 erosion 8-4, 11-26, 11-56
 grayscale 11-161
 opening 8-7, 11-26
 predefined operations 8-8
 removing spur pixels 11-27
 shrinking objects 11-26
 skeletonizing objects 11-26
 structuring elements 8-4
 thickening objects 11-27
 thinning objects 11-27

movies
 creating from images 2-13, 11-118
 playing 2-14
 multiframe images 1-9
 displaying 2-11, 11-157
 limitations 1-10
 multilevel thresholding 11-88

N

nearest neighbor interpolation 3-3
 neighborhood operations 4-2
 neighborhoods
 binary image operations 8-2, 8-10, 11-10
`nlfilter` 4-6, 11-159
 noise removal 7-20
 adding noise 11-119
 Gaussian noise 7-23, 11-119
 salt and pepper noise 7-21, 11-119
 speckle noise 11-119
 nonlinear filtering 4-5
`NTSC color space` 10-9, 11-160, 11-178
`ntsc2rgb` 10-9, 11-160

O

object selection 8-16, 11-30
 opening 8-7, 11-26
 order-statistic filtering 11-161
`ordfilt2` 11-161

P

padding borders
 binary image operations 8-2
 block processing 4-5
 linear filtering 5-5

PCX files 1-11, 11-114, 11-126, 11-135
 perimeter determination 8-12, 11-29
 phantom 6-25, 11-162
 pixel coordinates 1-16
 pixel values 7-3, 11-121, 11-165
 pixels
 definition 1-3
 pixval 7-4, 11-165
 plotting colormap values 11-180
 preferences 2-3
 getting values 11-142
 setting values 11-141
 Prewitt edge detector 11-52
 Prewitt filter 11-70

Q

qtdecomp 7-11, 11-166
 qtgetblk 11-169
 qtsetblk 11-171
 quadtree decomposition 7-11, 11-166
 getting block values 11-169
 setting block values 11-171
 quantization
 minimum variance quantization 10-6, 11-177
 uniform quantization 10-6, 11-177

R

radon 6-19, 11-172
 Radon transform 6-19, 11-172
 detecting lines 6-22
 inverse 6-25, 11-144
 region of interest
 binary masks 9-3
 filling 9-8, 11-182

filtering 9-6, 11-184
 selecting 9-3, 9-5, 11-181, 11-186
 resizing images 3-5, 11-129
 anti-aliasing 3-5
 RGB images 1-8, 11-150
 converting from indexed 11-140
 converting to indexed 11-176
 converting to intensity 11-174
 displaying 2-7
 reducing number of colors 10-5
 rgb2gray 1-12, 11-174
 rgb2hsv 10-10, 11-175
 rgb2ind 1-12, 10-5, 11-176
 rgb2ntsc 10-9, 11-178
 rgb2ycbcr 10-10
 rgbplot 11-180
 Roberts edge detector 11-52
 roi col or 9-5, 11-181
 roi fill 9-8, 11-182
 roi filt 2 9-6, 11-184
 roi poly 9-3, 11-186
 rotating an image 3-6, 11-131

S

salt and pepper noise 7-21
 separability in convolution 5-8
 setting preference values 11-141
 sliding neighborhood operations 4-4, 11-159
 Sobel edge detector 11-52
 Sobel filter 5-11, 11-70
 spatial coordinates 1-17
 std2 7-9, 11-188
 structuring elements 8-4
 center pixel 8-4
 subimage 2-16, 11-189
 subplot 2-15

T

template matching 6-13
texture mapping 2-17, 11-193
thresholding 11-88, 11-96
TIFF files 1-11, 11-114, 11-126, 11-135
transforms 6-2
 discrete cosine 6-15, 11-44
 Fourier 6-3, 11-58, 11-60, 11-61
 inverse discrete cosine 11-93
 inverse Fourier 11-94, 11-95
 inverse Radon 6-25, 11-144
 Radon 6-19, 11-172
truecolor images 1-8
truesize 2-4, 11-191

Y

YCbCr color space 10-10, 11-179, 11-197
ycbcr2rgb 10-10

Z

zero-cross edge detector 11-52
zoom 2-16, 11-198
zooming in 2-16, 11-198

U

uint8 11-192
uint8 arrays
 storing images 1-3
 supported operations 1-13, 11-192
uniform quantization 10-6, 11-177
unsharp filter 11-70

W

warp 2-17, 11-193
wiener2 7-23, 11-195
windowing method (filter design) 5-16, 11-77,
 11-81

X

XWD files 1-11, 11-114, 11-126, 11-135