

# Image Processing Toolbox

For Use with **MATLAB®**

- Computation
- Visualization
- Programming

User's Guide

*Version 5*



## How to Contact The MathWorks



www.mathworks.com  
comp.soft-sys.matlab  
www.mathworks.com/contact\_TS.html

Web  
Newsgroup  
Technical Support



suggest@mathworks.com  
bugs@mathworks.com  
doc@mathworks.com  
service@mathworks.com  
info@mathworks.com

Product enhancement suggestions  
Bug reports  
Documentation error reports  
Order status, license renewals, passcodes  
Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



The MathWorks, Inc.  
3 Apple Hill Drive  
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

### *Image Processing Toolbox User's Guide*

© COPYRIGHT 1993–2006 by The MathWorks, Inc.

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

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

### **Trademarks**

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks, and SimBiology, SimEvents, and SimHydraulics are trademarks of The MathWorks, Inc.

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

### **Patents**

The MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

### **Revision History**

August 1993	First printing	Version 1
May 1997	Second printing	Version 2
April 2001	Third printing	Revised for Version 3.0
June 2001	Online only	Revised for Version 3.1 (Release 12.1)
July 2002	Online only	Revised for Version 3.2 (Release 13)
May 2003	Fourth printing	Revised for Version 4.0 (Release 13.0.1)
September 2003	Online only	Revised for Version 4.1 (Release 13.SP1)
June 2004	Online only	Revised for Version 4.2 (Release 14)
August 2004	Online only	Revised for Version 5.0 (Release 14+)
October 2004	Fifth printing	Revised for Version 5.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 5.0.2 (Release 14SP2)
September 2005	Online only	Revised for Version 5.1 (Release 14SP3)
March 2006	Online only	Revised for Version 5.2 (Release 2006a)
September 2006	Online only	Revised for Version 5.3 (Release 2006b)



## Getting Started

**1**

<b>What Is the Image Processing Toolbox?</b> .....	<b>1-2</b>
Configuration Notes .....	<b>1-3</b>
Related Products .....	<b>1-3</b>
Compilability .....	<b>1-3</b>
<b>Example 1 — Some Basic Concepts</b> .....	<b>1-4</b>
Step 1: Read and Display an Image .....	<b>1-4</b>
Step 2: Check How the Image Appears in the Workspace ..	<b>1-5</b>
Step 3: Improve Image Contrast .....	<b>1-6</b>
Step 4: Write the Image to a Disk File .....	<b>1-8</b>
Step 5: Check the Contents of the Newly Written File ....	<b>1-8</b>
<b>Example 2 — Advanced Topics</b> .....	<b>1-10</b>
Step 1: Read and Display an Image .....	<b>1-11</b>
Step 2: Estimate the Value of Background Pixels .....	<b>1-11</b>
Step 3: View the Background Approximation as a Surface .....	<b>1-12</b>
Step 4: Create an Image with a Uniform Background ....	<b>1-14</b>
Step 5: Adjust the Contrast in the Processed Image ....	<b>1-14</b>
Step 6: Create a Binary Version of the Image .....	<b>1-15</b>
Step 7: Determine the Number of Objects in the Image ...	<b>1-16</b>
Step 8: Examine the Label Matrix .....	<b>1-17</b>
Step 9: Display the Label Matrix as a Pseudocolor Indexed Image .....	<b>1-18</b>
Step 10: Measure Object Properties in the Image .....	<b>1-19</b>
Step 11: Compute Statistical Properties of Objects in the Image .....	<b>1-21</b>
<b>Getting Help</b> .....	<b>1-23</b>
Online Help .....	<b>1-23</b>
Image Processing Demos .....	<b>1-23</b>
MATLAB Newsgroup .....	<b>1-24</b>
<b>Image Credits</b> .....	<b>1-25</b>

## 2

<b>Images in MATLAB and the Image Processing</b>	
<b>Toolbox</b> .....	2-2
Coordinate Systems .....	2-2
<b>Image Types in the Toolbox</b> .....	2-7
Binary Images .....	2-8
Indexed Images .....	2-8
Grayscale Images .....	2-10
Truecolor Images .....	2-11
<b>Converting Between Image Types</b> .....	2-15
Color Space Conversions .....	2-16
<b>Converting Between Image Classes</b> .....	2-17
Losing Information in Conversions .....	2-17
Converting Indexed Images .....	2-17
<b>Working with Image Sequences</b> .....	2-19
Example: Processing Image Sequences .....	2-22
Multi-Frame Image Arrays .....	2-23
<b>Image Arithmetic</b> .....	2-25
Image Arithmetic Saturation Rules .....	2-25
Nesting Calls to Image Arithmetic Functions .....	2-26

## Reading and Writing Image Data

## 3

<b>Getting Information About a Graphics File</b> .....	3-2
<b>Reading Image Data</b> .....	3-3
Reading Multiple Images from a Graphics File .....	3-4
<b>Writing Image Data</b> .....	3-5

Specifying Additional Format-Specific Parameters .....	3-5
Reading and Writing Binary Images in 1-Bit Format .....	3-6
Determining the Storage Class of the Output File .....	3-7
<b>Converting Graphics File Formats .....</b>	<b>3-8</b>
<b>Reading and Writing Data in Medical File Formats ...</b>	<b>3-9</b>
Reading Metadata from a DICOM File .....	3-9
Reading Image Data from a DICOM File .....	3-10
Writing Image Data or Metadata to a DICOM File .....	3-11
Using the Mayo Analyze 7.5 Format .....	3-16
Using the Interfile Format .....	3-17

## Displaying and Exploring Images

# 4

<b>Overview .....</b>	<b>4-3</b>
Understanding Handle Graphics Object Property Settings .....	4-4
<b>Using imshow to Display Images .....</b>	<b>4-5</b>
Specifying the Initial Image Magnification .....	4-6
Controlling the Appearance of the Figure .....	4-7
<b>Using the Image Tool to Explore Images .....</b>	<b>4-9</b>
Opening the Image Tool .....	4-11
Specifying the Initial Image Magnification .....	4-12
Specifying the Colormap .....	4-13
Importing Image Data from the Workspace .....	4-15
Exporting Image Data to the Workspace .....	4-16
Closing the Image Tool .....	4-17
Printing the Image in the Image Tool .....	4-17
<b>Using Image Tool Navigation Aids .....</b>	<b>4-18</b>
Overview Navigation .....	4-18
Panning the Image Displayed in the Image Tool .....	4-21
Zooming In and Out on an Image .....	4-22
Specifying the Magnification of the Image .....	4-22

<b>Getting Information about the Pixels in an Image</b> . . . .	<b>4-24</b>
Determining the Value of Individual Pixels . . . . .	4-24
Getting the Display Range of an Image . . . . .	4-26
Viewing Pixel Values with the Pixel Region Tool . . . . .	4-27
<b>Measuring Features in an Image</b> . . . . .	<b>4-31</b>
Using the Distance Tool . . . . .	4-31
Exporting Endpoint and Distance Data . . . . .	4-32
Customizing the Appearance of the Distance Tool . . . . .	4-33
<b>Getting Information About an Image</b> . . . . .	<b>4-34</b>
<b>Adjusting the Contrast and Brightness of an Image</b> . . .	<b>4-36</b>
Using the Adjust Contrast Tool . . . . .	4-38
Example: Adjusting Contrast and Brightness . . . . .	4-40
Using the Window/Level Tool . . . . .	4-43
Understanding Contrast Adjustment . . . . .	4-45
<b>Viewing Multiple Images</b> . . . . .	<b>4-47</b>
Displaying Each Image in a Separate Figure . . . . .	4-47
Displaying Multiple Images in the Same Figure . . . . .	4-48
<b>Displaying Different Image Types</b> . . . . .	<b>4-51</b>
Displaying Indexed Images . . . . .	4-51
Displaying Grayscale Images . . . . .	4-52
Displaying Binary Images . . . . .	4-54
Displaying Truecolor Images . . . . .	4-56
<b>Special Display Techniques</b> . . . . .	<b>4-58</b>
Adding a Colorbar . . . . .	4-58
Displaying All Frames of a Multiframe Image at Once . . .	4-60
Converting a Multiframe Image to a Movie . . . . .	4-61
Texture Mapping . . . . .	4-62
<b>Printing Images</b> . . . . .	<b>4-64</b>
Printing and Handle Graphics Object Properties . . . . .	4-64
<b>Setting Toolbox Display Preferences</b> . . . . .	<b>4-66</b>
Toolbox Preferences . . . . .	4-66
Retrieving the Values of Toolbox Preferences . . . . .	4-67



Setting the Values of Toolbox Preferences .....	4-68
---	------

## Building GUIs with Modular Tools

### 5

<b>Overview</b> .....	5-2
<b>Using Modular Tools</b> .....	5-6
Displaying the Target Image .....	5-7
Specifying the Target Image .....	5-8
Specifying the Parent of a Modular Tool .....	5-12
Positioning the Modular Tools in a GUI .....	5-15
Example: Building a Pixel Information GUI .....	5-17
Adding Navigation Aids to a GUI .....	5-19
Making Connections for Interactivity .....	5-25
<b>Creating Your Own Modular Tools</b> .....	5-31

## Spatial Transformations

### 6

<b>Interpolation</b> .....	6-3
Interpolation Methods .....	6-3
Interpolation and Image Types .....	6-4
<b>Resizing an Image</b> .....	6-5
Specifying the Size of the Output Image .....	6-5
Specifying the Interpolation Method .....	6-6
Using Filters to Prevent Aliasing .....	6-7
<b>Rotating an Image</b> .....	6-8
Specifying the Size of the Output Image .....	6-8
Specifying the Interpolation Method .....	6-8
<b>Cropping an Image</b> .....	6-10

<b>Performing General 2-D Spatial Transformations</b> .....	<b>6-11</b>
Example: Performing a Translation .....	<b>6-12</b>
Defining the Transformation Data .....	<b>6-17</b>
Creating TFORM Structures .....	<b>6-19</b>
Performing the Spatial Transformation .....	<b>6-20</b>
<b>Performing N-Dimensional Spatial Transformations</b> ..	<b>6-23</b>
<b>Example: Performing Image Registration</b> .....	<b>6-25</b>
Step 1: Read in Base and Unregistered Images .....	<b>6-25</b>
Step 2: Display the Unregistered Image .....	<b>6-25</b>
Step 3: Create a TFORM Structure .....	<b>6-26</b>
Step 4: Transform the Unregistered Image .....	<b>6-26</b>
Step 5: Overlay Registered Image Over Base Image .....	<b>6-27</b>
Step 6: Using XData and YData Input Parameters .....	<b>6-28</b>
Step 7: Using XData and YData Output Values .....	<b>6-29</b>

## Image Registration

# 7

<b>Registering an Image</b> .....	<b>7-2</b>
Point Mapping .....	<b>7-2</b>
Example: Registering to a Digital Orthophoto .....	<b>7-4</b>
<b>Types of Supported Transformations</b> .....	<b>7-11</b>
<b>Selecting Control Points</b> .....	<b>7-13</b>
Using the Control Point Selection Tool .....	<b>7-13</b>
Starting the Control Point Selection Tool .....	<b>7-15</b>
Viewing the Images .....	<b>7-16</b>
Specifying Matching Control Point Pairs .....	<b>7-21</b>
Saving Control Points .....	<b>7-28</b>
<b>Using Correlation to Improve Control Points</b> .....	<b>7-30</b>

# Linear Filtering and Filter Design

## 8

<b>Linear Filtering</b> .....	8-2
Convolution .....	8-2
Correlation .....	8-4
Filtering Using imfilter .....	8-5
Using Predefined Filter Types .....	8-13
<b>Filter Design</b> .....	8-15
FIR Filters .....	8-16
Frequency Transformation Method .....	8-16
Frequency Sampling Method .....	8-18
Windowing Method .....	8-19
Creating the Desired Frequency Response Matrix .....	8-21
Computing the Frequency Response of a Filter .....	8-22

## Transforms

## 9

<b>Fourier Transform</b> .....	9-2
Definition of Fourier Transform .....	9-2
Discrete Fourier Transform .....	9-7
Applications of the Fourier Transform .....	9-10
<b>Discrete Cosine Transform</b> .....	9-15
The DCT Transform Matrix .....	9-16
DCT and Image Compression .....	9-17
<b>Radon Transform</b> .....	9-19
Plotting the Radon Transform .....	9-21
Viewing the Radon Transform as an Image .....	9-23
Using the Radon Transform to Detect Lines .....	9-24
Inverse Radon Transform .....	9-27
Example: Reconstructing an Image from Parallel Projection Data .....	9-30
<b>Fan-Beam Projection Data</b> .....	9-35

Computing Fan-Beam Projection Data .....	9-36
Reconstructing an Image from Fan-Beam Projection Data .....	9-38
Working with Fan-Beam Projection Data .....	9-39

## Morphological Operations

# 10

<b>Dilation and Erosion</b> .....	10-3
Understanding Dilation and Erosion .....	10-3
Structuring Elements .....	10-6
Dilating an Image .....	10-10
Eroding an Image .....	10-11
Combining Dilation and Erosion .....	10-13
Dilation- and Erosion-Based Functions .....	10-15
 <b>Morphological Reconstruction</b> .....	10-18
Marker and Mask .....	10-18
Pixel Connectivity .....	10-22
Flood-Fill Operations .....	10-24
Finding Peaks and Valleys .....	10-27
 <b>Distance Transform</b> .....	10-37
 <b>Objects, Regions, and Feature Measurement</b> .....	10-40
Connected-Component Labeling .....	10-40
Selecting Objects in a Binary Image .....	10-42
Finding the Area of the Foreground of a Binary Image ...	10-42
Finding the Euler Number of a Binary Image .....	10-43
 <b>Lookup Table Operations</b> .....	10-44
Creating a Lookup Table .....	10-44
Using a Lookup Table .....	10-45

<b>Getting Information about Pixel Values and Statistics</b> .....	<b>11-2</b>
Getting Information About Image Pixels .....	<b>11-2</b>
Getting the Intensity Profile of an Image .....	<b>11-3</b>
Displaying a Contour Plot of Image Data .....	<b>11-7</b>
Creating an Image Histogram .....	<b>11-9</b>
Getting Summary Statistics About an Image .....	<b>11-10</b>
Computing Properties for Image Regions .....	<b>11-10</b>
<b>Analyzing an Image</b> .....	<b>11-11</b>
Detecting Edges .....	<b>11-11</b>
Tracing Boundaries .....	<b>11-13</b>
Detecting Lines Using the Hough Transform .....	<b>11-17</b>
Using Quadtree Decomposition .....	<b>11-21</b>
<b>Analyzing the Texture of an Image</b> .....	<b>11-24</b>
Using Texture Filter Functions .....	<b>11-24</b>
Using a Gray-Level Co-Occurrence Matrix (GLCM) .....	<b>11-28</b>
<b>Intensity Adjustment</b> .....	<b>11-34</b>
Adjusting Intensity Values to a Specified Range .....	<b>11-35</b>
Histogram Equalization .....	<b>11-39</b>
Contrast-Limited Adaptive Histogram Equalization .....	<b>11-41</b>
Decorrelation Stretching .....	<b>11-42</b>
<b>Noise Removal</b> .....	<b>11-47</b>
Using Linear Filtering .....	<b>11-47</b>
Using Median Filtering .....	<b>11-48</b>
Using Adaptive Filtering .....	<b>11-50</b>

## Region-Based Processing

<b>Specifying a Region of Interest</b> .....	<b>12-2</b>
Selecting a Polygon .....	<b>12-2</b>

Other Selection Methods .....	12-4
<b>Filtering a Region</b> .....	12-5
Example: Filtering a Region in an Image .....	12-5
Specifying the Filtering Operation .....	12-6
<b>Filling a Region</b> .....	12-8

## Image Deblurring

# 13

<b>Understanding Deblurring</b> .....	13-2
Causes of Blurring .....	13-2
Deblurring Model .....	13-2
Deblurring Functions .....	13-4
<b>Deblurring with the Wiener Filter</b> .....	13-6
Refining the Result .....	13-7
<b>Deblurring with a Regularized Filter</b> .....	13-8
Refining the Result .....	13-9
<b>Deblurring with the Lucy-Richardson Algorithm</b> .....	13-10
Reducing the Effect of Noise Amplification .....	13-10
Accounting for Nonuniform Image Quality .....	13-11
Handling Camera Read-Out Noise .....	13-11
Handling Undersampled Images .....	13-12
Example: Using the deconvlucy Function to Deblur an Image .....	13-12
Refining the Result .....	13-15
<b>Deblurring with the Blind Deconvolution Algorithm</b> ..	13-16
Example: Using the deconvblind Function to Deblur an Image .....	13-16
Refining the Result .....	13-21
<b>Creating Your Own Deblurring Functions</b> .....	13-23

## Color

# 14

<b>Working with Different Screen Bit Depths</b> .....	14-2
Determining Screen Bit Depth .....	14-2
Choosing a Screen Bit Depth .....	14-4
<b>Reducing the Number of Colors in an Image</b> .....	14-5
Color Approximation .....	14-6
Reducing Colors in an Indexed Image .....	14-11
Dithering .....	14-12
<b>Converting Color Data Between Color Spaces</b> .....	14-14
Converting Between Device-Independent Color Spaces ...	14-14
Performing Profile-Based Color Space Conversions .....	14-18
Converting Between Device-Dependent Color Spaces ...	14-22

## Neighborhood and Block Operations

# 15

<b>Block Processing Operations</b> .....	15-2
Types of Block Processing Operations .....	15-2
<b>Sliding Neighborhood Operations</b> .....	15-4
Determining the Center Pixel .....	15-4
Performing a Sliding Neighborhood Operation .....	15-5
Padding Borders .....	15-5
Implementing Linear and Nonlinear Filtering .....	15-6
<b>Distinct Block Operations</b> .....	15-8
Specifying Overlap .....	15-10
<b>Column Processing Operations</b> .....	15-12

Sliding Neighborhoods .....	15-12
Using Column Processing with Distinct Block Operations .....	15-13

## Functions — By Category

# 16

<b>Image Display and Exploration</b> .....	16-2
Image Display and Exploration .....	16-2
Image File I/O .....	16-2
Image Types and Type Conversions .....	16-3
<b>GUI Tools</b> .....	16-5
Modular Interactive Tools .....	16-5
Navigational tools for Image Scroll Panel .....	16-5
Utility Functions for Interactive Tools .....	16-6
<b>Spatial Transformation and Image Registration</b> .....	16-8
Spatial Transformations .....	16-8
Image Registration .....	16-9
<b>Image Analysis and Statistics</b> .....	16-10
Image Analysis .....	16-10
Texture Analysis .....	16-10
Pixel Values and Statistics .....	16-11
<b>Image Arithmetic</b> .....	16-12
<b>Image Enhancement and Restoration</b> .....	16-13
Image Enhancement .....	16-13
Image Restoration (Deblurring) .....	16-13
<b>Linear Filtering and Transforms</b> .....	16-15
Linear Filtering .....	16-15
Linear 2-D Filter Design .....	16-15
Image Transforms .....	16-16



<b>Morphological Operations</b> .....	<b>16-17</b>
Intensity and Binary Images .....	<b>16-17</b>
Binary Images .....	<b>16-18</b>
Structuring Element (STREL) Creation and Manipulation .....	<b>16-19</b>
 <b>Region-Based, Neighborhood, and Block Processing</b> ..	<b>16-20</b>
Region-Based Processing .....	<b>16-20</b>
Neighborhood and Block Processing .....	<b>16-20</b>
 <b>Colormap and Color Space Functions</b> .....	<b>16-21</b>
Colormap Manipulation .....	<b>16-21</b>
Color Space Conversions .....	<b>16-21</b>
 <b>Miscellaneous Functions</b> .....	<b>16-23</b>
Toolbox Preferences .....	<b>16-23</b>
Toolbox Utility Functions .....	<b>16-23</b>
Interactive Mouse Utility Functions .....	<b>16-24</b>
Array Operations .....	<b>16-24</b>
Demos .....	<b>16-24</b>
Performance .....	<b>16-24</b>

## Functions — Alphabetical List

**17**

### Examples

**A**

<b>Introductory Examples</b> .....	<b>A-2</b>
<b>Image Display</b> .....	<b>A-2</b>
<b>Modular Tools</b> .....	<b>A-2</b>
<b>Morphology Examples</b> .....	<b>A-2</b>

<b>Image Analysis</b> .....	<b>A-3</b>
<b>Image Enhancement</b> .....	<b>A-3</b>
<b>Working with Regions of Interest</b> .....	<b>A-3</b>
<b>Working with Color</b> .....	<b>A-3</b>

---

**Index**

# Getting Started

---

This chapter contains two examples to get you started doing image processing using MATLAB® and the Image Processing Toolbox. The examples contain cross-references to other sections in the documentation manual that have in-depth discussions on the concepts presented in the examples.

What Is the Image Processing  
Toolbox? (p. 1-2)

Introduces the Image Processing  
Toolbox and its capabilities

Example 1 — Some Basic Concepts  
(p. 1-4)

Guides you through an example of  
some of the basic image processing  
capabilities of the toolbox, including  
reading, writing, and displaying  
images

Example 2 — Advanced Topics  
(p. 1-10)

Guides you through some advanced  
image processing topics, including  
components labeling, object property  
measurement, image arithmetic,  
morphological image processing, and  
contrast enhancement

Getting Help (p. 1-23)

Provides pointers to additional  
sources of information

Image Credits (p. 1-25)

Provides information about the  
sources of the images used in the  
documentation

## What Is the Image Processing Toolbox?

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

- Spatial image transformations
- Morphological operations
- Neighborhood and block operations
- Linear filtering and filter design
- Transforms
- Image analysis and enhancement
- Image registration
- Deblurring
- Region of interest operations

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

```
type function_name
```

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

For a list of the new features in this version of the Image Processing Toolbox, see the Release Notes documentation.

## Configuration 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.

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 ([www.mathworks.com](http://www.mathworks.com)).

## Related Products

The MathWorks provides several products that are relevant to the kinds of tasks you can perform with the Image Processing Toolbox and that extend the capabilities of MATLAB. For information about these related products, see [www.mathworks.com/products/image/related.html](http://www.mathworks.com/products/image/related.html).

## Compilability

The Image Processing Toolbox is compilable with the MATLAB Compiler except for the following two functions that launch GUIs:

- `cpselect`
- `imtool`

## Example 1 – Some Basic Concepts

This example introduces some basic image processing concepts, including reading and writing images, performing histogram equalization on an image, and getting information about an image. The example breaks this process into the following steps:

- “Step 1: Read and Display an Image” on page 1-4
- “Step 2: Check How the Image Appears in the Workspace” on page 1-5
- “Step 3: Improve Image Contrast” on page 1-6
- “Step 4: Write the Image to a Disk File” on page 1-8
- “Step 5: Check the Contents of the Newly Written File” on page 1-8

Before beginning with this example, you should already have installed the Image Processing Toolbox and have started MATLAB. If you are new to MATLAB, read the MATLAB Getting Started documentation to learn about basic MATLAB concepts.

### Step 1: Read and Display an Image

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

```
close all
```

To read an image, use the `imread` command. The example reads one of the sample images included with the Image Processing Toolbox, `pout.tif`, and stores it in an array named `I`.

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

`imread` infers from the file that the graphics file format is Tagged Image File Format (TIFF). For the list of supported graphics file formats, see the `imread` function reference documentation.

Now display the image. The toolbox includes two image display functions: `imshow` and `imtool`. `imshow` is the toolbox’s fundamental image display function. `imtool` starts the Image Tool which presents an integrated environment for displaying images and performing some common image

processing tasks. The Image Tool provides all the image display capabilities of `imshow` but also provides access to several other tools for navigating and exploring images, such as scroll bars, the Pixel Region tool, Image Information tool, and the Contrast Adjustment tool. For more information, see Chapter 4, “Displaying and Exploring Images”. You can use either function to display an image. This example uses `imshow`.

```
imshow(I)
```



**Grayscale Image `pout.tif`**

## Step 2: Check How the Image Appears in the Workspace

To see how the `imread` function stores the image data in the workspace, check the Workspace browser in the MATLAB desktop. The Workspace browser displays information about all the variables you create during a MATLAB session. The `imread` function returned the image data in the variable `I`, which is a 291-by-240 element array of `uint8` data. MATLAB can store images as `uint8`, `uint16`, or `double` arrays.

You can also get information about variables in the workspace by calling the `whos` command.

```
whos
Name      Size      Bytes  Class
I         291x240   69840  uint8 array
```

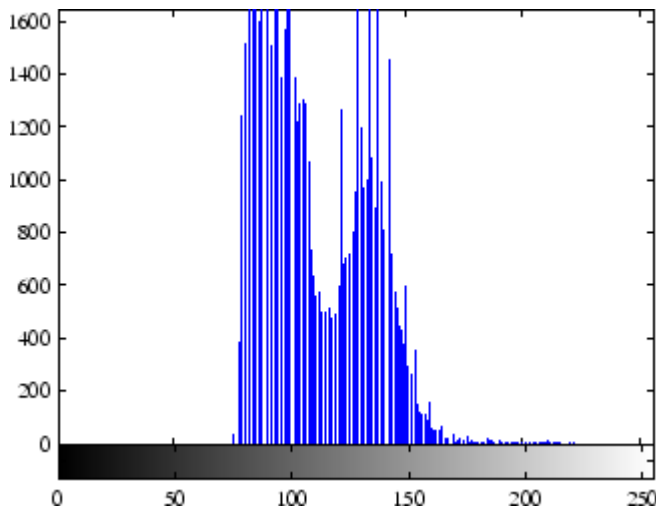
```
Grand total is 69840 elements using 69840 bytes
```

For more information about image storage classes, see “Converting Between Image Classes” on page 2-17.

### Step 3: Improve Image Contrast

`pout.tif` is a somewhat low contrast image. To see the distribution of intensities in `pout.tif`, 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)
```



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.

The toolbox provides several ways to improve the contrast in an image. One way is to call the `histeq` function to spread the intensity values over the full range of the image, a process called *histogram equalization*.

```
I2 = histeq(I);
```



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

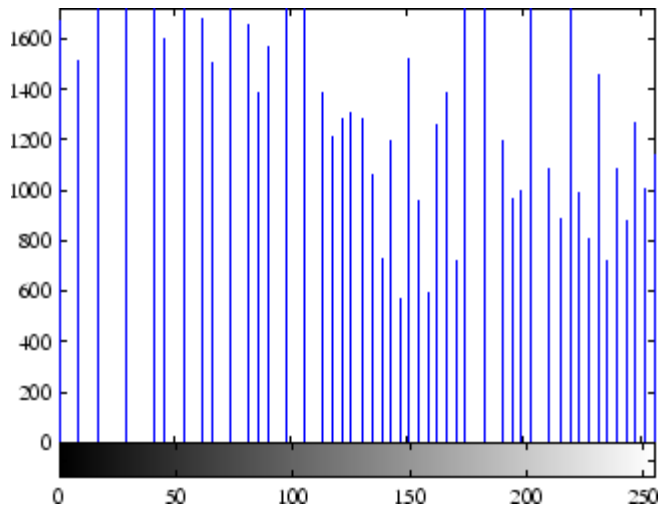
```
figure, imshow(I2)
```



#### **Equalized Version of pout.tif**

Call `imhist` again to create a histogram of the equalized image I2. If you compare the two histograms, the histogram of I2 is more spread out than the histogram of I1.

```
figure, imhist(I2)
```



The toolbox includes several other functions that perform contrast adjustment, including the `imadjust` and `adapthisteq` functions. See “Intensity Adjustment” on page 11-34 for more information. In addition, the toolbox includes an interactive tool, called the Adjust Contrast tool, that you can use to adjust the contrast and brightness of an image displayed in the Image Tool. To use this tool, call the `imcontrast` function or access the tool from the Image Tool. For more information, see “Adjusting the Contrast and Brightness of an Image” on page 4-36.

### **Step 4: Write the Image to a Disk File**

To write the newly adjusted image `I2` to a disk file, use the `imwrite` function. If you include the filename extension `'.png'`, the `imwrite` function writes the image to a file in Portable Network Graphics (PNG) format, but you can specify other formats.

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

See the `imwrite` function reference page for a list of file formats it supports. See also “Writing Image Data” on page 3-5 for a tutorial discussion on writing images using the Image Processing Toolbox.

### **Step 5: Check the Contents of the Newly Written File**

To see what `imwrite` wrote to the disk file, use the `imfinfo` function.

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

The `imfinfo` function returns information about the image in the file, such as its format, size, width, and height. See “Getting Information About a Graphics File” on page 3-2 for more information about using `imfinfo`.

```
ans =  
  
    Filename: 'pout2.png'  
    FileModDate: '29-Dec-2005 09:34:39'  
    FileSize: 36938  
    Format: 'png'  
    FormatVersion: []  
    Width: 240
```

```
        Height: 291
        BitDepth: 8
        ColorType: 'grayscale'
FormatSignature: [137 80 78 71 13 10 26 10]
        Colormap: []
        Histogram: []
        InterlaceType: 'none'
        Transparency: 'none'
SimpleTransparencyData: []
        BackgroundColor: []
        RenderingIntent: []
        Chromaticities: []
            Gamma: []
        XResolution: []
        YResolution: []
        ResolutionUnit: []
            XOffset: []
            YOffset: []
            OffsetUnit: []
        SignificantBits: []
        ImageModTime: '29 Dec 2005 14:34:39 +0000'
            Title: []
            Author: []
        Description: []
        Copyright: []
        CreationTime: []
            Software: []
        Disclaimer: []
            Warning: []
            Source: []
            Comment: []
        OtherText: []
```

## **Example 2 – Advanced Topics**

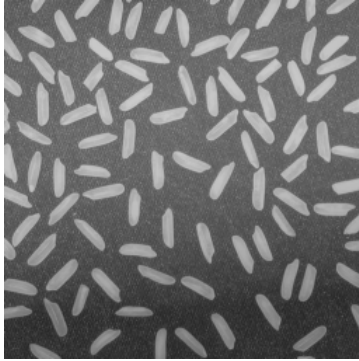
This example introduces some advanced image processing concepts. The example calculates statistics about objects in the image but, before it performs these calculations, it preprocesses the image to achieve better results. The preprocessing involves creating a uniform background in the image and converting the image into a binary image. The example breaks this process into the following steps:

- “Step 1: Read and Display an Image” on page 1-11
- “Step 2: Estimate the Value of Background Pixels” on page 1-11
- “Step 3: View the Background Approximation as a Surface” on page 1-12
- “Step 4: Create an Image with a Uniform Background” on page 1-14
- “Step 5: Adjust the Contrast in the Processed Image” on page 1-14
- “Step 6: Create a Binary Version of the Image” on page 1-15
- “Step 7: Determine the Number of Objects in the Image” on page 1-16
- “Step 8: Examine the Label Matrix” on page 1-17
- “Step 9: Display the Label Matrix as a Pseudocolor Indexed Image” on page 1-18
- “Step 10: Measure Object Properties in the Image” on page 1-19
- “Step 11: Compute Statistical Properties of Objects in the Image” on page 1-21

## Step 1: Read and Display an Image

Clear the MATLAB workspace of any variables, close open figure windows, and close all open Image Tools.

Read and display the grayscale image `rice.png`.



**Grayscale Image `rice.png`**

## Step 2: Estimate the Value of Background Pixels

In the sample image, the background illumination is brighter in the center of the image than at the bottom. In this step, the example uses a morphological opening operation to estimate the background illumination. Morphological opening is an erosion followed by a dilation, using the same structuring element for both operations. The opening operation has the effect of removing objects that cannot completely contain the structuring element. For more information about morphological image processing, see Chapter 10, “Morphological Operations”.

The example calls the `imopen` function to perform the morphological opening operation and then calls the `imshow` function to view the results. Note how the example calls the `strel` function to create a disk-shaped structuring element with a radius of 15. To remove the rice grains from the image, the structuring element must be sized so that it cannot fit entirely inside a single grain of rice.

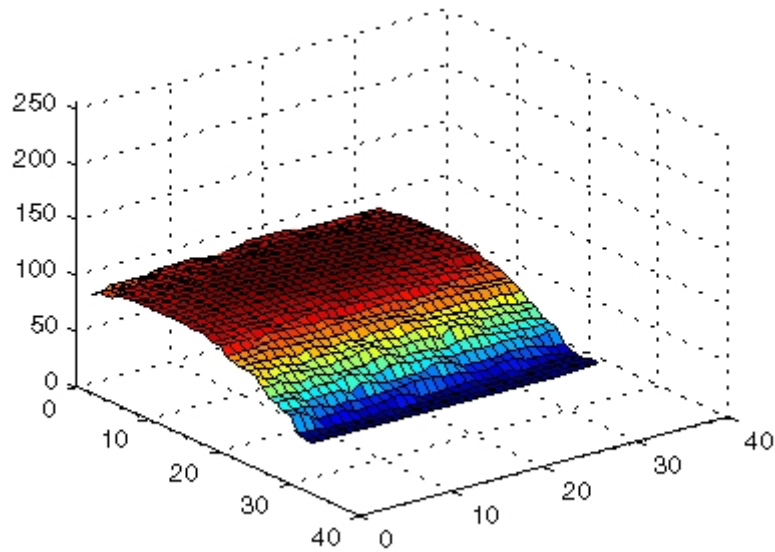
### **Step 3: View the Background Approximation as a Surface**

Use the `surf` command to create a surface display of the background approximation background. The `surf` command creates colored parametric surfaces that enable you to view mathematical functions over a rectangular region. The `surf` function requires data of class `double`, however, so you first need to convert `background` using the `double` command.

The example uses MATLAB indexing syntax to view only 1 out of 8 pixels in each direction; otherwise the surface plot would be too dense. The example also sets the scale of the plot to better match the range of the `uint8` data and reverses the  $y$ -axis of the display to provide a better view of the data (the pixels at the bottom of the image appear at the front of the surface plot).

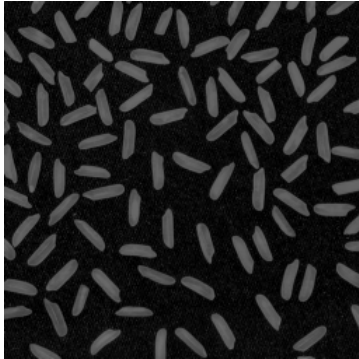
In the surface display, `[0, 0]` represents the origin, or upper left corner of the image. The highest part of the curve indicates that the highest pixel values of background (and consequently `rice.png`) occur near the middle rows of the image. The lowest pixel values occur at the bottom of the image and are represented in the surface plot by the lowest part of the curve.

The surface plot is a Handle Graphics® object. You can use object properties to fine-tune its appearance. For information on working with MATLAB graphics, see the MATLAB graphics documentation.



## Step 4: Create an Image with a Uniform Background

To create a more uniform background, subtract the background image, background, from the original image, I, and then view the image.

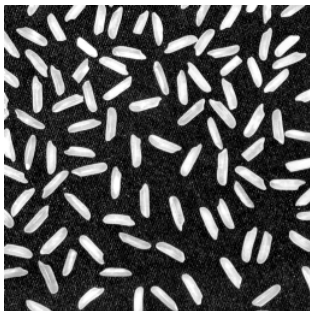


**Image with Uniform Background**

## Step 5: Adjust the Contrast in the Processed Image

After subtraction, the image has a uniform background but is now a bit too dark. Use `imadjust` to adjust the contrast of the image. `imadjust` increases the contrast of the image by saturating 1% of the data at both low and high intensities of I2 and by stretching the intensity values to fill the `uint8` dynamic range. See the reference page for `imadjust` for more information.

The following example adjusts the contrast in the image created in the previous step and displays it.

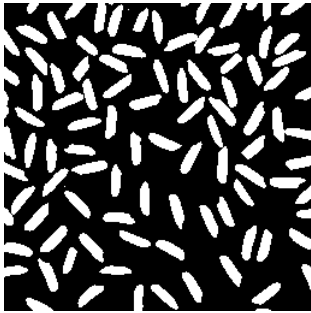


**Image After Intensity Adjustment**



## Step 6: Create a Binary Version of the Image

Create a binary version of the image so that you can use toolbox functions to count the number of rice grains. Use the `im2bw` function to convert the grayscale image into a binary image by using thresholding. The function `graythresh` automatically computes an appropriate threshold to use to convert the grayscale image to binary.



**Binary Version of the Image**

The binary image `bw` returned by `im2bw` is of class `logical`, as can be seen in this call to `whos`. The Image Processing Toolbox uses logical arrays to represent binary images. For more information, see “Binary Images” on page 2-8.

```
whos
```

MATLAB responds with

Name	Size	Bytes	Class
I	256x256	65536	uint8 array
I2	256x256	65536	uint8 array
I3	256x256	65536	uint8 array
background	256x256	65536	uint8 array
bw	256x256	65536	logical array
level	1x1	8	double array

```
Grand total is 327681 elements using 327688 bytes
```

## Step 7: Determine the Number of Objects in the Image

After converting the image to a binary image, you can use the `bwlabel` function to determine the number of grains of rice in the image. The `bwlabel` function labels all the components in the binary image `bw` and returns the number of components it finds in the image in the output value, `numObjects`.

The accuracy of the results depends on a number of factors, including

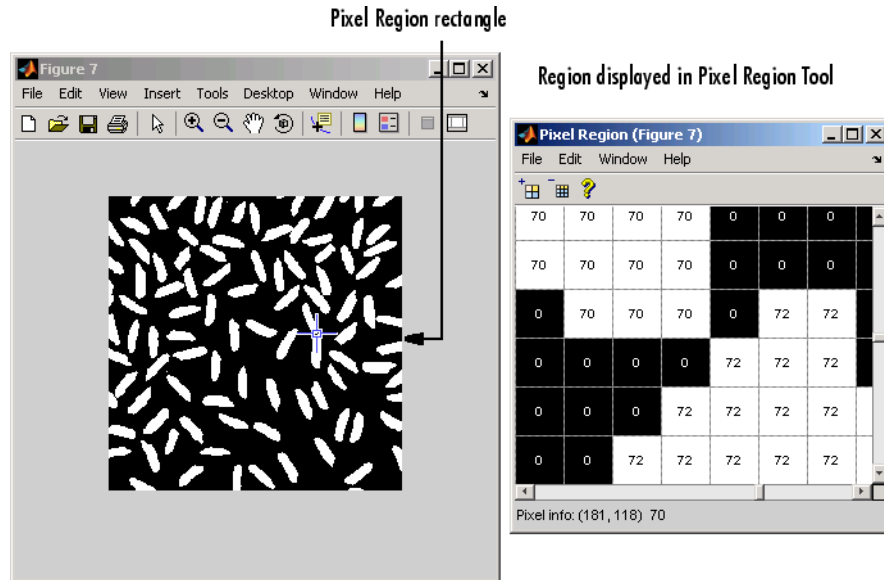
- The size of the objects
- Whether or not any objects are touching (in which case they might be labeled as one object)
- The accuracy of the approximated background
- The connectivity selected. The parameter 4, passed to the `bwlabel` function, means that pixels must touch along an edge to be considered connected. For more information about the connectivity of objects, see “Pixel Connectivity” on page 10-22.

## Step 8: Examine the Label Matrix

To better understand the label matrix returned by the `bwlabel` function, this step explores the pixel values in the image. There are several ways to get a close-up view of pixel values. For example, you can use `imcrop` to select a small portion of the image. Another way is to use toolbox Pixel Region tool to examine pixel values. The following example displays the label matrix, using `imshow`, and then starts a Pixel Region tool associated with the displayed image.

By default, it automatically associates itself with the image in the current figure. The Pixel Region tool draws a rectangle, called the *pixel region rectangle*, in the center of the visible part of the image. This rectangle defines which pixels are displayed in the Pixel Region tool. As you move the rectangle, the Pixel Region tool updates the pixel values displayed in the window. For more information about using the toolbox modular interactive tools, see Chapter 5, “Building GUIs with Modular Tools”.

The following figure shows the Image Viewer with the Pixel Region rectangle positioned over the edges of two rice grains. Note how all the pixels in the rice grains have the values assigned by the `bwlabel` function and the background pixels have the value 0 (zero).

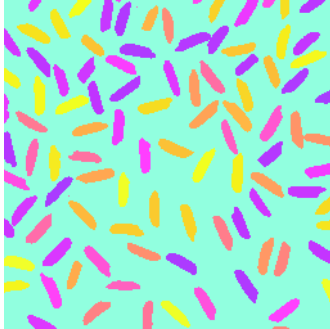


### Examining the Label Matrix with the Pixel Region Tool

## Step 9: Display the Label Matrix as a Pseudocolor Indexed Image

A good way to view a label matrix is to display it as a pseudocolor indexed image. In the pseudocolor image, the number that identifies each object in the label matrix maps to a different color in the associated colormap matrix. The colors in the image make objects easier to distinguish.

To view a label matrix in this way, use the `label2rgb` function. Using this function, you can specify the colormap, the background color, and how objects in the label matrix map to colors in the colormap.



**Label Matrix Displayed as Pseudocolor Image**

## Step 10: Measure Object Properties in the Image

The `regionprops` command measures object or region properties 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.

The following example uses `regionprops` to create a structure array containing some basic properties for labeled. When you set the `properties` parameter to `'basic'`, the `regionprops` function returns three commonly used measurements: area, centroid (or center of mass), and bounding box. The bounding box represents the smallest rectangle that can contain a region, or in this case, a grain of rice.

MATLAB responds with

```
graindata =  
  
101x1 struct array with fields:  
    Area  
    Centroid  
    BoundingBox
```

To find the area of the 51st labeled component, access the Area field in the 51st element in the `graindata` structure array. Note that structure field names are case sensitive.

returns the following results

```
ans =
```

```
140
```

To find the smallest possible bounding box and the centroid (center of mass) for the same component, use this code:

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

```
ans =
```

```
107.5000    4.5000    13.0000    20.0000
```

```
ans =
```

```
114.5000    15.4500
```

## Step 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. (In this example, the largest grain is actually two grains of rice that are touching.)

returns

```
ans =
```

```
404
```

Use the `find` command to return the component label of the grain of rice with this area.

returns

```
biggrain =
```

```
59
```

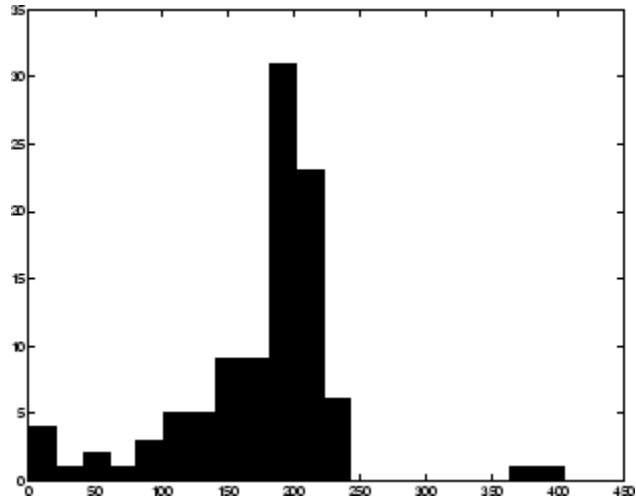
Find the mean of all the rice grain sizes.

returns

```
ans =
```

```
175.0396
```

Make a histogram containing 20 bins that show the distribution of rice grain sizes. The histogram shows that the most common sizes for rice grains in this image are in the range of 150 to 250 pixels.





## Getting Help

For more information about the topics covered in these exercises, read the tutorial chapters that make up the remainder of this documentation. For reference information about any of the Image Processing Toolbox functions, see the online Chapter 17, “Functions — Alphabetical List”, which complements the M-file help that is displayed in the MATLAB command window when you type

```
help functionname
```

For example,

```
help imtool
```

### Online Help

The Image Processing Toolbox User’s Guide documentation is available online in both HTML and PDF formats. To access the HTML help, select **Help** from the menu bar of the MATLAB desktop. In the Help Navigator pane, click the **Contents** tab and expand the **Image Processing Toolbox** topic in the list.

To access the PDF help, click **Image Processing Toolbox** in the **Contents** tab of the Help browser and go to the link under “Printable Documentation (PDF).” (Note that to view the PDF help, you must have Adobe’s Acrobat Reader installed.)

### 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.

To view all the Image Processing Toolbox demos, call the `iptdemos` function. This displays an HTML page in the MATLAB Help browser that lists all the Image Processing Toolbox demos.

You can also view this page by starting the MATLAB Help browser and clicking the **Demos** tab in the Help Navigator pane. From the list of products with demos, select the Image Processing Toolbox.

The toolbox demos are located under the subdirectory

```
matlabroot\toolbox\images\imdemos
```

where *matlabroot* represents your MATLAB installation directory.

## **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.

## Image Credits

This table lists the copyright owners of the images used in the Image Processing Toolbox documentation.

<b>Image</b>	<b>Source</b>
cameraman	Copyright Massachusetts Institute of Technology. Used with permission.
cell	Cancer cell from a rat's prostate, courtesy of Alan W. Partin, M.D., Ph.D., Johns Hopkins University School of Medicine.
circuit	Micrograph of 16-bit A/D converter circuit, courtesy of Steve Decker and Shujaat Nadeem, MIT, 1993.
concordaerial and westconcordaerial	Visible color aerial photographs courtesy of mPower3/Emerge.
concordorthophoto and westconcordorthophoto	Orthoregistered photographs courtesy of Massachusetts Executive Office of Environmental Affairs, MassGIS.
forest	Photograph of Carmanah Ancient Forest, British Columbia, Canada, courtesy of Susan Cohen.
LAN files	Permission to use Landsat data sets provided by Space Imaging, LLC, Denver, Colorado.
liftingbody	Picture of M2-F1 lifting body in tow, courtesy of NASA (Image number E-10962).
m83	M83 spiral galaxy astronomical image courtesy of Anglo-Australian Observatory, photography by David Malin.
moon	Copyright Michael Myers. Used with permission.
saturn	Voyager 2 image, 1981-08-24, NASA catalog #PIA01364.
solarspectra	Courtesy of Ann Walker. Used with permission.

<b>Image</b>	<b>Source</b>
tissue	Courtesy of Alan W. Partin, M.D., PhD., Johns Hopkins University School of Medicine.
trees	<i>Trees with a View</i> , watercolor and ink on paper, copyright Susan Cohen. Used with permission.

# Introduction

---

This chapter introduces you to the fundamentals of image processing using MATLAB and the Image Processing Toolbox.

Images in MATLAB and the Image Processing Toolbox (p. 2-2)

How images are represented in MATLAB and the Image Processing Toolbox

Image Types in the Toolbox (p. 2-7)

Fundamental image types supported by the Image Processing Toolbox

Converting Between Image Types (p. 2-15)

Converting between the image types

Converting Between Image Classes (p. 2-17)

Converting image data from one class to another

Working with Image Sequences (p. 2-19)

Working with sequences of images

Image Arithmetic (p. 2-25)

Adding, subtracting, multiplying, and dividing images

## 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 truecolor images, 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.

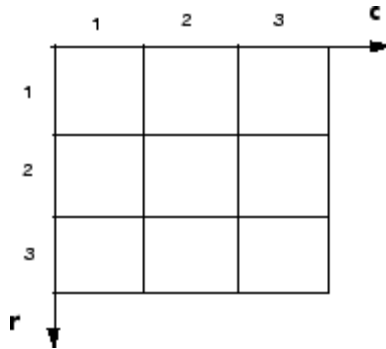
### Coordinate Systems

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” on page 2-3
- “Spatial Coordinates” on page 2-4
- “Using a Nondefault Spatial Coordinate System” on page 2-5

## 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 the following figure.



### 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 is displayed easy to understand. For example, the data for the pixel in the fifth row, second column is stored in the matrix element (5,2). You use normal MATLAB matrix subscripting to access values of individual pixels. For example, the MATLAB code

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

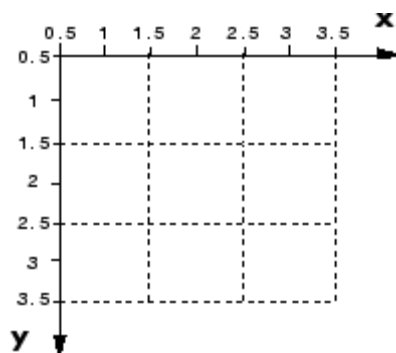
returns the value of the pixel at row 2, column 15 of the image I.

## 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).

The following figure illustrates the spatial coordinate system used for images. Notice that  $y$  increases downward.



### 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's 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 2-5 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.

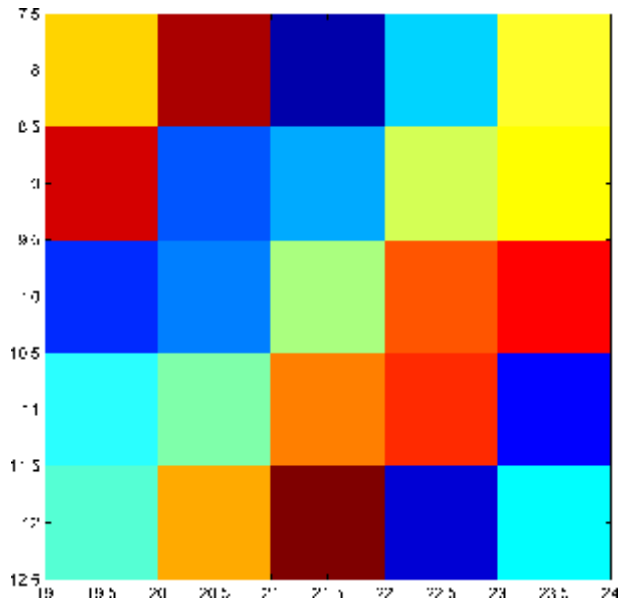
In some situations, however, you might want to use a nondefault spatial coordinate system. For example, you could specify that the upper left corner of an image is the point  $(19.0,7.5)$ , rather than  $(0.5,0.5)$ . If you call a function that returns coordinates for this image, the coordinates returned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



For information about the syntax variations that specify nondefault spatial coordinates, see the reference page for `imshow`.

## Image Types in the Toolbox

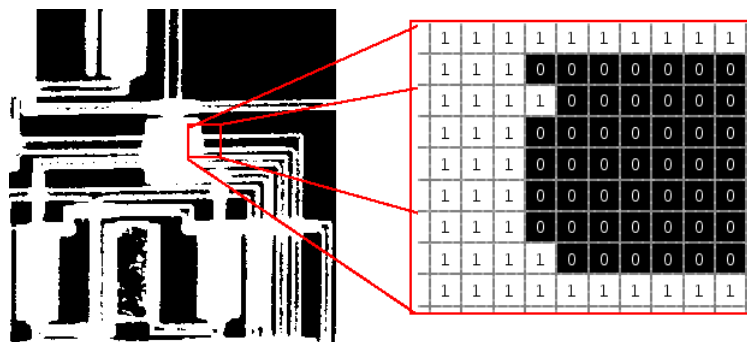
The Image Processing Toolbox defines four basic types of images, summarized in the following table. These image types determine the way MATLAB interprets data matrix elements as pixel intensity values. The sections that follow provide more information about each image type. See also “Converting Between Image Types” on page 2-15.

Image Type	Interpretation
Binary (Also known as a <i>bilevel</i> image)	Logical array containing only 0s and 1s, interpreted as black and white, respectively.
Indexed (Also known as a <i>pseudocolor</i> image)	Array of class <code>logical</code> , <code>uint8</code> , <code>uint16</code> , <code>single</code> , or <code>double</code> whose pixel values are direct indices into a colormap. The colormap is an $m$ -by-3 array of class <code>double</code> .  For <code>single</code> or <code>double</code> arrays, integer values range from $[1, p]$ . For <code>logical</code> , <code>uint8</code> , or <code>uint16</code> arrays, values range from $[0, p-1]$ .
Grayscale (Also known as an <i>intensity</i> image)	Array of class <code>uint8</code> , <code>uint16</code> , <code>int16</code> , <code>single</code> , or <code>double</code> whose pixel values specify intensity values.  For <code>single</code> or <code>double</code> arrays, values range from $[0, 1]$ . For <code>uint8</code> , values range from $[0, 255]$ . For <code>uint16</code> , values range from $[0, 65535]$ . For <code>int16</code> , values range from $[-32768, 32767]$ .
Truecolor (Also known as an <i>RGB</i> image )	$m$ -by- $n$ -by-3 array of class <code>uint8</code> , <code>uint16</code> , <code>single</code> , or <code>double</code> whose pixel values specify intensity values.  For <code>single</code> or <code>double</code> arrays, values range from $[0, 1]$ . For <code>uint8</code> , values range from $[0, 255]$ . For <code>uint16</code> , values range from $[0, 65535]$ .

## Binary Images

In a binary image, each pixel assumes one of only two discrete values: 1 or 0. A binary image is stored as a logical array. By convention, this documentation uses the variable name `BW` to refer to binary images.

The following figure shows a binary image with a close-up view of some of the pixel values.



**Pixel Values in a Binary Image**

## Indexed Images

An indexed image consists of an array and a colormap matrix. The pixel values in the array are direct indices into a colormap. By convention, this documentation uses the variable name `X` to refer to the array and `map` to refer to the colormap.

The colormap matrix 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`.

A colormap is often stored with an indexed image and is automatically loaded with the image when you use the `imread` function. After you read the image and the colormap into the MATLAB workspace as separate variables, you must keep track of the association between the image and colormap. However, you are not limited to using the default colormap--you can use any colormap that you choose.

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 `single` or `double`, it normally contains integer values 1 through  $p$ , where  $p$  is the length of the colormap. 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 `logical`, `uint8` or `uint16`, the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on.

The following figure illustrates the structure of an indexed image. In the figure, the image matrix is of class `double`, so the value 5 points to the fifth row of the colormap.

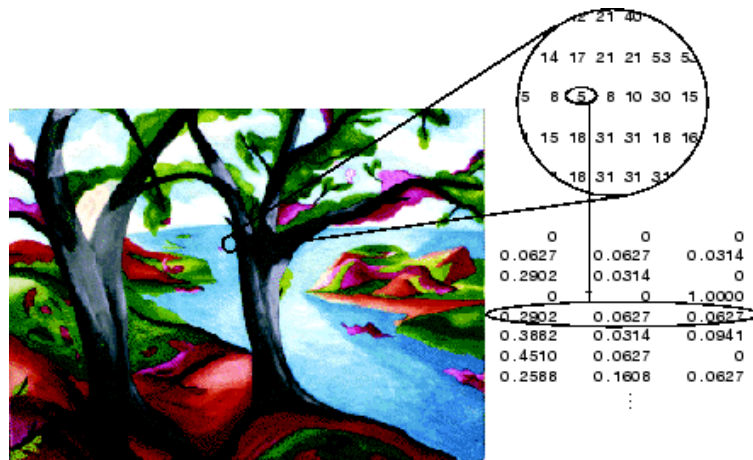


Image Courtesy of Susan Cohen

### Pixel Values Index to Colormap Entries in Indexed Images

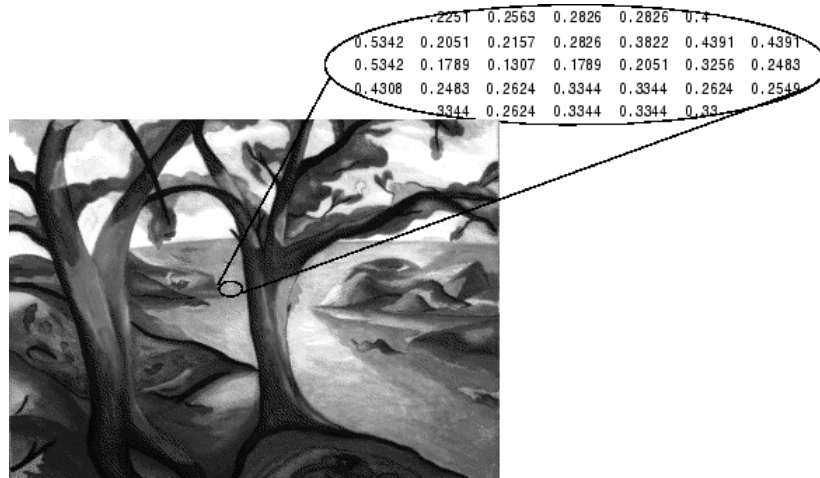
## Grayscale Images

A grayscale image (also spelled gray-scale) is a data matrix whose values represent intensities within some range. MATLAB stores a grayscale image as a individual matrix, with each element of the matrix corresponding to one image pixel. By convention, this documentation uses the variable name `I` to refer to grayscale images.

The matrix can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. While grayscale images are rarely saved with a colormap, MATLAB uses a colormap to display them.

For a matrix of class `single` or `double`, using the default grayscale colormap, the intensity 0 represents black and the intensity 1 represents white. For a matrix of type `uint8`, `uint16`, or `int16`, the intensity `intmin(class(I))` represents black and the intensity `intmax(class(I))` represents white.

The figure below depicts a grayscale image of class `double`.



**Pixel Values in a Grayscale Image Define Gray Levels**

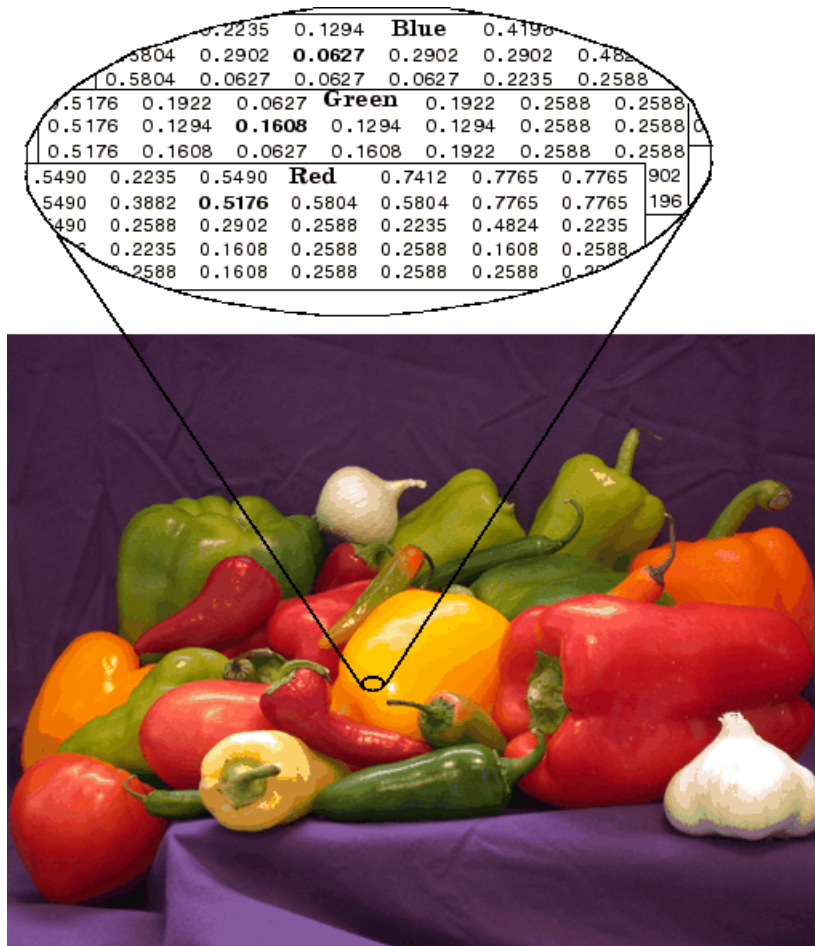
## Truecolor Images

A truecolor image is 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. MATLAB store truecolor images as an  $m$ -by- $n$ -by-3 data array that defines red, green, and blue color components for each individual pixel. Truecolor images do not use a colormap. 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 truecolor 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.

A truecolor array can be of class `uint8`, `uint16`, `single`, or `double`. In a truecolor array of class `single` or `double`, each color component is a value between 0 and 1. A pixel whose color components are (0,0,0) is displayed as black, and a pixel whose color components are (1,1,1) is displayed 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.

The following figure depicts a truecolor image of class double.



### The Color Planes of a Truecolor Image

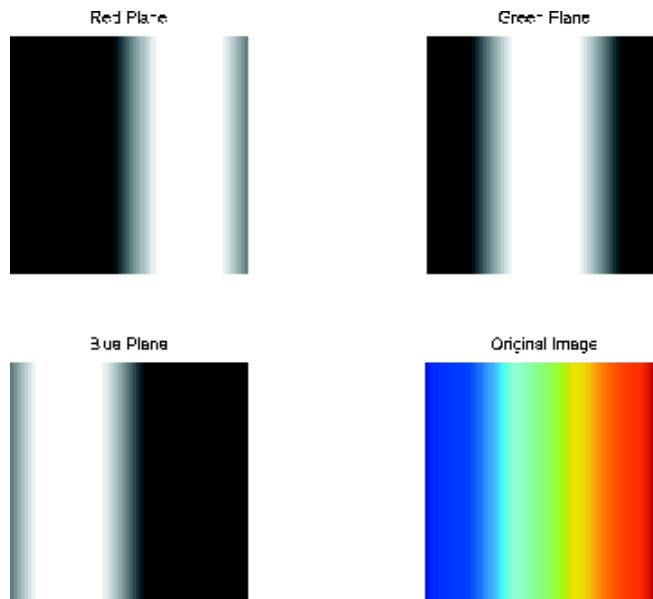
To determine the color of the pixel at (2,3), you would look at the RGB triplet stored in (2,3,1:3). Suppose (2,3,1) contains the value 0.5176, (2,3,2) contains 0.1608, and (2,3,3) contains 0.0627. The color for the pixel at (2,3) is

0.5176 0.1608 0.0627



To further illustrate the concept of the three separate color planes used in a truecolor image, the code sample below creates a simple 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). The example 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)
```



**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.,  $R = 0$ .

## Converting Between Image Types

You might need to convert an image from one type to another. For example, if you want to filter a color image that is stored as an indexed image, you must first convert it to truecolor format. When you apply the filter to the truecolor 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 might not be meaningful.

