Image Processing Toolbox

For Use with MATLAB®

Computation

Visualization

Programming

The MATH WORKS Inc.

User's Guide

Version 2

How to Contact The MathWorks:

	508-647-7000	Phone
	508-647-7001	Fax
	The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail
	`http://www.mathworks.com`	Web
	`ftp.mathworks.com`	Anonymous FTP server
	`comp.soft-sys.matlab`	Newsgroup
@	`support@mathworks.com`	Technical support
	`suggest@mathworks.com`	Product enhancement suggestions
	`bugs@mathworks.com`	Bug reports
	`doc@mathworks.com`	Documentation error reports
	`subscribe@mathworks.com`	Subscribing user registration
	`service@mathworks.com`	Order status, license renewals, passcodes
	`info@mathworks.com`	Sales, pricing, and general information

Image Processing Toolbox User's Guide

Printing History:	August 1993	First printing Version 1
	May 1997	Second printing Version 2
	January 1998	Revised for Version 2.1 (Online only)
	January 1999	Revised for Version 2.2 (Release 11) (Online only)
	September 2000	Revised for Version 2.2.2 (Release 12) (Online only)

Image Credits

moon	Copyright Michael Myers. Used with permission.
cameraman	Copyright Massachusetts Institute of Technology. Used with permission.
trees	*Trees with a View*, 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.
alumgrns bacteria blood1 bonemarr circles circlesm debye1 enamel flowers ic lily ngc4024l ngc4024m ngc4024s rice saturn shot1 testpat1 testpat2 text tire	Copyright J. C. Russ, *The Image Processing Handbook*, Second Edition, 1994, CRC Press, Boca Raton, ISBN 0-8493-2516-1. Used with permission.

Contents

Preface

1 Getting Started

2 Introduction

3 Displaying and Printing Images

4 Geometric Operations

5 Neighborhood and Block Operations

6 Linear Filtering and Filter Design

7 Transforms

8 Analyzing and Enhancing Images

9 Binary Image Operations

10 Region-Based Processing

11 Color

12 Function Reference

A Working with Function Functions

Preface

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.

What Can You Do with the Image Processing Toolbox?

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, which contain MATLAB code that implements 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 2.2

Version 2.2 offers the following new features: 16-bit image processing (most functions); speed optimization of many functions, including `bwfill`, `bwselect`, `bwlabel`, `dilate`, `erode`, `histeq`, `imresize`, `imrotate`, `ordfilt2`, `medfilt2`, and `im2uint8`; new border-padding options for `medfilt2` and `ordfilt2`; and a new function, `im2uint16`.

In addition, some of the new features and changes in MATLAB 5.3 (Release 11) are relevant to the operation of the Image Processing Toolbox 2.2. Relevant changes in MATLAB 5.3 include: improved support for integer types (`uint8`, int8, uint16, int16, uint32, and int32); support for two new file formats, PNG and HDF-EOS; 16-bit image display; and 16-bit TIFF file I/O.

Updates to Earlier Versions

Version 2.1 offered the following new features: inverse Radon transform; interactive pixel value display including distance between two pixels; advanced feature measurement; Canny edge detection; YCbCr color space support; easier data precision conversion; and a new feature for the bwfill function—the ability to automatically detect and fill holes in objects.

Version 2.0 offered the following new features: support for 8-bit image data; support for manipulating RGB and multiframe images as multidimensional arrays; optimization of some 1.0 functions; and many new functions.

For a detailed description of the changes in Versions 2.0, 2.1, and 2.2, see the *Release 11 New Features* document.

For a list of all of the functions in the Image Processing Toolbox, see "Functions by Category."

Related Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with the Image Processing Toolbox.

For more information about any of these products, see either:

- The online documentation for that product, if it is loaded or if you are reading the documentation from the CD
- The MathWorks Web site, at http://www.mathworks.com; see the "products" section

---

Note The products listed below all include functions that extend the Image Processing Toolbox's capabilities.

Product	Description
Fuzzy Logic Toolbox	Tool to help master fuzzy logic techniques and their application to practical control problems
Mapping Toolbox	Tool for analyzing and displaying geographically based information from within MATLAB
MATLAB	Integrated technical computing environment that combines numeric computation, advanced graphics and visualization, and a high-level programming language
Neural Network Toolbox	Comprehensive environment for neural network research, design, and simulation within MATLAB
Optimization Toolbox	Tool for general and large-scale optimization of nonlinear problems, as well as for linear programming, quadratic programming, nonlinear least squares, and solving nonlinear equations
Signal Processing Toolbox	Tool for algorithm development, signal and linear system analysis, and time-series data modeling
Statistics Toolbox	Tool for analyzing historical data, modeling systems, developing statistical algorithms, and learning and teaching statistics
Wavelet Toolbox	Tool for signal and image analysis, compression, and de-noising

The Signal Processing Toolbox and the Wavelet Toolbox are closely related products. The Signal Processing Toolbox is strongly recommended for 2-D FIR filter design to generate the inputs (1-D windows and 1-D filter prototypes) to the 2-D FIR design functions. (The Signal Processing Toolbox supports a wide range of signal processing operations, from waveform generation to filter design and implementation, parametric modeling, and spectral analysis.)

The Image Processing Toolbox 2.2 requires MATLAB 5.3. The Image Processing Toolbox uses MATLAB as the computational engine for most of its algorithms. Additionally, MATLAB offers powerful capabilities, such as advanced data manipulation and analysis, that you can use to complement and enhance the features in the Image Processing Toolbox.

See the MATLAB documentation for descriptions of the MATLAB language, including how to enter and manipulate data and how to use MATLAB's extensive collection of functions. It also explains how to create your own functions and scripts. The MATLAB Function Reference provides reference descriptions of the supplied MATLAB functions and commands.

Post Installation Notes

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.

Note For the most up-to-date information about system requirements, see the system requirements page, available in the products area at the MathWorks Web site (`http://www.mathworks.com`).

About This Manual

This manual has four main parts:

- Chapter 1, “Getting Started”, contains two step-by-step examples that will help you get started using the Image Processing Toolbox. This chapter is written as both an introduction to the most frequently used operations, as well as a demonstration of some of the image analysis that can be performed.
- Chapter 2, “Introduction”, and Chapter 3, “Displaying and Printing Images”, discuss working with image data and displaying images in MATLAB and the Image Processing Toolbox.
- Chapters 4 to 11 provide in-depth discussion of the concepts behind the software. Each chapter covers a different topic in image processing. For example, Chapter 7 discusses linear filtering, and Chapter 11 discusses binary image operations. Each chapter provides numerous examples that apply the toolbox to representative image processing tasks.
- Chapter 12, “Function Reference”, 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.

User Experience Levels

This section gives brief suggestions for how to use the documentation, depending on your level of experience in using the toolbox and MATLAB, and your knowledge of image processing concepts.

All new toolbox users should read Chapter 1, “Getting Started” and Chapter 2, “Introduction.”

Note If you are not familiar with MATLAB, it is strongly suggested that you start by reading and running the examples in *Getting Started with MATLAB*.

Users who are less knowledgeable about image processing concepts will find that the following chapters, in particular, contain valuable introductory discussions: Chapter 5, “Neighborhood and Block Operations”, Chapter 6, "Linear Filtering and Filter Design", Chapter 9, "Binary Image Operations", and Chapter 11, "Color."

Experienced toolbox users should read the *Release Notes*. This guide is available as an online document that can be opened by clicking on its title at the top of the **Contents** tab of the MATLAB Help browser. To open the Help browser, select **Help** from the MATLAB desktop's **View** menu.

While experienced users may primarily prefer to use the reference chapters of this user guide, they should note that some functions are demonstrated in longer examples in the tutorial chapters. To see if a function has an example in a tutorial chapter, check the index entry of the function name.

Words You Need to Know

At the beginning of each chapter we provide glossaries of key words you need to know in order to understand the information in the chapter. The chosen words are defined generally, and then sometimes include a MATLAB specific definition.

Many of the words are standard image processing terms that we define for your convenience. In some cases, the words are included because they can sometimes be confusing, even for domain experts. Here are some examples:

- In image processing, one word is sometimes used to describe more than one concept. For example, image *resolution* can be defined as the height and width of an image as a quantity of pixels in each direction, *or* it can be defined as the number of pixels per linear measure, such as 100 pixels per inch.
- In image processing, one concept is sometimes described by different terminology. For example, a *grayscale* image can also be called an *intensity* image. We use the word *intensity* in our documentation and include a definition for it, because it may be unfamiliar to those who use the word "grayscale." (It is also defined in order to explain how MATLAB stores an intensity image.)

If you want to know whether a word is defined in one of the chapter glossaries, look up the word in the index. If we have defined it, the index entry for the word will have a subentry of "definition."

For terminology that is new to you and not covered in our “Words You Need to Know” tables, we suggest you consult a more complete image processing glossary.

Typographical Conventions

This manual uses some or all of these general MathWorks documentation conventions, as well as some special ones described after the following table..

Item	Convention to Use	Example
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names/syntax	`Monospace` font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Keys	**Boldface** with an initial capital letter	Press the **Return** key.
Literal strings (in syntax descriptions in Reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,`**`'whole'`**`)`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$

MATLAB output	`Monospace` font	MATLAB responds with
		`A =`
		`    5`

Menu names, menu items, and controls	**Boldface** with an initial capital letter	Choose the **File** menu.

Item	Convention to Use	Example
New terms	*Italics*	An *array* is an ordered collection of information.
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'`*`method`*`')`

Image Processing Toolbox Typographical Conventions

We often use the variable names `I`, `RGB`, `X`, and `BW` in the code examples in this *User Guide*. `I` is used for intensity images, `RGB` for `RGB` images, `X` for indexed images, and `BW` for binary images (where it stands for “black and white”). In addition, `map` is often used as a variable name for the colormap associated with an indexed image.

See Chapter 2, “Introduction” for more information about these different representations.

We use conventions to differentiate data ranges from MATLAB vectors. While both are enclosed by square brackets, there are the following differences: commas signify a range, the lack of commas and the use of the monospace font signify a MATLAB vector.

[0, 1] is a range of pixel values from 0 to 1.

[0, 255] is a range of pixel values from 0 to 255.

`[0 1]` is a vector of two values, `0` and `1`.

Image Processing Demos

The Image Processing Toolbox is supported by a full complement of demo applications. These are very useful as templates for your own end-user applications, or for seeing how to use and combine your toolbox functions for powerful image analysis and enhancement. The toolbox demos are located under the subdirectory,

```
...\TOOLBOX\IMAGES\IMDEMOS
```

under the top-level directory in which MATLAB is installed.

The table below lists the demos available. Demos whose names begin with "`ipss`" operate as slide shows.

The easiest way to run an individual demo is to enter its name at the MATLAB command prompt. You can also launch MATLAB demos from the MATLAB demo window. To invoke this window, type `demo` at the command prompt. To see the list of available image processing demos, double-click on **Toolboxes** from the list on the left, and then select **Image Processing**. Select the desired demo and click the **Run** button.

To view the code in a demo, type

```
edit demoname
```

at the MATLAB command prompt.

For information on what is happening in the demo and how to use it, press the **Info** button, which is located in the lower right corner of each demo window. All demos that are not slide shows offer a selection of images on which to operate and a number of settings for you to experiment with. Most of these demos also have an **Apply** button, which must be pressed to see the results of your new settings.

Demos for the Image Processing Toolbox

Demo Name	Brief Description
dctdemo	Discrete cosine transform (DCT) image compression: you choose the number of coefficients and it shows you a reconstructed image and an error image.
edgedemo	Edge detection: all supported types with optional manual control over threshold, direction, and sigma, as appropriate to the method used.
firdemo	2-D Finite impulse response (FIR) filters: design your own filter by changing the cut-off frequency and filter order.
imadjdemo	Contrast adjustment and histogram equalization: adjust intensity values using brightness, contrast, and gamma correction, or by using histogram equalization.
ipss001	Connected components labeling slide show: includes double thresholding, feature-based logic, and binary morphology. All operations are performed on one image.
ipss002	Feature-based logic slide show containing two examples: the first example shows object selection using AND operations on the on pixels in two binary images; the second example shows filtering and thresholding on a single image.
ipss003	Correction of nonuniform illumination slide show: creates a coarse approximation of the background, subtracts it from the image, and then adjusts the pixel intensity values to fill the entire range.
landsatdemo	Landsat color composites: choose a scene and assign spectral bands to RGB intensities to create images that reveal topography, vegetation, and moisture; toggle saturation stretching to see its effect on image contrast.

Demos for the Image Processing Toolbox (Continued)

Demo Name	Brief Description
nrfiltdemo	Noise reduction using linear and nonlinear filters: enables you to add different types of noise with variable densities, and choose a filter neighborhood size.
qtdemo	Quadtree decomposition: enables you to select a threshold and see a representation of the sparse matrix, and a reconstruction of the original image.
roidemo	Region of Interest (ROI) selection: enables you to select an ROI and apply operations such as unsharp and fill. It displays the binary mask of the ROI.

MATLAB Newsgroup

If you read newsgroups on the Internet, you might be interested in the MATLAB newsgroup (`comp.soft-sys.matlab`). This newsgroup gives you access to an active MATLAB user community. It is an excellent way to seek advice and to share algorithms, sample code, and M-files with other MATLAB users.

Getting Started

Overview

This chapter contains two exercises to get you started doing image processing using MATLAB and the Image Processing Toolbox. The exercises include sections called "Here's What Just Happened" so that you can read further about the operations you just used. In addition, the exercises contain cross-references to other sections in this manual that have in-depth discussions on the concepts presented in the examples.

Note If you are new to MATLAB, you should first read *Getting Started with MATLAB*.

All of the images displayed by the exercises in this chapter are supplied with the Image Processing Toolbox. Note that the images shown in this documentation differ slightly from what you see on your screen because the surrounding MATLAB figure window has been removed to save space.

"Exercise 1 — Some Basic Topics" covers the basic tasks of reading and displaying an image, adjusting its contrast, and writing it back to disk. This exercise introduces you to one of the supported image types (the intensity image) and to one of the numeric storage classes used for images (`uint8`). "Exercise 2 — Advanced Topics" includes more sophisticated topics, such as components labeling and feature measurement, which are two of the many specialized types of image processing that you can perform using the Image Processing Toolbox.

Exercise 1 — Some Basic Topics

Before beginning with this exercise, start MATLAB. You should already have installed the Image Processing Toolbox, which runs seamlessly from MATLAB. For information about installing the toolbox, see the *MATLAB Installation Guide* for your platform.

1. Read and Display an Image

Clear the MATLAB workspace of any variables and close open figure windows.

```
clear, close all
```

To read an image use the `imread` command. Let's read in a TIFF image named `pout.tif` (which is one of the sample images that is supplied with the Image Processing Toolbox), and store it in an array named `I`.

```
I=imread('pout.tif');
```

Now call `imshow` to display `I`.

```
imshow(I)
```

2. Check the Image in Memory

Enter the `whos` command to see how `I` is stored in memory.

```
whos
```

MATLAB responds with

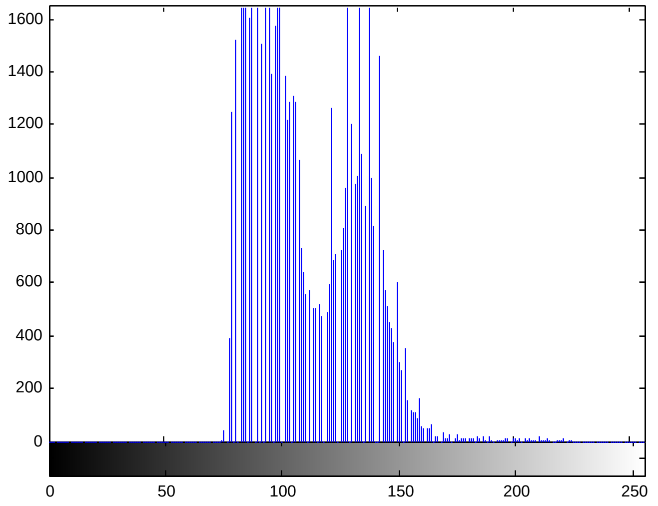
```
  Name      Size         Bytes  Class

    I       291x240        69840  uint8 array

  Grand total is 69840 elements using 69840 bytes
```

Here's What Just Happened
Step 1. The `imread` function recognized `pout.tif` as a valid TIFF file and stored it in the variable `I`. (For the list of graphics formats supported, see `imread` in the "Function Reference" chapter.) The functions `imread` and `imshow` read and display graphics images in MATLAB. In general, it is preferable to use `imshow` for displaying images because it handles the image-related MATLAB properties for you. (The MATLAB function `image` is for low-level programming tasks.) Note that if `pout.tif` were an *indexed* image, the appropriate syntax for `imread` would be, `[X, map] = imread('pout.tif');` (For more information on the supported image types, see "Image Types in the Toolbox" on page 2-5.) **Step 2.** You called the `whos` command to see how `pout.tif` had been stored into the MATLAB workspace. As you saw, `pout.tif` is stored as a 291-by-240 array. Since `pout.tif` was an 8-bit image, it gets stored in memory as an `uint8` array. MATLAB can store images in memory as `uint8`, `uint16`, or `double` arrays. (See "Reading a Graphics Image" on page 2-14 for an explanation of when the different storage classes are used.)

3. Perform Histogram Equalization

As you can see, `pout.tif` is a somewhat low contrast image. To see the distribution of intensities in `pout.tif` in its current state, you can create a histogram by calling the `imhist` function. (Precede the call to `imhist` with the figure command so that the histogram does not overwrite the display of the image I in the current figure window.)

```
figure, imhist(I) % Display a histogram of I in a new figure.
```



Notice how the intensity range is rather narrow. It does not cover the potential range of [0, 255], and is missing the high and low values that would result in good contrast.

Now call histeq to spread the intensity values over the full range, thereby improving the contrast of I. Return the modified image in the variable I2.

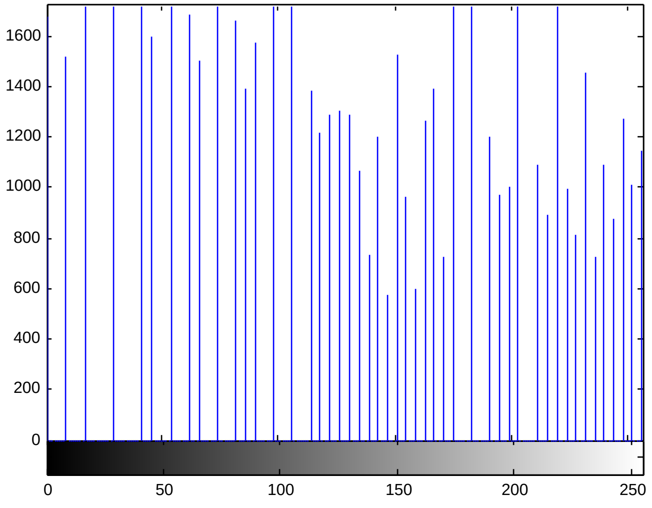
```
I2 = histeq(I); % Equalize I and output in new array I2.
```

Display the new equalized image, I2, in a new figure window.

```
figure, imshow(I2)  % Display the new equalized image I2.
```

Call `imhist` again, this time for `I2`.

```
figure, imhist(I2) % Show the histogram for the new image I2.
```



See how the pixel values now extend across the full range of possible values.

Here's What Just Happened

Step 3. You adjusted the contrast automatically by using the function `histeq` to evenly distribute the image's pixel values over the full potential range for the storage class of the image. For an image `X`, with a storage class of `uint8`, the full range is $0 \le X \le 255$, for `uint16` it is $0 \le X \le 65535$, and for `double` it is $0 \le X \le 1$. Note that the convention elsewhere in this user guide (and for all MATLAB documentation) is to denote the above ranges as [0,255], [0,65535], and [0,1], respectively.

If you compare the two histograms, you can see that the histogram of `I2` is more spread out and flat than the histogram of `I1`. The process that flattened and spread out this histogram is called *histogram equalization*.

For more control over adjusting the contrast of an image (for example, if you want to chose the range over which the new pixel values should span), you can use the `imadjust` function, which is demonstrated under "6. Adjust the Image Contrast" on page 1-17 in Exercise 2.

4. Write the Image

Write the newly adjusted image `I2` back to disk. Let's say you'd like to save it as a PNG file. Use `imwrite` and specify a filename that includes the extension `'png'`.

```
imwrite (I2, 'pout2.png');
```

Here's What Just Happened

Step 4. MATLAB recognized the file extension of `'png'` as valid and wrote the image to disk. It wrote it as an 8-bit image by default because it was stored as a `uint8` intensity image in memory. If `I2` had been an image array of type RGB and class `uint8`, it would have been written to disk as a 24-bit image. If you want to set the bit depth of your output image, use the `BitDepth` parameter with `imwrite`. This example writes a 4-bit PNG file.

```
imwrite(I2, 'pout2.png', 'BitDepth', '4');
```

Note that all output formats do not support the same set of output bit depths. For example, the toolbox does not support writing 1-bit BMP images. See `imwrite` in the "Reference" chapter for the list of valid bit depths for each format. See also "Writing a Graphics Image" on page 2-15 for a tutorial discussion on writing images using the Image Processing Toolbox.

5. Check the Contents of the Newly Written File

Now, use the `imfinfo` function to see what was written to disk. Be sure not to end the line with a semicolon so that MATLAB displays the results. Also, be sure to use the same path (if any) as you did for the call to `imwrite`, above.

```
imfinfo('pout2.png')
```

MATLAB responds with

```
ans =
         Filename:'pout2.png'
      FileModDate:'03-Jun-1999 15:50:25'
         FileSize:36938
           Format:'png'
    FormatVersion:[]
            Width:240
           Height:291
         BitDepth:8
        ColorType:'grayscale'
            . . .
```

Note The value in the `FileModDate` field for your file will be different from what is shown above. It will show the date and time that you used `imwrite` to create your image. Note also that we truncated the number of field names and values returned by this call.

Here's What Just Happened
Step 5. When you called `imfinfo`, MATLAB displayed all of the header fields for the PNG file format that are supported by the toolbox. You can modify many of these fields by using additional parameters in your call to `imwrite`. The additional parameters that are available for each file format are listed in tables in the reference entry for `imwrite`. (See "Querying a Graphics File" on page 2-16 for more information about using `imfinfo`.)

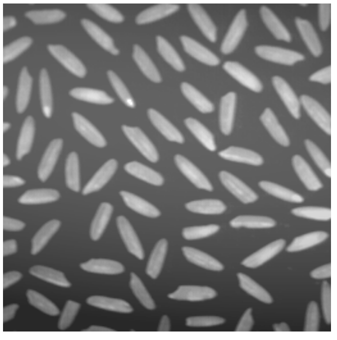
Exercise 2 — Advanced Topics

In this exercise you will work with another intensity image, `rice.tif` and explore some more advanced operations. The goals of this exercise are to remove the nonuniform background from `rice.tif`, convert the resulting image to a binary image by using thresholding, use components labeling to return the number of objects (grains or partial grains) in the image, and compute feature statistics.

1. Read and Display An Image

Clear the MATLAB workspace of any variables and close open figure windows. Read and display the intensity image `rice.tif`.

```
clear, close all
I = imread('rice.tif');
imshow(I)
```



2. Perform Block Processing to Approximate the Background

Notice that the background illumination is brighter in the center of the image than at the bottom. Use the `blkproc` function to find a coarse estimate of the background illumination by finding the minimum pixel value of each 32-by-32 block in the image.

```
backApprox = blkproc(I,[32 32],'min(x(:))');
```

To see what was returned to `backApprox`, type

```
backApprox
```

MATLAB responds with

```
backApprox =

    80    81    81    79    78    75    73    71
    90    91    91    90    89    87    84    83
    94    96    96    97    96    95    94    90
    90    93    93    95    96    95    94    93
    80    83    85    87    87    88    87    87
    68    69    72    74    76    76    77    76
    48    51    54    56    59    60    61    61
    40    40    40    40    40    40    40    41
```

Here's What Just Happened

Step 1. You used the toolbox functions `imread` and `imshow` to read and display an 8-bit intensity image. `imread` and `imshow` are discussed in Exercise 1, in "2. Check the Image in Memory" on page 1-3, under the "Here's What Just Happened" discussion.

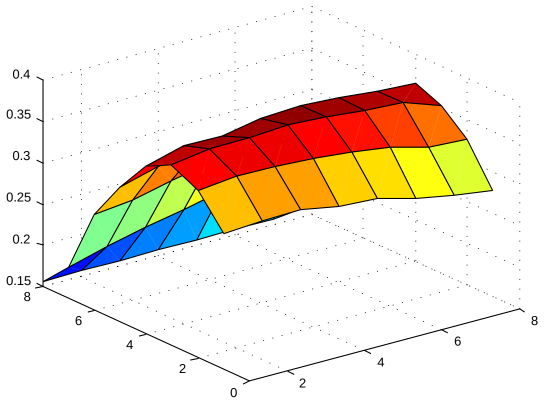
Step 2. `blkproc` found the minimum value of each 32-by-32 block of `I` and returned an 8-by-8 matrix, `backApprox`. You called `blkproc` with an input image of `I`, and a vector of `[32 32]`, which means that `I` will be divided into 32-by-32 blocks. `blkproc` is an example of a "function function," meaning that it enables you to supply your own function as an input argument. You can pass in the name of an M-file, the variable name of an inline function, or a string containing an expression (this is the method that you used above). The function defined in the string argument (`'min(x(:))'`) tells `blkproc` what operation to perform on each block. For detailed instructions on using function functions, see Appendix A.

MATLAB's `min` function returns the minimum value of each column of the array within parentheses. To get the minimum value of the entire block, use the notation `(x(:))`, which reshapes the entire block into a single column. For more information, see `min` in the MATLAB Function Reference.

3. Display the Background Approximation As a Surface

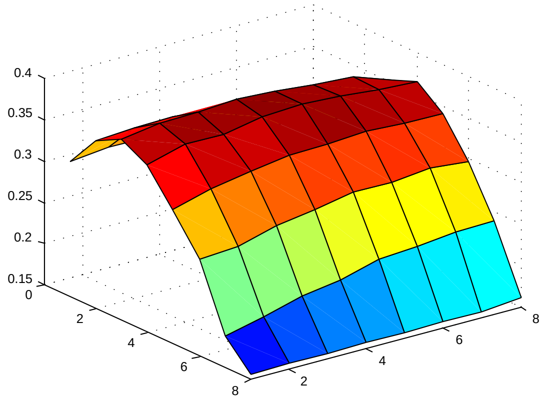
Use the `surf` command to create a surface display of the background approximation, `backApprox`. `surf` requires data of class `double`, however, so you first need to convert `backApprox` using the `double` command. You also need to divide the converted data by 255 to bring the pixel values into the proper range for an image of class `double`, [0 1].

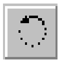
```
backApprox = double(backApprox)/255; % Convert image to double.
figure, surf(backApprox);
```



To see the other side of the surface, reverse the *y*-axis with this command,

```
set(gca,'ydir','reverse'); % Reverse the y-axis.
```





To rotate the surface in any direction, click on the rotate button in the toolbar (shown at left), then click and drag the surface to the desired view.

Here's What Just Happened

Step 3. You used the `surf` command to examine the background image. The `surf` command creates colored parametric surfaces that enable you to view mathematical functions over a rectangular region. In the first surface display, [0, 0] represents the origin, or upper-left corner of the image. The highest part of the curve indicates that the highest pixel values of `backApprox` (and consequently `rice.tif`) occur near the middle rows of the image. The lowest pixel values occur at the bottom of the image and are represented in the surface plot by the lowest part of the curve. Because the minimum intensity values in each block of this image make a smooth transition across the image, the surface is comprised of fairly smooth curves.

The surface plot is a Handle Graphics® object, and you can therefore fine-tune its appearance by setting properties. ("Handle Graphics" is the name for the collection of low-level graphics commands that create the objects you generate using MATLAB.) The call to reverse the *y*-axis is one of many property settings that you can make. It was made using the `set` command, which is used to set all properties. In the line,

```
set(gca,'ydir','reverse');
```

`gca` refers to the handle of the current axes object and stands for "get current axes." You can also set many properties through the **Property Editor**. To invoke the **Property Editor**, open the figure window's **Edit** menu, and select **Figure Properties**, **Axes Properties**, or **Current Object Properties**. To select an object to modify with the **Property Editor**, click the property the following button on the figure window, , then click on the object. You can also use the other buttons in the toolbar to add new text or line objects to your figure.

For information on working with MATLAB graphics, see the MATLAB graphics documentation.

4. Resize the Background Approximation

Our estimate of the background illumination is only 8-by-8. Expand the background to the same size as the original background image (256-by-256) by using the `imresize` function, then display it.

```
backApprox256 = imresize(backApprox, [256 256], 'bilinear');
figure, imshow(backApprox256) % Show resized background image.
```

Here's What Just Happened

Step 4. You used `imresize` with "bilinear" interpolation to resize your 8-by-8 background approximation, `backApprox`, to an image of size 256-by-256, so that it is now the same size as `rice.tif`. If you compare it to `rice.tif` you can see that it is a very good approximation. The good approximation is possible because of the low spatial frequency of the background. A high-frequency background, such as a field of grass, could not have been as accurately approximated using so few blocks.

The interpolation method that you choose for `imresize` determines the values for the new pixels you add to `backApprox` when increasing its size. (Note that interpolation is also used to find the best values for pixels when an image is decreased in size.) The other types of interpolation supported by `imresize` are "nearest neighbor" (the default), and "bicubic." For more information on interpolation and resizing operations, see "Interpolation" on page 4-4 and "Image Resizing" on page 4-6.

5. Subtract the Background Image from the Original Image

Now subtract the background image from the original image to create a more uniform background. First, change the storage class of I to double, because subtraction can only be performed on double arrays.

```
I = im2double(I); % Convert I to storage class of double.
```

Now subtract backApprox256 from I and store it in a new array, I2.

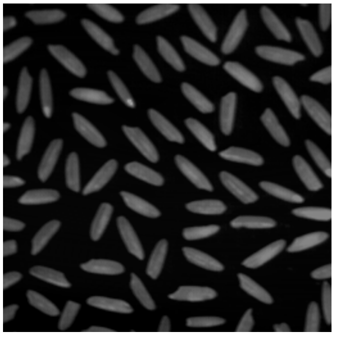
```
I2 = I - backApprox256; % Subtract the background from I.
```

Subtracting backApprox256 from I may yield some out-of-range values in the image. To correct the dynamic range of pixel values, use the max and min functions to clip pixel values outside the range [0,1].

```
I2 = max(min(I2,1),0); % Clip the pixel values to the valid range.
```

Now display the image with its more uniform background.

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



Here's What Just Happened

Step 5. You subtracted a background approximation image from `rice.tif`. Before subtracting the background approximation it was necessary to convert the image to class `double`. This is because subtraction, like many of MATLAB's mathematical operations, is only supported for data of class `double`.

The ease with which you can subtract one image from another is an excellent example of how MATLAB's matrix-based design makes it a very powerful tool for image processing.

After subtraction was completed, another step was required before displaying the new image, because subtraction often leads to out-of- range values. You therefore used the `min` and `max` functions to clip any values outside the range of [0 1].

The following call

```
I2 = max(0,min(1,I2));
```

can more easily be explained by dividing the expression into two steps, as follows.

```
I2=min(I2,1);
```

replaces each value in `I2` that is greater than 1 with 1, and

```
I2=max(I2,0);
```

replaces each value in `I2` that is less than 0 with 0.

The Image Processing Toobox has a demo, `ipss003`, that approximates and removes the background from an image. For information on how to run this (and other demos), see "Image Processing Demos" in the Preface.

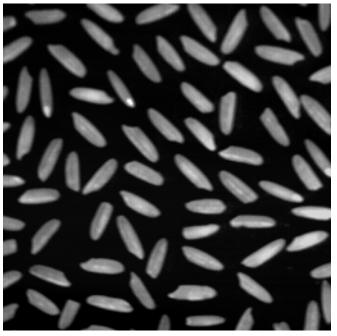
6. Adjust the Image Contrast

The image is now a bit too dark. Use `imadjust` to adjust the contrast.

```
I3 = imadjust(I2, [0 max(I2(:))], [0 1]); % Adjust the contrast.
```

Display the newly adjusted image.

```
figure, imshow(I3);
```



Here's What Just Happened

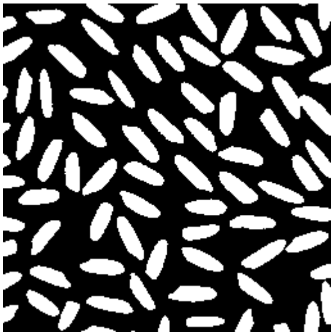
Step 6. You used the `imadjust` command to increase the contrast in the image. `imadjust` takes an input image and can also take two vectors: `[low high]` and `[bottom top]`. The output image is created by mapping the value `low` in the input image to the value `bottom` in the output image, mapping the value `high` in the input image to the value `top` in the output image, and linearly scaling the values in between. See the reference pages for `imadjust` for more information.

The expression `max(I2(:))` that you entered as the `high` value for the input image uses the MATLAB `max` command to reshape `I2` into a single column and return its maximum pixel value.

7. Apply Thresholding to the Image

Create a new binary thresholded image, `bw`, by comparing each pixel in `I3` to a threshold value of `0.2`.

```
bw=I3>0.2; % Make I3 binary using a threshold value of 0.2.
figure, imshow(bw)
```



Now call the `whos` command to see what type of array the thresholded image `bw` is.

```
whos
```

MATLAB responds with

```
Name                  Size         Bytes  Class

  I                   256x256      524288  double array
  I2                  256x256      524288  double array
  I3                  256x256      524288  double array
  backApprox            8x8           512  double array
  backApprox256       256x256      524288  double array
  bw                  256x256      524288  double array (logical)

Grand total is 327744 elements using 2621952 bytes
```

Here's What Just Happened

Step 7. You compared each pixel in `I3` with a threshold value of `0.2`. MATLAB interprets this command as a logical comparison and therefore outputs values of 1 or 0, where 1 means "true" and 0 means "false." The output value is 1 when the pixel in `I3` is greater than `0.2`, and 0 otherwise.

Notice that when you call the `whos` command, you see the expression `logical` listed after the class for `bw`. This indicates the presence of a logical flag. The flag indicates that bw is a logical matrix, and the Image Processing Toolbox treats logical matrices as binary images. Thresholding using MATLAB's logical operators always results in a logical image. For more information about binary images and the logical flag, see "Binary Images" on page 2-7.

Thresholding is the process of calculating each output pixel value based on a comparison of the corresponding input pixel with a threshold value. When used to separate objects from a background, you provide a threshold value over which a pixel is considered part of an object, and under which a pixel is considered part of the background. Due to the uniformity of the background in I3 and its high contrast with the objects in it, a fairly wide range of threshold values can produce a good separation of the objects from the background. Experiment with other threshold values. Note that if your goal were to calculate the area of the image that is made up of the objects, you would need to choose a more precise threshold value — one that would not allow the background to encroach upon (or "erode") the objects.

Note that the Image Processing Toolbox also supplies the function `im2bw`, which converts an RGB, indexed, or intensity image to a binary image based on the threshold value that you supply. You could have used this function in place of the MATLAB command, `bw = I3 > 0.2`. For example, `bw = im2bw(I3, 0.2)`. See the reference page for `im2bw` for more information.

8. Use Connected Components Labeling to Determine the Number of Objects in the Image

Use the `bwlabel` function to label all of the connected components in the binary image `bw`.

```
[labeled,numObjects] = bwlabel(bw,4);% Label components.
```

Show the number of objects found by `bwlabel`.

```
numObjects
```

MATLAB responds with

```
numObjects =

    80
```

You have just calculated how many objects (grains or partial grains of rice) are in `rice.tif`.

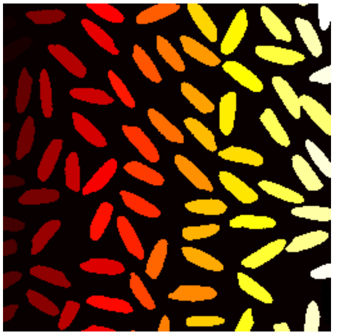
Note The accuracy of your results depends on a number of factors, including:

• The size of the objects
• The accuracy of your approximated background
• Whether you set the connected components parameter to 4 or 8
• The value you choose for thresholding
• Whether or not any objects are touching (in which case they may be labeled as one object)

In this case, some grains of rice are touching, so `bwlabel` treats them as one object.

To add some color to the figure, display `labeled` using a vibrant colormap created by the `hot` function.

```
map = hot(numObjects+1); % Create a colormap.
imshow(labeled+1,map); % Offset indices to colormap by 1.
```



Here's What Just Happened

Step 8. You called `bwlabel` to search for connected components and label them with unique numbers. `bwlabel` takes a binary input image and a value of 4 or 8 to specify the "connectivity" of objects. The value 4, as used in this example, means that pixels that touch only at a corner are not considered to be "connected." For more information about the connectivity of objects, see "Connected-Components Labeling" on page 9-16.

A labeled image was returned in the form of an indexed image, where zeros represent the background, and the objects have pixel values other than zero (meaning that they are labeled). Each object is given a unique number (you can see this when you go to the next step,"9. Examine an Object"). The pixel values are indices into the colormap created by `hot`.

Your last call to `imshow` uses the syntax that is appropriate for indexed images, which is,

```
imshow(labeled+1, map);
```

Because `labeled` is an indexed image, and 0 is meaningless as an index into a colormap, a value of 1 was added to all pixels before display. The `hot` function creates a colormap of the size you specify. We created a colormap with one more color than there are objects in the image because the first color is used for the background. MATLAB has several colormap-creating functions, including `gray`, `pink`, `copper`, and `hsv`. For information on these functions, see `colormap` in the MATLAB *Function Reference*.

You can also return the number of objects by asking for the maximum pixel value in the image. For example,

```
max(labeled(:))
ans =

   80
```

9. Examine an Object

You may find it helpful to take a closer look at `labeled` to see what `bwlabel` has done to it. Use the `imcrop` command to select and display pixels in a region of `labeled` that includes an object and some background.

To ensure that the output is displayed in the MATLAB window, do not end the line with a semicolon. In addition, choose a small rectangle for this exercise, so that the displayed pixel values don't wrap in the MATLAB command window.

The syntax shown below makes `imcrop` work interactively. Your mouse cursor becomes a cross-hair when placed over the image. Click at a position in `labeled` where you would like to select the upper left corner of a region. Drag the mouse to create the selection rectangle, and release the button when you are done.

```
grain=imcrop(labeled) % Crop a portion of labeled.
```

We chose the left edge of a grain and got the following results.

```
grain =

     0     0     0     0     0     0     0    60    60
     0     0     0     0     0    60    60    60    60
     0     0     0    60    60    60    60    60    60
     0     0     0    60    60    60    60    60    60
     0     0     0    60    60    60    60    60    60
     0     0     0    60    60    60    60    60    60
     0     0     0    60    60    60    60    60    60
     0     0     0     0     0    60    60    60    60
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
```

Here's What Just Happened

Step 9. You called `imcrop` and selected a portion of the image that contained both some background and part of an object. The pixel values were returned in the MATLAB window. If you examine the results above, you can see the corner of an object labeled with 60's, which means that it was the 60th object labeled by `bwlabel`. Notice how the 60's create an edge amidst a background of 0's.

```
0    0    0    0    0    60   60
0    0    0    60   60   60   60
0    60   60   60   60   60   60
0    60   60   60   60   60   60
0    60   60   60   60   60   60
0    60   60   60   60   60   60
0    60   60   60   60   60   60
0    0    0    60   60   60   60
0    0    0    0    0    0    0
0    0    0    0    0    0    0
```

`imcrop` can also take a vector specifying the coordinates for the crop rectangle. In this case, it does not operate interactively. For example, this call specifies a crop rectangle whose upper-left corner begins at (15, 25) and has a height and width of 10.

```
rect = [15 25 10 10];
roi = imcrop(labeled, rect)
```

You are not restricted to rectangular regions of interest. The toolbox also has a `roipoly` command that enables you to select polygonal regions of interest. Many image processing operations can be performed on regions of interest, including filtering and filling. See Chapter 10, "Region-Based Processing" for more information.

10. Compute Feature Measurements of Objects in the Image

The `imfeature` command computes feature measurements for objects in an image and returns them in a structure array. When applied to an image with labeled components, it creates one structure element for each component. Use `imfeature` to create a structure array containing some basic types of feature information for `labeled`.

```
grain=imfeature(labeled,'basic')
```

MATLAB responds with

```
grain =

80x1 struct array with fields:
    Area
    Centroid
    BoundingBox
```

Find the area of the grain labeled with 51's, or "grain 51." To do this, use dot notation to access the data in the `Area` field. Note that structure field names are case sensitive, so you need to capitalize the name as shown.

```
grain(51).Area
```

returns the following results

```
ans =

   323
```

Find the smallest possible bounding box and the centroid (center of mass) for grain 51.

```
grain(51).BoundingBox, grain(51).Centroid
```

returns

```
ans =

  141.5000   89.5000   26.0000   27.0000
ans =

  155.3437  102.0898
```

Create a new vector, `allgrains`, which holds just the area measurement for each grain. Then call the `whos` command to see how `allgrains` is allocated in the MATLAB workspace.

```
allgrains=[grain.Area];
```

```
whos allgrains
```

MATLAB responds with

```
  Name            Size         Bytes  Class

  allgrains       1x80           640  double array

Grand total is 80 elements using 640 bytes
```

allgrains contains a one-row array of 80 elements, where each element contains the area measurement of a grain. Check the area of the 51st element of allgrains.

```
allgrains(51)
```

returns

```
ans =

   323
```

which is the same result that you received when using dot notation to access the Area field of grains(51).

Here's What Just Happened

Step 10. You called `imfeature` to return a structure of basic feature measurements for each thresholded grain of rice. `imfeature` supports many types of feature measurement, but setting the `measurements` parameter to `basic` is a convenient way to return three of the most commonly used measurements: the area, the centroid (or center of mass), and the bounding box. The bounding box represents the smallest rectangle that can contain a region, or in this case, a grain. The four-element vector returned by the `BoundingBox` field,

```
[141.5000   89.5000   26.0000   27.0000]
```

shows that the upper left corner of the bounding box is positioned at `[141.5 89.5]`, and the box has a width of `26.0` and a height of `27.0`. (The position is defined in spatial coordinates, hence the decimal values. For more information on the spatial coordinate system, see "Spatial Coordinates" on page 2-22.) For more information about working with MATLAB structure arrays, see "Structures" in the MATLAB graphics documentation.

You used dot notation to access the `Area` field of all of the elements of `grain` and stored this data to a new vector `allgrains`. This step simplifies analysis made on area measurements because you do not have to use field names to access the area.

11. Compute Statistical Properties of Objects in the Image

Now use MATLAB functions to calculate some statistical properties of the thresholded objects. First use `max` to find the size of the largest grain. (If you have followed all of the steps in this exercise, the "largest grain" is actually two grains that are touching and have been labeled as one object).

```
max(allgrains)
```

returns

```
ans =

   749
```

Use the `find` command to return the component label of this large-sized grain.

```
biggrain=find(allgrains==749)
```

returns

```
biggrain =

    68
```

Find the mean grain size.

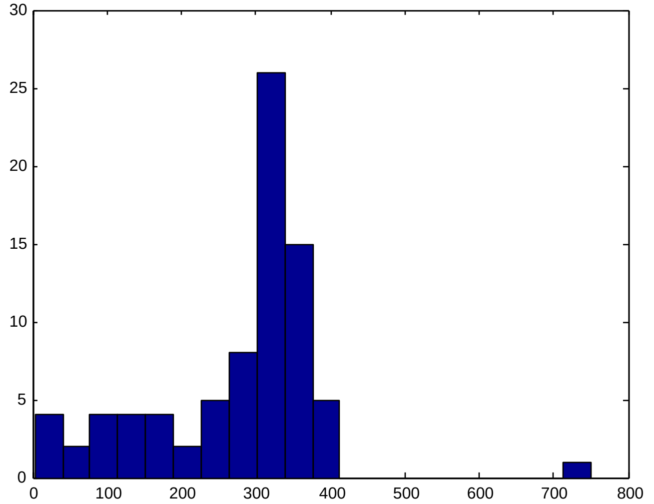
```
mean(allgrains)
```

returns

```
ans =

  275.8250
```

Make a histogram containing 20 bins that show the distribution of rice grain sizes.

```
hist(allgrains,20)
```



Here’s What Just Happened

Step 11. You used some of MATLAB’s statistical functions, `max`, `mean`, and `hist` to return the statistical properties for the thresholded objects in `rice.tif`.

The Image Processing Toolbox also has some statistical functions, such as `mean2` and `std2`, which are well suited to image data because they return a single value for two-dimensional data. The functions `mean` and `std` were suitable here because the data in `allgrains` was one dimensional.

The histogram shows that the most common sizes for rice grains in this image are in the range of 300 to 400 pixels.

Where to Go From Here

For more information about how the toolbox handles image types and storage classes, or to learn more about reading and writing images, see Chapter 2, “Introduction.” For instructions on displaying images of all types, please see Chapter 3, “Displaying and Printing Images.”

Tutorial discussions for image processing operations are contained in the chapters starting with Chapter 5, “Neighborhood and Block Operations.” Those chapters assume that you understand the information presented in the chapters starting with this chapter through Chapter 4, “Geometric Operations.”

The reference pages for all of the Image Processing Toolbox functions are contained in Chapter 12, “Function Reference.” These complement the M-file help that is displayed in the MATLAB command window when you type

```
help functionname
```

For example,

```
help imshow
```

Online Help

The *Image Processing Toolbox User’s Guide* is available online in both HTML and PDF formats. To access the HTML help, select **Help** from **View** menu of the MATLAB desktop. Then, from the left pane of the Help browser, expand the topic list next to **Image Processing Toolbox**. To access the PDF help, click on **Image Processing Toolbox** in the **Contents** tab of the Help browser, and go to the hyperlink under “Printable Documentation (PDF).” (Note that to view the PDF help, you must have Adobe's Acrobat Reader installed.)

Toolbox Demos

Some features of the Image Processing Toolbox are implemented in demo applications. The demos are useful for seeing the toolbox features put into action and for borrowing code for your own applications. See “Image Processing Demos” in the Preface for a complete list and summary of the demos, as well as instructions on how to run them.

2

Introduction

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.

Words You Need to Know

An understanding of the following terms will help you to use this chapter. For more explanation of this table and others like it, see "Words You Need to Know" in the Preface.

Words	Definitions
Binary image	An image containing only black and white pixels. In MATLAB, a binary image is represented by a `uint8` or `double` *logical* matrix containing 0's and 1's (which usually represent black and white, respectively). A matrix is logical when its "logical flag" is turned "on." We often use the variable name `BW` to represent a binary image in memory.
Image type	The defined relationship between array values and pixel colors. The toolbox supports binary, indexed, intensity, and RGB image types.
Indexed image	An image whose pixel values are direct indices into an RGB colormap. In MATLAB, an indexed image is represented by an array of class `uint8`, `uint16`, or `double`. The colormap is always an m-by-3 array of class `double`. We often use the variable name `X` to represent an indexed image in memory, and `map` to represent the colormap.

Words	Definitions
Intensity image	An image consisting of intensity (grayscale) values. In MATLAB, intensity images are represented by an array of class `uint8`, `uint16`, or `double`. While intensity images are not stored with colormaps, MATLAB uses a system colormap to display them. We often use the variable name `I` to represent an intensity image in memory. This term is synonymous with the term “grayscale.”
Multiframe image	An image file that contains more than one image, or *frame*. When in MATLAB memory, a multiframe image is a 4-D array where the fourth dimension specifies the frame number. This term is synonymous with the term “multipage image.”
RGB image	An image in which each pixel is specified by three values — one each for the red, blue, and green components of the pixel’s color. In MATLAB, an RGB image is represented by an m-by-n-by-3 array of class `uint8`, `uint16`, or `double`. We often use the variable name `RGB` to represent an RGB image in memory.
Storage class	The numeric storage class used to store an image in MATLAB. The storage classes used in MATLAB are `uint8`, `uint16`, and `double`. Some function descriptions in the reference chapter of this User’s Guide have a section entitled “Class Support” that specifies which image classes the function can operate on. When this section is absent, the function can operate on all supported storage classes.

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 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. Some images, such as RGB, require a three-dimensional array, where the first plane in the third dimension represents the red pixel intensities, the second plane represents the green pixel intensities, and the third plane represents the blue pixel intensities.

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`.

Storage Classes in the Toolbox

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` and `uint16`. The data in these arrays is stored as 8-bit or 16-bit unsigned integers. These arrays require one eighth or one fourth as much memory as `double` arrays.

Image Types in the Toolbox

The Image Processing Toolbox supports four basic types of images.

- Index 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 a data matrix, `X`, and a colormap matrix, `map`. `X` can be of class `uint8`, `uint16`, or `double`. `map` is an m-by-3 array of class `double` containing floating-point values in the range [0,1]. Each row of `map` specifies the red, green, and blue components of a single color. An indexed image uses "direct mapping" of pixel values to colormap values. The color of each image pixel is determined by using the corresponding value of `X` as an index into `map`. The value 1 points to the first row in `map`, the value 2 points to the second row, and so on.

A colormap is often stored with an indexed image and is automatically loaded with the image when you use the `imread` function. However, you are not limited to using the default colormap—you can use any colormap that you choose. 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.