---

**Note** When you convert an image from one format to another, the resulting image might look different from the original. For example, if you convert a color indexed image to a grayscale image, the resulting image is grayscale, not color.

---

The following table lists all the image type conversion functions in the Image Processing Toolbox.

Function	Description
dither	Use dithering to convert a grayscale image to a binary image or to convert a truecolor image to an indexed image
gray2ind	Convert a grayscale image to an indexed image
grayslice	Convert a grayscale image to an indexed image by using multilevel thresholding
im2bw	Convert a grayscale image, indexed image, or truecolor image, to a binary image, based on a luminance threshold
ind2gray	Convert an indexed image to a grayscale image
ind2rgb	Convert an indexed image to a truecolor image
mat2gray	Convert a data matrix to a grayscale image, by scaling the data
rgb2gray	Convert a truecolor image to a grayscale image
rgb2ind	Convert a truecolor image to an indexed image

You can perform certain conversions just using MATLAB syntax. For example, you can convert a grayscale image to truecolor format by concatenating three copies of the original matrix along the third dimension.

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

The resulting truecolor 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 functions, there are other functions that return a different image type as part of the operation they perform. For example, the region-of-interest functions return a binary image that you can use to mask an image for filtering or for other operations.

## **Color Space Conversions**

The Image Processing Toolbox represents colors as RGB values in both truecolor and indexed images. 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 functions to convert between 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 14, “Color”.

## Converting Between Image Classes

You can convert `uint8` and `uint16` image data to `double` using the MATLAB `double` function. However, converting between 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 classes, use one of these toolbox functions: `im2uint8`, `im2uint16`, `im2int16`, `im2single`, or `im2double`. These functions automatically handle the rescaling and offsetting of the original data of any image class. 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);
```

### Losing Information in Conversions

When you convert to a class that uses fewer bits to represent numbers, you generally lose some of the information in your image. For example, a `uint16` grayscale image is capable of storing up to 65,536 distinct shades of gray, but a `uint8` grayscale image can store only 256 distinct shades of gray. When you convert a `uint16` grayscale image to a `uint8` grayscale image, `im2uint8` *quantizes* the gray shades in the original image. In other words, all values from 0 to 127 in the original image become 0 in the `uint8` image, values from 128 to 385 all become 1, and so on.

### Converting Indexed Images

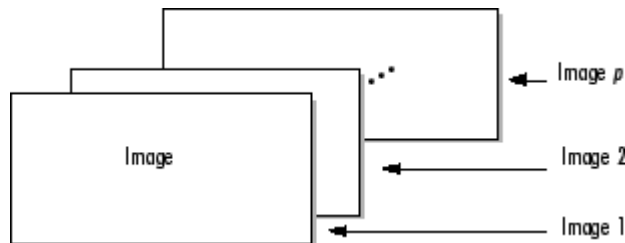
It is not always possible to convert an indexed image from one storage class to another. In an indexed image, the image matrix contains only indices into a colormap, rather than the color data itself, so no quantization of the color data is possible during the conversion.

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 “Reducing Colors in an Indexed Image” on page 14-11 for more information.

## Working with Image Sequences

Some applications work with collections of images related by time, such as frames in a movie, or by view (spatial location), such as magnetic resonance imaging (MRI) slices. These collections of images are referred to by a variety of names, such as image sequences or image stacks.

The ability to create N-dimensional arrays can provide a convenient way to store image sequences. For example, an  $m$ -by- $n$ -by- $p$  array can store an array of  $p$  two-dimensional images, such as grayscale or binary images, as shown in the following figure. An  $m$ -by- $n$ -by-3-by- $p$  array can store truecolor images where each image is made up of three planes.



### Multidimensional Array Containing an Image Sequence

Many toolbox functions can operate on multi-dimensional arrays and, consequently, can operate on image sequences. For example, if you pass a multi-dimensional array to the `imtransform` function, it applies the same 2-D transformation to all 2-D planes along the higher dimension.

Some toolbox functions that accept multi-dimensional arrays, however, do not by default interpret an  $m$ -by- $n$ -by- $p$  or an  $m$ -by- $n$ -by-3-by- $p$  array as an image sequence. To use these functions with image sequences, you must use particular syntax and be aware of other limitations. The following table lists these toolbox functions and provides guidelines about how to use them to process image sequences. To see an example of using one of these functions with an image sequence, see “Example: Processing Image Sequences” on page 2-22.

(Two toolbox functions, `immovie` and `montage`, work with 4-D arrays called *multi-frame image arrays*. See “Multi-Frame Image Arrays” on page 2-23 for more information.)

<b>Function</b>	<b>Image Sequence Dimensions</b>	<b>Guideline When Used with an Image Sequence</b>
<code>bwlabeln</code>	<i>m-by-n-by-p</i> only	Must use the <code>bwlabeln(BW,conn)</code> syntax with a 2-D connectivity.
<code>deconvblind</code>	<i>m-by-n-by-p</i> or <i>m-by-n-by-3-by-p</i>	PSF argument can be either 1-D or 2-D.
<code>deconvlucy</code>	<i>m-by-n-by-p</i> or <i>m-by-n-by-3-by-p</i>	PSF argument can be either 1-D or 2-D.
<code>edgetaper</code>	<i>m-by-n-by-p</i> or <i>m-by-n-by-3-by-p</i>	PSF argument can be either 1-D or 2-D.
<code>entropyfilt</code>	<i>m-by-n-by-p</i> only	<code>nhood</code> argument must be 2-D.
<code>imabsdiff</code>	<i>m-by-n-by-p</i> or <i>m-by-n-by-3-by-p</i>	Image sequences must be the same size.
<code>imadd</code>	<i>m-by-n-by-p</i> or <i>m-by-n-by-3-by-p</i>	Image sequences must be the same size. Cannot add scalar to image sequence.
<code>imbothat</code>	<i>m-by-n-by-p</i> only	SE argument must be 2-D.
<code>imclose</code>	<i>m-by-n-by-p</i> only	SE argument must be 2-D.
<code>imdilate</code>	<i>m-by-n-by-p</i> only	SE argument must be 2-D.
<code>imdivide</code>	<i>m-by-n-by-p</i> or <i>m-by-n-by-3-by-p</i>	Image sequences must be the same size.
<code>imerode</code>	<i>m-by-n-by-p</i> only	SE argument must be 2-D.
<code>imextendedmax</code>	<i>m-by-n-by-p</i> only	Must use the <code>imextendedmax(I,h,conn)</code> syntax with a 2-D connectivity.
<code>imextendedmin</code>	<i>m-by-n-by-p</i> only	Must use the <code>imextendedmin(I,h,conn)</code> syntax with a 2-D connectivity.
<code>imfilter</code>	<i>m-by-n-by-p</i> or <i>m-by-n-by-3-by-p</i>	With grayscale images, <code>h</code> can be 2-D. With truecolor images (RGB), <code>h</code> can be 2-D or 3-D.



<b>Function</b>	<b>Image Sequence Dimensions</b>	<b>Guideline When Used with an Image Sequence</b>
imhmax	<i>m-by-n-by-p</i> only	Must use the <code>imhmax(I,h,conn)</code> syntax with a 2-D connectivity.
imhmin	<i>m-by-n-by-p</i> only	Must use the <code>imhmin(I,h,conn)</code> syntax with a 2-D connectivity.
imlincomb	<i>m-by-n-by-p</i> or <i>m-by-n-by-3-by-p</i>	Image sequences must be the same size.
immultiply	<i>m-by-n-by-p</i> or <i>m-by-n-by-3-by-p</i>	Image sequences must be the same size.
imopen	<i>m-by-n-by-p</i> only	SE argument must be 2-D.
imregionalmax	<i>m-by-n-by-p</i> only	Must use the <code>imextendedmax(I,conn)</code> syntax with a 2-D connectivity.
imregionalmin	<i>m-by-n-by-p</i> only	Must use the <code>imextendedmin(I,conn)</code> syntax with a 2-D connectivity.
imtransform	<i>m-by-n-by-p</i> or <i>m-by-n-by-3-by-p</i>	TFORM argument must be 2-D.
imsubtract	<i>m-by-n-by-p</i> or <i>m-by-n-by-3-by-p</i>	Image sequences must be the same size.
imtophat	<i>m-by-n-by-p</i> only	SE argument must be 2-D.
padarray	<i>m-by-n-by-p</i> or <i>m-by-n-by-3-by-p</i>	PADSIZE argument must be a two-element vector.
rangefilt	<i>m-by-n-by-p</i> only	NHOOD argument must be 2-D.
stdfilt	<i>m-by-n-by-p</i> only	NHOOD argument must be 2-D.

Function	Image Sequence Dimensions	Guideline When Used with an Image Sequence
tformarray	$m$ -by- $n$ -by- $p$ or $m$ -by- $n$ -by-3-by- $p$	T must be 2-D to 2-D (compatible with imtransform). R must be 2-D. TDIMS_A and TDIMS_B must be 2-D, i.e., [2 1] or [1 2] TSIZE_B must be a two-element array [D1 D2], where D1 and D2 are the first and second transform dimensions of the output space. TMAP_B must be [TSIZE_B 2] F can be a scalar or a $p$ -by-1 array for $m$ -by- $n$ -by- $p$ arrays, or it can be a scalar, 1-by- $p$ array, 3-by-1 array, or 3-by- $p$ array, for $m$ -by- $n$ -by-3-by- $p$ arrays.
watershed	$m$ -by- $n$ -by- $p$ only	Must use watershed(I,conn) syntax with a 2-D connectivity.

## Example: Processing Image Sequences

This example starts by reading a series of images from a directory into the MATLAB workspace, storing the images in an  $m$ -by- $n$ -by- $p$  array. The example then passes the entire array to the `stdfilt` function and performs standard deviation filtering on each image in the sequence. Note that, to use `stdfilt` with an image sequence, you must use the `nhood` argument, specifying a 2-D neighborhood.

```
% Create an array of filenames that make up the image sequence
fileFolder = fullfile(matlabroot,'toolbox','images','imdemos');
dirOutput = dir(fullfile(fileFolder,'AT3_1m4_*.tif'));
fileNames = {dirOutput.name}';
numFrames = numel(fileNames);

I = imread(fileNames{1});

% Preallocate the array
```

```

sequence = zeros([size(I) numFrames],class(I));
sequence(:,:,1) = I;

% Create image sequence array
for p = 2:numFrames
    sequence(:,:,p) = imread(fileNames{p});
end

% Process sequence
sequenceNew = stdfilt(sequence,ones(3));

% View results
figure;
for k = 1:numFrames
    imshow(sequence(:,:,k));
    title(sprintf('Original Image # %d',k));
    pause(1);
    imshow(sequenceNew(:,:,k),[]);
    title(sprintf('Processed Image # %d',k));
    pause(1);
end

```

## Multi-Frame Image Arrays

The toolbox includes two functions, `immovie` and `montage`, that work with a specific type of multi-dimensional array called a multi-frame array. In this array, images, called *frames* in this context, are concatenated along the fourth dimension. Multi-frame arrays are either  $m$ -by- $n$ -by-1-by- $p$ , for grayscale, binary, or indexed images, or  $m$ -by- $n$ -by-3-by- $p$ , for truecolor images, where  $p$  is the number of frames.

For example, a multi-frame array containing five, 480-by-640 grayscale or indexed images would be 480-by-640-by-1-by-5. An array with five 480-by-640 truecolor images would be 480-by-640-by-3-by-5.

---

**Note** To process a multi-frame array of grayscale images as an image sequence, as described in “Working with Image Sequences” on page 2-19, you can use the `squeeze` function to remove the singleton dimension.

---

You can use the `cat` command to create a multi-frame array. For example, the following stores a group of images (`A1`, `A2`, `A3`, `A4`, and `A5`) in a single array.

```
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.

## Image Arithmetic

Image arithmetic is the implementation of standard arithmetic operations, such as addition, subtraction, multiplication, and division, on images. Image arithmetic has many uses in image processing both as a preliminary step in more complex operations and by itself. For example, image subtraction can be used to detect differences between two or more images of the same scene or object.

You can do image arithmetic using the MATLAB arithmetic operators. The Image Processing Toolbox also includes a set of functions that implement arithmetic operations for all numeric, nonsparse data types. The toolbox arithmetic functions accept any numeric data type, including `uint8`, `uint16`, and `double`, and return the result image in the same format. The functions perform the operations in double precision, on an element-by-element basis, but do not convert images to double-precision values in the MATLAB workspace. Overflow is handled automatically. The functions saturate return values to fit the data type. For details, see “Image Arithmetic Saturation Rules” on page 2-25.

---

**Note** On Intel architecture processors, the image arithmetic functions can take advantage of the Intel Performance Primitives Library (IPPL), thus accelerating their execution time. IPPL is only activated, however, when the data passed to these functions is of specific classes. See the reference pages for the individual arithmetic functions for more information.

---

### Image Arithmetic Saturation Rules

The results of integer arithmetic can easily overflow the data type allotted for storage. For example, the maximum value you can store in `uint8` data is 255. Arithmetic operations can also result in fractional values, which cannot be represented using integer arrays.

MATLAB arithmetic operators and the Image Processing Toolbox arithmetic functions use these rules for integer arithmetic:

- Values that exceed the range of the integer type are saturated to that range.
- Fractional values are rounded.

For example, if the data type is `uint8`, results greater than 255 (including `Inf`) are set to 255. The following table lists some additional examples.

Result	Class	Truncated Value
300	<code>uint8</code>	255
-45	<code>uint8</code>	0
10.5	<code>uint8</code>	11

## Nesting Calls to Image Arithmetic Functions

You can use the image arithmetic functions in combination to perform a series of operations. For example, to calculate the average of two images,

$$C = \frac{A+B}{2}$$

You could enter

```
I = imread('rice.png');
I2 = imread('cameraman.tif');
K = imdivide(imadd(I,I2), 2); % not recommended
```

When used with `uint8` or `uint16` data, each arithmetic function rounds and saturates its result before passing it on to the next operation. This can significantly reduce the precision of the calculation. A better way to perform this calculation is to use the `imlincomb` function. `imlincomb` performs all the arithmetic operations in the linear combination in double precision and only rounds and saturates the final result.

```
K = imlincomb(.5,I,.5,I2); % recommended
```

# Reading and Writing Image Data

---

This chapter describes how to get information about the contents of a graphics file, read image data from a file, and write image data to a file, using standard graphics and medical file formats.

Getting Information About a Graphics File (p. 3-2)

Describes how to get information about the contents of a graphics file by reading the metadata contained in the file

Reading Image Data (p. 3-3)

Describes how to read image data from a file

Writing Image Data (p. 3-5)

Describes how to write image data to a file

Converting Graphics File Formats (p. 3-8)

Describes how to change the file format used to store an image

Reading and Writing Data in Medical File Formats (p. 3-9)

Describes how to import image data into the MATLAB workspace and write image data to graphics files

## Getting Information About a Graphics File

The `imfinfo` function enables you to obtain information about a graphics file and its contents. You can use `imfinfo` with any of the formats supported by MATLAB. Use the `imformats` function to determine which formats are supported.

---

**Note** You can also get information interactively about an image displayed in the Image Tool — see “Getting Information About an Image” on page 4-34.

---

The information returned by `imfinfo` depends on the file format, but it always includes at least the following:

- Name of the file
- 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: truecolor (RGB), grayscale (intensity), or indexed

See `imfinfo` for more information about getting information about graphics files. For information about adding support for a new file format, see `imformats`.



## Reading Image Data

The `imread` function reads an image from any supported graphics image file format, in any of the supported bit depths. Most image file formats use 8 bits to store pixel values. When these are read into memory, MATLAB stores them as class `uint8`. For file formats that support 16-bit data, PNG and TIFF, MATLAB stores the images as class `uint16`.

For example, this code reads a truecolor image into the MATLAB workspace as the variable `RGB`.

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

This code reads an indexed image with its associated colormap into the MATLAB workspace in two separate variables.

```
[X,map] = imread('trees.tif');
```

---

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

---

In these examples, `imread` infers the file format to use from the contents of the file. You can also specify the file format as an argument to `imread`. MATLAB supports many common graphics file formats, such as Microsoft Windows Bitmap (BMP), Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), Portable Network Graphics (PNG), and Tagged Image File Format (TIFF) formats. For the latest information concerning the bit depths and/or image formats supported, see the reference pages for the `imread` and `imformats` functions.

## Reading Multiple Images from a Graphics File

MATLAB supports several graphics file formats, such as HDF and TIFF, that can contain multiple images. By default, `imread` imports only the first image from a file. To import additional images from the file, use the syntax supported by the file format.

For example, when used with TIFF files, you can use an index value with `imread` that identifies the image in the file you want to import. This example reads a series of 27 images from a TIFF file and stores the images in a four-dimensional array. You can use `imfinfo` to determine how many images are stored in the file.

```
mri = uint8(zeros(128,128,1,27)); % preallocate 4-D array

for frame=1:27
    [mri(:,:,,frame),map] = imread('mri.tif',frame);
end
```

When a file contains multiple images that are related in some way, such as a time sequence, you can store the images in MATLAB as a 4-D array. All the images must be the same size. For more information, see “Working with Image Sequences” on page 2-19.

## Writing Image Data

The `imwrite` function writes an image to a graphics file in one of the supported formats. 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. (For more information, see the reference page for the `imwrite` function.)

This example loads the indexed image `X` from a MAT-file, `clown.mat`, that contains the data matrix and the associated colormap and then writes the image to a BMP file.

```
load clown
whos
      Name           Size           Bytes   Class
      X              200x320         512000  double array
      caption        2x1              4       char array
      map            81x3             1944   double array

Grand total is 64245 elements using 513948 bytes

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

### Specifying Additional Format-Specific Parameters

When using `imwrite` with some graphics formats, you can specify additional parameters. For example, with PNG files, you can specify the bit depth as an additional parameter. This example writes a grayscale image `I` to a 4-bit PNG file.

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

This example writes an image `A` to a JPEG file, using an additional parameter to specify the compression quality parameter.

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

For more information about the additional parameters associated with certain graphics formats, see the reference pages for `imwrite`.

## Reading and Writing Binary Images in 1-Bit Format

In certain file formats, a binary image can be stored in a 1-bit format. If the file format supports it, MATLAB writes binary images as 1-bit images by default. When you read in a binary image in 1-bit format, MATLAB represents it in the workspace as a logical array.

This example reads in a binary image and writes it as a TIFF file. Because the TIFF format supports 1-bit images, the file is written to disk in 1-bit format.

```
BW = imread('text.png');  
imwrite(BW,'test.tif');
```

To verify the bit depth of `test.tif`, call `imfinfo` and check the `BitDepth` field.

```
info = imfinfo('test.tif');  
  
info.BitDepth  
ans =  
  
1
```

---

**Note** When writing binary files, MATLAB sets the `ColorType` field to `'grayscale'`.

---

## Determining the Storage Class of the Output File

`imwrite` uses the following rules to determine the storage class used in the output image.

Storage Class of Image	Storage Class of Output Image File
logical	<p>If the output image file format specified supports 1-bit images, <code>imwrite</code> creates a 1-bit image file.</p> <p>If the output image file format specified does not support 1-bit images, <code>imwrite</code> converts the image to a class <code>uint8</code> grayscale image.</p>
uint8	<p>If the output image file format specified supports unsigned 8-bit images, <code>imwrite</code> creates an unsigned 8-bit image file.</p>
uint16	<p>If the output image file format specified supports unsigned 16-bit images (PNG or TIFF), <code>imwrite</code> creates an unsigned 16-bit image file.</p> <p>If the output image file format specified does not support 16-bit images, <code>imwrite</code> scales the image data to class <code>uint8</code> and creates an 8-bit image file.</p>
int16	Partially supported; depends on file format.
single	Partially supported; depends on file format.
double	MATLAB scales the image data to <code>uint8</code> and creates an 8-bit image file, because most image file formats use 8 bits.

## Converting Graphics File Formats

To change the graphics format of an image, use `imread` to import the image into the MATLAB workspace and then use the `imwrite` function to export the image, specifying the appropriate file format.

To illustrate, this example uses the `imread` function to read an image in bitmap (BMP) format into the workspace. The example then writes the bitmap image to a file using Portable Network Graphics (PNG) format.

```
bitmap = imread('mybitmap.bmp','bmp');  
imwrite(bitmap,'mybitmap.png','png');
```

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 pages for `imread` and `imwrite`.

## Reading and Writing Data in Medical File Formats

The Image Processing Toolbox includes support for working with image data in the following commonly used medical file formats:

- Digital Imaging and Communications in Medicine (DICOM) format
- Mayo Clinic Analyze 7.5 format
- Interfile format

Topics covered in this section include

- “Reading Metadata from a DICOM File” on page 3-9
- “Reading Image Data from a DICOM File” on page 3-10
- “Writing Image Data or Metadata to a DICOM File” on page 3-11, including an example that reads image data and metadata from a DICOM file, modifies the image data, and writes the modified data to a new DICOM file
- “Using the Mayo Analyze 7.5 Format” on page 3-16
- “Using the Interfile Format” on page 3-17

### Reading Metadata from a DICOM File

DICOM files contain metadata that provide information about the image data, such as the size, dimensions, bit depth, modality used to create the data, the equipment settings used to capture the image, and information about the study. The DICOM specification defines many of these metadata fields, but files can contain additional fields, called private metadata.

To read metadata from a DICOM file, use the `dicominfo` function. `dicominfo` returns the information in a MATLAB structure where every field contains a specific piece of DICOM metadata. You can use the metadata structure returned by `dicominfo` to specify the DICOM file you want to read using `dicomread` — see “Reading Image Data from a DICOM File” on page 3-10.

The following example reads the metadata from a sample DICOM file that is included with the toolbox.

```
info = dicominfo('CT-MON02-16-ankle.dcm')

info =

    Filename: [1x47 char]
    FileModDate: '24-Dec-2000 19:54:47'
    FileSize: 525436
    Format: 'DICOM'
    FormatVersion: 3
    Width: 512
    Height: 512
    BitDepth: 16
    ColorType: 'grayscale'
    SelectedFrames: []
    FileStruct: [1x1 struct]
    StartOfPixelData: 1140
    MetaElementGroupLength: 192
    FileMetaInformationVersion: [2x1 double]
    MediaStorageSOPClassUID: '1.2.840.10008.5.1.4.1.1.7'
    MediaStorageSOPInstanceUID: [1x50 char]
    TransferSyntaxUID: '1.2.840.10008.1.2'
    ImplementationClassUID: '1.2.840.113619.6.5'
    .
    .
    .
```

## Reading Image Data from a DICOM File

To read image data from a DICOM file, use the `dicomread` function. The `dicomread` function reads files that comply with the DICOM specification but can also read certain common noncomplying files.

When using `dicomread`, you can specify the filename as an argument, as in the following example. The example reads the sample DICOM file that is included with the toolbox.

```
I = dicomread('CT-MON02-16-ankle.dcm');
```



You can also use the metadata structure returned by `dicominfo` to specify the file you want to read, as in the following example.

```
info = dicominfo('CT-MONO2-16-ankle.dcm');  
I = dicomread(info);
```

### Viewing Images from DICOM Files

To view the image data imported from a DICOM file, use one of the toolbox image display functions `imshow` or `imshow_tool`. Note, however, that because the image data in this DICOM file is signed 16-bit data, you must use the autoscaling syntax with either display function to make the image viewable.

```
imshow(I, 'DisplayRange', [])
```



### Writing Image Data or Metadata to a DICOM File

To write image data or metadata to a file in DICOM format, use the `dicomwrite` function. This example writes the image `I` to the DICOM file `ankle.dcm`.

```
dicomwrite(I, 'h:\matlab\work\ankle.dcm')
```

### Writing Metadata with the Image Data

When writing image data to a DICOM file, `dicomwrite` automatically includes the minimum set of metadata fields required by the type of DICOM information object (IOD) you are creating. `dicomwrite` supports the following DICOM IODs with full validation.

- Secondary capture (default)
- Magnetic resonance
- Computed tomography

`dicomwrite` can write many other types of DICOM data (e.g. X-ray, radiotherapy, nuclear medicine) to a file; however, `dicomwrite` does not perform any validation of this data. See `dicomwrite` for more information.

You can also specify the metadata you want to write to the file by passing to `dicomwrite` an existing DICOM metadata structure that you retrieved using `dicominfo`. In the following example, the `dicomwrite` function writes the relevant information in the metadata structure `info` to the new DICOM file.

```
info = dicominfo('CT-MON02-16-ankle.dcm');  
I = dicomread(info);  
dicomwrite(I, 'h:\matlab\tmp\ankle.dcm', info)
```

Note that the metadata written to the file is not identical to the metadata in the `info` structure. When writing metadata to a file, there are certain fields that `dicomwrite` must update. To illustrate, look at the instance ID in the original metadata with the ID in the new file.

```
info.SOPInstanceUID  
ans =  
  
1.2.840.113619.2.1.2411.1031152382.365.1.736169244
```

Now, read the metadata from the newly created DICOM file, using `dicominfo`, and check the `SOPInstanceUID` field. Note that they contain different values.

```
info2 = dicominfo('h:\matlab\tmp\ankle.dcm');

info2.SOPInstanceUID

ans =

1.2.841.113411.2.1.2411.10311244477.365.1.63874544
```

### Removing Confidential Information from a DICOM File

When using a DICOM file as part of a training set, blinded study, or a presentation, you might want to remove confidential patient information, a process called *anonymizing* the file. To do this, use the `dicomanon` function.

The `dicomanon` function creates a new series with new study values, changes some of the metadata, and then writes the file. For example, you could replace steps 4, 5, and 6 in the example in “Example: Creating a New Series” on page 3-13 with a call to the `dicomanon` function.

### Example: Creating a New Series

When writing a modified image to a DICOM file, you might want to make the modified image the start of a new series. In the DICOM standard, images can be organized into series. When you write an image with metadata to a DICOM file, `dicomwrite` puts the image in the same series by default. To create a new series, you must assign a new DICOM unique identifier to the `SeriesInstanceUID` metadata field. The following example illustrates this process.

- 1 Read an image from a DICOM file into the MATLAB workspace.

```
I = dicomread('CT-MON02-16-ankle.dcm');
```

To view the image, use either of the toolbox display functions `imshow` or `imtool`. Because the DICOM image data is signed 16-bit data, you must use the autoscaling syntax.

```
imtool(I, 'DisplayRange', [])
```



- 2 Read the metadata from the same DICOM file.

```
info = dicominfo('CT-MONO2-16-ankle.dcm');
```

To identify the series an image belongs to, view the value of the SeriesInstanceUID field.

```
info.SeriesInstanceUID
```

```
ans =
```

```
1.2.840.113619.2.1.2411.1031152382.365.736169244
```

- 3 You typically only start a new DICOM series when you modify the image in some way. This example removes all the text from the image.

The example finds the maximum and minimum values of all pixels in the image. The pixels that form the white text characters are set to the maximum pixel value.

```
max(I(:))
```

```
ans =
```

```
4080
```

```
min(I(:))
```

```
ans =
```

```
32
```

To remove these text characters, the example sets all pixels with the maximum value to the minimum value.

```
Imodified = I;  
Imodified(Imodified == 4080) = 32;
```

View the processed image.

```
imshow(Imodified)
```



- 4 Generate a new DICOM unique identifier (UID) using the `dicomuid` function. You need a new UID to write the modified image as a new series.

```
uid = dicomuid
```

```
uid =
```

```
1.3.6.1.4.1.9590.100.1.1.56461980611264497732341403390561061497
```

`dicomuid` is guaranteed to generate a unique UID.

- 5 Set the value of the `SeriesInstanceUID` field in the metadata associated with the original DICOM file to the generated value.

```
info.SeriesInstanceUID = uid;
```

- 6 Write the modified image to a new DICOM file, specifying the modified metadata structure, `info`, as an argument. Because you set the `SeriesInstanceUID` value, the image you write is part of a new series.

```
dicomwrite(Imodified, 'ankle_newseries.dcm', info);
```

To verify this operation, view the image and the `SeriesInstanceUID` metadata field in the new file.

For information about the syntax variations that specify nondefault spatial coordinates, see the reference page for `imshow`.

## Using the Mayo Analyze 7.5 Format

Analyze 7.5 is a file format, developed by the Mayo Clinic, for storing MRI data. An Analyze 7.5 data set consists of two files:

- Header file (`filename.hdr`) — Provides information about dimensions, identification, and processing history. You use the `analyze75info` function to read the header information.
- Image file (`filename.img`) — Image data, whose data type and ordering are described by the header file. You use `analyze75read` to read the image data into the MATLAB workspace.

---

**Note** The Analyze 7.5 format uses the same dual-file data set organization and the same filename extensions as the Interfile format; however, the file formats are not interchangeable. To learn how to read data from an Interfile data set, see “Using the Interfile Format” on page 3-17.

---

The following example calls the `analyze75info` function to read the metadata from the Analyze 7.5 header file. The example then passes the `info` structure returned by `analyze75info` to the `analyze75read` function to read the image

data from the image file. The file used in the example can be downloaded from <http://www.radiology.uiowa.edu/downloads/>.

```
info = analyze75info('CT_HAND.hdr');  
X = analyze75read(info);
```

## Using the Interfile Format

Interfile is a file format that was developed for the exchange of nuclear medicine image data.

An Interfile data set consists of two files:

- Header file (`filename.hdr`) — Provides information about dimensions, identification and processing history. You use the `interfileinfo` function to read the header information.
- Image file (`filename.img`) — Image data, whose data type and ordering are described by the header file. You use `interfileread` to read the image data into the MATLAB workspace.

---

**Note** The Interfile format uses the same dual-file data set organization and the same filename extensions as the Analyze 7.5 format; however, the file formats are not interchangeable. To learn how to read data from an Analyze 7.5 data set, see “Using the Mayo Analyze 7.5 Format” on page 3-16.

---

The following example calls the `interfileinfo` function to read the metadata from the Interfile header file. The example then reads the image data from the corresponding image file in the Interfile data set. The file used in the example can be downloaded from <http://www.keston.com/Phantoms/>.

```
info = interfileinfo('dyna');  
X = interfileread('dyna');
```





# Displaying and Exploring Images

---

This chapter describes the image display and exploration tools provided by the Image Processing Toolbox.

Overview (p. 4-3)	Comparison of toolbox display functions
Using <code>imshow</code> to Display Images (p. 4-5)	How to use the <code>imshow</code> display function
Using the Image Tool to Explore Images (p. 4-9)	How to use the Image Tool integrated display and exploration environment
Using Image Tool Navigation Aids (p. 4-18)	Image Tool navigation aids including the Overview tool, panning, and zooming
Getting Information about the Pixels in an Image (p. 4-24)	Image Tool pixel information tools, including the Pixel Region tool and the Pixel Information tool
Measuring Features in an Image (p. 4-31)	Image Tool includes the Distance tool to measure regions in an image
Getting Information About an Image (p. 4-34)	Image Tool's Image Information tool
Adjusting the Contrast and Brightness of an Image (p. 4-36)	Image Tool's Adjust Contrast tool
Viewing Multiple Images (p. 4-47)	Using <code>imshow</code> and <code>imtool</code> to view multiple images

Displaying Different Image Types (p. 4-51)	Using <code>imshow</code> and <code>imtool</code> with each image type
Special Display Techniques (p. 4-58)	Using the <code>colorbar</code> , <code>montage</code> , and <code>warp</code> functions
Printing Images (p. 4-64)	Print images from <code>imshow</code> and the Image Tool
Setting Toolbox Display Preferences (p. 4-66)	Setting toolbox preferences

## Overview

The Image Processing Toolbox includes two display functions, `imshow` and `imtool`. Both functions work within the Handle Graphics architecture: they create an image object and display it in an axes object contained in a figure object. The toolbox functions automatically set the values of certain figure, axes, and image object properties to control how the image data is displayed — see “Understanding Handle Graphics Object Property Settings” on page 4-4.

`imshow` is the toolbox’s fundamental image display function. Use `imshow` when you want to display any of the different image types supported by the toolbox, such as grayscale (intensity), truecolor (RGB), binary, and indexed. For more information, see “Using `imshow` to Display Images” on page 4-5. The `imshow` function is also a key building block for image applications you might want to create using the toolbox modular tools. For more information, see Chapter 5, “Building GUIs with Modular Tools”.

The other toolbox display function, `imtool`, launches the Image Tool, which presents an integrated environment for displaying images and performing some common image processing tasks. The Image Tool provides all the image display capabilities of `imshow` but also provides access to several other tools for navigating and exploring images, such as scroll bars, the Pixel Region tool, the Image Information tool, and the Adjust Contrast tool. For more information, see “Using the Image Tool to Explore Images” on page 4-9.

In general, using the toolbox functions to display images is preferable to using the MATLAB image display functions `image` and `imagesc`. The toolbox functions are easier to use and are optimized for displaying images.

## Understanding Handle Graphics Object Property Settings

When you display an image, `imshow` and `imtool` set the Handle Graphics properties that control how the image is displayed. The following table lists the relevant properties and their settings for each image type. The table uses standard toolbox terminology to refer to the various image types: `X` represents an indexed image, `I` represents a grayscale image, `BW` represents a binary image, and `RGB` represents a truecolor image.

---

**Note** Both `imshow` and `imtool` can perform automatic scaling of image data. When called with the syntax `imshow(I, 'DisplayRange', [])`, and similarly for `imtool`, the functions set the axes `CLim` property to `[min(I(:)) max(I(:))]`. `CDataMapping` is always scaled for grayscale images, so that the value `min(I(:))` is displayed using the first colormap color, and the value `max(I(:))` is displayed using the last colormap color.

---

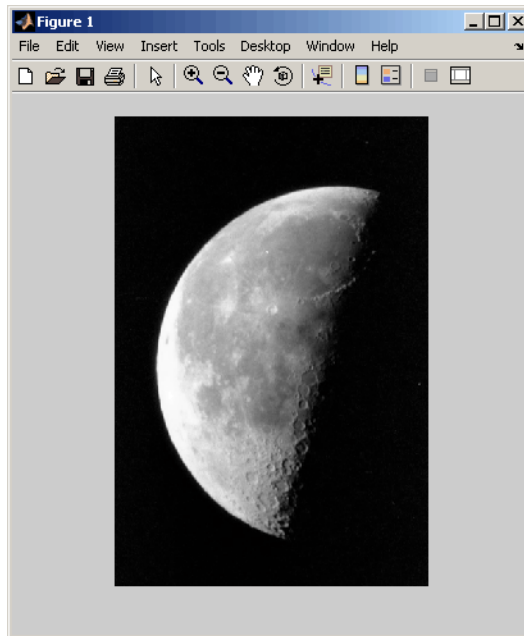
Handle Graphics Property	Indexed Images	Grayscale Images	Binary Images	Truecolor Images
<code>CData</code> (Image)	Set to the data in <code>X</code>	Set to the data in <code>I</code>	Set to data in <code>BW</code>	Set to data in <code>RGB</code>
<code>CDataMapping</code> (Image)	Set to 'direct'	Set to 'scaled'	Set to 'direct'	Ignored when <code>CData</code> is 3-D
<code>CLim</code> (Axes)	Does not apply	double: [0 1] uint8: [0 255] uint16: [0 65535]	Set to [0 1]	Ignored when <code>CData</code> is 3-D
<code>Colormap</code> (Figure)	Set to data in map	Set to grayscale colormap	Set to a grayscale colormap whose values range from black to white	Ignored when <code>CData</code> is 3-D

## Using imshow to Display Images

You can use the `imshow` function to display an image that has already been imported into the MATLAB workspace or to display an image stored in a graphics file. For example, this code reads an image into the MATLAB workspace and then displays it in a MATLAB figure window.

```
moon = imread('moon.tif');  
imshow(moon);
```

The `imshow` function displays the image in a MATLAB figure window, as shown in the following figure.



**Image Displayed in a Figure Window by imshow**

The `imshow` filename syntax

```
imshow('moon.tif');
```

can be 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, you must use the `getimage` function, which retrieves the image data from the current Handle Graphics image object. For example,

```
moon = getimage;
```

assigns the image data from `moon.tif` to the variable `moon` if the figure window in which it is displayed is currently active.

For more information about using `imshow`, see these additional topics.

- “Specifying the Initial Image Magnification” on page 4-6
- “Controlling the Appearance of the Figure” on page 4-7

For more information about using `imshow` to display the various image types supported by the toolbox, see “Displaying Different Image Types” on page 4-51.

### Specifying the Initial Image Magnification

By default, `imshow` attempts to display an image in its entirety at 100% magnification (one screen pixel for each image pixel). However, if an image is too large to fit in a figure window on the screen at 100% magnification, `imshow` scales the image to fit onto the screen and issues a warning message.

To override the default initial magnification behavior for a particular call to `imshow`, specify the `InitialMagnification` parameter. For example, to view an image at 150% magnification, use this code.

```
pout = imread('pout.tif');  
imshow(pout, 'InitialMagnification', 150)
```

`imshow` attempts to honor the magnification you specify. However, if the image does not fit on the screen at the specified magnification, `imshow` scales the image to fit and issues a warning message. You can also specify the text

string `'fit'` as the initial magnification value. In this case, `imshow` scales the image to fit the current size of the figure window.

You can also change the default initial magnification behavior of `imshow` by setting the `ImshowInitialMagnification` toolbox preference. To make this preference persist between sessions, include the command to set the preference in your `startup.m` file. To learn more about toolbox preferences, see “Setting the Values of Toolbox Preferences” on page 4-68.

When `imshow` scales an image, it uses interpolation to determine the values for screen pixels that do not directly correspond to elements in the image matrix. For more information, see “Interpolation” on page 6-3.

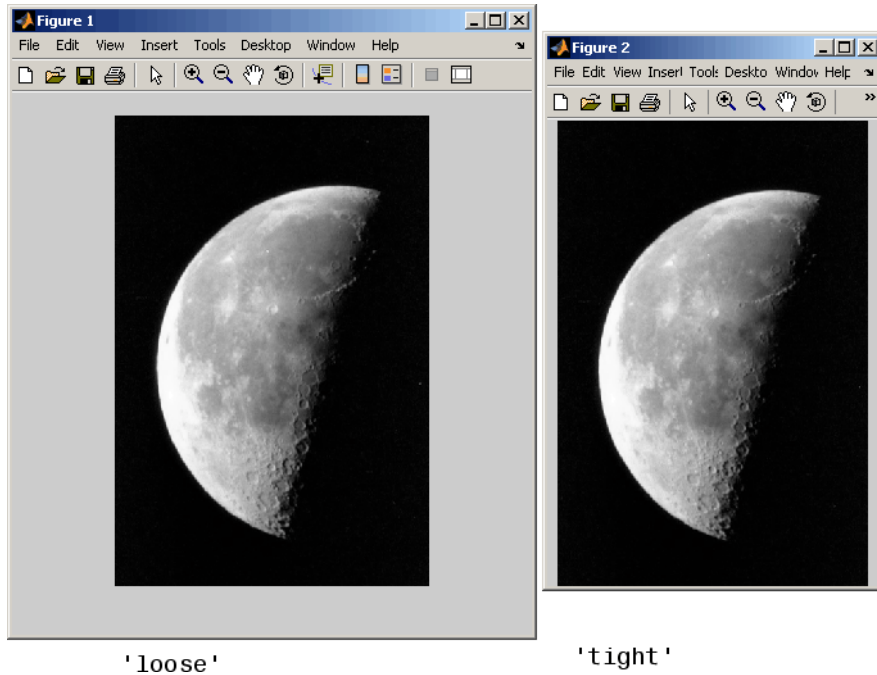
## Controlling the Appearance of the Figure

By default, when `imshow` displays an image in a figure, it surrounds the image with a gray border and does not include a visible axes box. If you want to display an image without the gray border or include a visible axes box with tick labels, you must set toolbox preferences. (For more information about setting toolbox preferences, see “Setting Toolbox Display Preferences” on page 4-66.)

For example, to display an image without a border, set the `ImshowBorder` preference to `'tight'`. By default, this preference is set to `'loose'`, which causes the border to be included. This code sets the preference to suppress the border and then displays an image.

```
iptsetpref('ImshowBorder','tight')
imshow('moon.tif')
```

The following figure shows the same image displayed with and without the border. Note that the image is the same size but the figure window takes up less space on your screen.



**Image Displayed With and Without a Border**



## Using the Image Tool to Explore Images

The Image Tool is an image display tool that also provides access to several other related tools, such as the Pixel Region tool, the Image Information tool, and the Adjust Contrast tool. The Image Tool also provides navigation aids that can help explore large images, such as scroll bars, the Overview tool, pan tool, and zoom buttons. The Image Tool presents an integrated environment for displaying images and performing common image processing tasks.

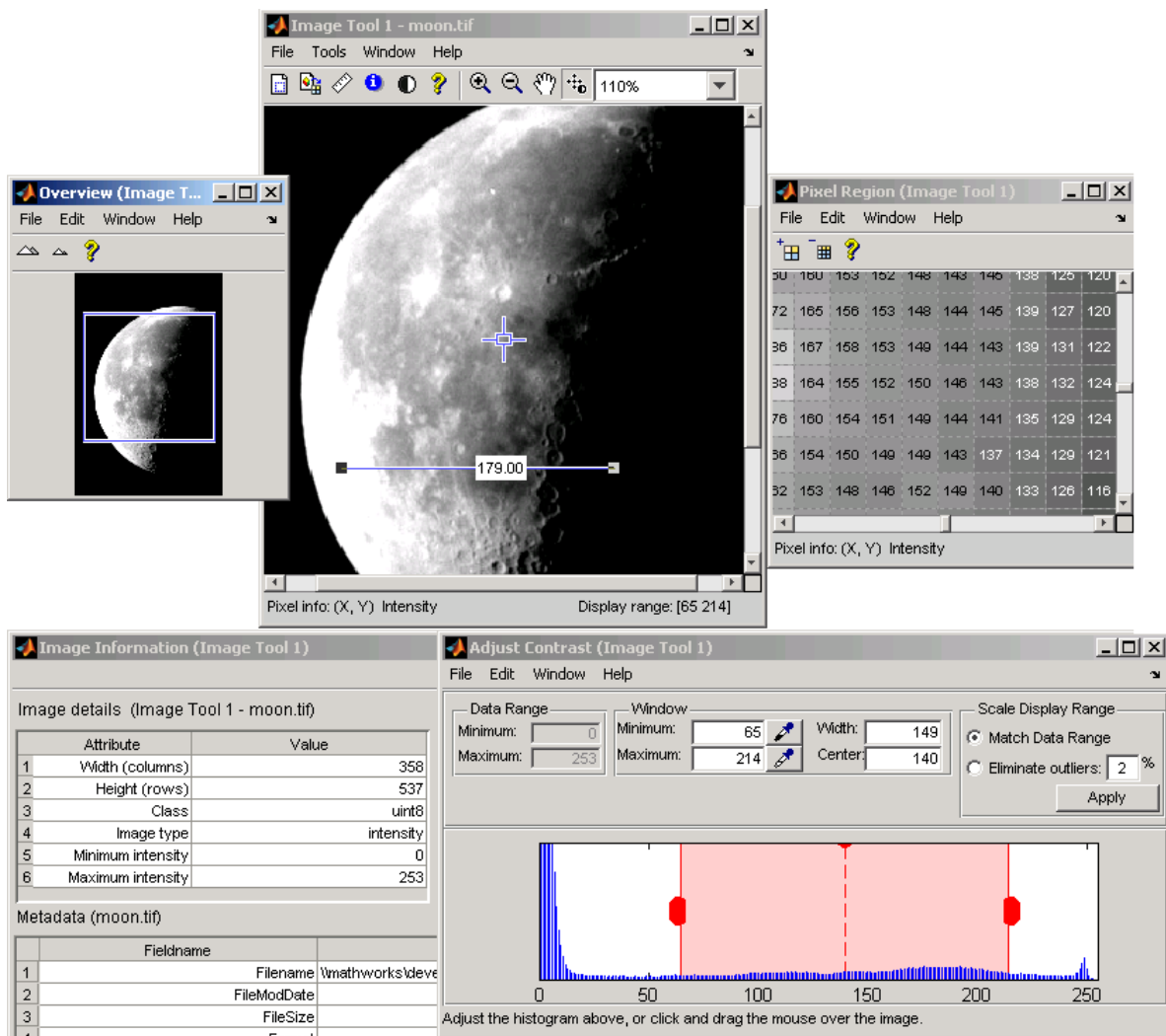
For example, this code reads the image from the file `moon.tif` and then displays it in the Image Tool.

```
imtool('moon.tif');
```

The following figure shows the image displayed in the Image Tool, with all of the related tools active. For more information about using the Image Tool and related tools, see the topics in the following list.

- “Opening the Image Tool” on page 4-11
- “Specifying the Initial Image Magnification” on page 4-12
- “Closing the Image Tool” on page 4-17
- “Specifying the Colormap” on page 4-13
- “Importing Image Data from the Workspace” on page 4-15
- “Exporting Image Data to the Workspace” on page 4-16
- “Closing the Image Tool” on page 4-17
- “Printing the Image in the Image Tool” on page 4-17
- “Using Image Tool Navigation Aids” on page 4-18
- “Getting Information about the Pixels in an Image” on page 4-24.
- “Measuring Features in an Image” on page 4-31
- “Getting Information About an Image” on page 4-34
- “Adjusting the Contrast and Brightness of an Image” on page 4-36
- “Displaying Different Image Types” on page 4-51

## 4 Displaying and Exploring Images



### Image Tool and Related Tools

## Opening the Image Tool

To start the Image Tool, use the `imtool` function. You can also start another Image Tool from within an existing Image Tool by using the **New** option from the **File** menu.

The `imtool` function supports many syntax options. For example, when called without any arguments, it opens an empty Image Tool.

```
imtool
```

To bring image data into this empty Image Tool, you can use either the **Open** or **Import from Workspace** options from the **File** menu — see “Importing Image Data from the Workspace” on page 4-15.

You can also specify the name of the MATLAB workspace variable that contains image data when you call `imtool`, as follows:

```
moon = imread('moon.tif');  
imtool(moon)
```

Alternatively, you can specify the name of the graphics file containing the image. This syntax can be useful for scanning through graphics files.

```
imtool('moon.tif');
```

---

**Note** When you use this syntax, the image data is not stored in a MATLAB workspace variable. To bring the image displayed in the Image Tool into the workspace, you must use the `getimage` function or the **Export from Workspace** option from the Image Tool **File** menu — see “Exporting Image Data to the Workspace” on page 4-16.

---

For more information about these syntax, see the `imtool` function reference page.

## Specifying the Initial Image Magnification

Like `imshow`, the `imshow` function attempts to display an image in its entirety at 100% magnification (one screen pixel for each image pixel). Unlike `imshow`, `imshow` always honors the specified numeric magnification, showing only a portion of the image if it is too big to fit in a figure on the screen and adding scroll bars to allow navigation to parts of the image that are not currently visible. If the specified magnification would make the image too large to fit on the screen, `imshow` scales the image to fit, without issuing a warning. This is the default behavior, specified by the `imshow` 'InitialMagnification' parameter value 'adaptive'.

To override this default initial magnification behavior for a particular call to `imshow`, specify the `InitialMagnification` parameter. For example, to view an image at 150% magnification, use this code.

```
pout = imread('pout.tif');  
imshow(pout, 'InitialMagnification', 150)
```

You can also specify the text string 'fit' as the initial magnification value. In this case, `imshow` scales the image to fit the default size of a figure window.

You can also change the default initial magnification behavior of `imshow` by setting the `ImshowInitialMagnification` toolbox preference. The magnification value you specify affects every call to `imshow` for the current MATLAB session. To make this preference persist between sessions, include the command to set the preference in your `startup.m` file. To learn more about toolbox preferences, see “Setting the Values of Toolbox Preferences” on page 4-68.

When `imshow` scales an image, it uses interpolation to determine the values for screen pixels that do not directly correspond to elements in the image matrix. For more information, see “Interpolation” on page 6-3.

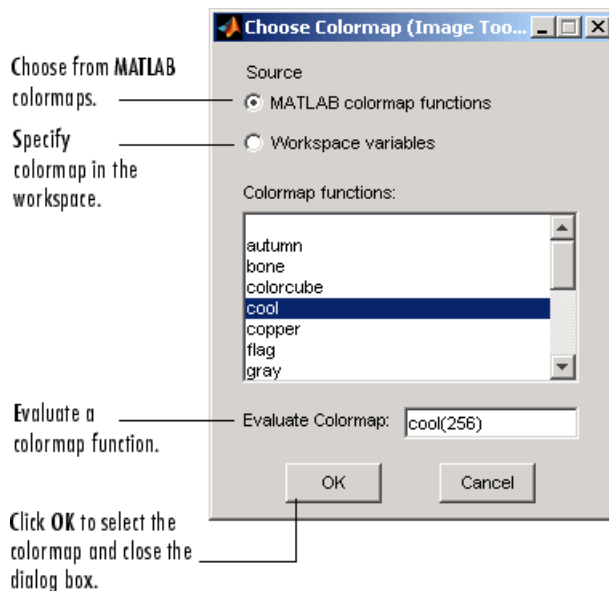
## Specifying the Colormap

A colormap is a matrix that can have any number of rows, but must have three columns. Each row in the colormap is interpreted as a color, with the first element specifying the intensity of red, the second green, and the third blue.

To specify the color map used to display an indexed image or a grayscale image in the Image Tool, select the **Choose Colormap** option on the **Tools** menu. This activates the Choose Colormap tool, shown below. Using this tool you can select one of the MATLAB colormaps or select a colormap variable from the MATLAB workspace.

When you select a colormap, the Image Tool executes the colormap function you specify and updates the image displayed. You can edit the colormap command in the **Evaluate Colormap** text box; for example, you can change the number of entries in the colormap (default is 256). You can enter your own colormap function in this field. Press **Enter** to execute the command.

When you choose a colormap, the image updates to use the new map. If you click **OK**, the Image Tool applies the colormap and closes the Choose Colormap tool. If you click **Cancel**, the image reverts to the previous colormap.

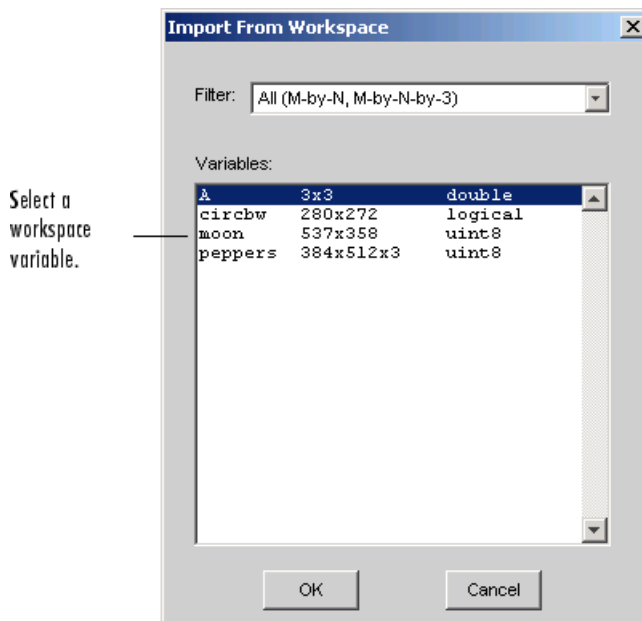


### Choose Colormap Tool

## Importing Image Data from the Workspace

To import image data from the MATLAB workspace into the Image Tool, use the **Import from Workspace** option on the Image Tool **File** menu. In the dialog box, shown below, you select the workspace variable that you want to import into the workspace.

The following figure shows the Import from Workspace dialog box. You can use the Filter menu to limit the images included in the list to certain image types, i.e., binary, indexed, intensity (grayscale), or truecolor.

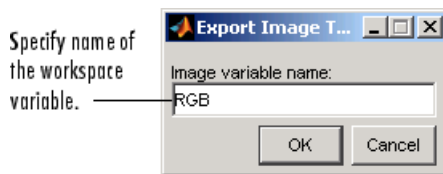


**Import from Workspace Dialog Box**

### Exporting Image Data to the Workspace

To export the image displayed in the Image Tool to the MATLAB workspace, you can use the **Export to Workspace** option on the Image Tool **File** menu. In the dialog box, shown below, you specify the name you want to assign to the variable in the workspace. By default, the Image Tool prefills the variable name field with `BW`, for binary images, `RGB`, for truecolor images, and `I` for grayscale or indexed images.

If the Image Tool contains an indexed image, this dialog box also contains a field where you can specify the name of the associated colormap.



#### Export Image to Workspace Dialog Box

### Using the `getimage` Function to Export Image Data

You can also use the `getimage` function to bring image data from the Image Tool into the MATLAB workspace.

The `getimage` function retrieves the image data (`CData`) from the current Handle Graphics image object. Because, by default, the Image Tool does not make handles to objects visible, you must use the toolbox function `imgca` to get a handle to the image axes displayed in the Image Tool. For example,

```
moon = getimage(imgca);
```

assigns the image data from `moon.tif` to the variable `moon` if the figure window in which it is displayed is currently active.



## Closing the Image Tool

To close the Image Tool window, use the **Close** button in the window title bar or select the **Close** option from the Image Tool **File** menu. You can also use the `imtool` function to return a handle to the Image Tool and use the handle to close the Image Tool. When you close the Image Tool, any related tools that are currently open also close.

Because the Image Tool does not make the handles to its figure objects visible, the Image Tool does not close when you call the MATLAB `close all` command. If you want to close multiple Image Tools, use the syntax

```
imtool close all
```

or select **Close all** from the Image Tool **File** menu.

## Printing the Image in the Image Tool

To print the image displayed in the Image Tool, select the **Print to Figure** option from the **File** menu. The Image Tool opens another figure window and displays the image. Use the **Print** option on the **File** menu of this figure window to print the image. See “Printing Images” on page 4-64 for more information.

# Using Image Tool Navigation Aids

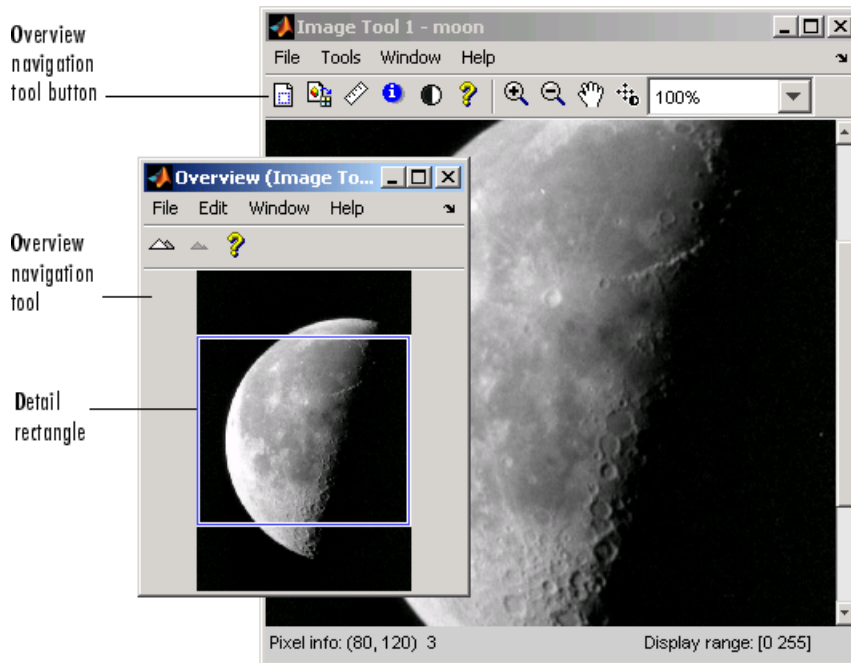
If an image is large or viewed at a large magnification, the Image Tool displays only a portion of the entire image. When this occurs, the Image Tool includes scroll bars to allow navigation around the image. In some cases, scroll bars might not be sufficient. To help navigate large images, the Image Tool includes the following navigation aids:

- Overview tool — Provides a view of the entire image to help you understand which portion of the image is currently displayed in the Image Tool. See “Overview Navigation” on page 4-18 for more information.
- Pan tool — Lets you click and grab the image displayed and move it in the Image Tool. See “Panning the Image Displayed in the Image Tool” on page 4-21 for more information.
- Zoom tools — Lets you zoom in or out on the image. See “Zooming In and Out on an Image” on page 4-22 for more information.
- Magnification Box — Lets you specify the exact magnification of the image. See “Specifying the Magnification of the Image” on page 4-22 for more information.

## Overview Navigation

To get an overview of the image displayed in the Image Tool, use the Overview tool. The Overview tool displays a view of the entire image, scaled to fit, in a separate window. Superimposed over this view of the image is a rectangle, called the *detail rectangle*. The detail rectangle shows which part of the image is currently visible in the Image Tool window. You can change the portion of the image visible in the Image Tool by moving the detail rectangle over the image in the Overview tool.

The following figure shows the Image Tool with the Overview tool.



### Image Tool with Overview Tool


The following sections provide more information about using the Overview tool.

- “Starting the Overview Tool” on page 4-19
- “Using the Overview Tool” on page 4-20
- “Specifying the Color of the Detail Rectangle” on page 4-20
- “Getting the Position and Size of the Detail Rectangle” on page 4-20
- “Printing the View of the Image in the Overview Tool” on page 4-21

### Starting the Overview Tool


The Overview tool starts automatically when you start the Image Tool. For example, execute the following command.