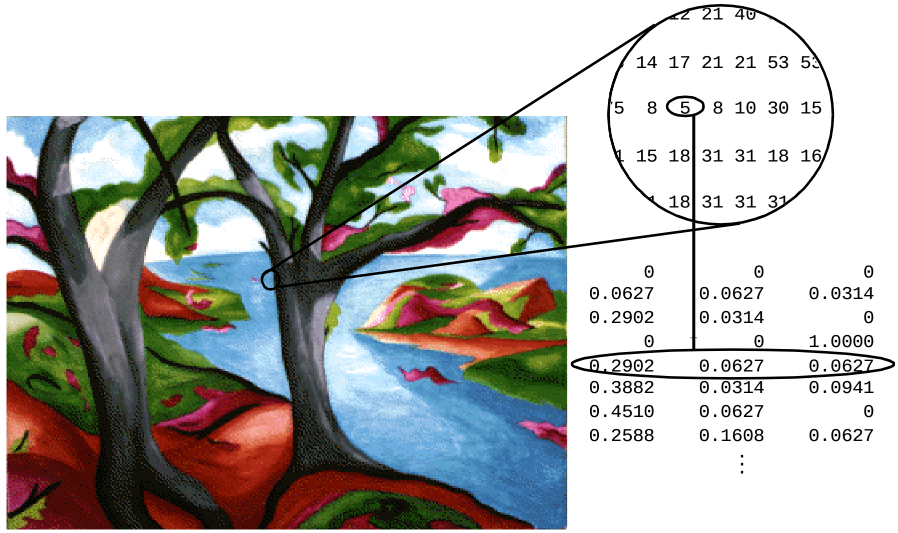


Figure 2-1: Pixel Values Are Indices to a Colormap in Indexed Images

The relationship between the values in the image matrix and the colormap depends on the class of the image matrix. 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` or `uint16`, 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 also used in graphics file formats to maximize the number of colors that can be supported. In the image above, the image matrix is of class `double`. Because there is no offset, the value 5 points to the fifth row of the colormap.

Note that the toolbox provides limited support for indexed images of class `uint16`. You can read these images into MATLAB and display them, but before you can process a `uint16` indexed image you must first convert it to either a `double` or a `uint8`. To convert to a `double`, call `im2double`; to reduce the image to 256 colors or fewer (`uint8`) call `imapprox`. For more information, see the reference pages for `im2double` and `imapprox`.

Intensity Images

An intensity image is a data matrix, `I`, whose values represent intensities within some range. 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`, `uint8`, or `uint16`. While intensity images are rarely saved with a colormap, MATLAB uses a colormap to display them.

The elements in the intensity matrix represent various intensities, or gray levels, where the intensity 0 usually represents black and the intensity 1, 255, or 65535 usually represents full intensity, or white.

The figure below depicts an intensity image of class `double`.

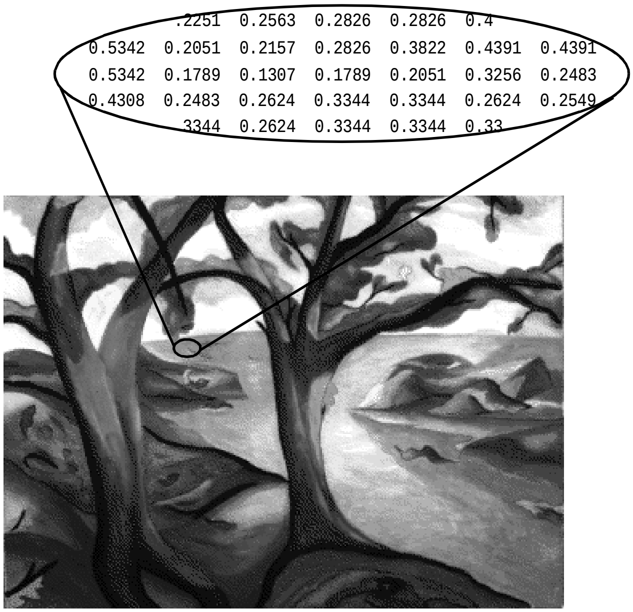


Figure 2-2: Pixel Values in an Intensity Image Define Gray Levels

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`. (The toolbox does not support binary images of class `uint16`.) An array of class `uint8` is generally preferable to an array of class `double`, because a `uint8` array 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 a logical flag to indicate the data range of a `uint8` logical array: if the logical flag is "on" the data range is [0,1]; if the logical flag is off, the toolbox assumes the data range is [0,255].)

The figure below depicts a binary image.

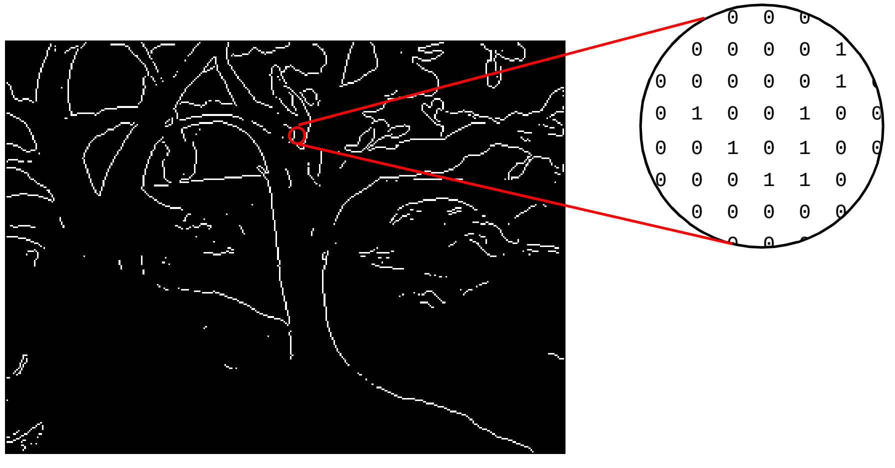


Figure 2-3: Pixels in A Binary Image Have Two Possible Values: 0 or 1

RGB Images

An RGB image, sometimes referred to as a "truecolor" image, is stored in MATLAB as an m-by-n-by-3 data array that defines red, green, and blue color components for each individual pixel. RGB images do not use a palette. The color of each pixel is determined by the combination of the red, green, and blue intensities stored in each color plane at the pixel's location. Graphics file formats store RGB images as 24-bit images, where the red, green, and blue components are 8 bits each. This yields a potential of 16 million colors. The precision with which a real-life image can be replicated has led to the commonly used term “truecolor image.”

An RGB MATLAB array can be of class `double`, `uint8`, or `uint16`. In an RGB array of class `double`, each color component is a value between 0 and 1. A pixel whose color components are (0,0,0) displays as black, and a pixel whose color components are (1,1,1) displays as white. The three color components for each pixel are stored along the third dimension of the data array. For example, the red, green, and blue color components of the pixel (10,5) are stored in `RGB(10,5,1)`, `RGB(10,5,2)`, and `RGB(10,5,3)`, respectively.

Figure 2-4 depicts an RGB image of class `double`.

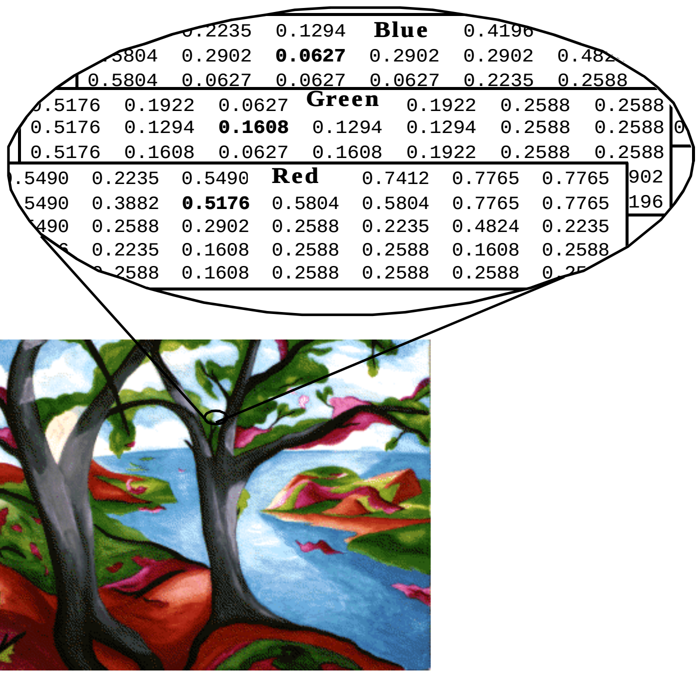


Figure 2-4: The Color Planes of an RGB Image

To determine the color of the pixel at (2,3), you would look at the RGB triplet stored in (2,3,1:3). Suppose (2,3,1) contains the value `0.5176`, (2,3,2) contains `0.1608`, and (2,3,3) contains `0.0627`. The color for the pixel at (2,3) is

```
0.5176 0.1608 0.0627
```

To further illustrate the concept of the three separate color planes used in an RGB image, the code sample below creates a simple RGB image containing uninterrupted areas of red, green, and blue, and then creates one image for each of its separate color planes (red, green, and blue). It displays each color plane image separately, and also displays the original image.

```
RGB=reshape(ones(64,1)*reshape(jet(64),1,192),[64,64,3]);
R=RGB(:,:,1);
G=RGB(:,:,2);
B=RGB(:,:,3);
imshow(R)
figure, imshow(G)
figure, imshow(B)
figure, imshow(RGB)
```

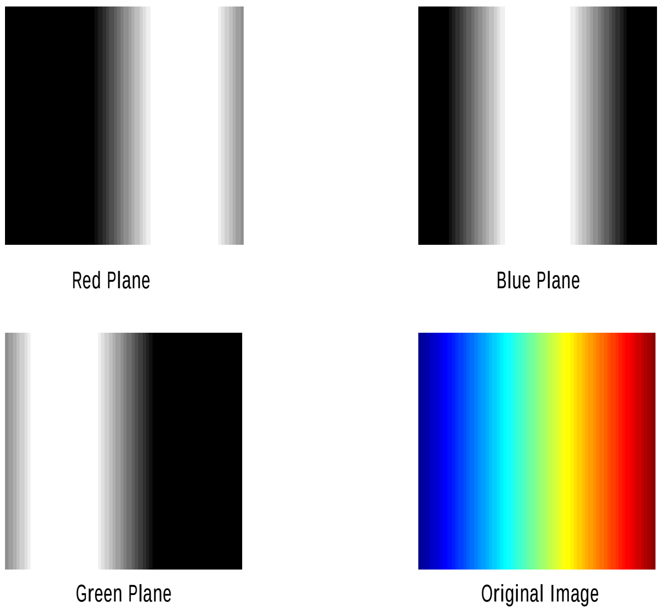


Figure 2-5: The Separated Color Planes of an RGB Image

Notice that each separated color plane in the figure contains an area of white. The white corresponds to the highest values (purest shades) of each separate color. For example, in the "Red Plane" image, the white represents the highest concentration of pure red values. As red becomes mixed with green or blue, gray pixels appear. The black region in the image shows pixel values that contain no red values, i.e., `R == 0`.

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.

Use the `cat` command to store separate images into one multiframe file. 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.

Multiframe Support 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 pages for information about how individual functions work with the dimensions of an image array. For more information about displaying multiframe images, see Chapter 3, “Displaying and Printing Images.”

Summary of Image Types and Numeric Classes

This table summarizes the way MATLAB interprets data matrix elements as pixel colors, depending on the image type and storage class.

Image Type	Class double	Class uint8 or uint16
Binary	Image is an m-by-n array of integers in the range [0,1] where the logical flag is on.	Image is an m-by-n array of integers in the range [0,1] where the logical flag is on.
Indexed	Image is an m-by-n array of integers in the range [1, p]. Colormap is a p-by-3 array of floating-point values in the range [0, 1].	Image is an m-by-n array of integers in the range [0, p – 1]. Colormap is a p-by-3 array of floating-point values in the range [0, 1].

Image Type	Class double	Class uint8 or uint16
Intensity	Image is an m-by-n array of floating-point values that are linearly scaled by MATLAB to produce colormap indices. The typical range of values is [0, 1]. Colormap is a *p*-by-3 array of floating-point values in the range [0, 1] and is typically grayscale.	Image is an m-by-n array of integers that are linearly scaled by MATLAB to produce colormap indices. The typical range of values is [0, 255] or [0, 65535]. Colormap is a *p*-by-3 array of floating-point values in the range [0, 1] and is typically grayscale.
RGB (Truecolor)	Image is an m-by-n-by-3 array of floating-point values in the range [0, 1].	Image is an m-by-n-by-3 array of integers in the range [0, 255] or [0, 65535].

Working with Image Data

MATLAB provides commands for reading, writing, and displaying several types of graphics file formats images. As with MATLAB-generated images, once a graphics file format image is displayed, it becomes a Handle Graphics® Image object. MATLAB supports the following graphics file formats:

- BMP (Microsoft Windows Bitmap)
- HDF (Hierarchical Data Format)
- JPEG (Joint Photographic Experts Group)
- PCX (Paintbrush)
- PNG (Portable Network Graphics)
- TIFF (Tagged Image File Format)
- XWD (X Window Dump)

For the latest information concerning the bit depths and/or image types supported for these formats, see the reference pages for `imread` and `imwrite`.

This section discusses how to read, write, and work with graphics images. It also describes how to convert the storage class or graphics format of an image.

Reading a Graphics Image

The function `imread` reads an image from any supported graphics image file in any of the supported bit depths. Most of the images that you will read are 8-bit. When these are read into memory, MATLAB stores them as class `uint8`. The main exception to this rule is that MATLAB supports 16-bit data for PNG and TIFF images. If you read a 16-bit PNG or TIFF image, it will be stored as class `uint16`.

Note For indexed images, `imread` always reads the colormap into an array of class `double`, even though the image array itself may be of class `uint8` or `uint16`.

To see the many syntax variations for reading an image, see the reference entry for `imread`. For our discussion here we will show one of the most basic syntax uses of `imread`. This example reads the image `ngc6543a.jpg`.

```
RGB = imread('ngc6543a.jpg');
```

You can write image data using the imwrite function. The statements

```
load clown
imwrite(X,map,'clown.bmp')
```

create a BMP file containing the clown image.

Writing a Graphics Image

The function imwrite writes an image to a graphics file in one of the supported formats. If the image is of class uint8 and the format you choose supports 8-bit images, by default the image is written as 8-bit. If the image is of class double, MATLAB's default behavior is to scale the data to class uint8 before writing it to file, since most graphics file format images do not support double-precision data. When the image is of class uint16, there are two possible default outcomes: if you write an image of class uint16 to a format that supports 16-bit images (PNG or TIFF), it is written as a 16-bit file; if you write to a format that does not support 16-bit files, MATLAB first scales the data to class uint8, as it does for images of class double.

The most basic syntax for imwrite takes the image variable name and a filename. If you include an extension in the filename, MATLAB infers the desired file format from it. This example writes an RGB image RGB to a BMP file.

```
imwrite(RGB,'myfile.bmp');
```

For some graphics formats, you can specify additional parameters. One of the additional parameters for PNG files sets the bit depth. This example writes an intensity image I to a 4-bit PNG file.

```
imwrite(I,'clown.png','BitDepth',4);
```

(The bit depths and image types supported for each format are shown in the reference pages for imwrite.)

This example writes an image A to a JPEG file with a compression quality setting of 100 (the default is 75).

```
imwrite(A, 'myfile.jpg', 'Quality', 100);
```

See the reference entry for imwrite for more information.

Querying a Graphics File

The `imfinfo` function enables you to obtain information about graphics files that are in any of the formats supported by the toolbox. The information you obtain depends on the type of file, but it always includes at least the following:

- Name of the file, including the directory path if the file is not in the current directory
- File format
- Version number of the file format
- File modification date
- File size in bytes
- Image width in pixels
- Image height in pixels
- Number of bits per pixel
- Image type: RGB (truecolor), intensity (grayscale), or indexed

See the reference entry for `imfinfo` for more information.

Converting The Image Type of Images

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, `ind2gray` 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 pages.

The table below summarizes these image conversion functions.

Function	Purpose
`dither`	Create a binary image from a grayscale intensity image by dithering; create an indexed image from an RGB image by dithering
`gray2ind`	Create an indexed image from a grayscale intensity image
`grayslice`	Create an indexed image from a grayscale intensity image by thresholding
`im2bw`	Create a binary image from an intensity image, indexed image, or RGB image, based on a luminance threshold
`ind2gray`	Create a grayscale intensity image from an indexed image
`ind2gray`	Create an RGB image from an indexed image
`mat2gray`	Create a grayscale intensity image from data in a matrix, by scaling the data
`rgb2gray`	Create a grayscale intensity image from an RGB image
`rgb2ind`	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 11, "Color."

Working with uint8 and uint16 Data

Use `imread` to read graphics images into MATLAB as `uint8` or `uint16` arrays; use `imshow` to display these images; and use `imwrite` to save these images. Most of the functions in the Image Processing Toolbox accept `uint8` and `uint16` input. See the reference entries for more information about `imread`, `imshow`, `uint8`, and `uint16`.

MATLAB provides limited support for storing images as 8-bit or 16-bit unsigned integers. In addition to reading and writing `uint8` and `uint16` arrays, MATLAB supports the following operations:

- 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
- The `find` function. Note that the returned array is of class `double`.

Mathematical Operations Support for uint8 and uint16

The following MATLAB mathematical operations support `uint8` and `uint16` data: `conv2`, `convn`, `fft2`, `fftn`, `sum`. In these cases, the output is always `double`.

If you attempt to perform an unsupported operation on one of these arrays, you will receive an error. For example,

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

Converting The Storage Class of Images

If you want to perform operations that are not supported for `uint8` or `uint16` arrays, you can convert the data to double precision using the MATLAB function, `double`. For example,

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

However, converting between storage classes 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.

For easier conversion of storage classes, use one of these toolbox functions: `im2double`, `im2uint8`, and `im2uint16`. 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);
```

Note that when you convert from one class to another that uses fewer bits to represent numbers, you generally lose some of the information in your image. For example, a `uint16` intensity image is capable of storing up to 65,536 distinct shades of gray, but a `uint8` intensity image can store only 256 distinct shades of gray. If you convert a `uint16` intensity image to a `uint8` intensity image, `im2uint8` must *quantize* the gray shades in the original image. In other words, all values from 0 to 128 in the original image become 0 in the `uint8` image, values from 129 to 385 all become 1, and so on. This loss of information is often not a problem, however, since 256 still exceeds the number of shades of gray that your eye is likely to discern.

In an indexed image, the image matrix contains only indices into a colormap, rather than the color data itself, so there is no quantization of the color data possible during the conversion. Therefore, it is not always possible to convert an indexed image from one storage class to another. For example, a `uint16` or `double` indexed image with 300 colors cannot be converted to `uint8`, because `uint8` arrays have only 256 distinct values. If you want to perform this conversion, you must first reduce the number of the colors in the image using the `imapprox` function. This function performs the quantization on the colors in the colormap, to reduce the number of distinct colors in the image. See "Using imapprox" on page 11-12 for more information. For more information on the storage class conversion functions see the reference pages for `im2double`, `im2uint8`, `im2uint16`.

Turning the Logical Flag on or off

As discussed in "Binary Images" on page 2-7, a `uint8` binary image must have its logical flag on. If you use `im2uint8` to convert a binary image of class `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;
```

Converting the Graphics File Format of an Image

Sometimes you will want to change the graphics format of an image, perhaps for compatibility with another software product. You can do this by reading in the image with `imread`, and then calling `imwrite` with the appropriate format setting specified. For example, to convert an image from a BMP to a PNG, read the BMP image using `imread`, convert the storage class if necessary, and then write the image using `imwrite`, with `'PNG'` specified as your target format. For the specifics of which bit depths are supported for the different graphics formats, and for how to specify the format type when writing an image to file, see the reference entries for `imread` and `imwrite`.

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 two coordinate systems are described in

- "Pixel Coordinates"
- "Spatial Coordinates"

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, as illustrated by Figure 2-6.

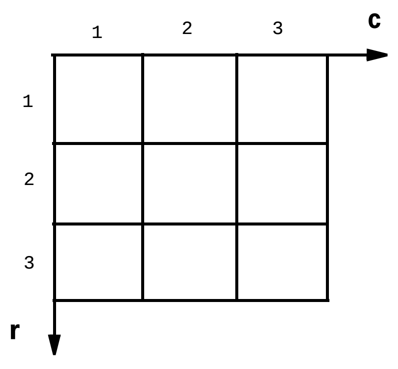


Figure 2-6: The Pixel Coordinate System

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. 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` (not `r` and `c` as in the pixel coordinate system).

Figure 2-7 illustrates the spatial coordinate system used for images. Notice that `y` increases downward.

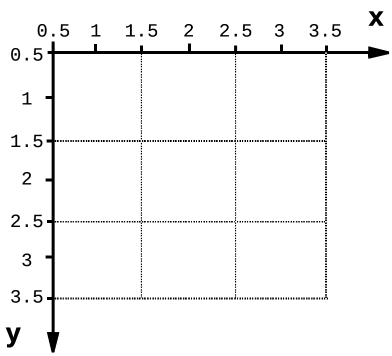


Figure 2-7: The Spatial Coordinate System

This spatial coordinate system corresponds 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 2-23 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. As mentioned earlier, pixel coordinates are expressed as (r,c), while spatial coordinates are expressed as (x,y). In the reference pages, 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.

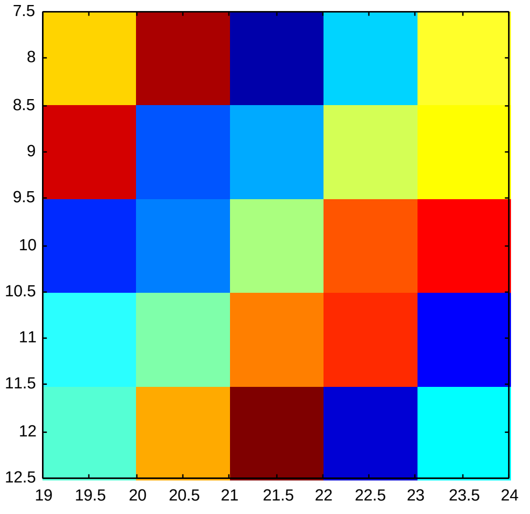
In some situations, however, you may want to use a nondefault spatial coordinate system. For example, you could specify that the upper-left corner of an image is the point (19.0,7.5), rather than (0.5,0.5). If you call a function that returns coordinates for this image, the coordinates returned will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

3

Displaying and Printing Images

Overview

The Image Processing Toolbox supports a number of image display techniques. For example, the function `imshow` displays any supported image type with a single function call. Other functions handle more specialized display needs. This chapter describes basic display techniques for each image type supported by the toolbox (e.g., RGB, intensity, and so on.), as well as how to set the toolbox preferences for the `imshow` function. It also discusses special display techniques, such as multiple image display and texture mapping. The final pages of this chapter include information about printing images and troubleshooting display problems

Words You Need to Know

An understanding of the following terms will help you to use this chapter. For more explanation of this table and others like it, see “Words You Need to Know” in the Preface.

Words	Definitions
Color approximation	There are two ways in which this term is used in MATLAB: • The method by which MATLAB chooses the best colors for an image whose number of colors you are decreasing • MATLAB’s automatic choice of screen colors when displaying on a system with limited color display capability
Screen bit depth	The number of bits per screen pixel
Screen color resolution	The number of distinct colors that can be produced by the screen

Displaying Images with imshow

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 called `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, using `imshow` for image processing applications is preferable to using `image` and `imagesc`. It is easier to use and in most cases, displays an image using one image pixel per screen pixel. (For more information about `image` and `imagesc`, see their pages in the MATLAB Function Reference or see the MATLAB graphics documentation.)

Note One of the most common toolbox usage errors is using the wrong syntax of `imshow` for your image type. This chapter shows which syntax is appropriate for each type of image. If you need help determining what type of image you are working with, see "Image Types in the Toolbox" on page 2-5.

Displaying Indexed Images

To display an indexed image with `imshow`, 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`. The relationship between the values in the image matrix and the colormap depends on whether the image matrix is of class `double`, `uint16`, 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` or `uint16`, 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.)

Each pixel in an indexed image is directly mapped to its corresponding colormap entry. If the colormap contains a greater number of colors than the image, the extra colors in the colormap will simply be ignored. If the colormap contains fewer colors than the image requires, all image pixels over the limits of the colormap's capacity will be set to the last color in the colormap, i.e., if an image of class `uint8` contains 256 colors, and you display it with a colormap that contains only 16 colors, all pixels with a value of 15 or higher are displayed with the last color in the colormap.

To change the default behavior of `imshow`, set the toolbox preferences. See "Setting the Preferences for imshow" on page 3-24 for more information.

The Image and Axes Properties of an Indexed Image

In most cases, it is not necessary to concern yourself with the Handle Graphics property settings made when you call `imshow`. Therefore, this section is not required reading, but rather information for those who really "want to know."

When you display an indexed image, `imshow` sets the Handle Graphics properties that control how colors display, as follows:

- The image `CData` property 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`.

Displaying Intensity Images

To display a intensity (grayscale) image, the most basic syntax is

```
imshow(I)
```

`imshow` displays the image by *scaling* the intensity values to serve as indices into a grayscale colormap. If `I` is `double`, a pixel value of 0.0 is displayed as black, a pixel value of 1.0 is displayed as white, and pixel values in between are displayed as shades of gray. If `I` is `uint8`, then a pixel value of 255 is displayed as white. If I is `uint16`, then a pixel value of 65535 is displayed as white.

Intensity images are similar to indexed images in that each uses an m-by-3 RGB colormap, but normally, you will not specify a colormap for an intensity image. MATLAB displays intensity images by using a grayscale system colormap (where R=G=B). By default, the number of levels of gray in the colormap is 256 on systems with 24-bit color, and 64 or 32 on other systems. (See "Working with Different Screen Bit Depths" on page 11-4 for a detailed explanation.)

Another syntax form of `imshow` for intensity images enables you to explicitly specify the number of gray levels to use. To display an image `I` with 32 gray levels, specify a value for `n`.

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

Because MATLAB scales intensity images to fill the colormap range, a colormap of any size can be used. Larger colormaps enable you to see more detail, but they also use up more color slots. The availability of color slots is discussed further in "Displaying Multiple Images" on page 3-19, and also in "Working with Different Screen Bit Depths" on page 11-4.

To change the default behavior of `imshow`, set the toolbox preferences. See "Setting the Preferences for imshow" on page 3-24 for more information.

Displaying Intensity Images That Have Unconventional Ranges

In some cases, you may have data you want to display as an intensity image, even though the data is outside the conventional toolbox range (i.e., [0,1] for `double` arrays, [0,255] for `uint8` arrays, or [0,65535] for `uint16` arrays). For example, if you filter an intensity image, some of the output data may fall outside the range of the original data.

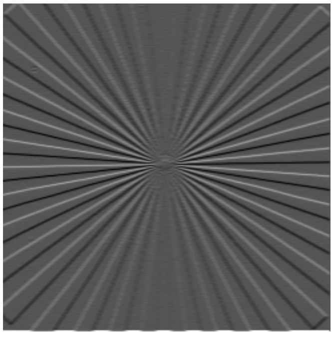
To display unconventional range data as an image, you can specify the data range directly, using

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

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. The next example filters an intensity image, creating unconventional range data. `imshow` is then called using an empty matrix.

```
I = imread('testpat1.tif');
J = filter2([1 2;-1 -2],I);
min(J(:)) %Find the minimum pixel value of the filtered image.
```

```
ans =
  -364

max(J(:)) %Find the maximum pixel value of the filtered image.

ans =
   723
imshow(J,[]);
```



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

The Image and Axes Properties of an Intensity Image

In most cases, it is not necessary to concern yourself with the Handle Graphics property settings made when you call `imshow`. Therefore, this section is not required reading, but rather information for those who really "want to know."

When you display an intensity image, `imshow` sets the Handle Graphics properties that control how colors display, as follows:

- The image `CData` property is set to the data in `I`.
- 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`, [0 255] if the matrix is of class `uint8`, or [0 65535] if it is of class `uint16`.
- The figure `Colormap` property is set to a grayscale colormap whose values range from black to white.

Displaying Binary Images

To display a binary image, the syntax is

```
imshow(BW)
```

In MATLAB, a binary image is a *logical* two-dimensional `uint8` or `double` matrix that contains only 0's and 1's. (The toolbox does not support `uint16` binary images.) All toolbox functions that return a binary image, return them as `uint8` logical arrays.

Generally speaking, working with binary images with the toolbox is very straightforward. In most cases you will load a 1-bit binary image, and MATLAB will create a logical `uint8` image in memory. You will normally not encounter `double` binary images unless you create them yourself using MATLAB.

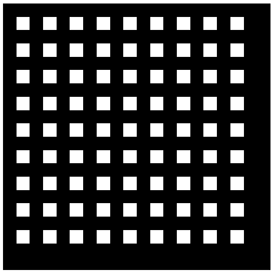
If you load an image with a bit depth greater than 1 bit per pixel, or use MATLAB to create a new `double` or `uint8` image containing only 0's and 1's, you may encounter unexpected results. For example, the mere presence of all 0's and 1's does not always indicate a binary image. For MATLAB to interpret the image as binary, it must be logical, meaning that its logical flag must be turned "on." Therefore, intensity images that happen to contain only 0's and 1's are not binary images.

To change the default behavior of `imshow`, set the toolbox preferences. See "Setting the Preferences for imshow" on page 3-24 for more information.

This example underscores the importance of having the logical flag turned on if you want your image to behave like a binary image.

Create an image of class `double` that contains only 0's and 1's.

```
BW1 = zeros(20,20);
BW1(2:2:18,2:2:18)=1;
imshow(BW1, 'notruesize');
```

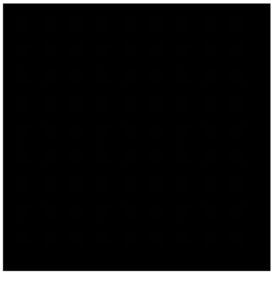


```
whos
  Name      Size         Bytes  Class
  BW1      20x20          3200  double array
Grand total is 400 elements using 3200 bytes
```

While this image may look like a binary image, it is really a grayscale image — it will not be recognized as a binary image by any toolbox function. If you were to save this image to a file (without specifying a bit depth) it would be saved as an 8-bit grayscale image containing 0's and 255's.

The fact that `BW1` is not really a binary image becomes evident when you convert it to class `uint8`.

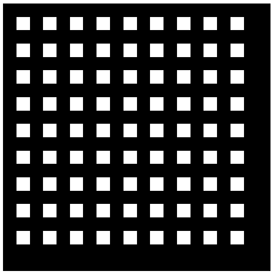
```
BW2=uint8(BW1);
figure,imshow(BW2,'notruesize')
```



`BW1` still contains 0's and 1's, but since the dynamic range of an `uint8` intensity image is [0 255], the value 1 is very close to black.

To make BW1 a true binary image use the NOT EQUAL (~=) operator, which will turn the logical flag on.

```
BW3 = BW2 ~= 0;
figure, imshow(BW3,'notruesize')
```



```
whos
  Name      Size         Bytes  Class
  BW1      20x20          3200  double array
  BW2      20x20           400  uint8 array
  BW3      20x20           400  uint8 array (logical)
Grand total is 1225 elements using 4025 bytes
```

Write BW3 to a file using one of the formats that supports writing 1-bit images. You will not need to specify a bit depth, because MATLAB automatically saves logical uint8 or double images as 1-bit images if the file format supports it.

```
imwrite(BW3, 'grid.tif'); % MATLAB supports writing 1-bit TIFFs.
```

You can check the bit depth of grid.tif by calling imfinfo. As you will see, the BitDepth field indicates that it has been saved as a 1-bit image, with the beginning of your output looking something like this.

```
imfinfo('BW1.tif')
ans =
                     Filename: 'd:\mystuff\grid.tif'
                  FileModDate: '25-Nov-1998 11:36:17'
                     FileSize: 340
                       Format: 'tif'
                FormatVersion: []
                        Width: 20
```

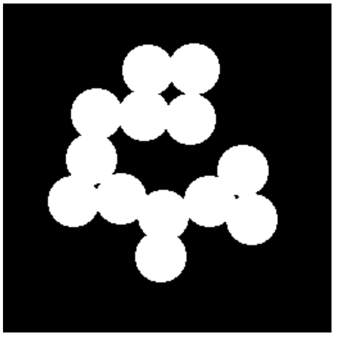
```
         Height: 20
       BitDepth: 1
      ColorType: 'grayscale'
FormatSignature: [73 73 42 0]
      ByteOrder: 'little-endian'
 NewSubfileType: 0
  BitsPerSample: 1
    Compression: 'CCITT 1D'
     ...
```

Note You may have noticed that the `ColorType` field of the binary image queried above has a value of `'grayscale'`. MATLAB sets this field to one of three values: `'grayscale'`, `'indexed'`, and `'truecolor'`. When reading an image, MATLAB evaluates the image type by checking both the `BitDepth` and the `ColorType` fields.

Changing the Display Colors of a Binary Image

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 (~) operator in MATLAB. For example,

```
BW = imread('circles.tif');
imshow(~BW)
```



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

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

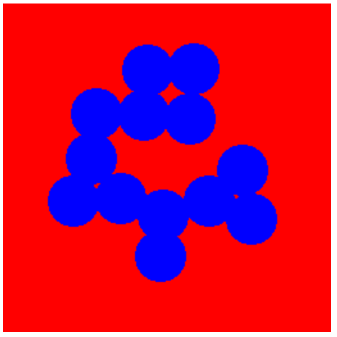


Figure 3-1: Binary Image Displayed with a Colormap

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.

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

The Image and Axes Properties of a Binary Image

In most cases, it is not necessary to concern yourself with the Handle Graphics property settings made when you call `imshow`. Therefore, this section is not required reading, but rather information for those who really "want to know."

`imshow` sets the Handle Graphics properties that control how colors display, as follows:

- The image `CData` is set to the data in `BW`.
- The image `CDataMapping` property is set to `direct`.
- The axes `CLim` property is set to [0 1].
- The figure `Colormap` property is set to a grayscale colormap whose values range from black to white.

Displaying RGB Images

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

To display an RGB image, the most basic 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`).

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 "Working with Different Screen Bit Depths" on page 11-4 for more information.

To change the default behavior of `imshow`, set the toolbox preferences. See "Setting the Preferences for imshow" on page 3-24 for more information.

The Image and Axes Properties of an RGB Image

In most cases, it is not necessary to concern yourself with the Handle Graphics property settings made when you call `imshow`. Therefore, this section is not required reading, but rather information for those who really "want to know."

When you display an RGB image, `imshow` sets the Handle Graphics properties that control how colors display, as follows.

- The image `CData` property is set to the data in `RGB`. The data will be three-dimensional. When `CData` is three-dimensional, MATLAB interprets the array as truecolor data, and ignores the values of the `CDataMapping`, `CLim`, and `Colormap` properties.
- The image `CDataMapping` property is ignored.
- The axes `CLim` property is ignored.
- The figure `Colormap` property is ignored.

Displaying Images Directly from Disk

Generally, when you want to display an image, you will first use `imread` to load it and the data is stored as one or more variables in the MATLAB workspace. However, if you do not want to load an image before displaying it, 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
```

If the image has multiple frames, `imshow` will only display the first frame. For information on the display options for multiframe images, see “Displaying Multiframe Images” on page 3-15.

This syntax is very useful for scanning through images. Note, however, that when you use this syntax, the image data is not stored in the MATLAB workspace. If you want to bring the image into the workspace, use the `getimage` function, which gets the image data from the current Handle Graphics image object. For example,

```
rgb = getimage;
```

will assign `flowers.tif` to `rgb` if the figure window in which it is displayed is currently active.

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

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 by using a colorbar is especially useful if you are displaying unconventional range data as an image, as described under "Displaying Intensity Images That Have Unconventional Ranges" on page 3-5.

In the example below, a grayscale image of class `uint8` is filtered, resulting in data that 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
```

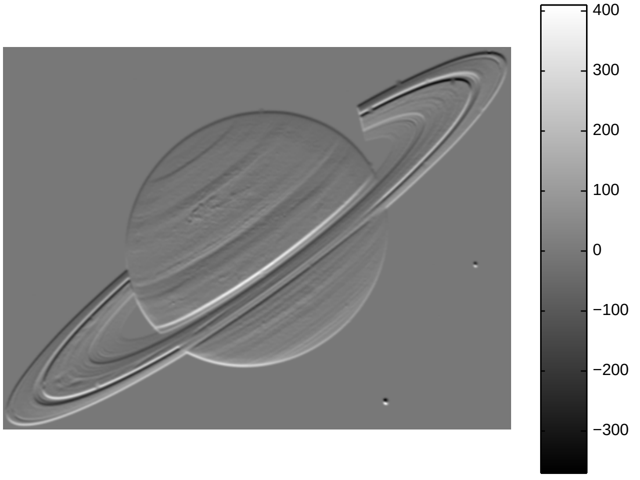


Figure 3-2: Image Displayed with Colorbar

Displaying Multiframe Images

A multiframe image is an image file that contains more than one image. The MATLAB-supported formats that enable the reading and writing of multiframe images are HDF and TIFF. See "Multiframe Image Arrays" on page 2-11 for more information about reading and writing multiframe images.

Once read into MATLAB, the image frames of a multiframe image are always handled in the fourth dimension. Multiframe images can be loaded from disk using a special syntax of `imread`, or created using MATLAB. Multiframe images can be displayed in several different ways; to display a multiframe image, you can

- Display the frames individually, using the `imshow` function. See "Displaying the Frames of a Multiframe Image Individually" on page 3-16 below.
- Display all of the frames at once, using the `montage` function. See "Displaying All Frames of a Multiframe Image at Once" on page 3-17.
- Convert the frames to a movie, using the `immovie` function. See "Converting a Multiframe Image to a Movie" on page 3-18.

Displaying the Frames of a Multiframe Image Individually

In MATLAB, the frames of a multiframe image are handled in the fourth dimension. 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))
```

The following example loads `mri.tif` and displays the third frame.

```
% Initialize an array to hold the 27 frames of mri.tif
mri = uint8(zeros(128,128,1,27));
for frame=1:27
   % Read each frame into the appropriate frame in memory
   [mri(:,:,:,frame),map] = imread('mri.tif',frame);
end
imshow(mri(:,:,:,3),map);
```

Intensity, indexed, and binary multiframe images have a dimension of m-by-n-by-1-by-k, where k represents the total number of frames, and 1 signifies that the image data has just one color plane. Therefore, the following call,

```
imshow(mri(:,:,:,3),map);
```

is equivalent to,

```
imshow(mri(:,:,1,3),map);
```

RGB multiframe images have a dimension of m-by-n-by-3-by-k, where k represents the total number of frames, and 3 signifies the existence of the three color planes used in RGB images. This example,

```
imshow(RGB(:,:,:,7));
```

shows all three color planes of the seventh frame, and is *not* equivalent to

```
imshow(RGB(:,:,3,7));
```

which shows only the third color plane (blue) of the seventh frame. These two calls will only yield the same results if the image is RGB grayscale (R=G=B).

Displaying All Frames of a Multiframe Image 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 All of the frames in a multiframe indexed array must use the same colormap.

This example loads and displays all frames of a multiframe indexed image.

```
% Initialize an array to hold the 27 frames of mri.tif.
mri = uint8(zeros(128,128,1,27));
for frame=1:27
    % Read each frame into the appropriate frame in memory.
    [mri(:,:,:,frame),map] = imread('mri.tif',frame);
end
montage(mri,map);
```

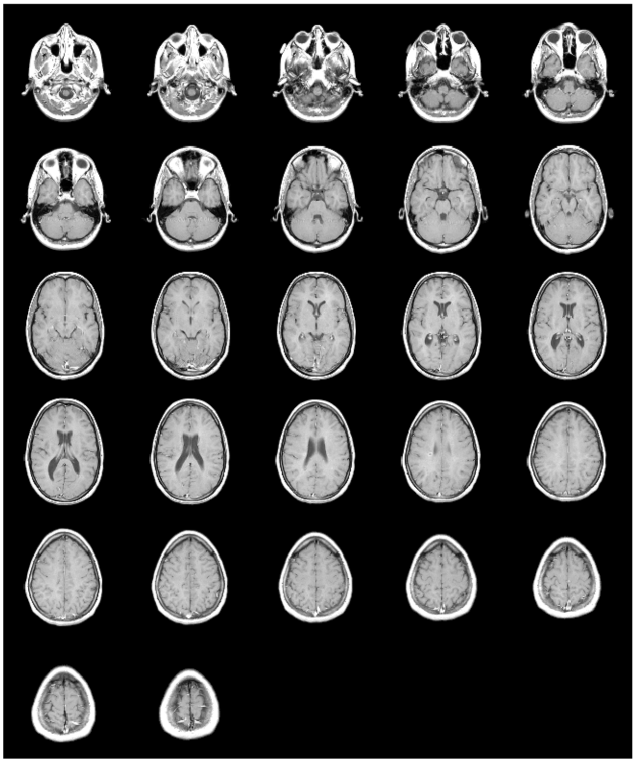


Figure 3-3: All Frames of Multiframe Image Displayed in One Figure

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.

Converting a Multiframe Image to a Movie

To create a MATLAB movie from a multiframe image array, use the `immovie` function. This function works only with indexed images; if your images are of another type, you must first convert them using one of the conversion functions described in "Converting The Image Type of Images" on page 2-16.

This call creates a movie from a multiframe indexed image X

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

where X is a four-dimensional array of structures that you want to use for the movie.

You can play the movie in MATLAB using the movie function.

```
colormap(map), movie(mov);
```

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

This example loads the multiframe image mri.tif and makes a movie out of it. It won't do any good to show the results here, so try it out; it's fun to watch.

```
% Initialize and array to hold the 27 frames of mri.tif.
mri = uint8(zeros(128,128,1,27));
for frame=1:27
   % Read each frame into the appropriate frame in memory.
   [mri(:,:,:,frame),map] = imread('mri.tif',frame);
end

mov = immovie(mri,map);
colormap(map), movie(mov);
```

Note that immovie displays the movie as it is being created, so you will actually see the movie twice. The movie runs much faster the second time (using movie).

Note MATLAB movies require MATLAB in order to be run. To make a movie that can be run outside of MATLAB, you can use the MATLAB avifile and addframe functions to create an AVI file. AVI files can be created using indexed and RGB images of classes uint8 and double, and don't require a multiframe image. For instructions on creating an AVI file, see the "Development Environment."

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 sections below describe how to display multiple figures separately, or within the same figure.

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 only display a maximum of 256 different colors, and you can therefore 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 “Working with Different Screen Bit Depths” on page 11-4 for more information.

This section discusses

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

It also includes information about working around system limitations.

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 an image in the current figure, so if you display two images in succession, the second image replaces the first image. To avoid replacing the image in the current figure, use the `figure` command to explicitly create a new empty figure before calling `imshow` for the next image. For example,

```
imshow(I)
figure, imshow(I2)
figure, imshow(I3)
```

When you use this approach, the figures you create are empty initially.

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 have identical colormaps, there will not be a problem, because only 128 color slots are used.) Remember that intensity images are also displayed using colormaps, so the color slots used by these images count toward the 256-color total.

In the next example, two indexed images are displayed on an 8-bit display. Since these images do not have similar colormaps and due to the limitation of the screen color resolution, the first image is forced to use the colormap of the second image, resulting in an inaccurate display.

```
[X1,map1]=imread('forest.tif');
[X2,map2]=imread('trees.tif');
imshow(X1,map1),figure,imshow(X2,map2);
```

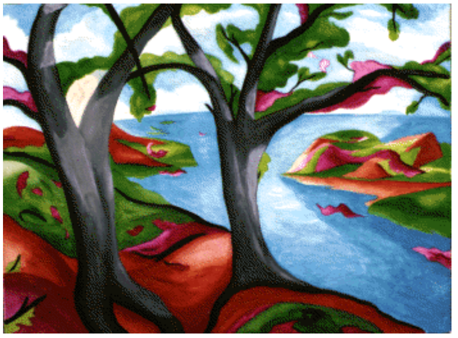


Figure 3-4: Displaying Two Indexed Images on an 8-bit Screen

As `X2` is displayed, `X1` is forced to use `X2`'s colormap (and now you can't see the forest for the trees). Note that the actual display results of this example will vary depending on what other application windows are open and using up system color slots.

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 "Reducing the Number of Colors in an Image" on page 11-6 for information.

Another solution is to convert images to RGB (truecolor) format for display, because MATLAB automatically uses dithering and color approximation to display these images. Use the `ind2rgb` function to convert indexed images to RGB.

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

Or, simply use the `cat` command 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 with some limitations. This discussion shows you how to do this in one of two ways:

1. By using `imshow` in conjunction with `subplot`

2 By using `subimage` in conjunction with `subplot`

`subplot` divides a figure into multiple display regions. The syntax of `subplot` is

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

This syntax divides the figure into an m-by-n matrix of display regions and makes the pth display region active.

For example, if you want to display two images side by side, use

```
[X1,map1]=imread('forest.tif');
[X2,map2]=imread('trees.tif');
subplot(1,2,1), imshow(X1,map2)
subplot(1,2,2), imshow(X2,map2)
```

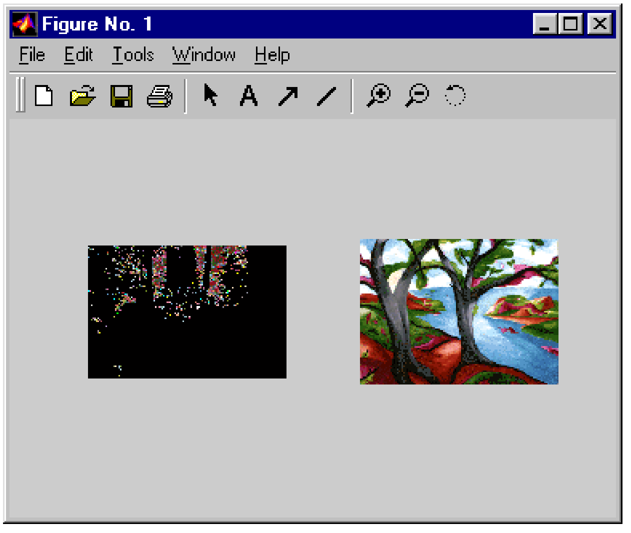


Figure 3-5: Two Images in Same Figure Using the Same Colormap

If sharing a colormap (using the `subplot` function) produces unacceptable display results as Figure 3-5 shows, use the `subimage` function (shown below). Or, as another alternative, you can map all images to the same colormap as you load them. See "Colormap Mapping" on page 11-11 for more information.

subimage converts images to RGB before displaying and therefore circumvents the colormap sharing problem. This example displays the same two images shown in Figure 3-5 with better results.

```
[X1,map1]=imread('forest.tif');
[X2,map2]=imread('trees.tif');
subplot(1,2,1), subimage(X1,map1)
subplot(1,2,2), subimage(X2,map2)
```

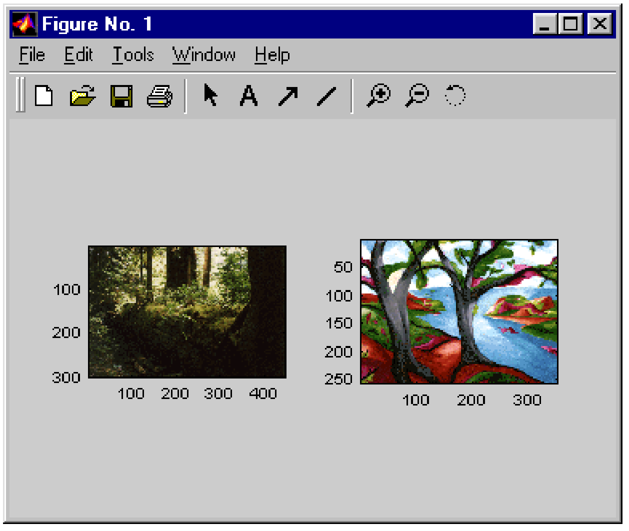


Figure 3-6: Two Images in Same Figure Using Separate Colormaps

Setting the Preferences for imshow

The behavior of imshow is influenced in part by the current settings of the *toolbox preferences*. Depending on the arguments you specify and the current settings of the toolbox preferences, imshow may

- Suppress the display of axes and tick marks.
- Include or omit a “border” around the image.
- Call the truesize function to display the image without interpolation.
- Set other figure and axes properties to tailor the display.

All of these settings can be changed by using the `iptsetpref` function, and the `truesize` preference, in particular, can also be changed by setting the `display_option` parameter of `imshow`. This section describes how to set the toolbox preferences and how to use the `display_option` parameter.

When you display an image using the `imshow` function, MATLAB also sets the Handle Graphics figure, axes, and image properties, which control the way image data is interpreted. These settings are optimized for each image type. The specific properties set are described under the following sections:

- "The Image and Axes Properties of an Indexed Image" on page 3-4
- "The Image and Axes Properties of an Intensity Image" on page 3-6
- "The Image and Axes Properties of a Binary Image" on page 3-11
- "The Image and Axes Properties of an RGB Image" on page 3-12

Toolbox Preferences

The toolbox preferences affect the behavior of `imshow` for the duration of the current MATLAB session. You can change these settings at any time by using the `iptsetpref` function. To preserve your preference settings from one session to the next, make your settings in your `startup.m` file. These are the preferences that you may set.

- The `ImshowBorder` preference controls whether `imshow` displays the figure window as larger than the image (leaving a border between the image axes and the edges of the figure), or the same size as the image (leaving no border).
- 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. This preference can be overridden for a single call to `imshow`; see "The truesize Function" below for more details.
- The `TrueSizeWarning` preference controls whether you will receive a warning message if an image is too large for the screen.

This example call to `iptsetpref` resizes the figure window so that it fits tightly around displayed images.

```
iptsetpref('ImshowBorder', 'tight');
```

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

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

The truesize Function

The `truesize` function assigns a single screen pixel to each image pixel, e.g., a 200-by-300 image will be 200 screen pixels in height and 300 screen pixels in width. This is generally the preferred way to display an image. In most situations, when the toolbox is operating under default behavior, `imshow` calls the `truesize` command automatically before displaying an image.

In some cases, you may not want `imshow` to automatically call `truesize` (for example, if you are working with a small image). If you display an image without calling `truesize`, the image displays at the default axis size. In such cases, MATLAB must use *interpolation* to determine the values for screen pixels that do not directly correspond to elements in the image matrix. (See "Interpolation" on page 4-4 for more information.)

There are two ways to affect whether or not MATLAB will automatically call `truesize`:

1. Set the preference for the current MATLAB session. This example sets the `ImshowTruesize` preference to `'manual'`, meaning that `truesize` will not be automatically called by `imshow`.

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

2. Set the preference for a single `imshow` command by setting the `display_option` parameter. This example sets the `display_option` parameter to `truesize`, so that `truesize` is called for the image displayed, regardless of the current preference setting.

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

For more information see the reference descriptions for `imshow` and `truesize`.

Zooming in on a Region of an Image

The simplest way to zoom in on a region of an image is to use the zoom buttons provided on the figure window. To enable zooming from the command line, use the `zoom` command.When you zoom in, the figure window 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.)

Once zooming in is enabled, there are two ways to zoom in on an image:

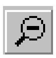
1. Single mouse click: click on a spot in the image by placing the cursor on the spot and the pressing the left mouse button. The image is magnified and the center of the new view is the spot where you clicked.

2. Click and drag the mouse: select a region by clicking on the image, holding down the left mouse button, and dragging the mouse. This creates a dotted rectangle. When you release the mouse button, the region enclosed by the rectangle is displayed with magnification.

Zooming In or Out With the Zoom Buttons

The zoom buttons in the MATLAB figure enable you to zoom in or out on an image using your mouse.

To zoom in, click the “magnifying glass” button with the plus sign in it. There are two ways to zoom in on an image after selecting the zoom in button. See “Zooming in on a Region of an Image” above.



To zoom out, click the “magnifying glass” button with the minus sign in it. Click your left mouse button over the spot in the image you would like to zoom out from.

Zooming In or Out from the Command Line

The `zoom` command enables you to zoom in or out on an image using your mouse.

To enable zooming (in or out), type

```
zoom on
```

There are two ways to zoom in on an image. See “Zooming in on a Region of an Image” above.

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 disable zooming, enter

```
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 map an image onto 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 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);
```

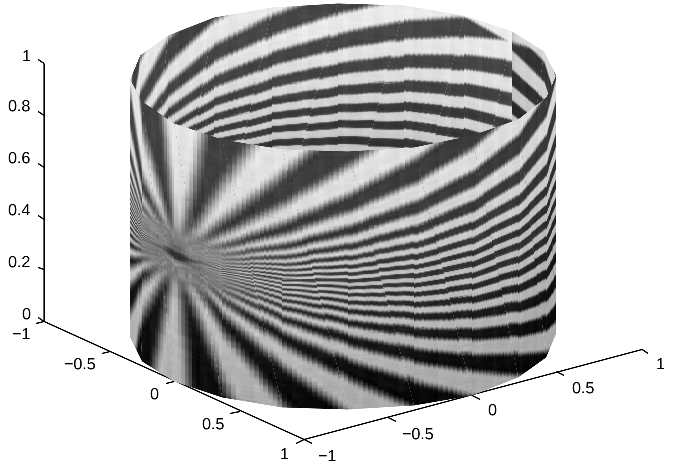


Figure 3-7: An Image Texture Mapped onto a Cylinder

The image may not map onto the surface in the way that you had expected. One way to modify the way the texture map appears is to change the settings of the `Xdir`, `Ydir`, and `Zdir` properties. For more information, see Changing Axis Direction in the MATLAB graphics documentation.

For more information about texture mapping, see the reference entry for the `warp` function.

Printing Images

If you want to output a MATLAB image to use in another application (such as a word-processing program or graphics editor), use `imwrite` to create a file in the appropriate format. See "Writing a Graphics Image" on page 2-15 for details.

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. Note that if you produce output in either of these ways, 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 are not affected by the `InvertHardcopy` figure 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`, you can add this line to your `startup.m` file.

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

For detailed information about printing with **File/Print** or the `print` command (and for information about Handle Graphics), see "Printing and Exporting Figures with MATLAB" in in the MATLAB graphics documentation. For a complete list of options for the `print` command, enter `help print` at the MATLAB command line prompt or see `print` in the MATLAB Function Reference.

Troubleshooting

This section contains three common scenarios (in bold text) which can occur unexpectedly, and what you can do to derive the expected results.

My color image is displaying as grayscale. Your image must be an indexed image, meaning that it should be displayed using a colormap. Perhaps you did not use the correct syntax for loading an indexed image, which is,

```
[X, map]=imread('filename.ext');
```

Also, be sure to use the correct form of `imshow` for an indexed image.

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

See "Displaying Indexed Images" on page 3-3 for more information about displaying indexed images.

My binary image displays as all black pixels. Check to see if its logical flag is "on." To do this, either use the `islogical` command or call `whos`. If the image is logical, the `whos` command will display the word "logical" after the word "array" under the class heading. If you have created your own binary image, chances are it is of class `uint8`, where a value of 1 is nearly black. Remember that the dynamic range of a `uint8` intensity image is [0 255], not [0 1]. For more information about valid binary images, see "Displaying Binary Images" on page 3-7.

I have loaded a multiframe image but MATLAB only displays one frame. You must load each frame of a multiframe image separately. This can be done using a `for` loop, and it may be helpful to first use `imfinfo` to find out how many frames there are, and what their dimensions are. To see an example that loads all of the frames of a multiframe image, go to "Displaying the Frames of a Multiframe Image Individually" on page 3-16.

4

Geometric Operations

Overview

This chapter describes the geometric functions, which are basic image processing tools. These functions modify the geometry of an image by resizing, rotating, or cropping the image. They 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.

Words You Need to Know

An understanding of the following terms will help you to use this chapter. For more explanation of this table and others like it, see “Words You Need to Know” in the Preface.

Words	Definitions
Aliasing	Artifacts in an image that can appear as a result of reducing an image’s size. When the size of an image is reduced, original pixels are downsampled to create fewer pixels. Aliasing that occurs as a result of size reduction normally appears as stair-step patterns (especially in high contrast images), or as Moire (ripple-effect) patterns.
Anti-aliasing	Any method for preventing aliasing (see above). The method discussed in this chapter is interpolation (see below).
Bicubic interpolation	Output pixel values are calculated from a weighted average of pixels in the nearest 4-by-4 neighborhood.
Bilinear interpolation	Output pixel values are calculated from a weighted average of pixels in the nearest 2-by-2 neighborhood.
Geometric operation	An operation that modifies the spatial relations between pixels in an image. Examples include resizing (growing or shrinking), rotating, and shearing.

Words	Definitions
Interpolation	The process by which we estimate an image value at a location in between image pixels.
Nearest neighbor interpolation	Output pixel values are assigned the value of the pixel that the point falls within. No other pixels are considered.

Interpolation

Interpolation is the process by which we estimate an image value at a location in between image 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 `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 "Intensity Profile" on page 8-5 for information about this function.)

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 computation is, so there is a trade-off 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, the command below 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. The command below 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 low-pass 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 low-pass 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 low-pass filter if nearest neighbor interpolation is used, unless you explicitly specify the filter. This interpolation method is primarily used for indexed images, and low-pass filtering is not appropriate for these images.

For information about specifying a different filter, see the reference page for `imresize`.

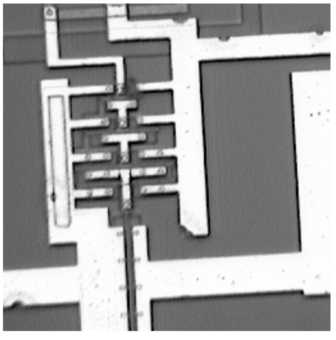
Image Rotation

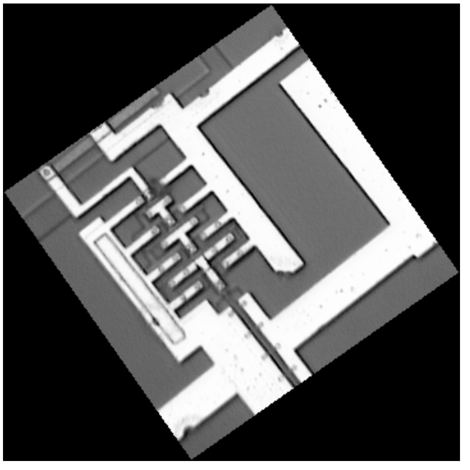
The `imrotate` function rotates an image, using a specified interpolation method and rotation angle. If you do not specify an interpolation method, the function uses nearest neighbor interpolation.

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)
```





In order to include the entire original image, `imrotate` pads the outside with 0's. This creates the black background in `J` and results in the output image being larger than the input image.

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

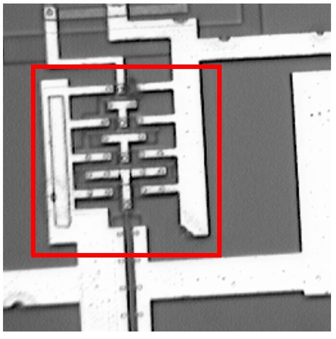
Image Cropping

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.

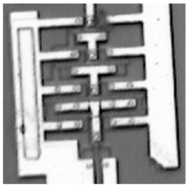
In this example, you display an image and call `imcrop`. The rectangle you select is shown in red.

```
imshow ic.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

Certain image processing operations involve processing an image in sections called *blocks*, rather than processing the entire image at once.

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.

Words You Need to Know

An understanding of the following terms will help you to use this chapter. For more explanation of this table and others like it, see "Words You Need to Know" in the Preface.

Words	Definitions
Block operation	An operation in which an image is processed in blocks rather than all at once. The blocks have the same size across the image. Some operation is applied to one block at a time. The blocks are reassembled to form an output image.
Border padding	Additional rows and columns temporarily added to the border(s) of an image when some of the blocks extend outside the image. The additional rows and columns normally contain zeros.
Center pixel	The pixel at the center of a neighborhood.
Column processing	An operation in which neighborhoods are reshaped into columns before processing in order to speed up computation time.
Distinct block operation	A block operation in which the blocks do not overlap.

Words	Definitions
Inline function	A user-defined function created using the MATLAB function `inline`. Toolbox functions whose syntax includes a parameter called `FUN` can take an inline function as an argument.
Neighborhood operation	An operation in which each output pixel is computed from a set of neighboring input pixels. Convolution, dilation, and median filtering are examples of neighborhood operations. A neighborhood operation can also be called a sliding neighborhood operation.
Overlap	Extra rows and columns of pixels outside a block whose values are taken into account when processing the block. These extra pixels cause distinct blocks to overlap one another. The `blkproc` function enables you to specify an overlap.

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 6, “Linear Filtering and Filter Design” and Chapter 9, “Binary Image Operations” 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.

Figure 5-1 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.

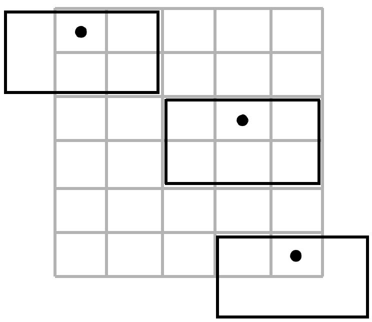


Figure 5-1: 2-by-3 Sliding Blocks for Sliding Neighborhood Operations

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

```
floor(([m n]+1)/2)
```

In the 2-by-3 block shown in Figure 5-1, the center pixel is (1,2), or, the pixel in the second column of the top row of the neighborhood.

To perform a sliding neighborhood operation

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 Figure 5-1 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 Figure 5-1 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 `conv` and `filter2` functions for performing convolution. See Chapter 6, "Linear Filtering and Filter Design" 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);
```

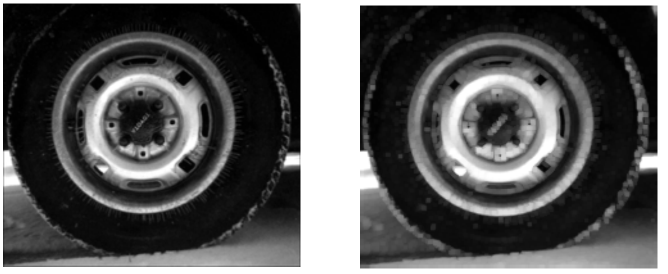


Figure 5-2: Each Output Pixel Set to Maximum Input Neighborhood Value

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 reference page for `colfilt`.

Note `nlfilter` is an example of a "function function." For more information on how to use this kind of function, see Appendix A. For more information on inline functions, see `inline` in the MATLAB Function Reference.

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. Figure 5-3 shows a 15-by-30 matrix divided into 4-by-8 blocks.

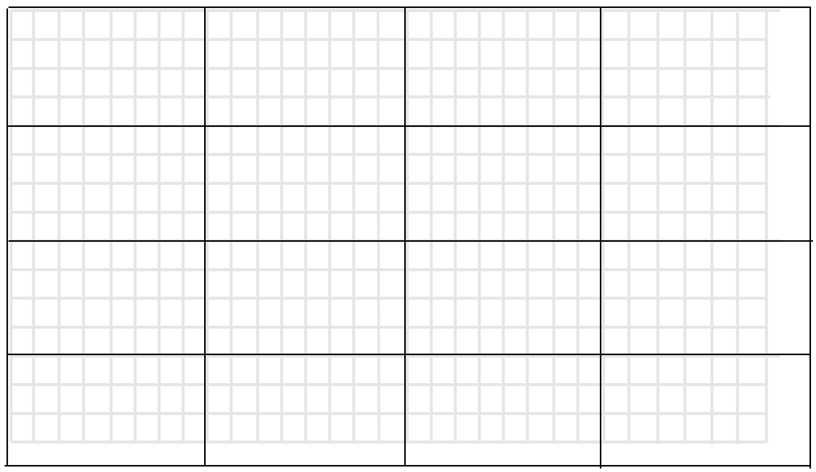


Figure 5-3: An Image Divided into Distinct Blocks

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`.

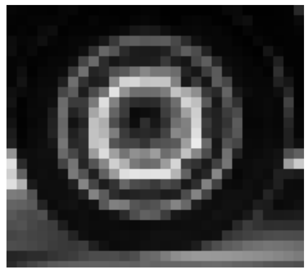
```
I2 = blkproc(I,[4 6],'myfun');
```

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 `inline` 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.

Note `blkproc` is an example of a "function function." For more information on how to use this kind of function, see "Working with Function Functions" (Appendix A).

Overlap

When you call `blkproc` to define distinct blocks, you can specify that the blocks overlap each other, that is, you can specify 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.

Figure 5-4 shows the overlap areas for some of the blocks in a 15-by-30 matrix with 1-by-2 overlaps. Each 4-by-8 block has a one-row overlap above and below, and a two-column overlap on each side. In the figure, shading indicates the overlap. The 4-by-8 blocks overlay the image matrix starting in the upper-left corner.

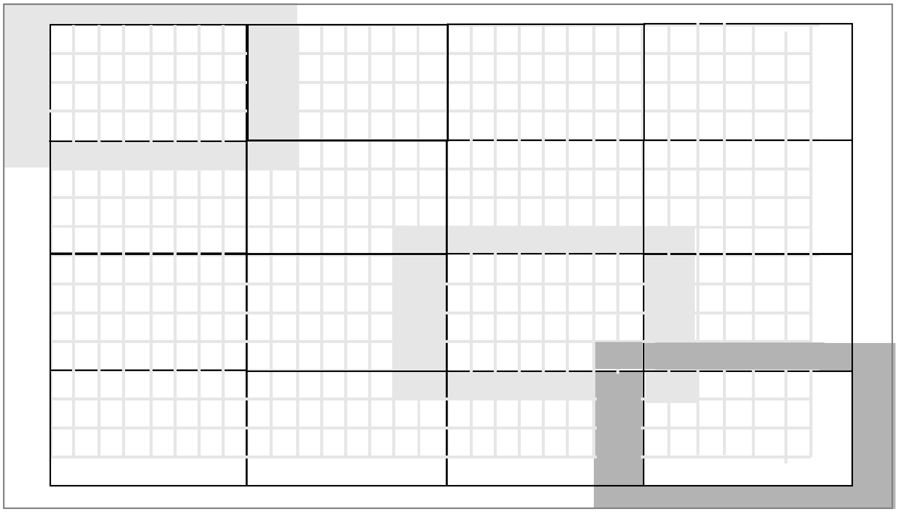


Figure 5-4: An Image Divided into Distinct Blocks With Specified Overlaps

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

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

Overlap often increases the amount of zero padding needed. For example, in Figure 5-3, the original 15-by-30 matrix became a 16-by-32 matrix with zero padding. When the 15-by-30 matrix includes a 1-by-2 overlap, the padded matrix becomes an 18-by-36 matrix. The outermost rectangle in the figure delineates the new boundaries of the image after padding has been added to accommodate the overlap plus block processing. Notice that in the figure above, padding has been added to the left and top of the original image, not just to the right and bottom.

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.

Figure 5-5 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.

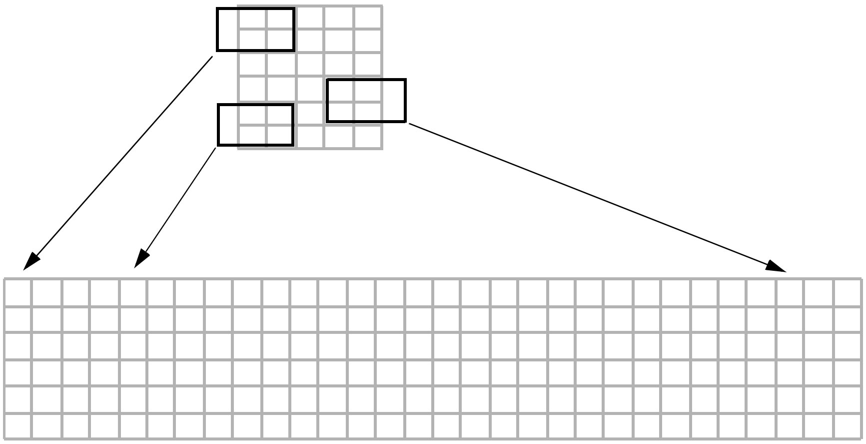


Figure 5-5: colfilt Creates a Temporary Matrix for Sliding Neighborhood

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 `nlfilter` 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 `nlfilter` example shown in Figure 5-2. 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.

Figure 5-6 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.

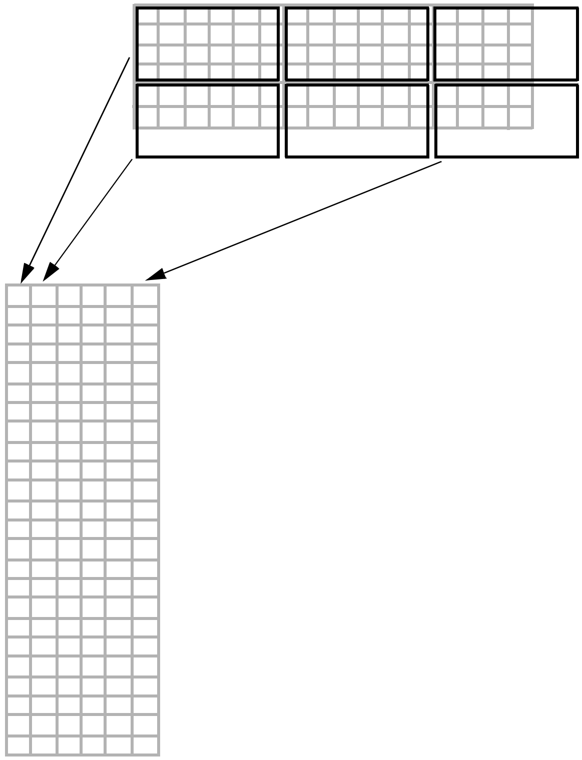


Figure 5-6: colfilt Creates a Temporary Matrix for Distinct Block Operation

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 in "Distinct Block Operations" on page 5-9.

```
I = im2double(imread('tire.tif'));
```

```
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`.

6

Linear Filtering and Filter Design

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.

Words You Need to Know

An understanding of the following terms will help you to use this chapter. For more explanation of this table and others like it, see “Words You Need to Know” in the Preface. Note that this table includes brief definitions of terms related to filter design; a detailed discussion of these terms and the theory behind filter design is outside the scope of this User Guide.

Words	Definitions
Computational molecule	A filter matrix used to perform correlation. The filter design functions in the Image Processing Toolbox return computational molecules. A computational molecule is a convolution kernel that has been rotated 180 degrees.
Convolution	A neighborhood operation in which each output pixel is a weighted sum of neighboring input pixels. The weights are defined by the convolution kernel. Image processing operations implemented with convolution include smoothing, sharpening, and edge enhancement.
Convolution kernel	A filter matrix used to perform convolution. A convolution kernel is a computational molecule that has been rotated 180 degrees.

Words	Definitions
Correlation	A neighborhood operation in which each output pixel is a weighted sum of neighboring input pixels. The weights are defined by the computational molecule. Image processing operations implemented with convolution include smoothing, sharpening, and edge enhancement. Correlation is closely related mathematically to convolution.
FIR filter	A filter whose response to a single point, or impulse, has finite extent. FIR stands for finite impulse response. An FIR filter can be implemented using convolution. All filter design functions in the Image Processing Toolbox return FIR filters.
Frequency response	A mathematical function describing the gain of a filter in response to different input frequencies.
Neighborhood operation	An operation in which each output pixel is computed from a set of neighboring input pixels. Convolution, dilation, and median filtering are examples of neighborhood operations.
Ripples	Oscillations around a constant value. The frequency response of a practical filter often has ripples where the frequency response of an ideal filter is flat.
Separable filter	A two-dimensional filter that can be implemented by a sequence of two one-dimensional filters. Separable filters can be implemented much faster than nonseparable filters. The function `filter2` checks a filter for separability before applying it to an image.
Window method	A filter design method that multiples the ideal impulse response by a window function, which tapers the ideal impulse response. The resulting filter's frequency response approximates a desired frequency response.

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. (See Chapter 5, "Neighborhood and Block Operations", for a general discussion of neighborhood operations.)

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

See "Filter Design" on page 6-14 for information about how to design filters.

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

```
k = [4 -3  1
     4  6  2]
```

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 = 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 Figure 6-1.

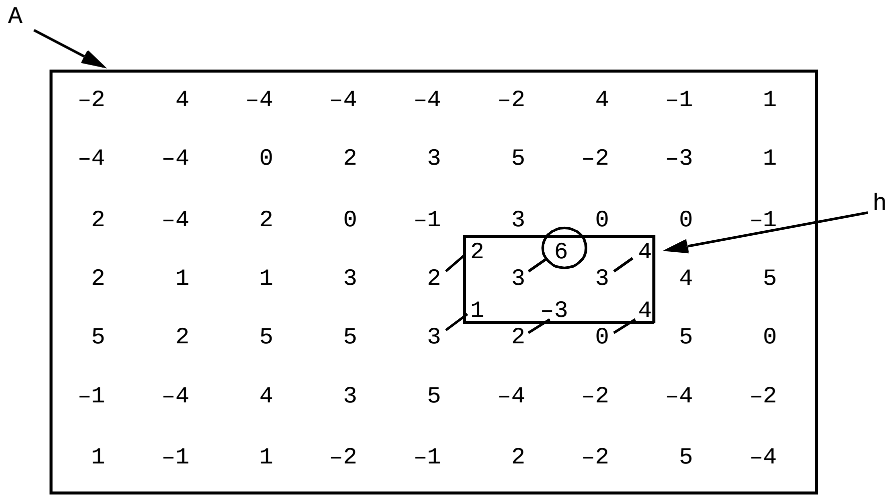


Figure 6-1: Overlaying the Computational Molecule for Convolution

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.

Figure 6-2 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.

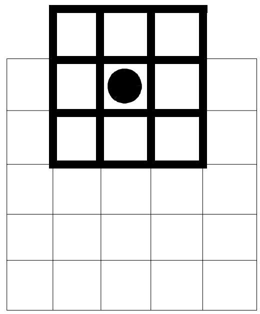


Figure 6-2: Computational Molecule Overhanging Top Row of Image

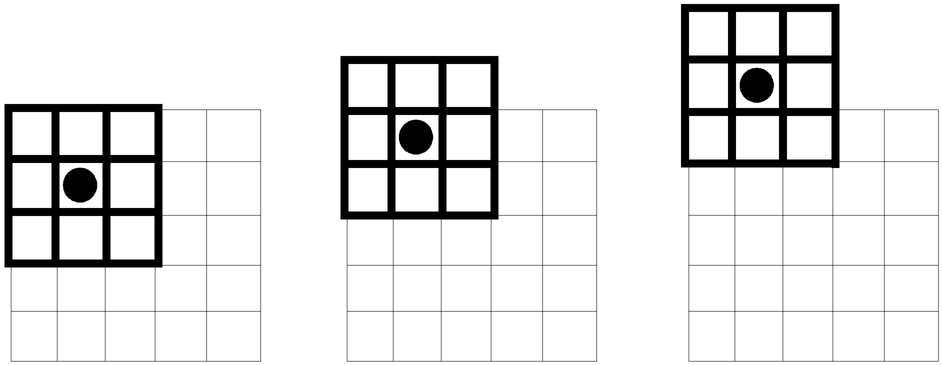
In order to compute output values for the border pixels, conv2 pads the image matrix with zeroes. In other words, the output values are computed by assuming that the input image is surrounded by additional rows and columns of zeroes. In the figure shown above, the elements in the top row of the computational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

- `'valid'` – 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.

Figure 6-3 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.

Figure 6-3: Computation Molecule Applied to Different Areas at Edge

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, the following 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.

Separability

If a filter has *separability*, meaning that it can be separated into two one-dimensional filters (one column vector and one row vector), the computation speed for the filter can be greatly enhanced. Before calling conv2 to perform two-dimensional convolution, filter2 first checks whether the filter is separable. 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);
```

where 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
```

If k is separable (that is, it has rank 1), then you can determine the corresponding column and row vectors with

```
[u,s,v] = svd(k);
kcol = u(:,1) * sqrt(s(1))

kcol =
    0.9036
    1.8072
    3.6144

krow = conj(v(:,1))' * sqrt(s(1))


krow =
    1.1067    2.2134    3.3200
```

Perform array multiplication on the separated vectors to verify your results.

```
kcol * krow

ans =
    1.0000    2.0000    3.0000
    2.0000    4.0000    6.0000
    4.0000    8.0000   12.0000
```

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 `convn` in the MATLAB *Function Reference*.

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/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)
```

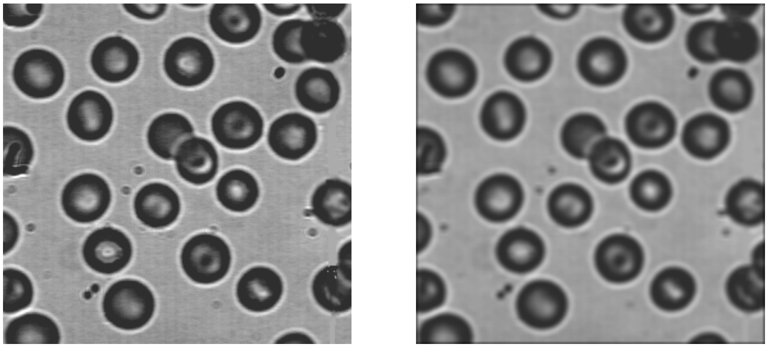


Figure 6-4: Blood.tif (left) and Blood.tif After Averaging Filter Applied (right)

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 is 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. For example, if the input image is of class `double`, the output array may include values outside the range [0,1]. 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 or uint16, you should not simply convert the output array to the same class as the input image, because the output may contain values outside the range that the class can represent. For example, if the input image is of class uint8, the output may include values that cannot be represented as 8-bit integers. 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));
```

You can also use imshow to display the output array without first rescaling the data. 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,[])
```

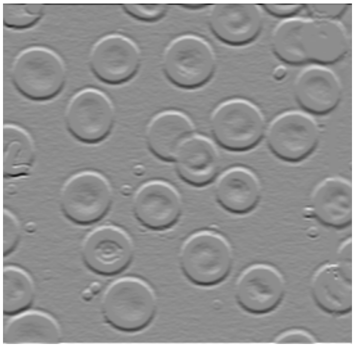


Figure 6-5: Blood.tif with Sobel Filter Applied

For a description of all the filter types fspecial provides, see the reference page for fspecial.

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])
```

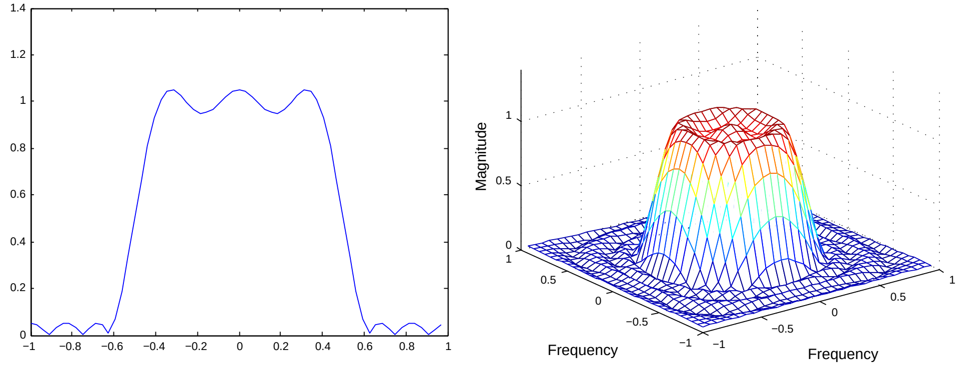


Figure 6-6: A One-Dimensional Frequency Response (left) and the Corresponding Two-Dimensional Frequency Response (right)

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. See "Computing the Frequency Response of a Filter" on page 6-19 for more information.)

```
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])
```

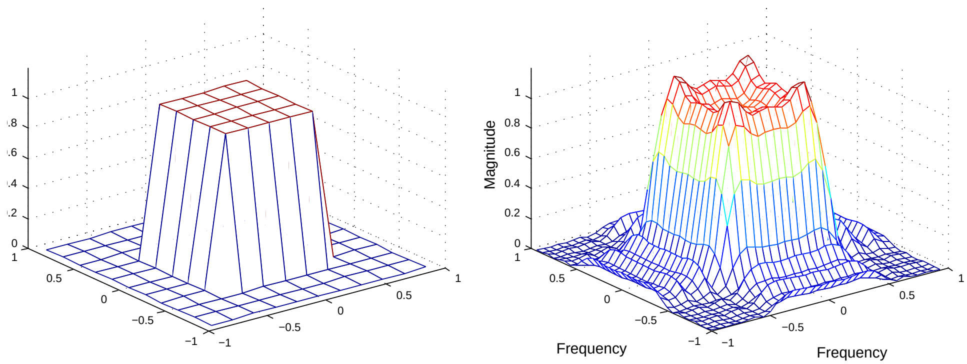


Figure 6-7: Desired Two-Dimensional Frequency Response (left) and Actual Two-Dimensional Frequency Response (right)

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, `fwind1` and `fwind2`. `fwind1` designs a two-dimensional filter by using a two-dimensional window that it creates from one or two one-dimensional windows that you specify. `fwind2` designs a two-dimensional filter by using a specified two-dimensional window directly.

fwind1 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 `fwind1` to create an 11-by-11 filter from the desired frequency response `Hd`. Here, the `hamming` function from the *Signal Processing Toolbox* is used to create a one-dimensional window, which `fwind1` 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 = fwind1(Hd,hamming(11));
figure, freqz2(h,[32 32]), axis([-1 1 -1 1 0 1.2])
```

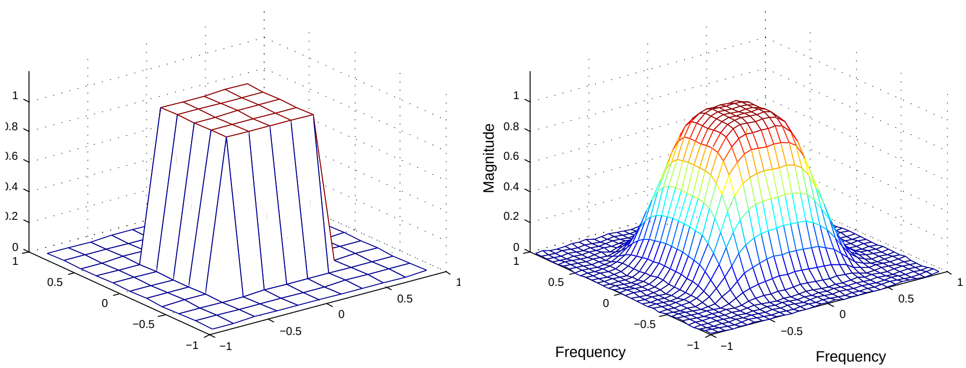


Figure 6-8: Desired Two-Dimensional Frequency Response (left) and Actual Two-Dimensional Frequency Response (right)

Creating the Desired Frequency Response Matrix

The filter design functions `fsamp2`, `fwind2`, 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)
```

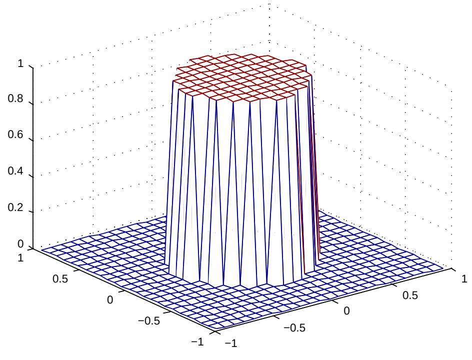


Figure 6-9: Ideal Circular Lowpass Frequency Response

Note that for this frequency response, the 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 (`f1 = 0`, `f2 = 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)
```

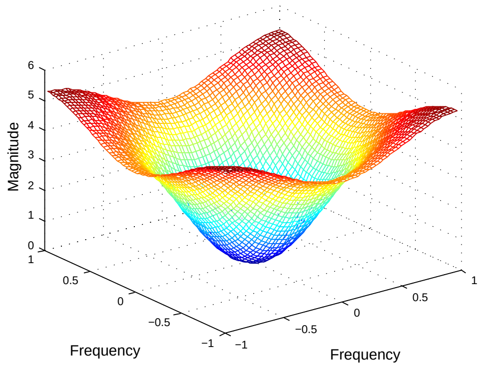


Figure 6-10: The Frequency Response of a Two-Dimensional Filter

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.

See "Fourier Transform" on page 7-4 for more information about the fast Fourier transform and its application to linear filtering and filter design.

Transforms

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.

Words You Need to Know

An understanding of the following terms will help you to use this chapter. For more explanation of this table and others like it, see “Words You Need to Know” in the Preface. Note that this table includes brief definitions of terms related to transforms; a detailed discussion of these terms and the theory behind transforms is outside the scope of this *User's Guide*.

Words	Definitions
Discrete transform	A transform whose input and output values are discrete samples, making it convenient for computer manipulation. Discrete transforms implemented by MATLAB and the Image Processing Toolbox include the discrete Fourier transform (DFT) and the discrete cosine transform (DCT).
Frequency domain	The domain in which an image is represented by a sum of periodic signals with varying frequency.
Inverse transform	An operation that when performed on a transformed image, produces the original image.
Spatial domain	The domain in which an image is represented by intensities at given points in space. This is the most common representation for image data.
Transform	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. Transforms are useful for a wide range of purposes, including convolution, enhancement, feature detection, and compression.

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 includes the following subsections:

- “Definition of Fourier Transform”
- “The Discrete Fourier Transform”, including a discussion of fast Fourier transform
- “Applications” (sample applications using Fourier transforms)

Definition of Fourier Transform

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 $-\pi \le \omega_1, \omega_2 \le \pi$ 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

$$f(m, n) = \frac{1}{4\pi^2} \int_{\omega_1=-\pi}^{\pi} \int_{\omega_2=-\pi}^{\pi} F(\omega_1, \omega_2) e^{j\omega_1 m} e^{j\omega_2 n} \, d\omega_1 \, d\omega_2$$

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.

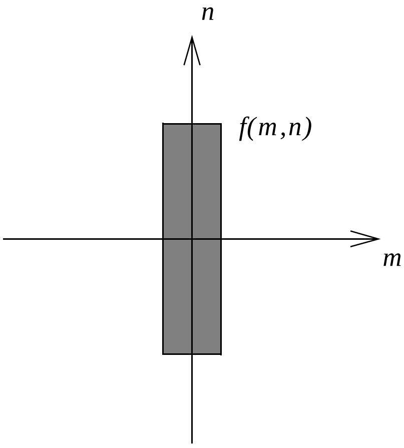


Figure 7-1: A Rectangular Function

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

Figure 7-2 shows the magnitude of the Fourier transform, $|F(\omega_1, \omega_2)|$, of Figure 7-1 as a mesh plot. The mesh plot of the magnitude is a common way to visualize the Fourier transform.

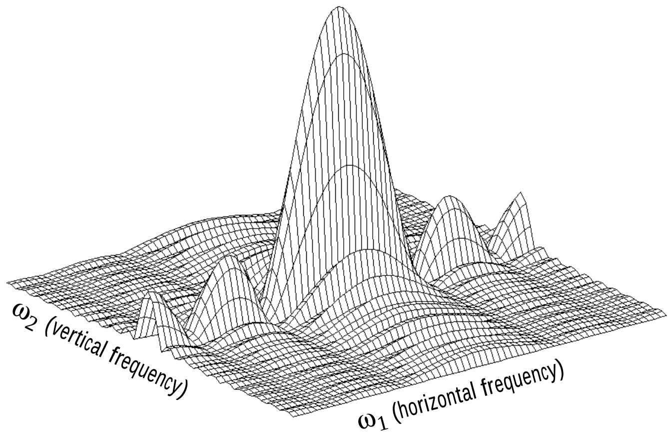


Figure 7-2: Magnitude Image of a Rectangular Function

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

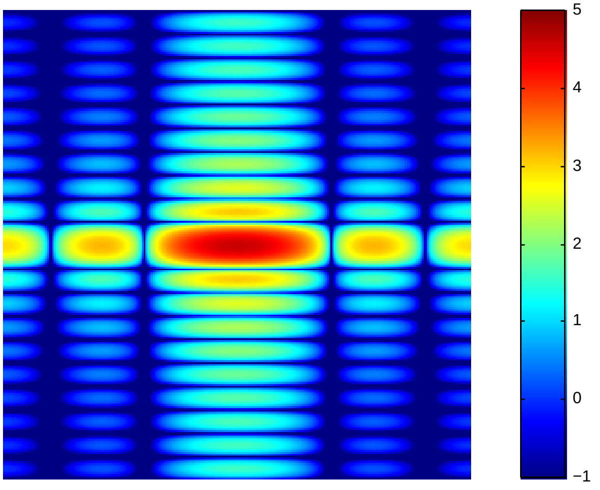


Figure 7-3: The Log of the Fourier Transform of a Rectangular Function

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.

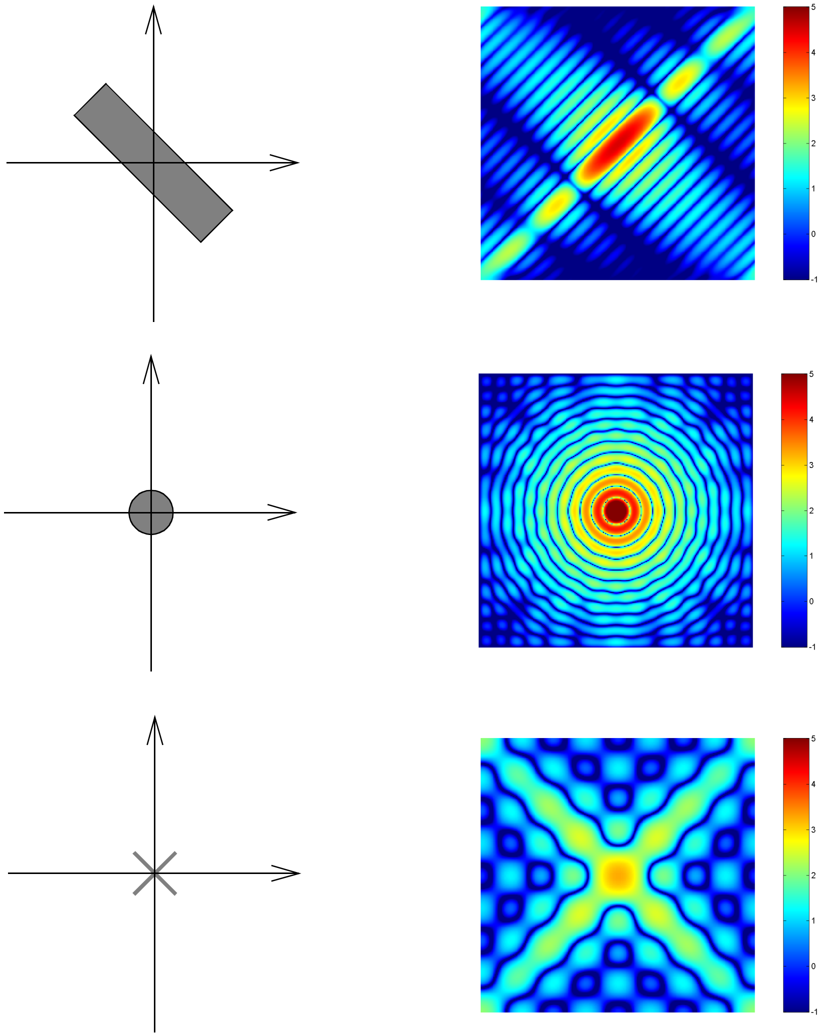


Figure 7-4: Fourier Transforms of Some Simple Shapes

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 $0 \le m \le M-1$ and $0 \le n \le N-1$. The two-dimensional M-by-N DFT and inverse M-by-N DFT relationships are given by

$$F(p, q) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) e^{-j(2\pi/M)pm} e^{-j(2\pi/N)qn} \qquad \begin{array}{l} p = 0, 1, \ldots, M-1 \\ q = 0, 1, \ldots, N-1 \end{array}$$

$$f(m, n) = \frac{1}{MN} \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} F(p, q) e^{j(2\pi/M)pm} e^{j(2\pi/N)qn} \qquad \begin{array}{l} m = 0, 1, \ldots, M-1 \\ n = 0, 1, \ldots, N-1 \end{array}$$

The values $F(p, q)$ are the DFT coefficients of $f(m, n)$ [1]. In particular, the value $F(0, 0)$ is often 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.

1. The zero-frequency coefficient, $F(0, 0)$ is often called the “DC component.” DC is an electrical engineering term that stands for direct current.

Relationship to the Fourier Transform

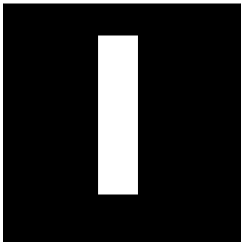
The DFT coefficients $F(p, q)$ are samples of the Fourier transform $F(\omega_1, \omega_2)$.

$$F(p,q) = F(\omega_1, \omega_2)\Big|_{\substack{\omega_1 = 2\pi p/M \\ \omega_2 = 2\pi q/N}} \qquad \begin{aligned} p &= 0, 1, \ldots, M-1 \\ q &= 0, 1, \ldots, N-1 \end{aligned}$$

Example

Let's construct a matrix `f` that is similar to the function *f(m,n)* in the example in "Definition of Fourier Transform" on page 7-4. Remember that *f(m,n)* is equal to 1 within the rectangular region and 0 elsewhere. We use a binary image to represent *f(m,n)*.

```
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
```

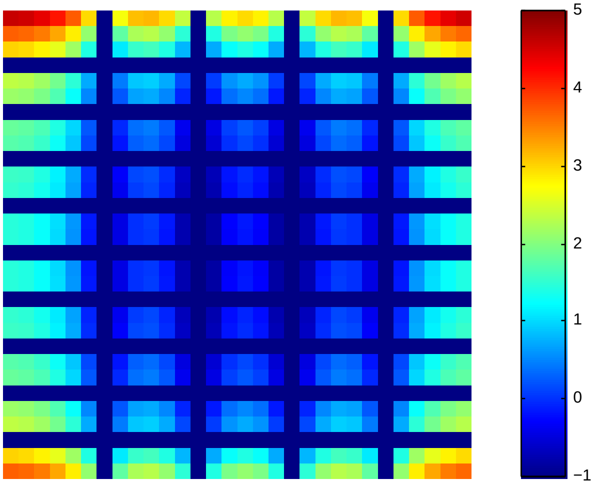


Figure 7-5: A Discrete Fourier Transform Computed Without Padding

This plot differs from the Fourier transform displayed on Figure 7-3. First, the sampling of the Fourier transform is much coarser. Second, the zero-frequency coefficient is displayed in the upper-left corner instead of the traditional location in the center.

We can obtain a finer sampling of the Fourier transform 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.

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

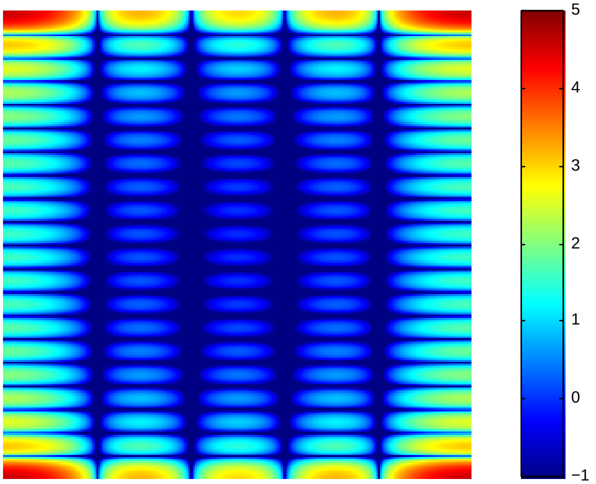


Figure 7-6: A Discrete Fourier Transform Computed With Padding

The zero-frequency 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 zero-frequency 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 Figure 7-3.