```
imtool('moon.tif')
```


You can also start the Overview tool by clicking the **Overview** button  in the Image Tool toolbar or by selecting the **Overview** option from the **Tools** menu in the Image Tool.

### Using the Overview Tool

To use the Overview tool to explore an image displayed in the Image Tool, follow this procedure:

- 1 Start the Overview tool by clicking the **Overview** button  in the Image Tool toolbar or by selecting **Overview** from the **Tools** menu. The Overview tool opens in a separate window containing a view of the entire image, scaled to fit.

The Image Tool opens the Overview tool, by default. If the Overview tool is already active, clicking the **Overview** button brings the tool to the front of the windows open on your screen.

- 2 Using the mouse, move the cursor into the detail rectangle. The cursor changes to the fleur shape, .
- 3 Press and hold the mouse button to drag the detail rectangle anywhere on the image. The Image Tool updates the view of the image to make the specified region visible.

### Specifying the Color of the Detail Rectangle

By default, the color of the detail rectangle in the Overview tool is blue. You might want to change the color of the rectangle to achieve better contrast with the predominant color of the underlying image. To do this, right-click anywhere inside the boundary of the detail rectangle and select a color from the **Set Rectangle Color** option on the context menu.

### Getting the Position and Size of the Detail Rectangle

To get the current position and size of the detail rectangle, right-click anywhere inside it and select **Copy Position** from the context menu. You can also access this option from the **Edit** menu of the Overview tool.

This option copies the position information to the clipboard. The position information is a vector of the form [xmin ymin width height]. To paste

this position vector into the MATLAB workspace or another application, right-click and select **Paste** from the context menu.




### **Printing the View of the Image in the Overview Tool**

You can print the view of the image displayed in the Overview tool. Select the **Print to Figure** option from the Overview tool **File** menu. See “Printing Images” on page 4-64 for more information.

### **Panning the Image Displayed in the Image Tool**

To change the portion of the image displayed in the Image Tool, you can use the Pan tool to move the image displayed in the window. This is called *panning* the image.

To pan an image displayed in the Image Tool,

- 1** Click the **Pan** tool button  in the toolbar or select **Pan** from the **Tools** menu. When the Pan tool is active, a checkmark appears next to the Pan selection in the menu.
- 2** Move the cursor over the image in the Image Tool, using the mouse. The cursor changes to an open-hand shape .
- 3** Press and hold the mouse button and drag the image in the Image Tool. When you drag the image, the cursor changes to the closed-hand shape .
- 4** To turn off panning, click the Pan tool button again or click the **Pan** option in the **Tools** menu.

---

**Note** As you pan the image in the Image Tool, the Overview tool updates the position of the detail rectangle — see “Overview Navigation” on page 4-18.

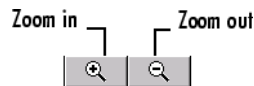
---

### Zooming In and Out on an Image

To enlarge an image to get a closer look or shrink an image to see the whole image in context, use the Zoom buttons on the toolbar. (You can also zoom in or out on an image by changing the magnification — see “Specifying the Magnification of the Image” on page 4-22.)

To zoom in or zoom out on an image,

- 1 Click the appropriate magnifying glass button in the Image Tool toolbar or select the **Zoom In** or **Zoom Out** option in the **Tools** menu. When the Zoom tool is active, a checkmark appears next to the appropriate Zoom selection in the menu.



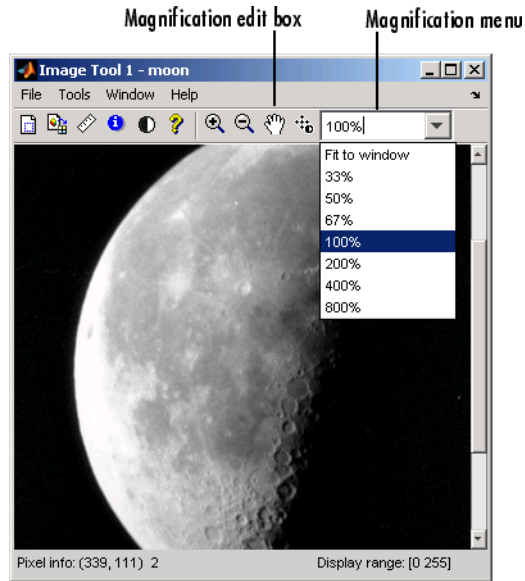
- 2 Move the cursor over the image you want to zoom in or out on, using the mouse. The cursor changes to the appropriate magnifying glass icon. With each click, the Image Tool changes the magnification of the image, centering the new view of the image on the spot where you clicked.

When you zoom in or out on an image, the magnification value displayed in the magnification edit box changes and the **Overview** window updates the position of the detail rectangle.

- 3 To leave zoom mode, click the active zoom button again to deselect it or click the **Zoom** option in the **Tools** menu.

### Specifying the Magnification of the Image

To enlarge an image to get a closer look or to shrink an image to see the whole image in context, you can use the magnification edit box, shown in the following figure. (You can also use the Zoom buttons to enlarge or shrink an image. See “Zooming In and Out on an Image” on page 4-22 for more information.)



### Image Tool Magnification Edit Box and Menu

To change the magnification of an image,

- 1 Move the cursor into the magnification edit box. The cursor changes to the text entry cursor.
- 2 Type a new value in the magnification edit box and press **Enter**. The Image Tool changes the magnification of the image and displays the new view in the window.

You can also specify a magnification by clicking the menu associated with the magnification edit box and selecting from a list of preset magnifications. If you choose the **Fit to Window** option, the Image Tool scales the image so that the entire image is visible.

### Getting Information about the Pixels in an Image

Often, you need to get information about the pixels in an image, such as their location and value. The Image Tool provides several ways to get this information, including:

- Pixel Information tool — Displays the location and value of the pixel under the cursor in the Image Tool window. See “Determining the Value of Individual Pixels” on page 4-24 for more information.
- Display Range tool — Displays the display range of the image in the Image Tool window. See “Getting the Display Range of an Image” on page 4-26 for more information.
- Pixel Region tool — Displays an extreme close-up view of the pixels in a specific region of an image. See “Viewing Pixel Values with the Pixel Region Tool” on page 4-27 for more information.

#### Determining the Value of Individual Pixels

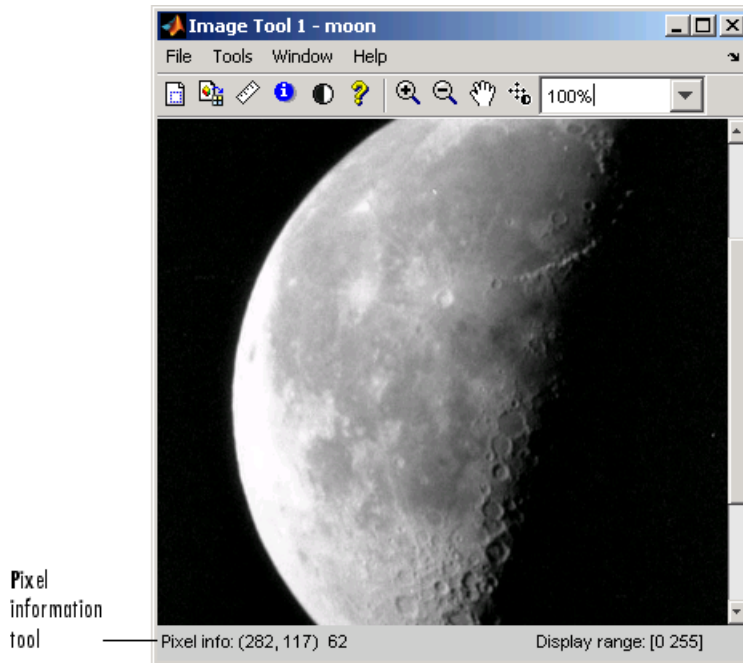
The Image Tool provides information about the location and value of individual pixels in an image. This information is displayed in the Pixel Information tool at the bottom left corner of the Image Tool window. The pixel value and location information represent the pixel under the current location of the cursor. The Image Tool updates this information as you move the cursor over the image.

For example, view an image in the Image Tool.

```
imtool('moon.tif')
```



The following figure shows the Image Tool with pixel location and value displayed in the Pixel Information tool. For more information, see “Saving the Pixel Value and Location Information” on page 4-25.



### Pixel Information in Image Tool

#### Saving the Pixel Value and Location Information

To save the pixel location and value information displayed, right-click a pixel in the image and choose the **Copy pixel info** option. The Image Tool copies the  $x$ - and  $y$ -coordinates and the pixel value to the clipboard.

To paste this position vector into the MATLAB workspace or another application, right-click and select **Paste** from the context menu.

### Getting the Display Range of an Image

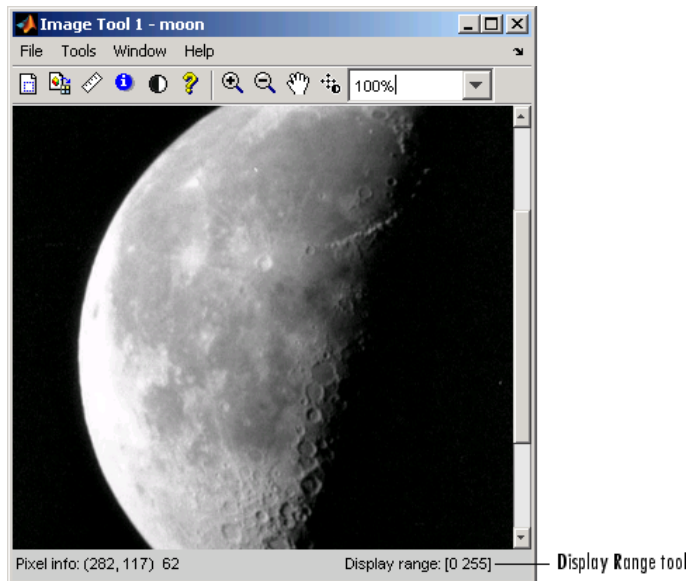
The Image Tool provides information about the display range of pixels in a grayscale image. The display range is the value of the axes `CLim` property, which controls the mapping of image `CData` to the figure colormap. `CLim` is a two-element vector `[cmin cmax]` specifying the `CData` value to map to the first color in the colormap (`cmin`) and the `CData` value to map to the last color in the colormap (`cmax`). Data values in between are linearly scaled.

The Image Tool displays this information in the Display Range tool at the bottom right corner of the window. The Image Tool does not show the display range for indexed, truecolor, or binary images.

For example, view an image in the Image Tool.

```
imtool('moon.tif')
```

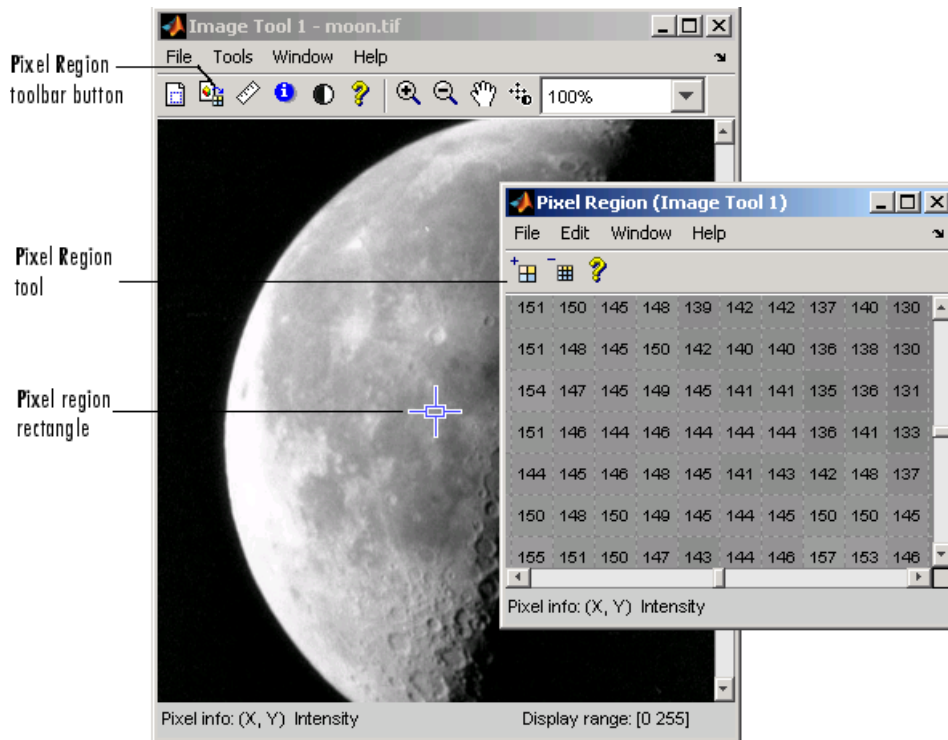
The following figure shows the Image Tool displaying the image with display range information.



### Display Range Information in Image Tool

## Viewing Pixel Values with the Pixel Region Tool

To view the values of pixels in a specific region of an image displayed in the Image Tool, use the Pixel Region tool. The Pixel Region tool superimposes a rectangle, called the *pixel region rectangle*, over the image displayed in the Image Tool. This rectangle defines the group of pixels that are displayed, in extreme close-up view, in the Pixel Region tool window. The following figure shows the Image Tool with the Pixel Region tool.



### Image Tool with Pixel Region Tool and Pixel Region Rectangle

The following sections provide more information about using the Pixel Region tool.

- “Starting the Pixel Region Tool” on page 4-28
- “Selecting a Region” on page 4-28



- “Customizing the View” on page 4-29
- “Determining the Location of the Pixel Region Rectangle” on page 4-29
- “Printing the View of the Image in the Pixel Region Tool” on page 4-30

### Starting the Pixel Region Tool

To start the Pixel Region tool, click the **Pixel Region** button  in the Image Tool toolbar or by selecting the **Pixel Region** option from the **Tools** menu in the Image Tool.

### Selecting a Region


To examine pixels in specific regions of an image, use the Pixel Region rectangle, as follows:

- 1 Start the Pixel Region tool by clicking the **Pixel Region** button  in the Image Tool toolbar or by selecting the **Pixel Region** option from the **Tools** menu. The Image Tool displays the pixel region rectangle  in the center of the target image and opens the Pixel Region tool.

---

**Note** Scrolling the image can move the pixel region rectangle off the part of the image that is currently displayed. To bring the pixel region rectangle back to the center of the part of the image that is currently visible, click the Pixel Region button again. For help finding the Pixel Region tool in large images, see “Determining the Location of the Pixel Region Rectangle” on page 4-29.

---

- 2 Using the mouse, position the pointer over the pixel region rectangle. The pointer changes to the fleur shape, .
- 3 Click the left mouse button and drag the pixel region rectangle to any part of the image. As you move the pixel region rectangle over the image, the Pixel Region tool updates the pixel values displayed. You can also move the pixel region rectangle by moving the scroll bars in the Pixel Region tool window.

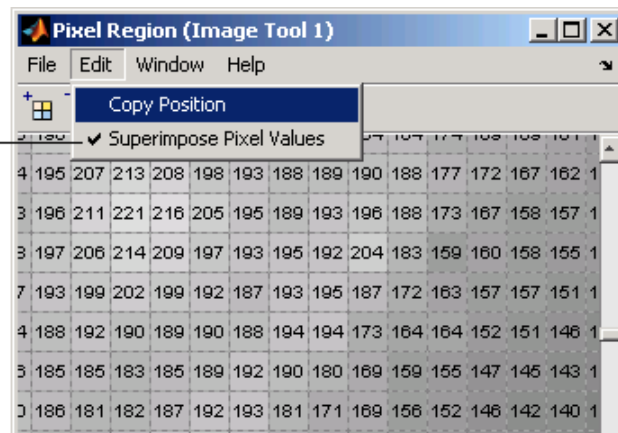
## Customizing the View

The pixel region rectangle defines the group of pixels that are displayed in the Pixel Region tool. To view a larger region, grab any side of the Pixel Region tool figure window and resize it, or use the zoom tools in the Pixel Region toolbar to zoom in or out on the image.

The Pixel Region tool displays the pixels at high magnification, overlaying each pixel with its numeric value. For RGB images, this information includes three numeric values, one for each band of the image. For indexed images, this information includes the index value and the associated RGB value.

If you would rather not see the numeric values in the display, go to the Pixel Region tool **Edit** menu and clear the **Superimpose Pixel Values** option.

Deselect to  
suppress pixel  
value display.



### Pixel Region Tool Edit Menu

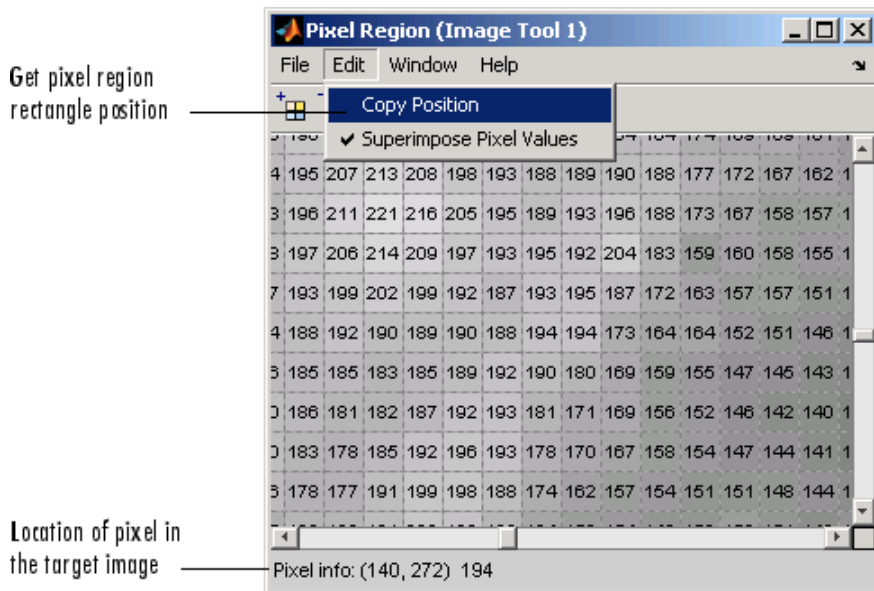
## Determining the Location of the Pixel Region Rectangle

To determine the current location of the pixel region in the target image, you can use the pixel information given at the bottom of the tool. This information includes the  $x$ - and  $y$ -coordinates of pixels in the target image coordinate system. When you move the pixel region rectangle over the target image, the pixel information given at the bottom of the tool is not updated until you move the cursor back over the Pixel Region tool.

You can also retrieve the current position of the pixel region rectangle by selecting the **Copy Position** option from the Pixel Region tool **Edit** menu. This option copies the position information to the clipboard. The position information is a vector of the form [xmin ymin width height].

To paste this position vector into the MATLAB workspace or another application, right-click and select **Paste** from the context menu.

The following figure shows these components of the Pixel Region tool.



### Pixel Region Rectangle Location Information

### Printing the View of the Image in the Pixel Region Tool

You can print the view of the image displayed in the Pixel Region tool. Select the **Print to Figure** option from the Pixel Region tool **File** menu. See “Printing Images” on page 4-64 for more information.

## Measuring Features in an Image

This section describes how to use the Distance tool to calculate the Euclidean distance between two points in an image displayed in the Image Tool. Topics covered include:


- “Using the Distance Tool” on page 4-31
- “Exporting Endpoint and Distance Data” on page 4-32
- “Customizing the Appearance of the Distance Tool” on page 4-33

### Using the Distance Tool

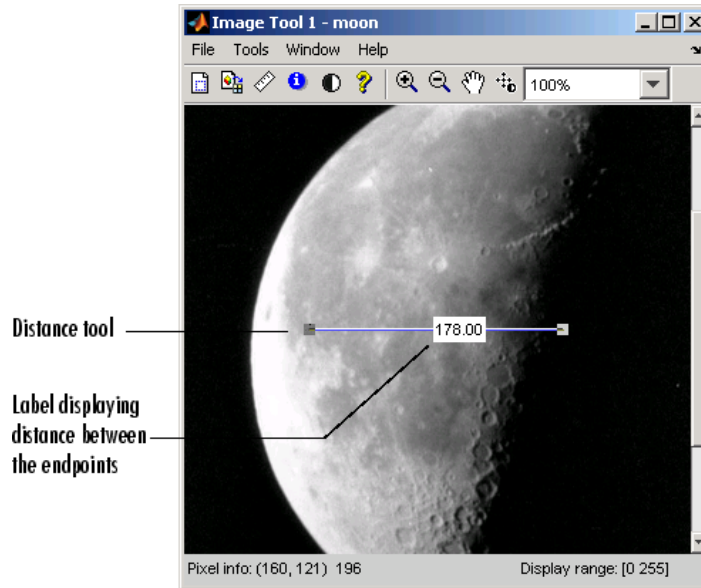
To use the Distance tool, follow this procedure.

- 1 Display an image in the Image Tool.

```
imtool('moon.tif')
```

- 2 Click the **Distance** tool button  in the Image Tool toolbar or select **Distance Tool** from the **Tools** menu. The Distance tool appears as a horizontal line displayed over the image, as shown in the following figure.

The Distance tool displays the distance between the two endpoints of the line in a label superimposed over the line. The tools specifies the distance in data units determined by the XData and YData properties, which is pixels, by default.

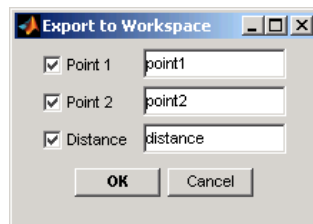


- Using the mouse, you can move the Distance tool over the image or, by grabbing either one of its endpoints, resize the tool.

### Exporting Endpoint and Distance Data

To save the endpoint locations and distance information, right-click the Distance tool and choose the **Copy pixel info** option from the context menu.

The Distance tool opens the Export to Workspace dialog box. You can use this dialog box to specify the names of the variables used to store this information.





After you click **OK**, the Distance tool creates the variables in the workspace, as in the following example.

whos			
Name	Size	Bytes	Class
distance	1x1	8	double array
moon	537x358	192246	uint8 array
point1	1x2	16	double array
point2	1x2	16	double array

## Customizing the Appearance of the Distance Tool

Using the Distance tool context menu, you can customize many aspects of the Distance tool appearance and behavior, including:

- Toggling the distance tool label on and off using the **Show Distance Label** option.
- Changing the color used to display the Distance tool line using the **Set line color** option.
- Constraining movement of the tool to either horizontal or vertical using the **Constrain drag** option.
- Deleting the distance tool object using the **Delete** option.

Right-click the Distance tool to access this context menu.

### Getting Information About an Image

To get information about the image displayed in the Image Tool, use the Image Information tool. The Image Information tool can provide two types of information about an image:

- Basic information — Includes width, height, class, and image type. For grayscale and indexed images, this information also includes the minimum and maximum intensity values.
- Image metadata — Displays all the metadata from the graphics file that contains the image. This is the same information returned by the `imfinfo` function or the `dicominfo` function.


---

**Note** The Image Information tool can display image metadata only when you specify the filename containing the image to Image Tool, e.g., `imtool('moon.tif')`.

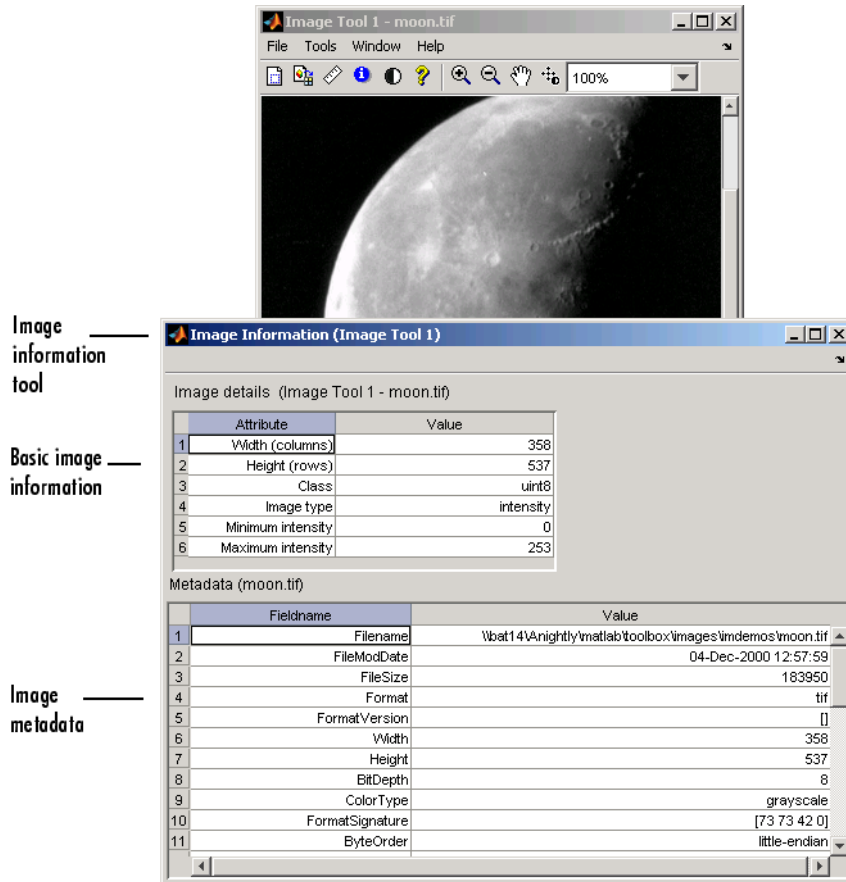
---

For example, view an image in the Image Tool.

```
imtool('moon.tif')
```

Start the Image Information tool by clicking the Image Information button  in the Image Tool toolbar or by selecting the **Image Information** option from the **Tools** menu in the Image Tool.

The following figure shows the Image Tool with the Image Information tool. In the figure, the Image Information tool displays both basic image information and image metadata because a file name was specified with `imtool`.



**Image Tool with Image Information Tool**

# Adjusting the Contrast and Brightness of an Image

To adjust the contrast and brightness of the image displayed in the Image Tool, use the Adjust Contrast tool.

When you start the Adjust Contrast tool, it opens a separate window containing a histogram of the image displayed in the Image Tool. The histogram shows the *data range* of the image and the display range of the image. The data range is the range of intensity values actually used in the image. The display range is the black-to-white mapping used to display the image, which is determined by the image class. The Adjust Contrast tool works by manipulating the display range; the data range of the image remains constant.

For example, in the following figure, the histogram for the image shows that the data range of the image is 74 to 224 and the display range is the default display range for the uint8 class, 0 to 255. Over this histogram, the Adjust Contrast tool overlays a red-tinted rectangular box, called a *window*. By changing the size of this window, you can modify the display range of the image and improve its contrast and brightness.

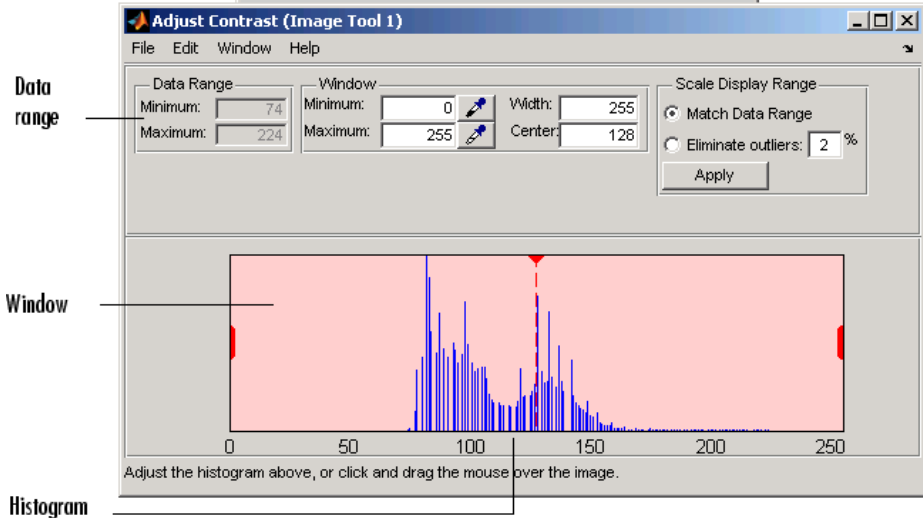
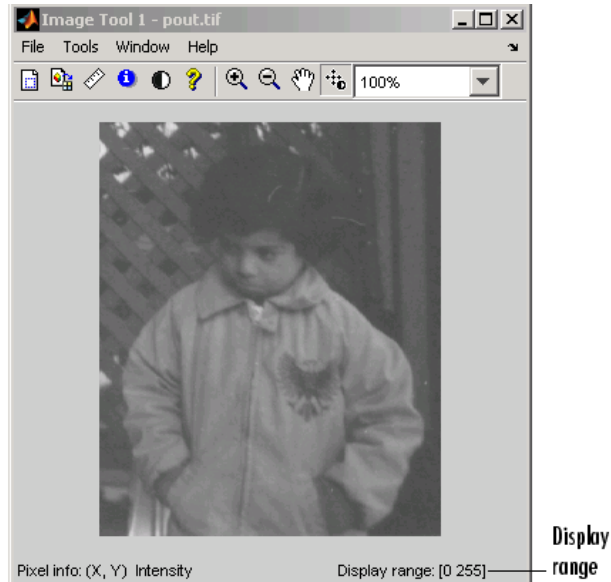
---

**Note** The Adjust Contrast tool just affects the display of the image; it does not change the values of pixels in the image. To change the intensity values and create a new output image, use `imadjust`.

---

For more information about using the Adjust Contrast tool, see these additional topics:

- “Using the Adjust Contrast Tool” on page 4-38
- “Example: Adjusting Contrast and Brightness” on page 4-40
- “Using the Window/Level Tool” on page 4-43
- “Understanding Contrast Adjustment” on page 4-45



**Image Tool with Adjust Contrast Tool**

### Using the Adjust Contrast Tool

This section describes how to use the Adjust Contrast tool. Topics covered include:

- “Starting the Adjust Contrast Tool” on page 4-38
- “Changing the Size of the Adjust Contrast Tool Window” on page 4-39

---

**Note** This section describes how to use the Adjust Contrast tool in the Image Tool. You can also use the Adjust Contrast tool independent of the Image Tool by calling the `imcontrast` function. See Chapter 5, “Building GUIs with Modular Tools” for more information.


---


#### Starting the Adjust Contrast Tool

To start the Adjust Contrast tool, follow this procedure:

- 1 View an image in the Image Tool.

```
imtool('pout.tif')
```

- 2 Click the **Adjust Contrast** button  in the Image Tool toolbar, or select the **Adjust Contrast** option from the Image Tool **Tools** menu.

When you start the Adjust Contrast tool, the Image Tool also activates the Window/Level tool, changing the cursor to the Window/Level cursor . The Window/Level tool provides another way to adjust contrast and brightness using the mouse — see “Using the Window/Level Tool” on page 4-43.

---

**Note** When you close the Adjust Contrast tool, the Window/Level tool remains active. To turn off the Window/Level tool, click the Window/Level button or one of the navigation buttons in the Image Tool toolbar.

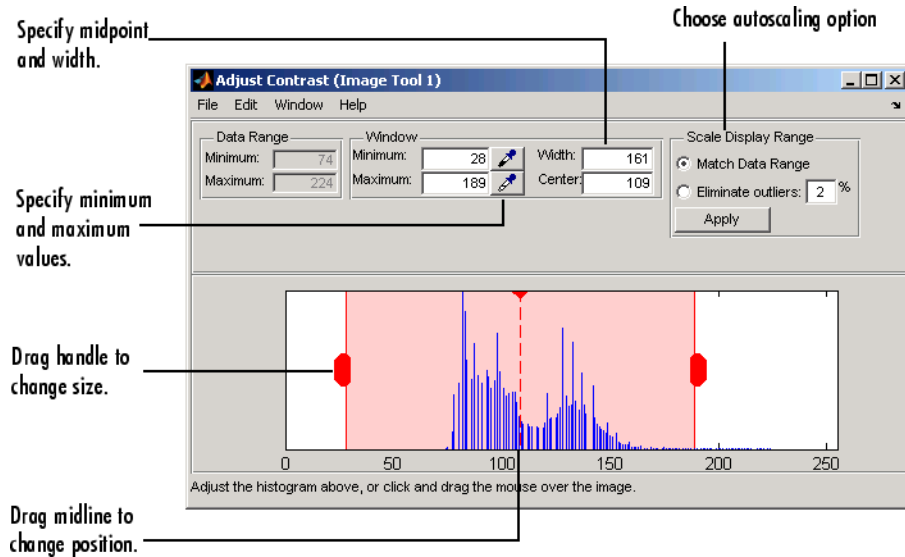
---

## Changing the Size of the Adjust Contrast Tool Window

You adjust the contrast and brightness of the displayed image by manipulating the window over the histogram in the Adjust Contrast tool. The tool provides several ways that you can modify the size and position of the window interactively:

- By grabbing one of the red handles on the right and left edges of the window and dragging it. You can also change the position of the window by grabbing the center line and dragging the window to the right or left.
- By specifying the size and position of the window in the **Minimum** and **Maximum** fields. You can also define these values by clicking the dropper button associated with these fields. When you do this, the cursor becomes an eye dropper shape. Position this cursor over the pixel in the image that you want to be the minimum (or maximum) value and click the mouse button.
- By specifying the size and position of the window in the **Width** and **Center** fields.
- By automatically scaling the display range to match the image data range. For example, with the `pout.tif` image, if you select the **Match data range** option, the window changes from the default display range (0 to 255) to the data range of the image (74 to 224).
- By automatically trimming outliers at the top and bottom of the image data range. If you select the **Eliminate outliers** option, the Adjust Contrast tool removes the top 1% and the bottom 1%, but you can specify other percentages. When you specify a percentage, the Adjust Contrast tool applies half the percentage to the top and half to the bottom. (You can perform this same operation from the command line using the `stretchlim` function.)

The following figure shows the Adjust Contrast tool after some interactive contrast adjustments.




### Example: Adjusting Contrast and Brightness

This example shows how to use the Adjust Contrast tool to change how pixel values display as black and white.

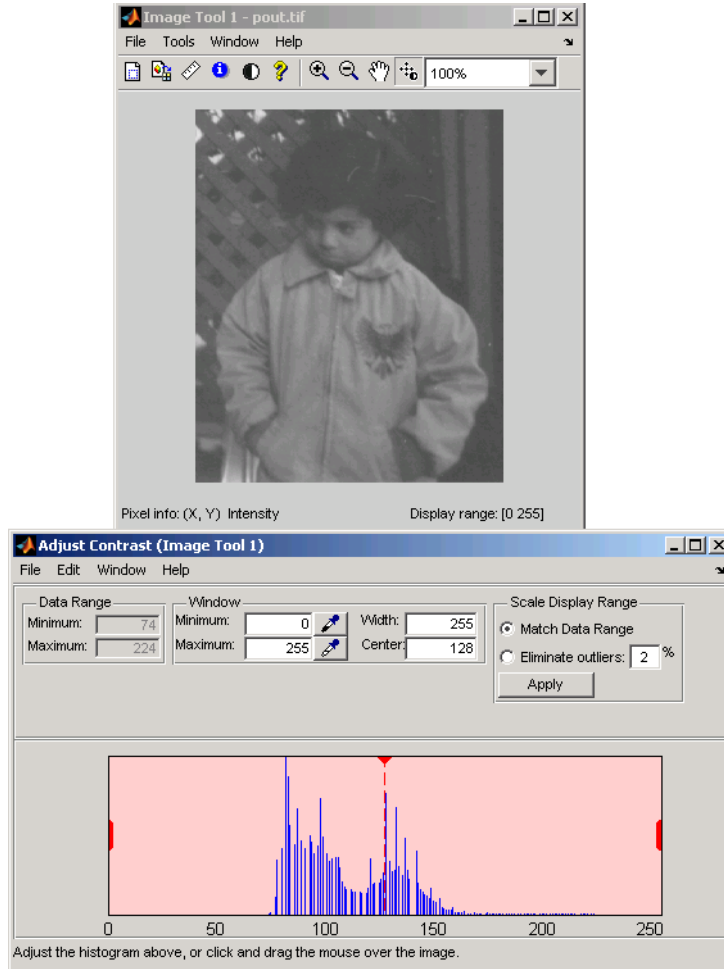
- 1 View an image in the Image Tool. This example opens the image `pout.tif`, which is a low-contrast image.

```
imtool('pout.tif')
```

- 2 Start the Adjust Contrast tool by clicking the **Adjust Contrast** button  in the Image Tool toolbar, or by selecting **Adjust Contrast** from the **Tools** menu in the Image Tool.



The following figure shows the image displayed in the Image Tool with the Adjust Contrast tool open in a separate window. In the figure, note how the image histogram shows that pixel values are clustered in the middle of the display range. The display range, shown in the lower right corner of the Image Tool, is the default display range for uint8.

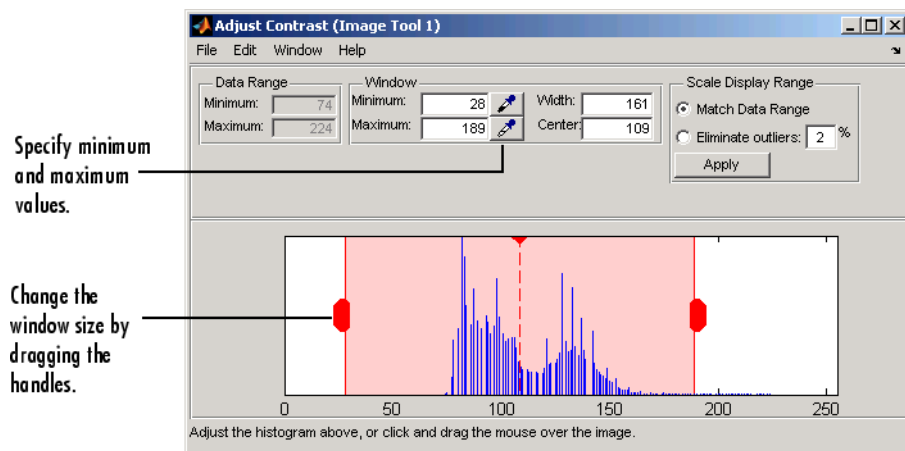


**Image with Default Pixel Value to Display Intensity Mapping**

- Adjust the contrast and brightness by changing the size and position of the window overlaid on the image histogram, using any of the methods described in “Changing the Size of the Adjust Contrast Tool Window” on page 4-39.

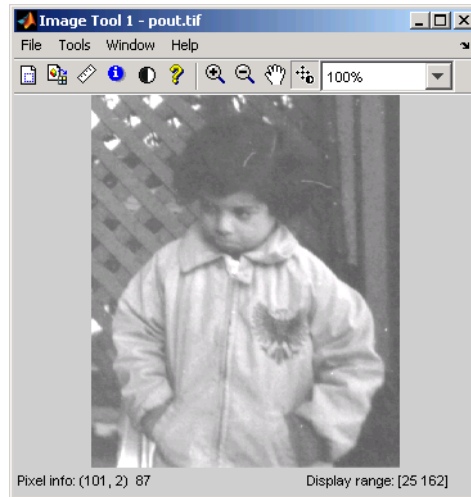
For example, you can grab either of the handles and resize the window, and grab the center line and reposition the window. Alternatively, you can adjust the contrast automatically by trimming outliers at the top and bottom of the image data range. Select the **Eliminate outliers** option and click the **Apply** button. By default, the Adjust Contrast tool removes the top 1% and the bottom 1%, but you can specify other percentages. (You can perform this same operation from the command line using the `stretchlim` function.)

The following figure shows the Adjust Contrast tool after some interactive contrast adjustments.





### Adjust Contrast Tool with Window Resized

The following figure shows the `pout.tif` image after contrast adjustment. In this version, note how the adjusted contrast reveals much more detail in the image background. The Image Tool updates the display range values displayed in the lower right corner of the Image Tool as you change the size of the window.







### Contrast Adjusted Image

## Using the Window/Level Tool

When you start the Adjust Contrast tool you also activate Window/Level mode; the cursor changes shape to the Window/Level cursor . You can also start the Window/Level tool by clicking the Window/Level button  in the Image Tool toolbar. (The name comes from medical applications.)

Using the Window/Level tool, you can change the contrast and brightness of an image by simply dragging the mouse over the image. Moving the mouse horizontally affects contrast; moving the mouse vertically affects brightness.

The following table summarizes how these mouse motions affect the size and position of the window in the Adjust Contrast tool.

Mouse Motion		Effect
Horizontally to the left		Shrinks the window from both sides.
Horizontally to the right		Expands the window from both sides.
Vertically up		Moves the window to the right over the histogram, increasing brightness.
Vertically down		Moves the window to the left over the image histogram, decreasing brightness.

To stop the Window/Level tool, click on the Window/Level button in the Image Tool toolbar, or click any of the navigation buttons in the toolbar.

### Example: Adjusting Contrast with the Window/Level Tool

The following example shows how to use the Window/Level tool to improve the contrast of an image.



- 1 Read an image from a sample DICOM file included with the toolbox.

```
I = dicomread('CT-MON02-16-ankle.dcm');
```

- 2 View the image data using the Image Tool. Because the image data is signed 16-bit data, this example uses the autoscaling syntax.

```
imtool(I, 'DisplayRange', [])
```



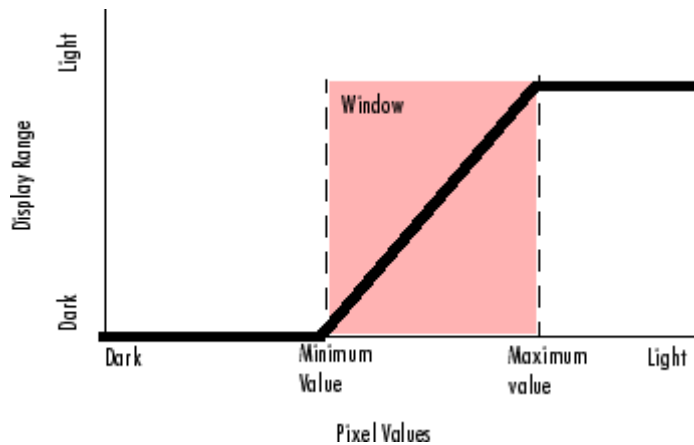
- 3 Click the **Window/Level** button  to start the tool, or select **Window/Level** from the **Tools** menu in the Image Tool. The Window/Level tool also starts when you start the Adjust Contrast tool.
- 4 Move the cursor over the image. The cursor changes to the Window/Level cursor .
- 5 Press and hold the left (or right) mouse button and move the cursor horizontally to the left or right to adjust the contrast, or vertically up or down to change the brightness.

## Understanding Contrast Adjustment

An image lacks contrast when there are no sharp differences between black and white. Brightness refers to the overall lightness or darkness of an image.

To change the contrast or brightness of an image, the Adjust Contrast tool performs *contrast stretching*. In this process, pixel values below a specified value are displayed as black, pixel values above a specified value are displayed as white, and pixel values in between these two values are displayed as shades of gray. The result is a linear mapping of a subset of pixel values to the entire range of grays, from black to white, producing an image of higher contrast.

The following figure shows this mapping. Note that the lower limit and upper limit mark the boundaries of the window, displayed graphically as the red-tinted window in the Adjust Contrast tool.



### Relationship of Pixel Values to Display Range

The Adjust Contrast tool accomplishes this contrast stretching by modifying the `CLim` property of the axes object that contains the image. The `CLim` property controls the mapping of image pixel values to display intensities.

By default, the Image Tool sets the `CLim` property to the default display range according to the data type. For example, the display range of an image of class `uint8` is from 0 to 255. When you use the Adjust Contrast tool, you change the contrast in the image by changing the display range which affects the mapping between image pixel values and the black-to-white range. You create a window over the range that defines which pixels in the image map to the black in the display range by shrinking the range from the bottom up.

## Viewing Multiple Images

If you specify a file that contains multiple images, `imshow` and `imtool` only display the first image in the file. To view all the images in the file, import the images into the MATLAB workspace by calling `imread`. See “Reading Image Data” on page 3-3 for more information.

Some applications create collections of images related by time or view, such as magnetic resonance imaging (MRI) slices or frames of data acquired from a video stream. The Image Processing Toolbox supports these collections of images as four-dimensional arrays, where each separate image is called a frame and the frames are concatenated along the fourth dimension. All the frames in a multiframe image must be the same size.

Once the images are in the MATLAB workspace, there are two ways to display them using `imshow`:

- Displaying each image in a separate figure window
- Displaying multiple frames in a single figure window

To view all the frames in a multiframe image at once, you can also use the `montage` function. See “Displaying All Frames of a Multiframe Image at Once” on page 4-60 for more information.

### Displaying Each Image in a Separate Figure

The simplest way to display multiple images is to display them in separate figure windows. MATLAB does not place any restrictions on the number of images you can display simultaneously.

The Image Tool can only display one image frame at a time. Each time you call `imtool`, it opens a new figure window. Use standard MATLAB indexing syntax to specify the frame to display.

```
imtool(multiframe_array(:,:, :, 1));
```

In contrast, `imshow` always displays an image in the current figure. If you display two images in succession, the second image replaces the first image. To view multiple figures with `imshow`, use the `figure` command to explicitly

create a new empty figure before calling `imshow` for the next image. For example, to view the first three frames in an array of grayscale images `I`,

```
imshow(I(:,:,1))  
figure, imshow(I(:,:,2))  
figure, imshow(I(:,:,3))
```

The Image Tool can only display one image frame at a time. Use standard MATLAB indexing syntax to specify the frame to display.

```
imtool(multiframe_array(:,:,1));
```

### Displaying Multiple Images in the Same Figure

You can use the `imshow` function with the MATLAB `subplot` function or the MATLAB `subimage` function to display multiple images in a single figure window.

---

**Note** `imtool` does not support this capability.

---

### Dividing a Figure Window into Multiple Display Regions

`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  $p$ th display region active.

---

**Note** When you use `subplot` to display multiple color images in one figure window, the images must share the colormap of the last image displayed. In some cases, as illustrated by the following example, the display results can be unacceptable. As an alternative, you can use the `subimage` function, described in “Using the `subimage` Function to Display Multiple Images” on page 4-50, or you can map all images to the same colormap as you load them.

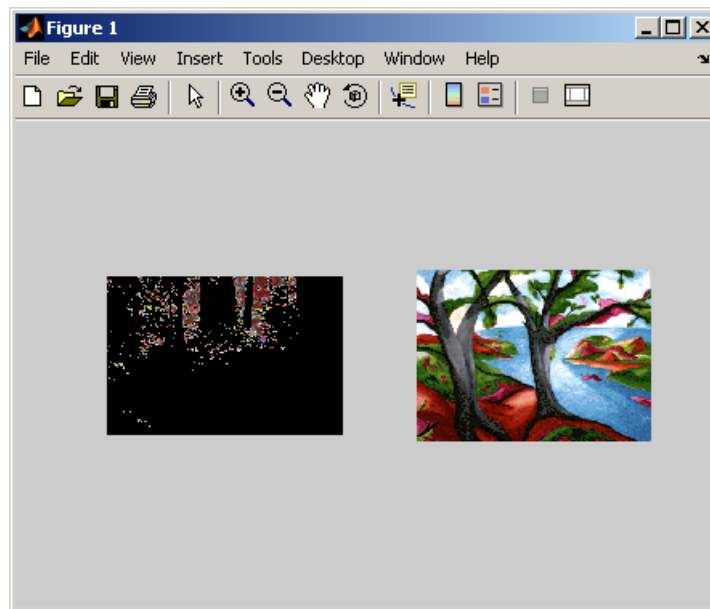
---



For example, you can use this syntax to display two images side by side.

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

In the figure, note how the first image displayed, X1, appears dark after the second image is displayed.

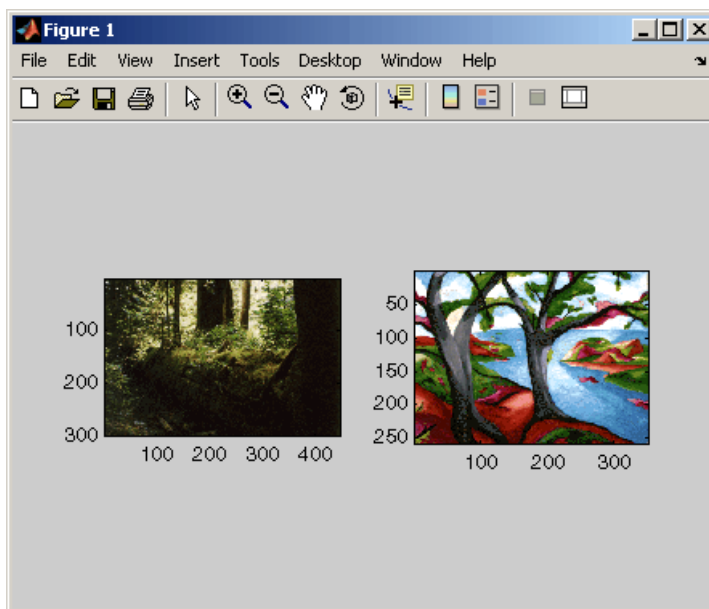


**Two Images in Same Figure Using the Same Colormap**

### Using the subimage Function to Display Multiple Images

subimage converts images to truecolor before displaying them and therefore circumvents the colormap sharing problem. This example uses subimage to display the forest and the trees images 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)
```



**Two Images in Same Figure Using Separate Colormaps**

## Displaying Different Image Types

This section describes how to use `imshow` and `imtool` with the different types of images supported by the Image Processing Toolbox.

- Indexed images
- Grayscale (intensity) images
- Binary images
- Truecolor (RGB) images

If you need help determining what type of image you are working with, see “Image Types in the Toolbox” on page 2-7.

### Displaying Indexed Images

To display an indexed image, using either `imshow` or `imtool`, specify both the image matrix and the colormap. This documentation uses the variable name `X` to represent an indexed image in the workspace, and `map` to represent the colormap.

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

or

```
imtool(X,map)
```

For each pixel in `X`, these functions display the color stored in the corresponding row of `map`. If the image matrix data 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. However, if the image matrix data is of class `uint8` or `uint16`, the value 0 (zero) points to the first row in the colormap, the value 1 points to the second row, and so on. This offset is handled automatically by the `imtool` and `imshow` functions.

If the colormap contains a greater number of colors than the image, the functions ignore the extra colors in the colormap. If the colormap contains fewer colors than the image requires, the functions set all image pixels over the limits of the colormap’s capacity to the last color in the colormap. For example, 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.

### Displaying Grayscale Images

To display a grayscale image, using either `imshow` or `imtool`, specify the image matrix as an argument. This documentation uses the variable name `I` to represent a grayscale image in the workspace.

```
imshow(I)
```

or

```
imtool(I)
```

Both functions display 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.

Grayscale images are similar to indexed images in that each uses an `m-by-3` RGB colormap, but you normally do not specify a colormap for a grayscale image. MATLAB displays grayscale 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 14-2 for a detailed explanation.)

### Displaying Grayscale Images That Have Unconventional Ranges

In some cases, the image data you want to display as a grayscale image might have a display range that is outside the conventional toolbox range (i.e., `[0,1]` for `single` or `double` arrays, `[0,255]` for `uint8` arrays, `[0,65535]` for `uint16` arrays, or `[-32767,32768]` for `int16` arrays). For example, if you filter a grayscale image, some of the output data might fall outside the range of the original data.

To display unconventional range data as an image, you can specify the display range directly, using this syntax for both the `imshow` and `imtool` functions.

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

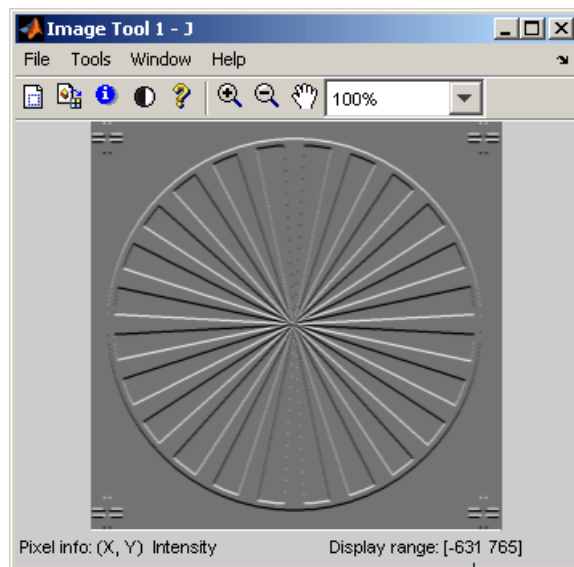
or

```
imtool(I,'DisplayRange',[low high])
```

If you use an empty matrix (`[]`) for the display range, these functions scale the data automatically, setting `low` and `high` to the minimum and maximum values in the array.

The next example filters a grayscale image, creating unconventional range data. The example calls `imtool` to display the image, using the automatic scaling option. If you execute this example, note the display range specified in the lower right corner of the Image Tool window.

```
I = imread('testpat1.png');
J = filter2([1 2;-1 -2],I);
imtool(J,'DisplayRange',[]);
```



### Displaying Binary Images

In MATLAB, a binary image is of class `logical`. Binary images contain only 0's and 1's. Pixels with the value 0 are displayed as black; pixels with the value 1 are displayed as white.

---

**Note** For the toolbox to interpret the image as binary, it must be of class `logical`. Grayscale images that happen to contain only 0's and 1's are not binary images.

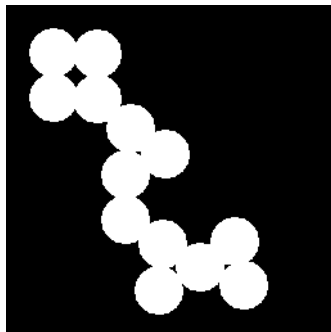
---

To display a binary image, using either `imshow` or `imtool`, specify the image matrix as an argument. For example, this code reads a binary image into the MATLAB workspace and then displays the image. This documentation uses the variable name `BW` to represent a binary image in the workspace

```
BW = imread('circles.png');  
imshow(BW)
```

or

```
imtool(BW)
```



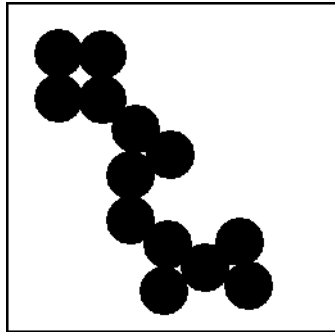
### Changing the Display Colors of a Binary Image

You might prefer to invert binary images when you display them, so that 0 values are displayed as white and 1 values are displayed as black. To do this, use the NOT (`~`) operator in MATLAB. (In this figure, a box is drawn around the image to show the image boundary.) For example:

```
imshow(~BW)
```

or

```
imtool(~BW)
```

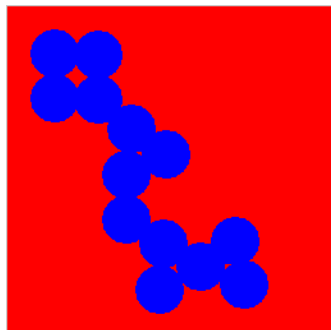


You can also display a binary image using the indexed image colormap syntax. For example, the following command specifies a two-row colormap that displays 0's as red and 1's as blue.

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

or

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



### Displaying Truecolor Images

Truecolor images, also called RGB images, represent color values directly, rather than through a colormap. A truecolor image is an  $m$ -by- $n$ -by-3 array. For each pixel  $(r, c)$  in the image, the color is represented by the triplet  $(r, c, 1:3)$ .

To display a truecolor image, using either `imshow` or `imtool`, specify the image matrix as an argument. For example, this code reads a truecolor image into the MATLAB workspace and then displays the image. This documentation uses the variable name `RGB` to represent a truecolor image in the workspace

```
RGB = imread('peppers.png');  
imshow(RGB)
```

or

```
imtool(RGB)
```





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, `imshow` displays the image using a combination of color approximation and dithering. See “Working with Different Screen Bit Depths” on page 14-2 for more information.

---

**Note** If you display a color image and it appears in black and white, check if the image is an indexed image. With indexed images, you must specify the colormap associated with the image. For more information, see “Displaying Indexed Images” on page 4-51.

---

# Special Display Techniques

In addition to `imshow` and `imtool`, 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” on page 4-58
- “Displaying All Frames of a Multiframe Image at Once” on page 4-60
- “Converting a Multiframe Image to a Movie” on page 4-61
- “Texture Mapping” on page 4-62

## Adding a Colorbar

To display an image with a colorbar that indicates the range of intensity values, first use the `imshow` function to display the image in a MATLAB figure window and then call the `colorbar` function to add the colorbar to the image.

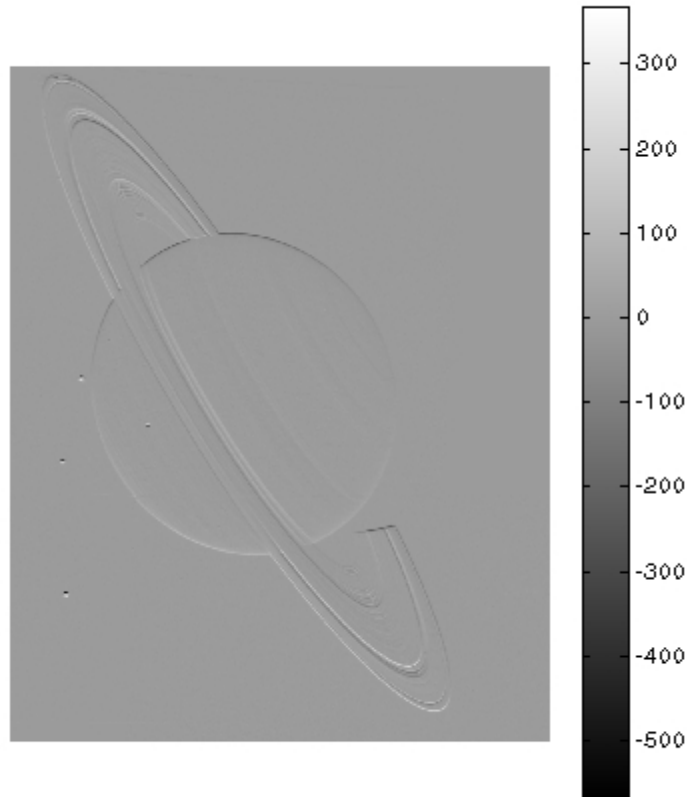
When you add a colorbar to an axes object that contains an image object, the colorbar indicates the data values that the different colors in the image correspond to.

If you want to add a colorbar to an image displayed in the Image Tool, select the **Print to Figure** option from the Image Tool **File** menu. The Image Tool displays the image in a separate figure window to which you can add a colorbar.

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 Grayscale Images That Have Unconventional Ranges” on page 4-52.

In the example below, a grayscale image of class `uint8` is filtered, resulting in data that is no longer in the range `[0,255]`.

```
RGB = imread('saturn.png');  
I = rgb2gray(RGB);  
h = [1 2 1; 0 0 0; -1 -2 -1];  
I2 = filter2(h,I);  
imshow(I2,'DisplayRange',[]), colorbar
```



### Displaying All Frames of a Multiframe Image at Once

To view all the frames in a multiframe array at one time, use the `montage` function. `montage` divides a figure window 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 grayscale image, the syntax is

```
montage(I)
```

To display a multiframe indexed image, the syntax is

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

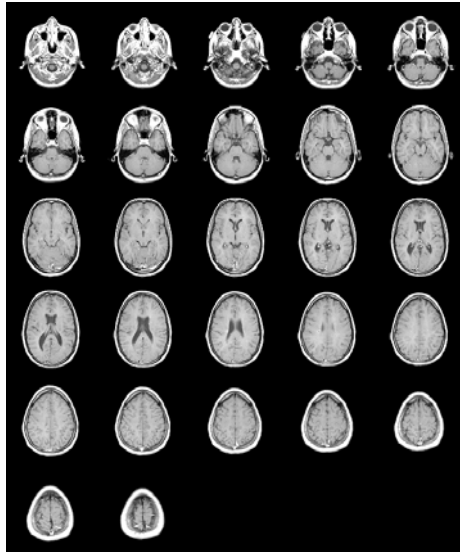
---

**Note** All the frames in a multiframe indexed array must use the same `colormap`.

---

This example loads and displays all frames of a multiframe indexed image. The example initializes an array to hold the 27 frames in the multiframe image file and then loops, using `imread` to read a single frame from the image file at each iteration.

```
mri = uint8(zeros(128,128,1,27));  
  
for frame=1:27  
    [mri(:,:,,frame),map] = imread('mri.tif',frame);  
end  
montage(mri,map);
```



### All Frames of Multiframe Image Displayed in One Figure

montage displays the first frame 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 example creates a movie from a multiframe indexed image.

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

In the example, `X` is a four-dimensional array of images that you want to use for the movie.

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

```
movie(mov);
```

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.

```
mri = uint8(zeros(128,128,1,27));
for frame=1:27
    [mri(:,:,frame),map] = imread('mri.tif',frame);
end

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

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

---

**Note** To view a MATLAB movie, you must have MATLAB installed. To make a movie that can be run outside MATLAB, 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.

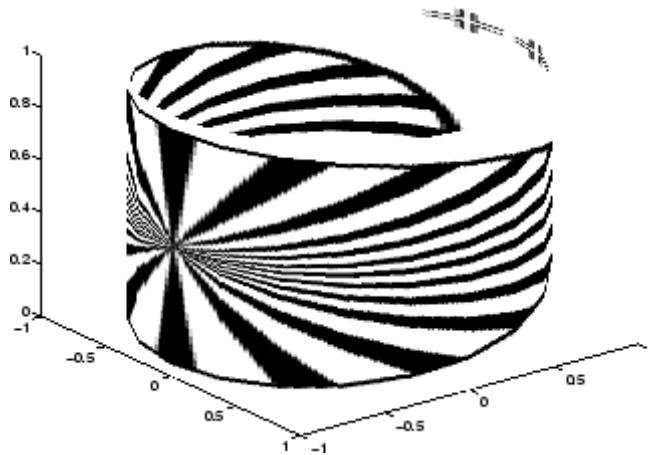
---

### Texture Mapping

When you use `imshow` or `imtool` to view an image, MATLAB displays the image in two dimensions. 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.png');
warp(x,y,z,I);
```



### **An Image Texture-Mapped onto a Cylinder**

The image might not map onto the surface in the way that you expect. 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 Image Data” on page 3-5 for details.

If you want to print an image, use `imshow` to display the image in a MATLAB figure window. If you are using the Image Tool, you must use the **Print to Figure** option on the Image Tool **File** menu. When you choose this option, the Image Tool opens a separate figure window and displays the image in it. You can access the standard MATLAB printing capabilities in this figure window. You can also use the **Print to Figure** option to print the image displayed in the Overview tool and the Pixel Region tool.

Once the image is displayed in a figure window, you can use either the MATLAB `print` command or the **Print** option from the **File** menu of the figure window to print the image. When you print from the figure window, the output includes nonimage elements such as labels, titles, and other annotations.

### Printing and Handle Graphics Object Properties

The output reflects the settings of various properties of Handle Graphic objects. In some cases, you might need to change the settings of certain properties to get the results you want. Here are some tips that might be helpful when you print images:

- Image colors print as shown on the screen. This means that images are not affected by the figure object’s `InvertHardcopy` property.
- To ensure that printed images have the proper size and aspect ratio, set the figure object’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” 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 the `print` command reference page in the MATLAB documentation.

## Setting Toolbox Display Preferences

You can use Image Processing Toolbox preferences to control certain characteristics of how `imshow` and `imshow_tool` display images on your screen. For example, using toolbox preferences, you can specify the initial magnification used by `imshow_tool` and `imshow`.

This section

- Lists the preferences supported by the toolbox
- Describes how to get the current value of a preference using the `iptgetpref` function
- Describes how to set the value of a preference using the `iptsetpref` function

### Toolbox Preferences

The Image Processing Toolbox supports several preferences that affect how `imshow` and `imshow_tool` display images. The following table lists these preferences with brief descriptions. For detailed information about toolbox preferences and their values, see the `iptsetpref` reference page.

Toolbox Preference	Description
<code>ImshowBorder</code>	Controls whether <code>imshow</code> 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).
<code>ImshowAxesVisible</code>	Controls whether <code>imshow</code> displays images with the axes box and tick labels.

<b>Toolbox Preference</b>	<b>Description</b>
ImshowInitialMagnification	Controls the magnification imshow uses when it initially displays an image. This preference can be overridden for a single call to imshow; see “Specifying the Initial Image Magnification” on page 4-6 for more details.
ImtoolInitialMagnification	Controls the magnification the Image Tool uses when it initially displays an image. This preference can be overridden for a single call to imtool; see “Specifying the Initial Image Magnification” on page 4-12 for more details.

## Retrieving the Values of Toolbox Preferences

To determine the current value of a preference, use the `iptgetpref` function. This example uses `iptgetpref` to determine the value of the `ImtoolInitialMagnification` preference.

```
iptgetpref('ImtoolInitialMagnification')  
  
ans =  
  
100
```

Preference names are case insensitive and can be abbreviated. For more information, see the `iptgetpref` reference page.

### Setting the Values of Toolbox Preferences

To specify the value of a toolbox preference, use the `iptsetpref` function. This example calls `iptsetpref` to specify that `imshow` resize the figure window so that it fits tightly around displayed images.

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

For detailed information about toolbox preferences and their values, see the `iptsetpref` reference page.

The value you specify lasts for the duration of the current MATLAB session. To preserve your preference settings from one session to the next, include the `iptsetpref` commands in your `startup.m` file.

# Building GUIs with Modular Tools

---

This chapter describes how to use the toolbox modular tools to create custom image processing applications.

Overview (p. 5-2)

Lists the modular interactive tools

Using Modular Tools (p. 5-6)

Describes how to use the modular tools to create GUIs

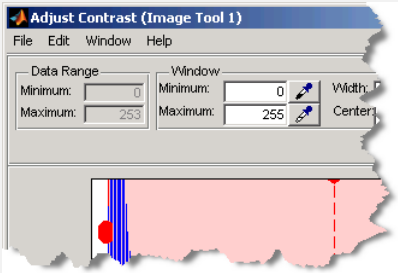
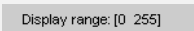
Creating Your Own Modular Tools (p. 5-31)


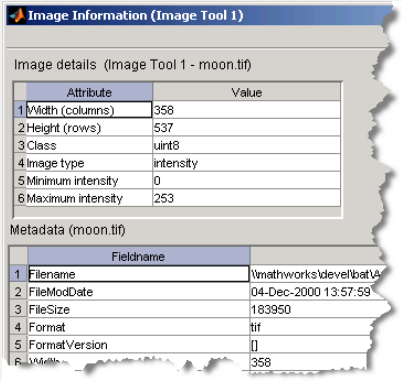
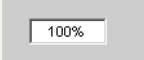
Describes the utility function the toolbox provides to help you create your own modular tools

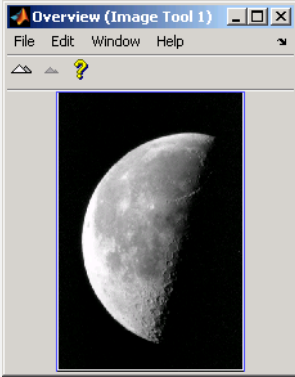
## Overview

The toolbox includes several new modular interactive tools that you can activate from the command line and use with images displayed in a MATLAB figure window, called the *target image* in this documentation. The tools are modular because they can be used independently or in combination to create custom graphical user interfaces (GUIs) for image processing applications. The Image Tool uses these modular tools — see “Using the Image Tool to Explore Images” on page 4-9

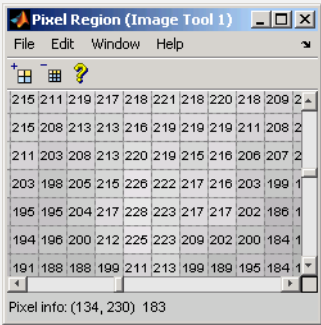
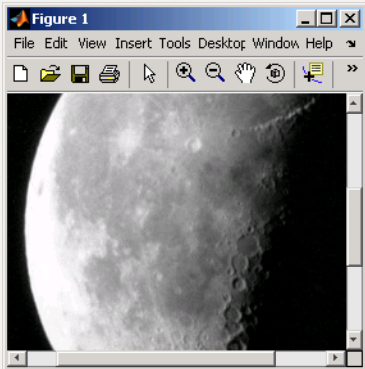
The following table lists the modular tools in alphabetical order. The table includes an illustration of each tool and the function you use to create it. For more information about how the tools operate, see “Using the Image Tool to Explore Images” on page 4-9. For more information about using tools to create GUIs, see “Using Modular Tools” on page 5-6.

Modular Tool	Example	Description
Adjust Contrast tool		<p>Displays a histogram of the target image and enables interactive adjustment of contrast and brightness by manipulation of the display range.</p> <p>Use the <code>imcontrast</code> function to create the tool in a separate figure window and associate it with an image.</p>
Display Range tool		<p>Displays a text string identifying the display range values of the associated image.</p> <p>Use the <code>imdisplayrange</code> function to create the tool, associate it with an image, and embed it in a figure or uipanel.</p>

Modular Tool	Example	Description																												
<p>Distance tool</p>		<p>Displays a draggable, resizable line on an image. Superimposed on the line is the distance between the two endpoints of the line. The distance is measured in units specified by the XData and YData properties, which is pixels by default.</p> <p>Use the <code>imdistline</code> function to create the tool and associate it with an image.</p>																												
<p>Image Information tool</p>	 <table border="1" data-bbox="376 673 706 808"> <thead> <tr> <th>Attribute</th> <th>Value</th> </tr> </thead> <tbody> <tr><td>1 Width (columns)</td><td>358</td></tr> <tr><td>2 Height (rows)</td><td>537</td></tr> <tr><td>3 Class</td><td>uint8</td></tr> <tr><td>4 Image type</td><td>intensity</td></tr> <tr><td>5 Minimum intensity</td><td>0</td></tr> <tr><td>6 Maximum intensity</td><td>253</td></tr> </tbody> </table> <table border="1" data-bbox="376 840 747 968"> <thead> <tr> <th>Fieldname</th> <th>Value</th> </tr> </thead> <tbody> <tr><td>1 Filename</td><td>\\mathworks\devel\bat\A</td></tr> <tr><td>2 FileModDate</td><td>04-Dec-2000 13:57:59</td></tr> <tr><td>3 FileSize</td><td>183950</td></tr> <tr><td>4 Format</td><td>tif</td></tr> <tr><td>5 FormatVersion</td><td>[]</td></tr> <tr><td>6 Width</td><td>358</td></tr> </tbody> </table>	Attribute	Value	1 Width (columns)	358	2 Height (rows)	537	3 Class	uint8	4 Image type	intensity	5 Minimum intensity	0	6 Maximum intensity	253	Fieldname	Value	1 Filename	\\mathworks\devel\bat\A	2 FileModDate	04-Dec-2000 13:57:59	3 FileSize	183950	4 Format	tif	5 FormatVersion	[]	6 Width	358	<p>Displays basic attributes about the target image. If the image displayed was specified as a graphics file, the tool displays any metadata that the image file might contain.</p> <p>Use the <code>imageinfo</code> function to create the tool in a separate figure window and associate it with an image.</p>
Attribute	Value																													
1 Width (columns)	358																													
2 Height (rows)	537																													
3 Class	uint8																													
4 Image type	intensity																													
5 Minimum intensity	0																													
6 Maximum intensity	253																													
Fieldname	Value																													
1 Filename	\\mathworks\devel\bat\A																													
2 FileModDate	04-Dec-2000 13:57:59																													
3 FileSize	183950																													
4 Format	tif																													
5 FormatVersion	[]																													
6 Width	358																													
<p>Magnification box</p>		<p>Creates a text edit box containing the current magnification of the target image. Users can change the magnification of the image by entering a new magnification value.</p> <p>Use <code>immagbox</code> to create the tool, associate it with an image, and embed it in a figure or uipanel.</p> <hr/> <p><b>Note</b> The target image must be contained in a scroll panel.</p> <hr/>																												

Modular Tool	Example	Description
<p>Overview tool</p>		<p>Displays the target image in its entirety with the portion currently visible in the scroll panel outlined by a rectangle superimposed on the image. Moving the rectangle changes the portion of the target image that is currently visible in the scroll panel.</p> <p>Use <code>imoverview</code> to create the tool in a separate figure window and associate it with an image.</p> <p>Use <code>imoverviewpanel</code> to create the tool in a <code>uipanel</code> that can be embedded within another figure or <code>uipanel</code>.</p> <hr/> <p><b>Note</b> The target image must be contained in a scroll panel.</p> <hr/>
<p>Pixel Information tool</p>	<p>Pixel info: (418, 261) 143</p>	<p>Displays information about the pixel the mouse is over in the target image.</p> <p>Use <code>impixelinfo</code> to create the tool, associate it with an image, and display it in a figure or <code>uipanel</code>.</p> <p>If you want to display only the pixel values, without the <b>Pixel info</b> label, use <code>impixelinfoval</code>.</p>



Modular Tool	Example	Description
<p>Pixel Region tool</p>	 <p>The screenshot shows a window titled "Pixel Region (Image Tool 1)". It has a menu bar with "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for a grid, a question mark, and a zoom. The main area is a grid of numbers representing pixel values. The grid is 10 columns wide and 10 rows high. The values range from 183 to 228. At the bottom, it says "Pixel info: (134, 230) 183".</p>	<p>Display pixel values for a specified region in the target image.</p> <p>Use <code>impixelregion</code> to create the tool in a separate figure window and associate it with an image.</p> <p>Use <code>impixelregionpanel</code> to create the tool as a <code>uipanel</code> that can be embedded within another figure or <code>uipanel</code>.</p>
<p>Scroll Panel tool</p>	 <p>The screenshot shows a window titled "Figure 1". It has a menu bar with "File", "Edit", "View", "Insert", "Tools", "Desktop", "Window", and "Help". Below the menu is a toolbar with icons for a scroll panel, a magnifying glass, a hand, and a refresh. The main area is a scrollable panel containing an image of the moon.</p>	<p>Display target image in a scrollable panel.</p> <p>Use <code>imscrollpanel</code> to add a scroll panel to an image displayed in a figure window.</p>

## Using Modular Tools

To use the modular tools to create custom graphical user interfaces (GUIs) for image processing applications, follow this general procedure:

- 1** Display the target image in a figure window.

Image processing applications typically use the `imshow` function to display the target image, i.e., the image being processed. See “Displaying the Target Image” on page 5-7 for more information.

- 2** Create the modular tool, specifying the target image.

When you create a tool, you must associate it with a target image. Most of the tools associate themselves with the image in the current axes, by default. But you can specify the handle to a specific image object, or a handle to a figure, axes, or uipanel object that contains an image. See “Specifying the Target Image” on page 5-8 for more information.

Depending on how you designed your GUI, you might also want to specify the parent object of the modular tool itself. This is optional; by default, the tools either use the same parent as the target image or open in a separate figure window. See “Specifying the Parent of a Modular Tool” on page 5-12 for more information.

In addition, when you create custom GUIs, you might need to specify the position of the graphics objects in the GUI, including the modular tools. See “Positioning the Modular Tools in a GUI” on page 5-15 for more information.

- 3** Set up interactivity between the tool and the target image.

This is an optional step. The modular tools all set up their interactive connections to the target image automatically. However, your GUI might require some additional connectivity. See “Making Connections for Interactivity” on page 5-25.

Many of the modular tools support application programmer interfaces (APIs) that let you assign values to their properties, get the values of their properties, and control other aspects of their functioning. See “Using Modular Tool APIs” on page 5-26 for more information.

The following sections provide more detail on these steps. For a complete illustration, see “Example: Building a Pixel Information GUI” on page 5-17.

## Displaying the Target Image

As the foundation for any image processing GUI you create, use `imshow` to display the target image (or images) in a MATLAB figure window. (You can also use the MATLAB `image` or `imagesc` functions.) Once the image is displayed in the figure, you can associate any of the modular tools with the image displayed in the figure.

This example uses `imshow` to display an image in a figure window.

```
himage = imshow('pout.tif');
```

Because some of the modular tools add themselves to the figure window containing the image, make sure that the Image Processing Toolbox `ImshowBorder` preference is set to `'loose'`, if you are using the `imshow` function. (This is the default setting.) By including a border, you ensure that the modular tools are not displayed over the image in the figure.

## Specifying the Target Image

To associate a modular tool with a target image displayed in a MATLAB figure window, create the tool using the appropriate tool creation function, specifying a handle to the target image as an argument. The function creates the tool and automatically sets up the interactivity connection between the tool and the target image.

This section covers the following topics:

- “Associating Modular Tools with the Default Target Image” on page 5-8
- “Associating Modular Tools with a Particular Image” on page 5-10
- “Getting the Handle of the Target Image” on page 5-11

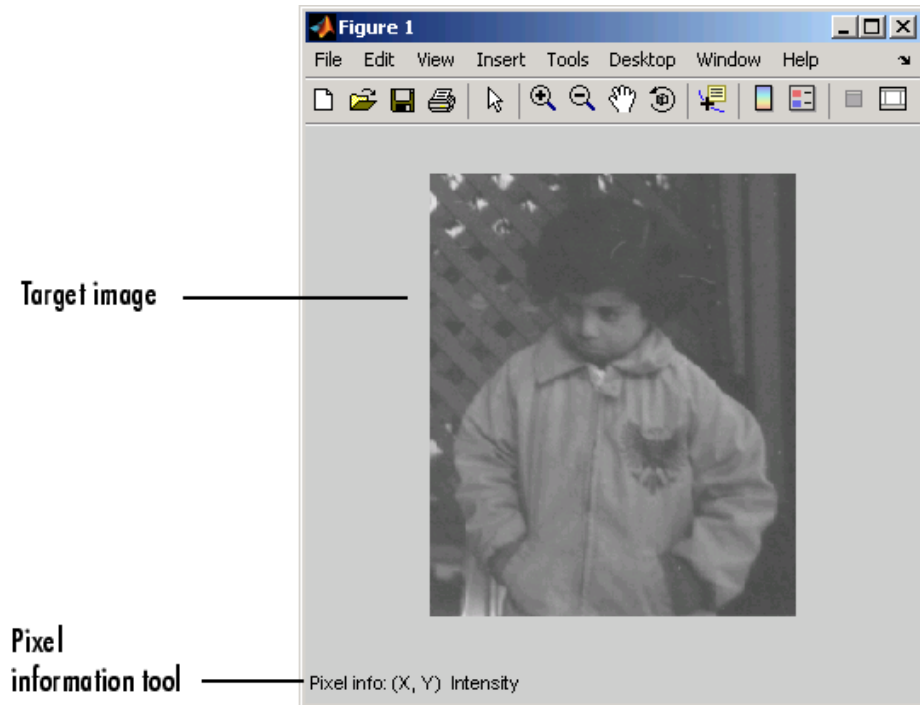
## Associating Modular Tools with the Default Target Image

By default, most of the modular tool creation functions support a no-argument syntax that uses the image in the current figure as the target image. If the current figure contains multiple images, the tools associate themselves with the first image in the figure object’s children (the last image created). `impixelinfo`, `impixelinfoval` and `imshowarrayrange` can work with multiple images in a figure.

For example, to use the Pixel Information tool with a target image, display the image in a figure window, using `imshow`, and then call the `impixelinfo` function to create the tool. In this example, the image in the current figure is the target image.

```
imshow('pout.tif');  
impixelinfo
```

The following figure shows the target image in a figure with the Pixel Information tool in the lower left corner of the window. The Pixel Information tool automatically sets up a connection to the target image: when you move the cursor over the image, the tool displays the  $x$ - and  $y$ -coordinates and value of the pixel under the cursor.



**Figure Window with Pixel Information Tool**

### **Associating Modular Tools with a Particular Image**

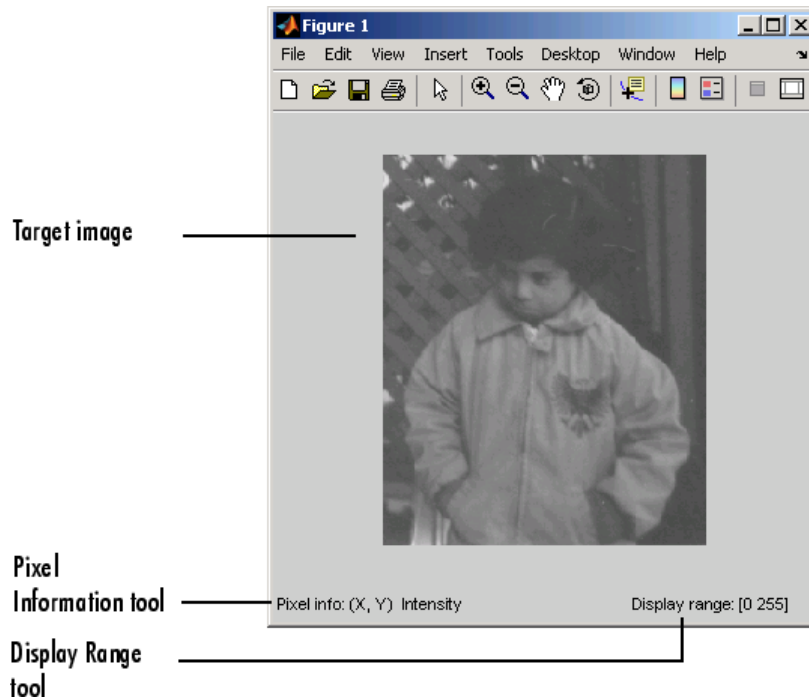
You can specify the target image of the modular tool when you create it by passing a handle to the target image as an argument to the modular tool creation function. You can also specify a handle to a figure, axes, or uipanel object that contains the target image.

Continuing the example in the previous section, you might want to add the Display Range tool to the figure window that already contains the Pixel Information tool. To do this, call the `imshow` function, specifying the handle to the target image. You could also have specified the handle of the figure, axes, or uipanel object containing the target image.

```
himage = imshow('pout.tif');  
hpixelinfopanel = impixelinfo(himage);  
hdrangepanel = imshowrange(himage);
```

Note that the example retrieves handles to the uipanel objects created by the `impixelinfo` and `imshowrange` functions; both tools are uipanel objects. It can be helpful to get handles to the tools if you want to change their positioning. See “Positioning the Modular Tools in a GUI” on page 5-15 for more information.

The following figure shows the target image in a figure with the Pixel Information tool in the lower left corner and the Display Range tool in the lower right corner of the window. The Display Range tool automatically sets up a connection to the target image: when you move the cursor over the image (or images) in the figure, the Display Range tool shows the display range of the image.



**Figure Window with Pixel Information and Display Range Tools**

## Getting the Handle of the Target Image

The examples in the previous section use the optional `imshow` syntax that returns a handle to the image displayed, `himage`. When creating GUIs with the modular tools, having a handle to the target image can be useful. You can get the handle when you first display the image, using this optional `imshow` syntax. You can also get a handle to the target image using the `imhandles` function. The `imhandles` function returns all the image objects that are children of a specified figure, axes, uipanel, or image object.

For example, `imshow` returns a handle to the image in this syntax.

```
hfig = figure;
himage = imshow('moon.tif')
himage =

    152.0055
```

When you call the `imhandles` function, specifying a handle to the figure (or axes) containing the image, it returns a handle to the same image.

```
himage2 = imhandles(hfig)
himage2 =

    152.0055
```

### Specifying the Parent of a Modular Tool

When you create a modular tool, in addition to specifying the target image, you can optionally specify the object that you want to be the parent of the tool. By specifying the parent, you determine where the tool appears on your screen. Using this syntax of the modular tool creation functions, you can add the tool to the figure window containing the target image, open the tool in a separate figure window, or create some other combination.

Specifying the parent is optional; the modular tools all have a default behavior. Some of the smaller tools, such as the Pixel Information tool, use the parent of the target image as their parent, inserting themselves in the same figure window as the target image. Other modular tools, such as the Pixel Region tool or the Overview tool, open in separate figures of their own.

### Tools With Separate Creation Functions

Two of the tools, the Pixel Region tool and the Overview tool, have a separate creation function to provide this capability. Their primary creation functions, `imoverview` and `impixelregion`, open the tools in a separate figure window. To specify a different parent, you must use the `imoverviewpanel` and `impixelregionpanel` functions.

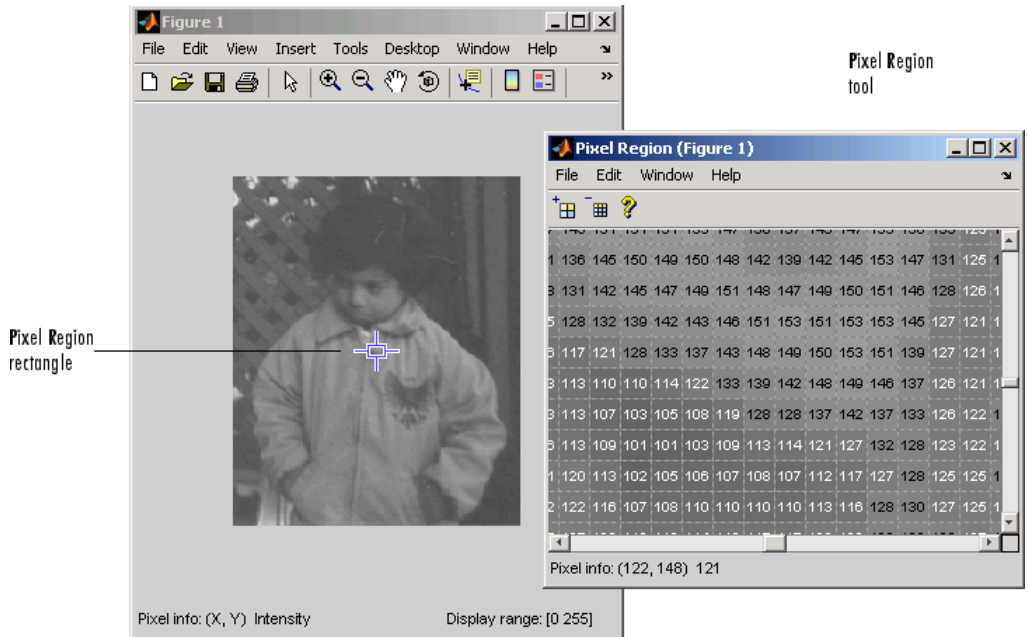


**Note** The Overview tool and the Pixel Region tool provide additional capabilities when created in their own figure windows. For example, both tools include zoom buttons that are not part of their uipanel versions.

### Example: Embedding the Pixel Region Tool in an Existing Figure

This example shows the default behavior when you create the Pixel Region tool using the `impixelregion` function. The tool opens in a separate figure window, as shown in the following figure.

```
himage = imshow('pout.tif')
hpixelinfopanel = impixelinfo(himage);
hdrangepanel = imdisplayrange(himage);
hpixreg = impixelregion(himage);
```



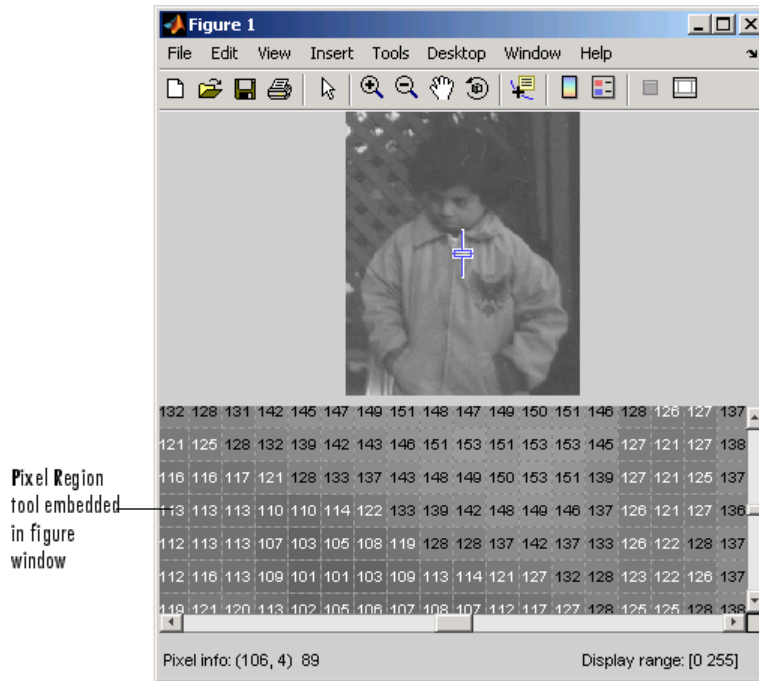
### Target Image with Pixel Region Tool in Separate Window (Default)

To embed the Pixel Region tool in the same window as the target image, you must specify the handle of the target image's parent figure as the parent of the Pixel Region tool when you create it.

The following example creates a figure and an axes object, getting handles to both objects. The example needs these handles to perform some repositioning of the objects in the figure to ensure their visibility. See “Positioning the Modular Tools in a GUI” on page 5-15 for more information. The example then creates the modular tools, specifying the figure containing the target image as the parent of the Pixel Region tool. Note that the example uses the `impixelregionpanel` function to create the tool.

```
hfig = figure;
hax = axes('units','normalized','position',[0 .5 1 .5]);
himage = imshow('pout.tif')
hpixelinfopanel = impixelinfo(himage);
hdrangepanel = imdisplayrange(himage);
hpixreg = impixelregionpanel(hfig,himage);
set(hpixreg, 'Units','normalized','Position',[0 .08 1 .4]);
```

The following figure shows the Pixel Region embedded in the same figure as the target image.



### Target Image with Embedded Pixel Region Tool

## Positioning the Modular Tools in a GUI

When you create the modular tools, they have default positioning behavior. For example, the `impixelinfo` function creates the tool as a `uipanel` object that is the full width of the figure window, positioned in the lower left corner of the target image figure window.

Because the modular tools are constructed from standard Handle Graphics objects, such as `uipanel` objects, you can use properties of the objects to change their default positioning or other characteristics.

For example, in “Specifying the Parent of a Modular Tool” on page 5-12, when the Pixel Region tool was embedded in the same figure window as the target

image, the example had to reposition both the image object and the Pixel Region tool uipanel object to make them both visible in the figure window.

### **Specifying the Position with a Position Vector**

To specify the position of a modular tool or other graphics object, set the value of the `Position` property of the object. As the value of this property, you specify a four-element position vector `[left bottom width height]`, where `left` and `bottom` specify the distance from the lower left corner of the parent container object, such as a figure. The `width` and `height` specify the dimensions of the object.

When you use a position vector, you can specify the units of the values in the vector by setting the value of the `Units` property of the object. To allow better resizing behavior, use normalized units because they specify the relative position, not the exact location in pixels. See the reference page for the `Handle Graphics` object for more details.

For example, when you first create an embedded Pixel Region tool in a figure, it appears to take over the entire figure because, by default, the position vector is set to `[0 0 1 1]`, in normalized units. This position vector tells the tool to align itself with the bottom left corner of its parent and fill the entire object. To accommodate the image and the Pixel Information tool and Display Range tools, change the position of the Pixel Region tool in the lower half of the figure window, leaving room at the bottom for the Pixel Information and Display Range tools. Here is the position vector for the Pixel Region tool.

```
set(hpixreg, 'Units', 'normalized', 'Position', [0 .08 1 .4])
```

To accommodate the Pixel Region tool, reposition the target image so that it fits in the upper half of the figure window, using the following position vector. To reposition the image, you must specify the `Position` property of the axes object that contains it; image objects do not have a `Position` property.

```
set(hax, 'Units', 'normalized', 'Position', [0 0.5 1 0.5])
```

## Example: Building a Pixel Information GUI

This example shows how to use the tools to create a simple GUI that provides information and pixels and features in an image. The GUI displays an image and includes the following modular pixel information tools:

- Display Range tool
- Distance tool
- Pixel Information tool
- Pixel Region tool panel

The example suppresses the figure window toolbar and menu bar because the standard figure zoom tools are not compatible with the toolbox modular navigation tools — see “Adding Navigation Aids to a GUI” on page 5-19.

```
function my_pixinfotool(im)
% Create figure, setting up properties
hfig = figure('Toolbar','none',...
             'Menubar', 'none',...
             'Name','My Pixel Info Tool',...
             'NumberTitle','off',...
             'IntegerHandle','off');

% Create axes and reposition the axes
% to accommodate the Pixel Region tool panel
hax = axes('Units','normalized',...
          'Position',[0 .5 1 .5]);

% Display image in the axes and get a handle to the image
himage = imshow(im);

% Add Distance tool, specifying axes as parent
hdist = imdistline(hax);

% Add Pixel Information tool, specifying image as parent
hpixinfo = impixelinfo(himage);

% Add Display Range tool, specifying image as parent
hdrange = imdisplayrange(himage);
```

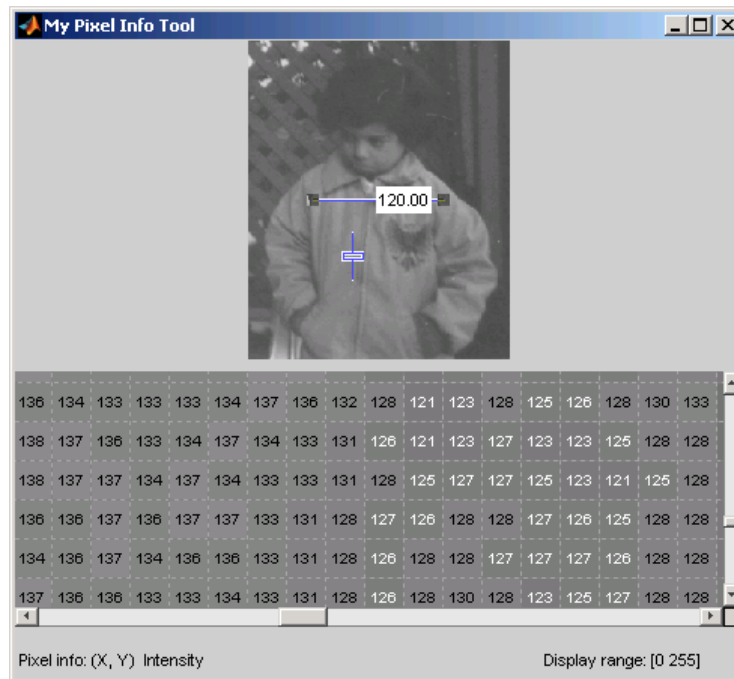
```
% Add Pixel Region tool panel, specifying figure as parent
% and image as target
hpixreg = impixelregionpanel(hfig,himage);

% Reposition the Pixel Region tool to fit in the figure
% window, leaving room for the Pixel Information and
% Display Range tools.
set(hpixreg, 'units','normalized','position',[0 .08 1 .4])
```

To use the tool, pass it an image that is already in the MATLAB workspace.

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

The tool opens a figure window, displaying the image in the upper half, with the Distance tool overlaid on the image, and the Pixel Information tool, Display Range tool, and the Pixel Region tool panel in the lower half of the figure.



### Custom Image Display Tool with Pixel Information

## Adding Navigation Aids to a GUI

---

**Note** The toolbox modular navigation tools are incompatible with standard MATLAB figure window navigation tools. When using these tools in a GUI, suppress the toolbar and menu bar in the figure windows to avoid conflicts between the tools.

---

The toolbox includes several modular tools that you can use to add navigation aids to a GUI application:

- Scroll Panel
- Overview tool
- Magnification box

The Scroll Panel is the primary navigation tool; it is a prerequisite for the other navigation tools. When you display an image in a Scroll Panel, the tool displays only a portion of the image, if it is too big to fit into the figure window. When only a portion of the image is visible, the Scroll Panel adds horizontal and vertical scroll bars, to enable viewing of the parts of the image that are not currently visible.

Once you create a Scroll Panel, you can optionally add the other modular navigation tools: the Overview tool and the Magnification tool. The Overview tool displays a view of the entire image, scaled to fit, with a rectangle superimposed over it that indicates the part of the image that is currently visible in the scroll panel. The Magnification Box displays the current magnification of the image and can be used to change the magnification.

The following sections provide more details.

- “Understanding Scroll Panels” on page 5-20 — Adding a scroll panel to an image display changes the relationship of the graphics objects used in the display. This section provides some essential background.
- “Example: Building a Navigation GUI for Large Images” on page 5-23 — This section shows how to add a scroll panel to an image display.

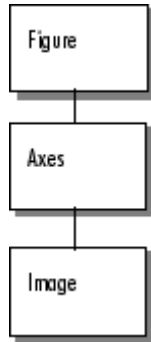
## **Understanding Scroll Panels**

When you display an image in a scroll panel, it changes the object hierarchy of your displayed image. This diagram illustrates the typical object hierarchy for an image displayed in an axes object in a figure object.

```
hfig = figure;  
himage = imshow('concordaerial.png');
```



The following figure shows this object hierarchy.

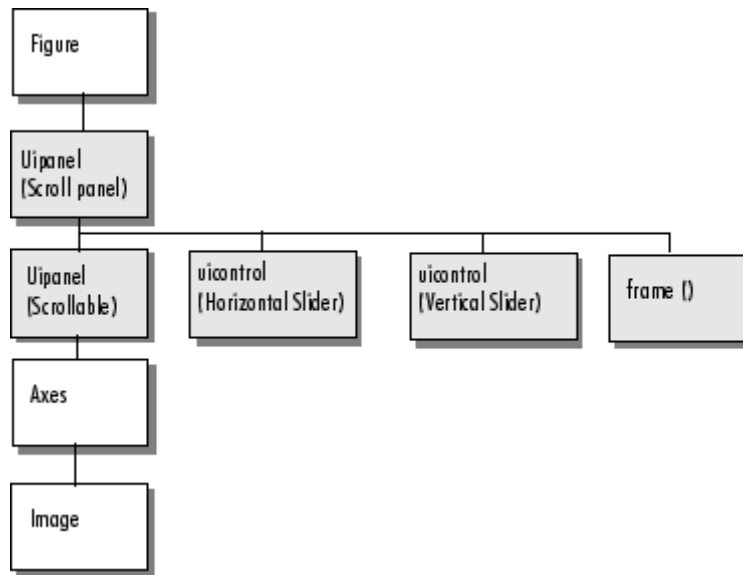


### **Object Hierarchy of Image Displayed in a Figure**

When you call the `imscrollpanel` function to put the target image in a scrollable window, this object hierarchy changes. For example, this code adds a scroll panel to an image displayed in a figure window, specifying the parent of the scroll panel and the target image as arguments. The example suppresses the figure window toolbar and menu bar because they are not compatible with the scroll panel navigation tools.

```
hfig = figure('Toolbar','none',...  
             'Menubar','none');  
himage = imshow('concordaerial.png');  
hpanel = imscrollpanel(hfig,himage);
```

The following figure shows the object hierarchy after the call to `imshow`. Note how `imshow` inserts new objects (shaded in gray) into the hierarchy between the figure object and the axes object containing the image.



**Object Hierarchy of Image Displayed in Scroll Panel**

The following figure shows how these graphics objects appear in the scrollable image as it is displayed on the screen.

Scrollable image



### Components of a Scroll Panel

#### **Example: Building a Navigation GUI for Large Images**

If your work typically requires that you view large images, you might want to create a custom image display function that includes the modular navigation tools.

This example creates a tool that accepts an image as an argument and displays the image in a scroll panel with an Overview tool and a Magnification box.

---

**Note** Because the toolbox scrollable navigation is incompatible with standard MATLAB figure window navigation tools, the example suppresses the toolbar and menu bar in the figure window.

---

```
function my_large_image_display(im)

% Create a figure without toolbar and menubar.
hfig = figure('Toolbar','none',...
             'Menubar','none',...
             'Name','My Large Image Display Tool',...
             'NumberTitle','off',...
             'IntegerHandle','off');

% Display the image in a figure with imshow.
himage = imshow(im);

% Add the scroll panel.
hpanel = imscrollpanel(hfig,himage);

% Position the scroll panel to accommodate the other tools.
set(hpanel,'Units','normalized','Position',[0 .1 1 .9]);

% Add the Magnification box.
hMagBox = immagbox(hfig,himage);

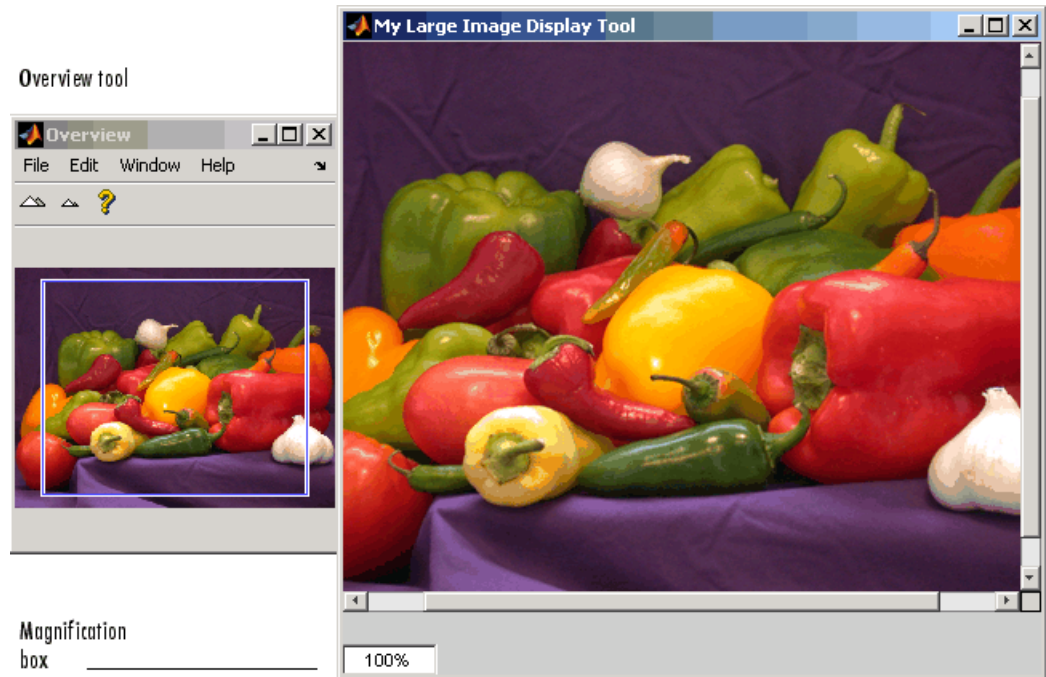
% Position the Magnification box
pos = get(hMagBox,'Position');
set(hMagBox,'Position',[0 0 pos(3) pos(4)]);

% Add the Overview tool.
hovervw = imoverview(himage);
```

To use the tool, pass it a large image that is already in the MATLAB workspace.

```
big_image = imread('peppers.png');
my_large_image_display(big_image)
```

The tool opens a figure window, displaying the image in a scroll panel with the Overview tool and the Magnification Box tool.



### Custom Image Display Tool with Navigation Aids

## Making Connections for Interactivity

When you create a modular tool and associate it with a target image, the tool automatically makes the necessary connections to the target image to do its job.

For example, the Pixel Information tool sets up a connection to the target image so that it can display the location and value of the pixel currently under the cursor. The Overview tool sets up a two-way connection to the target image:

- **Target image to the Overview tool** — If the visible portion of the image changes, by scrolling, panning, or by changing the magnification, the

Overview tool changes the size and location of the detail rectangle to the indicate the portion of the image that is now visible.

- **Overview tool to the target image** — If a user moves the detail rectangle in the Overview tool, the portion of the target image visible in the scroll panel is updated.

The modular tools accomplish this interactivity by using callback properties of the graphics objects. For example, the figure object supports a `WindowButtonMotionFcn` callback that executes whenever the mouse button is depressed.

## Using Modular Tool APIs

Many of the modular tools support an application programmer interface (API). This API is a set of functions that let you get information about the tool as it operates and set up callbacks to get notification of events.

For example, the Magnification box supports a single API function: `setMagnification`. You can use this API function to set the magnification value displayed in the Magnification box. The Magnification box automatically notifies the scroll panel to change the magnification of the image based on the value. The scroll panel also supports an extensive set of API functions. To get information about these APIs, see the reference page for the modular tool.

## Example: Building an Image Comparison Tool

To illustrate how to use callbacks to make the connections required for interactions between tools, this example uses the Scroll Panel API to build a simple image comparison GUI. This custom tool displays two images side by side in scroll panels that are synchronized in location and magnification. The custom tool also includes an Overview tool and a Magnification box.

```
function my_image_compare_tool(left_image, right_image)

% Create the figure
hFig = figure('ToolBar','none',...
             'MenuBar','none',...
             'Name','My Image Compare Tool',...
             'NumberTitle','off',...
             'IntegerHandle','off');
```

```
% Display left image
subplot(121)
hImL = imshow(left_image);

% Display right image
subplot(122)
hImR = imshow(right_image);

% Create a scroll panel for left image
hSpL = imscrollpanel(hFig,hImL);
set(hSpL,'Units','normalized',...
    'Position',[0 0.1 .5 0.9])

% Create scroll panel for right image
hSpR = imscrollpanel(hFig,hImR);
set(hSpR,'Units','normalized',...
    'Position',[0.5 0.1 .5 0.9])

% Add a Magnification box
hMagBox = immagbox(hFig,hImL);
pos = get(hMagBox,'Position');
set(hMagBox,'Position',[0 0 pos(3) pos(4)])

%% Add an Overview tool
imoverview(hImL)

%% Get APIs from the scroll panels
apiL = iptgetapi(hSpL);
apiR = iptgetapi(hSpR);

%% Synchronize left and right scroll panels
apiL.setMagnification(apiR.getMagnification())
apiL.setVisibleLocation(apiR.setVisibleLocation())

% When magnification changes on left scroll panel,
% tell right scroll panel
apiL.addNewMagnificationCallback(apiR.setMagnification);

% When magnification changes on right scroll panel,
```

```
% tell left scroll panel
apiR.addNewMagnificationCallback(apiL.setMagnification);

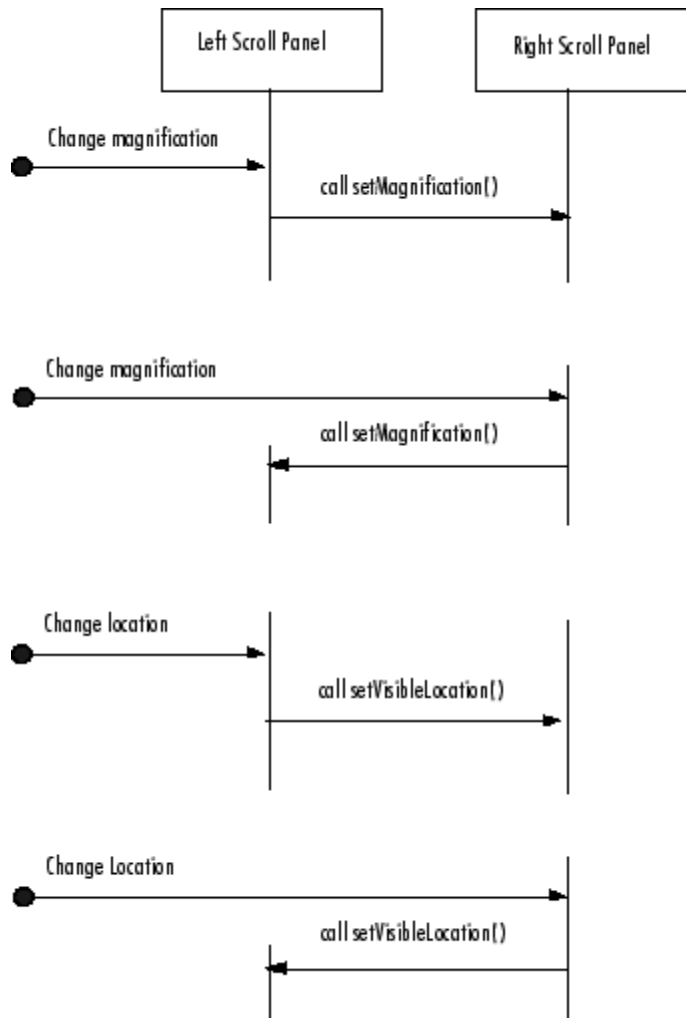
% When location changes on left scroll panel,
% tell right scroll panel
apiL.addNewLocationCallback(apiR.setVisibleLocation);

% When location changes on right scroll panel,
% tell left scroll panel
apiR.addNewLocationCallback(apiL.setVisibleLocation);
```

The tool sets up a complex interaction between the scroll panels with just a few calls to Scroll Panel API functions. In the code, the tool specifies a callback function to execute every time the magnification changes. The function specified is the `setMagnification` API function of the other scroll panel. Thus, whenever the magnification changes in one of the scroll panels, the other scroll panel changes its magnification to match. The tool sets up a similar connection for position changes.

The following figure is a sequence diagram that shows the interaction between the two scroll panels set up by the comparison tool for both changes in magnification and location.