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)
```

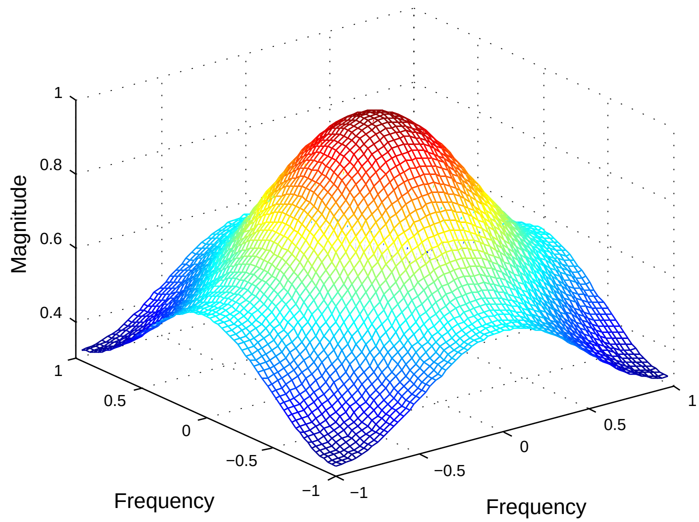


Figure 7-7: The Frequency Response of a Gaussian Filter

See Chapter 6, “Linear Filtering and Filter Design” 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)-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 an image containing text. This example reads in `text.tif` and creates a template image by extracting a letter “a” from it,

```
bw = imread('text.tif');
a=bw(59:71,81:91); %Extract one of the letters “a” from the image.
imshow(bw);
figure, imshow(a);
```

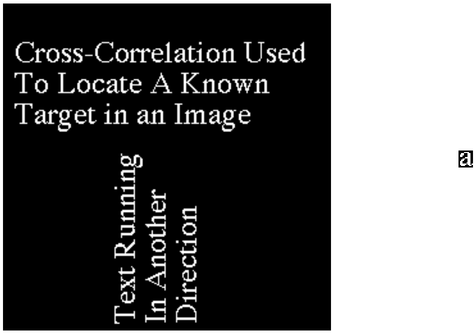


Figure 7-8: An Image (left) and the Template to Correlate (right)

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^{o} and then using the FFT-based convolution technique described above. (Note that convolution is equivalent to correlation if you rotate the convolution kernel by 180^{o}.) To match the template to the image, you can use the `fft2` and `ifft2` functions,

```
C = real(ifft2(fft2(bw) .* fft2(rot90(a,2),256,256)));
figure, imshow(C,[])%Display, scaling data to appropriate range.
max(C(:)) %Find max pixel value in C.

ans =

   51.0000

thresh = 45; %Use a threshold that’s a little less than max.
figure, imshow(C > thresh)%Display showing pixels over threshold.
```

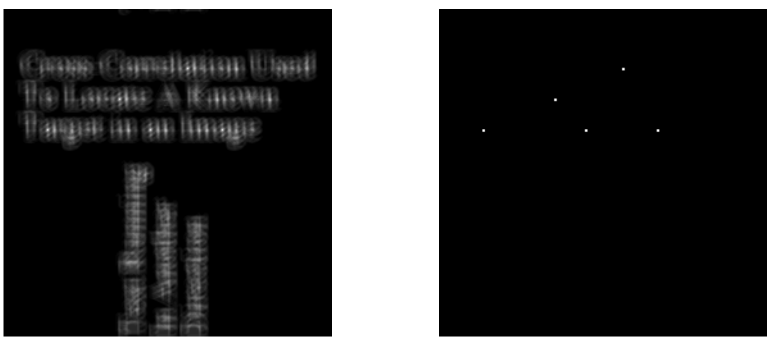


Figure 7-9: A Correlated Image (left) and its Thresholded Result (right)

The left image above 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.

Note that you could also have created the template image by zooming in on the image and using the interactive version of `imcrop`. For example, with `text.tif` displayed in the current figure window, enter

```
zoom on
a = imcrop
```

To determine the coordinates of features in an image, you can use the `pixval` function.

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.

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

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

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

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

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

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

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

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.

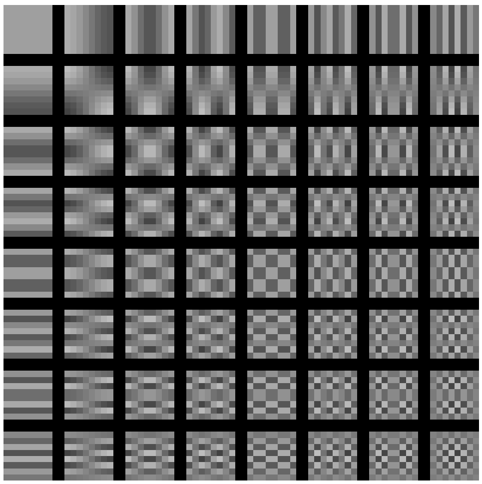


Figure 7-10: The 64 Basis Functions of an 8-by-8 Matrix

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

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

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 = im2double(I);
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. Figure 7-11 shows horizontal and vertical projections for a simple two-dimensional function.

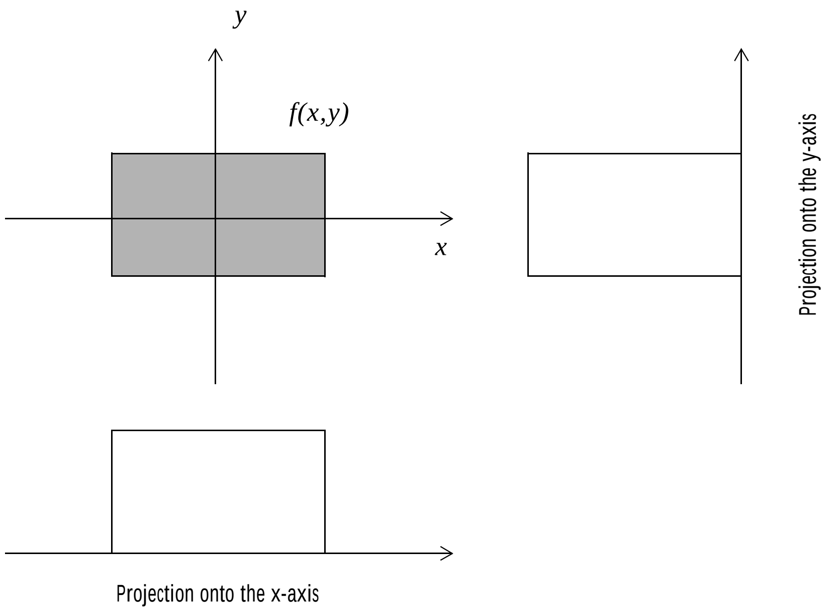


Figure 7-11: Horizontal and Vertical Projections of a Simple Function

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

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

Figure 7-12 illustrates the geometry of the Radon transform.

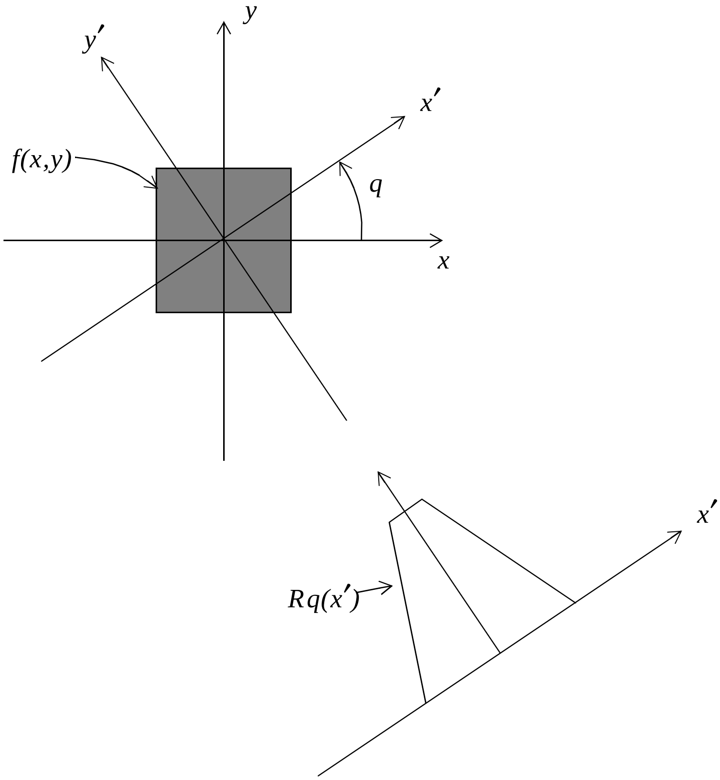


Figure 7-12: The Geometry of the Radon Transform

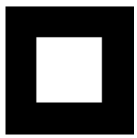
This command computes the Radon transform of `I` for the angles specified in the vector `theta`

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

The columns of R contain the Radon transform for each angle in theta. The vector xp contains the corresponding coordinates along the x'-axis. The "center pixel" of I is defined to be floor((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^o} (x\prime)')
figure; plot(xp,R(:,2)); title('R_{45^o} (x\prime)')
```

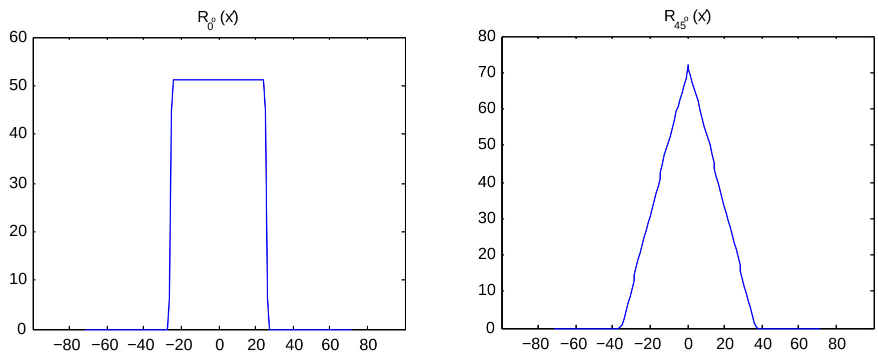


Figure 7-13: Two Radon Transforms of a Square Function

Note xp is the same for all projection angles.

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);
title('R_{\theta} (X\prime)');
xlabel('\theta (degrees)');
ylabel('X\prime');
set(gca,'XTick',0:20:180);
colormap(hot);
colorbar
```

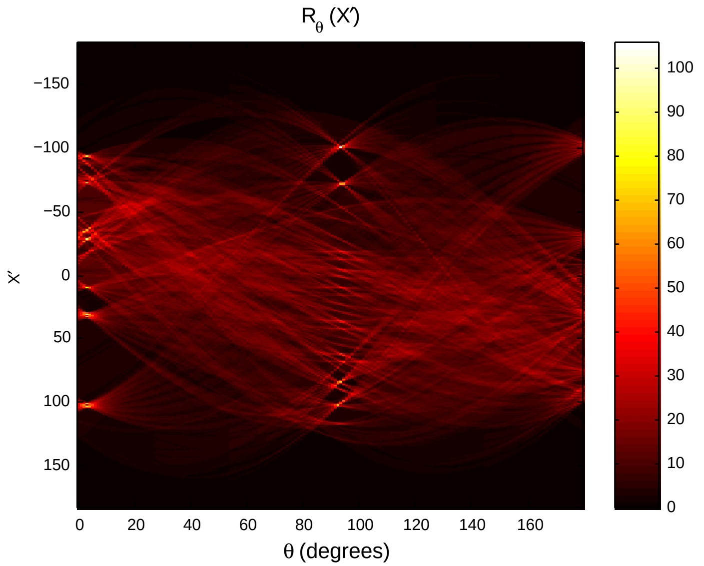


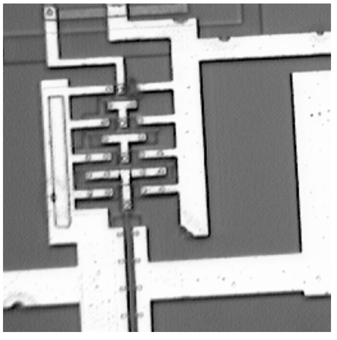
Figure 7-14: A Radon Transform Using 180 Projections

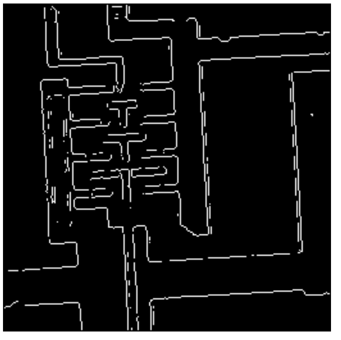
Using the Radon Transform to Detect Lines

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);
figure, imagesc(theta, xp, R); colormap(hot);
xlabel('\theta (degrees)'); ylabel('X\prime');
title('R_{\theta} (X\prime)');
colorbar
```

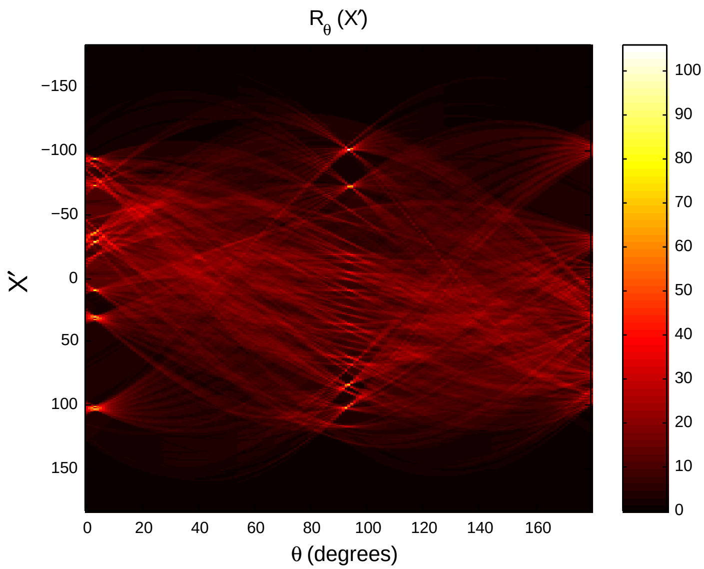


Figure 7-15: Radon Transform of an Edge Image

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 $\theta = 94°$ 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.

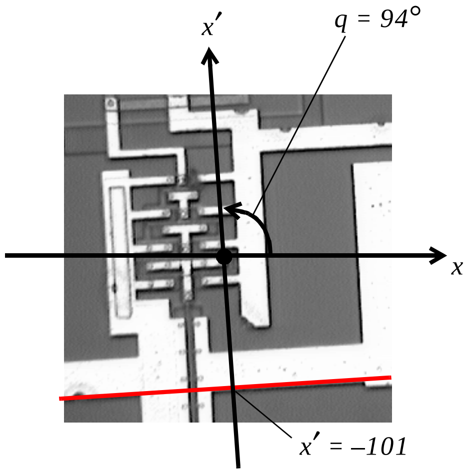


Figure 7-16: The Radon Transform Geometry and the Strongest Peak (Red)

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

The Inverse Radon Transform

The `iradon` function performs the inverse Radon transform, which is commonly used in tomography applications. This transform inverts the Radon transform (which was introduced in the previous section), and can therefore be used to reconstruct images from projection data.

As discussed in the previous section “Radon Transform” on page 7-21, given an image `I` and a set of angles `theta`, the function `radon` can be used to calculate the Radon transform.

```
R = radon(I,theta);
```

The function `iradon` can then be called to reconstruct the image `I`.

```
IR = iradon(R,theta);
```

In the example above, projections are calculated from the original image `I`. In most application areas, there is no *original image* from which projections are formed. For example, in X-ray absorption tomography, projections are formed by measuring the attenuation of radiation that passes through a physical specimen at different angles. The original image can be thought of as a cross section through the specimen, in which intensity values represent the density of the specimen. Projections are collected using special purpose hardware, and then an internal image of the specimen is reconstructed by `iradon`. This allows for noninvasive imaging of the inside of a living body or another opaque object.

`iradon` reconstructs an image from parallel beam projections. In *parallel beam geometry*, each projection is formed by combining a set of line integrals through an image at a specific angle.

Figure 7-17 below illustrates how parallel beam geometry is applied in X-ray absorption tomography. Note that there is an equal number of *n* emitters and *n* detectors. Each detector measures the radiation emitted from its corresponding emitter, and the attenuation in the radiation gives a measure of the integrated density, or mass, of the object. This corresponds to the line integral that is calculated in the Radon transform.

The parallel beam geometry used in the figure is the same as the geometry that was described under "Radon Transform" on page 7-21. `f(x,y)` denotes the brightness of the image and $R_q(x')$ is the projection at angle q.

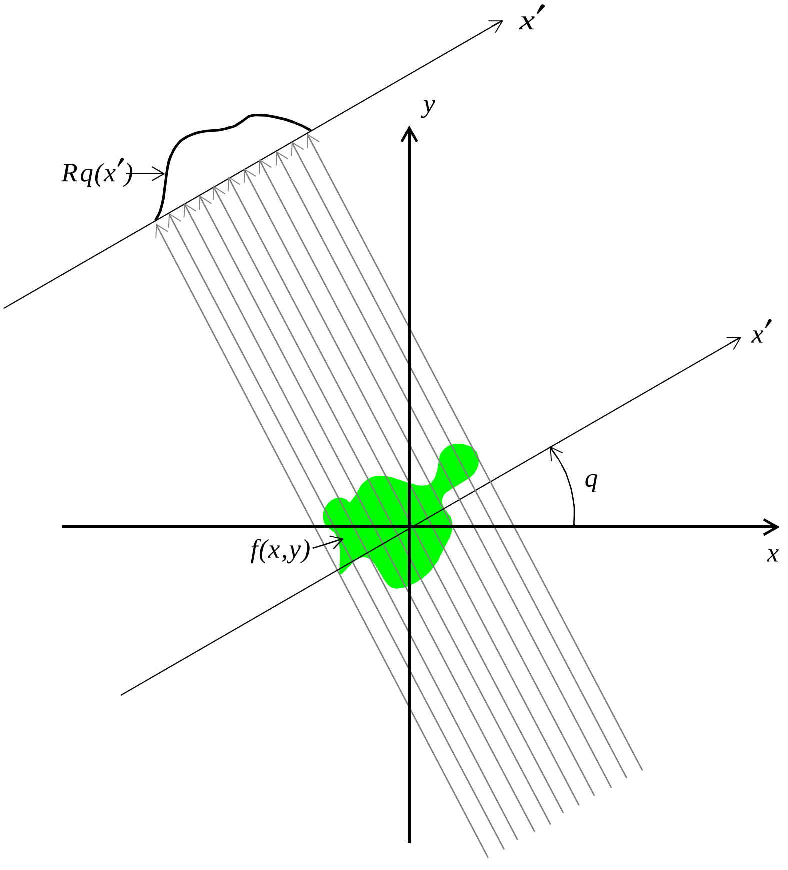


Figure 7-17: Parallel Beam Projections Through an Object

Another geometry that is commonly used is *fan beam* geometry, in which there is one emitter and *n* detectors. There are methods for resorting sets of fan beam projections into parallel beam projections, which can then be used by `iradon`. (For more information on these methods, see Kak & Slaney, *Principles of Computerized Tomographic Imaging*, IEEE Press, NY, 1988, pp. 92-93.)

`iradon` uses the *filtered backprojection* algorithm to compute the inverse Radon transform. This algorithm forms an approximation to the image `I` based on the projections in the columns of `R`. A more accurate result can be obtained by using more projections in the reconstruction. As the number of projections (the length of theta) increases, the reconstructed image IR more accurately approximates the original image I. The vector theta must contain monotonically increasing angular values with a constant incremental angle $\Delta\theta$. When the scalar $\Delta\theta$ is known, it can be passed to iradon instead of the array of theta values. Here is an example.

```
IR = iradon(R,Dtheta);
```

The filtered backprojection algorithm filters the projections in R and then reconstructs the image using the filtered projections. In some cases, noise can be present in the projections. To remove high frequency noise, apply a window to the filter to attenuate the noise. Many such windowed filters are available in iradon. The example call to iradon below applies a Hamming window to the filter. See the iradon reference page for more information.

```
IR = iradon(R,theta,'Hamming');
```

iradon also enables you to specify a normalized frequency, D, above which the filter has zero response. D must be a scalar in the range [0,1]. With this option, the frequency axis is rescaled, so that the whole filter is compressed to fit into the frequency range [0,D]. This can be useful in cases where the projections contain little high frequency information but there is high frequency noise. In this case, the noise can be completely suppressed without compromising the reconstruction. The following call to iradon sets a normalized frequency value of 0.85.

```
IR = iradon(R,theta,0.85);
```

Examples

The commands below illustrate how to use radon and iradon to form projections from a sample image and then reconstruct the image from the projections. The test image is the Shepp-Logan head phantom, which can be generated by the Image Processing Toolbox function phantom. The phantom image illustrates many of the qualities that are found in real-world tomographic imaging of human heads. The bright elliptical shell along the exterior is analogous to a skull, and the many ellipses inside are analogous to brain features or tumors.

```
P = phantom(256);
```

```
imshow(P)
```

As a first step the Radon transform of the phantom brain is calculated for three different sets of theta values. R1 has 18 projections, R2 has 36 projections, and R3 had 90 projections.

```
theta1 = 0:10:170; [R1,xp] = radon(P,theta1);
theta2 = 0:5:175;  [R2,xp] = radon(P,theta2);
theta3 = 0:2:178;  [R3,xp] = radon(P,theta3);
```

Now the Radon transform of the Shepp-Logan Head phantom is displayed using 90 projections (R3).

```
figure, imagesc(theta3,xp,R3); colormap(hot); colorbar
xlabel('\theta'); ylabel('x\prime');
```

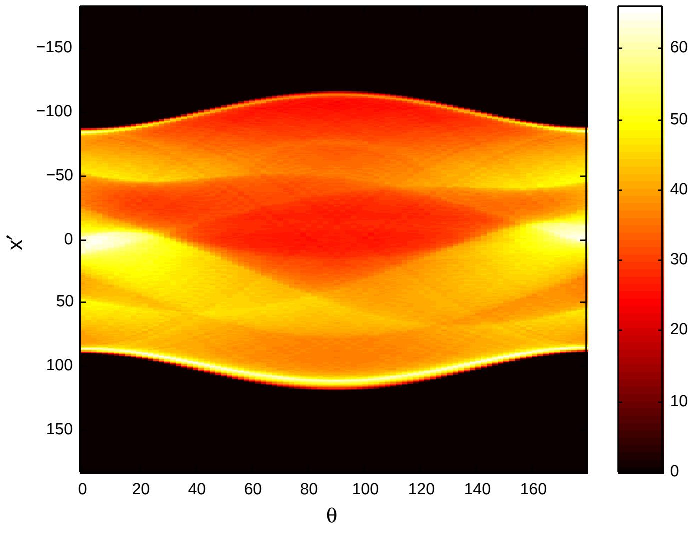


Figure 7-18: Radon Transform of Head Phantom Using 90 Projections

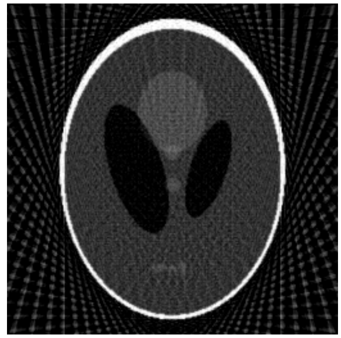
When we look at Figure 7-18, we can see some of the features of the input image. The first column in the Radon transform corresponds to a projection at 0° which is integrating in the vertical direction. The centermost column corresponds to a projection at 90°, which is integrating in the horizontal direction. The projection at 90° has a wider profile than the projection at 0° due to the larger vertical semi-axis of the outermost ellipse of the phantom.

Figure 7-19 shows the inverse Radon transforms of `R1`, `R2`, and `R3`, which were generated above. Image `I1` was reconstructed with the projections in `R1`, and it is the least accurate reconstruction, because it has the fewest projections. `I2` was reconstructed with the 36 projections in `R2`, and the quality of the reconstruction is better, but it is still not clear enough to discern clearly the three small ellipses in the lower portion of the test image. `I3` was reconstructed using the 90 projections in `R3`, and the result closely resembles the original image. Notice that when the number of projections is relatively small (as in `I1` and `I2`), the reconstruction may include some artifacts from the back projection. To avoid this, use a larger number of angles.

```
I1 = iradon(R1,10);
I2 = iradon(R2,5);
I3 = iradon(R3,2);
```

```
imshow(I1)
figure, imshow(I2)
figure, imshow(I3)
```

I1



I2

I3

Figure 7-19: Inverse Radon Transforms of the Shepp-Logan Head Phantom

Analyzing and Enhancing Images

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.

Words You Need to Know

An understanding of the following terms will help you to use this chapter. For more explanation of this table and others like it, see “Words You Need to Know” in the Preface.

Words	Definitions
Adaptive filter	A filter whose properties vary across an image depending on the local characteristics of the image pixels.
Contour	A path in an image along which the image intensity values are equal to a constant.
Edge	A curve that follows a path of rapid change in image intensity. Edges are often associated with the boundaries of objects in a scene. Edge detection is used to identify the edges in an image.
Feature	A quantitative measurement of an image or image region. Examples of image region features include centroid, bounding box, and area.

Words	Definitions
Histogram	A graph used in image analysis that shows the distribution of intensities in an image. The information in a histogram can be used to choose an appropriate enhancement operation. For example, if an image histogram shows that the range of intensity values is small, you can use an intensity adjustment function to spread the values across a wider range.
Noise	Errors in the image acquisition process that result in pixel values that do not reflect the true intensities of the real scene.
Profile	A set of intensity values taken from regularly spaced points along a line segment or multiline path in an image. For points that do not fall on the center of a pixel, the intensity values are interpolated.
Quadtree decomposition	An image analysis technique that partitions an image into homogeneous blocks.

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 (`pixval`, `impixel`)
- 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 toolbox includes two functions that provide information about the color data values of image pixels you specify:

- The `pixval` function 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.
- 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.

To use `pixval`, you first display an image and then enter the `pixval` command. `pixval` installs a black bar at the bottom of the figure, which displays the (x,y) coordinates for whatever pixel the cursor is currently over, and the color data for that pixel.

If you click on the image and hold down the mouse button while you move the cursor, `pixval` also displays the Euclidean distance between the point you clicked on and the current cursor location. `pixval` 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.

`pixval` gives you more immediate results than `impixel`, but `impixel` has the advantage of returning its results in a variable, and it can be called either interactively or noninteractively. If you call `impixel` with no input arguments, the cursor changes to a crosshair 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 the stars disappear.

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

```
imshow canoe.tif
vals = impixel
```

```
vals =

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.

For indexed images, `pixval` and `impixel` both show the RGB values stored in the colormap, not the index values.

Intensity Profile

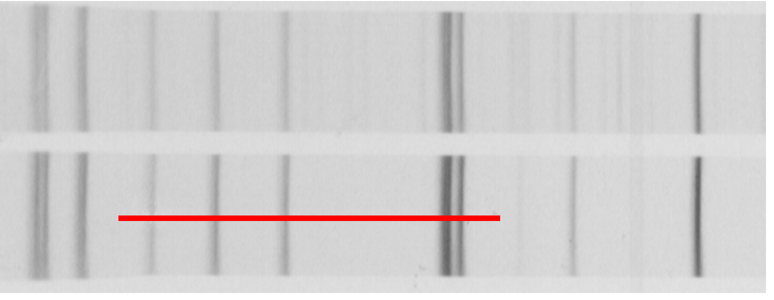
The `improfile` function calculates and plots the intensity values along a line segment or a multiline path in an image. You can supply the coordinates of the line segments as input arguments, or you can define the desired path using a mouse. In either case, `improfile` uses interpolation to determine 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 4, “Geometric Operations”, for a discussion of interpolation.) `improfile` works best with intensity and RGB images.

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.

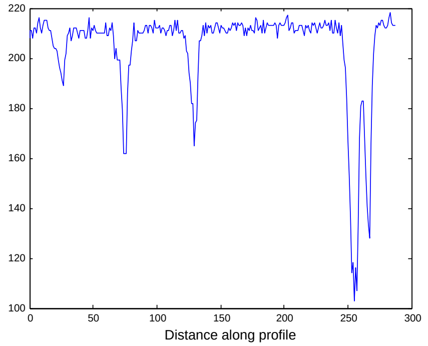


Figure 8-1: A Plot of Intensity Values Along a Line Segment in an Intensity Image

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 indicates where the line selection was made. Note that the line was drawn from top to bottom.

```
imshow flowers.tif
improfile
```

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

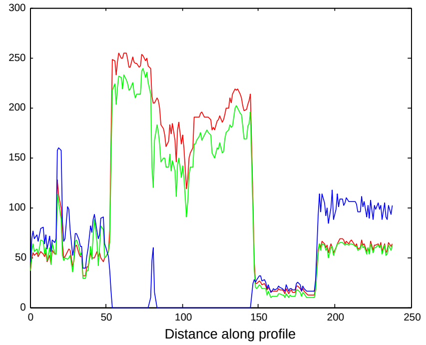


Figure 8-2: A Plot of Intensity Values Along a Line Segment in an RGB Image

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 an intensity image of grains of rice and a contour plot of the image data.

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

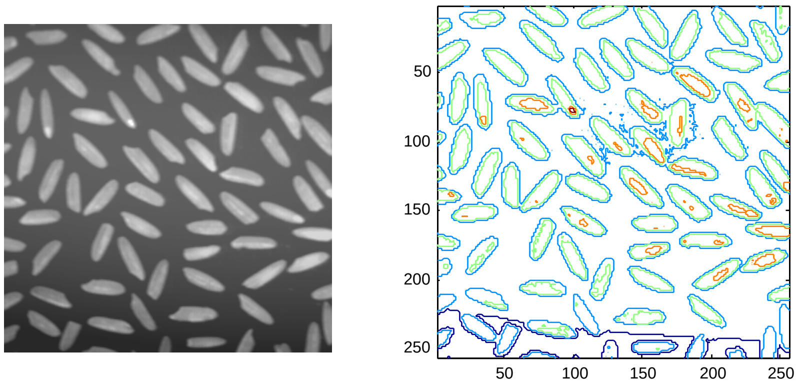


Figure 8-3: Rice.tif and Its Contour Plot

You can use the `clabel` function to label the levels of the contours. See the description of `clabel` in the 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 `imhist` 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)
```

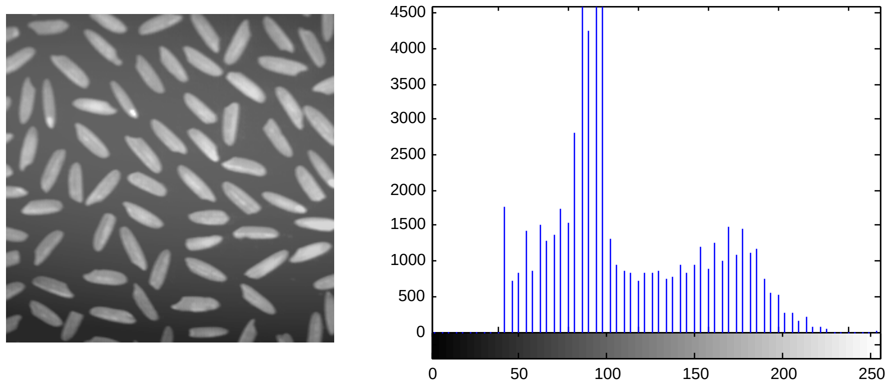


Figure 8-4: Rice.tif and Its Histogram

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

For information about how to modify an image by changing the distribution of its histogram, see "Intensity Adjustment" on page 8-15.

Summary Statistics

You can compute standard statistics of an image 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 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 page for `imfeature` 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 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

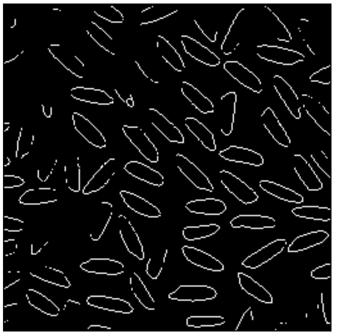
edge provides a number of derivative estimators, each of which implements one of the definitions above. For some of these estimators, you can specify whether the operation should be sensitive to horizontal or vertical edges, or both. edge returns a binary image containing 1's where edges are found and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

The most powerful edge-detection method that edge provides is the Canny method. The Canny method differs from the other edge-detection methods in that it uses two different 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.

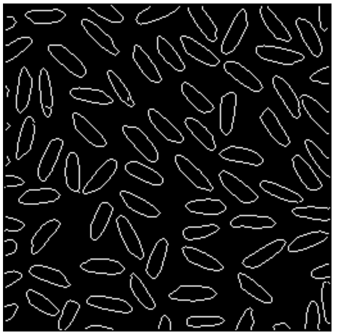
The example below illustrates the power of the Canny edge detector. It shows the results of applying the Sobel and Canny edge detectors to the rice.tif image.

```
I = imread('rice.tif');
BW1 = edge(I,'sobel');
BW2 = edge(I,'canny');
imshow(BW1)
```

```
figure, imshow(BW2)
```



Sobel Filter



Canny Filter

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 intensity 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(block(:)) – min(block(:)) <= 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 = 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 The threshold value is specified as a value between 0 and 1, regardless of the class of `I`. If `I` is `uint8`, the threshold value you supply is multiplied by 255 to determine the actual threshold to use; if `I` is `uint16`, the threshold value you supply is multiplied by 65535.

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.

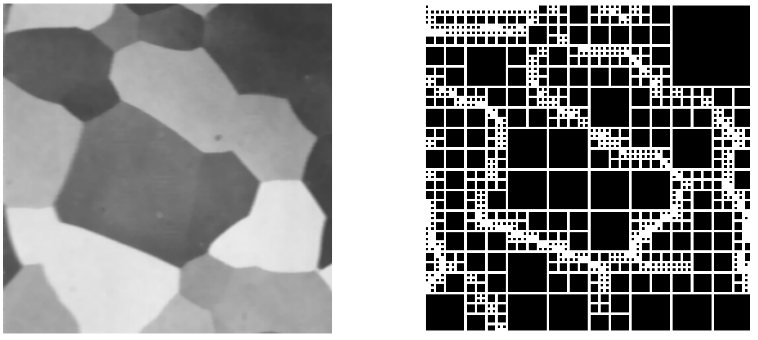


Figure 8-5: An Image (left) and a Representation of its Quadtree Decomposition

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 page for `qtdecomp` 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 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 pages for the individual functions.

Intensity Adjustment

Intensity adjustment is a technique for mapping an image's intensity values to a new range. For example, look at `rice.tif`. It is a low contrast image. If you look at a histogram of `rice.tif` in Figure 8-4, it indicates that there are 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]);
```

With the two vectors supplied, `imadjust` scales pixel values of 0.15 to 0 and 0.9 to 1. Notice that the intensities are specified as values between 0 and 1 regardless of the class of `I`. If `I` is `uint8`, the values you supply are multiplied by 255 to determine the actual values to use; if `I` is `uint16`, the values are multiplied by 65535.

Now display the adjusted image and its histogram.

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

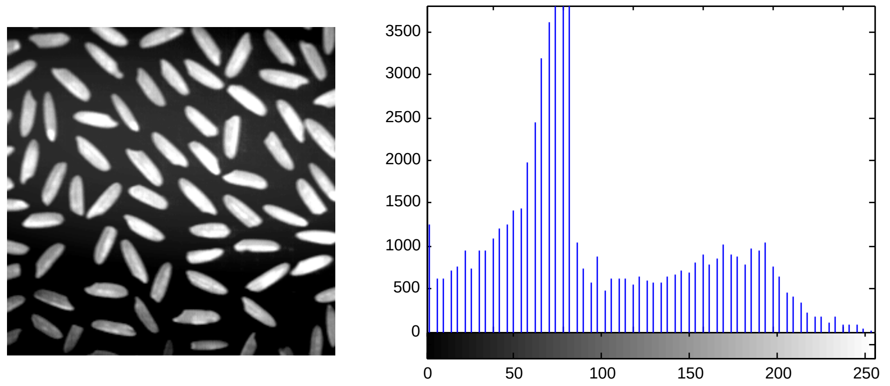


Figure 8-6: Rice.tif After an Intensity Adjustment and a Histogram of Its Adjusted Intensities

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 `uint8` input 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)
```

Figure 8-7: Cameraman.tif Before and After Remapping, and Widening its Dynamic Range

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.)

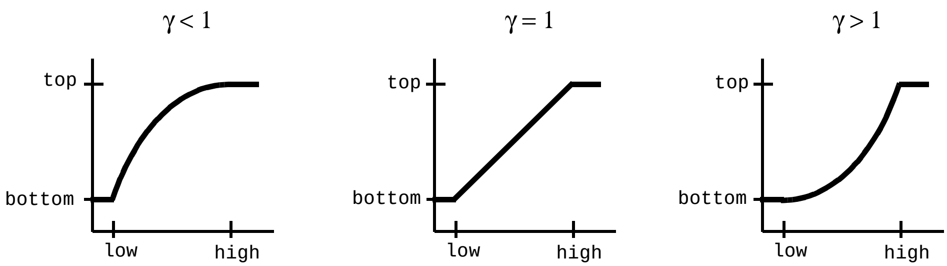


Figure 8-8: Plots Showing Three Different Gamma Correction Settings

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)
```

Figure 8-9: Forest.tif Before and After Applying Gamma Correction of 0.5

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 page for `histeq`.)

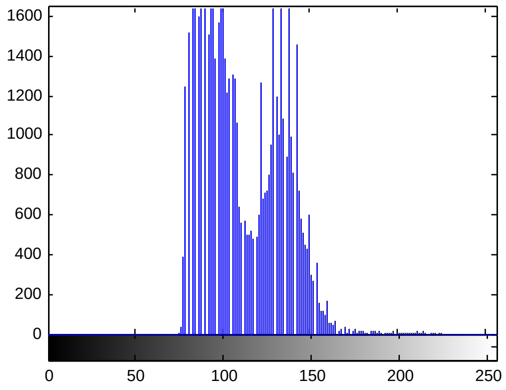
This example illustrates using `histeq` to adjust an intensity image. The original image has low contrast, with most values in the middle of the intensity range. `histeq` produces an output image having values evenly distributed throughout the range.

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

Figure 8-10: Pout.tif Before and After Histogram Equalization

The example below shows the histograms for the two images.

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



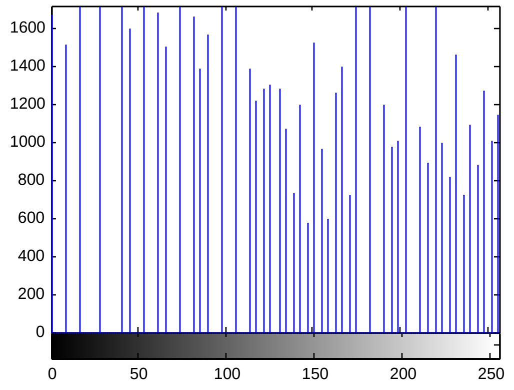
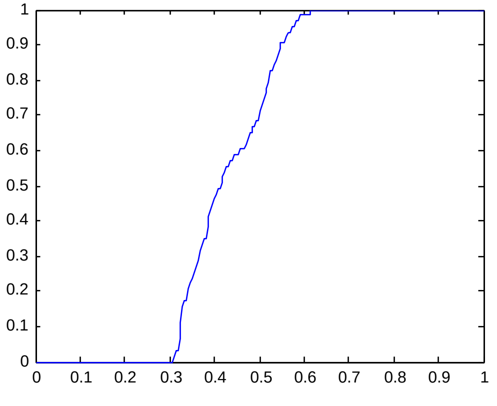


Figure 8-11: Histogram Before Equalization (left) and After Equalization (right)

`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], regardless of the class of the input image.) You can plot this data to get the transformation curve. For example,

```
I = imread('pout.tif');
[J,T] = histeq(I);
figure,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 `imadjdemo`.

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.

See “Linear Filtering” on page 6-4 for more information.

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)
```

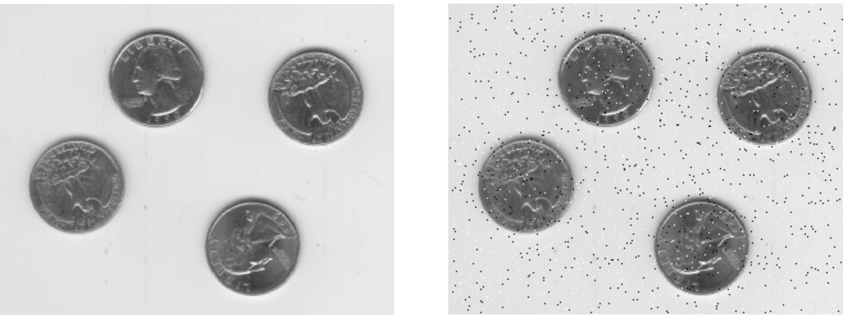


Figure 8-12: Eight.tif Before and After Adding Salt-and-Pepper Noise

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]);
figure, imshow(K)
figure, imshow(L)
```

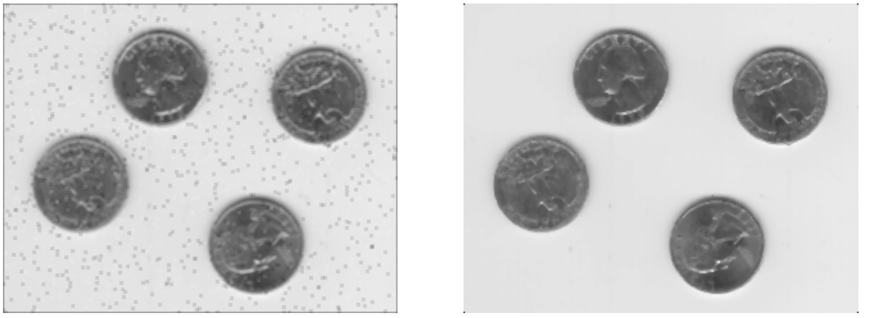


Figure 8-13: Noisy Version of Eight.tif Filtered with Averaging Filter (left) and Median Filter (right)

Median filtering is a specific case of *order-statistic filtering*, also known as *rank filtering*. For information about order-statistic filtering, see the reference page for the `ordfilt2` function.

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` works best when the noise is constant-power ("white") additive noise, such as Gaussian noise. The example below 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)
```

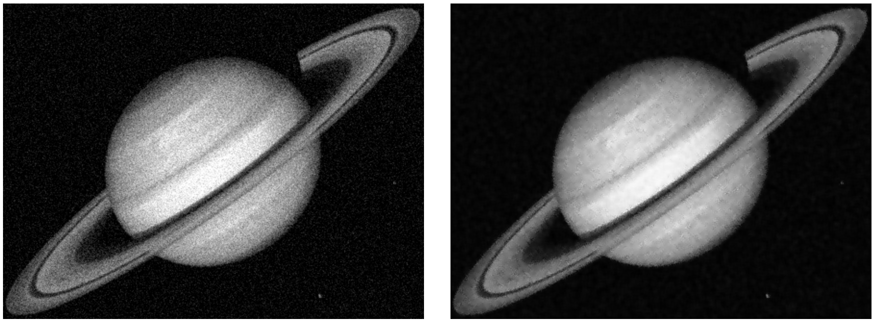


Figure 8-14: Noisy Version of Saturn.tif Before and After Adaptive Filtering

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

9

Binary Image Operations

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).

Words You Need to Know

An understanding of the following terms will help you to use this chapter. For more explanation of this table and others like it, see "Words You Need to Know" in the Preface.

Words	Definitions
Background	The set of black (or `off`) pixels in a binary object.
Binary image	An image containing only black and white pixels. In MATLAB, a binary image is represented by a `uint8` or `double` *logical* matrix containing 0's and 1's (which usually represent black and white, respectively). A matrix is logical when its "logical flag" is turned "on." We often use the variable name `BW` to represent a binary image in memory.
Connected component	A set of white pixels that form a connected group. A connected component is "8-connected" if diagonally adjacent pixels are considered to be touching, otherwise, it is "4-connected." In the context of this chapter, "object" is a synonym for "connected component."
Foreground	The set of white (or `on`) pixels in a binary object.

Words	Definitions
Morphology	A broad set of binary image operations that process images based on shapes. Morphological operations apply a structuring element to an input image, creating an output image of the same size. The most basic morphological operations are dilation and erosion.
Neighborhood	A set of pixels that are defined by their locations relative to the pixel of interest. In binary image operations, a neighborhood can be defined by a structuring element or by using the criterion for a 4- or 8-connected neighborhood.
Object	A set of white pixels that form a connected group. In the context of this chapter, “object” and “connected component” are basically equivalent. See “Connected component” above.
Structuring element	A matrix used to define a neighborhood shape and size for binary image operations, including dilation and erosion. It consists of only 0’s and 1’s and can have an arbitrary shape and size. The pixels with values of 1 define the neighborhood. By choosing a proper structuring element shape, you can construct a morphological operation that is sensitive to specific shapes.

Neighborhoods

Most binary image algorithms work with groups of pixels called *neighborhoods*. A pixel’s neighborhood is some set of pixels that are 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 the image’s 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. Their extent depends on the size of the neighborhood.

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 "Displaying Binary Images" on page 3-7 for more information.

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

- "Morphological Operations" on page 9-5
- "Object-Based Operations" on page 9-11
- "Feature Measurement" on page 9-19
- "Lookup Table Operations" on page 9-21

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 `floor((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.

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

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. Figure 9-1 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.

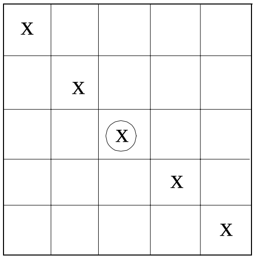


Figure 9-1: A Neighborhood That Will Represented as a Structuring Element

The structuring element is therefore

```
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
```

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. (See the earlier section, "Padding of Borders" on page 9-3.) 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)
```

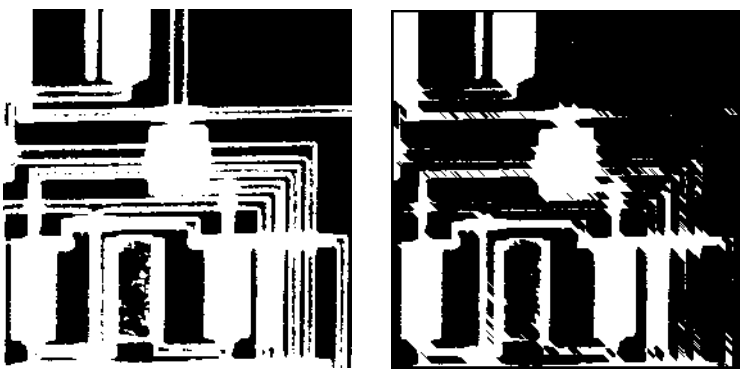


Figure 9-2: Circbw.tif Before and After Erosion with a Diagonal Structuring Element

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.

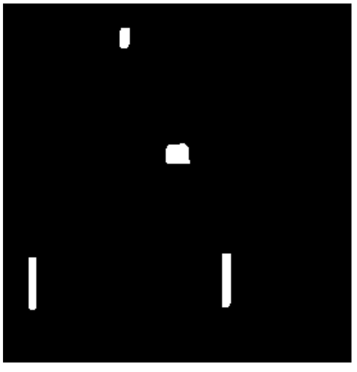
For example, suppose you want to remove all the circuit lines from the original circuit image, leaving only the rectangular outlines of microchips. You can accomplish this through opening.

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);
```

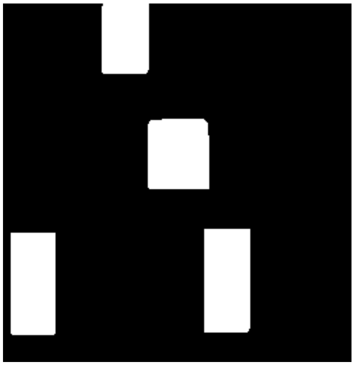
Next, you perform the erosion. This removes all of the lines, but also shrinks the rectangles.

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

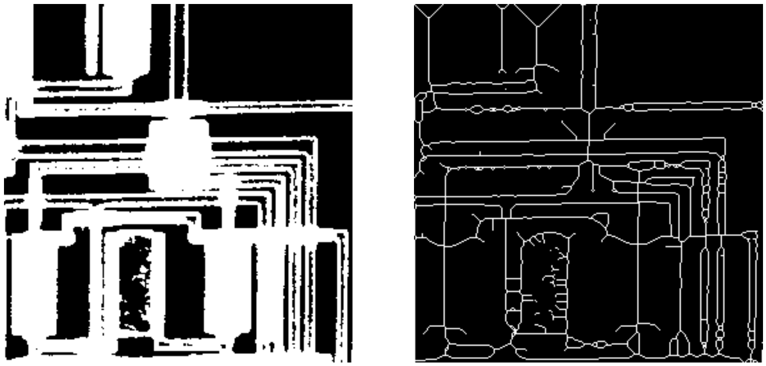


Figure 9-3: Circbw.tif Before and After Skeletonization

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 page for `bwmorph`.

Object-Based Operations

In a binary image, an *object* is any set of connected pixels with the value 1. (meaning that they are "on"). 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.

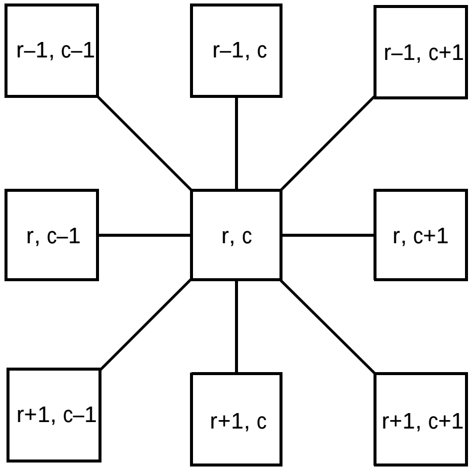


Figure 9-4: An 8-Connected Neighborhood

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

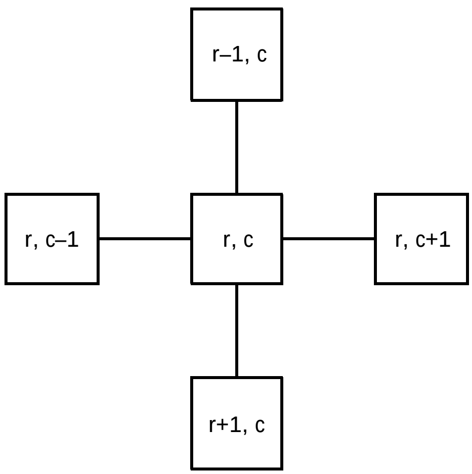


Figure 9-5: A 4-Connected Neighborhood

The type of neighborhood you choose affects the number of objects found 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('circbw.tif');
BW2 = bwperim(BW1);
imshow(BW1)
figure, imshow(BW2)
```

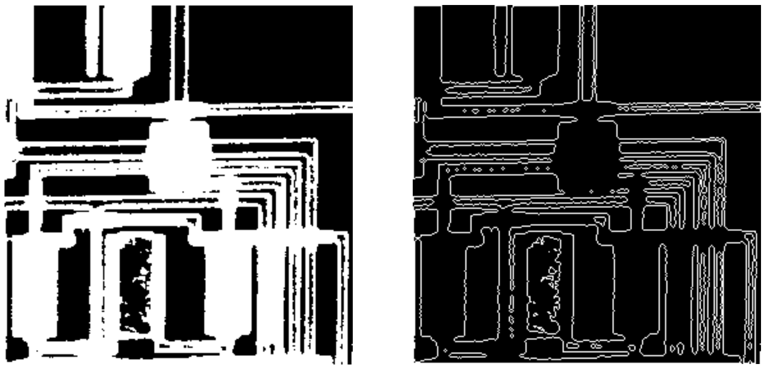


Figure 9-6: Circbw.tif Before and After Perimeter Determination

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 flood-fill operation 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.

For example,

```
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 page for `bwfill` for more information.

Connected-Components Labeling

The `bwlabel` function performs *connected-components labeling*, which is a method for indicating each discrete object in a binary image. You specify an 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')
```

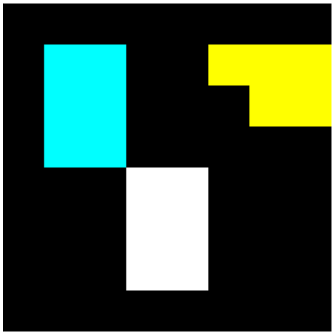


Figure 9-7: A Binary Image Displayed with a Colormap After Connected-Components Labeling

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 page for `bwselect` for more information.

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 page for `imfeature` 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 the percentage area increase in `circbw.tif` that results from 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%.

See the reference page for `bwarea` for more information about the weighting pattern.

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.

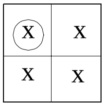
```
BW1 = imread('circbw.tif');
eul = bweuler(BW1,8)

eul =

   -85
```

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

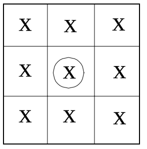
Lookup Table Operations

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.