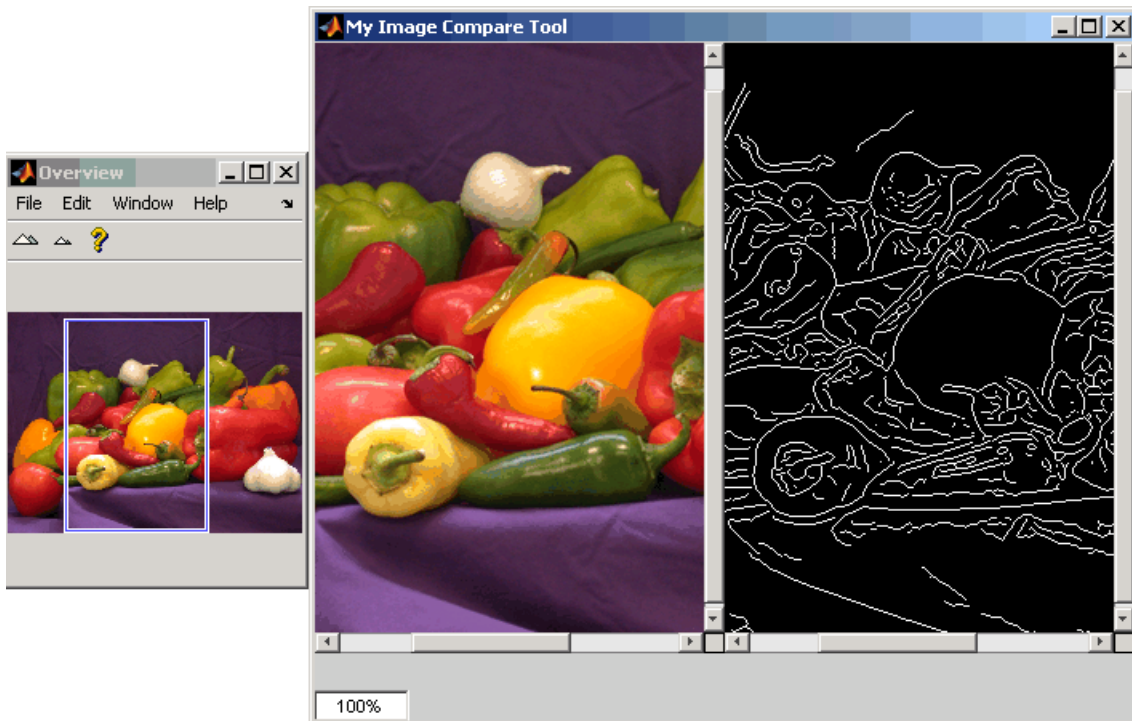
### Scroll Panel Connections in Custom Image Comparison Tool

To use the image comparison tool, pass it two images as arguments.

```

left_image = imread('peppers.png');
right_image = edge(left_image(:,:,1),'canny');
my_image_compare_tool(left_image,right_image);
  
```

The tool opens a figure window, displaying the two images side by side, in separate scroll panels. The custom compare tool also includes an Overview tool and a Magnification box. When you move the detail rectangle in the Overview tool or change the magnification in one image, both images respond.



**Custom Image Comparison Tool with Synchronized Scroll Panels**

## Creating Your Own Modular Tools

Because the toolbox uses an open architecture for the modular interactive tools, you can extend the toolbox by creating your own modular interactive tools, using standard Handle Graphics concepts and techniques. To help you create tools that integrate well with the existing modular interactive tools, the toolbox includes many utility functions that perform commonly needed tasks.

The utility functions can help check the input arguments to your tool, add callback functions to a callback list or remove them from a list, draw a draggable point, line, or rectangle over an image, and align figure windows in relation to a fixed window. The following table lists these utility functions in alphabetical order. See the function's reference page for more detailed information.

Utility Function	Description
<code>getimagemodel</code>	Retrieve image model objects from image handles
<code>getrangefromclass</code>	Get default display range of image, based on its class
<code>imagemodel</code>	Access to properties of an image relevant to its display
<code>imattributes</code>	Return information about image attributes
<code>imgca</code>	Get handle to most recent current axis containing an image
<code>imgcf</code>	Get handle to most recent current figure containing an image
<code>imgetfile</code>	Display Open Image dialog box
<code>imhandles</code>	Get all image handles
<code>imline</code>	Create a line that can be dragged and resized interactively
<code>impoint</code>	Create a point that can be dragged interactively
<code>imrect</code>	Create a rectangle that can be dragged interactively
<code>iptaddcallback</code>	Add function handle to a callback list

<b>Utility Function</b>	<b>Description</b>
<code>iptcheckconn</code>	Check validity of connectivity argument
<code>iptcheckhandle</code>	Check validity of image handle argument
<code>iptcheckinput</code>	Check validity of input argument
<code>iptcheckmap</code>	Check validity of colormap argument
<code>iptchecknargin</code>	Check number of input arguments
<code>iptcheckstrs</code>	Check validity of string argument
<code>iptgetapi</code>	Get application programmer interface (API) for a handle
<code>iptGetPointerBehavior</code>	Retrieve pointer behavior from HG object
<code>ipticondir</code>	Return names of directories containing IPT and MATLAB icons
<code>iptnum2ordinal</code>	Convert positive integer to ordinal string
<code>iptPointerManager</code>	Install mouse pointer manager in figure
<code>iptremovecallback</code>	Delete function handle from callback list
<code>iptSetPointerBehavior</code>	Store pointer behavior in HG object
<code>iptwindowalign</code>	Align figure windows

# Spatial Transformations

---

This chapter describes the spatial transformation functions in the Image Processing Toolbox. A spatial transformation (also known as a geometric operation) modifies the spatial relationship between pixels in an image, mapping pixel locations in an input image to new locations in an output image. The toolbox includes functions that perform certain specialized spatial transformations, such as resizing and rotating an image. In addition, the toolbox includes functions that you can use to perform many types of 2-D and N-D spatial transformations, including custom transformations.

Interpolation (p. 6-3)	Provides background information about spatial transformations and interpolation
Resizing an Image (p. 6-5)	Describes how to use the <code>imresize</code> function to change the size of an image
Rotating an Image (p. 6-8)	Describes how to use the <code>imrotate</code> function to rotate an image
Cropping an Image (p. 6-10)	Describes how to use the <code>imcrop</code> function to extract a rectangular portion of an image
Performing General 2-D Spatial Transformations (p. 6-11)	Describes how to perform a general spatial transformation of a 2-D image

Performing N-Dimensional Spatial Transformations (p. 6-23)

Describes the toolbox functions you can use to perform N-D spatial transformations of arrays

Example: Performing Image Registration (p. 6-25)

Shows how to use some capabilities of `imtransform` to view the results of image registration

## Interpolation

Interpolation is the process used to 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 toolbox uses interpolation to determine the values for the additional pixels. 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 “Getting the Intensity Profile of an Image” on page 11-3 for information about this function.

### Interpolation Methods

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, they find the point in the input image that the output pixel corresponds to. They then assign a value to the output pixel by computing a weighted average of some set of pixels in the vicinity of the point. The weightings are based on the distance each pixel is from the point.

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

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

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

### **Interpolation and Image Types**

You can use all three types of interpolation with grayscale, truecolor, or binary images; however, bilinear or bicubic are recommended. Nearest-neighbor is the only type of interpolation that can be used with indexed images.



## Resizing an Image

To change the size of an image, use the `imresize` function. Using `imresize`, you can

- Specify the size of the output image
- Specify the interpolation method used
- Specify the filter to use to prevent aliasing

### Specifying the Size of the Output Image

Using `imresize`, you can specify the size of the output image in two ways:

- By specifying the magnification factor to be used on the image
- By specifying the dimensions of the output image

### Using the Magnification Factor

To enlarge an image, specify a magnification factor greater than 1. To reduce an image, specify a magnification factor between 0 and 1. For example, the command below increases the size of an image by 1.25 times.

```
I = imread('circuit.tif');
J = imresize(I,1.25);
imshow(I)
figure, imshow(J)
```

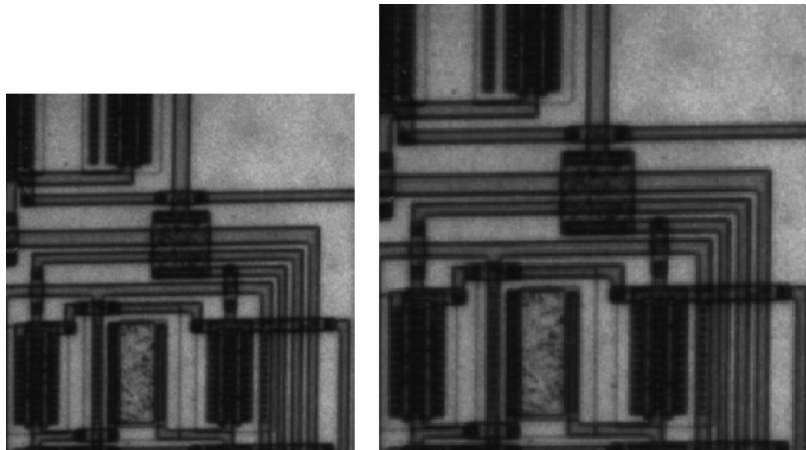


Image Courtesy of Steve Decker and Shujaat Nadeem

### Specifying the Size of the Output Image

You can specify the size of the output image by passing a vector that contains the number of rows and columns in the output image. If the specified size does not produce the same aspect ratio as the input image, the output image will be distorted.

The following command creates an output image with 100 rows and 150 columns.

```
I = imread('circuit.tif');  
J = imresize(I,[100 150]);  
imshow(I)  
figure, imshow(J)
```

### Specifying the Interpolation Method

By default, `imresize` uses nearest-neighbor interpolation to determine the values of pixels in the output image, but you can specify other interpolation methods. This table lists the supported interpolation methods in order of complexity. See “Interpolation” on page 6-3 for more information about these methods.

Argument Value	Interpolation Method
'nearest'	Nearest-neighbor interpolation (the default)
'bilinear'	Bilinear interpolation
'bicubic'	Bicubic interpolation

In this example, `imresize` uses the bilinear interpolation method.

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

## Using Filters to Prevent Aliasing

When you reduce the size of an image, you lose some of the original pixels because there are fewer pixels in the output image. Aliasing that occurs as a result of size reduction normally appears as “stair-step” patterns (especially in high-contrast images), or as moiré (ripple-effect) patterns in the output image.

When you specify either bilinear or bicubic as the interpolation method, `imresize` automatically applies a lowpass filter to the image before interpolation to limit the impact of aliasing on the output image.

---

**Note** Even with lowpass filtering, resizing an image can introduce artifacts, because information is always lost when you reduce the size of an image.

---

The `imresize` function does not apply a lowpass filter if nearest-neighbor interpolation is used. Nearest-neighbor interpolation is primarily used for indexed images, and lowpass filtering is not appropriate for these images.

When using `imresize` to reduce the size of an image, you can specify the size of the lowpass filter or specify a filter of your own creation. For example, the following code specifies a 9-by-9 filter. (The default size is 11-by-11.) If you specify the value 0 (zero), `imresize` does not perform lowpass filtering.

```
J = imresize(I,'bilinear',9);
```

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

## Rotating an Image

To rotate an image, use the `imrotate` function. `imrotate` accepts two primary arguments:

- The image to be rotated
- The rotation angle

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. This example rotates the image `I` 35 degrees in the counterclockwise direction.

```
J = imrotate(I,35);
```

As optional arguments to `imrotate`, you can also specify

- The size of the output image
- The interpolation method

### Specifying the Size of the Output Image

By default, `imrotate` creates an output image large enough to include the entire original image. Pixels that fall outside the boundaries of the original image are set to 0 and appear as a black background in the output image. If you specify the text string `'crop'` as an argument, `imrotate` crops the output image to be the same size as the input image. (See the reference page for `imrotate` for an example of cropping.)

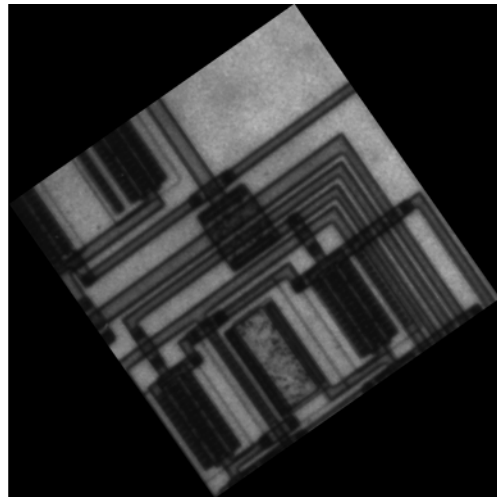
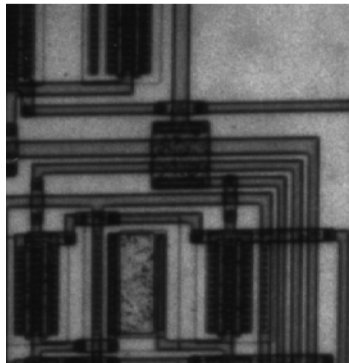
### Specifying the Interpolation Method

By default, `imrotate` uses nearest-neighbor interpolation to determine the value of pixels in the output image, but you can specify other interpolation methods. This table lists the supported interpolation methods in order of complexity. See “Interpolation” on page 6-3 for more information about these methods.

Argument Value	Interpolation Method
'nearest'	Nearest-neighbor interpolation (the default)
'bilinear'	Bilinear interpolation
'bicubic'	Bicubic interpolation

For example, these commands rotate an image 35° counterclockwise and use bilinear interpolation.

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



## Cropping an Image

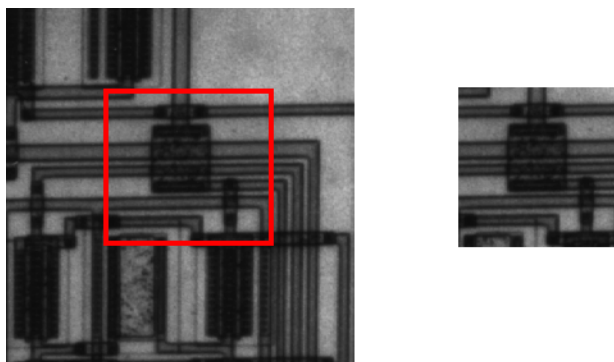
To extract a rectangular portion of an image, use the `imcrop` function. `imcrop` accepts two primary arguments:

- The image to be cropped
- The coordinates of a rectangle that defines the crop area

If you call `imcrop` without specifying the crop rectangle, you can specify the crop rectangle interactively. In this case, the cursor changes to crosshairs when it is over the image. Position the crosshairs over a corner of the crop region and press and hold the left mouse button. When you drag the crosshairs over the image you specify the rectangular crop region. `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 `imcrop` function displays the image in a figure window and waits for you to draw the cropping rectangle on the image. In the figure, the rectangle you select is shown in red. The example then calls `imshow` to view the cropped image.

```
imshow circuit.tif  
I = imcrop;  
imshow(I);
```



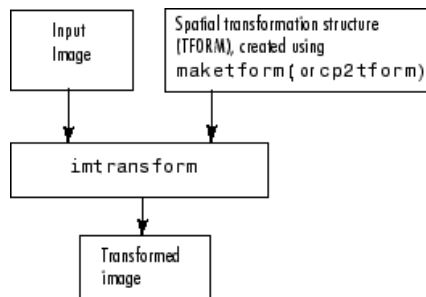
## Performing General 2-D Spatial Transformations

This section describes two toolbox functions that you can use to perform general 2-D spatial transformations. (For information about performing transformations of arrays of higher dimension, see “Performing N-Dimensional Spatial Transformations” on page 6-23.)

- `maketform`
- `imtransform`

You use the `maketform` function to define the 2-D spatial transformation you want to perform. `maketform` creates a MATLAB structure called a TFORM that contains all the parameters required to perform the transformation. You can define many types of spatial transformations in a TFORM, including affine transformations, such as translation, scaling, rotation, and shearing, projective transformations, and custom transformations. For more information, see “Creating TFORM Structures” on page 6-19. (You can also create a TFORM using the `cp2tform` function — see Chapter 7, “Image Registration”)

After you create the TFORM, you use the `imtransform` function to perform the transformation, passing `imtransform` the image to be transformed and the TFORM structure. The following figure illustrates this process. The next section provides an example that illustrates each step — see “Example: Performing a Translation” on page 6-12.



### Overview of General 2-D Spatial Transformation Process

### Example: Performing a Translation

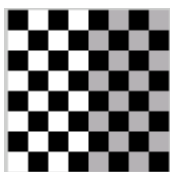
This example illustrates how to use the `maketform` and `imtransform` functions to perform a 2-D spatial transformation of an image. The example performs a simple affine transformation called a translation. In a translation, you shift an image in coordinate space by adding a specified value to the  $x$ - and  $y$ -coordinates. The example illustrates the following steps:

- “Step 1: Import the Image to Be Transformed” on page 6-12
- “Step 2: Define the Spatial Transformation” on page 6-12
- “Step 3: Create the TFORM Structure” on page 6-13
- “Step 4: Perform the Transformation” on page 6-13
- “Step 5: View the Output Image” on page 6-15

#### Step 1: Import the Image to Be Transformed

Bring the image to be transformed into the MATLAB workspace. This example creates a checkerboard image, using the `checkerboard` function. By default, `checkerboard` creates an 80-by-80 pixel image.

```
cb = checkerboard;  
imshow(cb)
```



**Original Image**

#### Step 2: Define the Spatial Transformation

You must define the spatial transformation that you want to perform. For many types of 2-D spatial transformations, such as affine transformations, you can use a 3-by-3 transformation matrix to specify the transformation. You can also use sets of points in the input and output images to specify the transformation and let `maketform` create the transformation matrix. For more information, see “Defining the Transformation Data” on page 6-17.



This example uses the following transformation matrix to define a spatial transformation called a translation.

```
xform = [ 1  0  0
          0  1  0
          40 40  1 ]
```

In this matrix, `xform(3,1)` specifies the number of pixels to shift the image in the horizontal direction and `xform(3,2)` specifies the number of pixels to shift the image in the vertical direction.

### Step 3: Create the TFORM Structure

You use the `maketform` function to create a TFORM structure. As arguments, you specify the type of transformation you want to perform and the transformation matrix (or set of points) that you created to define the transformation. For more information, see “Creating TFORM Structures” on page 6-19.

This example calls `maketform`, specifying 'affine' as the type of transformation, because translation is a type of affine transformation, and `xform`, the transformation matrix created in step 2.

```
tform_translate = maketform('affine',xform);
```

### Step 4: Perform the Transformation

To perform the transformation, call the `imtransform` function, specifying the image you want to transform and the TFORM structure that stores all the required transformation parameters. For more information, see “Performing the Spatial Transformation” on page 6-20.

The following example passes to the `imtransform` function the checkerboard image, created in Step 1, and the TFORM structure created in Step 3. `imtransform` returns the transformed image.

```
[cb_trans xdata ydata]= imtransform(cb, tform_translate);
```

The example includes two optional output arguments: `xdata` and `ydata`. These arguments return the location of the output image in output coordinate

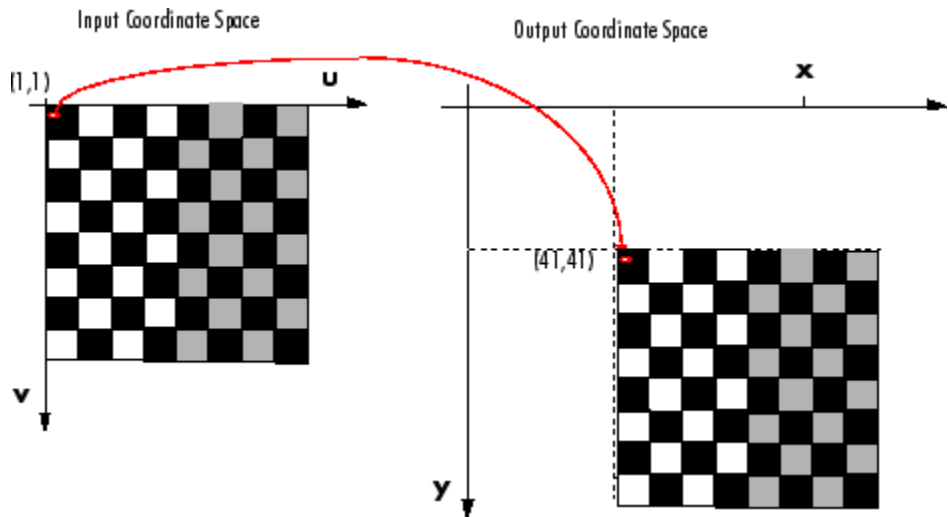
space. `xdata` contains the  $x$ -coordinates of the pixels at the corners of the output image. `ydata` contains the  $y$ -coordinates of these same pixels.

---

**Note** This section uses the spatial coordinate system when referring to pixel locations. In the spatial coordinates system, the  $x$ - and  $y$ -coordinates specify the center of the pixel. For more information about the distinction between spatial coordinates and pixel coordinates, see “Coordinate Systems” on page 2-2.

---

The following figure illustrates this translation graphically. By convention, the axes in input space are labeled  $u$  and  $v$  and the axes in output space are labelled  $x$  and  $y$ . In the figure, note how `imtransform` modifies the spatial coordinates that define the locations of pixels in the input image. The pixel at  $(1,1)$  is now positioned at  $(41,41)$ . (In the checkerboard image, each black, white, and gray square is 10 pixels high and 10 pixels wide.)



**Input Image Translated**

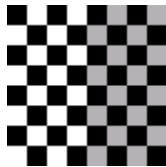
**Pixel Values and Pixel Locations.** The previous figure shows how `imtransform` changes the locations of pixels between input space and output space. The pixel located at (1,1) in the input image is now located at (41,41) in the output image. Note, however, that the value at that pixel location has not changed. Pixel (1,1) in the input image is black and so is pixel (41,41) in the output image.

`imtransform` determines the value of pixels in the output image by mapping the new locations back to the corresponding locations in the input image (inverse mapping). In a translation, because the size and orientation of the output image is the same as the input image, this is a one to one mapping of pixel values to new locations. For other types of transformations, such as scaling or rotation, `imtransform` interpolates within the input image to compute the output pixel value. For more information about the interpolation methods used by `imtransform`, see “Interpolation” on page 6-3.

### Step 5: View the Output Image

After performing the transformation, you might want to view the transformed image. The example uses the `imshow` function to display the transformed image.

```
figure, imshow(cb_trans)
```



**Translated Image**

**Understanding the Display of the Transformed Image.** When viewing the transformed image, especially for a translation operation, it might appear that the transformation had no effect. The transformed image looks identical to the original image. However, if you check the `xdata` and `ydata` values returned by `imtransform`, you can see that the spatial coordinates have changed. The upper left corner of the input image with spatial coordinates (1,1) is now (41,41). The lower right corner of the input image with spatial coordinates (80,80) is now (120,120). The value 40 has been added to each, as expected.

```
xdata =  
    41    120  
  
ydata =  
    41    120
```

The reason that no change is apparent in the visualization is because `imtransform` sizes the output image to be just large enough to contain the entire transformed image but not the entire output coordinate space. To see the effect of the translation in relation to the original image, you can use several optional input parameters that specify the size of output image and how much of the output space is included in the output image.

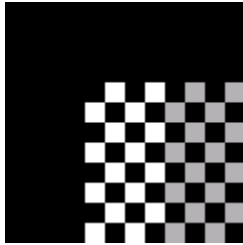
The example uses two of these optional input parameters, `XData` and `YData`, to specify how much of the output coordinate space to include in the output image. The example sets the `XData` and `YData` to include the origin of the original image and be large enough to contain the entire translated image.

---

**Note** All the pixels that are now in the output image that do not correspond to locations in the input image are black. `imtransform` assigns a value, called a *fill value*, to these pixels. This example uses the default fill value but you can specify a different one — see “Specifying Fill Values” on page 6-21.

---

```
cb_trans2 = imtransform(cb, tform_translate,...  
                        'XData',[1 (size(cb,2)+xform(3,1))],...  
                        'YData',[1 (size(cb,1)+xform(3,2))]);  
figure, imshow(cb_trans2)
```



**View of the Translated Image in Relation to Original Coordinate Space**

## Defining the Transformation Data

Before you can perform a spatial transformation, you must first define the parameters of the transformation. The following sections describe two ways you can define a spatial transformation.

- “Using a Transformation Matrix” on page 6-17
- “Using Sets of Points” on page 6-18





With either method, you pass the result to the `maketform` function to create the `TFORM` structure required by `imtransform`.

### Using a Transformation Matrix

The `maketform` function can accept transformation matrices of various sizes for N-D transformations. Because `imtransform` only performs 2-D transformations, you can only specify 3-by-3 transformation matrices.

For example, you can use a 3-by-3 matrix to specify any of the affine transformations. For affine transformations, the last column must contain `0 0 1` (`[zeros(N,1); 1]`). You can specify a 3-by-2 matrix. In this case, `imtransform` automatically adds this third column.

The following table lists the affine transformations you can perform with `imtransform` along with the transformation matrix used to define them. You can combine multiple affine transformations into a single matrix.

Affine Transform	Example	Transformation Matrix	
Translation		$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$	$t_x$ specifies the displacement along the $x$ axis $t_y$ specifies the displacement along the $y$ axis.
Scale		$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$s_x$ specifies the scale factor along the $x$ axis $s_y$ specifies the scale factor along the $y$ axis.
Shear		$\begin{bmatrix} 1 & sh_y & 0 \\ sh_x & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$sh_x$ specifies the shear factor along the $x$ axis $sh_y$ specifies the shear factor along the $y$ axis.
Rotation		$\begin{bmatrix} \cos(q) & \sin(q) & 0 \\ -\sin(q) & \cos(q) & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$q$ specifies the angle of rotation.

### Using Sets of Points

Instead of specifying a transformation matrix, you optionally use sets of points to specify a transformation and let `maketform` infer the transformation matrix.

To do this for an affine transformation, you must pick three non-collinear points in the input image and in the output image. (The points form a triangle.) For a projective transformation, you must pick four points. (The points form a quadrilateral.)

This example picks three points in the input image and three points in the output image created by the translation performed in “Example: Performing a Translation” on page 6-12. The example passes these points to `maketform` and lets `maketform` infer the transformation matrix. The three points mark three

corners of one of the checkerboard squares in the input image and the same square in the output image.

```
in_points = [11 11;21 11; 21 21]

out_points = [51 51;61 51;61 61]

tform2 = maketform('affine',inpts,outpts)
```

## Creating TFORM Structures

After defining the transformation data (“Defining the Transformation Data” on page 6-17), you must create a TFORM structure to specify the spatial transformation. You use the `maketform` function to create a TFORM structure. You pass the TFORM structure to the `imtransform` to perform the transformation. (You can also create a TFORM using the `cp2tform` function. For more information, see Chapter 7, “Image Registration”.)

The example creates a TFORM structure that specifies the parameters necessary for the translation operation.

```
tform_translate = maketform('affine',xform)
```

To create a TFORM you must specify the type of transformation you want to perform and pass in the transformation data. The example specifies 'affine' as the transformation type because translation is an affine transformation but `maketform` also supports projective transformations. In addition, by using the `custom` and `composite` options you can specify a virtually limitless variety of spatial transformations to be used with `imtransform`. The following table lists the transformation types supported by `maketform`.

Transformation Type	Description
'affine'	Transformation that can include translation, rotation, scaling, and shearing. Straight lines remain straight, and parallel lines remain parallel, but rectangles might become parallelograms.
'projective'	Transformation in which straight lines remain straight but parallel lines converge toward vanishing points. (The vanishing points can fall inside or outside the image -- even at infinity.)
'box'	Special case of an affine transformation where each dimension is shifted and scaled independently.
'custom'	User-defined transformation, providing the forward and/or inverse functions that are called by <code>imtransform</code> .
'composite'	Composition of two or more transformations.

## Performing the Spatial Transformation

Once you specify the transformation in a `TFORM` struct, you can perform the transformation by calling `imtransform`. The `imtransform` function performs the specified transformation on the coordinates of the input image and creates an output image.

The translation example called `imtransform` to perform the transformation, passing it the image to be transformed and the `TFORM` structure. `imtransform` returns the transformed image.

```
cb_trans = imtransform(cb, tform_translate);
```

`imtransform` supports several optional input parameters that you can use to control various aspects of the transformation so as the size of the output image and the fill value used. To see an example of using the `XData` and `YData` input parameters, see “Example: Performing Image Registration” on page 6-25. For more information about specifying fill values, see “Specifying Fill Values” on page 6-21.



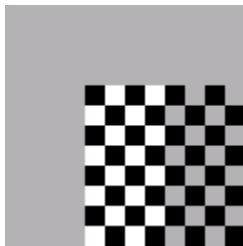
## Specifying Fill Values

When you perform a transformation, there are often pixels in the output image that are not part of the original input image. These pixels must be assigned some value, called a *fill value*. By default, `imtransform` sets these pixels to zero and they are displayed as black. Using the `FillValues` parameter with the `imtransform` function, you can specify a different color.

**Grayscale Images.** If the image being transformed is a grayscale image, you must specify a scalar value that specifies a shade of gray.

For example, in “Step 5: View the Output Image” on page 6-15, where the example displays the translated checkerboard image in relation to the original coordinate space, you can specify a fill value that matches the color of the gray squares, rather than the default black color.

```
cb_fill = imtransform(cb, tform_translate,...
                    'XData', [1 (size(cb,2)+xform(3,1))],...
                    'YData', [1 (size(cb,1)+xform(3,2))],...
                    'FillValues', .7 );
figure, imshow(cb_fill)
```



### Translated Image with Gray Fill Value

**RGB Images.** If the image being transformed is an RGB image, you can use either a scalar value or a 1-by-3 vector. If you specify a scalar, `imtransform` uses that shade of gray for each plane of the RGB image. If you specify a 1-by-3 vector, `imtransform` interprets the values as an RGB color value.

For example, this code translates an RGB image, specifying an RGB color value as the fill value. The example specifies one of the light green values in the image as the fill value.

```
rgb = imread('onion.png');
xform = [ 1 0 0
          0 1 0
          40 40 1 ]
tform_translate = maketform('affine',xform);
cb_rgb = imtransform(rgb, tform_translate,...
                    'XData', [1 (size(rgb,2)+xform(3,1))],...
                    'YData', [1 (size(rgb,1)+xform(3,2))],...
                    'FillValues', [187;192;57]);
figure, imshow(cb_rgb)
```



### Translated RGB Image with Color Fill Value

If you are transforming multiple RGB images, you can specify different fill values for each RGB image. For example, if you want to transform a series of 10 RGB images, a 4-D array with dimensions 200-by-200-by-3-by-10, you have several options. You can specify a single scalar value and use a grayscale fill value for each RGB image. You can also specify a 1-by-3 vector to use a single color value for all the RGB images in the series. To use a different color fill value for each RGB image in the series, specify a 3-by-10 array containing RGB color values.

## Performing N-Dimensional Spatial Transformations

The following functions, when used in combination, provide a vast array of options for defining and working with 2-D, N-D, and mixed-D spatial transformations:

- `maketform`
- `fliptform`
- `tformfwd`
- `tforminv`
- `findbounds`
- `makesampler`
- `tformarray`
- `imtransform`

The `imtransform`, `findbounds`, and `tformarray` functions use the `tformfwd` and `tforminv` functions internally to encapsulate the forward transformations needed to determine the extent of an output image or array and/or to map the output pixels/array locations back to input locations. You can use `tformfwd` and `tforminv` to explore the geometric effects of a transformation by applying them to points and lines and plotting the results. They support a consistent handling of both image and pointwise data.

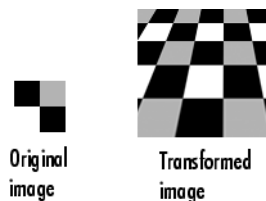
The following example, “Performing the Spatial Transformation” on page 6-20, uses the `makesampler` function with a standard interpolation method. You can also use it to obtain special effects or custom processing. For example, you could specify your own separable filtering/interpolation kernel, build a custom resampler around the MATLAB `interp2` or `interp3` functions, or even implement an advanced antialiasing technique.

And, as noted, you can use `tformarray` to work with arbitrary-dimensional array transformations. The arrays do not even need to have the same dimensions. The output can have either a lower or higher number of dimensions than the input.

For example, if you are sampling 3-D data on a 2-D slice or manifold, the input array might have a lower dimensionality. The output dimensionality might be higher, for example, if you combine multiple 2-D transformations into a single 2-D to 3-D operation.

For example, this code uses `imtransform` to perform a projective transformation of a checkerboard image.

```
I = checkerboard(20,1,1);  
figure; imshow(I)  
T = maketform('projective',[1 1; 41 1; 41 41; 1 41],...  
             [5 5; 40 5; 35 30; -10 30]);  
R = makesampler('cubic','circular');  
K = imtransform(I,T,R,'Size',[100 100],'XYScale',1);  
figure, imshow(K)
```



The `imtransform` function options let you control many aspects of the transformation. For example, note how the transformed image appears to contain multiple copies of the original image. This is accomplished by using the `'Size'` option, to make the output image larger than the input image, and then specifying a padding method that extends the input image by repeating the pixels in a circular pattern. The Image Processing Toolbox Image Transformation demos provide more examples of using the `imtransform` function and related functions to perform different types of spatial transformations.

## Example: Performing Image Registration

This example is intended to clarify the spatial relationship between the output image and the base image in image registration. The example illustrates use of the optional 'XData' and 'YData' input parameters and the optional xdata and ydata output values.

### Step 1: Read in Base and Unregistered Images

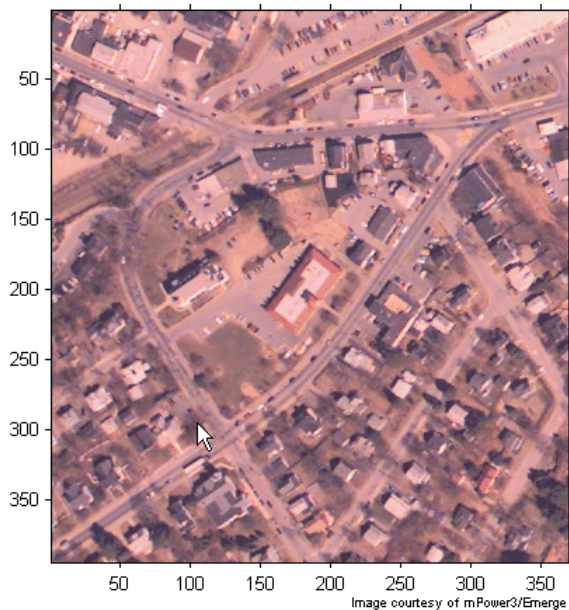
Read the base and unregistered images from sample data files that come with the Image Processing Toolbox.

```
base = imread('westconcordorthophoto.png');  
unregistered = imread('westconcordaerial.png');
```

### Step 2: Display the Unregistered Image

Display the unregistered image.

```
iptsetpref('ImshowAxesVisible','on')  
imshow(unregistered)  
text(size(unregistered,2),size(unregistered,1)+30, ...  
      'Image courtesy of mPower3/Emerge', ...  
      'FontSize',7,'HorizontalAlignment','right');
```



### Step 3: Create a TFORM Structure

Create a TFORM structure using preselected control points. Start by loading a MAT-file that contains preselected control points for the base and unregistered images.

```
load westconcordpoints
tform = cp2tform(input_points, base_points, 'projective');
```

### Step 4: Transform the Unregistered Image

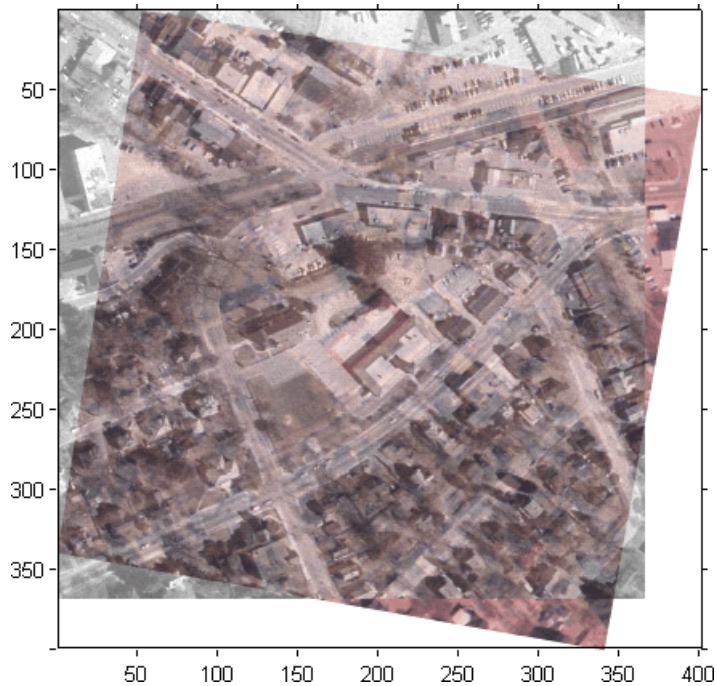
Use `imtransform` to perform the transformations necessary to register the unregistered image with the base image. This code uses the optional `FillValues` input parameter to specify a fill value (white). This fill value helps when the example overlays the transformed image, registered, on the base image to check the registration in a later step.

```
registered = imtransform(unregistered, tform,...
                        'FillValues', 255);
```

## Step 5: Overlay Registered Image Over Base Image

Overlay a semitransparent version of the registered image over the base image. Notice how the two images appear misregistered because the example assumes that the images are in the same spatial coordinate system. The next steps provide two ways to remedy this display problem.

```
figure; imshow(registered);  
hold on  
h = imshow(base, gray(256));  
set(h, 'AlphaData', 0.6)
```



**Registered Image with Base Image Overlay**

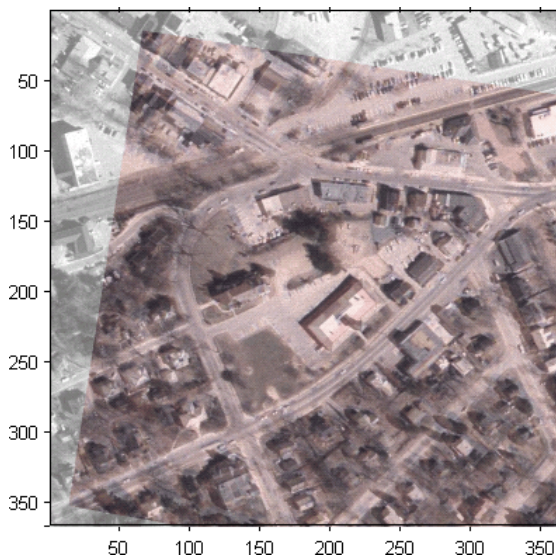
## Step 6: Using XData and YData Input Parameters

One way to ensure that the registered image appears registered with the base image is to truncate the registered image by discarding any areas that would extrapolate beyond the extent of the base image. You use the 'XData' and 'YData' parameters to do this.

```
registered1 = imtransform(unregistered,tform,...
                        'FillValues', 255,...
                        'XData', [1 size(base,2)],...
                        'YData', [1 size(base,1)]);
```

Display the registered image, overlaying a semitransparent version of the base image for comparison. The registration is evident, but part of the unregistered image has been discarded. The next step provides another solution in which the entire registered image is visible.

```
figure; imshow(registered1)
hold on
h = imshow(base, gray(256));
set(h, 'AlphaData', 0.6)
```



**Registered Image Truncated with Base Image Overlay**



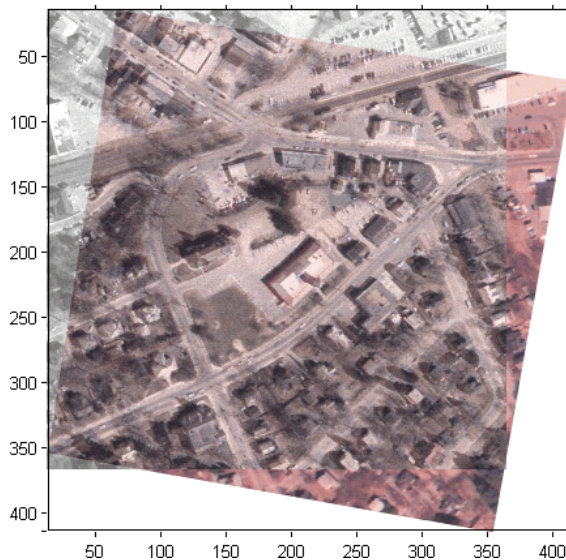
## Step 7: Using XData and YData Output Values

Another approach is to compute the full extent of the registered image and use the optional `imtransform` syntax that returns the  $x$ - and  $y$ -coordinates that indicate the transformed image's position relative to the base image's pixel coordinates.

```
[registered2 xdata ydata] = imtransform(unregistered, tform,...
                                       'FillValues', 255);
```

Display the registered image. Overlay a semi-transparent version of the base image for comparison. Adjust the axes to include the full base image. In this case, notice how the registration is evident and the full extent of both images is visible as well.

```
figure; imshow(registered2, 'XData', xdata, 'YData', ydata)
hold on
h = imshow(base, gray(256));
set(h, 'AlphaData', 0.6)
ylim = get(gca, 'YLim');
set(gca, 'YLim', [0.5 ylim(2)])
```





# Image Registration

---

This chapter describes the image registration capabilities of the Image Processing Toolbox. Image registration is the process of aligning two or more images of the same scene. Image registration is often used as a preliminary step in other image processing applications.

Registering an Image (p. 7-2)	Steps you through an example of the image registration process
Types of Supported Transformations (p. 7-11)	Lists the types of supported transformations
Selecting Control Points (p. 7-13)	Describes how to use the Control Point Selection Tool (cpselect) to select control points in pairs of images
Using Correlation to Improve Control Points (p. 7-30)	Describes how to use the cpcorr function to fine-tune your control point selections

## Registering an Image

Image registration is the process of aligning two or more images of the same scene. Typically, one image, called the *base* image or *reference* image, is considered the reference to which the other images, called *input* images, are compared. The object of image registration is to bring the input image into alignment with the base image by applying a spatial transformation to the input image. The differences between the input image and the output image might have occurred as a result of terrain relief and other changes in perspective when imaging the same scene from different viewpoints. Lens and other internal sensor distortions, or differences between sensors and sensor types, can also cause distortion.

A spatial transformation maps locations in one image to new locations in another image. (For more details, see Chapter 6, “Spatial Transformations”) Determining the parameters of the spatial transformation needed to bring the images into alignment is key to the image registration process.

Image registration is often used as a preliminary step in other image processing applications. For example, you can use image registration to align satellite images of the earth’s surface or images created by different medical diagnostic modalities (MRI and SPECT). After registration, you can compare features in the images to see how a river has migrated, how an area is flooded, or to see if a tumor is visible in an MRI or SPECT image.

### Point Mapping

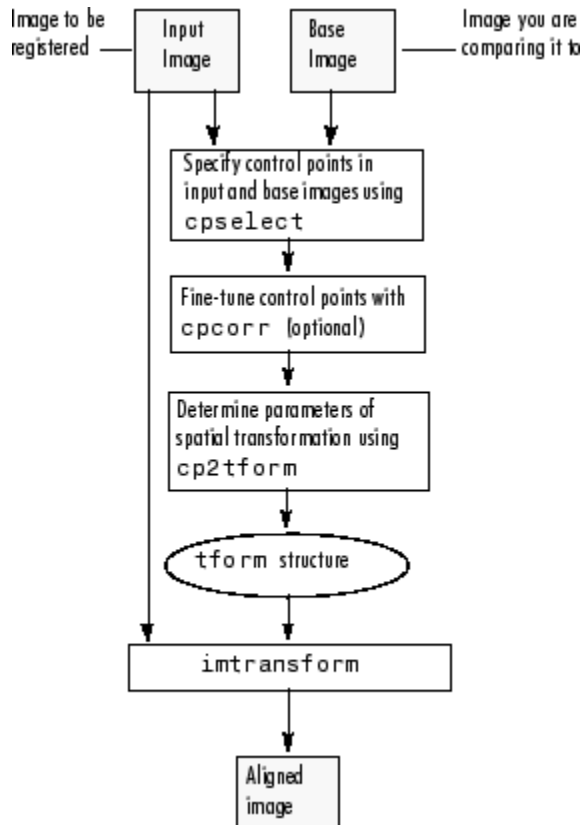
The Image Processing Toolbox provides tools to support point mapping to determine the parameters of the transformation required to bring an image into alignment with another image. In point mapping, you pick points in a pair of images that identify the same feature or landmark in the images. Then, a spatial mapping is inferred from the positions of these control points.

---

**Note** You might need to perform several iterations of this process, experimenting with different types of transformations, before you achieve a satisfactory result. In some cases, you might perform successive registrations, removing gross global distortions first, and then removing smaller local distortions in subsequent passes.

---

The following figure provides a graphic illustration of this process. This process is best understood by looking at an example. See “Example: Registering to a Digital Orthophoto” on page 7-4 for an extended example.



**Overview of Image Registration Process**

## **Example: Registering to a Digital Orthophoto**

This example illustrates the steps involved in performing image registration using point mapping. These steps include:

- 1** Read the images into the MATLAB workspace.
- 2** Specify control point pairs in the images.
- 3** Save the control point pairs.
- 4** Fine-tune the control points using cross-correlation. (This is an optional step.)
- 5** Specify the type of transformation to be used and infer its parameters from the control point pairs.
- 6** Transform the unregistered image to bring it into alignment.

To illustrate this process, the example registers a digital aerial photograph to a digital orthophoto covering the same area. Both images are centered on the business district of West Concord, Massachusetts.

The aerial image is geometrically uncorrected: it includes camera perspective, terrain and building relief, and internal (lens) distortions, and it does not have any particular alignment or registration with respect to the earth.

The orthophoto, supplied by the Massachusetts Geographic Information System (MassGIS), has been orthorectified to remove camera, perspective, and relief distortions (via a specialized image transformation process). It is also georegistered (and geocoded)--the columns and rows of the digital orthophoto image are aligned to the axes of the Massachusetts State Plane coordinate system, each pixel center corresponds to a definite geographic location, and every pixel is 1 meter square in map units.

## Step 1: Read the Images into MATLAB

In this example, the base image is `westconcordorthophoto.png`, the MassGIS georegistered orthophoto. It is a panchromatic (grayscale) image. The image to be registered is `westconcordaerial.png`, a digital aerial photograph supplied by mPower3/Emerge, and is a visible-color RGB image.

```
orthophoto = imread('westconcordorthophoto.png');  
figure, imshow(orthophoto)  
unregistered = imread('westconcordaerial.png');  
figure, imshow(unregistered)
```

You do not have to read the images into the MATLAB workspace. The `cpslect` function accepts file specifications for grayscale images. However, if you want to use cross-correlation to tune your control point positioning, the images must be in the workspace.



Aerial Photo Image

Image Courtesy of mPower3/Emerge



Orthophoto Image

Image Courtesy of MassGIS

## Step 2: Choose Control Points in the Images

The toolbox provides an interactive tool, called the Control Point Selection Tool, that you can use to pick pairs of corresponding control points in both images. Control points are landmarks that you can find in both images, like a road intersection, or a natural feature.

To start this tool, enter `cpselect` at the MATLAB prompt, specifying as arguments the input and base images.

---

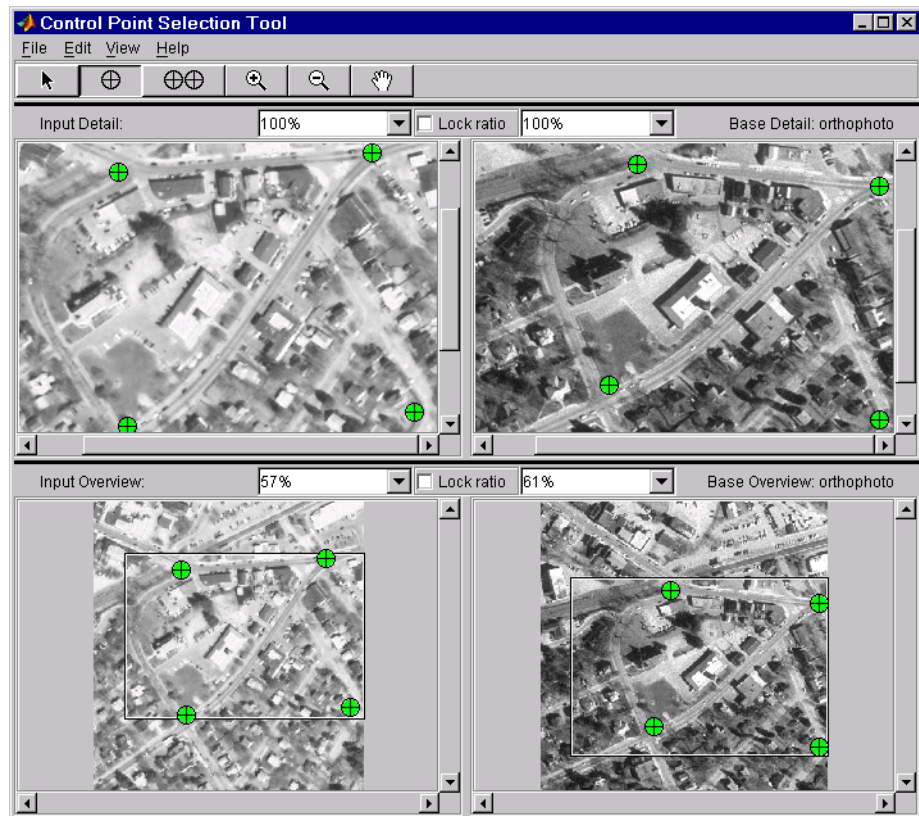
**Note** The unregistered image is an RGB image. Because the Control Point Selection Tool only accepts grayscale images, the example passes only one plane of the color image to `cpselect`.

---

```
cpselect(unregistered(:,:,1),orthophoto)
```

The `cpselect` function displays two views of both the input image and the base image in which you can pick control points by pointing and clicking. For more information, see “Selecting Control Points” on page 7-13. This figure shows the Control Point Selection Tool with four pairs of control points selected. The number of control point pairs you pick is at least partially determined by the type of transformation you want to perform (specified in Step 5). See “Types of Supported Transformations” on page 7-11 for information about the minimum number of points required by each transformation.





### Step 3: Save the Control Point Pairs to the MATLAB Workspace

In the Control Point Selection Tool, click the **File** menu and choose the **Save Points to Workspace** option. See “Saving Control Points” on page 7-28 for more information.

For example, the Control Point Selection Tool returns the following set of control points in the input image. These values represent spatial coordinates; the left column are  $x$ -coordinates, the right column are  $y$ -coordinates.

```
input_points =  
 120.7086  93.9772  
 319.2222  78.9202  
 127.9838 291.6312  
 352.0729 281.1445
```

#### **Step 4: Fine-Tune the Control Point Pair Placement**

This is an optional step that uses cross-correlation to adjust the position of the control points you selected with `cpselect`. See “Using Correlation to Improve Control Points” on page 7-30 for more information.

---

**Note** `cpcorr` can only adjust points for images that are the same scale and have the same orientation. Because the Concord image is rotated in relation to the base image, `cpcorr` cannot tune the control points. When it cannot tune the points, `cpcorr` returns the input points unmodified.

---

```
input_points_corr = cpcorr(input_points,base_points,...  
                          unregistered(:,:,1),orthophoto)  
  
input_points_corr =  
 120.7086  93.9772  
 319.2222  78.9202  
 127.1046 289.8935  
 352.0729 281.1445
```

#### **Step 5: Specify the Type of Transformation and Infer Its Parameters**

In this step, you pass the control points to the `cp2tform` function that determines the parameters of the transformation needed to bring the image into alignment. `cp2tform` is a data-fitting function that determines the transformation based on the geometric relationship of the control points. `cp2tform` returns the parameters in a geometric transformation structure, called a `TFORM` structure.

When you use `cp2tform`, you must specify the type of transformation you want to perform. The `cp2tform` function can infer the parameters for five types of transformations. You must choose which transformation will correct the type of distortion present in the input image. See “Types of Supported Transformations” on page 7-11 for more information. Images can contain more than one type of distortion.

The predominant distortion in the aerial image of West Concord (the input image) results from the camera perspective. Ignoring terrain relief, which is minor in this area, image registration can correct for this using a projective transformation. The projective transformation also rotates the image into alignment with the map coordinate system underlying the base digital orthophoto image. (Given sufficient information about the terrain and camera, you could correct these other distortions at the same time by creating a composite transformation with `maketform`. See “Performing General 2-D Spatial Transformations” on page 6-11 for more information.)

```
mytform = cp2tform(input_points,base_points,'projective');
```

### **Step 6: Transform the Unregistered Image**

As the final step in image registration, transform the input image to bring it into alignment with the base image. You use `imtransform` to perform the transformation, passing it the input image and the `TFORM` structure, which defines the transformation. `imtransform` returns the transformed image. For more information about using `imtransform`, see Chapter 6, “Spatial Transformations”

```
registered = imtransform(unregistered,mytform)
```

---

**Note** `imtransform` applies the transformation defined in `mytform`, which is based on control points picked in only one plane of the RGB image, to all three planes of the input image.

---

Compare the transformed image to the base image to see how the registration came out.



Registered Image



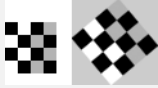

Orthophoto Image





## Types of Supported Transformations

The `cp2tform` function can infer the parameters for six types of transformations. This table lists the transformations in order of complexity, with examples of each type of distortion.

The first four transformations, 'linear conformal', 'affine', 'projective', and 'polynomial' are global transformations. In these transformations, a single mathematical expression applies to an entire image. The last two transformations, 'piecewise linear' and 'lwm' (local weighted mean), are local transformations. In these transformations, different mathematical expressions apply to different regions within an image.

When exploring how different transformations affect the images you are working with, try the global transformations first. If these transformations are not satisfactory, try the local transformations: the piecewise linear transformation first and then the local weighted mean transformation.

Transformation Type	Description	Minimum Control Points	Example
'linear conformal'	Use this transformation when shapes in the input image are unchanged, but the image is distorted by some combination of translation, rotation, and scaling. Straight lines remain straight, and parallel lines are still parallel.	2 pairs	
'affine'	Use this transformation when shapes in the input image exhibit shearing. Straight lines remain straight, and parallel lines remain parallel, but rectangles become parallelograms.	3 pairs	

Transformation Type	Description	Minimum Control Points	Example
'projective'	Use this transformation when the scene appears tilted. Straight lines remain straight, but parallel lines converge toward vanishing points (which might or might not fall within the image).	4 pairs	
'polynomial'	Use this transformation when objects in the image are curved. The higher the order of the polynomial, the better the fit, but the result can contain more curves than the base image.	6 pairs (order 2) 10 pairs (order 3) 16 pairs (order 4)	
'piecewise linear'	Use this transformation when parts of the image appear distorted differently.	4 pairs	
'lwm'	Use this transformation (local weighted mean), when the distortion varies locally and piecewise linear is not sufficient.	6 pairs (12 pairs recommended)	

## Selecting Control Points

The toolbox includes an interactive tool that enables you to specify control points in the images you want to register. The tool displays the images side by side. When you are satisfied with the number and placement of the control points, you can save the control points.

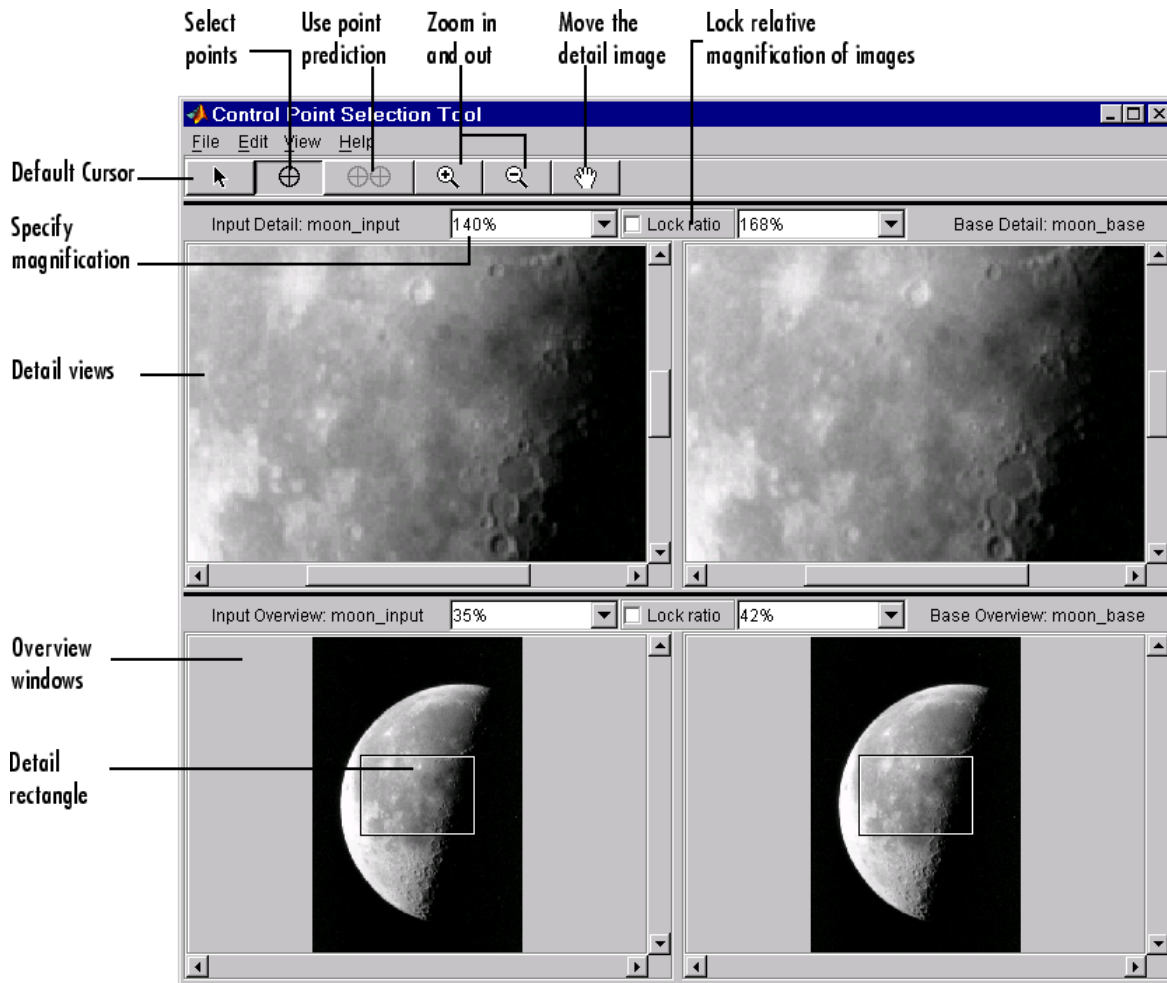
### Using the Control Point Selection Tool

To specify control points in a pair of images you want to register, use the Control Point Selection Tool, `cpselect`. The tool displays the image you want to register, called the *input* image, next to the image you want to compare it to, called the *base* image or *reference* image.

Specifying control points is a four-step process:

- 1** Start the tool, specifying the input image and the base image.
- 2** View the images, looking for visual elements that you can identify in both images. `cpselect` provides many ways to navigate around the image, panning and zooming to view areas of the image in more detail.
- 3** Specify matching control point pairs in the input image and the base image.
- 4** Save the control points in the MATLAB workspace.

The following figure shows the default appearance of the tool when you first start it.



**Control Point Selection Tool**



## Starting the Control Point Selection Tool

To use the Control Point Selection Tool, enter the `cpselect` command at the MATLAB prompt. As arguments, specify the image you want to register (the input image), and the image you want to compare it to (the base image).

To illustrate, this code fragment reads an image into a variable, `moon_base`, in the MATLAB workspace. It then creates another version of the image with a deliberate size distortion, called `moon_input`. This is the image that needs registration to remove the size distortion. The code then starts the `cpselect` tool, specifying the two images.

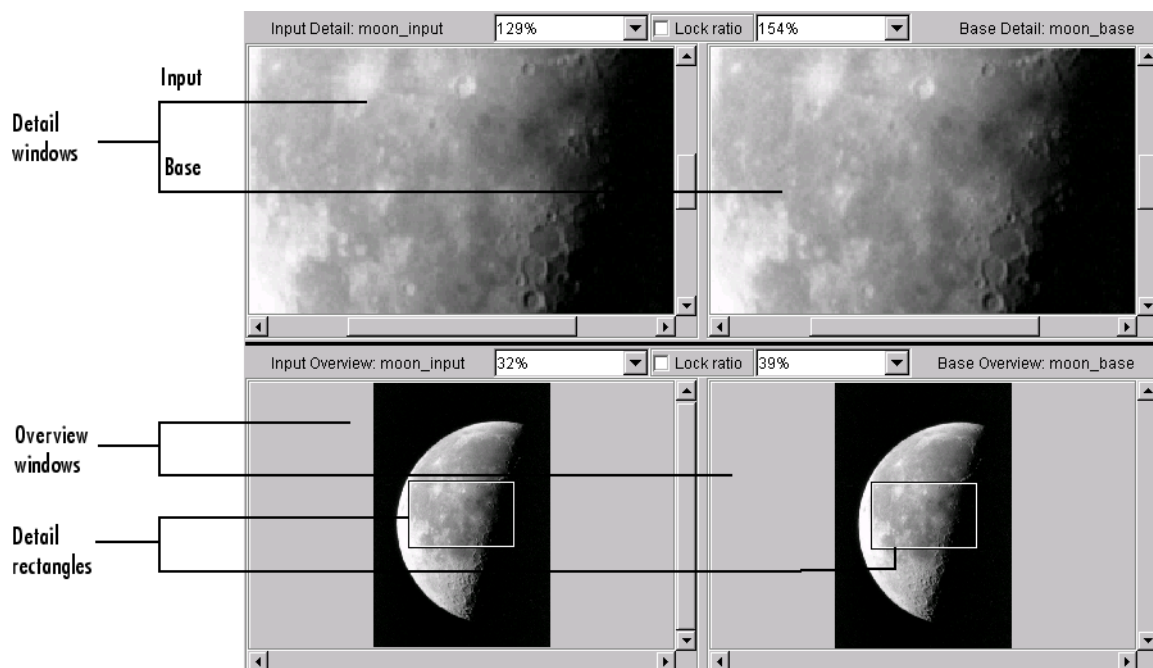
```
moon_base = imread('moon.tif');  
moon_input = imresize(moon_base, 1.2);  
cpselect(moon_input, moon_base);
```

The `cpselect` command has other optional arguments. For example, you can restart a control point selection session by including a `cpstruct` structure as the third argument. For more information about restarting sessions, see “Saving Control Points” on page 7-28. For complete details, see the `cpselect` reference page.

## Default Views of the Images

When the Control Point Selection Tool starts, it contains four image display windows. The top two windows are called the Detail windows. These windows show a closeup view of a portion of the images you are working with. The input image is on the left and the base image is on the right. The two windows at the bottom of the interface are called the Overview windows. These windows show the images in their entirety, at the largest scale that fits the window. The input overview image is on the left and the base overview image is on the right.

Superimposed on the image in the Overview windows is a rectangle, called the detail rectangle. This rectangle defines the part of the image that is visible in the Detail window. By default, at startup, the detail rectangle covers one quarter of the entire image and is positioned over the center of the image.



## Viewing the Images

By default, cpselect displays the entire base and input images in the Overview windows and displays a closeup view of a portion of these images in the Detail windows. However, to find visual elements that are common to both images, you might want to change the section of the image displayed in the detail view or zoom in on a part of the image to view it in more detail. The following sections describe the different ways to change your view of the images:

- “Using Scroll Bars to View Other Parts of an Image” on page 7-17
- “Using the Detail Rectangle to Change the View” on page 7-17
- “Panning the Image Displayed in the Detail Window” on page 7-18
- “Zooming In and Out on an Image” on page 7-18
- “Specifying the Magnification of the Images” on page 7-19

- “Locking the Relative Magnification of the Input and Base Images” on page 7-20

### Using Scroll Bars to View Other Parts of an Image



To view parts of an image that are not visible in the Detail or Overview windows, use the scroll bars provided in each window.

As you scroll the image in the Detail window, note how the detail rectangle moves over the image in the Overview window. The position of the detail rectangle always shows the portion of the image in the Detail window.

### Using the Detail Rectangle to Change the View

To get a closer view of any part of the image, move the detail rectangle in the Overview window over that section of the image. `cpselect` displays that section of the image in the Detail window at a higher magnification than the overview window.

To move the detail rectangle,

- 1 Click the **Default Cursor** button  in the toolbar.
- 2 Move the pointer into the detail rectangle. The cursor changes to the fleur shape, .
- 3 Press and hold the mouse button to drag the detail rectangle anywhere on the image.

---



**Note** As you move the detail rectangle over the image in the Overview window, the view of the image displayed in the Detail window changes.

---

## Panning the Image Displayed in the Detail Window

To change the section of the image displayed in the Detail window, use the pan tool to move the image in the window.

To use the pan tool,

- 1 Click the Drag Images to **Pan** button  in the toolbar.
- 2 Move the pointer over the image in the Detail window. The cursor changes to the fleur shape, .
- 3 Press and hold the mouse button and drag the image in the Detail window.

---

**Note** As you move the image in the Detail window, the detail rectangle in the Overview window moves.

---


## Zooming In and Out on an Image

To enlarge an image to get a closer look or shrink an image to see the whole image in context, use the Zoom buttons on the button bar. (You can also zoom in or out on an image by changing the magnification. See “Specifying the Magnification of the Images” on page 7-19 for more information.)

To zoom in or zoom out on the base or input images,

- 1 Click the appropriate magnifying glass button.



- 2 Move the pointer over the image you want to zoom in or out on. The cursor changes to crosshairs, .

You can zoom in or out on either the input or the base images, in either the Detail or Overview windows. To keep the relative magnifications of the base and input images synchronized, click the **Lock ratio** check box. See

“Locking the Relative Magnification of the Input and Base Images” on page 7-20 for more information.

---

**Note** If you zoom in close on the image displayed in the Overview window, the detail rectangle might no longer be visible.

---

You can use the zoom tool in two ways:

- Position the cursor over a location in the image and click the mouse. With each click, `cpselect` changes the magnification of the image by a preset amount. (See “Specifying the Magnification of the Images” on page 7-19 for a list of some of these magnifications.) `cpselect` centers the new view of the image on the spot where you clicked.
- Alternatively, you can position the cursor over a location in the image and, while pressing and holding the mouse button, draw a rectangle defining the area you want to zoom in or out on. `cpselect` magnifies the image so that the chosen section fills the Detail window. `cpselect` resizes the detail rectangle in the **Overview** window as well.

---

**Note** When you zoom in or out on an image, notice how the magnification value changes.

---

## Specifying the Magnification of the Images

To enlarge an image to get a closer look or to shrink an image to see the whole image in context, use the magnification edit box. (You can also use the Zoom buttons to enlarge or shrink an image. See “Zooming In and Out on an Image” on page 7-18 for more information.)

To change the magnification of an image,

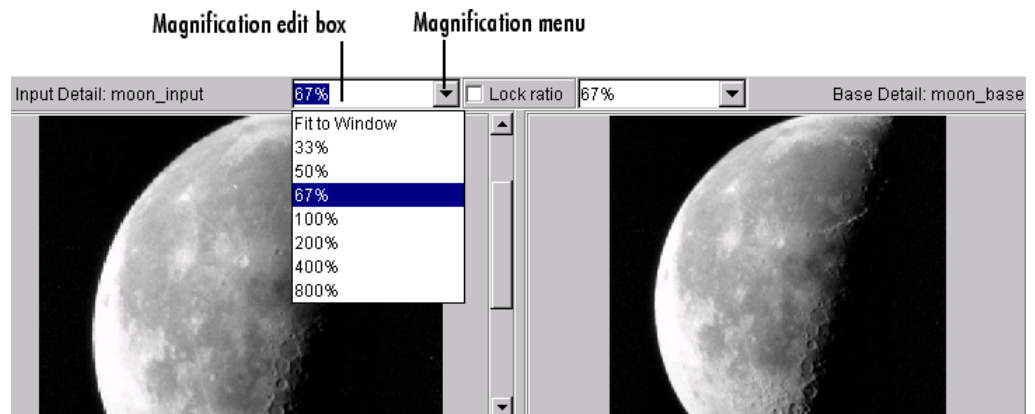
- 1 Move the cursor into the magnification edit box of the window you want to change. The cursor changes to the text entry cursor.

---

**Note** Each Detail window and Overview window has its own magnification edit box.

---

- 2 Type a new value in the magnification edit box and press **Enter**, or click the menu associated with the edit box and choose from a list of preset magnifications. `cpselect` changes the magnification of the image and displays the new view in the appropriate window.



## Locking the Relative Magnification of the Input and Base Images

To keep the relative magnification of the input and base images automatically synchronized in the Detail or Overview windows, click the **Lock Ratio** check box. The two Detail windows and the two Overview windows each have their own **Lock ratio** check boxes.

When the **Lock Ratio** check box is selected, cpselect changes the magnification of *both* the input and base images when you zoom in or out on either one of the images or specify a magnification value for either of the images.



## Specifying Matching Control Point Pairs

The primary function of the Control Point Selection Tool is to enable you to pick control points in the image to be registered, the input image, and the image to which you are comparing it, the base image. When you start cpselect, the point selection tool is enabled, by default.



You specify control points by pointing and clicking in the input and base images, in either the Detail or the Overview windows. Each point you specify in the input image must have a match in the base image. The following sections describe the ways you can use the Control Point Selection Tool to choose control point pairs:

- “Picking Control Point Pairs Manually” on page 7-21
- “Using Control Point Prediction” on page 7-23

This section also describes how to move control points after you’ve created them and how to delete control points.

### Picking Control Point Pairs Manually

To specify a pair of control points in your images,

- 1 Click the **Control Point Selection** button . Control point selection mode is active by default.
- 2 Position the cursor over a feature you have visually selected in any of the images displayed. The cursor changes to a pointing finger .

You can pick control points in either of the Detail windows, input or base, or in either of the Overview windows, input or base. You also can work in either direction: input-to-base image, or base-to-input image.

- 3 Click the mouse button. `cpselect` places a control point symbol at the position you specified, in both the Detail window and the Overview window. (The appearance of the control point symbol indicates its current state. Initially, control points are in an active, unmatched state. See “Control Point States” on page 7-25 for more information.

---

**Note** Depending on where in the image you pick control points, the symbol for the point might be visible in the Overview window, but not in the Detail window.

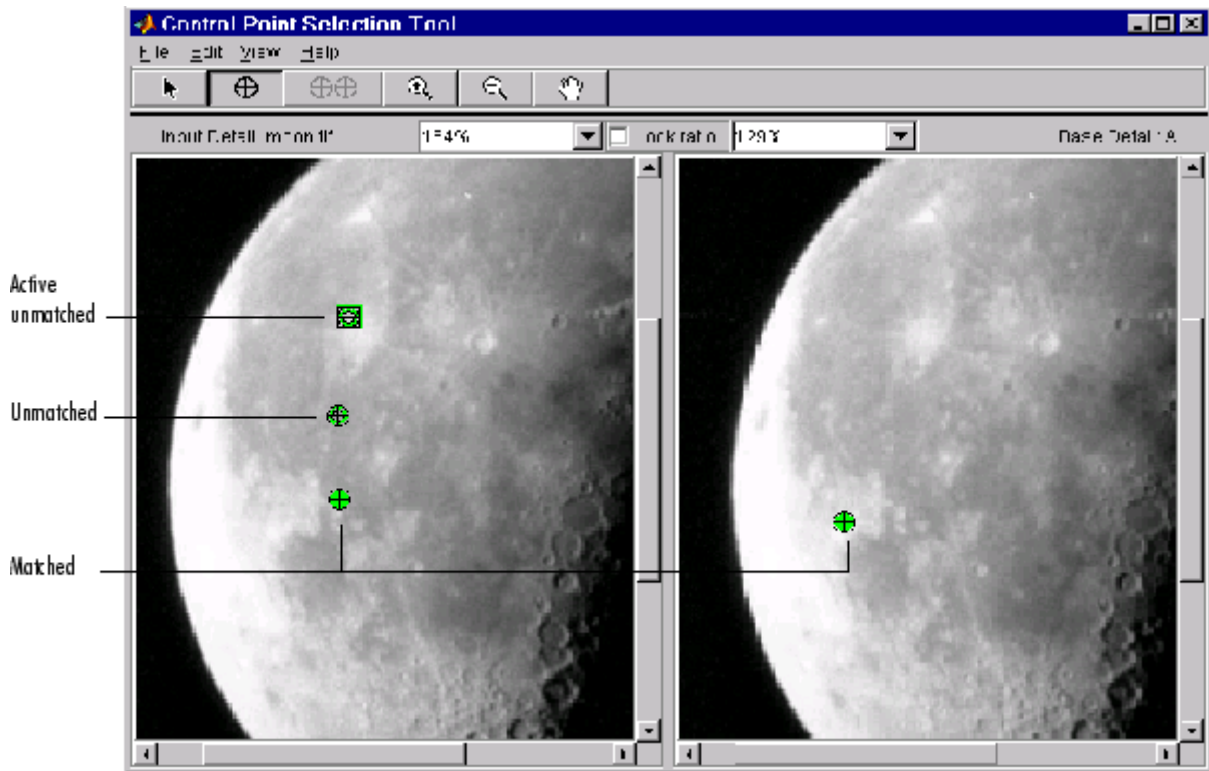
---

- 4 To create the match for this control point, move the cursor into the corresponding Detail or Overview window. For example, if you started in an input window, move the cursor to a base window.
- 5 Click the mouse button. `cpselect` places a control point symbol at the position you specified, in both the Detail and Overview windows. Because this control point completes a pair, the appearance of this symbol indicates an active, matched state. Note that the appearance of the first control point you selected (in step 3) also changes to an active, matched state.

You pick pairs of control points by moving from a view of the input image to a view of the base image, or vice versa. You can pick several control points in one view of the image, and then move to the corresponding window to locate their matches. To match an unmatched control point, select it to make it active, and then pick a point in the corresponding view window. When you select a match for a control point, the symbols for both points change to indicate their matched state. You can move or delete control points after you create them.

The following figure illustrates control points in several states.





### Using Control Point Prediction

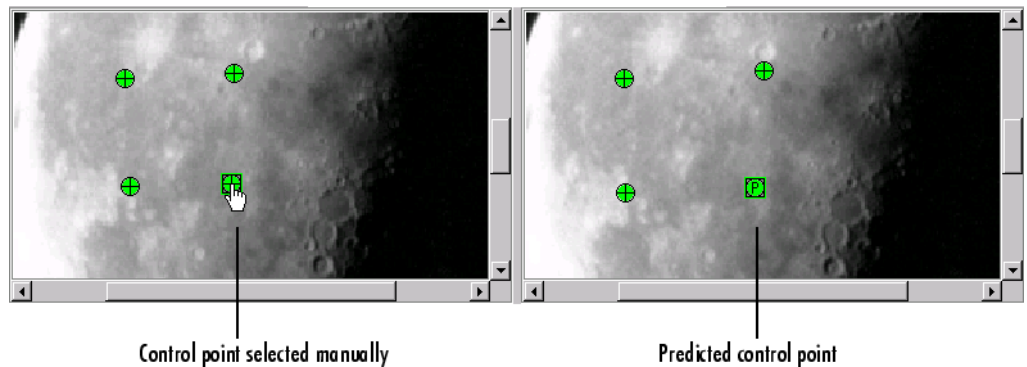
Instead of picking matching control points by moving the cursor between corresponding Detail or Overview windows, you can let the Control Point Selection Tool estimate the match for the control points you specify, automatically. The Control Point Selection Tool determines the position of the matching control point based on the geometric relationship of the previously selected control points.

---

**Note** By default, the Control Point Selection Tool does not include predicted points in the set of valid control points returned in `input_points` or `base_points`. To include predicted points, you must accept them by selecting the points and fine-tuning their position with the cursor. When you move a predicted point, the Control Point Selection Tool changes the symbol to indicate that it has changed to a standard control point. For more information, see “Moving Control Points” on page 7-26.

---

To illustrate point prediction, this figure shows four control points selected in the input image, where the points form the four corners of a square. (The control points selections in the figure do not attempt to identify any landmarks in the image.) The figure shows the picking of a fourth point, in the left window, and the corresponding predicted point in the right window. Note how the Control Point Selection Tool places the predicted point at the same location relative to the other control points, forming the bottom right corner of the square.





---

**Note** Because the Control Point Selection Tool predicts control point locations based on the locations of the previous control points, you cannot use point prediction until you have a minimum of two pairs of matched points. Until this minimum is met, the Control Point Prediction button is disabled.

---

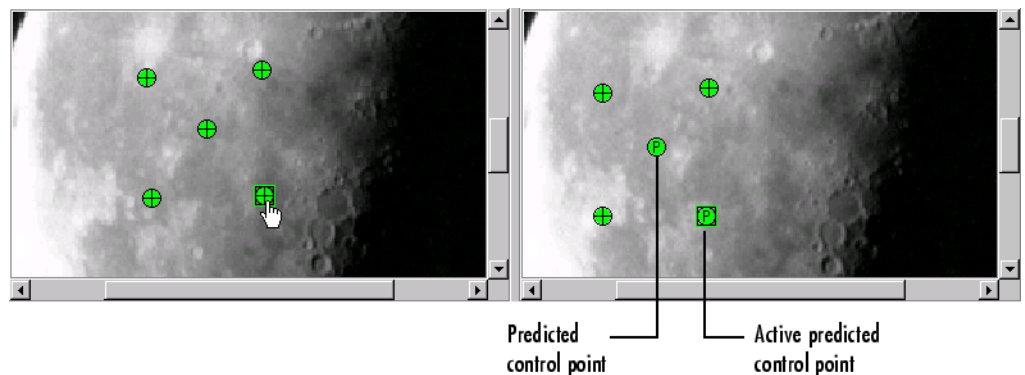
To use control point prediction,

- 1 Click the **Control Point Prediction** button .
- 2 Position the cursor anywhere in any of the images displayed. The cursor changes to a pointing finger, .

You can pick control points in either of the Detail windows, input or base, or in either of the Overview windows, input or base. You also can work in either direction: input-to-base image or base-to-input image.

- 3 Click either mouse button. The Control Point Selection Tool places a control point symbol at the position you specified and places another control point symbol for a matching point in all the other windows. The symbol for the predicted point contains the letter "P," indicating that it's a predicted control point.

This figure illustrates predicted points in active unmatched, matched, and predicted states. For a complete description of all point states, see “Control Point States” on page 7-25.









### Control Point States

The appearance of control point symbols indicates their current state. When you first pick a control point, its state is active and unmatched. When you pick the match for a control point, the appearance of both symbols changes to indicate their matched status.



This table lists all the possible control point states with their symbols. cpselect displays this list in a separate window called a **Legend**. The Legend is visible by default, but you can control its visibility using the **Legend** option from the **View** menu.

### Control Point States

Symbol	State	Description
	Active unmatched	The point is currently selected but does not have a matching point. This is the initial state of most points.
	Active matched	The point is currently selected and has a matching point.
	Active predicted	The point is a predicted point. If you move its position, the point changes to active matched state.
	Unmatched	The point is not selected and it is unmatched. You must select it before you can create its matching point.
	Matched	The point has a matching point.
	Predicted	This point was added by cpselect during point prediction.

### Moving Control Points



To move a control point,

- 1 Click the **Control Point Selection** button  or the **Default Cursor** button .
- 2 Position the cursor over the control point you want to move.
- 3 Press and hold the mouse button and drag the control point. The state of the control point changes to active when you move it.

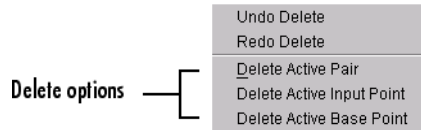
If you move a predicted control point, the state of the control point changes to a regular (nonpredicted) control point.

## Deleting Control Points

To delete a control point, and optionally its matching point,

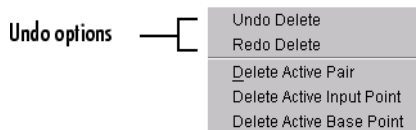
- 1 Click the **Control Point Selection** button  or the **Default Cursor** button .
- 2 Click the control point you want to delete. Its state changes to active. If the control point has a match, both points become active.
- 3 Delete the point (or points) using one of these methods:
  - Pressing the **Backspace** key
  - Pressing the **Delete** key
  - Choosing one of the delete options from the **Edit** menu

Using this menu you can delete individual points or pairs of matched points, in the input or base images.



## Undoing and Redoing Control Point Selections

You can undo a deletion or series of deletions using the **Undo Delete** option on the cpselect **Edit** menu.



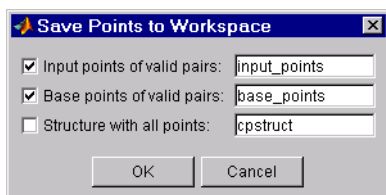
After undoing a deletion, you can delete the points again using the **Redo** option, also on the **Edit** menu.

## Saving Control Points

After you specify control point pairs, you must save them in the MATLAB workspace to make them available for the next step in image registration, processing by `cp2tform`.

To save control points to the MATLAB workspace,

- 1 Select **File** on the Control Point Selection Tool menu bar.
- 2 Choose the **Save Points to Workspace** option. The Control Point Selection Tool displays this dialog box:



By default, the Control Point Selection Tool saves the  $x$ -coordinates and  $y$ -coordinates that specify the locations of the control points you selected in two arrays named `input_points` and `base_points`, although you can specify other names. These are  $n$ -by-2 arrays, where  $n$  is the number of valid control point pairs you selected. For example, this is an example of the `input_points` array if you picked four pairs of control points. The values in the left column represent the  $x$ -coordinates; the values in the right column represent the  $y$ -coordinates.

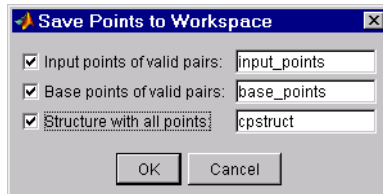
```
input_points =

    215.6667    262.3333
    225.7778    311.3333
    156.5556    340.1111
    270.8889    368.8889
```

Whenever you exit the Control Point Selection Tool, it asks if you want to save your control points.

## Saving Your Control Point Selection Session

To save the current state of the Control Point Selection Tool, select the **Structure with all points** check box in the Save Points to Workspace dialog box.



This option saves the positions of all the control points you specified and their current states in a cpstruct structure.

```
cpstruct =

    inputPoints: [4x2 double]
    basePoints: [4x2 double]
    inputBasePairs: [4x2 double]
    ids: [4x1 double]
    inputIdPairs: [4x2 double]
    baseIdPairs: [4x2 double]
    isInputPredicted: [4x1 double]
    isBasePredicted: [4x1 double]
```

You can use the cpstruct to restart a control point selection session at the point where you left off.

This option is useful if you are picking many points over a long time and want to preserve unmatched and predicted points when you resume work. The Control Point Selection Tool does not include unmatched and predicted points in the input\_points and base\_points arrays.

To extract the arrays of valid control point coordinates from a cpstruct, use the cpstruct2pairs function.

## Using Correlation to Improve Control Points

You might want to fine-tune the control points you selected using `cpselect`. Using cross-correlation, you can sometimes improve the points you selected by eye using the Control Point Selection Tool.

To use cross-correlation, pass sets of control points in the input and base images, along with the images themselves, to the `cpcorr` function.

```
input_pts_adj= cpcorr(input_points, base_points, input, base);
```

The `cpcorr` function defines 11-by-11 regions around each control point in the input image and around the matching control point in the base image, and then calculates the correlation between the values at each pixel in the region. Next, the `cpcorr` function looks for the position with the highest correlation value and uses this as the optimal position of the control point. The `cpcorr` function only moves control points up to 4 pixels based on the results of the cross-correlation.

---

**Note** Features in the two images must be at the same scale and have the same orientation. They cannot be rotated relative to each other.

---

If `cpcorr` cannot correlate some of the control points, it returns their values in `input_points` unmodified.



# Linear Filtering and Filter Design

---

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.

Linear Filtering (p. 8-2)

Provides 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.

Filter Design (p. 8-15)

Discusses designing two-dimensional finite impulse response (FIR) filters. This section assumes you are familiar with working in the frequency domain.

## Linear Filtering

Filtering is a technique for modifying or enhancing an image. For example, you can filter an image to emphasize certain features or remove other features. Image processing operations implemented with filtering include smoothing, sharpening, and edge enhancement.

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 15, "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.

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

- A description of filtering, using convolution and correlation.
- A description of how to perform filtering using the `imfilter` function
- A discussion about using predefined filter types

See "Filter Design" on page 8-15 for information about how to design filters.

### Convolution

Linear filtering of an image is accomplished through an operation called *convolution*. Convolution is a neighborhood operation in which each output pixel is the weighted sum of neighboring input pixels. The matrix of weights is called the *convolution kernel*, also known as the *filter*. A convolution kernel is a correlation kernel that has been rotated 180 degrees.

For example, suppose the image is

$$A = \begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \end{bmatrix}$$

10 12 19 21 3  
11 18 25 2 9]

and the convolution kernel is

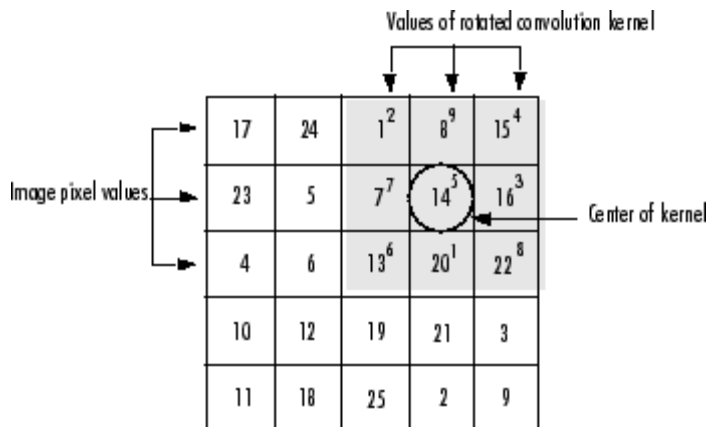
h = [8 1 6  
3 5 7  
4 9 2]

The following figure shows how to compute the (2,4) output pixel using these steps:

- 1 Rotate the convolution kernel 180 degrees about its center element.
- 2 Slide the center element of the convolution kernel so that it lies on top of the (2,4) element of A.
- 3 Multiply each weight in the rotated convolution kernel by the pixel of A underneath.
- 4 Sum the individual products from step 3.

Hence the (2,4) output pixel is

$$1 \cdot 2 + 8 \cdot 9 + 15 \cdot 4 + 7 \cdot 7 + 14 \cdot 5 + 16 \cdot 3 + 13 \cdot 6 + 20 \cdot 1 + 22 \cdot 8 = 575$$



**Computing the (2,4) Output of Convolution**

## Correlation

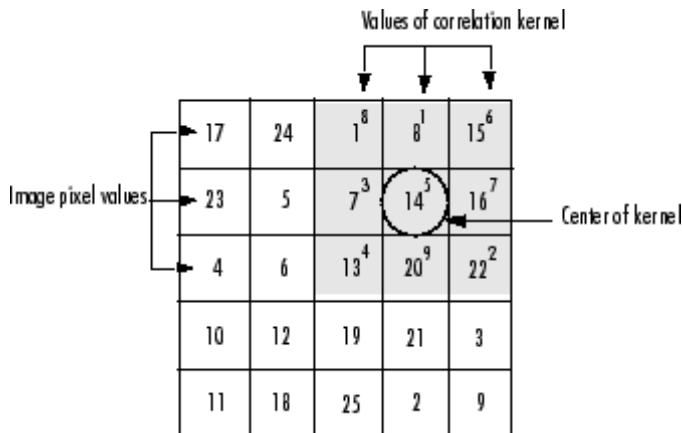
The operation called *correlation* is closely related to convolution. In correlation, the value of an output pixel is also computed as a weighted sum of neighboring pixels. The difference is that the matrix of weights, in this case called the *correlation kernel*, is not rotated during the computation. The filter design functions in the Image Processing Toolbox return correlation kernels.

The following figure shows how to compute the (2,4) output pixel of the correlation of A, assuming h is a correlation kernel instead of a convolution kernel, using these steps:

- 1 Slide the center element of the correlation kernel so that lies on top of the (2,4) element of A.
- 2 Multiply each weight in the correlation kernel by the pixel of A underneath.
- 3 Sum the individual products from step 3.

The (2,4) output pixel from the correlation is

$$1 \cdot 8 + 8 \cdot 1 + 15 \cdot 6 + 7 \cdot 3 + 14 \cdot 5 + 16 \cdot 7 + 13 \cdot 4 + 20 \cdot 9 + 22 \cdot 2 = 585$$



### Computing the (2,4) Output of Correlation

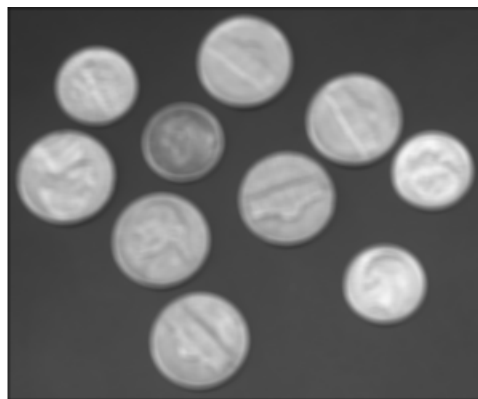
## Filtering Using `imfilter`

Filtering of images, either by correlation or convolution, can be performed using the toolbox function `imfilter`. This example filters an image with a 5-by-5 filter containing equal weights. Such a filter is often called an *averaging filter*.

```
I = imread('coins.png');  
h = ones(5,5) / 25;  
I2 = imfilter(I,h);  
imshow(I), title('Original Image');  
figure, imshow(I2), title('Filtered Image')
```



Original Image



Filtered Image

## Data Types

The `imfilter` function handles data types similarly to the way the image arithmetic functions do, as described in “Image Arithmetic Saturation Rules” on page 2-25. The output image has the same data type, or numeric class, as the input image. The `imfilter` function computes the value of each output pixel using double-precision, floating-point arithmetic. If the result exceeds the range of the data type, the `imfilter` function truncates the result to that data type’s allowed range. If it is an integer data type, `imfilter` rounds fractional values.

Because of the truncation behavior, you might sometimes want to consider converting your image to a different data type before calling `imfilter`. In

this example, the output of `imfilter` has negative values when the input is of class `double`.

```
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

h = [-1 0 1]

h =
    -1     0     1

imfilter(A,h)

ans =
    24   -16   -16    14    -8
     5   -16     9     9   -14
     6     9    14     9  -20
    12     9     9   -16  -21
    18    14   -16   -16    -2
```

Notice that the result has negative values. Now suppose `A` is of class `uint8`, instead of `double`.

```
A = uint8(magic(5));
imfilter(A,h)

ans =

    24     0     0    14     0
     5     0     9     9     0
     6     9    14     9     0
    12     9     9     0     0
    18    14     0     0     0
```

Since the input to `imfilter` is of class `uint8`, the output also is of class `uint8`, and so the negative values are truncated to 0. In such cases, it might be appropriate to convert the image to another type, such as a signed integer type, `single`, or `double`, before calling `imfilter`.

## Correlation and Convolution Options

The `imfilter` function can perform filtering using either correlation or convolution. It uses correlation by default, because the filter design functions, described in “Filter Design” on page 8-15, and the `fspecial` function, described in “Using Predefined Filter Types” on page 8-13, produce correlation kernels.

However, if you want to perform filtering using convolution instead, you can pass the string `'conv'` as an optional input argument to `imfilter`. For example:

```
A = magic(5);
h = [-1 0 1]
imfilter(A,h)    % filter using correlation

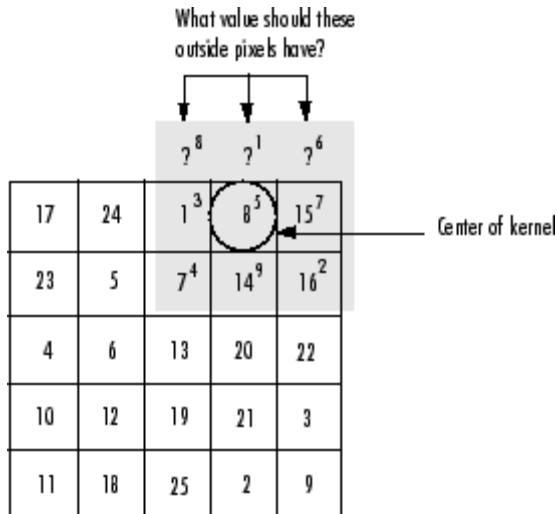
ans =
    24   -16   -16    14    -8
     5   -16     9     9   -14
     6     9    14     9   -20
    12     9     9   -16   -21
    18    14   -16   -16    -2

imfilter(A,h,'conv')    % filter using convolution

ans =
   -24    16    16   -14     8
    -5    16    -9    -9    14
    -6    -9   -14    -9    20
   -12    -9    -9    16    21
   -18   -14    16    16     2
```

### Boundary Padding Options

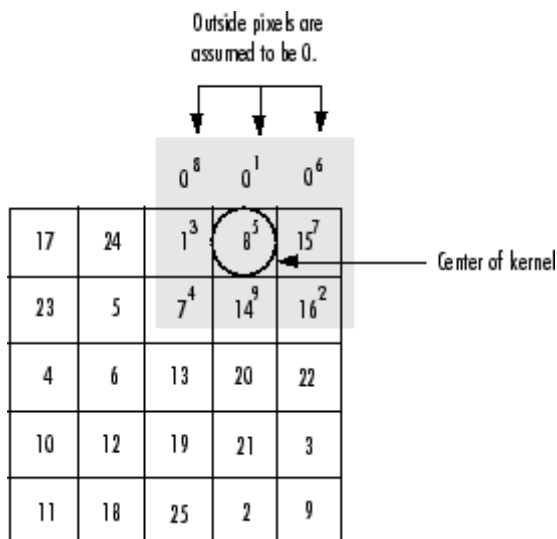
When computing an output pixel at the boundary of an image, a portion of the convolution or correlation kernel is usually off the edge of the image, as illustrated in the following figure.



### When the Values of the Kernel Fall Outside the Image

The `imfilter` function normally fills in these off-the-edge image pixels by assuming that they are 0. This is called zero padding and is illustrated in the following figure.

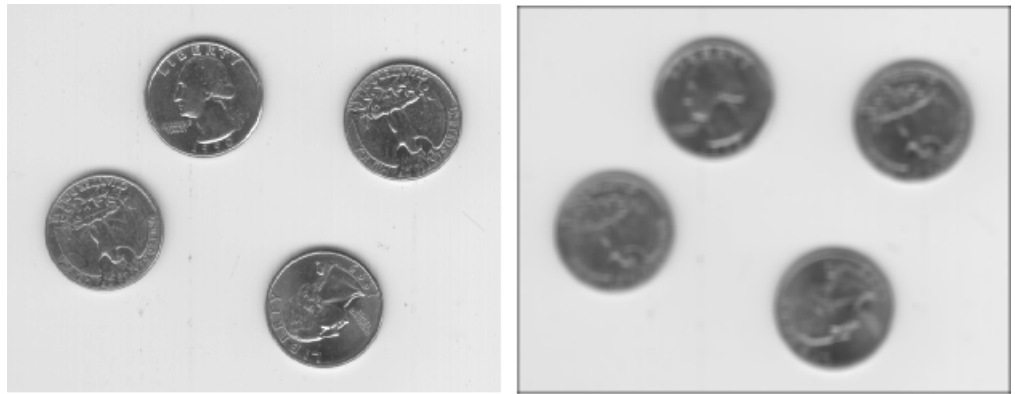




### Zero Padding of Outside Pixels

When you filter an image, zero padding can result in a dark band around the edge of the image, as shown in this example.

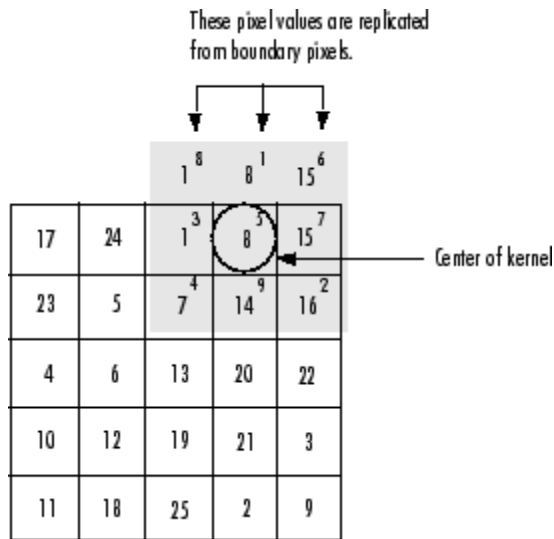
```
I = imread('eight.tif');
h = ones(5,5) / 25;
I2 = imfilter(I,h);
imshow(I), title('Original Image');
figure, imshow(I2), title('Filtered Image with Black Border')
```



Original Image

Filtered Image with Black Border

To eliminate the zero-padding artifacts around the edge of the image, `imfilter` offers an alternative boundary padding method called *border replication*. In border replication, the value of any pixel outside the image is determined by replicating the value from the nearest border pixel. This is illustrated in the following figure.



Replicated Boundary Pixels

To filter using border replication, pass the additional optional argument 'replicate' to `imfilter`.

```
I3 = imfilter(I,h,'replicate');  
figure, imshow(I3);  
title('Filtered Image with Border Replication')
```



**Filtered Image with Border Replication**

The `imfilter` function supports other boundary padding options, such as 'circular' and 'symmetric'. See the reference page for `imfilter` for details.

## Multidimensional Filtering

The `imfilter` function can handle both multidimensional images and multidimensional filters. A convenient property of filtering is that filtering a three-dimensional image with a two-dimensional filter is equivalent to filtering each plane of the three-dimensional image individually with the same two-dimensional filter. This example shows how easy it is to filter each color plane of a truecolor image with the same filter:

**1** Read in an RGB image and display it.

```
rgb = imread('peppers.png');  
imshow(rgb);
```



2 Filter the image and display it.

```
h = ones(5,5)/25;  
rgb2 = imfilter(rgb,h);  
figure, imshow(rgb2)
```



### Relationship to Other Filtering Functions

MATLAB has several two-dimensional and multidimensional filtering functions. The function `filter2` performs two-dimensional correlation, `conv2` performs two-dimensional convolution, and `convn` performs multidimensional convolution. Each of these filtering functions always converts the input to double, and the output is always double. These other filtering functions always assume the input is zero padded, and they do not support other padding options.

In contrast, the `imfilter` function does not convert input images to double. The `imfilter` function also offers a flexible set of boundary padding options, as described in “Boundary Padding Options” on page 8-8.

### Using Predefined Filter Types

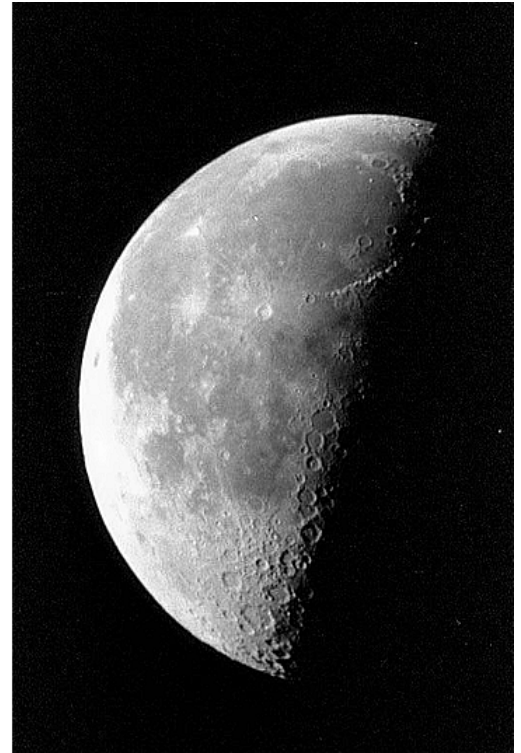
The `fspecial` function produces several kinds of predefined filters, in the form of correlation kernels. After creating a filter with `fspecial`, you can apply it directly to your image data using `imfilter`. This example illustrates applying an *unsharp masking* filter to a grayscale image. The unsharp masking filter has the effect of making edges and fine detail in the image more crisp.

```
I = imread('moon.tif');  
h = fspecial('unsharp');  
I2 = imfilter(I,h);  
imshow(I), title('Original Image')  
figure, imshow(I2), title('Filtered Image')
```



Image Courtesy of Michael Myers

**Original Image**



**Filtered Image**

## 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 might 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 a finite extent to a single point, or impulse. All filter design functions in the Image Processing Toolbox return FIR filters.

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**

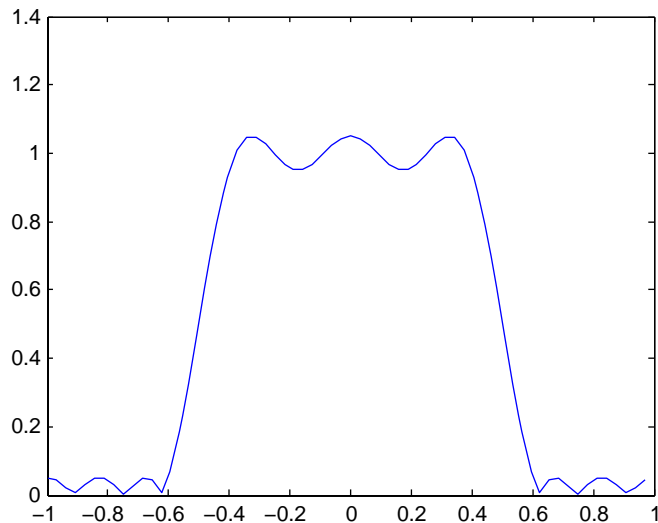
The frequency transformation method transforms a one-dimensional FIR filter into a two-dimensional FIR filter. The frequency transformation method preserves most of the characteristics of the one-dimensional filter, particularly the transition bandwidth and ripple characteristics. This method uses a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

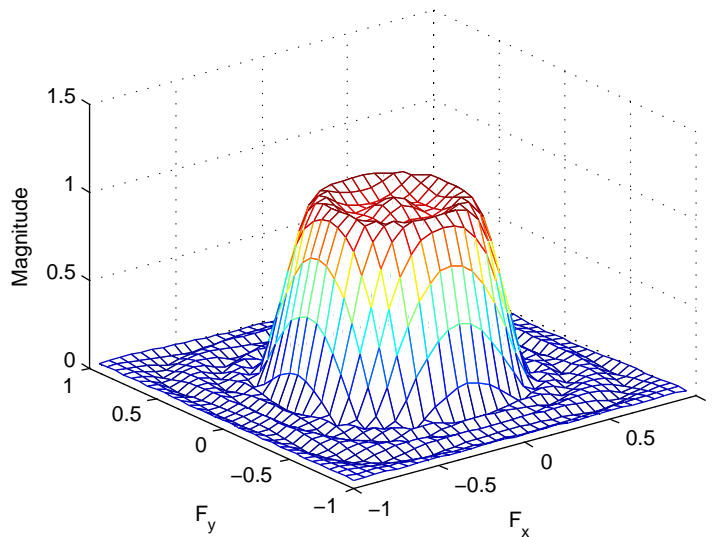


The frequency transformation method generally produces very good results, as it is easier to design a one-dimensional filter with particular characteristics than a corresponding two-dimensional filter. For instance, the next example designs an optimal equiripple one-dimensional FIR filter and uses it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



**One-Dimensional Frequency Response**



**Corresponding Two-Dimensional Frequency Response**

## Frequency Sampling Method

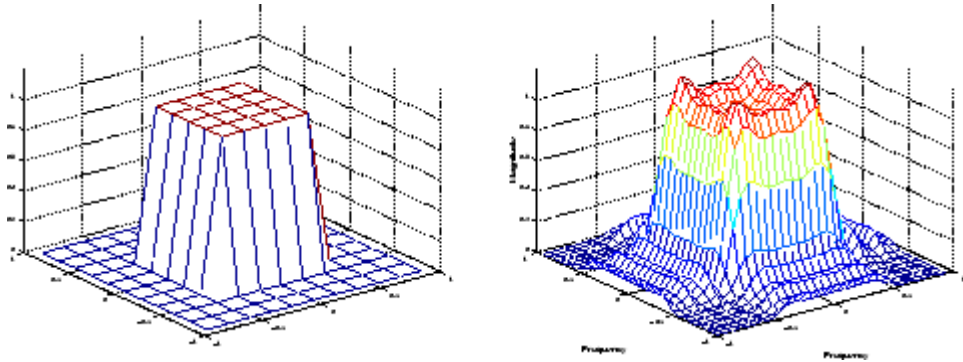
The frequency sampling method creates a filter based on a desired frequency response. Given a matrix of points that define the shape of the frequency response, this method creates a filter whose frequency response passes through those points. Frequency sampling places no constraints on the behavior of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

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 8-22 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])

```



**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, which tapers the ideal impulse response. 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.

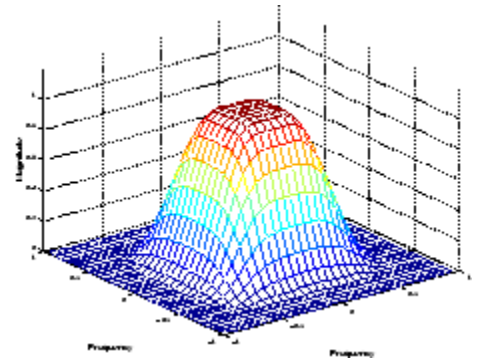
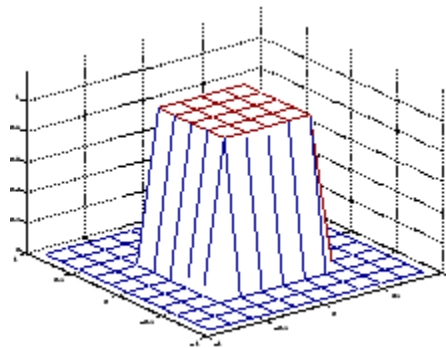
The toolbox provides two functions for window-based filter design, `fwind1` and `fwind2`. `fwind1` designs a two-dimensional filter by using a two-dimensional window that it creates from one or two one-dimensional windows that you specify. `fwind2` designs a two-dimensional filter by using a specified 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])
```



**Desired Two-Dimensional Frequency Response (left) and Actual Two-Dimensional Frequency Response (right)**

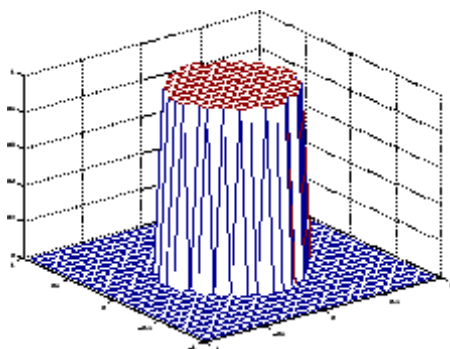
## Creating the Desired Frequency Response Matrix

The filter design functions `fsamp2`, `fwind2`, and `fwind2` all create filters based on a desired frequency response magnitude matrix. Frequency response is a mathematical function describing the gain of a filter in response to different input frequencies.

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 might 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)
```



### 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 ( $f_1 = 0$ ,  $f_2 = 0$ ).

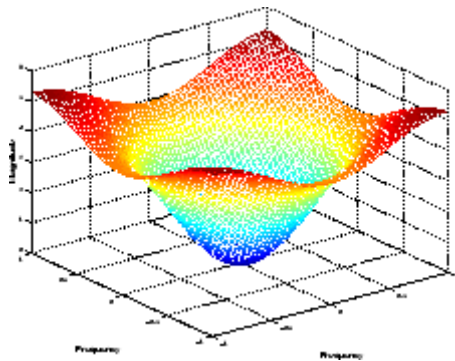
## Computing the Frequency Response of a Filter

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

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

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

```
freqz2(h)
```



### 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  $\pi$  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 9-2 for more information about the fast Fourier transform and its application to linear filtering and filter design.





# Transforms

---

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. This is called the *spatial domain*. 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 is called the *frequency domain*. Transforms are useful for a wide range of purposes, including convolution, enhancement, feature detection, and compression.

This chapter defines several important transforms and shows examples of their application to image processing.

Fourier Transform (p. 9-2)	Defines the Fourier transform and some of its applications in image processing
Discrete Cosine Transform (p. 9-15)	Describes the discrete cosine transform (DCT) of an image and its application, particularly in image compression
Radon Transform (p. 9-19)	Describes how the Image Processing Toolbox radon function computes projections of an image matrix along specified directions
Fan-Beam Projection Data (p. 9-35)	Describes how the Image Processing Toolbox radon function computes projections of an image matrix along specified directions

## Fourier Transform

The Fourier transform is a representation of an image as a sum of complex exponentials of varying magnitudes, frequencies, and phases. 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” on page 9-2
- “Discrete Fourier Transform” on page 9-7, including a discussion of fast Fourier transform
- “Applications of the Fourier Transform” on page 9-10 (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 the *two-dimensional Fourier transform* of  $f(m, n)$  is defined 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  $\omega_1$  and  $\omega_2$  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  $\omega_1$  and  $\omega_2$ , with period  $2\pi$ . Because of the periodicity, usually only the range  $-\pi \leq \omega_1, \omega_2 \leq \pi$  is displayed. Note that  $F(0, 0)$  is the sum of all the values of  $f(m, n)$ . For this reason,  $F(0, 0)$  is often called the *constant component* or *DC component* of the Fourier transform. (DC stands for direct current; it is an electrical engineering term that refers to a constant-voltage power source, as opposed to a power source whose voltage varies sinusoidally.)

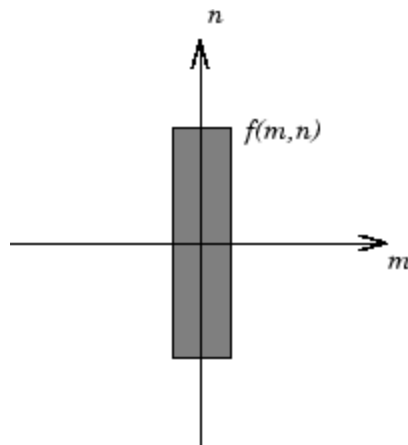
The inverse of a transform is an operation that when performed on a transformed image produces the original image. The inverse two-dimensional Fourier transform is given by

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

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  $(\omega_1, \omega_2)$  are given by  $F(\omega_1, \omega_2)$ .

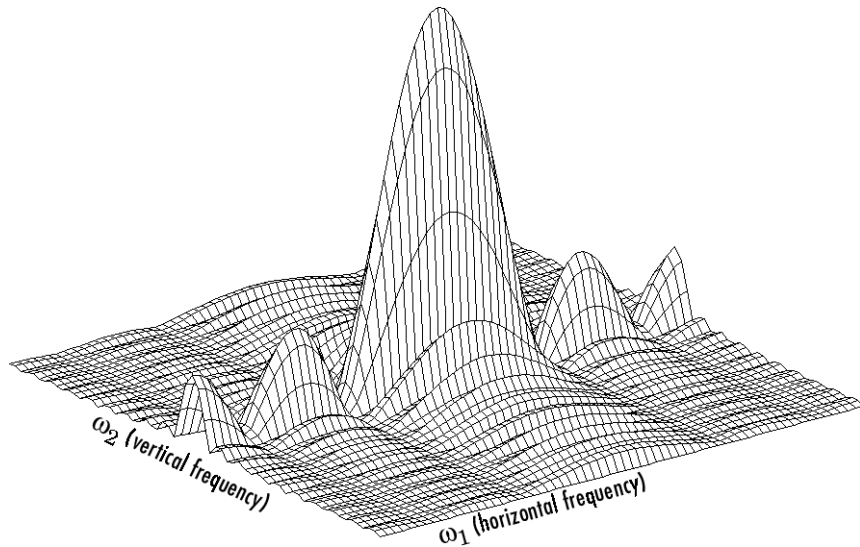
### Visualizing the Fourier Transform

To illustrate, consider a function  $f(m, n)$  that equals 1 within a rectangular region and 0 everywhere else. To simplify the diagram,  $f(m, n)$  is shown as a continuous function, even though the variables  $m$  and  $n$  are discrete.



### Rectangular Function

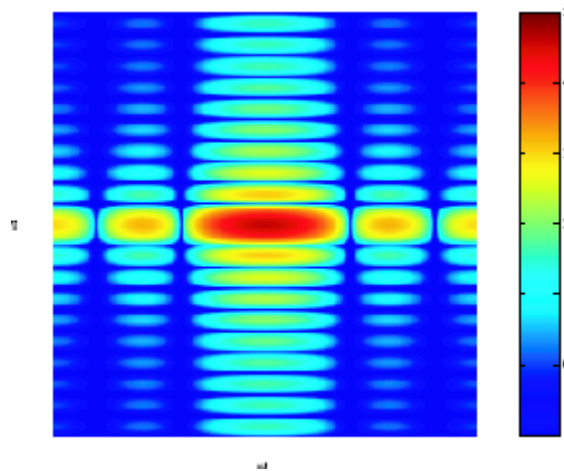
The following figure shows, as a mesh plot, the magnitude of the Fourier transform,  $|F(\omega_1, \omega_2)|$ , of the rectangular function shown in the preceding figure. The mesh plot of the magnitude is a common way to visualize the Fourier transform.



### 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.

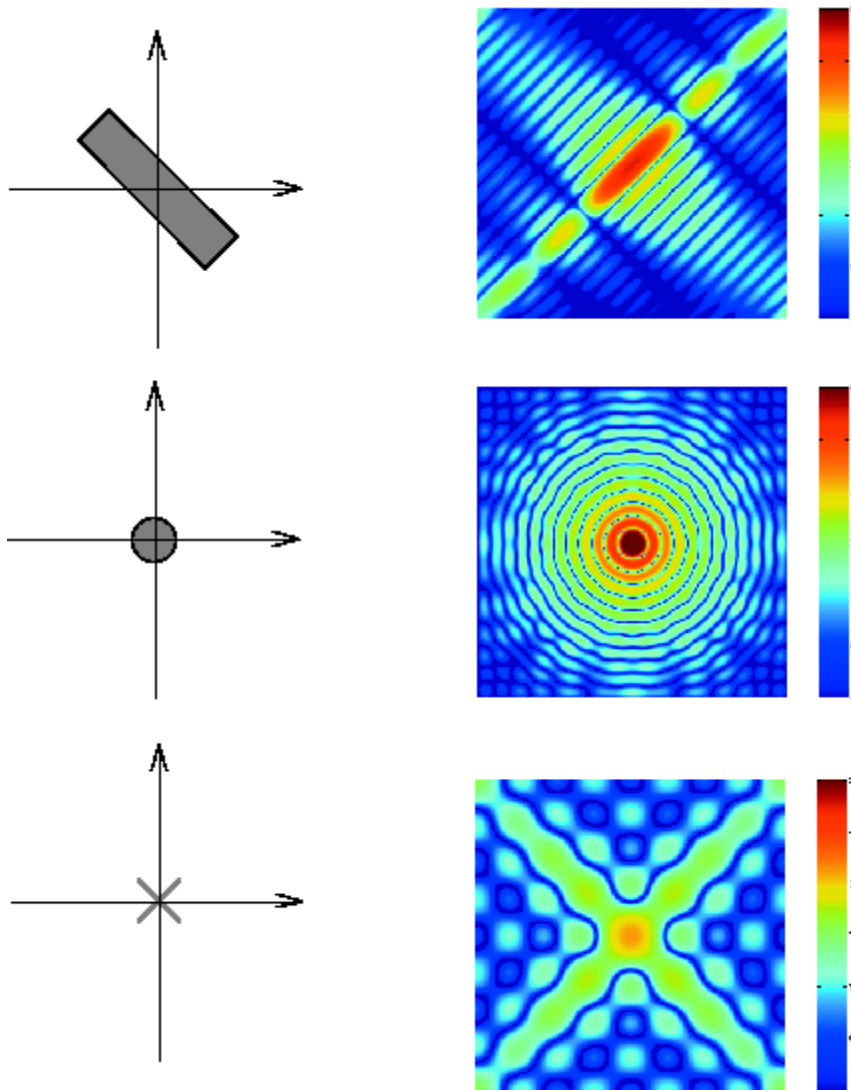
Another common way to visualize the Fourier transform is to display  $\log|F(\omega_1, \omega_2)|$  as an image, as shown.



### 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.



**Fourier Transforms of Some Simple Shapes**

## 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). A discrete transform is a transform whose input and output values are discrete samples, making it convenient for computer manipulation. There are two principal reasons for using this form of the transform:

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

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

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

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

The values  $F(p, q)$  are the DFT coefficients of  $f(m, n)$ . The zero-frequency coefficient,  $F(0, 0)$ , is often called the "DC component." DC is an electrical engineering term that stands for direct current. (Note that matrix indices in MATLAB always start at 1 rather than 0; therefore, the matrix elements  $f(1, 1)$  and  $F(1, 1)$  correspond to the mathematical quantities  $f(0, 0)$  and  $F(0, 0)$ , respectively.)

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

## Relationship to the Fourier Transform

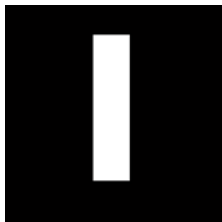
The DFT coefficients  $F(p, q)$  are samples of the Fourier transform  $F(\omega_1, \omega_2)$ .

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

## Example

- 1 Construct a matrix  $f$  that is similar to the function  $f(m, n)$  in the example in “Definition of Fourier Transform” on page 9-2. Remember that  $f(m, n)$  is equal to 1 within the rectangular region and 0 elsewhere. Use a binary image to represent  $f(m, n)$ .

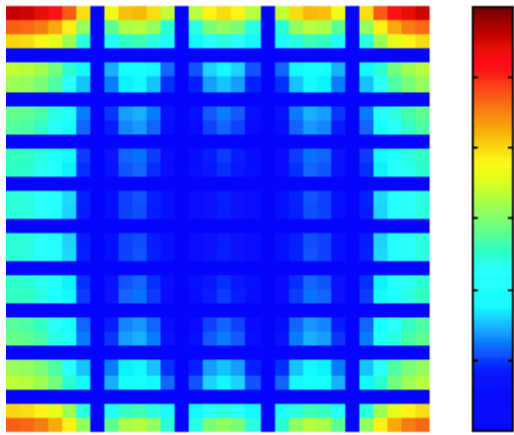
```
f = zeros(30,30);
f(5:24,13:17) = 1;
imshow(f, 'notruesize')
```



- 2 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
```





### Discrete Fourier Transform Computed Without Padding

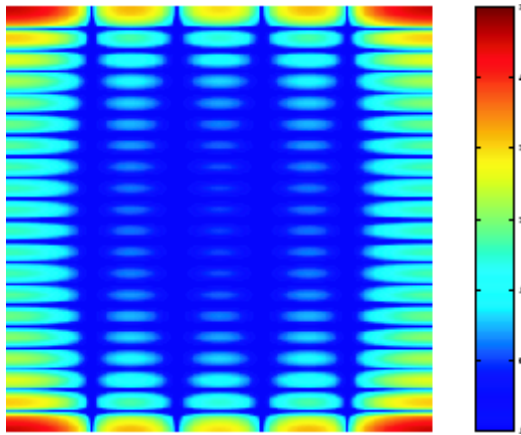
This plot differs from the Fourier transform displayed in “Visualizing the Fourier Transform” on page 9-3. First, the sampling of the Fourier transform is much coarser. Second, the zero-frequency coefficient is displayed in the upper left corner instead of the traditional location in the center.