2-by-2 neighborhood



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)
```

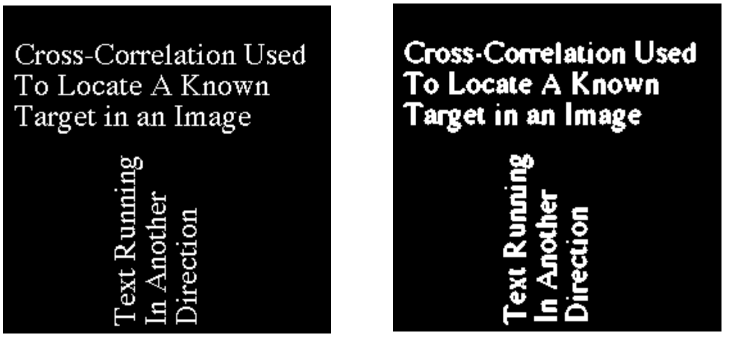


Figure 9-8: Text.tif Before and After Applying a Lookup Table Operation

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

Note 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.

10

Region-Based Processing

Overview

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

- "Specifying a Region of Interest" on page 10-4
- "Filtering a Region" on page 10-7
- "Filling a Region" on page 10-9

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

Words You Need to Know

An understanding of the following terms will help you to use this chapter. For more explanation of this table and others like it, see "Words You Need to Know" in the Preface.

Words	Definitions
Binary mask	A binary image with 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.
Filling a region	A process that "fills" a region of interest by interpolating the pixel values from the borders of the region. This process can be used to make objects in an image seem to disappear as they are replaced with values that blend in with the background area.
Filtering a region	The process of applying a filter to a region of interest. For example, you can apply an intensity adjustment filter to certain regions of an image.
Interpolation	The process by which we estimate an image value at a location in between image pixels.

Words	Definitions
Masked filtering	An operation that applies filtering only to the regions of interest in an image that are identified by a binary mask. Filtered values are returned for pixels where the binary mask contains 1's; unfiltered values are returned for pixels where the binary mask contains 0's.
Region of interest	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. There can be more than one region defined in an image. The regions can be "geographic" in nature, such as polygons that encompass contiguous pixels, or they can be defined by a range of intensities. In the latter case, the pixels are not necessarily contiguous.

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 with 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.

The following subsections discuss methods for creating binary masks:

- "Selecting a Polygon" on page 10-4
- "Other Selection Methods" on page 10-5 (using any binary mask or the `roicolor` function)

Selecting a Polygon

You can use the `roipoly` function to specify a polygonal region of interest. If you call `roipoly` 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**; `roipoly` 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 the interactive syntax of `roipoly` to create a binary mask. The border of the selected region in Figure 10-1, which was created using a mouse, is shown in red.

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

Figure 10-1: A Polygonal Region of Interest Selected Using roipoly

```
imshow(BW)
```

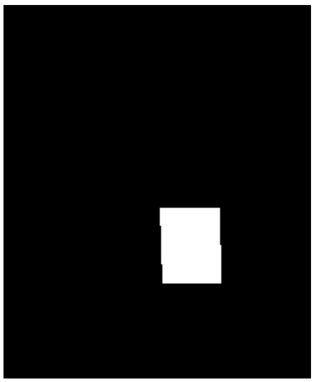


Figure 10-2: A Binary Mask Created for the Region Shown in Figure 10-1.

You can also use `roipoly` noninteractively. See the reference page for `roipoly` for more information.

Other Selection Methods

`roipoly` 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`, filtering only those pixels whose values are greater than 0.5. You can create the appropriate mask with this command.

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

You can also use the `roicolor` function to define the region of interest based on a color or intensity range. For more information, see the reference page for `roicolor`.

Filtering a Region

You can use the `roifilt2` function to process a region of interest. When you call `roifilt2`, you specify an intensity image, a binary mask, and a filter. `roifilt2` 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 in the example in "Selecting a Polygon" on page 10-4 to increase the contrast of the logo on the girl's coat.

```
h = fspecial('unsharp');
I2 = roifilt2(h,I,BW);
imshow(I)
figure, imshow(I2)
```

Figure 10-3: An Image Before and After Using an Unsharp Filter on the Region of Interest.

`roifilt2` also enables you to specify your own 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)
```

Figure 10-4: An Image Brightened Using a Binary Mask Containing Text

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`, [0,65535] for images of class `uint16`). For more information, see the reference page for `roifilt2`.

Filling a Region

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

roifill performs the fill operation 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 roipoly, you select the region of interest with the mouse. When you complete the selection, roifill returns an image with the selected region filled in.

This example uses roifill 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 = roifill;
```

Figure 10-5: A Region of Interest Selected for Filling

```
imshow(I2)
```

Figure 10-6: The Region of Interest Shown in Figure 10-5 Has Been Filled

11

Color

Overview

This chapter describes the toolbox functions that help you work with color image data. Note that "color" includes shades of gray; therefore much of the discussion in this chapter applies to grayscale images as well as color images. The following topics are discussed:

- "Working with Different Screen Bit Depths" on page 11-4
- "Reducing the Number of Colors in an Image" on page 11-6
- "Converting to Other Color Spaces" on page 11-15

For additional information about how MATLAB handles color, see the MATLAB graphics documentation.

Words You Need to Know

An understanding of the following terms will help you to use this chapter. For more explanation of this table and others like it, see "Words You Need to Know" in the Preface.

Words	Definitions
Approximation	The method by which the software chooses replacement colors in the event that direct matches cannot be found. The methods of approximation discussed in this chapter are colormap mapping, uniform quantization, and minimum variance quantization.
Indexed image	An image whose pixel values are direct indices into an RGB colormap. In MATLAB, an indexed image is represented by an array of class `uint8`, `uint16`, or `double`. The colormap is always an m-by-3 array of class `double`. We often use the variable name `X` to represent an indexed image in memory, and `map` to represent the colormap.
Intensity image	An image consisting of intensity (grayscale) values. In MATLAB, intensity images are represented by an array of class `uint8`, `uint16`, or `double`. While intensity images are not stored with colormaps, MATLAB uses a system colormap to display them. We often use the variable name `I` to represent an intensity image in memory. This term is synonymous with the term "grayscale".

Words	Definitions
RGB image	An image in which each pixel is specified by three values — one each for the red, blue, and green components of the pixel's color. In MATLAB, an RGB image is represented by an m-by-n-by-3 array of class `uint8`, `uint16`, or `double`. We often use the variable name `RGB` to represent an RGB image in memory.
Screen bit depth	The number of bits per screen pixel.
Screen color resolution	The number of distinct colors that can be produced by the screen.

Working with Different Screen Bit Depths

Most computer displays use 8, 16, or 24 bits per screen pixel. The number of bits per screen pixel determines the display's *screen bit depth*. The screen bit depth determines the *screen color resolution*, which is how many distinct colors the display can produce. Regardless of the number of colors your system can display, MATLAB can store and process images with very high bit depths: 2^{24} colors for `uint8` RGB images, 2^{48} colors for `uint16` RGB images, and 2^{159} for `double` RGB images. These images display best on systems with 24-bit color, but usually look fine on 16-bit systems as well. This section describes the different screen bit depths and how to determine the screen bit depth of your display.

To determine your system's screen bit depth, enter this command at the MATLAB prompt.

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

MATLAB returns an integer representing the number of bits per screen pixel.

A 24-bit display provides optimal color resolution. It uses 8 bits for each of the three color components, resulting in 256 (i.e., 2^8) levels each of red, green, and blue. This supports 16,777,216 (i.e., 2^{24}) different colors. (Of these colors, 256 are shades of gray. Shades of gray occur where R=G=B.) The 16 million possible colors supported by 24-bit display can render a life-like image.

16-bit displays also support a large number of colors. They usually use 5 bits for each color component, resulting in 32 (i.e., 2^5) levels each of red, green, and blue. This supports 32,768 (i.e., 2^{15}) distinct colors (of which 32 are shades of gray). Alternatively, the extra bit can be used to increase the number of levels of green displayed. In this case, the number of different colors supported by a 16-bit display is actually 64,536 (i.e. 2^{16}).

8-bit displays support a 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 if all 256 shades of gray are used, they take up all of the available color slots.)

Note It is possible that your system's screen bit depth is 32 bits per pixel. Normally, 32-bit displays use 24 bits for color information and 8 bits for transparency data (alpha channel). So, although the command, `get(0,'ScreenDepth')` returns the value 32, MATLAB does not currently support transparency data.

Depending on your system, you may be able to choose the screen bit depth you want to use. (There may be trade-offs between screen bit depth and screen color resolution.) In general, 24-bit display mode produces the best results. If you need to use a lower screen bit depth, 16-bit is generally preferable to 8-bit. However, keep in mind that a 16-bit display has certain limitations, such as:

- 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, which provides up to 256 shades of gray.

The next section shows how to reduce the number of colors used by an image.

Reducing the Number of Colors in an Image

This section describes how to reduce the number of colors in an indexed or RGB image. A discussion is also included about dithering, which is used by the toolbox's color-reduction functions (see below.) Dithering is used to increase the apparent number of colors in an image.

The table below summarizes the Image Processing Toolbox functions for color reduction.

Function	Purpose
`imapprox`	Reduces the number of colors used by an indexed image, enabling you specify the number of colors in the new colormap.
`rgb2ind`	Converts an RGB image to an indexed image, enabling you to specify the number of colors to store in the new colormap.

.On systems with 24-bit color displays, RGB (truecolor) images can display up to 16,777,216 (i.e., 2^{24}) colors. On systems with lower screen bit depths, RGB images still displays reasonably well, because MATLAB automatically uses color approximation and dithering if needed.

Indexed images, however, may cause problems if they have a large number of colors. In general, you should limit indexed images to 256 colors for the following reasons.

- On systems with 8-bit display, indexed images with more than 256 colors will need to be dithered or mapped and, therefore, may not display well.
- 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, but generally uses an array of class `double` instead, making the storage size of the image much larger (each pixel uses 64 bits).
- Most image file formats limit indexed images to 256 colors. If you write an indexed image with more than 256 colors (using `imwrite`) to a format that does not support more than 256 colors, you will receive an error.

Using rgb2ind

`rgb2ind` converts an RGB image to an indexed image, reducing the number of colors in the process. This function provides the following methods for approximating the colors in the original image:

- Quantization
  - Uniform quantization
  - Minimum variance 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 “Dithering” on page 11-13 for a description of dithering and how to enable or disable it.

Quantization

Reducing the number of colors in an image involves *quantization*. The function `rgb2ind` uses quantization as part of its color reduction algorithm. `rgb2ind` supports two quantization methods: *uniform quantization* and *minimum variance quantization*.

An important term in discussions of image quantization is *RGB color cube*, which is used frequently throughout this section. The RGB color cube is a three-dimensional array of all of the colors that are defined for a particular data type. Since RGB images in MATLAB can be of type `uint8`, `uint16`, or `double`, three possible color cube definitions exist. For example, if an RGB image is of class `uint8`, 256 values are defined for each color plane (red, blue, and green), and, in total, there will be 2^{24} (or 16,777,216) colors defined by the color cube. This color cube is the same for all `uint8` RGB images, regardless of which colors they actually use.

The `uint8`, `uint16`, and `double` color cubes all have the same range of colors. In other words, the brightest red in an `uint8` RGB image displays the same as the brightest red in a `double` RGB image. The difference is that the `double` RGB color cube has many more shades of red (and many more shades of all colors). Figure 11-1, below, shows an RGB color cube for a `uint8` image.

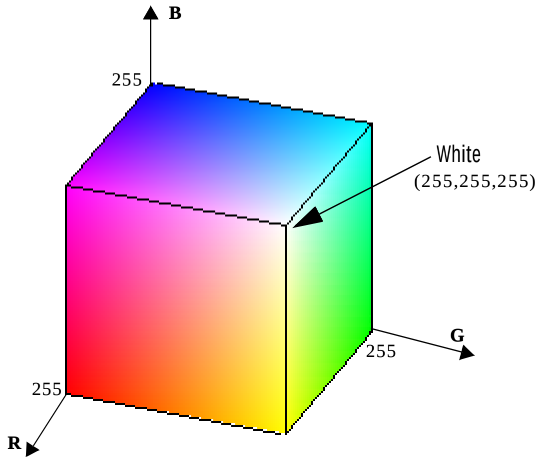


Figure 11-1: RGB Color Cube for uint8 Images

Quantization involves dividing the RGB color cube into a number of smaller boxes, and then mapping all colors that fall within each box to the color value at the *center* of that box.

Uniform quantization and minimum variance quantization differ in the approach used to divide up the RGB color cube. With uniform quantization, the color cube is cut up into equal-sized boxes (smaller cubes). With minimum variance quantization, the color cube is cut up into boxes (not necessarily cubes) of different sizes; the sizes of the boxes depend on how the colors are distributed in the image.

Uniform Quantization. To perform uniform quantization, call `rgb2ind` and specify a *tolerance*. The tolerance determines the size of the cube-shaped boxes into which the RGB color cube is divided. The allowable range for a tolerance setting is [0,1]. For example, if you specify a tolerance of `0.1`, the edges of the boxes are one-tenth the length of the RGB color cube and the maximum total number of boxes is

```
n = (floor(1/tol)+1)^3
```

The commands below perform uniform quantization with a tolerance of 0.1.

```
RGB = imread('flowers.tif');
[x,map] = rgb2ind(RGB, 0.1);
```

Figure 11-2 illustrates uniform quantization of a `uint8` image. For clarity, the figure shows a two-dimensional slice (or color plane) from the color cube where Red=0, and Green and Blue range from 0 to 255. The actual pixel values are denoted by the centers of the x's.

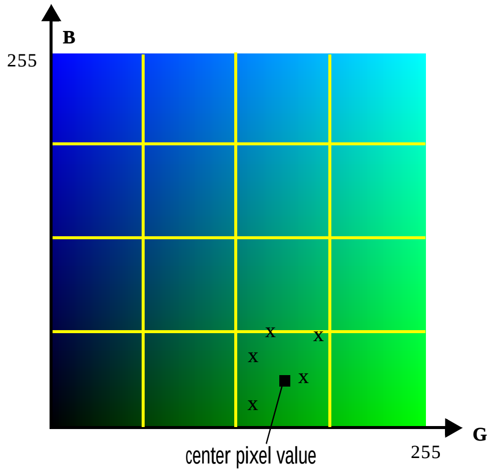


Figure 11-2: Uniform Quantization on a Slice of the RGB Color Cube

After the color cube has been divided, all empty boxes are thrown out. Therefore, only one of the boxes in Figure 11-2 is used to produce a color for the colormap. As shown earlier, the maximum length of a colormap created by uniform quantization can be predicted, but the colormap can be smaller than the prediction because `rgb2ind` removes any colors that do not appear in the input image.

Minimum Variance Quantization. To perform minimum variance quantization, call `rgb2ind` and specify the maximum number of colors in the output image's colormap. The number you specify determines the number of boxes into which the RGB color cube is divided. These commands use minimum variance quantization to create an indexed image with 185 colors.

```
RGB = imread('flowers.tif');
```

```
[X,map] = rgb2ind(RGB,185);
```

Minimum variance quantization works by associating pixels into groups based on the variance between their pixel values. For example, a set of blue pixel values may be grouped together because none of their values is greater than 5 from the center pixel of the group.

In minimum variance quantization, the boxes that divide the color cube vary in size, and do not necessarily fill the color cube. If some areas of the color cube do not have pixels, there are no boxes in these areas.

While you set the number of boxes, `n`, to be used by `rgb2ind`, the placement is determined by the algorithm as it analyzes the color data in your image. Once the image is divided into `n` optimally located boxes, the pixels within each box are mapped to the pixel value at the center of the box, as in uniform quantization.

The resulting colormap usually has the number of entries you specify. This is 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 uses fewer colors than the number you specify, the output colormap will have fewer than `n` colors, and the output image will contain all of the colors of the input image.

Figure 11-3 shows the same two-dimensional slice of the color cube as was used in Figure 11-2 (for demonstrating uniform quantization). Eleven boxes have been created using minimum variance quantization.

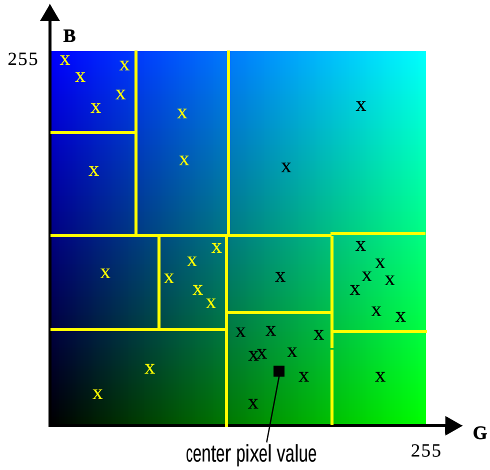


Figure 11-3: Minimum Variance Quantization on a Slice of the RGB Color Cube

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. It allocates fewer entries to colors that appear infrequently. As a result, the accuracy of the colors is higher than with uniform quantization. For example, if the input image has many shades of green and few shades of red, there will be more greens than reds in the output colormap. Note that the computation for minimum variance quantization takes longer than that for uniform quantization.

Colormap Mapping

If you specify an actual colormap to use, `rgb2ind` uses *colormap mapping* (instead of quantization) 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 mapping them all to 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 used for the two images is created on the fly using the MATLAB function `colorcube`, which creates an RGB colormap containing the number of colors that you specify. (`colorcube` always creates the same colormap for a given number of colors.) Because the colormap includes colors all throughout the RGB color cube, the output images can reasonably approximate the input images.

```
RGB1 = imread('autumn.tif');
RGB2 = imread('flowers.tif');
X1 = rgb2ind(RGB1,colorcube(128));
X2 = rgb2ind(RGB2,colorcube(128));
```

Note The function `subimage` is also helpful for displaying multiple indexed images. For more information see "Displaying Multiple Images in the Same Figure" on page 3-22 or the reference page for `subimage`.

Using imapprox

Use `imapprox` when you need to reduce the number of colors in an indexed image. `imapprox` is based on `rgb2ind` and uses the same approximation methods. Essentially, `imapprox` first calls `ind2rgb` 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);
imshow(Y, newmap);
```

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 "Dithering" on page 11-13 for a description of dithering and how to enable or disable it.

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 number of colors in the output image. Dithering changes the colors of pixels in a neighborhood so that the average color in each neighborhood approximates the original RGB color.

For an example of how dithering works, consider an image that contains a number of dark pink pixels for which there is no exact match in the colormap. To create the appearance of this shade of pink, the Image Processing Toolbox selects a combination of colors from the colormap, that, taken together as a six-pixel group, approximate the desired shade of pink. From a distance, the pixels appear to be correct shade, but if you look up close at the image, you can see a blend of other shades, perhaps red and pale pink pixels. The commands below load a 24-bit image, and then use `rgb2ind` to create two indexed images with just eight colors each.

```
rgb=imread('lily.tif');
imshow(rgb);
[X_no_dither,map]=rgb2ind(rgb,8,'nodither');
[X_dither,map]=rgb2ind(rgb,8,'dither');
figure, imshow(X_no_dither,map);
figure, imshow(X_dither,map);
```

Original Image   Without Dithering   With Dithering

Figure 11-4: Examples of Color Reduction with and Without Dithering

Notice that the dithered image has a larger number of apparent colors but is somewhat fuzzy-looking. The image produced without dithering has fewer apparent colors, but an improved spatial resolution when compared to the dithered image. One risk in doing color reduction without dithering is that the new image my contain false contours (see the rose in the upper-right corner).

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, where the colormap is stored in RGB format). 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. However, the toolbox provides support for other color spaces though a set of conversion functions. You can use these functions to convert between RGB and the following color spaces:

- National Television Systems Committee (NTSC)
- YCbCr
- Hue, saturation, value (HSV)

These section describes these color spaces and the conversion routines for working with them

- "NTSC Color Space"
- "YCbCr Color Space"
- "HSV Color Space"

NTSC Color Space

The NTSC color space 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 the 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 `flowers` image to NTSC format.

```
RGB = imread('flowers.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. In fact, 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);
```

Note In YIQ color space, I is one of the two color components, not the grayscale component.

YCbCr Color Space

The YCbCr color space 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. 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 (nonimage) 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 Color Space

The HSV color space (hue, saturation, value) is often used by people who are selecting 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. The functions rgb2hsv and hsv2rgb convert images between the RGB and HSV color spaces.

As hue varies from 0 to 1.0, the corresponding colors vary from red, through yellow, green, cyan, blue, and magenta, back to red, so that there are actually red values both at 0 and 1.0. As saturation varies from 0 to 1.0, the corresponding colors (hues) 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.

Figure 11-5 illustrates the HSV color space.

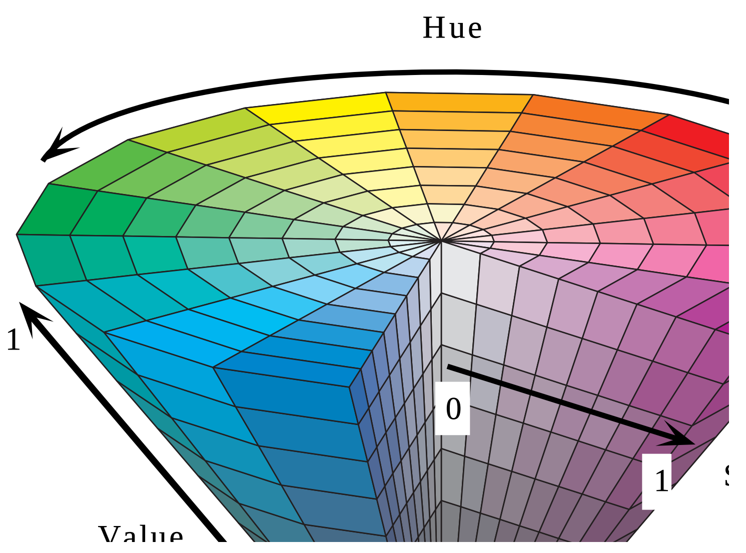


Figure 11-5: Illustration of the HSV Color Space

The function rgb2hsv converts colormaps or RGB images to the HSV color space. hsv2rgb performs the reverse operation. These commands convert an RGB image to HSV color space.

```
RGB = imread('flowers.tif');
HSV = rgb2hsv(RGB);
```

For closer inspection of the HSV color space, the next block of code displays the separate color planes (hue, saturation, and value) of an HSV image.

```
RGB=reshape(ones(64,1)*reshape(jet(64),1,192),[64,64,3]);
HSV=rgb2hsv(RGB);
H=HSV(:,:,1);
S=HSV(:,:,2);
V=HSV(:,:,3);
imshow(H)
figure, imshow(S);
figure, imshow(V);
figure, imshow(RGB);
```

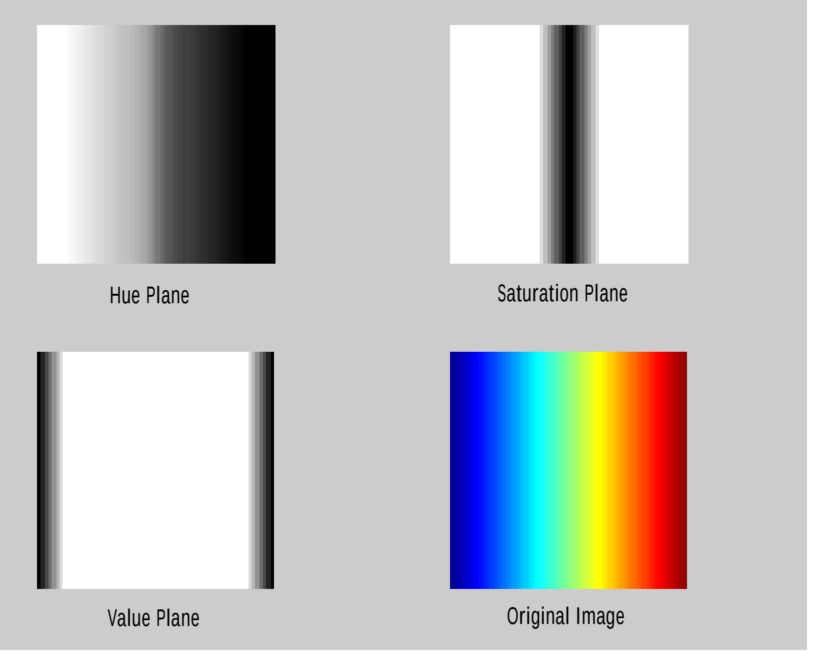


Figure 11-6: The Separated Color Planes of an HSV Image

The images in Figure 11-6 can be scrutinized for a better understanding of how the HSV color space works. As you can see by looking at the hue plane image, hue values make a nice linear transition from high to low. If you compare the hue plane image against the original image, you can see that shades of deep blue have the highest values, and shades of deep red have the lowest values. (In actuality, there are values of red on both ends of the hue scale, which you can see if you look back at the model of the HSV color space in Figure 11-5. To avoid confusion, our sample image uses only the red values from the *beginning* of the hue range.) Saturation can be thought of as the purity of a color. As the saturation plane image shows, the colors with the highest saturation have the highest values and are represented as white. In the center of the saturation image, notice the various shades of gray. These correspond to a mixture of colors; the cyans, greens, and yellow shades are mixtures of true colors. Value is roughly equivalent to brightness, and you will notice that the brightest areas of the value plane image correspond to the brightest colors in the original image.

12

Function 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.

Functions by Category

The tables below list all functions in the Image Processing Toolbox, plus a few functions in MATLAB that are especially useful for image processing. All of the functions listed have reference entries in this *User's Guide*, with the following exceptions:

- Most MATLAB functions. To see the reference entries for most of the MATLAB functions listed here, see the MATLAB Function Reference. The MATLAB functions `imread`, `imfinfo`, and `imwrite` have entries in this reference because they are essential to image file I/O.
- The Image Processing Toolbox demo functions and slideshow functions. For information about any of these functions, see "Image Processing Demos" in the Preface.

Image Display	
`colorbar`	Display colorbar. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`getimage`	Get image data from axes
`image`	Create and display image object. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`imagesc`	Scale data and display as image. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`immovie`	Make movie from multiframe indexed image
`imshow`	Display image

Image Display (Continued)	
`montage`	Display multiple image frames as rectangular montage
`subimage`	Display multiple images in single figure
`truesize`	Adjust display size of image
`warp`	Display image as texture-mapped surface
`zoom`	Zoom in and out of image or 2-D plot. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)

Image File I/O	
`imfinfo`	Return information about image file. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`imread`	Read image file. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`imwrite`	Write image file. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)

Geometric Operations	
`imcrop`	Crop image
`imresize`	Resize image

Geometric Operations (Continued)	
`imrotate`	Rotate image
`interp2`	2-D data interpolation. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)

Pixel Values and Statistics	
`corr2`	Compute 2-D correlation coefficient
`imcontour`	Create contour plot of image data
`imfeature`	Compute feature measurements for image regions
`imhist`	Display histogram of image data
`impixel`	Determine pixel color values
`improfile`	Compute pixel-value cross-sections along line segments
`mean2`	Compute mean of matrix elements
`pixval`	Display information about image pixels
`std2`	Compute standard deviation of matrix elements

Image Analysis	
`edge`	Find edges in intensity image
`qtdecomp`	Perform quadtree decomposition
`qtgetblk`	Get block values in quadtree decomposition
`qtsetblk`	Set block values in quadtree decomposition

Image Enhancement	
`histeq`	Enhance contrast using histogram equalization
`imadjust`	Adjust image intensity values or colormap
`imnoise`	Add noise to an image
`medfilt2`	Perform 2-D median filtering
`ordfilt2`	Perform 2-D order-statistic filtering
`wiener2`	Perform 2-D adaptive noise-removal filtering

Linear Filtering	
conv2	Perform 2-D convolution. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
convmtx2	Compute 2-D convolution matrix
convn	Perform N-D convolution. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
filter2	Perform 2-D filtering. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
fspecial	Create predefined filters

Linear 2-D Filter Design	
freqspace	Determine 2-D frequency response spacing. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
freqz2	Compute 2-D frequency response
fsamp2	Design 2-D FIR filter using frequency sampling
ftrans2	Design 2-D FIR filter using frequency transformation
fwind1	Design 2-D FIR filter using 1-D window method
fwind2	Design 2-D FIR filter using 2-D window method

Image Transforms	
`dct2`	Compute 2-D discrete cosine transform
`dctmtx`	Compute discrete cosine transform matrix
`fft2`	Compute 2-D fast Fourier transform. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`fftn`	Compute N-D fast Fourier transform. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`fftshift`	Reverse quadrants of output of FFT. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`idct2`	Compute 2-D inverse discrete cosine transform
`ifft2`	Compute 2-D inverse fast Fourier transform. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`ifftn`	Compute N-D inverse fast Fourier transform. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`iradon`	Compute inverse Radon transform
`phantom`	Generate a head phantom image
`radon`	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

Binary Image Operations (Continued)	
`bwperim`	Determine perimeter of objects in binary image
`bwselect`	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	
`roicolor`	Select region of interest, based on color
`roifill`	Smoothly interpolate within arbitrary region
`roifilt2`	Filter a region of interest
`roipoly`	Select polygonal region of interest

Colormap Manipulation	
`brighten`	Brighten or darken colormap. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`cmpermute`	Rearrange colors in colormap

Colormap Manipulation (Continued)	
`cmunique`	Find unique colormap colors and corresponding image
`colormap`	Set or get color lookup table. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`imapprox`	Approximate indexed image by one with fewer colors
`rgbplot`	Plot RGB colormap components. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)

Color Space Conversions	
`hsv2rgb`	Convert HSV values to RGB color space. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`ntsc2rgb`	Convert NTSC values to RGB color space
`rgb2hsv`	Convert RGB values to HSV color space. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`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	
dither	Convert image using dithering
double	Convert data to double precision. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
gray2ind	Convert intensity image to indexed image
grayslice	Create indexed image from intensity image by thresholding
im2bw	Convert image to binary image by thresholding
im2double	Convert image array to double precision
im2uint16	Convert image array to 16-bit unsigned integers
im2uint8	Convert image array to 8-bit unsigned integers
ind2gray	Convert indexed image to intensity image
ind2rgb	Convert indexed image to RGB image
isbw	Return true for binary image
isgray	Return true for intensity image
isind	Return true for indexed image
isrgb	Return true for RGB image

Image Types and Type Conversions (Continued)	
`mat2gray`	Convert matrix to intensity image
`rgb2gray`	Convert RGB image or colormap to grayscale
`rgb2ind`	Convert RGB image to indexed image
`uint16`	Convert data to unsigned 16-bit integers. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`uint8`	Convert data to unsigned 8-bit integers. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)

Toolbox Preferences	
`iptgetpref`	Get value of Image Processing Toolbox preference
`iptsetpref`	Set value of Image Processing Toolbox preference

Demos	
`dctdemo`	2-D DCT image compression demo
`edgedemo`	Edge detection demo
`firdemo`	2-D FIR filtering and filter design demo
`imadjdemo`	Intensity adjustment and histogram equalization demo

Demos (Continued)	
`nrfiltdemo`	Noise reduction filtering demo
`qtdemo`	Quadtree decomposition demo
`roidemo`	Region-of-interest processing demo

Slide Shows	
`ipss001`	Region labeling of steel grains
`ipss002`	Feature-based logic
`ipss003`	Correction of nonuniform illumination

Alphabetical List of Functions

Purpose Perform neighborhood operations on binary images, using lookup tables

Syntax

```
A = applylut(BW,lut)
```

Description `A = 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 `makelut`. 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 `uint8` or `double`. If the elements of `lut` are all integers between 0 and 255 (regardless of the class of `lut`), then the class of `A` is `uint8`; otherwise, the class of `A` is `double`.

Algorithm `applylut` 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, `length(lut)` is 16. There are four pixels in each neighborhood, and two possible states for each pixel, so the total number of permutations is $2^4 = 16$.

To produce the matrix of indices, `applylut` convolves the binary image `BW` with this matrix.

```
8    2
4    1
```

The resulting convolution contains integer values in the range [0,15]. `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 [1,16]. 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 nine 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.

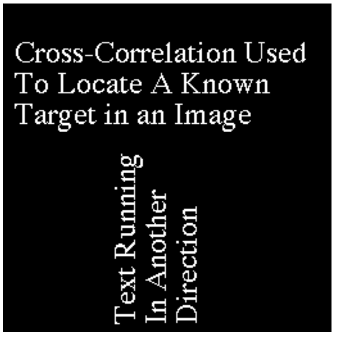
```
256    32     4
128    16     2
 64     8     1
```

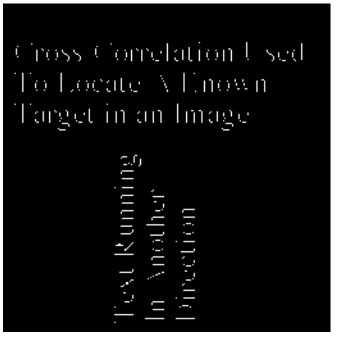
The resulting convolution contains integer values in the range [0,511]. `applylut` uses the central part of the convolution, of the same size as `BW`, and adds 1 to each value to shift the range to [1,512]. It then constructs `A` by replacing the values in the cells of the index matrix with the values 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`

Purpose

Determine block size for block processing

Syntax

```
siz = bestblk([m n],k)
[mb,nb] = bestblk([m n],k)
```

Description

`siz = bestblk([m n],k)` 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. `siz` is a 1-by-2 vector containing the row and column dimensions for the block.

`[mb,nb] = bestblk([m n],k)` returns the row and column dimensions for the block in `mb` and `nb`, respectively.

Algorithm

`bestblk` returns the optimal block size given `m`, `n`, and `k`. The algorithm for determining `siz` is:

- 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.

The same algorithm is then repeated for `n`.

Example

```
siz = bestblk([640 800],72)

siz =

     64    50
```

See Also

`blkproc`

blkproc

Purpose

Implement distinct block processing for an image

Syntax

```
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',...)
```

Description

`B = blkproc(A,[m n],fun)` 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` is a function that accepts an `m`-by-`n` matrix, `x`, and return a matrix, vector, or scalar `y`.

```
y = fun(x)
```

`blkproc` 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`.

`B = blkproc(A,[m n],fun,P1,P2,...)` passes the additional parameters `P1,P2,...`, to `fun`.

`B = blkproc(A,[m n],[mborder nborder],fun,...)` defines an overlapping border around the blocks. `blkproc` 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)`. `blkproc` pads the border with zeros, if necessary, on the edges of `A`. `fun` should operate on the extended block.

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.

```
B = blkproc(A,[4 6],[2 3],fun,...)
```

`B = blkproc(A,'indexed',...)` processes `A` as an indexed image, padding with zeros if the class of `A` is `uint8` or `uint16`, 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`.

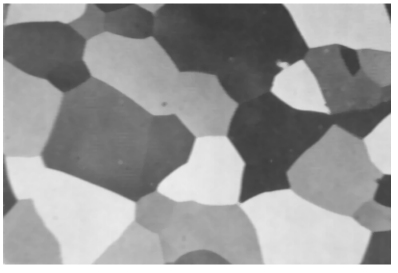
Example

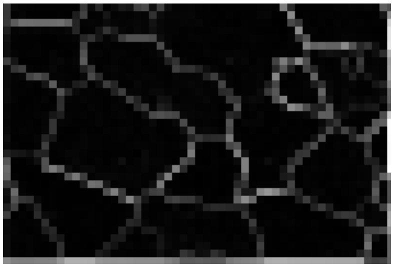
fun can be a function_handle created using @. This example uses blkproc to compute the 2-D DCT of each 8-by-8 block to the standard deviation of the elements in that block.

```
I = imread('cameraman.tif');
fun = @dct2;
J = blkproc(I,[8 8],fun);
imagesc(J), colormap(hot)
```

fun can also be an inline object. 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');
fun = inline('std2(s)*ones(size(x))');
I2 = blkproc(I,[8 8],'std2(x)*ones(size(x))');
imshow(I)
figure, imshow(I2,[]);
```





See Also

colfilt, nlfilter, inline

brighten

Purpose Brighten or darken a colormap

Syntax

```
brighten(beta)
newmap = brighten(beta)
newmap = brighten(map,beta)
brighten(fig,beta)
```

Description `brighten(beta)` replaces the current colormap with a brighter or darker map that has essentially the same colors. The map is brighter if $0 <$ `beta` ≤ 1 and darker if $-1 \leq$ `beta` < 0.

`brighten(beta)` followed by `brighten(–beta)` restores the original map.

`newmap = brighten(beta)` returns a brighter or darker version of the current colormap without changing the display.

`newmap = brighten(map,beta)` returns a brighter or darker version of the specified colormap without changing the display.

`brighten(fig,beta)` brightens all of the objects in the figure `fig`.

Remarks `brighten` is a function in MATLAB.

See Also `imadjust`, `rgbplot`

`colormap` in the MATLAB Function Reference

Purpose

Compute the area of the objects in a binary image

Syntax

```
total = bwarea(BW)
```

Description

`total = bwarea(BW)` estimates the area of the objects in binary image `BW`. `total` 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

`BW` can be of class `uint8` or `double`. `total` is of class `double`.

Algorithm

`bwarea` 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:

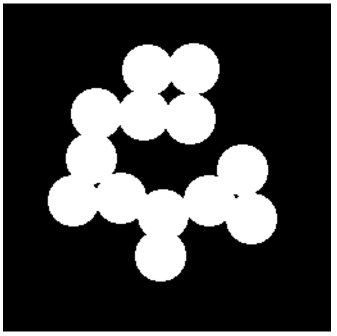
- Patterns with zero on pixels (area = 0)
- Patterns with one on pixel (area = 1/4)
- Patterns with two adjacent on pixels (area = 1/2)
- Patterns with two diagonal on pixels (area = 3/4)
- Patterns with three on pixels (area = 7/8)
- Patterns with all four on pixels (area = 1)

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 [1] Pratt, William K. *Digital Image Processing*. New York: John Wiley & Sons, Inc., 1991. p. 634.

Purpose Compute the Euler number of a binary image

Syntax

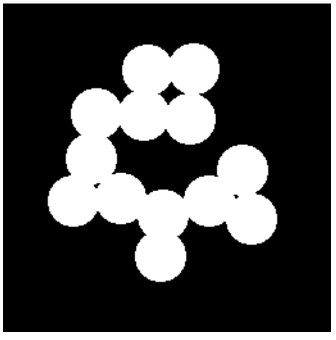
```
eul = bweuler(BW,n)
```

Description `eul = bweuler(BW,n)` returns the Euler number for the binary image `BW`. `eul` is a scalar whose value is the total number of objects in the image minus the total number of holes in those objects. `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.

Class Support `BW` can be of class `uint8` or `double`. `eul` is of class `double`.

Example

```
BW = imread('circles.tif');
imshow(BW);
```



```
bweuler(BW)

ans =

    -2
```

Algorithm `bweuler` computes the Euler number by considering patterns of convexity and concavity in local 2-by-2 neighborhoods. See [2] for a discussion of the algorithm used.

See Also `bwmorph`, `bwperim`

References

[1] Horn, Berthold P. K., *Robot Vision*. New York: McGraw-Hill, 1986. pp. 73-77.

[2] Pratt, William K. *Digital Image Processing*. New York: John Wiley & Sons, Inc., 1991. p. 633.

Purpose

Fill background regions in a binary image

Syntax

```
BW2 = bwfill(BW1,c,r,n)
BW2 = bwfill(BW1,n)
[BW2,idx] = bwfill(...)

BW2 = bwfill(x,y,BW1,xi,yi,n)
[x,y,BW2,idx,xi,yi] = bwfill(...)

BW2 = bwfill(BW1,'holes',n)
[BW2,idx] = bwfill(BW1,'holes',n)
```

Description

`BW2 = bwfill(BW1,c,r,n)` performs a flood-fill operation on the input binary image `BW1`, starting from the pixel (`r`,`c`). If `r` and `c` are equal-length vectors, the fill is performed in parallel from the starting pixels (`r(k)`,`c(k)`). `n` 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 `BW1` comprises the on pixels (i.e., having value of 1).

`BW2 = bwfill(BW1,n)` displays the image `BW1` on the screen and lets you select the starting points using the mouse. If you omit `BW1`, `bwfill` 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.

`[BW2,idx] = bwfill(...)` returns the linear indices of all pixels filled by `bwfill`.

`BW2 = bwfill(x,y,BW1,xi,yi,n)` 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.

`[x,y,BW2,idx,xi,yi] = bwfill(...)` returns the `XData` and `YData` in `x` and `y`; the output image in `BW2`; linear indices of all filled pixels in `idx`; and the fill starting points in `xi` and `yi`.

`BW2 = bwfill(BW1,'holes',n)` fills the holes in the binary image `BW1`. `bwfill` automatically determines which pixels are in object holes, and then changes the value of those pixels from 0 to 1. `n` defaults to 8 if you omit the argument.

`[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`.

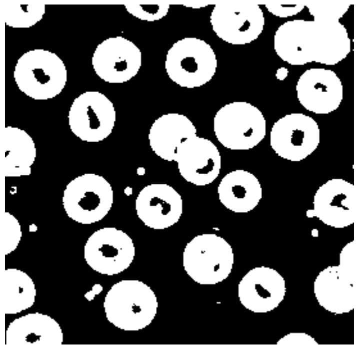
Class Support

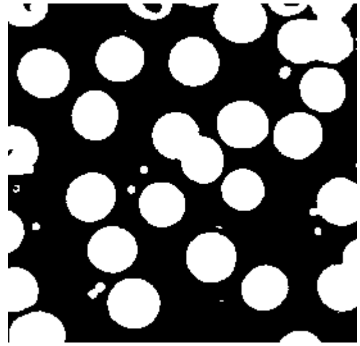
The input image `BW1` can be of class `double` or `uint8`. The output image `BW2` is of class `uint8`.

Example

```
BW1 = [1     0     0     0     0     0     0     0
       1     1     1     1     1     0     0     0
       1     0     0     0     1     0     1     0
       1     0     0     0     1     1     1     0
       1     1     1     1     0     1     1     1
       1     0     0     1     1     0     1     0
       1     0     0     0     1     0     1     0
       1     0     0     0     1     1     1     0]

BW2 = bwfill(BW1,3,3,8)

BW2 =

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

I = imread('blood1.tif');
BW3 = ~im2bw(I);
BW4 = bwfill(BW3,'holes');
imshow(BW3)
```

```
figure, imshow(BW4)
```





See Also bwselect, roifill

bwlabel

Purpose Label connected components in a binary image

Syntax

```
L = bwlabel(BW,n)
[L,num] = bwlabel(BW,n)
```

Description `L = 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] = bwlabel(BW,n)` returns in `num` the number of connected objects found in `BW`.

Class Support The input image `BW` can be of class `double` or `uint8`. The output matrix `L` is of class `double`.

Remarks You can use the MATLAB `find` function in conjunction with `bwlabel` 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] = find(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.

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     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     3     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

[1] Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I.* Addison-Wesley, 1992. pp. 28-48.

bwmorph

Purpose Perform morphological operations on binary images

Syntax

```
BW2 = bwmorph(BW1,operation)
BW2 = bwmorph(BW1,operation,n)
```

Description

`BW2 = bwmorph(BW1,`*`operation`*`)` applies a specific morphological operation to the binary image `BW1`.

`BW2 = bwmorph(BW1,`*`operation`*`,n)` applies the operation `n` times. `n` can be `Inf`, in which case the operation is repeated until the image no longer changes.

operation is a string that can have one of the values listed below.

'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,

```
1  0  0                  1  0  0
1  0  1     becomes      1  1  1
0  0  1                  0  0  1
```

`'clean'` removes isolated pixels (individual 1's that are surrounded by 0's), such as the center pixel in this pattern.

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

`'close'` performs binary closure (dilation followed by erosion).

`'diag'` 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
```

`'dilate'` performs dilation using the structuring element `ones(3)`.

`'erode'` performs erosion 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
```

`'majority'` 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.

`'shrink'`, 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` = `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` = `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.

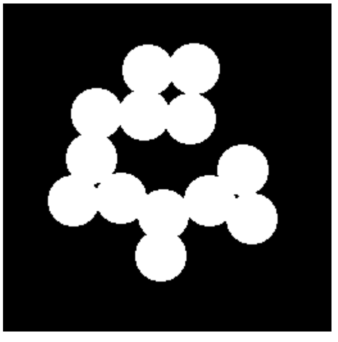
`'tophat'` ("top hat") returns the image minus the binary opening of the image.

Class Support

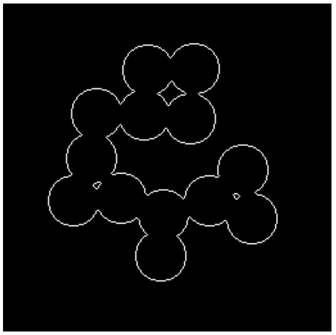
The input image `BW1` can be of class `double` or `uint8`. The output image `BW2` is of class `uint8`.

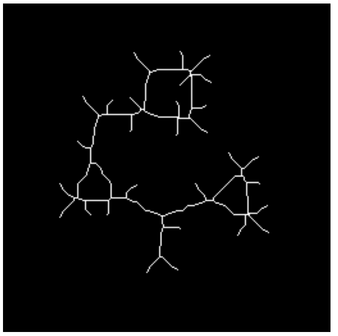
Example

```
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

[1] Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I*. Addison-Wesley, 1992.

[2] Pratt, William K. *Digital Image Processing*. John Wiley & Sons, Inc., 1991.

bwperim

Purpose Determine the perimeter of the objects in a binary image

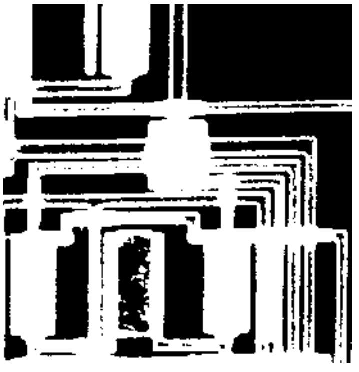
Syntax `BW2 = bwperim(BW1,n)`

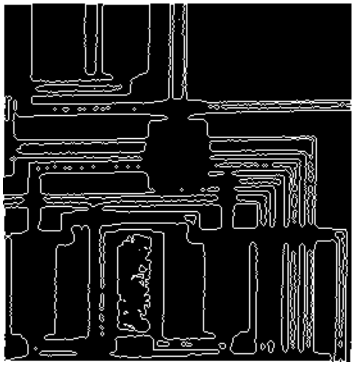
Description `BW2 = bwperim(BW1,n)` returns a binary image containing only the perimeter pixels of objects in the input image `BW1`. A pixel is part of the perimeter if its value is 1 and there is at least one zero-valued pixel in its neighborhood. `n` can have a value of either 4 or 8, where 4 specifies 4-connected neighborhoods and 8 specifies 8-connected neighborhoods; if the argument is omitted, it defaults to 4.

Class Support The input image `BW1` can be of class `double` or `uint8`. The output image `BW2` is of class `uint8`.

Example

```
BW1 = imread('circbw.tif');
BW2 = bwperim(BW1,8);
imshow(BW1)
figure, imshow(BW2)
```





See Also `bwarea`, `bweuler`, `bwfill`

Purpose Select objects in a binary image

Syntax

```
BW2 = bwselect(BW1,c,r,n)
BW2 = bwselect(BW1,n)
[BW2,idx] = bwselect(...)

BW2 = bwselect(x,y,BW1,xi,yi,n)
[x,y,BW2,idx,xi,yi] = bwselect(...)
```

Description

`BW2 = bwselect(BW1,c,r,n)` 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).

`BW2 = bwselect(BW1,n)` displays the image `BW1` on the screen and lets you select the `(r,c)` coordinates using the mouse. If you omit `BW1`, `bwselect` 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.

`[BW2,idx] = bwselect(...)` returns the linear indices of the pixels belonging to the selected objects.

`BW2 = bwselect(x,y,BW1,xi,yi,n)` 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.

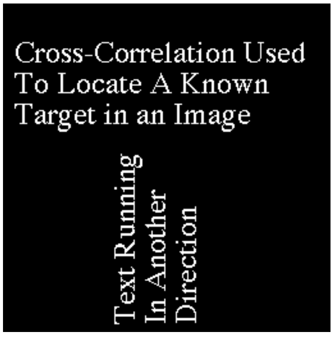
`[x,y,BW2,idx,xi,yi] = bwselect(...)` returns the `XData` and `YData` in `x` and `y`; the output image in `BW2`; linear indices of all the pixels belonging to the selected objects in `idx`; and the specified spatial coordinates in `xi` and `yi`.

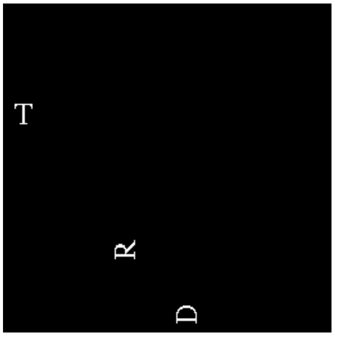
If `bwselect` is called with no output arguments, the resulting image is displayed in a new figure.

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`

Purpose Rearrange the colors in a colormap

Syntax

```
[Y,newmap] = cmpermute(X,map)
[Y,newmap] = cmpermute(X,map,index)
```

Description `[Y,newmap] = cmpermute(X,map)` randomly reorders the colors in `map` to produce a new colormap `newmap`. `cmpermute` also modifies the values in `X` to maintain correspondence between the indices and the colormap, and returns the result in `Y`. The image `Y` and associated colormap `newmap` produce the same image as `X` and `map`.

`[Y,newmap] = cmpermute(X,map,index)` uses an ordering matrix (such as the second output of `sort`) to define the order of colors in the new colormap.

Class Support The input image `X` can be of class `uint8` or `double`. `Y` is returned as an array of the same class as `X`.

Example To order a colormap by luminance, use

```
ntsc = rgb2ntsc(map);
[dum,index] = sort(ntsc(:,1));
[Y,newmap] = cmpermute(X,map,index);
```

See Also `randperm`, `sort` in the MATLAB Function Reference

cmunique

Purpose Find unique colormap colors and the corresponding image

Syntax

```
[Y,newmap] = cmunique(X,map)
[Y,newmap] = cmunique(RGB)
[Y,newmap] = cmunique(I)
```

Description `[Y,newmap] = cmunique(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. `cmunique` removes duplicate rows from the colormap and adjusts the indices in the image matrix accordingly.

`[Y,newmap] = cmunique(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`.)

`[Y,newmap] = cmunique(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`.

Class Support The input image can be of class `uint8`, `uint16`, or `double`. The class of the output image `Y` is `uint8` 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 `double`.

See Also `gray2ind`, `rgb2ind`

Purpose Rearrange matrix columns into blocks

Syntax

```
A = col2im(B,[m n],[mm nn],block_type)
A = col2im(B,[m n],[mm nn])
```

Description `col2im` rearranges matrix columns into blocks. *block_type* is a string with one of these values:

- `'distinct'` for `m`-by-`n` distinct blocks
- `'sliding'` for `m`-by-`n` sliding blocks (default)

`A = col2im(B,[m n],[mm nn],'distinct')` rearranges each column of `B` into a distinct `m`-by-`n` block to create the matrix `A` of size `mm`-by-`nn`. If `B = [A11(:) A12(:) A21(:) A22(:)]`, where each column has length `m*n`, then `A = [A11 A12;A21 A22]` where each `Aij` is `m`-by-`n`.