- 3 To obtain a finer sampling of the Fourier transform, add zero padding to  $f$  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
```



**Discrete Fourier Transform Computed with Padding**

- 4 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 shown in “Visualizing the Fourier Transform” on page 9-3.

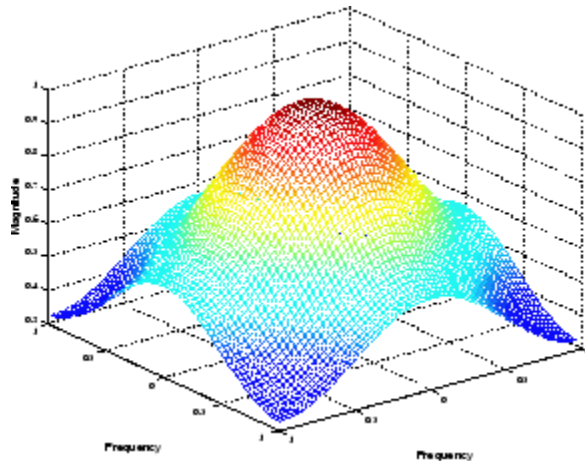
## Applications of the Fourier Transform

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)
```



### Frequency Response of a Gaussian Filter

See Chapter 8, “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.

---

**Note** The FFT-based convolution method is most often used for large inputs. For small inputs it is generally faster to use `imfilter`.

---

To illustrate, this example performs the convolution of A and B, where A is an M-by-N matrix and B is a P-by-Q matrix:

1 Create two matrices.

```
A = magic(3);  
B = ones(3);
```

- 2** 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.) The example pads the matrices to be 8-by-8.

```
A(8,8) = 0;
B(8,8) = 0;
```

- 3** Compute the two-dimensional DFT of A and B using `fft2`.
- 4** Multiply the two DFTs together.
- 5** Compute the inverse two-dimensional DFT of the result using `ifft2`.

The following code performs steps 3, 4, and 5 in the procedure.

```
C = ifft2(fft2(A).*fft2(B));
```

- 6** Extract the nonzero portion of the result and remove the imaginary part caused by roundoff error.

```
C = C(1:5,1:5);
C = real(C)
```

```
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
```

## 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*.

This example illustrates how to use correlation to locate occurrences of the letter "a" in an image containing text:

- 1** Read in the sample image.

```
bw = imread('text.png');
```

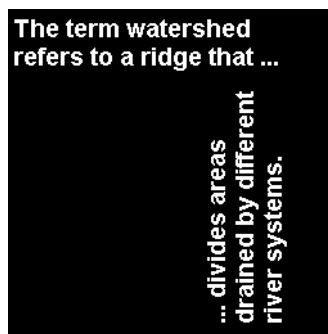
- 2** Create a template for matching by extracting the letter "a" from the image.

```
a = bw(32:45,88:98);
```

You can also create the template image by using the interactive version of `imcrop`, using the `pixval` function to determine the coordinates of features in an image.

The following figure shows both the original image and the template.

```
imshow(bw);
figure, imshow(a);
```



**Image (left) and the Template to Correlate (right)**

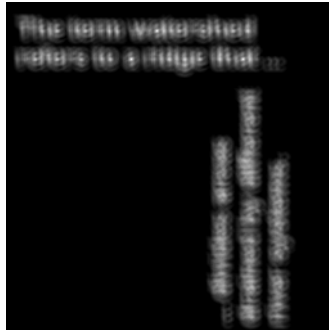
- 3** Compute the correlation of the template image with the original image by rotating the template image by  $180^\circ$  and then using the FFT-based convolution technique described in “Fast Convolution” on page 9-11.

(Convolution is equivalent to correlation if you rotate the convolution kernel by  $180^\circ$ .) To match the template to the image, use the `fft2` and `ifft2` functions.

```
C = real(ifft2(fft2(bw) .* fft2(rot90(a,2),256,256)));
```

The following image shows the result of the correlation. Bright peaks in the image correspond to occurrences of the letter.

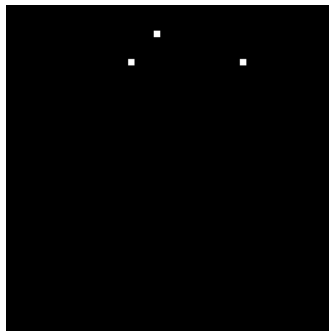
```
figure, imshow(C,[]) % Scale image to appropriate display range.
```



### Correlated Image

- 4 To view the locations of the template in the image, find the maximum pixel value and then define a threshold value that is less than this maximum. The locations of these peaks are indicated by the white spots in the thresholded correlation image. (To make the locations easier to see in this figure, the thresholded image has been dilated to enlarge the size of the points.)

```
max(C(:))
ans =
    68.0000
thresh = 60; % Use a threshold that's a little less than max.
figure, imshow(C > thresh)% Display showing pixels over
threshold.
```



### Correlated, Thresholded Image Showing Template Locations

## Discrete Cosine Transform

The discrete cosine transform (DCT) represents an image as a sum of sinusoids of varying magnitudes and frequencies. 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.

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

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

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

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

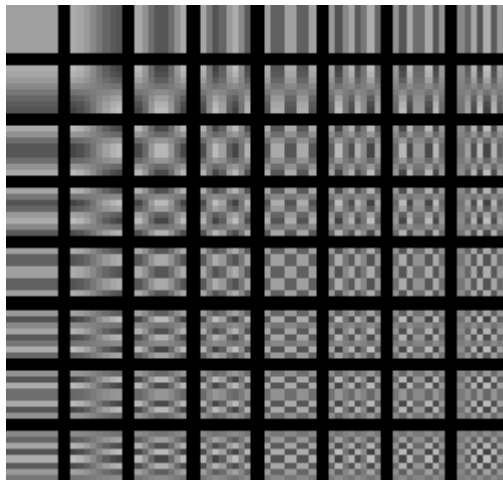
$$A_{mn} = \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} \alpha_p \alpha_q B_{pq} \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N}, \quad \begin{array}{l} 0 \leq m \leq M-1 \\ 0 \leq n \leq N-1 \end{array}$$

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

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

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

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.



**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



might 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

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

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

## 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)

```

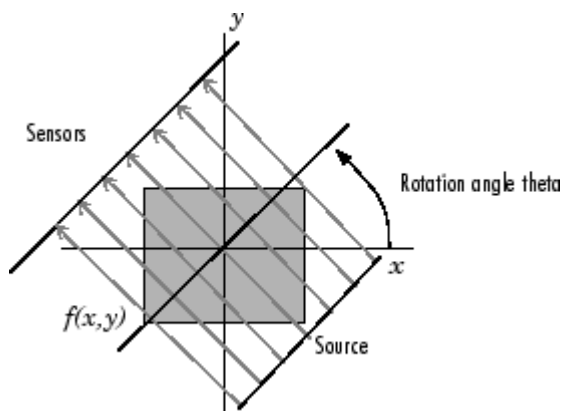


Image Courtesy of MIT

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 set of line integrals. The `radon` function computes the line integrals from multiple sources along parallel paths, or *beams*, in a certain direction. The beams are spaced 1 pixel unit apart. To represent an image, the `radon` function takes multiple, parallel-beam projections of the image from different angles by rotating the source around the center of the image. The following figure shows a single projection at a specified rotation angle.



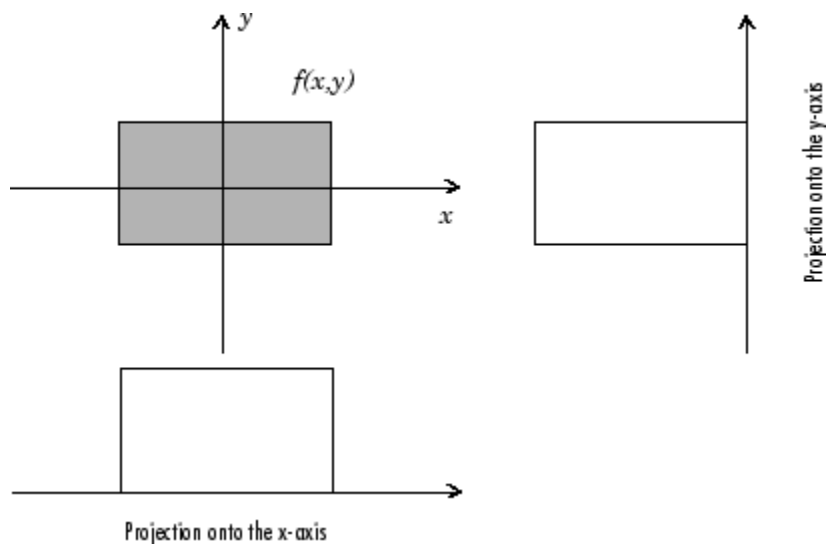
### Parallel-Beam Projection at Rotation Angle Theta

---

**Note** For information about creating projection data from line integrals along paths that radiate from a single source, called fan-beam projections, see “Fan-Beam Projection Data” on page 9-35. To convert parallel-beam projection data to fan-beam projection data, use the `para2fan` function.

---

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. The following figure shows horizontal and vertical projections for a simple two-dimensional function.



### Horizontal and Vertical Projections of a Simple Function

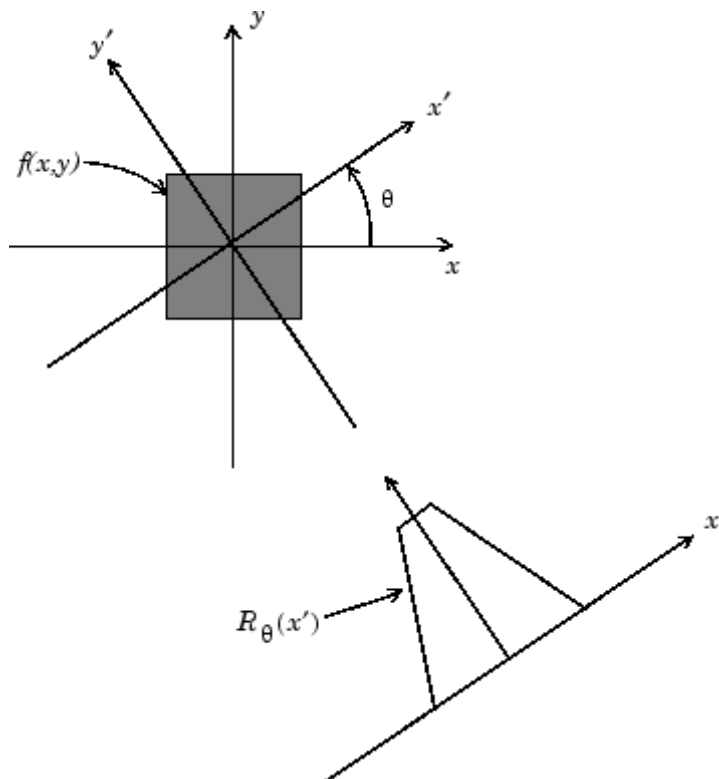
Projections can be computed along any angle  $[\theta]$ . In general, the Radon transform of  $f(x,y)$  is the line integral of  $f$  parallel to the  $y'$ -axis

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

where

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

The following figure illustrates the geometry of the Radon transform.



### Geometry of the Radon Transform

## Plotting the Radon Transform

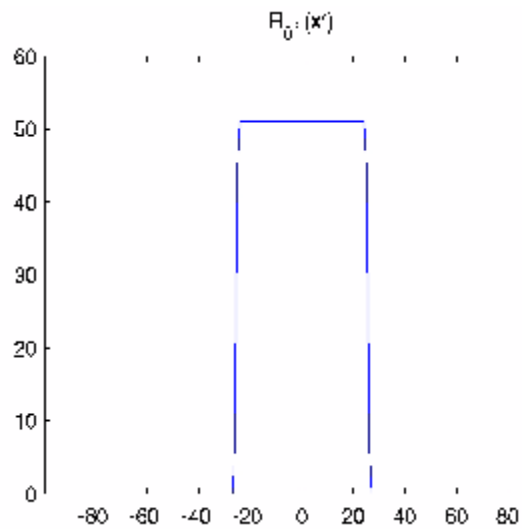
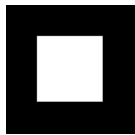
You can compute the Radon transform of an image  $I$  for the angles specified in the vector  $\theta$  using the `radon` function with this syntax.

```
[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  $x' = 0$ .

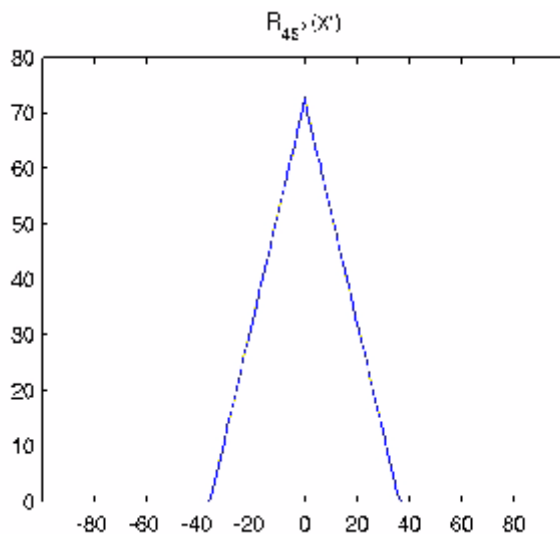
The commands below compute and plot the Radon transform at  $0^\circ$  and  $45^\circ$  of an image containing a single square object.  $x_p$  is the same for all projection angles.

```
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)')
```



### Radon Transform of a Square Function at 0 Degrees

```
figure; plot(xp,R(:,2)); title('R_{45^o} (x\prime)')
```

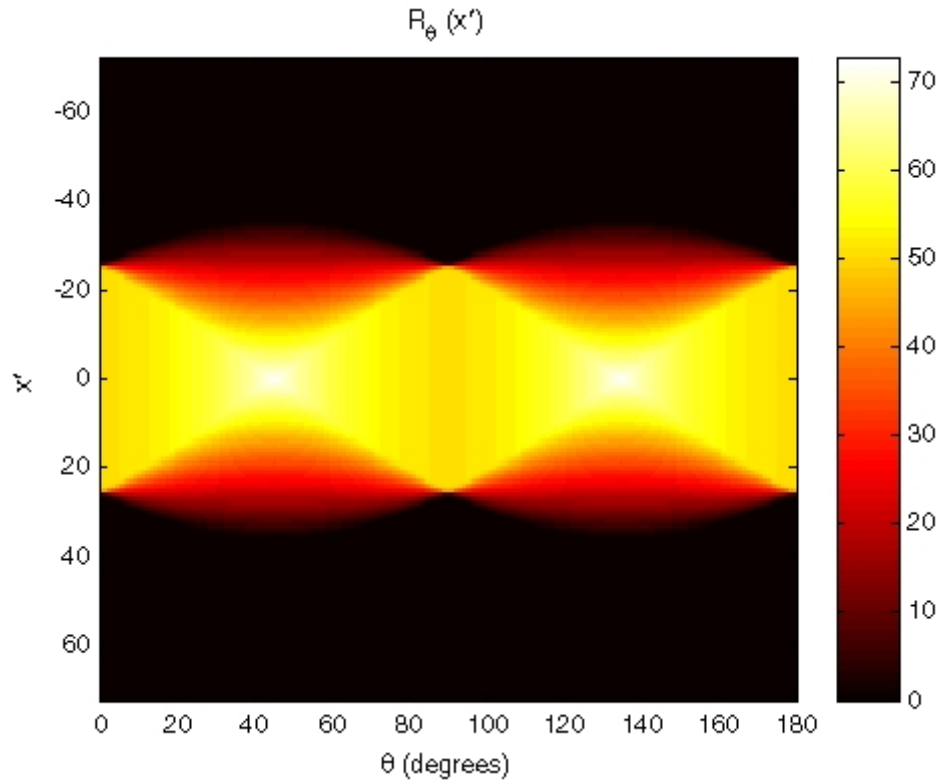


### Radon Transform of a Square Function at 45 Degrees

## Viewing the Radon Transform as an Image

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  $0^\circ$  to  $180^\circ$ , in  $1^\circ$  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
```



### 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 = fitsread('solarspectra.fts');
I = mat2gray(I);
BW = edge(I);
imshow(I), figure, imshow(BW)
```



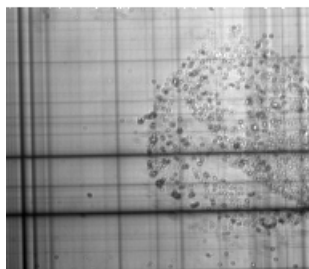
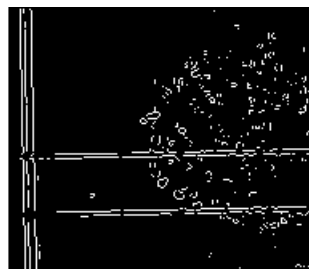


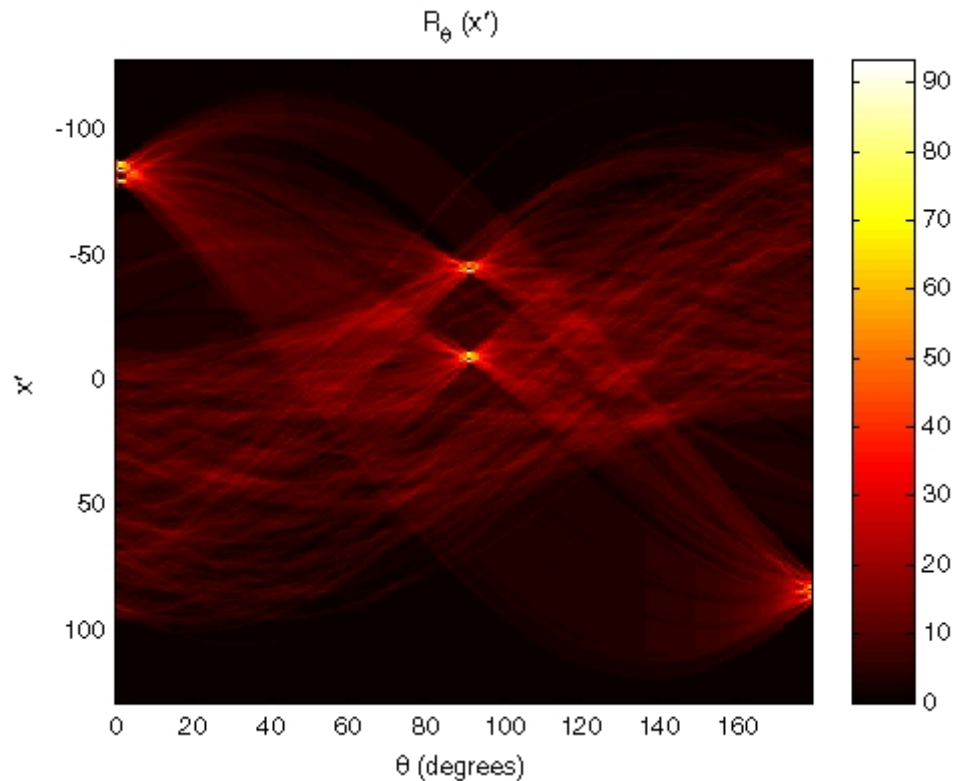
Image Courtesy of Ann Walker  
Original Image



Edge Image

**2** Compute the Radon transform of the edge image.

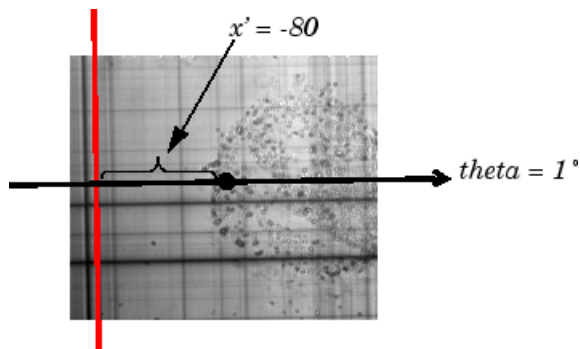
```
theta = 0:179;  
[R, xp] = radon(BW, theta);  
figure, imagesc(theta, xp, R); colormap(hot);  
xlabel('\theta (degrees)'); ylabel('x\prime');  
title('R_{\theta} (x\prime)');  
colorbar
```



### Radon Transform of an Edge Image

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

In the following figure, the strongest peaks in  $R$  correspond to  $\theta = 1^\circ$  and  $x' = -80$ . The line perpendicular to that angle and located at  $x' = -80$  is shown below, superimposed in red on the original image. The Radon transform geometry is shown in black. Notice that the other strong lines parallel to the red line also appear as peaks at  $\theta = 1^\circ$  in the transform. Also, the lines perpendicular to this line appear as peaks at  $\theta = 91^\circ$ .



### Radon Transform Geometry and the Strongest Peak (Red)

## 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 described in “Radon Transform” on page 9-19, given an image *I* and a set of angles *theta*, the radon function can be used to calculate the Radon transform.

$$R = \text{radon}(I, \text{theta});$$

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

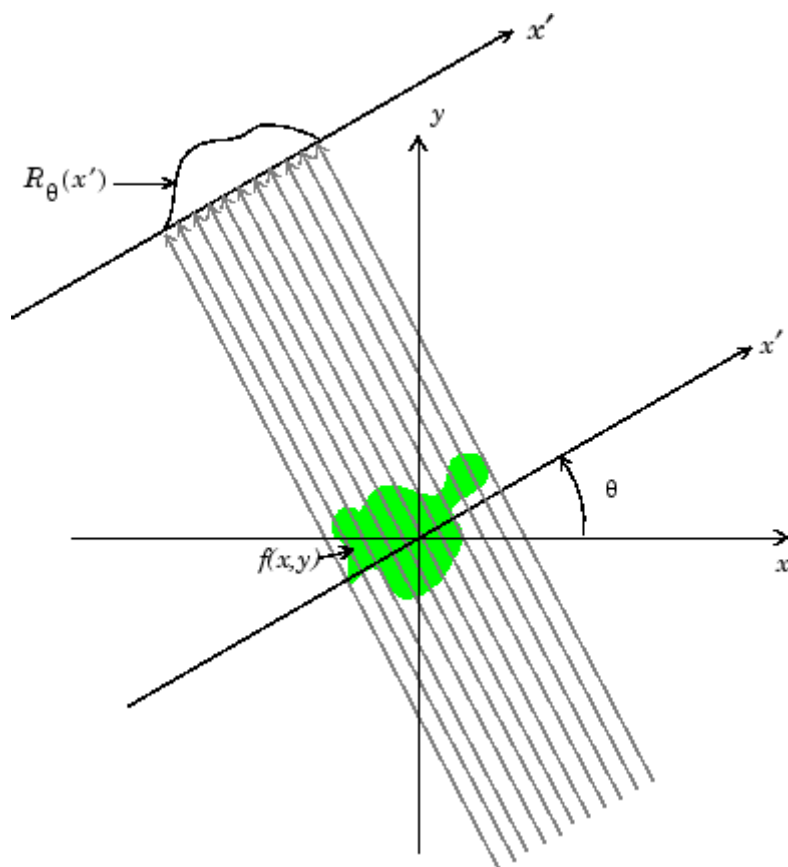
$$IR = \text{iradon}(R, \text{theta});$$

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.

The following figure illustrates how parallel-beam geometry is applied in X-ray absorption tomography. Note that there is an equal number of  $n$  emitters and  $n$  sensors. Each sensor 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 in “Radon Transform” on page 9-19.  $f(x,y)$  denotes the brightness of the image and  $R_{\theta}(x')$  is the projection at angle theta.



### Parallel-Beam Projections Through an Object

Another geometry that is commonly used is *fan-beam* geometry, in which there is one source and  $n$  sensors. For more information, see “Fan-Beam Projection Data” on page 9-35. To convert parallel-beam projection data into fan-beam projection data, use the `para2fan` function.

### Improving the Results

`iradon` uses the *filtered backprojection* algorithm to compute the inverse Radon transform. This algorithm forms an approximation of 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  $\Delta[\text{[THETA]}]$ . When the scalar  $\Delta[\text{[THETA]}]$  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);
```

## Example: Reconstructing an Image from Parallel Projection Data

The commands below illustrate how to reconstruct an image from parallel projection data. 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.

- 1 Create a Shepp-Logan head phantom image.

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

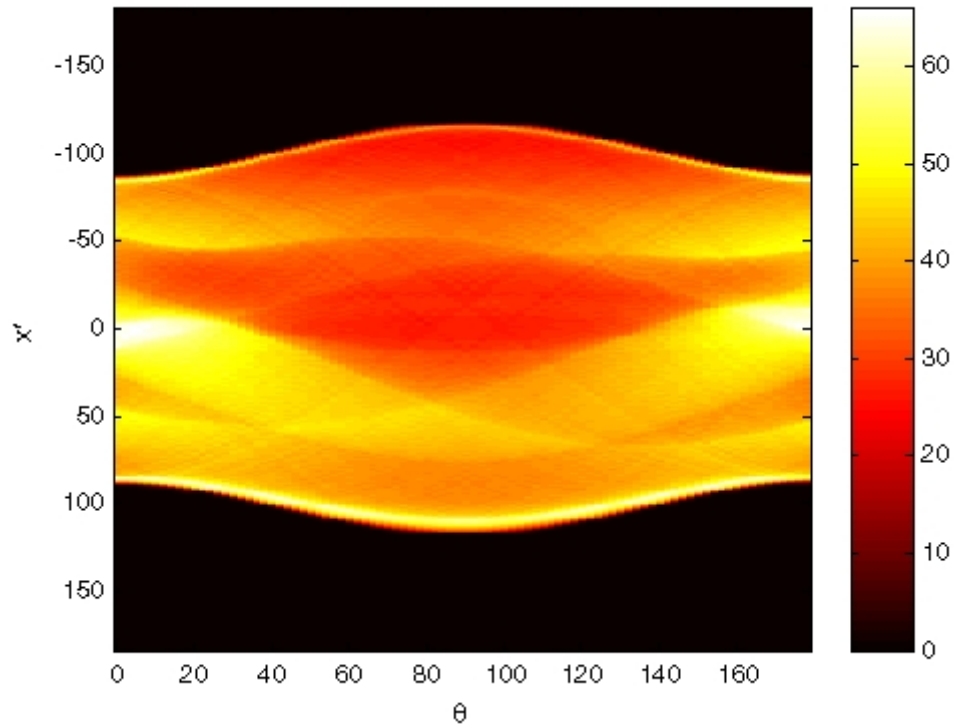


- 2 Compute the Radon transform of the phantom brain for three different sets of theta values. R1 has 18 projections, R2 has 36 projections, and R3 has 90 projections.

```
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);
```

- 3 Display a plot of one of the Radon transforms of the Shepp-Logan head phantom. The following figure shows R3, the transform with 90 projections.

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



### Radon Transform of Head Phantom Using 90 Projections

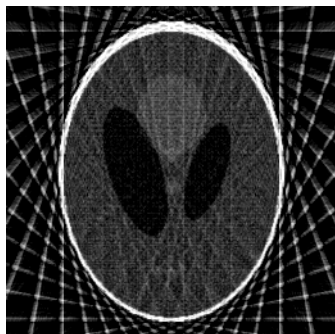
Note how some of the features of the input image appear in this image of the transform. The first column in the Radon transform corresponds to a projection at  $0^\circ$  that is integrating in the vertical direction. The centermost column corresponds to a projection at  $90^\circ$ , which is integrating in the horizontal direction. The projection at  $90^\circ$  has a wider profile than the projection at  $0^\circ$  due to the larger vertical semi-axis of the outermost ellipse of the phantom.



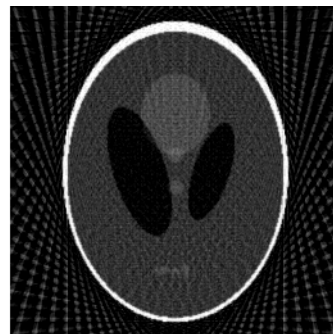
- 4** Reconstruct the head phantom image from the projection data created in step 2 and display the results.

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

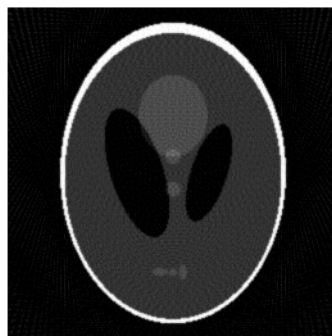
The following figure shows the results of all three reconstructions. Notice how image I1, which was reconstructed from only 18 projections, is the least accurate reconstruction. Image I2, which was reconstructed from 36 projections, is better, but it is still not clear enough to discern clearly the small ellipses in the lower portion of the image. I3, reconstructed using 90 projections, most closely resembles the original image. Notice that when the number of projections is relatively small (as in I1 and I2), the reconstruction can include some artifacts from the back projection.



I1



I2

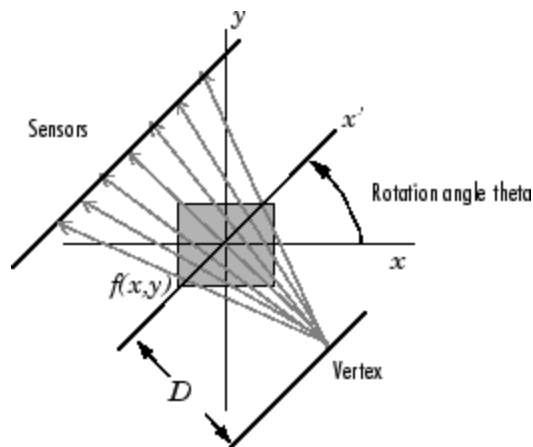


I3

**Inverse Radon Transforms of the Shepp-Logan Head Phantom**

## Fan-Beam Projection Data

The fanbeam 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 set of line integrals. The fanbeam function computes the line integrals along paths that radiate from a single source, forming a fan shape. To represent an image, the fanbeam function takes multiple projections of the image from different angles by rotating the source around the center of the image. The following figure shows a single fan-beam projection at a specified rotation angle.



### Fan-Beam Projection at Rotation Angle Theta

This section

- Describes how to use the fanbeam function to generate fan-beam projection data
- Describes how to reconstruct an image from fan-beam projection data
- Shows an example that creates a fan-beam projection of an image and then reconstructs the image from the fan-beam projection data

---

**Note** For information about creating projection data from line integrals along parallel paths, see “Radon Transform” on page 9-19. To convert fan-beam projection data to parallel-beam projection data, use the `fan2para` function.

---

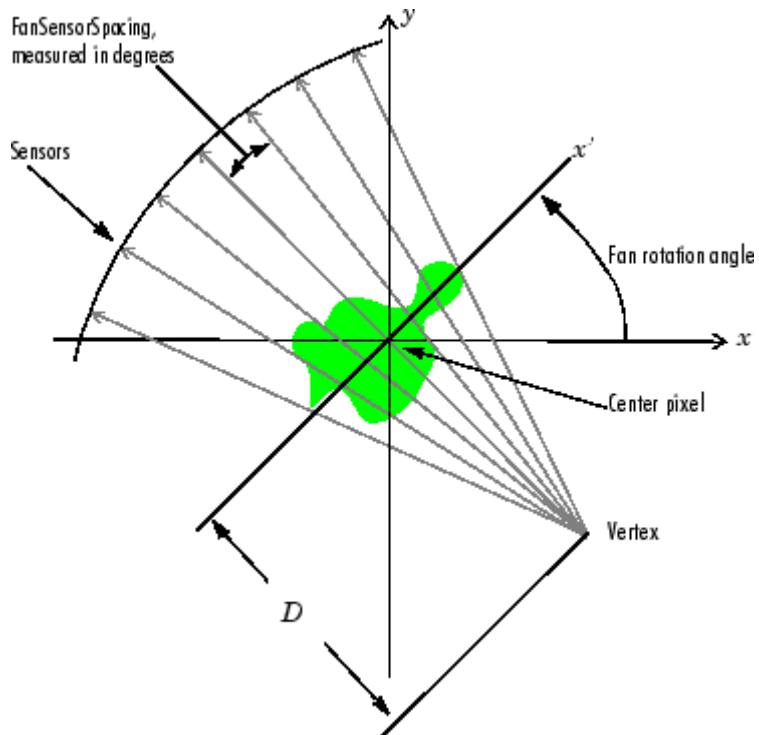
## Computing Fan-Beam Projection Data

To compute fan-beam projection data, use the `fanbeam` function. You specify as arguments an image and the distance between the vertex of the fan-beam projections and the center of rotation (the center pixel in the image). The `fanbeam` function determines the number of beams, based on the size of the image and the settings of `fanbeam` parameters.

The `FanSensorGeometry` parameter specifies how sensors are aligned. If you specify the value `'arc'` for `FanSensorGeometry` (the default), `fanbeam` positions the sensors along an arc, spacing the sensors at 1 degree intervals. Using the `FanSensorSpacing` parameter, you can control the distance between sensors by specifying the angle between each beam. If you specify the value `'line'` for `FanSensorGeometry` parameter, `fanbeam` position sensors along a straight line, rather than an arc. With `'line'` geometry, the `FanSensorSpacing` parameter specifies the distance between the sensors, in pixels, along the  $x'$  axis.

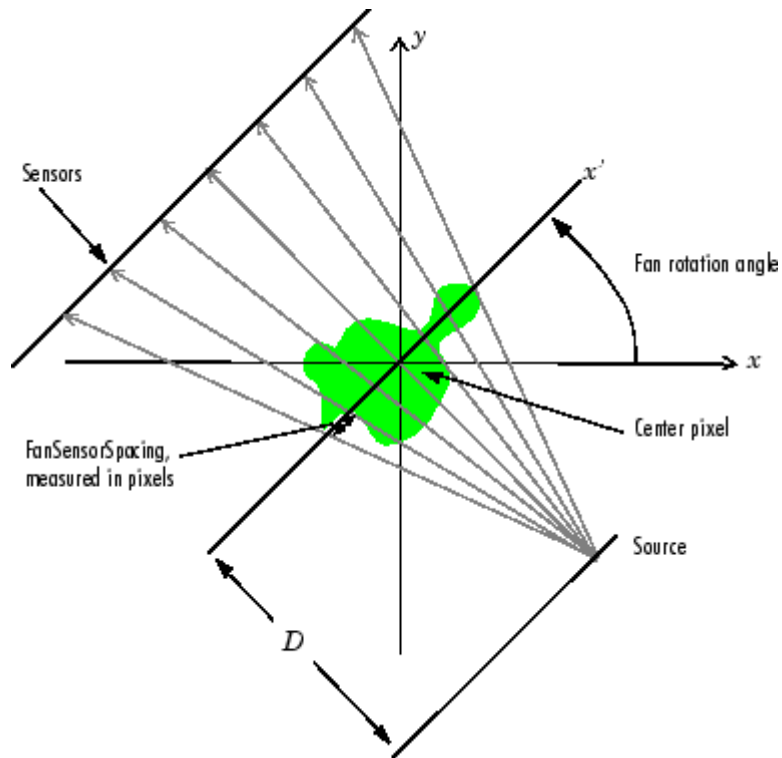
`fanbeam` takes projections at different angles by rotating the source around the center pixel at 1 degree intervals. Using the `FanRotationIncrement` parameter you can specify a different rotation angle increment.

The following figures illustrate both these geometries. The first figure illustrates geometry used by the `fanbeam` function when `FanSensorGeometry` is set to `'arc'` (the default). Note how you specify the distance between sensors by specifying the angular spacing of the beams.



### Fan-Beam Projection with Arc Geometry

The following figure illustrates the geometry used by the fanbeam function when `FanSensorGeometry` is set to 'line'. In this figure, note how you specify the position of the sensors by specifying the distance between them in pixels along the  $x'$  axis.



### Fan-Beam Projection with Line Geometry

## Reconstructing an Image from Fan-Beam Projection Data

To reconstruct an image from fan-beam projection data, use the `ifanbeam` function. With this function, you specify as arguments the projection data and the distance between the vertex of the fan-beam projections and the center of rotation when the projection data was created. For example, this code recreates the image `I` from the projection data `P` and distance `D`.

```
I = ifanbeam(P,D);
```

By default, the `ifanbeam` function assumes that the fan-beam projection data was created using the arc fan sensor geometry, with beams spaced at 1 degree angles and projections taken at 1 degree increments over a full 360 degree

range. As with the `fanbeam` function, you can use `ifanbeam` parameters to specify other values for these characteristics of the projection data. Use the same values for these parameters that were used when the projection data was created. For more information about these parameters, see “Computing Fan-Beam Projection Data” on page 9-36.

The `ifanbeam` function converts the fan-beam projection data to parallel-beam projection data with the `fan2para` function, and then calls the `iradon` function to perform the image reconstruction. For this reason, the `ifanbeam` function supports certain `iradon` parameters, which it passes to the `iradon` function. See “Inverse Radon Transform” on page 9-27 for more information about the `iradon` function.

## Working with Fan-Beam Projection Data

The commands below illustrate how to use `fanbeam` and `ifanbeam` 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.

- 1 Generate the test image and display it.

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



- 2 Compute fan-beam projection data of the test image, using the `FanSensorSpacing` parameter to vary the sensor spacing. The example

uses the fanbeam arc geometry, so you specify the spacing between sensors by specifying the angular spacing of the beams. The first call spaces the beams at 2 degrees; the second at 1 degree; and the third at 0.25 degrees. In each call, the distance between the center of rotation and vertex of the projections is constant at 250 pixels. In addition, fanbeam rotates the projection around the center pixel at 1 degree increments.

```
D = 250;

dsensor1 = 2;
F1 = fanbeam(P,D,'FanSensorSpacing',dsensor1);

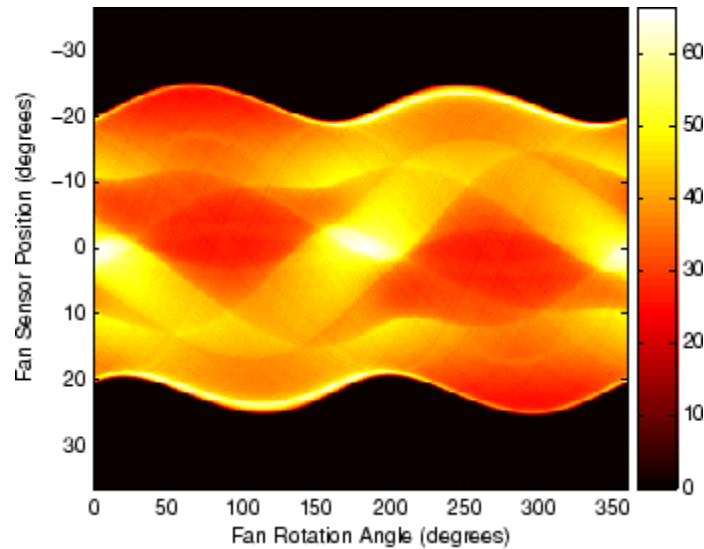
dsensor2 = 1;
F2 = fanbeam(P,D,'FanSensorSpacing',dsensor2);

dsensor3 = 0.25
[F3, sensor_pos3, fan_rot_angles3] = fanbeam(P,D,...
'FanSensorSpacing',dsensor3);
```

- 3** Plot the projection data F3. Because fanbeam calculates projection data at rotation angles from 0 to 360 degrees, the same patterns occur at an offset of 180 degrees. The same features are being sampled from both sides. Compare this plot to the plot of the parallel-beam projection data of the head phantom using 90 projections in “Example: Reconstructing an Image from Parallel Projection Data” on page 9-30.

```
figure, imagesc(fan_rot_angles3, sensor_pos3, F3)
colormap(hot); colorbar
xlabel('Fan Rotation Angle (degrees)')
ylabel('Fan Sensor Position (degrees)')
```





- 4** Reconstruct the image from the fan-beam projection data using `ifanbeam`. In each reconstruction, match the fan sensor spacing with the spacing used when the projection data was created in step 2. The example uses the `OutputSize` parameter to constrain the output size of each reconstruction to be the same as the size of the original image  $|P|$ .

```
output_size = max(size(P));

Ifan1 = ifanbeam(F1,D,
    'FanSensorSpacing',dsensor1,'OutputSize',output_size);
figure, imshow(Ifan1)

Ifan2 = ifanbeam(F2,D,
    'FanSensorSpacing',dsensor2,'OutputSize',output_size);
figure, imshow(Ifan2)

Ifan3 = ifanbeam(F3,D,
    'FanSensorSpacing',dsensor3,'OutputSize',output_size);
figure, imshow(Ifan3)
```

The following figure shows the result of each transform. Note how the quality of the reconstruction gets better as the number of beams in the

projection increases. The first image,  $I_{fan1}$ , was created using 2 degree spacing of the beams; the second image,  $I_{fan2}$ , was created using 1 degree spacing of the beams; the third image,  $I_{fan3}$ , was created using 0.25 spacing of the beams.

 $I_{fan1}$  $I_{fan2}$  $I_{fan3}$ 

**Reconstructions of the Head Phantom Image from Fan-Beam Projections**

# Morphological Operations

---

*Morphology* is a broad set of image processing 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. In a morphological operation, the value of each pixel in the output image is based on a comparison of the corresponding pixel in the input image with its neighbors. By choosing the size and shape of the neighborhood, you can construct a morphological operation that is sensitive to specific shapes in the input image.

This chapter describes the Image Processing Toolbox morphological functions. You can use these functions to perform common image processing tasks, such as contrast enhancement, noise removal, thinning, skeletonization, filling, and segmentation.

Dilation and Erosion (p. 10-3)

Defines the two fundamental morphological operations, dilation and erosion, and some of the morphological image processing operations that are based on combinations of these operations

Morphological Reconstruction  
(p. 10-18)

Describes morphological reconstruction and the toolbox functions that use this type of processing

Distance Transform (p. 10-37)

Describes how to use the `bwdist` function to compute the distance transform of an image

Objects, Regions, and Feature  
Measurement (p. 10-40)

Describes functions that return  
information about a binary image

Lookup Table Operations (p. 10-44)

Describes functions that perform  
lookup table operations

## Dilation and Erosion

Dilation and erosion are two fundamental morphological operations. Dilation adds pixels to the boundaries of objects in an image, while erosion removes pixels on object boundaries. The number of pixels added or removed from the objects in an image depends on the size and shape of the *structuring element* used to process the image.

The following sections

- Provide important background information about how the dilation and erosion functions operate
- Describe structuring elements and how to create them
- Describe how to perform a morphological dilation
- Describe how to perform a morphological erosion
- Describe some of the common operations that are based on dilation and erosion
- Describe toolbox functions that are based on dilation and erosion

To view an extended example that uses morphological processing to solve an image processing problem, see the Image Processing Toolbox watershed segmentation demo.

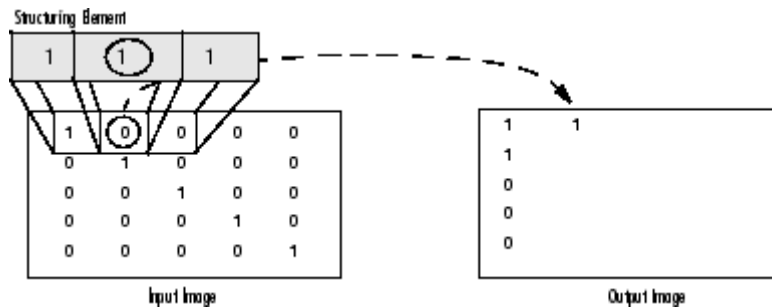
### Understanding Dilation and Erosion

In the morphological dilation and erosion operations, the state of any given pixel in the output image is determined by applying a rule to the corresponding pixel and its neighbors in the input image. The rule used to process the pixels defines the operation as a dilation or an erosion. This table lists the rules for both dilation and erosion.

**Rules for Dilation and Erosion**

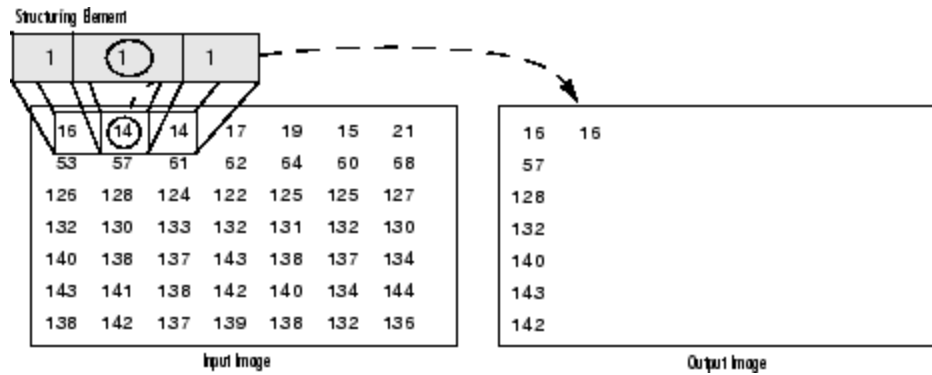
Operation	Rule
Dilation	The value of the output pixel is the <i>maximum</i> value of all the pixels in the input pixel's neighborhood. In a binary image, if any of the pixels is set to the value 1, the output pixel is set to 1.
Erosion	The value of the output pixel is the <i>minimum</i> value of all the pixels in the input pixel's neighborhood. In a binary image, if any of the pixels is set to 0, the output pixel is set to 0.

The following figure illustrates the dilation of a binary image. Note how the structuring element defines the neighborhood of the pixel of interest, which is circled. (See "Structuring Elements" on page 10-6 for more information.) The dilation function applies the appropriate rule to the pixels in the neighborhood and assigns a value to the corresponding pixel in the output image. In the figure, the morphological dilation function sets the value of the output pixel to 1 because one of the elements in the neighborhood defined by the structuring element is on.



**Morphological Dilation of a Binary Image**

The following figure illustrates this processing for a grayscale image. The figure shows the processing of a particular pixel in the input image. Note how the function applies the rule to the input pixel's neighborhood and uses the highest value of all the pixels in the neighborhood as the value of the corresponding pixel in the output image.



### Morphological Dilation of a Grayscale Image

#### Processing Pixels at Image Borders (Padding Behavior)

Morphological functions position the origin of the structuring element, its center element, over the pixel of interest in the input image. For pixels at the edge of an image, parts of the neighborhood defined by the structuring element can extend past the border of the image.

To process border pixels, the morphological functions assign a value to these undefined pixels, as if the functions had padded the image with additional rows and columns. The value of these padding pixels varies for dilation and erosion operations. The following table describes the padding rules for dilation and erosion for both binary and grayscale images.

### Rules for Padding Images

Operation	Rule
Dilation	<p>Pixels beyond the image border are assigned the minimum value afforded by the data type.</p> <p>For binary images, these pixels are assumed to be set to 0. For grayscale images, the minimum value for uint8 images is 0.</p>
Erosion	<p>Pixels beyond the image border are assigned the <i>maximum</i> value afforded by the data type.</p> <p>For binary images, these pixels are assumed to be set to 1. For grayscale images, the maximum value for uint8 images is 255.</p>

---

**Note** By using the minimum value for dilation operations and the maximum value for erosion operations, the toolbox avoids *border effects*, where regions near the borders of the output image do not appear to be homogeneous with the rest of the image. For example, if erosion padded with a minimum value, eroding an image would result in a black border around the edge of the output image.

---

### Structuring Elements

An essential part of the dilation and erosion operations is the structuring element used to probe the input image. A structuring element is a matrix consisting of only 0's and 1's that can have any arbitrary shape and size. The pixels with values of 1 define the neighborhood.

Two-dimensional, or *flat*, structuring elements are typically much smaller than the image being processed. The center pixel of the structuring element, called the *origin*, identifies the pixel of interest -- the pixel being processed. The pixels in the structuring element containing 1's define the *neighborhood* of the structuring element. These pixels are also considered in dilation or erosion processing.



Three-dimensional, or *nonflat*, structuring elements use 0's and 1's to define the extent of the structuring element in the  $x$ - and  $y$ -planes and add height values to define the third dimension.

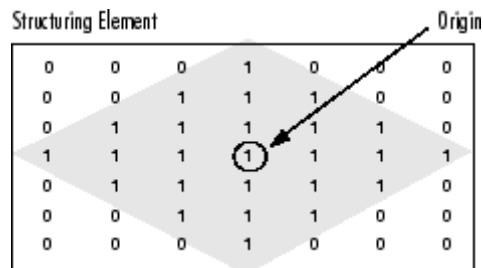
### The Origin of a Structuring Element

The morphological functions use this code to get the coordinates of the origin of structuring elements of any size and dimension.

```
origin = floor((size(nhood)+1)/2)
```

(In this code `nhood` is the neighborhood defining the structuring element. Because structuring elements are MATLAB objects, you cannot use the size of the STREL object itself in this calculation. You must use the STREL `getnhood` method to retrieve the neighborhood of the structuring element from the STREL object. For information about other STREL object methods, see the `strel` function reference page.)

For example, the following illustrates a diamond-shaped structuring element.



### Origin of a Diamond-Shaped Structuring Element

### Creating a Structuring Element

The toolbox dilation and erosion functions accept structuring element objects, called STRELS. You use the `strel` function to create STRELS of any arbitrary size and shape. The `strel` function also includes built-in support for many common shapes, such as lines, diamonds, disks, periodic lines, and balls.

---

**Note** You typically choose a structuring element the same size and shape as the objects you want to process in the input image. For example, to find lines in an image, create a linear structuring element.

---

For example, this code creates a flat, diamond-shaped structuring element.

```
se = strel('diamond',3)
se =
```

```
Flat STREL object containing 25 neighbors.
```

```
Decomposition: 3 STREL objects containing a total of 13 neighbors
```

```
Neighborhood:
```

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

### Structuring Element Decomposition

To enhance performance, the `strel` function might break structuring elements into smaller pieces, a technique known as *structuring element decomposition*.

For example, dilation by an 11-by-11 square structuring element can be accomplished by dilating first with a 1-by-11 structuring element, and then with an 11-by-1 structuring element. This results in a theoretical speed improvement of a factor of 5.5, although in practice the actual speed improvement is somewhat less.

Structuring element decompositions used for the 'disk' and 'ball' shapes are approximations; all other decompositions are exact. Decomposition is not used with an arbitrary structuring element unless it is a flat structuring element whose neighborhood is all 1's.

To view the sequence of structuring elements used in a decomposition, use the STREL `getsequence` method. The `getsequence` function returns an array of the structuring elements that form the decomposition. For example, here are the structuring elements created in the decomposition of a diamond-shaped structuring element.

```
sel = strel('diamond',4)
sel =
Flat STREL object containing 41 neighbors.
Decomposition: 3 STREL objects containing a total of 13 neighbors
```

Neighborhood:

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

```
seq = getsequence(sel)
seq =
3x1 array of STREL objects
```

```
seq(1)
ans =
Flat STREL object containing 5 neighbors.
```

Neighborhood:

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

```
seq(2)
ans =
Flat STREL object containing 4 neighbors.
```

Neighborhood:

```

    0   1   0
    1   0   1
    0   1   0

```

```

seq(3)
ans =
Flat STREL object containing 4 neighbors.

```

```

Neighborhood:
    0   0   1   0   0
    0   0   0   0   0
    1   0   0   0   1
    0   0   0   0   0
    0   0   1   0   0

```

## Dilating an Image

To dilate an image, use the `imdilate` function. The `imdilate` function accepts two primary arguments:

- The input image to be processed (grayscale, binary, or packed binary image)
- A structuring element object, returned by the `strel` function, or a binary matrix defining the neighborhood of a structuring element

`imdilate` also accepts two optional arguments: `PADOPT` and `PACKOPT`. The `PADOPT` argument affects the size of the output image. The `PACKOPT` argument identifies the input image as packed binary. (Packing is a method of compressing binary images that can speed up the processing of the image. See the `bwpack` reference page for information.)

This example dilates a simple binary image containing one rectangular object.

```

BW = zeros(9,10);
BW(4:6,4:7) = 1
BW =
    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   1   1   1   1   0   0   0
    0   0   0   1   1   1   1   0   0   0

```

```

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  0  0  0  0
0  0  0  0  0  0  0  0  0  0

```

To expand all sides of the foreground component, the example uses a 3-by-3 square structuring element object. (For more information about using the `strel` function, see “Structuring Elements” on page 10-6.)

```

SE = strel('square',3)
SE =

```

Flat STREL object containing 3 neighbors.