`A = col2im(B,[m n],[mm nn],'sliding')` 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 `im2col(...,'sliding')` using a column compression function (such as `sum`).

`A = col2im(B,[m n],[mm nn])` uses the default `block_type` of `'sliding'`.

Class Support `B` can be of class `double` or of any integer class. `A` is of the same class as `B`.

See Also `blkproc`, `colfilt`, `im2col`, `nlfilter`

colfilt

Purpose

Perform neighborhood operations using columnwise functions

Syntax

```
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',...)
```

Description

`colfilt` processes distinct or sliding blocks as columns. `colfilt` can perform similar operations to `blkproc` and `nlfilter`, but often executes much faster.

`B = colfilt(A,[m n],`*`block_type`*`,fun)` 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 a `function_handle`, created using `@`, or an inline object. `colfilt` zero pads `A`, if necessary.

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`.

block_type is a string with one of these values:

- `'distinct'` for `m`-by-`n` distinct blocks
- `'sliding'` for `m`-by-`n` sliding neighborhoods

`B = colfilt(A,[m n],'distinct',fun)` 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.

`B = colfilt(A,[m n],'sliding',fun)` 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`.

`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 `uint16`, 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.

```
I = imread('tire.tif')
imshow(I)
I2 = uint8(colfilt(I,[5 5],'sliding',@mean));
figure, imshow(I2)
```

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

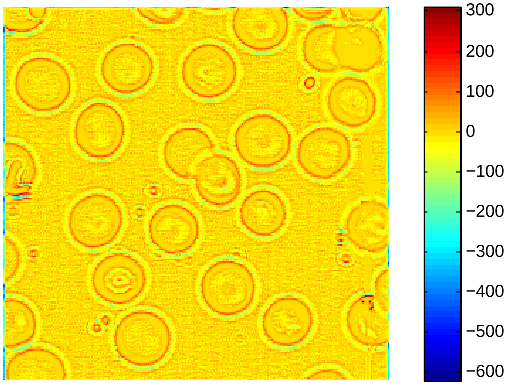
`colorbar` is a function in MATLAB.

Example

Display a colorbar to view values for a filtered image.

```
I = imread('blood1.tif');
h = fspecial('log');
I2 = filter2(h,I);
```

```
imshow(I2,[],'notruesize'), colormap(jet(64)), colorbar
```



See Also `imagesc`

Purpose Perform two-dimensional convolution

Syntax

```
C = conv2(A,B)
C = conv2(hcol,hrow,A)
C = conv2(...,shape)
```

Description `C = 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 `[ma,mb]` and the size of `B` is `[mb,nb]`, then the size of `C` is `[ma+mb-1,na+nb-1]`.

`C = conv2(hcol,hrow,A)` convolves `A` separably with `hcol` in the column direction and `hrow` in the row direction. `hcol` and `hrow` are both vectors.

`C = conv2(...,`*`shape`*`)` returns a subsection of the two-dimensional convolution, as specified by the *shape* parameter. *shape* is a string with one of these values:

- `'full'` (the default) returns the full two-dimensional 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 the zero-padded edges. Using this option, `size(C) = [ma-mb+1, na-nb+1]` when `size(A)` > `size(B)`.

For image filtering, `A` should be the image matrix and `B` should be the filter (convolution kernel) if the `shape` parameter is `'same'` or `'valid'`. If the `shape` parameter is `'full'`, the order does not matter, because full convolution is commutative.

Class Support All vector and matrix inputs to `conv2` can be of class `double` or of any integer class. The output matrix `C` is of class `double`.

Remarks `conv2` is a function in MATLAB.

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

`xcorr`, `xcorr2` in the *Signal Processing Toolbox User's Guide*

`conv`, `deconv` in the MATLAB Function Reference

convmtx2

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)`.

Class Support

The inputs are all of class `double`. The output matrix `T` is of class `sparse`. 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*

Purpose Perform N-dimensional convolution

Syntax

```
C = convn(A,B)
C = convn(A,B,shape)
```

Description `C = convn(A,B)` computes the N-dimensional convolution of matrices `A` and `B`.

`C = convn(A,B,`*shape*`)` returns a subsection of the N-dimensional convolution, as specified by the *shape* parameter. *shape* is a string with one of these values:

- `'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 `double` or of any integer class. The output matrix `C` is of class `double`.

Remarks `convn` is a function in MATLAB.

See Also `conv2`

corr2

Purpose Compute the two-dimensional correlation coefficient between two matrices

Syntax

```
r = corr2(A,B)
```

Description `r = corr2(A,B)` 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 `double` or of any integer class. `r` is a scalar of class `double`.

Algorithm `corr2` 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 $\bar{A}$ = `mean2(A)`, and $\bar{B}$ = `mean2(B)`.

See Also `std2`

`corrcoef` in the MATLAB Function Reference

Purpose

Compute two-dimensional discrete cosine transform

Syntax

```
B = dct2(A)
B = dct2(A,m,n)
B = dct2(A,[m n])
```

Description

`B = 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 = dct2(A,m,n)` or `B = 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 `double` or of any integer class. The returned matrix `B` is of class `double`.

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

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

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

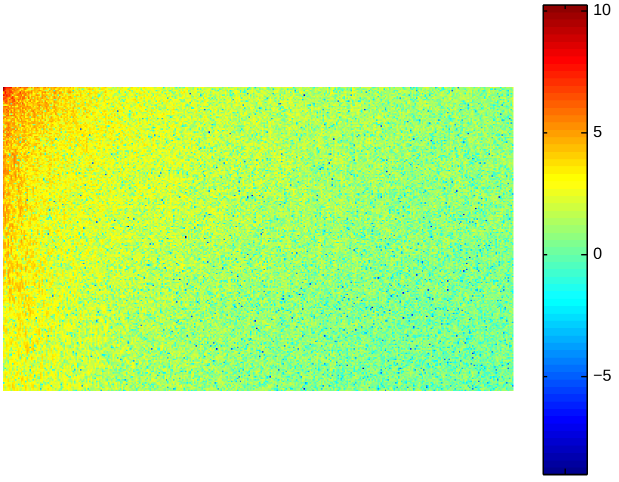
where M and N are the row and column size of `A`, respectively. If you apply the DCT to real data, the result is also real. The DCT tends to concentrate information, making it useful for image compression applications.

This transform can be inverted using `idct2`.

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 `idct2`.

```
J(abs(J) < 10) = 0;
K = idct2(J)/255;
imshow(K)
```

See Also `fft2`, `idct2`, `ifft2`

References

[1] Jain, Anil K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. pp. 150-153.

[2] 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

```
D = dctmtx(n)
```

Description `D = dctmtx(n)` returns the `n`-by-`n` DCT (discrete cosine 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 `dct2`, 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 `dctmtx` 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 `dct2` for each individual block.

See Also `dct2`

Purpose Perform dilation on a binary image

Syntax

```
BW2 = dilate(BW1,SE)
BW2 = dilate(BW1,SE,alg)
BW2 = dilate(BW1,SE,...,n)
```

Description `BW2 = 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 = dilate(BW1,SE,`*`alg`*`)` performs dilation using the specified algorithm. *alg* is a string that can have one of these values:

- `'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 trade-offs 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 = dilate(BW1,SE,...,n)` performs the dilation operation `n` times.

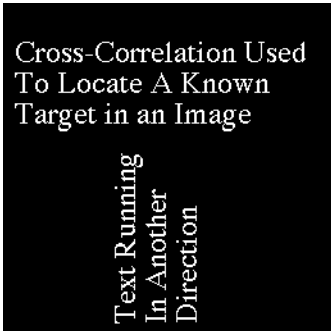
Class Support The input image `BW1` can be of class `double` or `uint8`. The output image `BW2` is of class `uint8`.

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 *slower* 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(6,2);
BW2 = dilate(BW1,SE);
imshow(BW1)
```

```
figure, imshow(BW2)
```



See Also `bwmorph`, `erode`

References

[1] Gonzalez, Rafael C., and Richard E. Woods. *Digital Image Processing.* Addison-Wesley, 1992. p. 518.

[2] Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I.* Addison-Wesley, 1992. p. 158.

Purpose Convert an image, increasing apparent color resolution by dithering

Syntax

```
X = dither(RGB,map)
BW = dither(I)
```

Description `X = dither(RGB,map)` creates an indexed image approximation of the RGB image in the array `RGB` by dithering the colors in colormap `map`.

`X = dither(RGB,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.

`BW = dither(I)` converts the intensity image in the matrix `I` to the binary (black and white) image `BW` by dithering.

Class Support The input image (`RGB` or `I`) can be of class `uint8`, `uint16`, or `double`. All other input arguments must be of class `double`. The output image (`X` or `BW`) is of class `uint8` if it is a binary image or if it is an indexed image with 256 or fewer colors; otherwise its class is `double`.

Algorithm `dither` increases the apparent color resolution of an image by applying Floyd-Steinberg's error diffusion dither algorithm.

References

[1] Floyd, R. W. and L. Steinberg. "An Adaptive Algorithm for Spatial Gray Scale," *International Symposium Digest of Technical Papers*. Society for Information Displays, 1975. p. 36.

[2] Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 469-476.

See Also `rgb2ind`

double

Purpose

Convert data to double precision

Syntax

```
B = double(A)
```

Description

`B = double(A)` creates a double-precision array `B` from the array `A`. If `A` is a `double` array, `B` is identical to `A`.

`double` is useful if you have a `uint8` image array that you want to perform arithmetic operations on, because MATLAB does not support these operations on `uint8` data.

Remarks

`double` is a MATLAB built-in function.

Example

```
A = imread('saturn.tif');
B = sqrt(double(A));
```

See Also

`im2double`, `im2uint8`, `im2uint16`, `uint8`

Purpose

Find edges in an intensity image

Syntax

```
BW = edge(I,'sobel')
BW = edge(I,'sobel',thresh)
BW = edge(I,'sobel',thresh,direction)
[BW,thresh] = edge(I,'sobel',...)

BW = edge(I,'prewitt')
BW = edge(I,'prewitt',thresh)
BW = edge(I,'prewitt',thresh,direction)
[BW,thresh] = edge(I,'prewitt',...)

BW = edge(I,'roberts')
BW = edge(I,'roberts',thresh)
[BW,thresh] = edge(I,'roberts',...)

BW = edge(I,'log')
BW = edge(I,'log',thresh)
BW = edge(I,'log',thresh,sigma)
[BW,threshold] = edge(I,'log',...)

BW = edge(I,'zerocross',thresh,h)
[BW,thresh] = edge(I,'zerocross',...)

BW = edge(I,'canny')
BW = edge(I,'canny',thresh)
BW = edge(I,'canny',thresh,sigma)
[BW,threshold] = edge(I,'canny',...)
```

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 direction of detection 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 direction of detection 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`-by-`n`, where `n` = `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*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`, `uint16`, or `double`. `BW` is of class `uint8`.

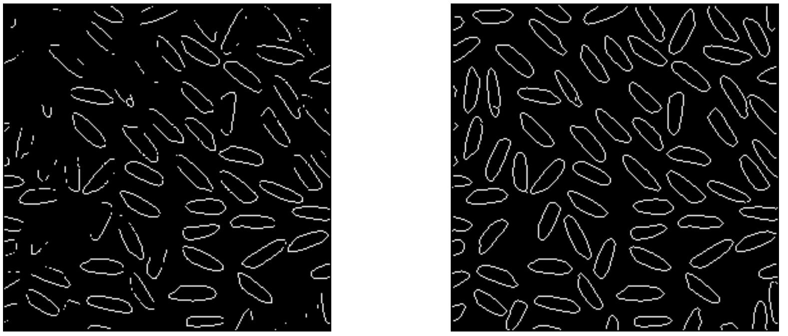
Remarks

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

[3] 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.

[4] Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 478-488.

[5] Parker, James R. *Algorithms for Image Processing and Computer Vision.* New York: John Wiley & Sons, Inc., 1997. pp. 23-29.

erode

Purpose

Perform erosion on a binary image

Syntax

```
BW2 = erode(BW1,SE)
BW2 = erode(BW1,SE,alg)
BW2 = erode(BW1,SE,...,n)
```

Description

`BW2 = erode(BW1,SE)` performs erosion on the binary image `BW1`, using the binary structuring element `SE`. `SE` is a matrix containing only 1's and 0's.

`BW2 = erode(BW1,SE,`*alg*`)` performs erosion using the specified algorithm. *alg* is a string that can have one of these values:

- `'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 = erode(BW1,SE,...,n)` performs the erosion operation `n` times.

Class Support

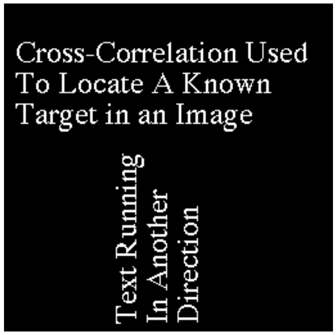
The input image `BW1` can be of class `double` or `uint8`. The output image `BW2` is of class `uint8`.

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 *slower* 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);
BW2 = erode(BW1,SE);
imshow(BW1)
figure, imshow(BW2)
```



See Also `bwmorph`, `dilate`

References

[1] Gonzalez, Rafael C., and Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992. p. 518.

[2] Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I*. Addison-Wesley, 1992. p. 158.

Purpose Compute two-dimensional fast Fourier transform

Syntax

```
B = fft2(A)
B = fft2(A,m,n)
```

Description `B = fft2(A)` performs a two-dimensional fast Fourier transform (FFT), returning the result in `B`. `B` is the same size as `A`; if `A` is a vector, `B` has the same orientation as `A`.

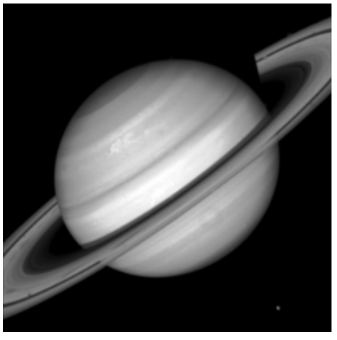
`B = fft2(A,m,n)` truncates or zero pads `A`, if necessary, to create an `m`-by-`n` matrix before performing the FFT. The result `B` is also `m`-by-`n`.

Class Support The input matrix `A` can be of class `double` or of any integer class. The output matrix `B` is of class `double`.

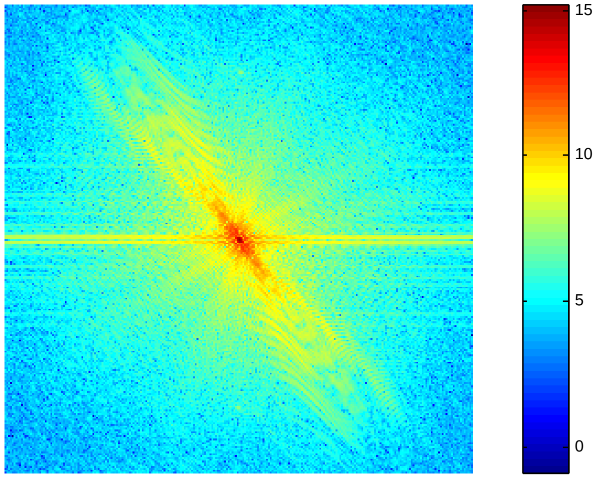
Remarks `fft2` is a function in MATLAB.

Example

```
load imdemos saturn2
imshow(saturn2)
```



```
B = fftshift(fft2(saturn2));
imshow(log(abs(B)),[],'notruesize'), colormap(jet(64)), colorbar
```



Algorithm

`fft2(A)` is simply

```
fft(fft(A).').'
```

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 MATLAB Function Reference

fftn

Purpose Compute N-dimensional fast Fourier transform

Syntax

```
B = fftn(A)
B = fftn(A,siz)
```

Description `B = fftn(A)` performs the N-dimensional fast Fourier transform. The result `B` is the same size as `A`.

`B = fftn(A,siz)` pads `A` with zeros (or truncates `A`) to create an N-dimensional array of size `siz` before doing the transform. The size of the result is `siz`.

Class Support The input matrix `A` can be of class `double` or of any integer class. The output matrix `B` is of class `double`.

Remarks `fftn` is a function in MATLAB.

Algorithm `fftn(A)` is equivalent to:

```
B = A;
for p = 1:length(size(A))
    B = fft(B,[],p);
end
```

This code computes the one-dimensional fast Fourier transform along each dimension of `A`. The time required to compute `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 `fft2`, `ifftn`

`fft` in the MATLAB Function Reference

Purpose Shift zero-frequency component of fast Fourier transform to center of spectrum

Syntax

```
B = fftshift(A)
```

Description `B = fftshift(A)` rearranges the outputs of `fft`, `fft2`, and `fftn` by moving the zero frequency component to the center of the array.

For vectors, `fftshift(A)` swaps the left and right halves of `A`. For matrices, `fftshift(A)` swaps quadrants one and three of `A` with quadrants two and four. For higher-dimensional arrays, `fftshift(A)` swaps "half-spaces" of `A` along each dimension.

Class Support The input matrix `A` can be of class `double` or of any integer class. The output matrix `B` is of the same class as `A`.

Remarks `fftshift` is a function in MATLAB.

Example

```
B = fftn(A);
C = fftshift(B);
```

See Also `fft2`, `fftn`, `ifftshift`

`fft` in the MATLAB Function Reference

Purpose

Perform two-dimensional linear filtering

Syntax

```
B = filter2(h,A)
B = filter2(h,A,shape)
```

Description

`B = 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 = filter2(h,A,`*shape*`)` returns the part of `B` specified by the *shape* parameter. *shape* is a string with one of these values:

- `'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 `filter2` can be of class `double` or of any integer class. The output matrix `B` is of class `double`.

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 `filter2` performs linear filtering.

`filter2` 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, roifilt2

freqspace

Purpose Determine frequency spacing for two-dimensional frequency response

Syntax

```
[f1,f2] = freqspace(n)
[f1,f2] = freqspace([m n])
[x1,y1] = freqspace(...,'meshgrid')
f = freqspace(N)
f = freqspace(N,'whole')
```

Description

`freqspace` returns the implied frequency range for equally spaced frequency responses. `freqspace` is useful when creating desired frequency responses for `fsamp2`, `fwind1`, and `fwind2`, as well as for various one-dimensional applications.

`[f1,f2] = freqspace(n)` returns the two-dimensional frequency vectors `f1` and `f2` for an `n`-by-`n` matrix.

For `n` odd, both `f1` and `f2` are `[-n+1:2:n-1]/n`.

For `n` even, both `f1` and `f2` are `[-n:2:n-2]/n`.

`[f1,f2] = freqspace([m n])` returns the two-dimensional frequency vectors `f1` and `f2` for an `m`-by-`n` matrix.

`[x1,y1] = freqspace(...,'meshgrid')` is equivalent to

```
[f1,f2] = freqspace(...);
[x1,y1] = meshgrid(f1,f2);
```

`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.

`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`.

Remarks `freqspace` is a function in MATLAB.

See Also `fsamp2`, `fwind1`, `fwind2`

`meshgrid` in the MATLAB Function Reference

Purpose

Compute two-dimensional frequency response

Syntax

```
[H,f1,f2] = freqz2(h,n1,n2)
[H,f1,f2] = freqz2(h,[n2 n1])
[H,f1,f2] = freqz2(h,f1,f2)
[H,f1,f2] = freqz2(h)
[...] = freqz2(h,...,[dx dy])
[...] = freqz2(h,...,dx)
freqz2(...)
```

Description

`[H,f1,f2] = 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.

`[H,f1,f2] = freqz2(h,[n2 n1])` returns the same result returned by `[H,f1,f2] = freqz2(h,n1,n2)`.

`[H,f1,f2] = freqz2(h)` uses `[n2 n1] = [64 64]`.

`[H,f1,f2] = 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.

`[...] = 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.

`[...] = freqz2(h,...,dx)` uses `dx` to determine the intersample spacing in both dimensions.

With no output arguments, `freqz2(...)` produces a mesh plot of the two-dimensional magnitude frequency response.

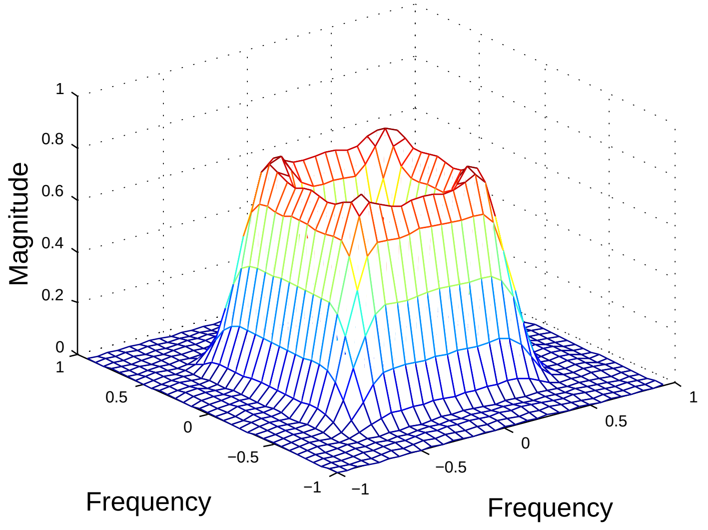
Class Support

The input matrix `h` can be of class `double` or of any integer class. All other inputs to `freqz2` must be of class `double`. All outputs are of class `double`.

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 = fwind1(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 User's Guide*

Purpose

Design two-dimensional FIR filter using frequency sampling

Syntax

```
h = fsamp2(Hd)
h = fsamp2(f1,f2,Hd,[m n])
```

Description

fsamp2 designs two-dimensional FIR filters based on a desired two-dimensional frequency response sampled at points on the Cartesian plane.

h = fsamp2(Hd) designs a two-dimensional FIR filter with frequency response Hd, and returns the filter coefficients in matrix h. (fsamp2 returns h as a computational molecule, which is the appropriate form to use with filter2.) The filter h has a frequency response that passes through points in Hd. If Hd is m-by-n, then h is also m-by-n.

Hd is a matrix containing the desired frequency response sampled at equally spaced points between -1.0 and 1.0 along the x and y frequency axes, where 1.0 corresponds to half the sampling frequency, or π radians.

$$H_d(f_1, f_2) = H_d(\omega_1, \omega_2)\big|_{\omega_1 = \pi f_1,\, \omega_2 = \pi f_2}$$

For accurate results, use frequency points returned by freqspace to create Hd. (See the entry for freqspace for more information.)

h = fsamp2(f1,f2,Hd,[m n]) produces an m-by-n FIR filter by matching the filter response at the points in the vectors f1 and f2. The frequency vectors f1 and f2 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 m*n desired frequency points. fsamp2 issues a warning if you specify fewer than m*n points.

Class Support

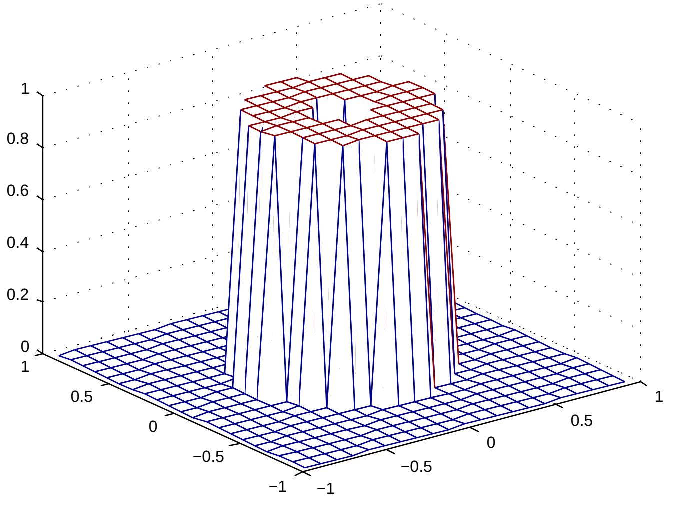
The input matrix Hd can be of class double or of any integer class. All other inputs to fsamp2 must be of class double. All outputs are of class double.

Example

Use fsamp2 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):

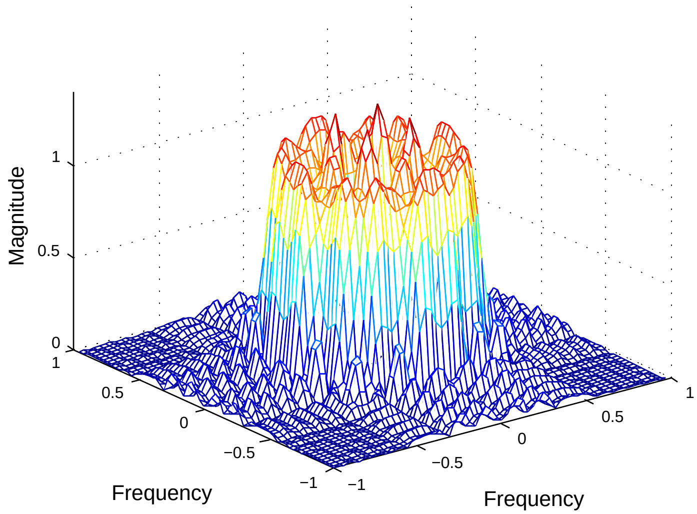
1. Create a matrix Hd that contains the desired bandpass response. Use freqspace to create the frequency range vectors f1 and f2.

```
[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`, `fwind1`, `fwind2`

Reference

[1] Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 213-217.

Purpose Create predefined filters

Syntax

```
h = fspecial(type)
h = fspecial(type,parameters)
```

Description

h = fspecial(*type*) creates a two-dimensional filter h of the specified *type*. (fspecial returns h as a computational molecule, which is the appropriate form to use with filter2.) *type* is a string having one of these values:

- 'gaussian' for a Gaussian lowpass filter
- 'sobel' for a Sobel horizontal edge-emphasizing filter
- 'prewitt' for a Prewitt horizontal edge-emphasizing filter
- 'laplacian' for a filter approximating the two-dimensional Laplacian operator
- 'log' for a Laplacian of Gaussian filter
- 'average' for an averaging filter
- 'unsharp' for an unsharp contrast enhancement filter

h = fspecial(*type*,parameters) accepts a filter *type* plus additional modifying parameters particular to the type of filter chosen. If you omit these arguments, fspecial uses default values for the parameters.

The following list shows the syntax for each filter type. Where applicable, additional parameters are also shown.

- h = fspecial('gaussian',n,sigma) returns a rotationally symmetric Gaussian lowpass filter with standard deviation sigma (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 [3 3] for n and 0.5 for sigma.
- h = fspecial('sobel') returns this 3-by-3 horizontal edge-finding and *y*-derivative approximation filter:

  [1 2 1
  0 0 0
  -1 -2 -1]

  To find vertical edges, or for *x*-derivatives, use h'.

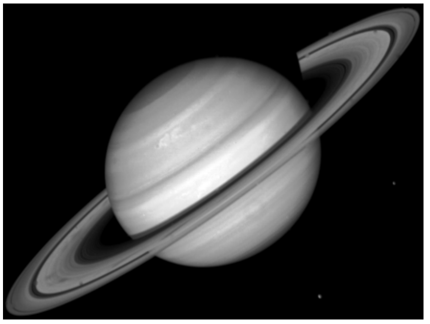
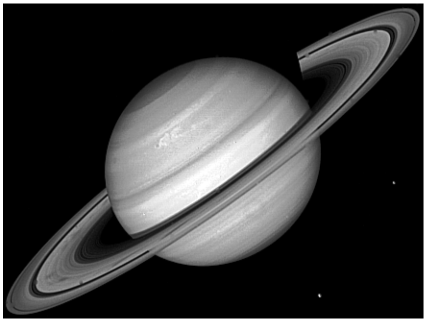
- `h = fspecial('prewitt')` returns this 3-by-3 horizontal edge-finding and *y*-derivative approximation filter:

  [1 1 1
  0 0 0
  -1 -1 -1]

  To find vertical edges, or for *x*-derivatives, use `h'`.
- `h = fspecial('laplacian',alpha)` returns a 3-by-3 filter approximating the two-dimensional Laplacian operator. The parameter `alpha` 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 `alpha`.
- `h = fspecial('log',n,sigma)` returns a rotationally symmetric Laplacian of Gaussian filter with standard deviation `sigma` (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 `sigma`.
- `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',alpha)` returns a 3-by-3 unsharp contrast enhancement filter. `fspecial` creates the unsharp filter from the negative of the Laplacian filter with parameter `alpha`. `alpha` 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 `alpha`.

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 MATLAB Function Reference

ftrans2

Purpose

Design two-dimensional FIR filter using frequency transformation

Syntax

```
h = ftrans2(b,t)
h = ftrans2(b)
```

Description

`h = 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)`-by-`((n-1)*(Q-1)/2+1)`.

`h = 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`,

$$H(\omega_1, \omega_2) = B(\omega)\big|_{\cos\omega = T(\omega_1, \omega_2)}$$

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 $H(\omega_1,\omega_2)$.

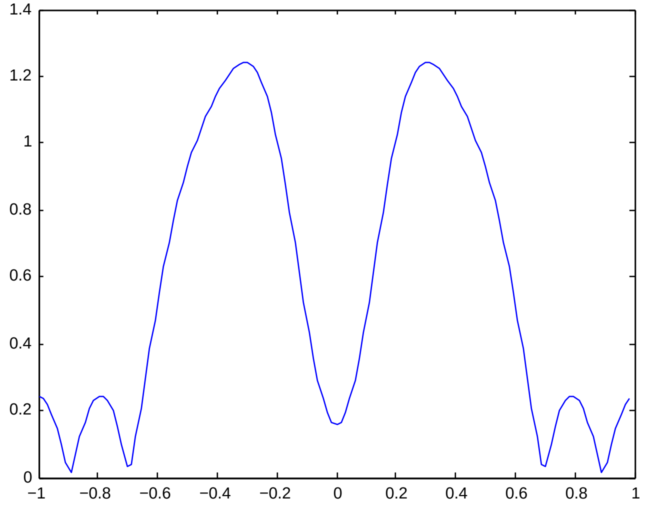
$$h(n_1, n_2) = \frac{1}{(2\pi)^2}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} H(\omega_1, \omega_2)e^{j\omega_1 n_1}e^{j\omega_2 n_2}d\omega_1 d\omega_2$$

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):

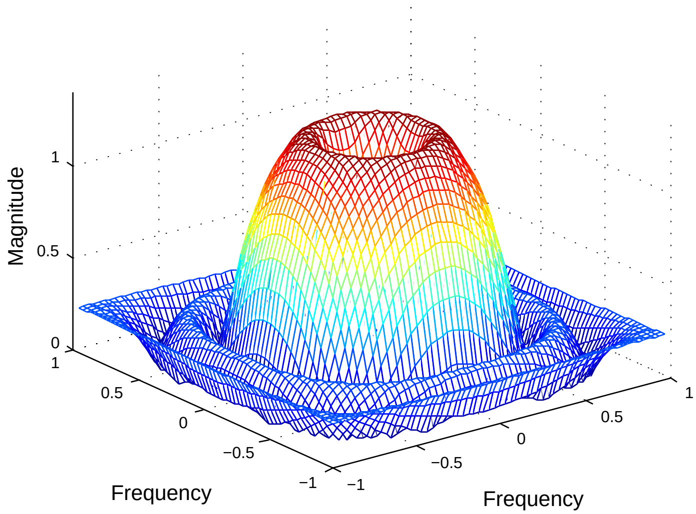
1. Since `ftrans2` transforms a one-dimensional FIR filter to create a two-dimensional filter, first design a one-dimensional FIR bandpass filter using the Signal Processing Toolbox function `remez`.

```
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);
freqz2(h)
```



See Also `conv2`, `filter2`, `fsamp2`, `fwind1`, `fwind2`

Reference

[1] 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 = fwind1(Hd,win)
h = fwind1(Hd,win1,win2)
h = fwind1(f1,f2,Hd,...)
```

Description

`fwind1` designs two-dimensional FIR filters using the window method. `fwind1` uses a one-dimensional window specification to design a two-dimensional FIR filter based on the desired frequency response `Hd`. `fwind1` works with one-dimensional windows only; use `fwind2` to work with two-dimensional windows.

`h = fwind1(Hd,win)` designs a two-dimensional FIR filter `h` with frequency response `Hd`. (`fwind1` returns `h` as a computational molecule, which is the appropriate form to use with `filter2`.) `fwind1` uses the one-dimensional window `win` to form an approximately circularly symmetric two-dimensional window using Huang's method. You can specify `win` using windows from the Signal Processing Toolbox, such as `boxcar`, `hamming`, `hanning`, `bartlett`, `blackman`, `kaiser`, or `chebwin`. If `length(win)` is `n`, then `h` is `n`-by-`n`.

`Hd` is a matrix containing the desired frequency response sampled at equally spaced points between -1.0 and 1.0 (in normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians) along the *x* and *y* frequency axes. For accurate results, use frequency points returned by `freqspace` to create `Hd`. (See the entry for `freqspace` for more information.)

`h = fwind1(Hd,win1,win2)` uses the two one-dimensional windows `win1` and `win2` to create a separable two-dimensional window. If `length(win1)` is `n` and `length(win2)` is `m`, then `h` is `m`-by-`n`.

`h = fwind1(f1,f2,Hd,...)` lets you specify the desired frequency response `Hd` at arbitrary frequencies (`f1` and `f2`) along the *x* and *y* axes. The frequency vectors `f1` and `f2` should be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or π radians. The length of the window(s) controls the size of the resulting filter, as above.

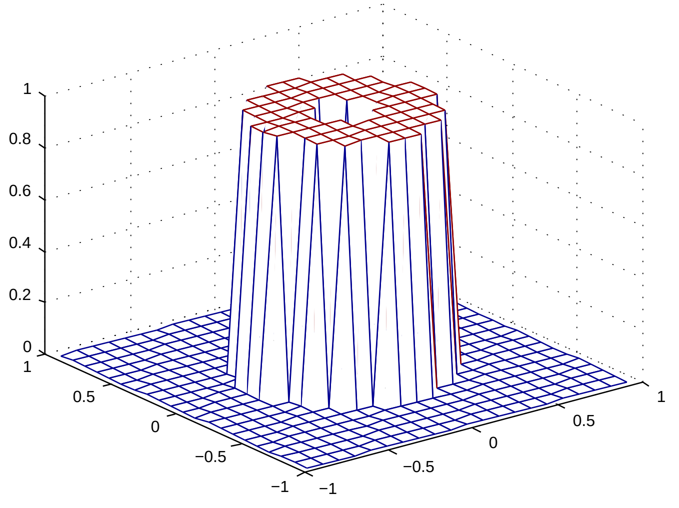
Class Support The input matrix `Hd` can be of class `double` or of any integer class. All other inputs to `fwind1` must be of class `double`. All outputs are of class `double`.

Example

Use `fwind1` 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):

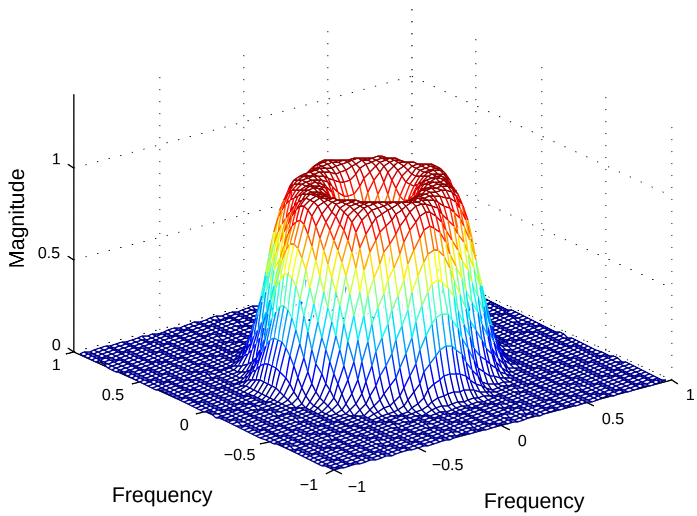
1 Create a matrix `Hd` that contains the desired bandpass response. Use `freqspace` to create the frequency range vectors `f1` and `f2`.

```
[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.

$$h_d(n_1, n_2) = \frac{1}{(2\pi)^2}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} H_d(\omega_1, \omega_2)e^{j\omega_1 n_1}e^{j\omega_2 n_2}d\omega_1 d\omega_2$$

$$h(n_1, n_2) = h_d(n_1, n_2)w(n_1, n_2)$$

See Also `conv2`, `filter2`, `fsamp2`, `freqspace`, `ftrans2`, `fwind2`

Reference

[1] 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 = fwind2(Hd,win)
h = fwind2(f1,f2,Hd,win)
```

Description

Use `fwind2` to design two-dimensional FIR filters using the window method. `fwind2` uses a two-dimensional window specification to design a two-dimensional FIR filter based on the desired frequency response `Hd`. `fwind2` works with two-dimensional windows; use `fwind1` to work with one-dimensional windows.

`h = fwind2(Hd,win)` produces the two-dimensional FIR filter `h` using an inverse Fourier transform of the desired frequency response `Hd` and multiplication by the window `win`. `Hd` is a matrix containing the desired frequency response at equally spaced points in the Cartesian plane. `fwind2` returns `h` as a computational molecule, which is the appropriate form to use with `filter2`. `h` is the same size as `win`.

For accurate results, use frequency points returned by `freqspace` to create `Hd`. (See the entry for `freqspace` for more information.

`h = fwind2(f1,f2,Hd,win)` lets you specify the desired frequency response `Hd` at arbitrary frequencies (`f1` and `f2`) along the x and y axes. The frequency vectors `f1` and `f2` should be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or π radians. `h` is the same size as `win`.

Class Support

The input matrix `Hd` can be of class `double` or of any integer class. All other inputs to `fwind2` must be of class `double`. All outputs are of class `double`.

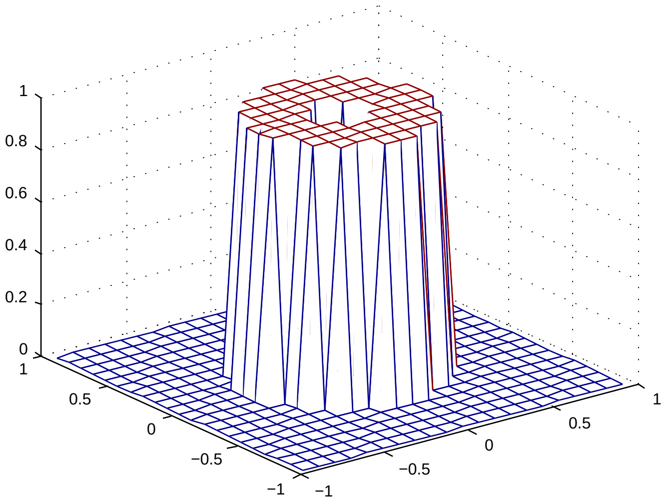
Example

Use `fwind2` 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):

1 Create a matrix `Hd` that contains the desired bandpass response. Use `freqspace` to create the frequency range vectors `f1` and `f2`.

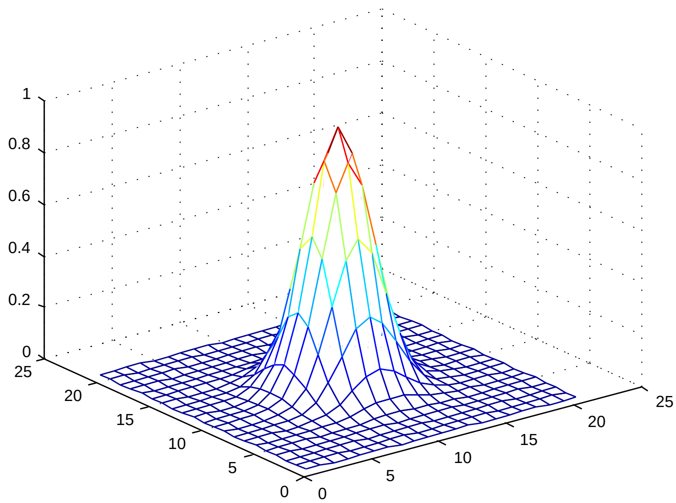
```
[f1,f2] = freqspace(21,'meshgrid');
Hd = ones(21);
r = sqrt(f1.^2 + f2.^2);
Hd((r<0.1)|(r>0.5)) = 0;
```

```
colormap(jet(64))
mesh(f1,f2,Hd)
```



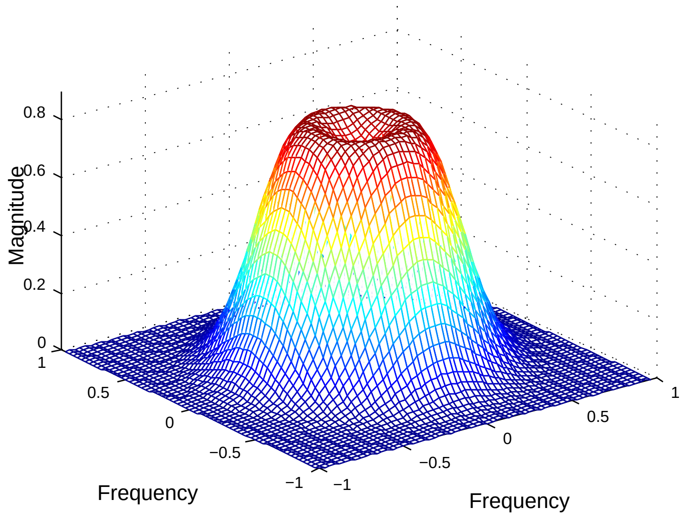
2 Create a two-dimensional Gaussian window using `fspecial`.

```
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);
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

[1] 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; `A` is returned as an empty matrix
`1`	Intensity image with values in standard range ([0,1] for `double` arrays, [0,255] for `uint8` arrays, [0,65535] for `uint16` 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`.

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,map] = gray2ind(I,n)
```

Description `gray2ind` scales, then rounds, an intensity image to produce an equivalent indexed image.

`[X,map] = gray2ind(I,n)` converts the intensity image `I` to an indexed image `X` with colormap `gray(n)`. If `n` is omitted, it defaults to 64.

Class Support The input image `I` can be of class `uint8`, `uint16`, or `double`. The class of the output image `X` is `uint8` if the colormap length is less than or equal to 256. If the colormap length is greater than 256, `X` is of class `double`.

See Also `ind2gray`

grayslice

Purpose Create indexed image from intensity image, using multilevel thresholding

Syntax

```
X = grayslice(I,n)
X = grayslice(I,v)
```

Description `X = grayslice(I,n)` thresholds the intensity image `I` using cutoff values $\frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}$, returning an indexed image in `X`.

`X = 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`.

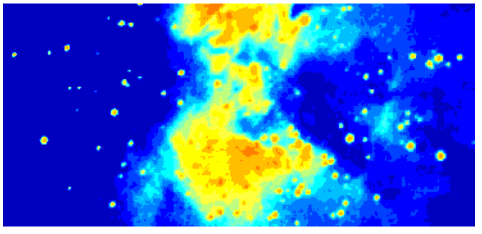
You can view the thresholded image using `imshow(X,map)` with a colormap of appropriate length.

Class Support The input image `I` can be of class `uint8`, `uint16`, or `double`. Note that the threshold values are always between 0 and 1, even if `I` is of class `uint8` or `uint16`. In this case, each threshold value is multiplied by 255 or 65535 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 = histeq(I,hgram)
J = histeq(I,n)
[J,T] = histeq(I,...)

newmap = histeq(X,map,hgram)
newmap = histeq(X,map)
[newmap,T] = histeq(X,...)
```

Description `histeq` 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.

`J = histeq(I,hgram)` transforms the intensity image `I` so that the histogram of the output intensity image `J` with `length(hgram)` bins approximately matches `hgram`. The vector `hgram` should contain integer counts for equally spaced bins with intensity values in the appropriate range: [0, 1] for images of class `double`, [0, 255] for images of class `uint8`, and [0, 65535] for images of class `uint16`. `histeq` automatically scales `hgram` so that `sum(hgram) = prod(size(I))`. The histogram of `J` will better match `hgram` when `length(hgram)` is much smaller than the number of discrete levels in `I`.

`J = histeq(I,n)` transforms the intensity image `I`, returning in `J` an intensity image with `n` discrete gray levels. A roughly equal number of pixels is mapped to each of the `n` levels in `J`, so that the histogram of `J` is approximately flat. (The histogram of `J` is flatter when `n` is much smaller than the number of discrete levels in `I`.) The default value for `n` is 64.

`[J,T] = histeq(I,...)` returns the gray scale transformation that maps gray levels in the intensity image `I` to gray levels in `J`.

`newmap = histeq(X,map,hgram)` transforms the colormap associated with the indexed image `X` so that the histogram of the gray component of the indexed image `(X,newmap)` approximately matches `hgram`. `histeq` returns the transformed colormap in `newmap`. `length(hgram)` must be the same as `size(map,1)`.

`newmap = histeq(X,map)` transforms the values in the colormap so that the histogram of the gray component of the indexed image `X` is approximately flat. It returns the transformed colormap in `newmap`.

`[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`, `uint16`, 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`.

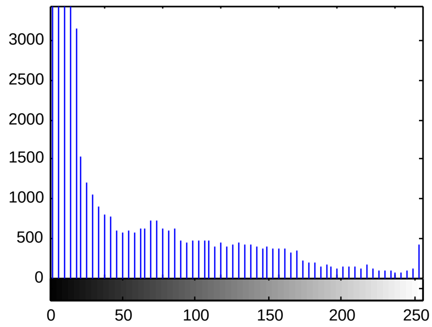
Example

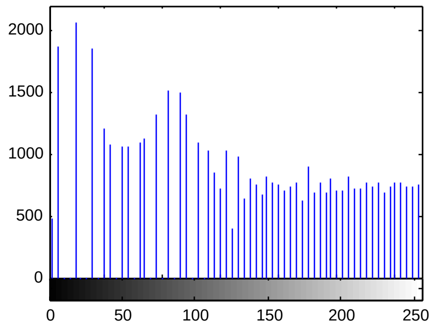
Enhance the contrast of an intensity image using histogram equalization.

```
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

$$\left| c_1(T(k)) - c_0(k) \right|$$

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

```
rgbmap = hsv2rgb(hsvmap)
RGB = hsv2rgb(HSV)
```

Description `rgbmap = hsv2rgb(hsvmap)` converts the HSV values in `hsvmap` to RGB color space. `hsvmap` is an `m`-by-3 matrix that contains hue, saturation, and value components as its three columns, and `rgbmap` is returned as an `m`-by-3 matrix that represents the same set of colors as red, green, and blue values. Both `rgbmap` and `hsvmap` contain values in the range 0 to 1.0.

`RGB = hsv2rgb(HSV)` 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.

Class Support The input array to `hsv2rgb` must be of class `double`. The output array is of class `double`.

Remarks `hsv2rgb` is a function in MATLAB.

See Also `rgb2hsv`, `rgbplot`

`colormap` in the MATLAB Function Reference

Purpose Compute two-dimensional inverse discrete cosine transform

Syntax

```
B = idct2(A)
B = idct2(A,m,n)
B = idct2(A,[m n])
```

Description `B = idct2(A)` returns the two-dimensional inverse discrete cosine transform (DCT) of `A`.

`B = idct2(A,m,n)` or `B = idct2(A,[m n])` pads `A` with zeros to size `m`-by-`n` before transforming. If `[m n]` < `size(A)`, `idct2` crops `A` before transforming.

For any `A`, `idct2(dct2(A))` equals `A` to within roundoff error.

Class Support The input matrix `A` can be of class `double` or of any integer class. The output matrix `B` is of class `double`.

Algorithm `idct2` computes the two-dimensional inverse DCT using

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

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

See Also `dct2`, `dctmtx`, `fft2`, `ifft2`

References

[1] Jain, Anil K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. pp. 150-153.

[2] Pennebaker, William B., and Joan L. Mitchell. *JPEG: Still Image Data Compression Standard*. New York: Van Nostrand Reinhold, 1993.

ifft2

Purpose

Compute two-dimensional inverse fast Fourier transform

Syntax

```
B = ifft2(A)
B = ifft2(A,m,n)
```

Description

`B = 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 = ifft(A,m,n)` pads matrix `A` with zeros to size `m`-by-`n`. If `[m n]` < `size(A)`, `ifft2` crops `A` before transforming.

For any `A`, `ifft2(fft2(A))` equals `A` to within roundoff error. If `A` is real, `ifft2(fft2(A))` may have small imaginary parts.

Class Support

The input matrix `A` can be of class `double` or of any integer class. The output matrix `B` is of class `double`.

Remarks

`ifft2` is a function in MATLAB.

Algorithm

The algorithm for `ifft2(A)` is the same as the algorithm for `fft2(A)`, except for a sign change and scale factors of `[m,n]` = `size(A)`. Like `fft2`, the execution time is fastest when `m` and `n` are powers of 2 and slowest when they are large prime numbers.

See Also

`fft2`, `fftshift`, `idct2`

`dftmtx`, `filter`, `freqz`, `specplot`, `spectrum` in the *Signal Processing Toolbox User's Guide*

`fft`, `ifft` in the MATLAB Function Reference

Purpose Compute N-dimensional inverse fast Fourier transform

Syntax

```
B = ifftn(A)
B = ifftn(A,siz)
```

Description `B = ifftn(A)` performs the N-dimensional inverse fast Fourier transform. The result `B` is the same size as `A`.

`B = ifftn(A,siz)` pads `A` with zeros (or truncates `A`) to create an N-dimensional array of size `siz` before doing the inverse transform.

For any `A`, `ifftn(fftn(A))` equals `A` within roundoff error. If `A` is real, `ifftn(fftn(A))` may have small imaginary parts.

Class Support The input matrix `A` can be of class `double` or of any integer class. The output matrix `B` is of class `double`.

Remarks `ifftn` is a function in MATLAB.

Algorithm `ifftn(A)` is equivalent to

```
B = A;
for p = 1:length(size(A))
    B = ifft(B,[],p);
end
```

This code computes the one-dimensional inverse fast Fourier transform along each dimension of `A`. The time required to compute `ifftn(A)` depends most 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 `fft2`, `fftn`, `ifft2`

im2bw

Purpose

Convert an image to a binary image, based on threshold

Syntax

```
BW = im2bw(I,level)
BW = im2bw(X,map,level)
BW = im2bw(RGB,level)
```

Description

`im2bw` 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 `BW` has values of 0 (black) for all pixels in the input image with luminance less than `level` and 1 (white) for all other pixels. (Note that you specify `level` in the range [0,1], regardless of the class of the input image.)

`BW = im2bw(I,level)` converts the intensity image `I` to black and white.

`BW = im2bw(X,map,level)` converts the indexed image `X` with colormap `map` to black and white.

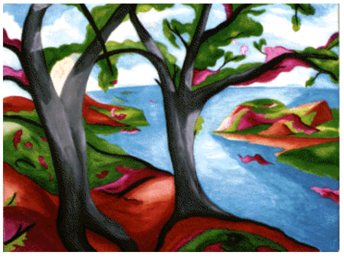
`BW = im2bw(RGB,level)` converts the RGB image `RGB` to black and white.

Class Support

The input image can be of class `uint8`, `uint16`, or `double`. The output image `BW` is of class `uint8`.

Example

```
load trees
BW = im2bw(X,map,0.4);
imshow(X,map)
figure, imshow(BW)
```



See Also

`ind2gray`, `rgb2gray`

Purpose

Rearrange image blocks into columns

Syntax

```
B = im2col(A,[m n],block_type)
B = im2col(A,[m n])
B = im2col(A,'indexed',...)
```

Description

`B = im2col(A,[m n],`*`block_type`*`)` rearranges image blocks into columns. *block_type* is a string that can have one of these values:

- `'distinct'` for `m`-by-`n` distinct blocks
- `'sliding'` for `m`-by-`n` sliding blocks (default)

`B = im2col(A,[m n],'distinct')` rearranges each distinct `m`-by-`n` block in the image `A` into a column of `B`. `im2col` pads `A` with zeros, if necessary, so its size is an integer multiple of `m`-by-`n`. If `A = [A11 A12;A21 A22]`, where each A_{ij} is `m`-by-`n`, then `B = [A11(:) A12(:) A21(:) A22(:)]`.

`B = im2col(A,[m n],'sliding')` converts each sliding `m`-by-`n` block of `A` into a column of `B`, with no zero padding. `B` has `m*n` rows and will contain as many columns as there are `m`-by-`n` neighborhoods of `A`. If the size of `A` is `[mm nn]`, then the size of `B` is (`m*n`)-by-(`(mm-m+1)*(nn-n+1)`).

`B = im2col(A,[m n])` uses the default `block_type` of `'sliding'`.

For the sliding block case, each column of `B` contains the neighborhoods of `A` reshaped as `nhood(:)` where `nhood` is a matrix containing an `m`-by-`n` neighborhood of `A`. `im2col` orders the columns of `B` so that they can be reshaped to form a matrix in the normal way. For example, suppose you use a function, such as `sum(B)`, that returns a scalar for each column of `B`. You can directly store the result in a matrix of size (`mm-m+1`)-by-(`nn-n+1`), using these calls.

```
B = im2col(A,[m n],'sliding');
C = reshape(sum(B),mm-m+1,nn-n+1);
```

`B = im2col(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 class `double` or of any integer class. The output matrix `B` is of the same class as the input image.

See Also

`blkproc`, `col2im`, `colfilt`, `nlfilter`

im2double

Purpose Convert image array to double precision

Syntax

```
I2 = im2double(I1)
RGB2 = im2double(RGB1)
BW2 = im2double(BW1)
X2 = im2double(X1,'indexed')
```

Description

`im2double` takes an image as input, and returns an image of class `double`. If the input image is of class `double`, the output image is identical to it. If the input image is of class `uint8` or `uint16`, `im2double` returns the equivalent image of class `double`, rescaling or offsetting the data as necessary.

`I2 = im2double(I1)` converts the intensity image `I1` to double precision, rescaling the data if necessary.

`RGB2 = im2double(RGB1)` converts the truecolor image `RGB1` to double precision, rescaling the data if necessary.

`BW2 = im2double(BW1)` converts the binary image `BW1` to double precision.

`X2 = im2double(X1,'indexed')` converts the indexed image `X1` to double precision, offsetting the data if necessary.

See Also `double`, `im2uint8`, `uint8`

Purpose Convert image array to eight-bit unsigned integers

Syntax

```
I2 = im2uint8(I1)
RGB2 = im2uint8(RGB1)
BW2 = im2uint8(BW1)
X2 = im2uint8(X1,'indexed')
```

Description `im2uint8` takes an image as input, and returns an image of class `uint8`. If the input image is of class `uint8`, the output image is identical to it. If the input image is of class `uint16` or `double`, `im2uint8` returns the equivalent image of class `uint8`, rescaling or offsetting the data as necessary.

`I2 = im2uint8(I1)` converts the intensity image `I1` to `uint8`, rescaling the data if necessary.

`RGB2 = im2uint8(RGB1)` converts the truecolor image `RGB1` to `uint8`, rescaling the data if necessary.

`BW2 = im2uint8(BW1)` converts the binary image `BW1` to `uint8`.

`X2 = im2uint8(X1,'indexed')` converts the indexed image `X1` to `uint8`, offsetting the data if necessary. Note that it is not always possible to convert an indexed image to `uint8`. If `X1` is of class `double`, `max(X1(:))` must be 256 or less; if `X1` is of class `uint16`, `max(X1(:))` must be 255 or less. To convert a `uint16` indexed image to `uint8` by reducing the number of colors, use `imapprox`.

See Also `im2uint16`, `double`, `im2double`, `uint8`, `imapprox`, `uint16`

im2uint16

Purpose Convert image array to sixteen-bit unsigned integers

Syntax

```
I2 = im2uint16(I1)
RGB2 = im2uint16(RGB1)
X2 = im2uint16(X1,'indexed')
```

Description `im2uint16` takes an image as input, and returns an image of class `uint16`. If the input image is of class `uint16`, the output image is identical to it. If the input image is of class `double` or `uint8`, `im2uint16` returns the equivalent image of class `uint16`, rescaling or offsetting the data as necessary.

`I2 = im2uint16(I1)` converts the intensity image `I1` to `uint16`, rescaling the data if necessary.

`RGB2 = im2uint16(RGB1)` converts the truecolor image `RGB1` to `uint16`, rescaling the data if necessary.

`X2 = im2uint16(X1,'indexed')` converts the indexed image `X1` to `uint16`, offsetting the data if necessary. Note that it is not always possible to convert an indexed image to `uint16`. If `X1` is of class `double`, `max(X1(:))` must be 65536 or less.

Note `im2uint16` does not support binary images.

See Also `im2uint8`, `double`, `im2double`, `uint8`, `uint16`, `imapprox`

Purpose Adjust image intensity values or colormap

Syntax

```
J = imadjust(I,[low_in high_in],[low_out high_out],gamma)
newmap = imadjust(map,[low_in high_in],[low_out high_out],gamma)
RGB2 = imadjust(RGB1,...)
```

Description `J = imadjust(I,[low_in high_in],[low_out high_out],gamma)` maps the values in intensity image `I` to new values in `J` such that values between `low_in` and `high_in` map to values between `low_out` and `high_out`. Values below `low_in` and above `high_in` are clipped; that is, values below `low_in` map to `low_out`, and those above `high_in` map to `high_out`. You can use an empty matrix (`[]`) for `[low_in high_in]` or for `[low_out high_out]` to specify the default of `[0 1]`. `gamma` specifies the shape of the curve describing the relationship between the values in `I` and `J`. 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. If you omit the argument, `gamma` defaults to 1 (linear mapping).

`newmap = imadjust(map,[low_in; high_in],[low_out;high_out],gamma)` transforms the colormap associated with an indexed image. If `low_in`, `high_in`, `low_out`, `high_out`, and `gamma` are scalars, then the same mapping applies to red, green and blue components. Unique mappings for each color component are possible when: `low_in` and `high_in` are both 1-by-3 vectors, `low_out` and `high_out` are both 1-by-3 vectors, or `gamma` is a 1-by-3 vector. The rescaled colormap, `newmap`, is the same size as `map`.

`RGB2 = imadjust(RGB1,...)` performs the adjustment on each image plane (red, green, and blue) of the RGB image `RGB1`. As with the colormap adjustment, you can apply unique mappings to each plane.

Note If `high_out < low_out`, the output image is reversed, as in a photographic negative.

Class Support For syntax variations that include an input image (rather than a colormap), the input image can be of class `uint8`, `uint16`, or `double`. The output image has the same class as the input image. For syntax variations that include a colormap, the input and output colormaps are of class `double`.

Example

```
I = imread('pout.tif');
J = imadjust(I,[0.3 0.7],[]);
imshow(I), figure, imshow(J)
```

```
RGB1 = imread('flowers.tif');
RGB2 = imadjust(RGB1,[.2 .3 0; .6 .7 1],[]);
imshow(RGB1), figure, imshow(RGB2)
```

See Also

brighten, histeq

Purpose Approximate indexed image by one with fewer colors

Syntax

```
[Y,newmap] = imapprox(X,map,n)
[Y,newmap] = imapprox(X,map,tol)
Y = imapprox(X,map,newmap)
[...] = imapprox(...,dither_option)
```

Description `[Y,newmap] = imapprox(X,map,n)` approximates the colors in the indexed image `X` and associated colormap `map` by using minimum variance quantization. `imapprox` returns indexed image `Y` with colormap `newmap`, which has at most `n` colors.

`[Y,newmap] = imapprox(X,map,tol)` approximates the colors in `X` and `map` through uniform quantization. `newmap` contains at most `(floor(1/tol)+1)^3` colors. `tol` must be between 0 and 1.0.

`Y = imapprox(X,map,newmap)` approximates the colors in `map` by using colormap mapping to find the colors in `newmap` that best match the colors in `map`.

`Y = imapprox(...,`*`dither_option`*`)` enables or disables dithering. *dither_option* is a string that can have one of these values:

- `'dither'` dithers, if necessary, to achieve better color resolution at the expense of spatial resolution (default).
- `'nodither'` maps each color in the original image to the closest color in the new map. No dithering is performed.

Class Support The input image `X` can be of class `uint8`, `uint16`, or `double`. The output image `Y` is of class `uint8` if the length of `newmap` is less than or equal to 256. If the length of `newmap` is greater than 256, `X` is of class `double`.

Algorithm `imapprox` uses `rgb2ind` to create a new colormap that uses fewer colors.

See Also `cmunique`, `dither`, `rgb2ind`

imcontour

Purpose Create a contour plot of image data

Syntax

```
imcontour(I,n)
imcontour(I,v)
imcontour(x,y,...)
imcontour(...,LineSpec)
[C,h] = imcontour(...)
```

Description `imcontour(I,n)` draws a contour plot of the intensity image `I`, automatically setting up the axes so their orientation and aspect ratio match the image. `n` 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.

`imcontour(I,v)` draws a contour plot of `I` with contour lines at the data values specified in vector `v`. The number of contour levels is equal to `length(v)`.

`imcontour(x,y,...)` uses the vectors `x` and `y` to specify the x- and y-axis limits.

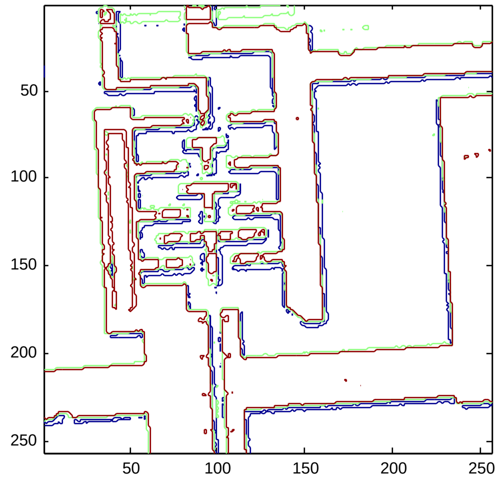
`imcontour(...,LineSpec)` draws the contours using the line type and color specified by `LineSpec`. Marker symbols are ignored.

`[C,h] = imcontour(...)` returns the contour matrix `C` 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 `clabel` function with the contour matrix `C` to add contour labels to the plot.

Class Support The input image can be of class `uint8`, `uint16`, or `double`.

Example

```
I = imread('ic.tif');
imcontour(I,3)
```



See Also

`clabel`, `contour`, `LineSpec` in the MATLAB Function Reference

imcrop

Purpose

Crop an image

Syntax

```
I2 = imcrop(I)
X2 = imcrop(X,map)
RGB2 = imcrop(RGB)

I2 = imcrop(I,rect)
X2 = imcrop(X,map,rect)
RGB2 = imcrop(RGB,rect)

[...] = imcrop(x,y,...)
[A,rect] = imcrop(...)
[x,y,A,rect] = imcrop(...)
```

Description

`imcrop` crops an image to a specified rectangle. In the syntaxes below, `imcrop` displays the input image and waits for you to specify the crop rectangle with the mouse.

```
I2 = imcrop(I)
X2 = imcrop(X,map)
RGB2 = imcrop(RGB)
```

If you omit the input arguments, `imcrop` operates on the image in the current axes.

To specify the rectangle:

- 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.

If you hold down the **Shift** key while dragging, or if you press the right mouse button on a 2- or 3-button mouse, `imcrop` constrains the bounding rectangle to be a square.

When you release the mouse button, `imcrop` returns the cropped image in the supplied output argument. If you do not supply an output argument, `imcrop` displays the output image in a new figure.

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 four-element vector with the form `[xmin ymin 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 two-element vectors specifying the `XData` and `YData`. For example,

```
[...] = imcrop(x,y,...)
```

If you supply additional output arguments, `imcrop` returns information about the selected rectangle and the coordinate system of the input image. For example,

```
[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

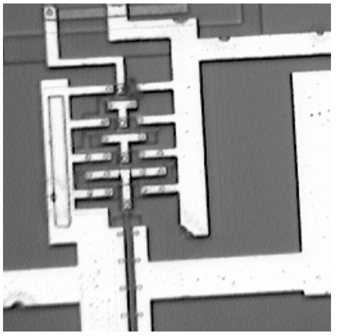
The input image `A` can be of class `uint8`, `uint16`, or `double`. The output image `B` is of the same class as `A`. `rect` is always of class `double`.

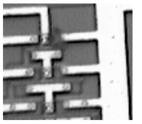
Remarks

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 *or partially* enclosed by the rectangle.

Example

```
I = imread('ic.tif');
I2 = imcrop(I,[60 40 100 90]);
imshow(I)
figure, imshow(I2)
```





See Also zoom

Purpose

Compute feature measurements for image regions

Syntax

```
stats = imfeature(L,measurements)
stats = imfeature(L,measurements,n)
```

Description

`stats = imfeature(L,`*`measurements`*`)` computes a set of measurements for each labeled region in the label matrix `L`. Positive integer elements of `L` correspond to different regions. For example, the set of elements of `L` equal to 1 corresponds to region 1; the set of elements of `L` equal to 2 corresponds to region 2; and so on. `stats` is a structure array of length `max(L(:))`. The fields of the structure array denote different measurements for each region, as specified by *measurements*.

measurements can be a comma-separated list of strings, a cell array containing strings, the single string `'all'`, or the single string `'basic'`. The set of valid measurement strings includes the following.

`'Area'`	`'Image'`	`'EulerNumber'`
`'Centroid'`	`'FilledImage'`	`'Extrema'`
`'BoundingBox'`	`'FilledArea'`	`'EquivDiameter'`
`'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.

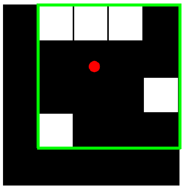
Definitions

`'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.



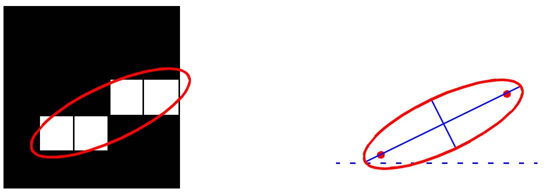
`'MajorAxisLength'` – Scalar; the length (in pixels) of the major axis of the ellipse that has the same second-moments as the region.

`'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.

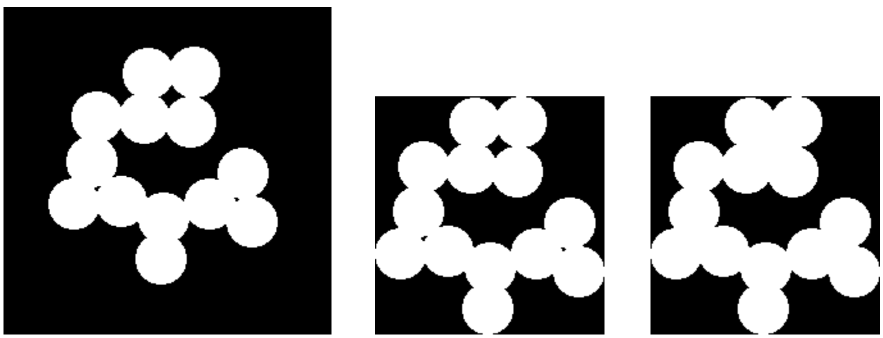


`'Image'` – Binary image (`uint8`) of the same size as the bounding box of the region; the on pixels correspond to the region, and all other pixels are `off`.

`'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.

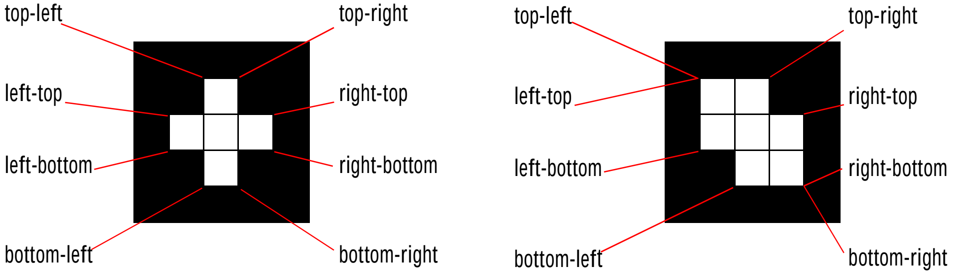
'ConvexImage' – Binary image (uint8); the convex hull, with all pixels within the hull filled in (i.e., set to on). (For pixels that the boundary of the hull passes through, imfeature uses the same logic as roipoly to determine whether the pixel is inside or outside the hull.) The image is the size of 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.



'EquivDiameter' – Scalar; the diameter of a circle with the same area as the region. Computed as sqrt(4*Area/pi).

'Solidity' – Scalar; the proportion of the pixels in the convex hull that are also in the region. Computed as Area/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 `double` or of any integer class.

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]
      EquivDiameter: 10.6451
           Solidity: 0.4341
             Extent: 0.3708
          PixelList: [89x2  double]
```

See Also

`bwlabel`

`ismember` in the MATLAB Function Reference

Purpose

Information about graphics file

Syntax

```
info = imfinfo(filename,fmt)
info = imfinfo(filename)
```

Description

`info = imfinfo(filename,`*`fmt`*`)` returns a structure whose fields contain information about an image in a graphics file. `filename` is a string that specifies the name of the graphics file, and `fmt` 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 `imfinfo` cannot find a file named `filename`, it looks for a file named `filename.fmt`.

This table lists the possible values for *`fmt`*.

Format	File Type
`'bmp'`	Windows Bitmap (BMP)
`'cur'`	Windows Cursor resources (CUR)
`'hdf'`	Hierarchical Data Format (HDF)
`'ico'`	Windows Icon resources (ICO)
`'jpg'` or `'jpeg'`	Joint Photographic Experts Group (JPEG)
`'pcx'`	Windows Paintbrush (PCX)
`'png'`	Portable Network Graphics (PNG)
`'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
Filename	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
FileModDate	A string containing the date when the file was last modified
FileSize	An integer indicating the size of the file in bytes
Format	A string containing the file format, as specified by fmt; for JPEG and TIFF files, the three-letter variant is returned
FormatVersion	A string or number describing the version of the format
Width	An integer indicating the width of the image in pixels
Height	An integer indicating the height of the image in pixels
BitDepth	An integer indicating the number of bits per pixel
ColorType	A string indicating the type of image; either 'truecolor' for a truecolor RGB image, 'grayscale' for a grayscale intensity image, or 'indexed' 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.

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`

imhist

Purpose Display a histogram of image data

Syntax

```
imhist(I,n)
imhist(X,map)
[counts,x] = imhist(...)
```

Description

`imhist(I,n)` displays a histogram with `n` bins for the intensity image `I` above a grayscale colorbar of length `n`. If you omit the argument, `imhist` uses a default value of `n` = 256 if `I` is a grayscale image, or `n` = 2 if `I` is a binary image.

`imhist(X,map)` 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.

`[counts,x] = imhist(...)` returns the histogram counts in `counts` and the bin locations in `x` so that `stem(x,counts)` shows the histogram. For indexed images, it returns the histogram counts for each colormap entry; the length of `counts` is the same as the length of the colormap.

Note For intensity images, the `n` bins of the histogram are each half-open intervals of width $A/(n-1)$. In particular, the p th bin is the half-open interval

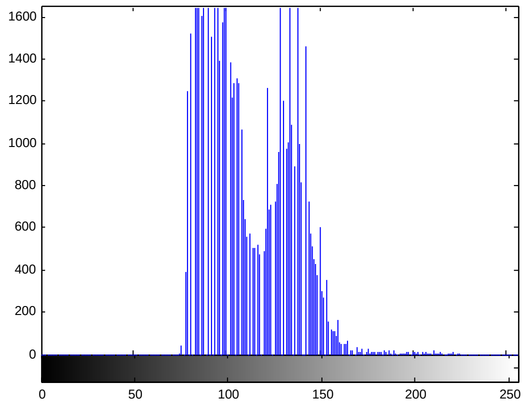
$$A(p-1.5)/(n-1) \le x < A(p-0.5)/(n-1)$$

The scale factor A depends on the image class. A is 1 if the intensity image is `double`; A is 255 if the intensity image is `uint8`; and A is 65535 if the intensity image is `uint16`.

Class Support The input image can be of class `uint8`, `uint16`, or `double`.

Example

```
I = imread('pout.tif');
imhist(I)
```



See Also

`histeq`

`hist` in the MATLAB Function Reference

immovie

Purpose Make a movie of a multiframe indexed image

Syntax

```
mov = immovie(X,map)
```

Description `mov = immovie(X,map)` returns the movie matrix `mov` from the images in the multiframe indexed image `X`. As it creates the movie matrix, it displays the movie frames on the screen. You can play the movie using the MATLAB `movie` function.

`X` comprises multiple indexed images, all having the same size and all using the colormap `map`. `X` is an `m`-by-`n`-by-1-by-`k` array, where `k` is the number of images.

Class Support `X` can be of class `uint8`, `uint16`, or `double`. `mov` is of class `double`.

Example

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

See Also `montage`

`avifile`, `getframe`, `movie`, `movie2avi` in the MATLAB Function Reference

Remarks You can also make movies from images by using the MATLAB function `avifile`, which creates AVI files. In addition, you can convert an existing MATLAB movie into an AVI file by using the `movie2avi` function.

Purpose

Add noise to an image

Syntax

```
J = imnoise(I,type)
J = imnoise(I,type,parameters)
```

Description

`J = imnoise(I,`*`type`*`)` adds noise of type to the intensity image `I`. *type* is a string that can have one of these values:

- `'gaussian'` for Gaussian white noise
- `'salt & pepper'` for "on and off" pixels
- `'speckle'` for multiplicative noise

`J = imnoise(I,`*`type`*`,parameters)` accepts an algorithm *type* plus additional modifying parameters particular to the type of algorithm chosen. If you omit these arguments, `imnoise` uses default values for the parameters. Here are examples of the different noise types and their parameters:

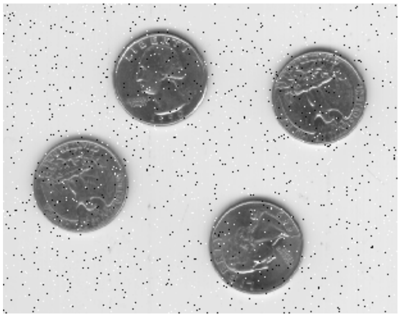
- `J = imnoise(I,'gaussian',m,v)` adds Gaussian white noise of mean `m` and variance `v` to the image `I`. The default is zero mean noise with 0.01 variance.
- `J = imnoise(I,'salt & pepper',d)` adds salt and pepper noise to the image I, where `d` is the noise density. This affects approximately `d*prod(size(I))` pixels. The default is 0.05 noise density.
- `J = imnoise(I,'speckle',v)` adds multiplicative noise to the image `I`, using the equation `J = I + n*I`, where `n` is uniformly distributed random noise with mean 0 and variance `v`. The default for `v` is 0.04.

Class Support

The input image `I` can be of class `uint8`, `uint16`, or `double`. The output image `J` is of the same class as `I`.

Example

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



See Also rand, randn in the 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 kth 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. For example,

```
[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 two-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, uint16, or double. All other inputs and outputs are of class double.

Remarks

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.

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

improfile

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.

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 two-element vectors specifying the image `XData` and `YData`.

`[...] = improfile(...,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.

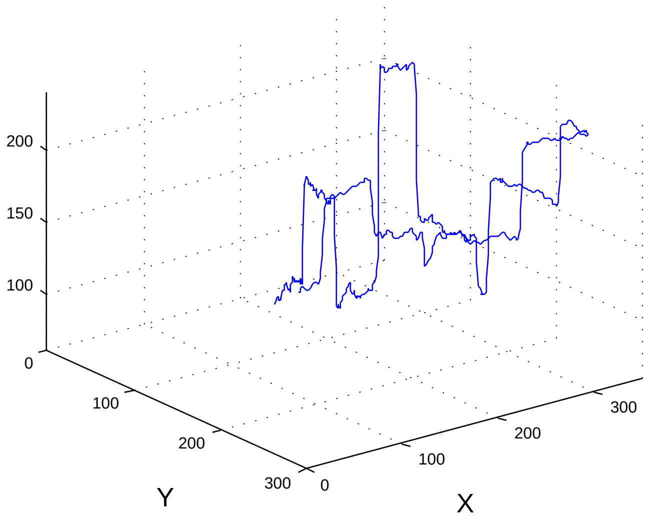
If you omit the `method` argument, `improfile` uses the default method of `'nearest'`.

Class Support

The input image can be of class `uint8`, `uint16`, or `double`. All other inputs and outputs are of class `double`.

Example

```
I = imread('alumgrns.tif');
x = [35 338 346 103];
y = [253 250 17 148];
improfile(I,x,y), grid on
```



See Also

`impixel`, `pixval`

`interp2` in the MATLAB Function Reference

Purpose Read image from graphics files

Syntax

```
A = imread(filename,fmt)
[X,map] = imread(filename,fmt)
[...] = imread(filename)
[...] = imread(...,idx)    (CUR, ICO, and TIFF only)
[...] = imread(...,ref)    (HDF only)
[...] = imread(...,'BackgroundColor',BG)  (PNG only)
[A,map,alpha] = imread(...)  (PNG only)
```

Description

`A = imread(filename,`*`fmt`*`)` reads a grayscale or truecolor image named `filename` into `A`. If the file contains a grayscale intensity image, `A` is a two-dimensional array. If the file contains a truecolor (RGB) image, `A` is a three-dimensional (m-by-n-by-3) array.

`[X,map] = imread(filename,`*`fmt`*`)` reads the indexed image in `filename` into `X` and its associated colormap into `map`. The colormap values are rescaled to the range [0,1]. `A` and `map` are two-dimensional arrays.

`[...] = imread(filename)` attempts to infer the format of the file from its content.

`filename` is a string that specifies the name of the graphics file, and *fmt* is a string that specifies the format of the file. If the file is not in the current directory or in a directory in the MATLAB path, specify the full pathname for a location on your system. If `imread` cannot find a file named `filename`, it looks for a file named `filename.fmt`. If you do not specify a string for *fmt*, the toolbox will try to discern the format of the file by checking the file header.

This table lists the possible values for *fmt*.

Format	File Type
`'bmp'`	Windows Bitmap (BMP)
`'cur'`	Windows Cursor resources (CUR)
`'hdf'`	Hierarchical Data Format (HDF)
`'ico'`	Windows Icon resources (ICO)

Format	File Type
`'jpg'` or `'jpeg'`	Joint Photographic Experts Group (JPEG)
`'pcx'`	Windows Paintbrush (PCX)
`'png'`	Portable Network Graphics (PNG)
`'tif'` or `'tiff'`	Tagged Image File Format (TIFF)
`'xwd'`	X Windows Dump (XWD)

Special Case Syntax:

TIFF-Specific Syntax

`[...] = 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.

PNG-Specific Syntax

The discussion in this section is only relevant to PNG files that contain transparent pixels. A PNG file does not necessarily contain transparency data. Transparent pixels, when they exist, will be identified by one of two components: a *transparency chunk* or an *alpha channel*. (A PNG file can only have one of these components, not both.)

The transparency chunk identifies which pixel values will be treated as transparent, e.g., if the value in the transparency chunk of an 8-bit image is 0.5020, all pixels in the image with the color 0.5020 can be displayed as transparent. An alpha channel is an array with the same number of pixels as are in the image, which indicates the transparency status of each corresponding pixel in the image (transparent or nontransparent).

Another potential PNG component related to transparency is the *background color chunk*, which (if present) defines a color value that can be used behind all transparent pixels. This section identifies the default behavior of the toolbox for reading PNG images that contain either a transparency chunk or an alpha channel, and describes how you can override it.

Case 1. You do not ask to output the alpha channel and do not specify a background color to use. For example,

```
[A,map] = imread(filename);
A = imread(filename);
```

If the PNG file contains a background color chunk, the transparent pixels will be composited against the specified background color.

If the PNG file does not contain a background color chunk, the transparent pixels will be composited against `0` for grayscale (black), `1` for indexed (first color in map), or `[0 0 0]` for RGB (black).

Case 2. You do not ask to output the alpha channel but you specify the background color parameter in your call. For example,

```
[...] = imread(...,'BackgroundColor',bg);
```

The transparent pixels will be composited against the specified color. The form of `bg` depends on whether the file contains an indexed, intensity (grayscale), or RGB image. If the input image is indexed, `bg` should be an integer in the range `[1,P]` where `P` is the colormap length. If the input image is intensity, `bg` should be an integer in the range [0,1]. If the input image is RGB, `bg` should be a three-element vector whose values are in the range [0,1].

There is one exception to the toolbox's behavior of using your background color. If you set background to `'none'` no compositing will be performed. For example,

```
[...] = imread(...,'Back','none');
```

Note If you specify a background color, you *cannot* output the alpha channel.

Case 3. You ask to get the alpha channel as an output variable. For example,

```
[A,map,alpha] = imread(filename);
[A,map,alpha] = imread(filename,fmt);
```

No compositing is performed; the alpha channel will be stored separately from the image (not merged into the image as in cases 1 and 2). This form of `imread` returns the alpha channel if one is present, and also returns the image and any associated colormap. If there is no alpha channel, `alpha` returns `[]`. If there is no colormap, or the image is grayscale or truecolor, map may be empty.

HDF-Specific Syntax

`[...] = 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. You can use `imfinfo` to match up image order with reference number.) If you omit this argument, `imread` reads the first image in the file.

CUR- and ICO-Specific Syntax

`[...] = imread(...,idx)` reads in one image from a multi-image icon or cursor file. `idx` is an integer value that specifies the order that 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.

`[A,map,alpha] = imread(...)` returns the AND mask for the resource, which can be used to determine the transparency information. For cursor files, this mask may contain the only useful data.

Note By default, Microsoft Windows cursors are 32-by-32 pixels. MATLAB pointers must be 16-by-16. You will probably need to scale your image. If you have the Image Processing Toolbox, you can use the `imresize` function.

Format Support

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
CUR	1-bit, 4-bit, and 8-bit uncompressed images
HDF	8-bit raster image datasets, with or without associated colormap; 24-bit raster image datasets
ICO	1-bit, 4-bit, and 8-bit uncompressed images

Format	Variants
JPEG	Any baseline JPEG image (8 or 24-bit); JPEG images with some commonly used extensions
PCX	1-bit, 8-bit, and 24-bit images
PNG	Any PNG image, including 1-bit, 2-bit, 4-bit, 8-bit, and 16-bit grayscale images; 8-bit and 16-bit indexed images; 24-bit and 48-bit RGB images
TIFF	Any baseline TIFF image, including 1-bit, 8-bit, and 24-bit uncompressed images; 1-bit, 8-bit, 16-bit, and 24-bit images with packbits compression; 1-bit images with CCITT compression; also 16-bit grayscale, 16-bit indexed, and 48-bit RGB images.
XWD	1-bit and 8-bit ZPixmaps; XYBitmaps; 1-bit XYPixmaps

Class Support

In most of the image file formats supported by `imread`, pixels are stored using eight or fewer bits per color plane. When reading such a file, the class of the output (`A` or `X`) is `uint8`. `imread` also supports reading 16-bit-per-pixel data from TIFF and PNG files; for such image files, the class of the output (`A` or `X`) is `uint16`. Note that for indexed images, `imread` always reads the colormap into an array of class `double`, even though the image array itself may be of class `uint8` or `uint16`.

Remarks

`imread` is a function in MATLAB.

Examples

This example reads the sixth image in a TIFF file.

```
[X,map] = imread('flowers.tif',6);
```

This example reads the fourth image in an HDF file.

```
info = imfinfo('skull.hdf');
[X,map] = imread('skull.hdf',info(4).Reference);
```

This example reads a 24-bit PNG image and sets any of its fully transparent (alpha channel) pixels to red.

```
bg = [255 0 0];
```

```
A = imread('image.png','BackgroundColor',bg);
```

This example returns the alpha channel (if any) of a PNG image.

```
[A,map,alpha] = imread('image.png');
```

This example reads an ICO image, applies a transparency mask, and then displays the image.

```
[a,b,c] = imread('myicon.ico');
% Augment colormap for background color (white).
b2 = [b; 1 1 1];
% Create new image for display.
d = ones(size(a)) * (length(b2) - 1);
% Use the AND mask to mix the background and
% foreground data on the new image
d(c == 0) = a(c == 0);
% Display new image
imshow(uint8(d), b2)
```

See Also `double`, `fread`, `imfinfo`, `imwrite`, `uint8`, `uint16`

Purpose Resize an image

Syntax

```
B = imresize(A,m,method)
B = imresize(A,[mrows ncols],method)

B = imresize(...,method,n)
B = imresize(...,method,h)
```

Description

`imresize` resizes an image of any type using 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, `imresize` uses the default method of `'nearest'`.

`B = imresize(A,m,`*method*`)` returns an image that is `m` times the size of `A`. If `m` is between 0 and 1.0, `B` is smaller than `A`. If `m` is greater than 1.0, `B` is larger than `A`.

`B = imresize(A,[mrows ncols],`*method*`)` returns an image of size `[mrows ncols]`. If the specified size does not produce the same aspect ratio as the input image has, the output image is distorted.

When the specified output size is smaller than the size of the input image, and `method` is `'bilinear'` or `'bicubic'`, `imresize` applies a lowpass filter before interpolation to reduce aliasing. The default filter size is 11-by-11.

You can specify a different order for the default filter using

```
[...] = imresize(...,method,n)
```

`n` is an integer scalar specifying the size of the filter, which is `n`-by-`n`. If `n` is 0 (zero), `imresize` omits the filtering step.

You can also specify your own filter `h` using

```
[...] = imresize(...,method,h)
```

h is any two-dimensional FIR filter (such as those returned by ftrans2, fwind1, fwind2, or fsamp2).

Class Support

The input image can be of class uint8, uint16, or double. The output image is of the same class as the input image.

See Also

interp2 in the MATLAB Function Reference

Purpose

Rotate an image

Syntax

```
B = imrotate(A,angle,method)
B = imrotate(A,angle,method,'crop')
```

Description

`B = imrotate(A,angle,`*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:

- `'nearest'` (default) uses nearest neighbor interpolation.
- `'bilinear'` uses bilinear interpolation.
- `'bicubic'` uses bicubic interpolation.

If you omit the *method* argument, `imrotate` uses the default method of `'nearest'`.

The returned image matrix `B` is, in general, larger than `A` to include the whole rotated image. `imrotate` sets invalid values on the periphery of `B` to 0.

`B = imrotate(A,angle,`*method*`,'crop')` rotates the image `A` through `angle` degrees and returns the central portion which is the same size as `A`.

Class Support

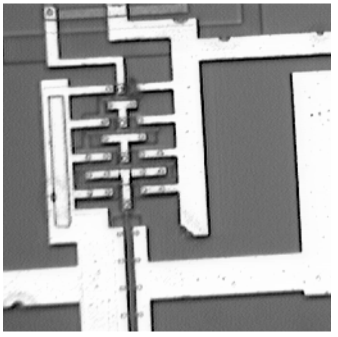
The input image can be of class `uint8`, `uint16`, or `double`. The output image is of the same class as the input image.

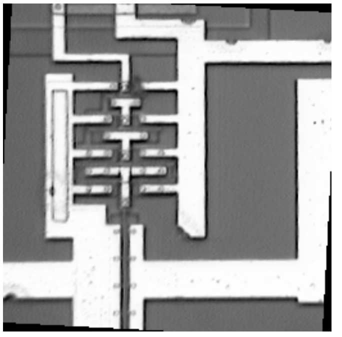
Remarks

To rotate the image clockwise, specify a negative angle.

Example

```
I = imread('ic.tif');
J = imrotate(I,-4,'bilinear','crop');
imshow(I)
figure, imshow(J)
```





See Also imcrop, imresize

Purpose

Display an image

Syntax

```
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(...)
```

Description

`imshow(I,n)` displays the intensity image `I` with `n` discrete levels of gray. If you omit `n`, `imshow` uses 256 gray levels on 24-bit displays, or 64 gray levels on other systems.

`imshow(I,[low high])` displays `I` as a grayscale intensity image, specifying the data range for `I`. The value `low` (and any value less than `low`) displays as black, the value `high` (and any value greater than `high`) displays as white, and values in between display as intermediate shades of gray. `imshow` uses the default number of gray levels. If you use an empty matrix (`[]`) for `[low high]`, `imshow` uses `[min(I(:)) max(I(:))]`; the minimum value in `I` displays as black, and the maximum value displays as white.

`imshow(BW)` displays the binary image `BW`. Values of 0 display as black, and values of 1 display as white.

`imshow(X,map)` displays the indexed image `X` with the colormap `map`.

`imshow(RGB)` displays the truecolor image `RGB`.

`imshow(...,display_option)` displays the image, calling `truesize` if `display_option` is `'truesize'`, or suppressing the call to `truesize` if `display_option` is `'notruesize'`. Either option string can be abbreviated. If you do not supply this argument, `imshow` determines whether to call `truesize` based on the setting of the `'ImshowTruesize'` preference.

`imshow(x,y,A,...)` uses the two-element vectors `x` and `y` to establish a nondefault spatial coordinate system, by specifying the image `XData` and `YData`.

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, uint16, or double.

Remarks

You can use the iptsetpref function to set several toolbox preferences that modify the behavior of imshow. For example,

- '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_option 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 MATLAB Function Reference

Purpose Write image to graphics file

Syntax

```
imwrite(A,filename,fmt)
imwrite(X,map,filename,fmt)
imwrite(...,filename)
imwrite(...,Param1,Val1,Param2,Val2...)
```

Description

`imwrite(A,filename,`*`fmt`*`)` writes the image in `A` to `filename` in the format specified by *`fmt`*. `A` can be either a grayscale image (M-by-N) or a truecolor image (M-by-N-by-3). If `A` is of class `uint8 or uint16`, `imwrite` writes the actual values in the array to the file. If `A` is of class `double`, `imwrite` rescales the values in the array before writing, using `uint8(round(255*A))`. This operation converts the floating-point numbers in the range [0,1] to 8-bit integers in the range [0,255].

`imwrite(X,map,filename,`*`fmt`*`)` writes the indexed image in `X` and its associated colormap `map` to `filename` in the format specified by *`fmt`*. If `X` is of class `uint8 or uint16`, `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)`. (See note below for an exception.) `map` must be a valid MATLAB colormap of class `double`; `imwrite` rescales the values in `map` using `uint8(round(255*map))`. Note that most image file formats do not support colormaps with more than 256 entries.

Note If the image is `double`, and you specify PNG as the output format and a bit depth of 16 bpp, the values in the array will be offset using `uint16(X-1)`.

`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(...,Param1,Val1,Param2,Val2...)` specifies parameters that control various characteristics of the output file. Parameter settings can currently be made for HDF, PNG, JPEG, and TIFF files. For example, if you are writing a JPEG file, you can set the "quality" of the JPEG compression. For the lists of parameters available for each format, see the tables below.

`filename` is a string that specifies the name of the output file, and *fmt* is a string that specifies the format of the file.

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)
'png'	Portable Network Graphics (PNG)
'tif' or 'tiff'	Tagged Image File Format (TIFF)
'xwd'	X Windows Dump (XWD)

This table describes the available parameters for HDF files.

Parameter	Values	Default
'Compression'	One of these strings: 'none' (the default), 'rle', 'jpeg'. 'rle' is valid only for grayscale and indexed images. 'jpeg' is valid only for grayscale and RGB images.	'rle'
'Quality'	A number between 0 and 100; this parameter applies only if 'Compression' is 'jpeg'. Higher numbers mean higher *quality* (less image degradation due to compression), but the resulting file size is larger.	75
'WriteMode'	One of these strings: 'overwrite' (the default), or 'append'.	'overwrite'

This table describes the available parameters for JPEG files.

Parameter	Values	Default
`'Quality'`	A number between 0 and 100; higher numbers mean higher *quality* (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'`, `'fax3'`, or `'fax4'`. The `'ccitt'`, `'fax3'`, and `'fax4'` compression schemes are valid for binary images only.	`'ccitt'` for binary images; `'packbits'` for nonbinary images
`'Description'`	Any string; fills in the `ImageDescription` field returned by `imfinfo`.	empty
`'Resolution'`	A two-element vector containing the XResolution and YResolution, or a scalar indicating both resolutions.	72
`'WriteMode'`	One of these strings: `'overwrite'` or `'append'`	`'overwrite'`

This table describes the available parameters for PNG files.

Parameter	Values	Default
`'Author'`	A string	Empty
`'Description'`	A string	Empty
`'Copyright'`	A string	Empty
`'CreationTime'`	A string	Empty
`'Software'`	A string	Empty
`'Disclaimer'`	A string	Empty

Parameter	Values	Default
'Warning'	A string	Empty
'Source'	A string	Empty
'Comment'	A string	Empty
'InterlaceType'	Either 'none' or 'adam7'	'none'
'BitDepth'	A scalar value indicating desired bit depth. For grayscale images this can be 1, 2, 4, 8, or 16. For grayscale images with an alpha channel this can be 8 or 16. For indexed images this can be 1, 2, 4, or 8. For truecolor images with or without an alpha channel this can be 8 or 16.	8 bits per pixel if image is double or uint8 16 bits per pixel if image is uint16 1 bit per pixel if image is logical
'Transparency'	This value is used to indicate transparency information only when no alpha channel is used. Set to the value that indicates which pixels should be considered transparent. (If the image uses a colormap, this value will represent an index number to the colormap.) For indexed images: a Q-element vector in the range [0,1] where Q is no larger than the colormap length and each value indicates the transparency associated with the corresponding colormap entry. In most cases, Q=1. For grayscale images: a scalar in the range [0,1]. The value indicates the grayscale color to be considered transparent. For truecolor images: a three-element vector in the range [0,1]. The value indicates the truecolor color to be considered transparent. You cannot specify 'Transparency' and 'Alpha' at the same time.	Empty

Parameter	Values	Default
'Background'	The value specifies background color to be used when compositing transparent pixels. For indexed images: an integer in the range [1,P], where P is the colormap length. For grayscale images: a scalar in the range [0,1]. For truecolor images: a three-element vector in the range [0,1].	Empty
'Gamma'	A nonnegative scalar indicating the file gamma	Empty
'Chromaticities'	An eight-element vector `[wx wy rx ry gx gy bx by]` that specifies the reference white point and the primary chromaticities	Empty
'XResolution'	A scalar indicating the number of pixels/unit in the horizontal direction	Empty
'YResolution'	A scalar indicating the number of pixels/unit in the vertical direction	Empty
'ResolutionUnit'	Either 'unknown' or 'meter'	Empty
'Alpha'	A matrix specifying the transparency of each pixel individually. The row and column dimensions must be the same as the data array; they can be `uint8`, `uint16`, or `double`, in which case the values should be in the range [0,1].	Empty
'SignificantBits'	A scalar or vector indicating how many bits in the data array should be regarded as significant; values must be in the range [1,BitDepth]. For indexed images: a three-element vector. For grayscale images: a scalar. For grayscale images with an alpha channel: a two-element vector. For truecolor images: a three-element vector. For truecolor images with an alpha channel: a four-element vector	Empty

In addition to these PNG parameters, you can use any parameter name that satisfies the PNG specification for keywords, including only printable characters, 80 characters or fewer, and no leading or trailing spaces. The value corresponding to these user-specified parameters must be a string that contains no control characters other than linefeed.

Format Support

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; uncompressed or with RLE or JPEG compression
JPEG	Baseline JPEG images (8 or 24-bit) **Note:** Indexed images are converted to RGB before writing out JPEG files, because the JPEG format does not support indexed images.
PCX	8-bit images
PNG	1-bit, 2-bit, 4-bit, 8-bit, and 16-bit grayscale images; 8-bit and 16-bit grayscale images with alpha channels; 1-bit, 2-bit, 4-bit, and 8-bit indexed images; 24-bit and 48-bit truecolor images with or without alpha channels
TIFF	Baseline TIFF images, including 1-bit, 8-bit, and 24-bit uncompressed images; 1-bit, 8-bit, and 24-bit images with packbits compression; 1-bit images with CCITT 1D, Group 3, and Group 4 compression
XWD	8-bit ZPixmaps

Class Support

Most of the supported image file formats store `uint8` data. PNG and TIFF additionally support `uint16` data. For grayscale and RGB images, if the data array is `double`, the assumed dynamic range is [0,1]. The data array is automatically scaled by 255 before being written out as `uint8`. If the data array is `uint8` or `uint16` (PNG and TIFF only), then it is written out without scaling as `uint8` or `uint16`, respectively.

Note If a logical `double` or `uint8` is written to a PNG or TIFF file, it is assumed to be a binary image and will be written with a bit depth of 1.

For indexed images, if the index array is `double`, then the indices are first converted to zero-based indices by subtracting 1 from each element, and then they are written out as `uint8`. If the index array is `uint8` or `uint16` (PNG and TIFF only), then it is written out without modification as `uint8` or `uint16`, respectively. When writing PNG files, you can override this behavior with the `'BitDepth'` parameter; see the PNG table in this `imwrite` reference for details.

Remarks

`imwrite` is a function in MATLAB.

Example

This example appends an indexed image `X` and its colormap `map` to an existing uncompressed multipage HDF file named `flowers.hdf`.

```
imwrite(X,map,'flowers.hdf','Compression','none',...
'WriteMode','append')
```

See Also

`fwrite`, `imfinfo`, `imread`

ind2gray

Purpose Convert an indexed image to an intensity image

Syntax

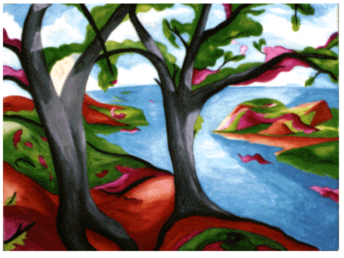
```
I = ind2gray(X,map)
```

Description `I = ind2gray(X,map)` converts the image `X` with colormap `map` to an intensity image `I`. `ind2gray` removes the hue and saturation information from the input image while retaining the luminance.

Class Support `X` can be of class `uint8`, `uint16`, or `double`. `I` is of class `double`.

Example

```
load trees
I = ind2gray(X,map);
imshow(X,map)
figure,imshow(I)
```



Algorithm `ind2gray` converts the colormap to NTSC coordinates using `rgb2ntsc`, and sets the hue and saturation components (*I* and *Q*) to zero, creating a gray colormap. `ind2gray` then replaces the indices in the image `X` with the corresponding grayscale intensity values in the gray colormap.

See Also `gray2ind`, `imshow`, `rgb2ntsc`

Purpose Convert an indexed image to an RGB image

Syntax

```
RGB = ind2rgb(X,map)
```

Description `RGB = ind2rgb(X,map)` converts the matrix `X` and corresponding colormap `map` to RGB (truecolor) format.

Class Support `X` can be of class `uint8`, `uint16`, or `double`. `RGB` is an `m`-by-`n`-by-3 array of class `double`.

See Also `ind2gray`, `rgb2ind`

iptgetpref

Purpose

Get Image Processing Toolbox preference

Syntax

```
value = iptgetpref(prefname)
```

Description

`value = iptgetpref(prefname)` returns the value of the Image Processing Toolbox preference specified by the string `prefname`. Preference names are case insensitive and can be abbreviated.

`iptgetpref` without an input argument displays the current setting of all Image Processing Toolbox preferences.

Example

```
value = iptgetpref('ImshowAxesVisible')

value =

off
```

See Also

`imshow`, `iptsetpref`

Purpose Set Image Processing Toolbox preference

Syntax

```
iptsetpref(prefname,value)
```

Description

`iptsetpref(prefname,value)` sets the Image Processing Toolbox preference specified by the string `prefname` to `value`. 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 `startup.m` file.)

This table describes the available preferences. Note that the preference names are case insensitive and can be abbreviated.

Preference Name	Values
`'ImshowBorder'`	`'loose'` (default) or `'tight'`
If `'ImshowBorder'` is `'loose'`, `imshow` 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 `'ImshowBorder'` is `'tight'`, `imshow` 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.)	
`'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`

`axis` in the MATLAB Function Reference

Purpose Compute inverse Radon transform

Syntax

```
I = iradon(P,theta)
I = iradon(P,theta,interp,filter,d,n)
[I,h] = iradon(...)
```

Description

`I = iradon(P,theta)` reconstructs the image `I` from projection data in the two-dimensional array `P`. The columns of `P` are parallel beam projection data. `iradon` assumes that the center of rotation is the center point of the projections, which is defined as `ceil(size(P,1)/2)`.

`theta` describes the angles (in degrees) at which the projections were taken. It can be either a vector containing the angles or a scalar specifying `D_theta`, the incremental angle between projections. If `theta` is a vector, it must contain angles with equal spacing between them. If `theta` is a scalar specifying `D_theta`, the projections are taken at angles `theta = m*D_theta`, where `m = 0,1,2,...,size(P,2)–1`. If the input is the empty matrix (`[]`), `D_theta` defaults to `180/size(P,2)`.

`I = iradon(P,theta,`*`interp`*`,`*`filter`*`,d,n)` specifies parameters to use in the inverse Radon transform. You can specify any combination of the last four arguments. `iradon` uses default values for any of these arguments that you omit.

interp specifies the type of interpolation to use in the backprojection. The available options are listed in order of increasing accuracy and computational complexity:

- `'nearest'` – nearest neighbor interpolation
- `'linear'` – linear interpolation (default)
- `'spline'` – spline interpolation

filter specifies the filter to use for frequency domain filtering. *filter* is a string that specifies any of the following standard filters:

- `'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 = 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.

Example

```
P = phantom(128);
R = radon(P,0:179);
I = iradon(R,0:179,'nearest','Hann');
imshow(P)
figure, imshow(I)
```

Algorithm iradon 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

[1] Kak, Avinash C., and Malcolm Slaney, *Principles of Computerized Tomographic Imaging.* New York: IEEE Press.

isbw

Purpose Return true for a binary image

Syntax

```
flag = isbw(A)
```

Description `flag = isbw(A)` returns 1 if `A` is a binary image and 0 otherwise.

`isbw` uses these criteria to decide if `A` is a binary image:

- If `A` is of class `double`, all values must be either 0 or 1, the logical flag must be on, and the number of dimensions of `A` must be 2.
- If `A` is of class `uint8`, the logical flag must be on, and the number of dimensions of `A` must be 2.
- If `A` is of class `uint16`, it is not a binary image. (The toolbox does not support `uint16` binary images.)

Note A four-dimensional array that contains multiple binary images returns 0, not 1.

Class Support `A` can be of class `uint8`, `uint16`, or `double`.

See Also `isind`, `isgray`, `isrgb`

Purpose Return true for intensity image

Syntax

```
flag = isgray(A)
```

Description `flag = isgray(A)` returns 1 if `A` is a grayscale intensity image and 0 otherwise.

`isgray` uses these criteria to decide if `A` is an intensity image:

- If `A` is of class `double`, all values must be in the range [0,1], and the number of dimensions of `A` must be 2.
- If `A` is of class `uint16` or `uint8`, the number of dimensions of `A` must be 2.

Note A four-dimensional array that contains multiple intensity images returns 0, not 1.

Class Support `A` can be of class `uint8`, `uint16`, or `double`.

See Also `isbw`, `isind`, `isrgb`

isind

Purpose

Return true for an indexed image

Syntax

```
flag = isind(A)
```

Description

`flag = isind(A)` returns 1 if `A` is an indexed image and 0 otherwise.

`isind` uses these criteria to determine if `A` is an indexed image:

- If `A` is of class `double`, 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 `uint8`, its logical flag must be off, and the number of dimensions of `A` must be 2.
- If `A` is of class `uint16`, the number of dimensions of `A` must be 2.

Note A four-dimensional array that contains multiple indexed images returns 0, not 1.

Class Support

`A` can be of class `uint8`, `uint16`, or `double`.

See Also

`isbw`, `isgray`, `isrgb`

Purpose Return true for an RGB image

Syntax

```
flag = isrgb(A)
```

Description

`flag = isrgb(A)` returns 1 if `A` is an RGB truecolor image and 0 otherwise.

`isrgb` uses these criteria to determine if `A` is an RGB image:

- If `A` is of class `double`, all values must be in the range [0,1], and `A` must be `m`-by-`n`-by-3.
- If `A` is of class `uint16` or `uint8`, `A` must be `m`-by-`n`-by-3.

Note A four-dimensional array that contains multiple RGB images returns 0, not 1.

Class Support

`A` can be of class `uint8`, `uint16`, or `double`.

See Also

`isbw`, `isgray`, `isind`

makelut

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`.

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`

mat2gray

Purpose Convert a matrix to a grayscale intensity image

Syntax

```
I = mat2gray(A,[amin amax])
I = mat2gray(A)
```

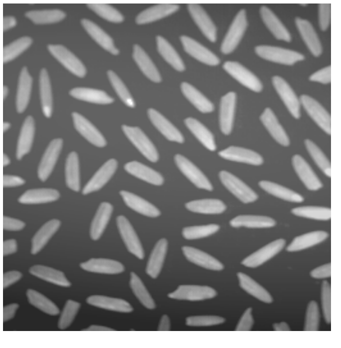
Description `I = mat2gray(A,[amin amax])` converts the matrix `A` to the intensity image `I`. The returned matrix `I` contains values in the range 0 (black) to 1.0 (full intensity or white). `amin` and `amax` are the values in `A` that correspond to 0 and 1.0 in `I`.

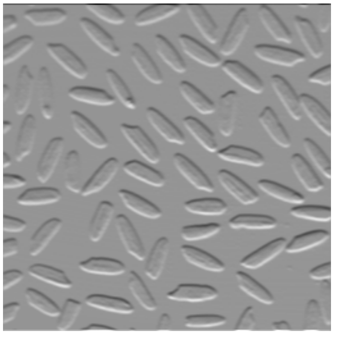
`I = mat2gray(A)` sets the values of `amin` and `amax` to the minimum and maximum values in `A`.

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`

Purpose Compute the mean of the elements of a matrix

Syntax

```
b = mean2(A)
```

Description `b = mean2(A)` computes the mean of the values in `A`.

Class Support `A` is an array of class `double` or of any integer class. `b` is a scalar of class `double`.

Algorithm `mean2` computes the mean of an array `A` using `mean(A(:))`.

See Also `std2`

`mean`, `std` in the MATLAB Function Reference

medfilt2

Purpose Perform two-dimensional median filtering

Syntax

```
B = medfilt2(A,[m n])
B = medfilt2(A)
B = medfilt2(A,'indexed',...)
```

Description 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.

`B = medfilt2(A,[m n])` 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. `medfilt2` 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.

`B = medfilt2(A)` performs median filtering of the matrix `A` using the default 3-by-3 neighborhood.

`B = medfilt2(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 class `uint8`, `uint16`, or `double` (unless the `'indexed'` syntax is used, in which case `A` cannot be of class `uint16`). The output image `B` is of the same class as `A`.

Remarks If the input image `A` is of class `uint8`, all of the output values are returned as `uint8` 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.

For example, suppose you call `medfilt2` using 2-by-2 neighborhoods, and the input image is a `uint8` array that includes this neighborhood.

```
1 5
4 8
```

`medfilt2` returns an output value of 4 for this neighborhood, although the true median is 4.5.

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

[1] Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 469-476.

montage

Purpose

Display multiple image frames as a rectangular montage

Syntax

```
montage(I)
montage(BW)
montage(X,map)
montage(RGB)
h = montage(...)
```

Description

`montage` 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.

`montage(I)` displays the `k` frames of the intensity image array `I`. `I` is m-by-n-by-1-by-k.

`montage(BW)` displays the `k` frames of the binary image array `BW`. `BW` is m-by-n-by-1-by-k.

`montage(X,map)` 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.

`montage(RGB)` displays the `k` frames of the truecolor image array `RGB`. `RGB` is m-by-n-by-3-by-k.

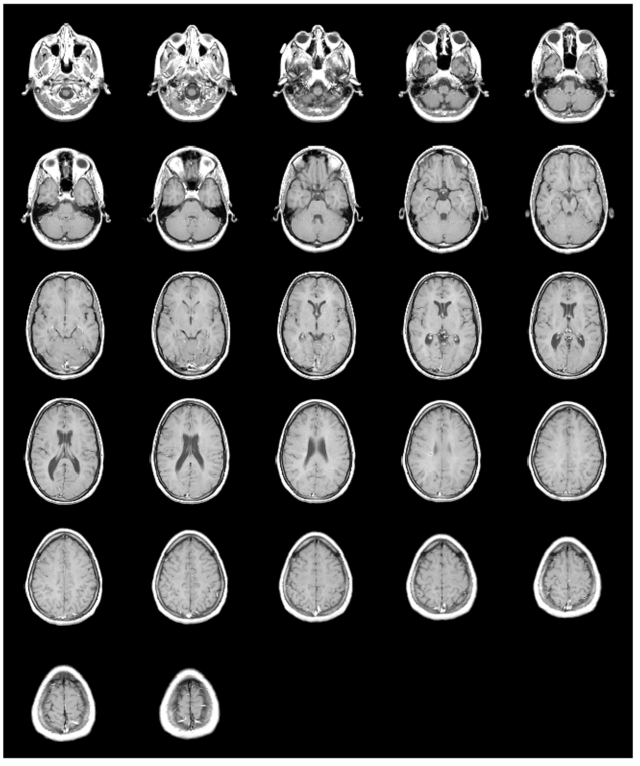
`h = montage(...)` returns the handle to the image object.

Class Support

The input image can be of class `uint8`, `uint16`, or `double`.

Example

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



See Also `immovie`

nlfilter

Purpose Perform general sliding-neighborhood operations

Syntax

```
B = nlfilter(A,[m n],fun)
B = nlfilter(A,[m n],fun,P1,P2,...)
B = nlfilter(A,'indexed',...)
```

Description `B = nlfilter(A,[m n],fun)` applies the function `fun` to each `m`-by-`n` sliding block of `A`. `fun` is a function that accepts an `m`-by-`n` matrix as input, and returns a scalar result.

```
c = fun(x)
```

`c` is the output value for the center pixel in the `m`-by-`n` block `x`. `nlfilter` calls `fun` for each pixel in `A`. `nlfilter` zero pads the `m`-by-`n` block at the edges, if necessary.

`B = nlfilter(A,[m n],fun,P1,P2,...)` passes the additional parameters `P1,P2,...,` to `fun`.

`B = nlfilter(A,'indexed',...)` processes `A` as an indexed image, padding with ones if `A` is of class `double` and zeros if `A` is of class `uint8`.

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`.

Remarks `nlfilter` can take a long time to process large images. In some cases, the `colfilt` function can perform the same operation much faster.

Example `fun` can be a `function_handle`, created using `@`. This example produces the same result as calling `medfilt2` with a 3-by-3 neighborhood.

```
B = nlfilter(A,[3 3],@myfun);
```

where `myfun` is an M-file containing

```
function scalar = myfun(x)
scalar = median(x(:));
```

fun can also be an inline object. The example above can be written as

```
fun = inline('median(x(:))');
```

See Also `blkproc`, `colfilt`

Purpose Convert NTSC values to RGB color space

Syntax

```
rgbmap = ntsc2rgb(yiqmap)
RGB = ntsc2rgb(YIQ)
```

Description `rgbmap = ntsc2rgb(yiqmap)` converts the `m`-by-3 NTSC (television) color values in `yiqmap` to RGB color space. If `yiqmap` is `m`-by-3 and contains the NTSC luminance *(Y)* and chrominance *(I* and *Q)* color components as columns, then `rgbmap` is an `m`-by-3 matrix that contains the red, green, and blue values equivalent to those colors. Both `rgbmap` and `yiqmap` 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.

`RGB = ntsc2rgb(YIQ)` converts the NTSC image `YIQ` to the equivalent truecolor image `RGB`.

`ntsc2rgb` computes the RGB values from the NTSC components using

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.621 \\ 1.000 & -0.272 & -0.647 \\ 1.000 & -1.106 & 1.703 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

Class Support The input image or colormap must be of class `double`. The output is of class `double`.

See Also `rgb2ntsc`, `rgb2ind`, `ind2rgb`, `ind2gray`

ordfilt2

Purpose

Perform two-dimensional order-statistic filtering

Syntax

```
B = ordfilt2(A,order,domain)
B = ordfilt2(A,order,domain,S)
B = ordfilt2(...,padopt)
```

Description

`B = ordfilt2(A,order,domain)` replaces each element in `A` by the `order`-th element in the sorted set of neighbors specified by the nonzero elements in `domain`.

`B = ordfilt2(A,order,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.

`B = ordfilt2(...,padopt)` controls how the matrix boundaries are padded. Set `padopt` to '`zeros`' (the default), or '`symmetric`'. If `padopt` is '`zeros`', `A` is padded with zeros at the boundaries. If `padopt` is '`symmetric`', `A` is symmetrically extended at the boundaries.

Class Support

`A` can be of class `uint8`, `uint16`, or `double`. The class of `B` is the same as the class of `A`, unless the additive offset form of `ordfilt2` is used, in which case the class of `B` 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, `B = ordfilt2(A,5,ones(3,3))` implements a 3-by-3 median filter; `B = ordfilt2(A,1,ones(3,3))` implements a 3-by-3 minimum filter; and `B = ordfilt2(A,9,ones(3,3))` implements a 3-by-3 maximum filter. `B = ordfilt2(A,1,[0 1 0; 1 0 1; 0 1 0])` replaces each element in `A` 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

[1] Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I*. Addison-Wesley, 1992.

phantom

Purpose

Generate a head phantom image

Syntax

```
P = phantom(def,n)
P = phantom(E,n)
[P,E] = phantom(...)
```

Description

`P = phantom(`*def*`,n)` generates an image of a head phantom that can be used to test the numerical accuracy of `radon` and `iradon` or other two-dimensional reconstruction algorithms. `P` is a grayscale intensity image that consists of one large ellipse (representing the brain) containing several smaller ellipses (representing features in the brain).

def is a string that specifies the type of head phantom to generate. Valid values are:

- `'Shepp-Logan'` – a test image used widely by researchers in tomography.
- `'Modified Shepp-Logan'` (default) – a variant of the Shepp-Logan phantom in which the contrast is improved for better visual perception.

`n` is a scalar that specifies the number of rows and columns in `P`. If you omit the argument, `n` defaults to 256.

`P = phantom(E,n)` generates a user-defined phantom, where each row of the matrix `E` specifies an ellipse in the image. `E` has six columns, with each column 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	Parameter	Meaning
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].

Example

```
P = phantom('Modified Shepp-Logan',200);
imshow(P)
```

Reference

[1] Jain, Anil K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. p. 439.

See Also

`radon`, `iradon`

Purpose

Display information about image pixels

Syntax

```
pixval on
pixval off
pixval
pixval(fig,option)
```

Purpose

`pixval on` turns on interactive display of information about image pixels in the current figure. `pixval` installs a black bar at the bottom of the figure, which displays the (x,y) 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, `pixval` also displays the Euclidean distance between the point you clicked on and the current cursor location. `pixval` 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.

`pixval off` 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.

`pixval` toggles interactive display on or off in the current figure.

`pixval(fig,option)` applies the `pixval` command to the figure specified by `fig`. `option` is string containing `'on'` or `'off'`.

See Also

`impixel`, `improfile`

qtdecomp

Purpose Perform quadtree decomposition

Syntax

```
S = qtdecomp(I)
S = qtdecomp(I,threshold)
S = qtdecomp(I,threshold,mindim)
S = qtdecomp(I,threshold,[mindim maxdim])

S = qtdecomp(I,fun)
S = qtdecomp(I,fun,P1,P2,...)
```

Description

`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.

`S = qtdecomp(I)` performs a quadtree decomposition on the intensity image `I`, and returns the quadtree structure in the sparse matrix `S`. If `S(k,m)` is nonzero, then `(k,m)` is the upper-left corner of a block in the decomposition, and the size of the block is given by `S(k,m)`. By default, `qtdecomp` splits a block unless all elements in the block are equal.

`S = qtdecomp(I,threshold)` splits a block if the maximum value of the block elements minus the minimum value of the block elements is greater than `threshold`. `threshold` is specified as a value between 0 and 1, even if `I` is of class `uint8` or `uint16`. If `I` is `uint8`, the threshold value you supply is multiplied by 255 to determine the actual threshold to use; if `I` is `uint16`, the threshold value you supply is multiplied by 65535.

`S = qtdecomp(I,threshold,mindim)` will not produce blocks smaller than `mindim`, even if the resulting blocks do not meet the threshold condition.

`S = qtdecomp(I,threshold,[mindim maxdim])` will not produce blocks smaller than `mindim` or larger than `maxdim`. Blocks larger than `maxdim` are split even if they meet the threshold condition. `maxdim/mindim` must be a power of 2.

`S = qtdecomp(I,fun)` uses the function `fun` to determine whether to split a block. `qtdecomp` calls `fun` with all the current blocks of size `m`-by-`m` stacked into an `m`-by-`m`-by-`k` array, where `k` is the number of `m`-by-`m` blocks. `fun` should return a k-element vector, containing only 1's and 0's, where 1 indicates that the 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 function_handle, created using @, or an inline object.

S = qtdecomp(I,fun,P1,P2,...) passes P1,P2,..., as additional arguments to fun.

Class Support

For the syntaxes that do not include a function, the input image can be of class uint8, uint16, 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.

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

qtgetblk

Purpose

Get block values in quadtree decomposition

Syntax

```
[vals,r,c] = qtgetblk(I,S,dim)
[vals,idx] = qtgetblk(I,S,dim)
```

Description

`[vals,r,c] = qtgetblk(I,S,dim)` returns in `vals` 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. `vals` 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.

`[vals,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 `uint8`, `uint16`, or `double`. `S` is of class `sparse`.

Remarks

The ordering of the blocks in `vals` matches the columnwise order of the blocks in `I`. For example, if `vals` is 4-by-4-by-2, `vals(:,:,1)` contains the values from the first 4-by-4 block in `I`, and `vals(:,:,2)` contains the values from the second 4-by-4 block.

Example

This example continues the `qtdecomp` example.

```
[vals,r,c] = qtgetblk(I,S,4)

vals(:,:,1) =

     1     1     1     1
     1     1     2     1
     1     1     1     1
     1     1     1     1
```

```
vals(:,:,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

```
J = qtsetblk(I,S,dim,vals)
```

Description

J = qtsetblk(I,S,dim,vals) replaces each dim-by-dim block in the quadtree decomposition of I with the corresponding dim-by-dim block in vals. S is the sparse matrix returned by qtdecomp; it contains the quadtree structure. vals is a dim-by-dim-by-k array, where k is the number of dim-by-dim blocks in the quadtree decomposition.

Class Support

I can be of class uint8, uint16, or double. S is of class sparse.

Remarks

The ordering of the blocks in vals must match the columnwise order of the blocks in I. For example, if vals is 4-by-4-by-2, vals(:,:,1) contains the values used to replace the first 4-by-4 block in I, and vals(:,:,2) contains the values for the second 4-by-4 block.

Example

This example continues the qtgetblock example.

```
newvals = cat(3,zeros(4),ones(4));
J = qtsetblk(I,S,4,newvals)

J =

     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

qtdecomp, qtgetblk

Purpose

Compute Radon transform

Syntax

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

Description

The `radon` function computes the Radon transform, which is the projection of the image intensity along a radial line oriented at a specified angle.

`R = radon(I,theta)` returns the Radon transform of the intensity image `I` for the angle `theta` degrees. If `theta` is a scalar, the result `R` is a column vector containing the Radon transform for `theta` degrees. If `theta` is a vector, then `R` is a matrix in which each column is the Radon transform for one of the angles in `theta`. If you omit `theta`, it defaults to 0:179.

`R = radon(I,theta,n)` returns a Radon transform with the projection computed at `n` points. `R` has `n` rows. If you do not specify `n`, the number of points at which the projection is computed is

```
2*ceil(norm(size(I)–floor((size(I)–1)/2)–1))+3
```

This number is sufficient to compute the projection at unit intervals, even along the diagonal.

`[R,xp] = radon(...)` returns a vector `xp` containing the radial coordinates corresponding to each row of `R`.

Class Support

`I` can be of class `double` or of any integer class. All other inputs and outputs are of class `double`.

Remarks

The radial coordinates returned in `xp` are the values along the x'-axis, which is oriented at `theta` degrees counterclockwise from the x-axis. The origin of both axes is the center pixel of the image, which is defined as

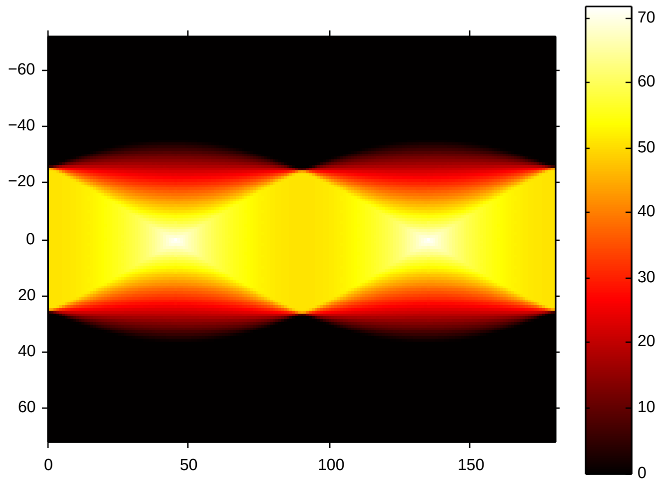
```
floor((size(I)+1)/2)
```

For example, in a 20-by-30 image, the center pixel is (10,15).

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,[],'notruesize'), colormap(hot), colorbar
```



See Also `iradon`, `phantom`

References

Bracewell, Ronald N. *Two-Dimensional Imaging*. Englewood Cliffs, NJ: Prentice Hall, 1995. pp. 505-537.

Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 42-45.

Purpose

Convert an RGB image or colormap to grayscale

Syntax

```
I = rgb2gray(RGB)
newmap = rgb2gray(map)
```

Description

`rgb2gray` converts RGB images to grayscale by eliminating the hue and saturation information while retaining the luminance.

`I = rgb2gray(RGB)` converts the truecolor image `RGB` to the grayscale intensity image `I`.

`newmap = rgb2gray(map)` returns a grayscale colormap equivalent to `map`.

Class Support

If the input is an RGB image, it can be of class `uint8`, `uint16`, or `double`; the output image `I` is of the same class as the input image. If the input is a colormap, the input and output colormaps are both of class `double`.

Algorithm

`rgb2gray` 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

`ind2gray`, `ntsc2rgb`, `rgb2ind`, `rgb2ntsc`

Purpose Convert RGB values to hue-saturation-value (HSV) color space

Syntax

```
hsvmap = rgb2hsv(rgbmap)
HSV = rgb2hsv(RGB)
```

Description `hsvmap = rgb2hsv(rgbmap)` converts the `m`-by-3 RGB values in `RGB` to HSV color space. `hsvmap` 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 `rgbmap` and `hsvmap` are of class `double` and contain values in the range 0 to 1.0.

`HSV = rgb2hsv(RGB)` converts the truecolor image `RGB` to the equivalent HSV image `HSV`.

Class Support If the input is an RGB image, it can be of class `uint8`, `uint16`, or `double`; the output image is of class `double`. If the input is a colormap, the input and output colormaps are both of class `double`.

Remarks `rgb2hsv` is a function in MATLAB.

See Also `hsv2rgb`, `rgbplot`

`colormap` in the MATLAB Function Reference

rgb2ind

Purpose

Convert an RGB image to an indexed image

Syntax

```
[X,map] = rgb2ind(RGB,tol)
[X,map] = rgb2ind(RGB,n)
X = rgb2ind(RGB,map)
[...] = rgb2ind(...,dither_option)
```

Description

`rgb2ind` converts RGB images to indexed images using one of three different methods: uniform quantization, minimum variance quantization, and colormap mapping. For all of these methods, `rgb2ind` also dithers the image unless you specify `'nodither'` for `dither_option`.

`[X,map] = rgb2ind(RGB,tol)` converts the RGB image to an indexed image `X` using uniform quantization. `map` contains at most `(floor(1/tol)+1)^3` colors. `tol` must be between 0 and 1.0.

`[X,map] = rgb2ind(RGB,n)` converts the RGB image to an indexed image `X` using minimum variance quantization. `map` contains at most `n` colors.

`X = rgb2ind(RGB,map)` converts the RGB image to an indexed image `X` with colormap `map` by matching colors in `RGB` with the nearest color in the colormap `map`.

`[...] = rgb2ind(...,`*`dither_option`*`)` enables or disables dithering. *`dither_option`* is a string that can have one of these values:

- `'dither'` (default) dithers, if necessary, to achieve better color resolution at the expense of spatial resolution.
- `'nodither'` 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 `uint8`, `uint16`, or `double`. The output image is of class `uint8` if the length of `map` is less than or equal to 256. It is `double` 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 = (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

cmunique, dither, imapprox, ind2rgb, rgb2gray

rgb2ntsc

Purpose Convert RGB values to NTSC color space

Syntax

```
yiqmap = rgb2ntsc(rgbmap)
YIQ = rgb2ntsc(RGB)
```

Description `yiqmap = rgb2ntsc(rgbmap)` converts the `m`-by-3 RGB values in `rbgmap` to NTSC color space. `yiqmap` 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.

`YIQ = rgb2ntsc(RGB)` converts the truecolor image `RGB` to the equivalent NTSC image `YIQ`.

`rgb2ntsc` 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 `uint8`, `uint16`, or `double`; the output image is of class `double`. If the input is a colormap, the input and output colormaps are both of class `double`.

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 `ntsc2rgb`, `rgb2ind`, `ind2rgb`, `ind2gray`

Purpose Convert RGB values to YCbCr color space

Syntax

```
ycbcrmap = rgb2ycbcr(rgbmap)
YCBCR = rgb2ycbcr(RGB)
```

Description `ycbcrmap = rgb2ycbcr(rgbmap)` converts the RGB values in `rbgmap` to the YCbCr color space. `ycbcrmap` 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.

`YCBCR = rgb2ycbcr(RGB)` converts the truecolor image `RGB` to the equivalent image in the YCbCr color space.

Class Support If the input is an RGB image, it can be of class `uint8`, `uint16`, or `double`; 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 `double`.

See Also `ntsc2rgb`, `rgb2ntsc`, `ycbcr2rgb`

rgbplot

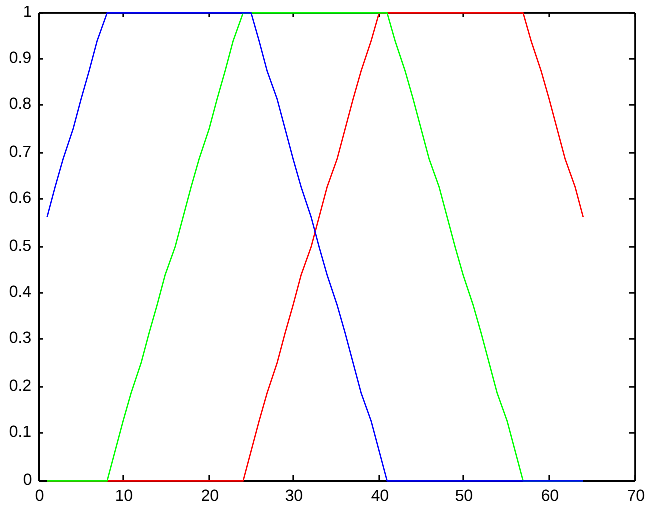
Purpose Plot colormap

Syntax `rgbplot(map)`

Description `rgbplot(map)` plots the three columns of `map`, where `map` is an `m`-by-3 colormap matrix. `rgbplot` draws the first column in red, the second in green, and the third in blue.

Example

```
rgbplot(jet)
```



See Also `colormap` in the MATLAB Function Reference

Purpose Select region of interest, based on color

Syntax

```
BW = roicolor(A,low,high)
BW = roicolor(A,v)
```

Description `roicolor` 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 `roifilt2`.)

`BW = roicolor(A,low,high)` returns a region of interest selected as those pixels that lie within the colormap range `[low high]`.

```
BW = (A >= low) & (A <= high)
```

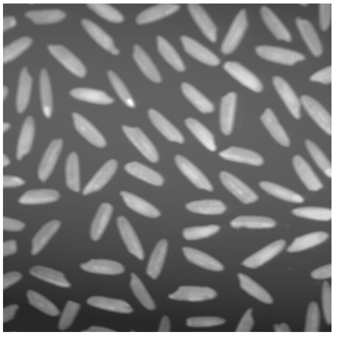
`BW` is a binary image with 0's outside the region of interest and 1's inside.

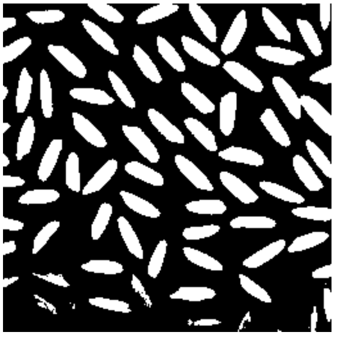
`BW = roicolor(A,v)` returns a region of interest selected as those pixels in `A` that match the values in vector `v`. `BW` is a binary image with 1's where the values of `A` match the values of `v`.

Class Support The input array `A` can be of class `double` or of any integer class. The output array `BW` is of class `uint8`.

Example

```
I = imread('rice.tif');
BW = roicolor(I,128,255);
imshow(I);
figure, imshow(BW)
```





See Also `roifilt2`, `roipoly`

roifill

Purpose Smoothly interpolate within an arbitrary image region

Syntax

```
J = roifill(I,c,r)
J = roifill(I)

J = roifill(I,BW)
[J,BW] = roifill(...)

J = roifill(x,y,I,xi,yi)
[x,y,J,BW,xi,yi] = roifill(...)
```

Description

`roifill` 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. `roifill` can be used, for example, to "erase" small objects in an image.

`J = roifill(I,c,r)` fills in the polygon specified by `c` and `r`, 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))`.

`J = roifill(I)` displays the image `I` on the screen and lets you specify the polygon using the mouse. If you omit `I`, `roifill` 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.

`J = roifill(I,BW)` uses `BW` (a binary image the same size as `I`) as a mask. `roifill` fills in the regions in `I` corresponding to the nonzero pixels in `BW`. If there are multiple regions, `roifill` performs the interpolation on each region independently.

`[J,BW] = roifill(...)` returns the binary mask used to determine which pixels in `I` get filled. `BW` is a binary image the same size as `I` with 1's for pixels corresponding to the interpolated region of `I` and 0's elsewhere.

`J = roifill(x,y,I,xi,yi)` uses the vectors `x` and `y` to establish a nondefault spatial coordinate system. `xi` and `yi` are equal-length vectors that specify polygon vertices as locations in this coordinate system.

`[x,y,J,BW,xi,yi] = roifill(...)` 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 `roifill(I,BW)` form is used.

If `roifill` is called with no output arguments, the resulting image is displayed in a new figure.

Class Support

The input image `I` can of class `uint8`, `uint16`, or `double`. The binary mask `BW` can be of class `uint8` or `double`. The output image `J` is of the same class as `I`. All other inputs and outputs are of class `double`.

Example

```
I = imread('eight.tif');
c = [222 272 300 270 221 194];
r = [21 21 75 121 121 75];
J = roifill(I,c,r);
imshow(I)
figure, imshow(J)
```

See Also

`roifilt2`, `roipoly`

roifilt2

Purpose

Filter a region of interest

Syntax

```
J = roifilt2(h,I,BW)
J = roifilt2(I,BW,fun)
J = roifilt2(I,BW,fun,P1,P2,...)
```

Description

`J = roifilt2(h,I,BW)` filters the data in `I` with the two-dimensional linear filter `h`. `BW` is a binary image the same size as `I` that is used as a mask for filtering. `roifilt2` returns an image that consists of filtered values for pixels in locations where `BW` contains 1's, and unfiltered values for pixels in locations where `BW` contains 0's. For this syntax, `roifilt2` calls `filter2` to implement the filter.

`J = roifilt2(I,BW,fun)` processes the data in `I` using the function `fun`. The result `J` contains computed values for pixels in locations where `BW` contains 1's, and the actual values in `I` for pixels in locations where `BW` contains 0's.

`fun` can be a `function_handle`, created using `@`, or an inline object. `fun` should take a matrix as a single argument and return a matrix of the same size.

```
y = fun(x)
```

`J = roifilt2(I,BW,fun,P1,P2,...)` passes the additional parameters `P1,P2,...,` to `fun`.

Class Support

For the syntax that includes a filter `h`, the input image `I` can be of class `uint8`, `uint16`, or `double`, and the output array `J` is of class `double`. For the syntax that includes a function, `I` can be of any class supported by `fun`, and the class of `J` depends on the class of the output from `fun`.

Example

This example continues the `roipoly` example.

```
I = imread('eight.tif');
c = [222 272 300 270 221 194];
r = [21 21 75 121 121 75];
BW = roipoly(I,c,r);
h = fspecial('unsharp');
J = roifilt2(h,I,BW);
imshow(J), figure, imshow(J)
```

See Also filter2, roipoly

roipoly

Purpose

Select a polygonal region of interest

Syntax

```
BW = roipoly(I,c,r)
BW = roipoly(I)

BW = roipoly(x,y,I,xi,yi)
[BW,xi,yi] = roipoly(...)
[x,y,BW,xi,yi] = roipoly(...)
```

Description

Use `roipoly` to select a polygonal region of interest within an image. `roipoly` returns a binary image that you can use as a mask for masked filtering.

`BW = roipoly(I,c,r)` returns the region of interest selected by the polygon described by vectors `c` and `r`. `BW` is a binary image the same size as `I` with 0's outside the region of interest and 1's inside.

`BW = roipoly(I)` displays the image `I` on the screen and lets you specify the polygon using the mouse. If you omit `I`, `roipoly` 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.

`BW = roipoly(x,y,I,xi,yi)` uses the vectors `x` and `y` to establish a nondefault spatial coordinate system. `xi` and `yi` are equal-length vectors that specify polygon vertices as locations in this coordinate system.

`[BW,xi,yi] = roipoly(...)` returns the polygon coordinates in `xi` and `yi`. Note that `roipoly` 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 `xi` and `yi` is equal to the number of points specified. If the points specified do not describe a closed polygon, `roipoly` adds a final point having the same coordinates as the first point. (In this case the length of `xi` and `yi` is one greater than the number of points specified.)

`[x,y,BW,xi,yi] = roipoly(...)` returns the `XData` and `YData` in `x` and `y`; the mask image in `BW`; and the polygon coordinates in `xi` and `yi`.

If `roipoly` is called with no output arguments, the resulting image is displayed in a new figure.

Class Support

The input image `I` can be of class `uint8`, `uint16`, or `double`. The output image `BW` is of class `uint8`. All other inputs and outputs are of class `double`.

Remarks

For any of the `roipoly` 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 = roipoly(100,200,c,r);
```

If you specify `m` and `n` with an interactive form of `roipoly`, an `m`-by-`n` black image is displayed, and you use the mouse to specify a polygon within this image.

Example

```
I = imread('eight.tif');
c = [222 272 300 270 221 194];
r = [21 21 75 121 121 75];
BW = roipoly(I,c,r);
imshow(I)
figure, imshow(BW)
```

See Also

`roifilt2`, `roicolor`, `roifill`

std2

Purpose Compute the standard deviation of the elements of a matrix

Syntax

```
b = std2(A)
```

Description `b = std2(A)` computes the standard deviation of the values in `A`.

Class Support `A` is an array of class `double` or of any integer class. `b` is a scalar of class `double`.

Algorithm `std2` computes the standard deviation of the array `A` using `std(A(:))`.

See Also `corr2`, `mean2`

`std`, `mean` in the MATLAB Function Reference

Purpose Display multiple images in the same figure

Syntax

```
subimage(X,map)
subimage(I)
subimage(BW)
subimage(RGB)
subimage(x,y,...)
h = subimage(...)
```

Description You can use `subimage` in conjunction with `subplot` to create figures with multiple images, even if the images have different colormaps. `subimage` works by converting images to truecolor for display purposes, thus avoiding colormap conflicts.

`subimage(X,map)` displays the indexed image `X` with colormap `map` in the current axes.

`subimage(I)` displays the intensity image `I` in the current axes.

`subimage(BW)` displays the binary image `BW` in the current axes.

`subimage(RGB)` displays the truecolor image `RGB` in the current axes.

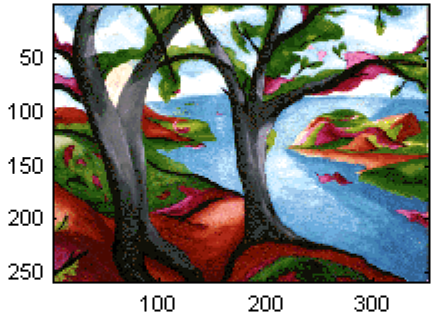
`subimage(x,y...)` displays an image using a nondefault spatial coordinate system.

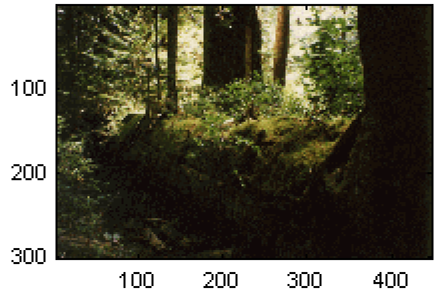
`h = subimage(...)` returns a handle to an image object.

Class Support The input image can be of class `uint8`, `uint16`, or `double`.

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 MATLAB Function Reference

Purpose

Adjust display size of an image

Syntax

```
truesize(fig,[mrows mcols])
truesize(fig)
```

Description

`truesize(fig,[mrows ncols])` adjusts the display size of an image. `fig` is a figure containing a single image or a single image with a colorbar. `[mrows ncols]` is a 1-by-2 vector that specifies the requested screen area in pixels that the image should occupy.

`truesize(fig)` uses the image height and width for `[mrows ncols]`. This results in the display having one screen pixel for each image pixel.

If you omit the figure argument, `truesize` works on the current figure.

Remarks

If the`'TruesizeWarning'` toolbox preference is `'on'`, `truesize` 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`.

See Also

`imshow`, `iptsetpref`, `iptgetpref`

uint8

Purpose Convert data to unsigned 8-bit integers

Syntax

```
B = uint8(A)
```

Description

`B = uint8(A)` creates the unsigned 8-bit integer array `B` from the array `A`. If `A` is a `uint8` array, `B` is identical to `A`.

The elements of a `uint8` array can range from 0 to 255. Values outside this range are mapped to 0 or 255. If `A` is already an unsigned 8-bit integer array, `uint8` has no effect.

The fractional part of each value in `A` is discarded on conversion. This means, for example, that `uint8(102.99)` is 102, not 103. Therefore, it is often a good idea to round off the values in `A` before converting to `uint8`. For example,

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

MATLAB 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

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

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

Remarks `uint8` is a MATLAB built-in function.

Example

```
a = [1 3 5];
b = uint8(a);
whos
Name     Size          Bytes  Class
  a      1x3             24  doublearray
  b      1x3              3  uint8 array
```

See Also

`double`, `im2double`, `im2uint8`

uint16

Purpose Convert data to unsigned 16-bit integers

Syntax

```
I = uint16(X)
```

Description

`I = uint16(X)` converts the vector `X` into an unsigned 16-bit integer. `X` can be any numeric object (such as a `double`). The elements of a `uint16` range from 0 to 65535. Values outside this range are mapped to 0 or 65535. If `X` is already an unsigned 16-bit integer array, `uint16` has no effect.

The `uint16` class is primarily meant to be used to store integer values. Hence most operations that manipulate arrays without changing their elements are defined, for example, the functions `reshape` and `size`, the relational operators, subscripted assignment, and subscripted reference. While most MATLAB arithmetic operations cannot be performed on `uint16` data, the following operations are supported: `sum`, `conv2`, `convn`, `fft2`, and `fftn`. In these cases the output will always be `double`. If you attempt to perform an unsupported operation you will receive an error such as `Function '+' not defined for variables of class 'uint16'`.

You can define your own methods for `uint16` (as you can for any object) by placing the appropriately named method in an `@uint16` directory within a directory on your path.

Other operations and functions supported for `uint16` data include:

- Displaying data values
- Indexing into arrays using standard MATLAB subscripting
- Logical operators
- Saving to and loading from MAT-files
- The functions `cat`, `permute`, `all`, and `any`

Most functions in the Image Processing Toolbox accept `uint16` input. See the individual reference entries for information about `uint16` support.

Class Support The input image can be of class `uint8` or `double`.

Remarks `uint16` is a MATLAB built-in function.

Example

```
a = [1 3 5];
b = uint16(a);
whos
Name     Size          Bytes  Class
  a      1x3              24  double array
  b      1x3               6  uint16 array
```

See Also

`double`, `uint8`, `uint32`, `int8`, `int16`, `int32`

warp

Purpose Display an image as a texture-mapped surface

Syntax

```
warp(X,map)
warp(I,n)
warp(BW)
warp(RGB)
warp(z,...)
warp(x,y,z,...)
h = warp(...)
```

Description `warp(X,map)` displays the indexed image `X` with colormap `map` as a texture map on a simple rectangular surface.

`warp(I,n)` displays the intensity image `I` with gray scale colormap of length `n` as a texture map on a simple rectangular surface.

`warp(BW)` displays the binary image `BW` as a texture map on a simple rectangular surface.

`warp(RGB)` displays the RGB image in the array `RGB` as a texture map on a simple rectangular surface.

`warp(z,...)` displays the image on the surface `z`.

`warp(x,y,z...)` displays the image on the surface `(x,y,z)`.

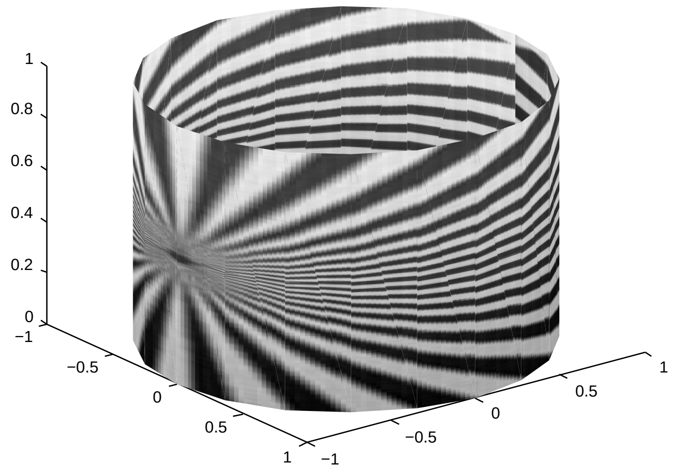
`h = warp(...)` returns a handle to a texture mapped surface.

Class Support The input image can be of class `uint8`, `uint16`, or `double`.

Remarks Texture-mapped surfaces generally render more slowly than images.

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 MATLAB Function Reference

wiener2

Purpose Perform two-dimensional adaptive noise-removal filtering

Syntax

```
J = wiener2(I,[m n],noise)
[J,noise] = wiener2(I,[m n])
```

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.

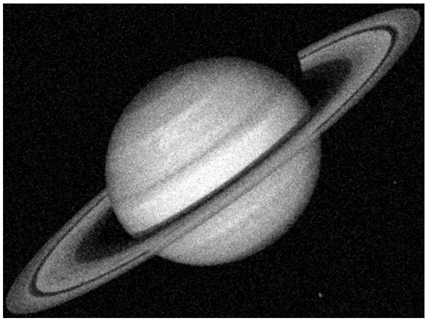
`J = wiener2(I,[m n],noise)` filters the image `I` using pixel-wise adaptive Wiener filtering, using neighborhoods of size `m`-by-`n` to estimate the local image mean and standard deviation. If you omit the `[m n]` argument, `m` and `n` default to 3. The additive noise (Gaussian white noise) power is assumed to be `noise`.

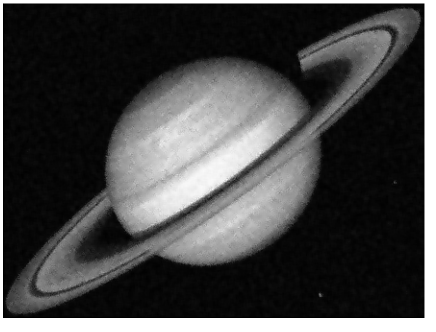
`[J,noise] = wiener2(I,[m n])` also estimates the additive noise power before doing the filtering. `wiener2` returns this estimate in `noise`.

Class Support The input image `I` can be of class `uint8`, `uint16`, or `double`. The output image `J` is of the same class as `I`.

Example Degrade and then restore an intensity image using adaptive Wiener filtering.

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





Algorithm `wiener2` estimates the local mean and variance around each pixel

$$\mu = \frac{1}{NM} \sum_{n_1, n_2 \in \eta} a(n_1, n_2)$$

$$\sigma^2 = \frac{1}{NM} \sum_{n_1, n_2 \in \eta} a^2(n_1, n_2) - \mu^2$$

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

$$b(n_1, n_2) = \mu + \frac{\sigma^2 - \nu^2}{\sigma^2}(a(n_1, n_2) - \mu)$$

where ν^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

[1] Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 536-540.

ycbcr2rgb

Purpose

Convert YCbCr values to RGB color space

Syntax

```
rgbmap = ycbcr2rgb(ycbcrmap)
RGB = ycbcr2rgb(YCBCR)
```

Description

`rgbmap = ycbcr2rgb(ycbcrmap)` converts the YCbCr values in the colormap `ycbcrmap` to the RGB color space. If `ycbcrmap` is `m`-by-3 and contains the YCbCr luminance (*Y*) and chrominance (*Cb* and *Cr*) color components as its columns, then `rgbmap` is returned as an `m`-by-3 matrix that contains the red, green, and blue values equivalent to those colors.

`RGB = ycbcr2rgb(YCBCR)` converts the YCbCr image `YCBCR` to the equivalent truecolor image `RGB`.

Class Support

If the input is a YCbCr image, it can be of class `uint8`, `uint16`, or `double`; 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 `double`.

See Also

`ntsc2rgb`, `rgb2ntsc`, `rgb2ycbcr`

Purpose Zoom in and out of an image

Syntax

```
zoom on
zoom off
zoom out
zoom reset
zoom
zoom xon
zoom yon
zoom(factor)
zoom(fig,option)
```

Description

`zoom on` 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 on or out from the point:

- 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.

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.

Double-clicking within an axes returns the axes to its initial zoom setting.

`zoom off` turns zoom off in the current figure.

`zoom out` returns the plot to its initial zoom setting.

`zoom reset` remembers the current zoom setting as the initial zoom setting. Later calls to zoom out, or double-clicks when interactive zoom mode is enabled, return to this zoom level.

`zoom` toggles the interactive zoom status.

`zoom xon` and `zoom yon` set zoom on for the *x*- and *y*-axis, respectively.

`zoom(factor)` 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/`factor`.

`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`

A

Working with Function Functions

The Image Processing Toolbox contains several functions called *function functions*, so named because they enable you to supply one of your own functions as an input argument. For example, `blkproc` enables you to input your own block processing function, and `qtdecomp` enables you to input your own algorithm for defining a criterion of homogeneity. This section shows you the different ways in which you can input your own function to a function function.

Note As you may know, MATLAB has a directory named `funfun` containing function functions. However, the function functions of the Image Processing Toolbox are not included in this directory. For a discussion of the MATLAB functions in `funfun`, see the section in the MATLAB documentation entitled "Function Functions."

There are three different methods for passing your own function to a function function:

- Pass in a function handle to an M-file function
- Pass in an inline function
- Pass in a string containing an expression

This appendix contains three examples — one to demonstrate each method.

Note Function handles are a new class in MATLAB 6.0. One advantage to using them is that you can call a function function with a function handle to a private function or subfunction. In previous versions of MATLAB, your function had to be on the MATLAB path. For more information, see `function_handle` in the MATLAB Function Reference.

All three examples use the function function `blkproc`. The following `blkproc` syntax variation is used.

```
B = BLKPROC(A,[m n],fun,P1,P2,...)
```

This syntax takes as its arguments an image `A`, a block size `[m n]` used to divide the image, and a function `fun`, which is used to process each block. This syntax also takes any number of parameters (`P1`, `P2`, etc.) that may be needed by `fun`.

All three examples use the same simple function to alter the brightness of a grayscale image.

$$f(x) = x \times P1$$

x represents a block of size `[m n]`, and $P1$ can take any value. Note that this function was chosen because it works well for illustrative purposes; if you really want to brighten an image, you should use the `imadjust` function.

Passing an M-File Function to a Function Function

Create an M-file containing your block-processing function. Using the example function above, your M-file might contain the following lines.

```
function y = myblkfun(x,P1)
% For an input block of x, divide the pixel values by P1.
% Temporarily make x double so you can perform arithmetic on it.
y = uint8(double(x)*P1);
```

To use your M-file with `blkproc`, create a function handle (`function_handle`) to it, and pass in the handle and any desired value for `P1`. For example,

```
I = imread('cameraman.tif');
f = @myblkfun; % Create a function handle.
I2 = blkproc(I, [10 10], f, 2);
imshow(I);
figure, imshow(I2);
```

Figure A-1: Original Image (left) and Brightened Image (right)

Passing an Inline Object to a Function Function

Create an inline object at the MATLAB prompt. Pass the inline object and any desired value for `P1` to `blkproc`. For example,

```
myblkfun = inline('uint8((double(x)*2))',1);
I = imread('cameraman.tif');
I2 = blkproc(I,[10 10], myblkfun, 2);
```

The results are the same as those shown in Figure A-1. For more information about inline functions, see the online MATLAB reference page for `inline`.

Passing a String to a Function Function

You can also pass an expression to a function function. Just set the `fun` parameter to a string containing your expression. For example,

```
I = imread('cameraman.tif');
I2 = blkproc(I,[10 10],'uint8((double(x)*2))');
```

The results are the same as those shown in Figure A-1.

Index

Numerics

A

B

J

L

S

U

V

W

X

Y

Z