```

Neighborhood:
 1  1  1
 1  1  1
 1  1  1

```

To dilate the image, pass the image `BW` and the structuring element `SE` to the `imdilate` function. Note how dilation adds a rank of 1's to all sides of the foreground object.

```

BW2 = imdilate(BW,SE)

```

```

BW2 =
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  1  1  1  1  1  1  0  0
0  0  1  1  1  1  1  1  0  0
0  0  1  1  1  1  1  1  0  0
0  0  1  1  1  1  1  1  0  0
0  0  1  1  1  1  1  1  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0

```

## Eroding an Image

To erode an image, use the `imerode` function. The `imerode` function accepts two primary arguments:

- The input image to be processed (grayscale, binary, or packed binary image)
- A structuring element object, returned by the `strel` function, or a binary matrix defining the neighborhood of a structuring element

`imerode` also accepts three optional arguments: `PADOPT`, `PACKOPT`, and `M`.

The `PADOPT` argument affects the size of the output image. The `PACKOPT` argument identifies the input image as packed binary. If the image is packed binary, `M` identifies the number of rows in the original image. (Packing is a method of compressing binary images that can speed up the processing of the image. See the `bwpack` reference page for more information.)

The following example erodes the binary image `circbw.tif`:

- 1 Read the image into the MATLAB workspace.

```
BW1 = imread('circbw.tif');
```

- 2 Create a structuring element. The following code creates a diagonal structuring element object. (For more information about using the `strel` function, see “Structuring Elements” on page 10-6.)

```
SE = strel('arbitrary',eye(5));  
SE=
```

Flat STREL object containing 5 neighbors.

Neighborhood:

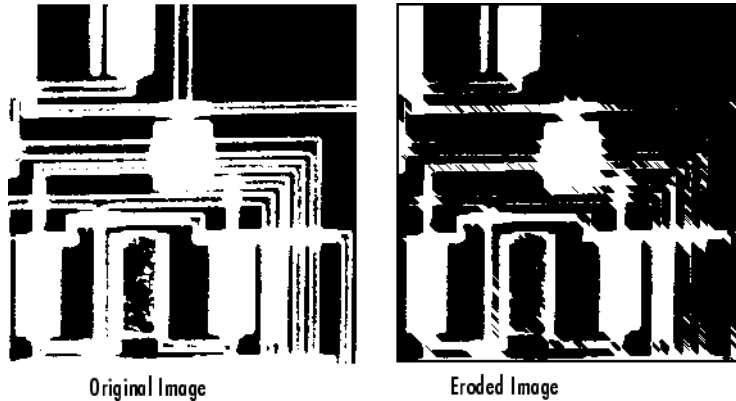
```
 1   0   0   0   0  
 0   1   0   0   0  
 0   0   1   0   0  
 0   0   0   1   0  
 0   0   0   0   1
```

- 3 Call the `imerode` function, passing the image `BW` and the structuring element `SE` as arguments.

```
BW2 = imerode(BW1,SE);
```

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

```
imshow(BW1)
figure, imshow(BW2)
```



## Combining Dilation and Erosion

Dilation and erosion are often used in combination to implement image processing operations. For example, the definition of a morphological *opening* of an image is an erosion followed by a dilation, using the same structuring element for both operations. The related operation, morphological *closing* of an image, is the reverse: it consists of dilation followed by an erosion with the same structuring element.

The following section uses `imdilate` and `imerode` to illustrate how to implement a morphological opening. Note, however, that the toolbox already includes the `imopen` function, which performs this processing. The toolbox includes functions that perform many common morphological operations. See “Dilation- and Erosion-Based Functions” on page 10-15 for a complete list.

## Morphological Opening

You can use morphological opening to remove small objects from an image while preserving the shape and size of larger objects in the image. For example, you can use the `imopen` function to remove all the circuit lines from the original circuit image, `circbw.tif`, creating an output image that contains only the rectangular shapes of the microchips.

To morphologically open the image, perform these steps:

- 1 Read the image into the MATLAB workspace.

```
BW1 = imread('circbw.tif');
```

- 2 Create a structuring element.

```
SE = strel('rectangle',[40 30]);
```

The 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 contiguous patches of foreground pixels.

- 3 Erode the image with the structuring element.

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

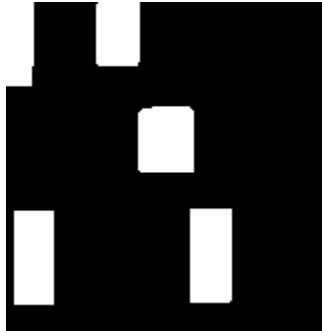
This removes all the lines, but also shrinks the rectangles.



- 4 To restore the rectangles to their original sizes, dilate the eroded image using the same structuring element, SE.

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





## Dilation- and Erosion-Based Functions

This section describes two common image processing operations that are based on dilation and erosion:

- Skeletonization
- Perimeter determination

This table lists other functions in the toolbox that perform common morphological operations that are based on dilation and erosion. For more information about these functions, see their reference pages.

### Dilation- and Erosion-Based Functions

Function	Morphological Definition
<code>bwhitmiss</code>	Logical AND of an image, eroded with one structuring element, and the image's complement, eroded with a second structuring element.
<code>imbothat</code>	Subtracts the original image from a morphologically closed version of the image. Can be used to find intensity troughs in an image.
<code>imclose</code>	Dilates an image and then erodes the dilated image using the same structuring element for both operations.

**Dilation- and Erosion-Based Functions (Continued)**

Function	Morphological Definition
imopen	Erodes an image and then dilates the eroded image using the same structuring element for both operations.
imtophat	Subtracts a morphologically opened image from the original image. Can be used to enhance contrast in an image.

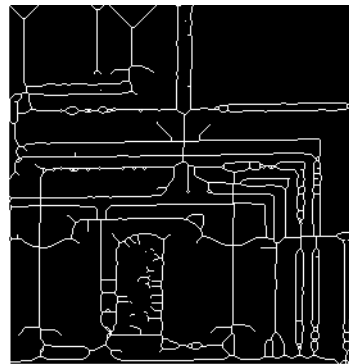
**Skeletonization**

To reduce all objects in an image to lines, without changing the essential structure of the image, use the `bwmorph` function. This process is known as *skeletonization*.

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



Original Image



Skeletonization of Image

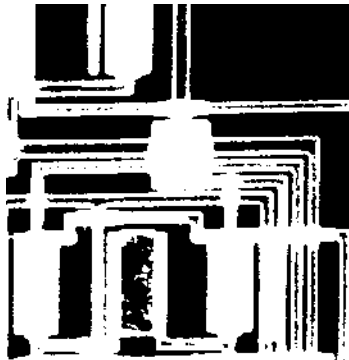
## Perimeter Determination

The `bwperim` function determines the perimeter pixels of the objects in a binary image. A pixel is considered a perimeter pixel if it satisfies both of these criteria:

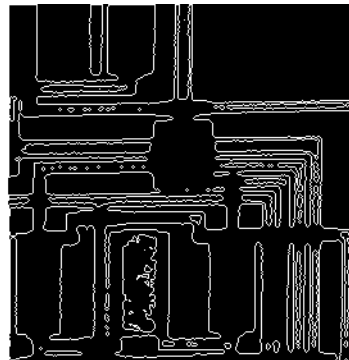
- The pixel is on.
- One (or more) of the pixels in its neighborhood is off.

For example, this code finds the perimeter pixels in a binary image of a circuit board.

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



Original Image



Perimeters Determined

## Morphological Reconstruction

Morphological reconstruction is another major part of morphological image processing. Based on dilation, morphological reconstruction has these unique properties:

- Processing is based on two images, a marker and a mask, rather than one image and a structuring element.
- Processing repeats until stability; i.e., the image no longer changes.
- Processing is based on the concept of connectivity, rather than a structuring element.

This section

- Provides background information about morphological reconstruction and describes how to use the `imreconstruct` function
- Describes how pixel connectivity affects morphological reconstruction
- Describes how to use the `imfill` function, which is based on morphological reconstruction
- Describes a group of functions, all based on morphological reconstruction, that process image extrema, i.e., the areas of high and low intensity in images

### Marker and Mask

Morphological reconstruction processes one image, called the *marker*, based on the characteristics of another image, called the *mask*. The high points, or peaks, in the marker image specify where processing begins. The processing continues until the image values stop changing.

To illustrate morphological reconstruction, consider this simple image. It contains two primary regions, the blocks of pixels containing the values 14 and 18. The background is primarily all set to 10, with some pixels set to 11.

```

A = [10  10  10  10  10  10  10  10  10  10;
     10  14  14  14  10  10  11  10  11  10;
     10  14  14  14  10  10  10  11  10  10;
     10  14  14  14  10  10  11  10  11  10;
     10  10  10  10  10  10  10  10  10  10;
     10  11  10  10  10  18  18  18  10  10;
     10  10  10  11  10  18  18  18  10  10;
     10  10  11  10  10  18  18  18  10  10;
     10  11  10  11  10  10  10  10  10  10;
     10  10  10  10  10  10  11  10  10  10];

```

To morphologically reconstruct this image, perform these steps:

- 1 Create a marker image. As with the structuring element in dilation and erosion, the characteristics of the marker image determine the processing performed in morphological reconstruction. The peaks in the marker image should identify the location of objects in the mask image that you want to emphasize.

One way to create a marker image is to subtract a constant from the mask image, using `imsubtract`.

```

marker = imreadsubtract(A,2)
marker =
     8     8     8     8     8     8     8     8     8     8
     8    12    12    12     8     8     9     8     9     8
     8    12    12    12     8     8     8     9     8     8
     8    12    12    12     8     8     9     8     9     8
     8     8     8     8     8     8     8     8     8     8
     8     9     8     8     8    16    16    16     8     8
     8     8     8     9     8    16    16    16     8     8
     8     8     9     8     8    16    16    16     8     8
     8     9     8     9     8     8     8     8     8     8
     8     8     8     8     8     8     9     8     8     8

```

- 2 Call the `imreconstruct` function to morphologically reconstruct the image. In the output image, note how all the intensity fluctuations except the intensity peak have been removed.

```

recon = imreconstruct(marker, mask)

```

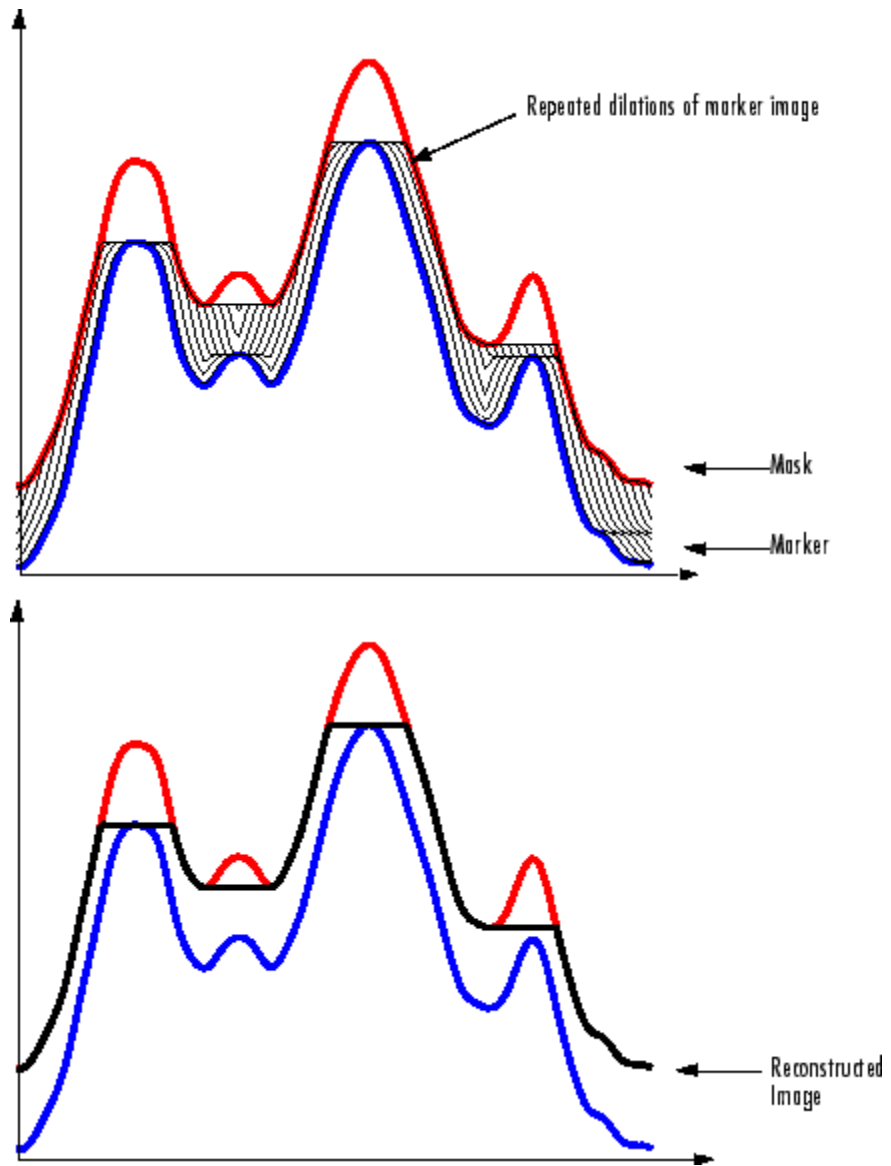
```

recon =
    10    10    10    10    10    10    10    10    10    10
    10    12    12    12    10    10    10    10    10    10
    10    12    12    12    10    10    10    10    10    10
    10    12    12    12    10    10    10    10    10    10
    10    10    10    10    10    10    10    10    10    10
    10    10    10    10    10    16    16    16    10    10
    10    10    10    10    10    16    16    16    10    10
    10    10    10    10    10    16    16    16    10    10
    10    10    10    10    10    10    10    10    10    10
    10    10    10    10    10    10    10    10    10    10
  
```

## Understanding Morphological Reconstruction

Morphological reconstruction can be thought of conceptually as repeated dilations of the marker image until the contour of the marker image fits under the mask image. In this way, the peaks in the marker image “spread out,” or dilate.

This figure illustrates this processing in 1-D. Each successive dilation is constrained to lie underneath the mask. When further dilation ceases to change the image, processing stops. The final dilation is the reconstructed image. (Note: the actual implementation of this operation in the toolbox is done much more efficiently. See the `imreconstruct` reference page for more details.) The figure shows the successive dilations of the marker.



**Repeated Dilations of Marker Image, Constrained by Mask**

## Pixel Connectivity

Morphological processing starts at the peaks in the marker image and spreads throughout the rest of the image based on the connectivity of the pixels. Connectivity defines which pixels are connected to other pixels. A set of pixels in a binary image that form a connected group is called an *object* or a *connected component*.

For example, this binary image contains one foreground object--all the pixels that are set to 1. If the foreground is 4-connected, the image has one background object, and all the pixels are set to 0. However, if the foreground is 8-connected, the foreground makes a closed loop and the image has two separate background objects: the pixels in the loop and the pixels outside the loop.

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

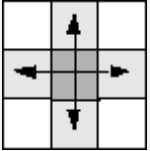
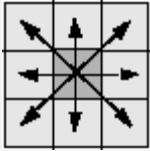
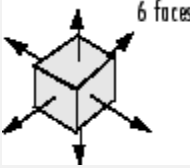
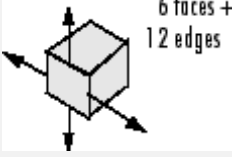
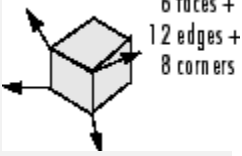
## Defining Connectivity in an Image

The following table lists all the standard two- and three-dimensional connectivities supported by the toolbox. See these sections for more information:

- “Choosing a Connectivity” on page 10-24
- “Specifying Custom Connectivities” on page 10-24



## Supported Connectivities

<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. 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.	
8-connected	Pixels are connected if their edges or corners touch. 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.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch.	
18-connected	Pixels are connected if their faces or edges touch.	
26-connected	Pixels are connected if their faces, edges, or corners touch.	

### Choosing a Connectivity

The type of neighborhood you choose affects the number of objects found in an image and the boundaries of those objects. For this reason, the results of many morphology operations often differ depending upon the type of connectivity you specify.

For example, if you specify a 4-connected neighborhood, this binary image contains two objects; if you specify an 8-connected neighborhood, the image has one object.

```
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
```

### Specifying Custom Connectivities

You can also define custom neighborhoods by specifying a 3-by-3-by-...-by-3 array of 0's and 1's. The 1-valued elements define the connectivity of the neighborhood relative to the center element.

For example, this array defines a "North/South" connectivity that has the effect of breaking up an image into independent columns.

```
CONN = [ 0 1 0; 0 1 0; 0 1 0 ]
CONN =
    0     1     0
    0     1     0
    0     1     0
```

---

**Note** Connectivity arrays must be symmetric about their center element. Also, you can use a 2-D connectivity array with a 3-D image; the connectivity affects each "page" in the 3-D image.

---

### Flood-Fill Operations

The `imfill` function performs a *flood-fill* operation on binary and grayscale images. For binary images, `imfill` changes connected background pixels (0's)

to foreground pixels (1's), stopping when it reaches object boundaries. For grayscale images, `imfill` brings the intensity values of dark areas that are surrounded by lighter areas up to the same intensity level as surrounding pixels. (In effect, `imfill` removes regional minima that are not connected to the image border. See “Finding Areas of High or Low Intensity” on page 10-29 for more information.) This operation can be useful in removing irrelevant artifacts from images.

This section includes information about

- Specifying the connectivity in flood-fill operations
- Specifying the starting point for binary image fill operations
- Filling holes in binary or grayscale images

### Specifying Connectivity

For both binary and grayscale images, the boundary of the fill operation is determined by the connectivity you specify.

---

**Note** `imfill` differs from the other object-based operations in that it operates on *background* pixels. When you specify connectivity with `imfill`, you are specifying the connectivity of the background, not the foreground.

---

The implications of connectivity can be illustrated with this matrix.

```
BW = [ 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];
```

If the background is 4-connected, this binary image contains two separate background elements (the part inside the loop and the part outside). If the background is 8-connected, the pixels connect diagonally, and there is only one background element.

### Specifying the Starting Point

For binary images, you can specify the starting point of the fill operation by passing in the location subscript or by using `imfill` in interactive mode, selecting starting pixels with a mouse. See the reference page for `imfill` for more information about using `imfill` interactively.

For example, if you call `imfill`, specifying the pixel `BW(4,3)` as the starting point, `imfill` only fills the inside of the loop because, by default, the background is 4-connected.

```
imfill(BW,[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
```

If you specify the same starting point, but use an 8-connected background connectivity, `imfill` fills the entire image.

```
imfill(BW,[4 3],8)
```

```
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
```

### Filling Holes

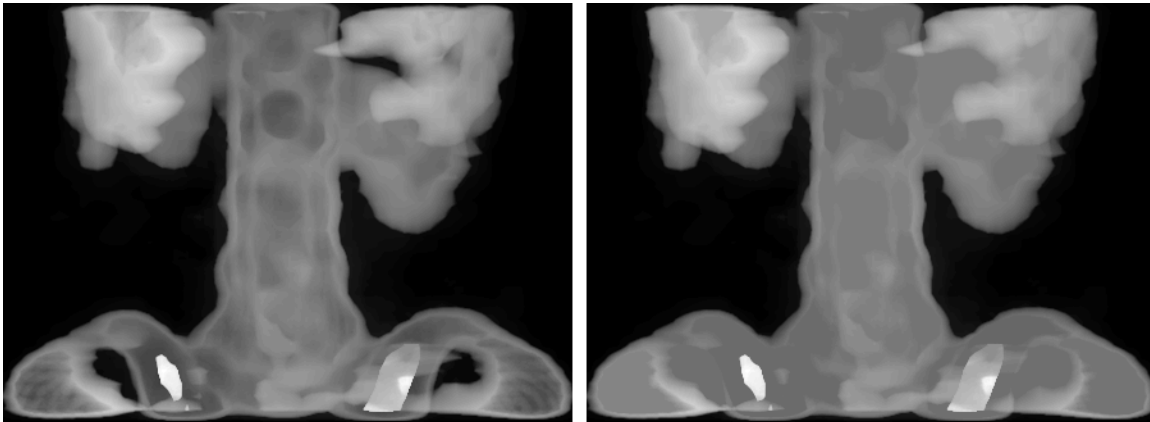
A common use of the flood-fill operation is to fill holes in images. For example, suppose you have an image, binary or grayscale, in which the foreground

objects represent spheres. In the image, these objects should appear as disks, but instead are donut shaped because of reflections in the original photograph. Before doing any further processing of the image, you might want to first fill in the “donut holes” using `imfill`.

Because the use of flood-fill to fill holes is so common, `imfill` includes special syntax to support it for both binary and grayscale images. In this syntax, you just specify the argument `'holes'`; you do not have to specify starting locations in each hole.

To illustrate, this example fills holes in a grayscale image of a spinal column.

```
[X,map] = imread('spine.tif');  
I = ind2gray(X,map);  
Ifill = imfill(I,'holes');  
imshow(I);figure, imshow(Ifill)
```



Original

After Filling Holes

## Finding Peaks and Valleys

Grayscale images can be thought of in three dimensions: the  $x$ - and  $y$ -axes represent pixel positions and the  $z$ -axis represents the intensity of each pixel. In this interpretation, the intensity values represent elevations, as in a topographical map. The areas of high intensity and low intensity in an image,

peaks and valleys in topographical terms, can be important morphological features because they often mark relevant image objects.

For example, in an image of several spherical objects, points of high intensity could represent the tops of the objects. Using morphological processing, these maxima can be used to identify objects in an image.

This section covers these topics:

- “Terminology” on page 10-28
- “Understanding the Maxima and Minima Functions” on page 10-29
- “Finding Areas of High or Low Intensity” on page 10-29
- “Suppressing Minima and Maxima” on page 10-31
- “Imposing a Minimum” on page 10-33

### **Terminology**

This section uses the following terms.

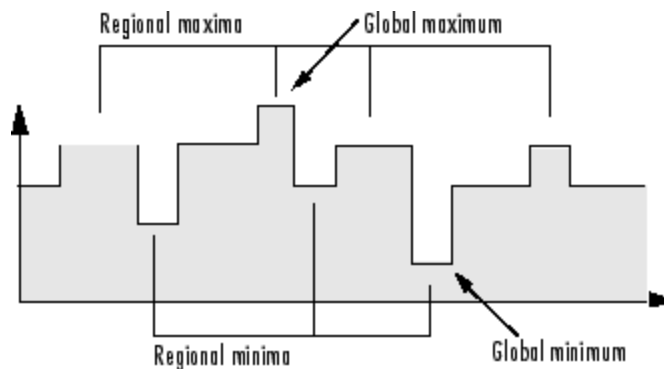
<b>Term</b>	<b>Definition</b>
<b>global maxima</b>	Highest regional maxima in the image. See the entry for regional maxima in this table for more information.
<b>global minima</b>	Lowest regional minima in the image. See the entry for regional minima in this table for more information.

Term	Definition
<b>regional maxima</b>	Connected set of pixels of constant intensity from which it is impossible to reach a point with higher intensity without first descending; that is, a connected component of pixels with the same intensity value, $t$ , surrounded by pixels that all have a value less than $t$ .
<b>regional minima</b>	Connected set of pixels of constant intensity from which it is impossible to reach a point with lower intensity without first ascending; that is, a connected component of pixels with the same intensity value, $t$ , surrounded by pixels that all have a value greater than $t$ .

### Understanding the Maxima and Minima Functions

An image can have multiple regional maxima or minima but only a single global maximum or minimum. Determining image peaks or valleys can be used to create marker images that are used in morphological reconstruction.

This figure illustrates the concept in 1-D.



### Finding Areas of High or Low Intensity

The toolbox includes functions that you can use to find areas of high or low intensity in an image:

- The `imregionalmax` and `imregionalmin` functions identify *all* regional minima or maxima.
- The `imextendedmax` and `imextendedmin` functions identify all regional minima or maxima that are greater than or less than a specified threshold.

The functions accept a grayscale image as input and return a binary image as output. In the output binary image, the regional minima or maxima are set to 1; all other pixels are set to 0.

For example, this simple image contains two primary regional maxima, the blocks of pixels containing the value 13 and 18, and several smaller maxima, set to 11.

```
A = [10  10  10  10  10  10  10  10  10  10;
      10  13  13  13  10  10  11  10  11  10;
      10  13  13  13  10  10  10  11  10  10;
      10  13  13  13  10  10  11  10  11  10;
      10  10  10  10  10  10  10  10  10  10;
      10  11  10  10  10  18  18  18  10  10;
      10  10  10  11  10  18  18  18  10  10;
      10  10  11  10  10  18  18  18  10  10;
      10  11  10  11  10  10  10  10  10  10;
      10  10  10  10  10  10  11  10  10  10];
```

The binary image returned by `imregionalmax` pinpoints all these regional maxima.

```
B = imregionalmax(A)
```

```
B =
  0  0  0  0  0  0  0  0  0  0
  0  1  1  1  0  0  1  0  1  0
  0  1  1  1  0  0  0  1  0  0
  0  1  1  1  0  0  1  0  1  0
  0  0  0  0  0  0  0  0  0  0
  0  1  0  0  0  1  1  1  0  0
  0  0  0  1  0  1  1  1  0  0
  0  0  1  0  0  1  1  1  0  0
  0  1  0  1  0  0  0  0  0  0
  0  0  0  0  0  0  1  0  0  0
```



You might want only to identify areas of the image where the change in intensity is extreme; that is, the difference between the pixel and neighboring pixels is greater than (or less than) a certain threshold. For example, to find only those regional maxima in the sample image, A, that are at least two units higher than their neighbors, use `imextendedmax`.

```
B = imextendedmax(A,2)
```

B =

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

### Suppressing Minima and Maxima

In an image, every small fluctuation in intensity represents a regional minimum or maximum. You might only be interested in significant minima or maxima and not in these smaller minima and maxima caused by background texture.

To remove the less significant minima and maxima but retain the significant minima and maxima, use the `imhmax` or `imhmin` function. With these functions, you can specify a contrast criteria or threshold level,  $h$ , that suppresses all maxima whose height is less than  $h$  or whose minima are greater than  $h$ .

---

**Note** The `imregionalmin`, `imregionalmax`, `imextendedmin`, and `imextendedmax` functions return a binary image that marks the locations of the regional minima and maxima in an image. The `imhmax` and `imhmin` functions produce an altered image.

---

For example, this simple image contains two primary regional maxima, the blocks of pixels containing the value 14 and 18, and several smaller maxima, set to 11.

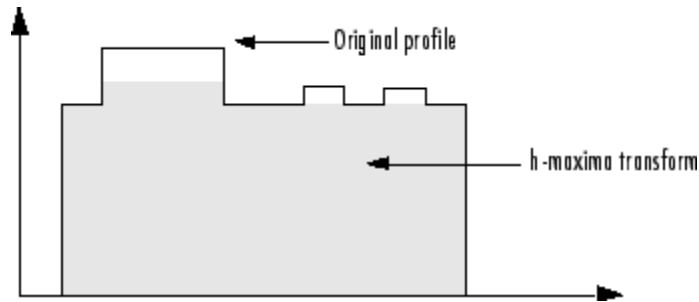
```
A = [10  10  10  10  10  10  10  10  10  10;
      10  14  14  14  10  10  11  10  11  10;
      10  14  14  14  10  10  10  11  10  10;
      10  14  14  14  10  10  11  10  11  10;
      10  10  10  10  10  10  10  10  10  10;
      10  11  10  10  10  18  18  18  10  10;
      10  10  10  11  10  18  18  18  10  10;
      10  10  11  10  10  18  18  18  10  10;
      10  11  10  11  10  10  10  10  10  10;
      10  10  10  10  10  10  11  10  10  10];
```

To eliminate all regional maxima except the two significant maxima, use `imhmax`, specifying a threshold value of 2. Note that `imhmax` only affects the maxima; none of the other pixel values are changed. The two significant maxima remain, although their heights are reduced.

```
B = imhmax(A,2)
```

```
B =
10  10  10  10  10  10  10  10  10  10
10  12  12  12  10  10  10  10  10  10
10  12  12  12  10  10  10  10  10  10
10  12  12  12  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  16  16  16  10  10
10  10  10  10  10  16  16  16  10  10
10  10  10  10  10  16  16  16  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
```

This figure takes the second row from the sample image to illustrate in 1-D how `imhmax` changes the profile of the image.



### Imposing a Minimum

You can emphasize specific minima (dark objects) in an image using the `imimposemin` function. The `imimposemin` function uses morphological reconstruction to eliminate all minima from the image except the minima you specify.

To illustrate the process of imposing a minimum, this code creates a simple image containing two primary regional minima and several other regional minima.

```
mask = uint8(10*ones(10,10));
mask(6:8,6:8) = 2;
mask(2:4,2:4) = 7;
mask(3,3) = 5;
mask(2,9) = 9;
mask(3,8) = 9;
mask(9,2) = 9;
mask(8,3) = 9;
```

```
mask = 10  10  10  10  10  10  10  10  10  10
      10  7  7  7  10  10  10  10  9  10
      10  7  6  7  10  10  10  9  10  10
      10  7  7  7  10  10  10  10  10  10
      10  10  10  10  10  10  10  10  10  10
      10  10  10  10  10  2  2  2  10  10
      10  10  10  10  10  2  2  2  10  10
      10  10  9  10  10  2  2  2  10  10
      10  9  10  10  10  10  10  10  10  10
      10  10  10  10  10  10  10  10  10  10
```

### Creating a Marker Image

To obtain an image that emphasizes the two deepest minima and removes all others, create a marker image that pinpoints the two minima of interest. You can create the marker image by explicitly setting certain pixels to specific values or by using other morphological functions to extract the features you want to emphasize in the mask image.

This example uses `imextendedmin` to get a binary image that shows the locations of the two deepest minima.

```
marker = imextendedmin(mask,1)
```

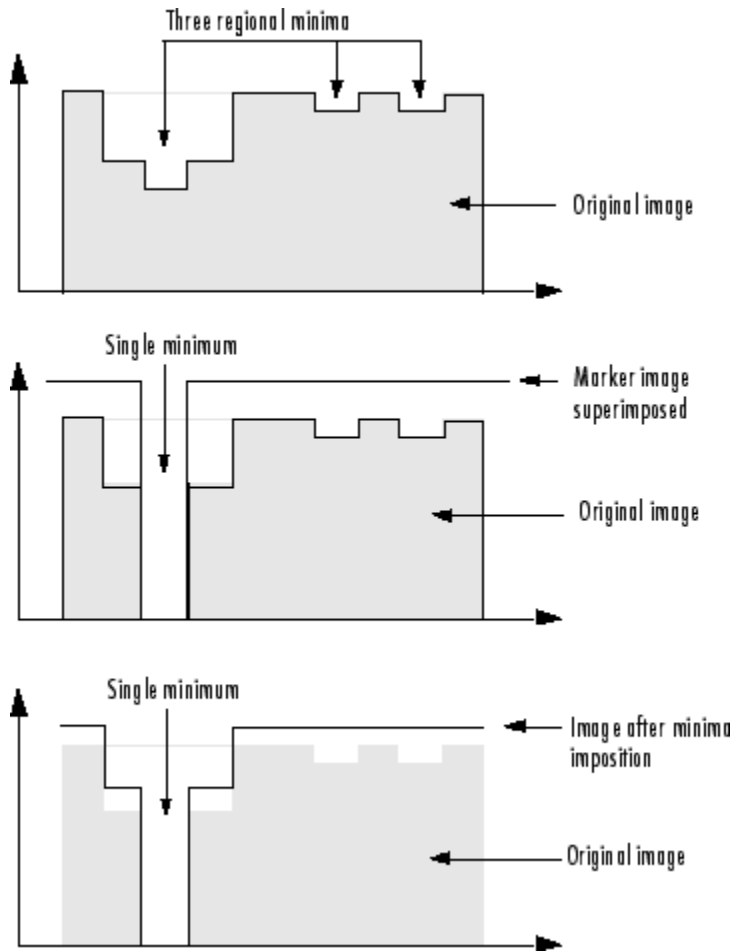
```
marker = 0  0  0  0  0  0  0  0  0  0
          0  0  0  0  0  0  0  0  0  0
          0  0  1  0  0  0  0  0  0  0
          0  0  0  0  0  0  0  0  0  0
          0  0  0  0  0  0  0  0  0  0
          0  0  0  0  0  1  1  1  0  0
          0  0  0  0  0  1  1  1  0  0
          0  0  0  0  0  1  1  1  0  0
          0  0  0  0  0  0  0  0  0  0
          0  0  0  0  0  0  0  0  0  0
```

### Applying the Marker Image to the Mask

Now use `imimposemin` to create new minima in the mask image at the points specified by the marker image. Note how `imimposemin` sets the values of pixels specified by the marker image to the lowest value supported by the datatype (0 for `uint8` values). `imimposemin` also changes the values of all the other pixels in the image to eliminate the other minima.

```
I = imimposemin(mask,marker)
I =
    11    11    11    11    11    11    11    11    11    11
    11     8     8     8    11    11    11    11    11    11
    11     8     0     8    11    11    11    11    11    11
    11     8     8     8    11    11    11    11    11    11
    11    11    11    11    11    11    11    11    11    11
    11    11    11    11    11     0     0     0    11    11
    11    11    11    11    11     0     0     0    11    11
    11    11    11    11    11     0     0     0    11    11
    11    11    11    11    11    11    11    11    11    11
    11    11    11    11    11    11    11    11    11    11
```

This figure illustrates in 1-D how `imimposemin` changes the profile of row 2 of the image.



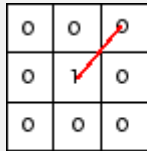
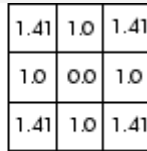
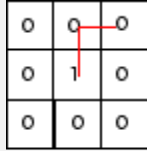

**Imposing a Minimum**

## Distance Transform

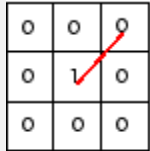
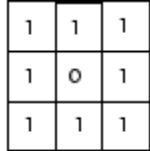
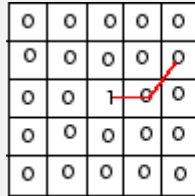
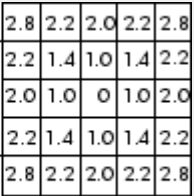
The distance transform provides a metric or measure of the separation of points in the image. The Image Processing Toolbox provides a function, `bwdist`, that calculates the distance between each pixel that is set to off (0) and the nearest nonzero pixel for binary images.

The `bwdist` function supports several distance metrics, listed in the following table.

### Distance Metrics

Distance Metric	Description	Illustration
Euclidean	The Euclidean distance is the straight-line distance between two pixels.	 
City Block	The city block distance metric measures the path between the pixels based on a 4-connected neighborhood. Pixels whose edges touch are 1 unit apart; pixels diagonally touching are 2 units apart.	 

**Distance Metrics (Continued)**

Distance Metric	Description	Illustration
Chessboard	The chessboard distance metric measures the path between the pixels based on an 8-connected neighborhood. Pixels whose edges or corners touch are 1 unit apart.	 
Quasi-Euclidean	The quasi-Euclidean metric measures the total Euclidean distance along a set of horizontal, vertical, and diagonal line segments.	 

This example creates a binary image containing two intersecting circular objects.

```

center1 = -10;
center2 = -center1;
dist = sqrt(2*(2*center1)^2);
radius = dist/2 * 1.4;
lims = [floor(center1-1.2*radius) ceil(center2+1.2*radius)];
[x,y] = meshgrid(lims(1):lims(2));
bw1 = sqrt((x-center1).^2 + (y-center1).^2) <= radius;
bw2 = sqrt((x-center2).^2 + (y-center2).^2) <= radius;
bw = bw1 | bw2;
figure, imshow(bw), title('bw')
    
```



To compute the distance transform of the complement of the binary image, use the `bwdist` function. In the image of the distance transform, note how the centers of the two circular areas are white.



```
D = bwdist(~bw);  
figure, imshow(D,[]), title('Distance transform of ~bw')
```



## Objects, Regions, and Feature Measurement

The toolbox includes several functions that return information about the features in a binary image, including

- Connected-component labeling, and using the label matrix to get statistics about an image
- Selecting objects in a binary image
- Finding the area of the foreground of a binary image
- Finding the Euler number of a binary image

Pixels that are on, i.e., set to the value 1, in a binary image are considered to be the foreground. When you view a binary image, the foreground pixels appear white. Pixels that are off, i.e., set to the value 0, are considered to be the background. When you view a binary image, the background pixels appear black.

### Connected-Component Labeling

The `bwlabel` and the `bwlabeln` functions perform *connected-component labeling*, which is a method for identifying each object in a binary image. The `bwlabel` function supports 2-D inputs only; the `bwlabeln` function supports inputs of any dimension.

These functions return a matrix, called a *label matrix*. A label matrix is an image, the same size as the input image, in which the objects in the input image are distinguished by different integer values in the output matrix. For example, `bwlabel` can identify the objects in this binary image.

```
BW = [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];
```

```

X = bwlabel(BW,4)
X =
     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. (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.)

### Viewing a Label Matrix

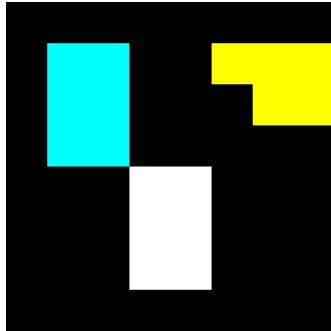
The label matrix returned by `bwlabel` or `bwlabeln` is of class `double`; it is not a binary image. One way to view it is to display it as a pseudocolor indexed image, using `label2rgb`. In the pseudocolor image, each number that identifies an object in the label matrix is used as an index value into the associated colormap matrix. When you view a label matrix as an pseudocolor image, the objects in the image are easier to distinguish.

To illustrate this technique, this example uses `label2rgb` to view the label matrix `X`. The call to `label2rgb` specifies one of the standard MATLAB colormaps, `jet`. The third argument, `'k'`, specifies the background color (black).

```

X = bwlabel(BW1,4);
RGB = label2rgb(X, @jet, 'k');
imshow(RGB,'notruesize')

```



## Using Color to Distinguish Objects in a Binary Image

### Selecting Objects in a Binary Image

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 objects in the image displayed in the current axes. You type

```
BW2 = bwselect;
```

The cursor changes to crosshairs when it is over the image. Click the objects you want to select; `bwselect` displays a small star over each pixel you click. 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.

### Finding the Area of the Foreground of a Binary Image

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 pixels set to on, 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.

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

    0.3456
```

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

## Finding the Euler Number of a Binary Image

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. This section includes the following topics:

- “Creating a Lookup Table” on page 10-44
- “Using a Lookup Table” on page 10-45

### Creating a Lookup Table

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.

⊗	x
x	x

2-by-2 neighborhood

x	x	x
x	⊗	x
x	x	x

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.

---

**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.

---

## Using a Lookup Table

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

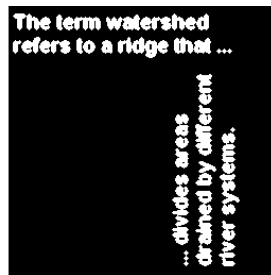
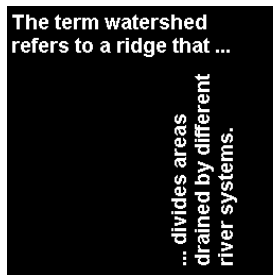
The example below illustrates using lookup table operations to modify an image containing text. The example creates an anonymous function that returns 1 if three or more pixels in the 3-by-3 neighborhood are 1; otherwise, it returns 0. The example then calls `makelut`, passing in this function as the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

```
f = @(x) 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.png');
BW2 = applylut(BW1,lut);
imshow(BW1)
figure, imshow(BW2)
```



### Image Before and After Applying 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`.





# Analyzing and Enhancing Images

---

This chapter describes the Image Processing Toolbox functions that support a range of standard image processing operations for analyzing and enhancing images.

Getting Information about Pixel Values and Statistics (p. 11-2)

Return information about the data values that make up an image

Analyzing an Image (p. 11-11)

Return information about the structure of an image

Analyzing the Texture of an Image (p. 11-24)

Return information about the texture of an image

Intensity Adjustment (p. 11-34)

Improve an image by intensity adjustment

Noise Removal (p. 11-47)

Improve an image by removing noise

## Getting Information about 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

- “Getting Information About Image Pixels” on page 11-2
- “Getting the Intensity Profile of an Image” on page 11-3
- “Displaying a Contour Plot of Image Data” on page 11-7
- “Creating an Image Histogram” on page 11-9
- “Getting Summary Statistics About an Image” on page 11-10
- “Computing Properties for Image Regions” on page 11-10

### Getting Information About Image Pixels

To determine the values of one or more pixels in an image and return the values in a variable, use the `impixel` function. You can specify the pixels by passing their coordinates as input arguments or you can select the pixels interactively using a mouse. `impixel` returns the value of specified pixels in a variable in the MATLAB workspace.

---

**Note** You can also get pixel value information interactively using the Image Tool -- see “Getting Information about the Pixels in an Image” on page 4-24.

---

This example illustrates how to use `impixel` to get pixel values.

- 1 Display an image.

```
imshow canoe.tif
```

- 2 Call `impixel`. When called with no input arguments, `impixel` associates itself with the image in the current axes.

```
vals = impixel
```

- 3** Select the points you want to examine in the image by clicking the mouse. `impixel` places a star at each point you select.



- 4** When you are finished selecting points, press **Return**. `impixel` returns the pixel values in an  $n$ -by-2 array, where  $n$  is the number of points you selected. The stars used to indicate selected points disappear from the image.

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

## Getting the Intensity Profile of an Image

The intensity profile of an image is the 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.

To create an intensity profile, use the `improfile` function. This function calculates and plots the intensity values along a line segment or a multiline

path in an image. You define the line segment (or segments) by specifying their coordinates as input arguments. You can define the line segments using a mouse. (By default, `improfile` uses nearest-neighbor interpolation, but you can specify a different method. For more information, see “Interpolation” on page 6-3.) `improfile` works best with grayscale and truecolor 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 crosshairs when it is over the image. You can then specify line segments by clicking 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. In this figure, the line is shown in red, and is drawn from top to bottom.

```
I = fitsread('solarspectra.fts');  
imshow(I,[]);  
improfile
```

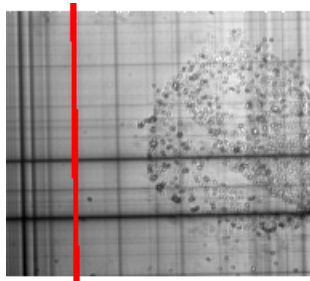
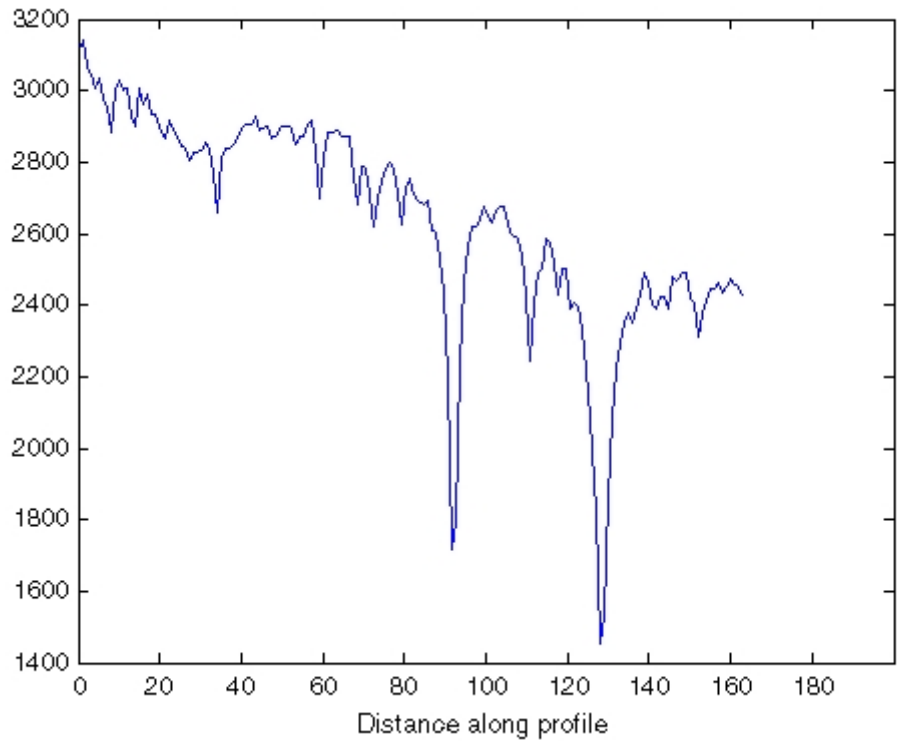


Image Courtesy of Ann Walker

`improfile` displays a plot of the data along the line. Notice the peaks and valleys and how they correspond to the light and dark bands in the image.



### Plot Produced by `improfile`

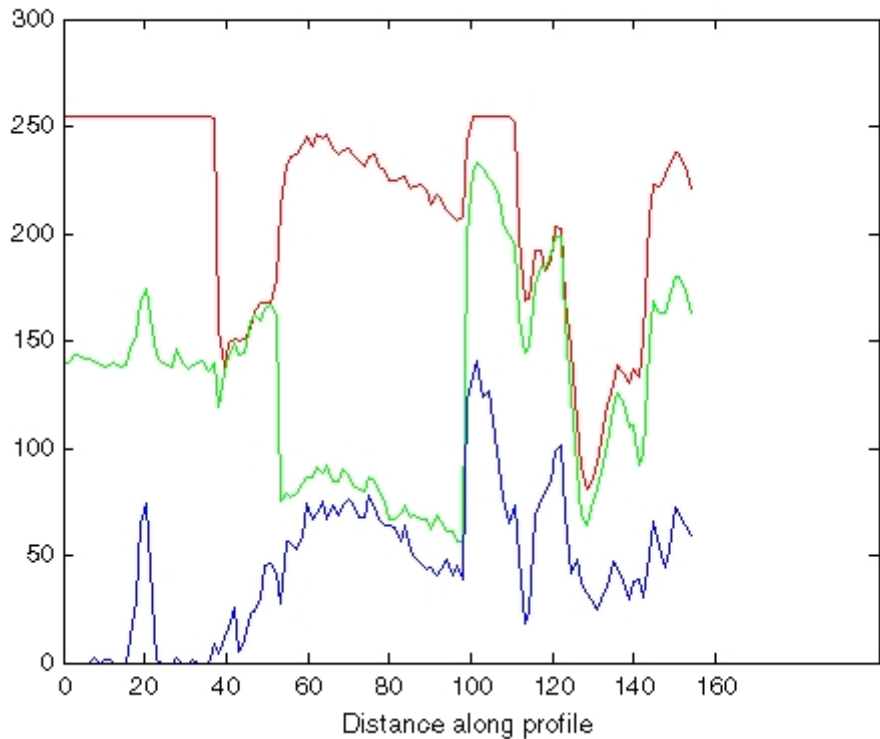
The example below shows how `improfile` works with an RGB image. Use `imshow` to display the image in a figure window. Call `improfile` without any arguments and trace a line segment in the image interactively. In the figure, the black line indicates a line segment drawn from top to bottom. Double-click to end the line segment.

```
imshow peppers.png
improfile
```



**RGB Image with Line Segment Drawn with `improfile`**

The `improfile` function displays a plot of the intensity values along the line segment. The plot includes separate lines for the red, green, and blue intensities. In the plot, notice how low the blue values are at the beginning of the plot where the line traverses the orange pepper.



### Plot of Intensity Values Along a Line Segment in an RGB Image

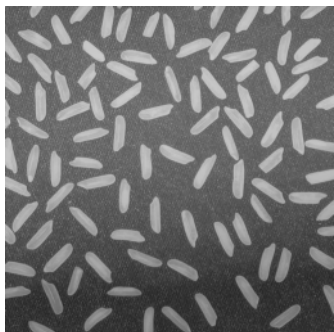
## Displaying a Contour Plot of Image Data

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

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

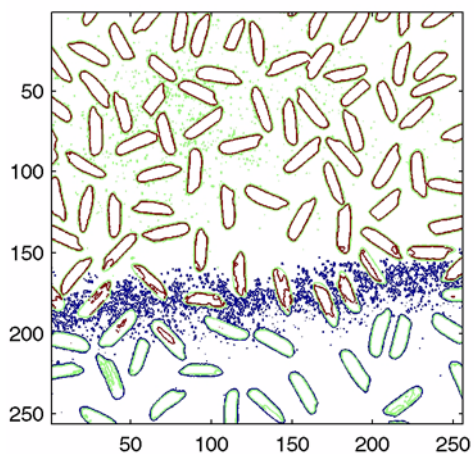
- 1 Read a grayscale image and display it.

```
I = imread('rice.png');  
imshow(I)
```



- 2 Display a contour plot of the grayscale image.

```
figure, imcontour(I,3)
```



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.



## Creating an Image Histogram

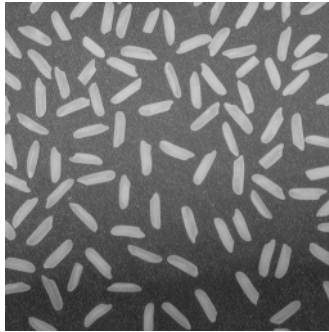
An *image histogram* is a chart that shows the distribution of intensities in an indexed or grayscale image. You can use the information in a histogram to choose an appropriate enhancement operation. For example, if an image histogram shows that the range of intensity values is small, you can use an intensity adjustment function to spread the values across a wider range.

To create an image histogram, use the `imhist` function. This function creates a histogram plot by making `n` equally spaced bins, each representing a range of data values. It then calculates the number of pixels within each range.

The following example displays an image of grains of rice and a histogram based on 64 bins. The histogram shows a peak at around 100, corresponding to the dark gray background in the image. For information about how to modify an image by changing the distribution of its histogram, see “Adjusting Intensity Values to a Specified Range” on page 11-35.

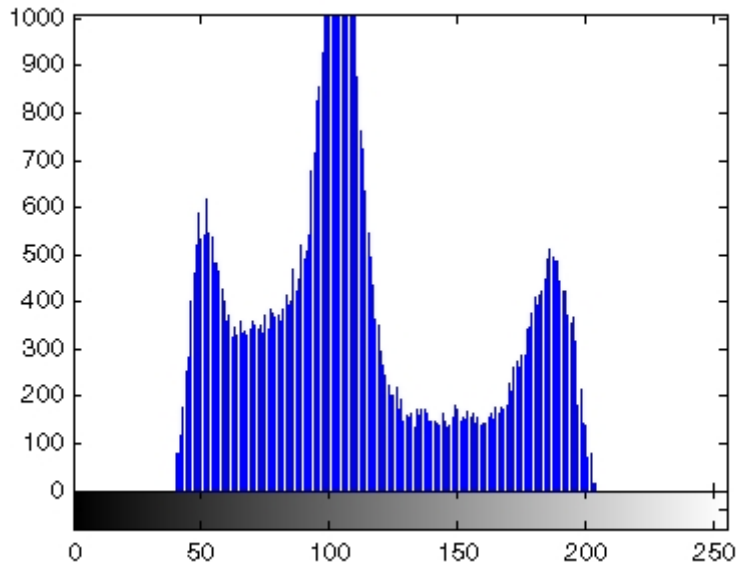
### 1 Read image and display it.

```
I = imread('rice.png');  
imshow(I)
```



### 2 Display histogram of image.

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



## Getting Summary Statistics About an Image

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.

## Computing Properties for Image Regions

You can use the `regionprops` function to compute properties for image regions. For example, `regionprops` can measure such properties as the area, center of mass, and bounding box for a region you specify. See the reference page for `regionprops` for more information.

## Analyzing an Image

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:

- “Detecting Edges” on page 11-11
- “Tracing Boundaries” on page 11-13
- “Detecting Lines Using the Hough Transform” on page 11-17
- “Using Quadtree Decomposition” on page 11-21

The toolbox also includes functions that return information about the texture of an image. See “Analyzing the Texture of an Image” on page 11-24 for more information.

### Detecting Edges

In an image, an edge is 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.

To find edges, you can use the edge function. 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 edges, 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 following example illustrates the power of the Canny edge detector by showing the results of applying the Sobel and Canny edge detectors to the same image:

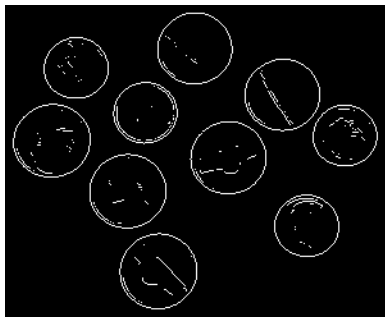
**1** Read image and display it.

```
I = imread('coins.png');  
imshow(I)
```

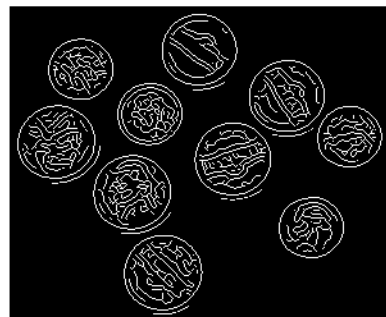


**2** Apply the Sobel and Canny edge detectors to the image and display them.

```
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`.

## Tracing Boundaries

The toolbox includes two functions you can use to find the boundaries of objects in a binary image:

- `bwtraceboundary`
- `bwboundaries`

The `bwtraceboundary` function returns the row and column coordinates of all the pixels on the border of an object in an image. You must specify the location of a border pixel on the object as the starting point for the trace.

The `bwboundaries` function returns the row and column coordinates of border pixels of all the objects in an image.

For both functions, the nonzero pixels in the binary image belong to an object and pixels with the value 0 (zero) constitute the background.

The following example uses `bwtraceboundary` to trace the border of an object in a binary image and then uses `bwboundaries` to trace the borders of all the objects in the image:

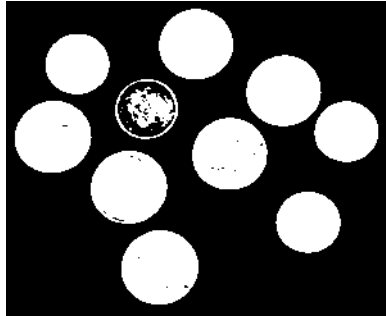
1 Read image and display it.

```
I = imread('coins.png');  
imshow(I)
```



- 2** Convert the image to a binary image. `bwtraceboundary` and `bwboundaries` only work with binary images.

```
BW = im2bw(I);  
imshow(BW)
```



- 3** Determine the row and column coordinates of a pixel on the border of the object you want to trace. `bwboundary` uses this point as the starting location for the boundary tracing.

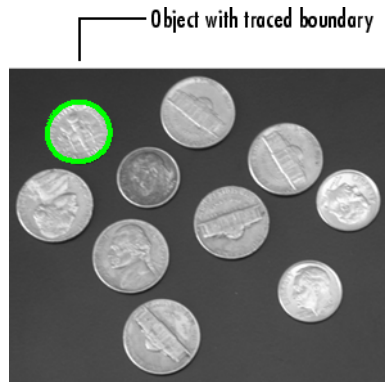
```
dim = size(BW)  
col = round(dim(2)/2)-90;  
row = min(find(BW(:,col)))
```

- 4** Call `bwtraceboundary` to trace the boundary from the specified point. As required arguments, you must specify a binary image, the row and column coordinates of the starting point, and the direction of the first step. The example specifies north ('N'). For information about this parameter, see “Choosing the First Step and Direction for Boundary Tracing” on page 11-16.

```
boundary = bwtraceboundary(BW,[row, col],'N');
```

- 5** Display the original grayscale image and use the coordinates returned by `bwtraceboundary` to plot the border on the image.

```
imshow(I)  
hold on;  
plot(boundary(:,2),boundary(:,1),'g','LineWidth',3);
```



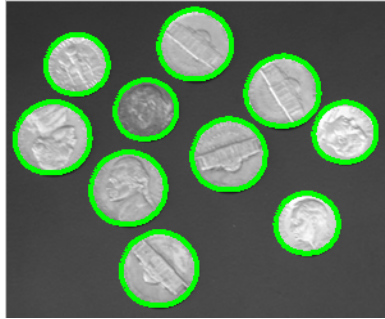
- 6** To trace the boundaries of all the coins in the image, use the `bwboundaries` function. By default, `bwboundaries` finds the boundaries of all objects in an image, including objects inside other objects. In the binary image used in this example, some of the coins contain black areas that `bwboundaries` interprets as separate objects. To ensure that `bwboundaries` only traces the coins, use `imfill` to fill the area inside each coin.

```
BW_filled = imfill(BW, 'holes');
boundaries = bwboundaries(BW_filled);
```

`bwboundaries` returns a cell array, where each cell contains the row/column coordinates for an object in the image.

- 7** Plot the borders of all the coins on the original grayscale image using the coordinates returned by `bwboundaries`.

```
for k=1:10
    b = boundaries{k};
    plot(b(:,2),b(:,1),'g','LineWidth',3);
end
```



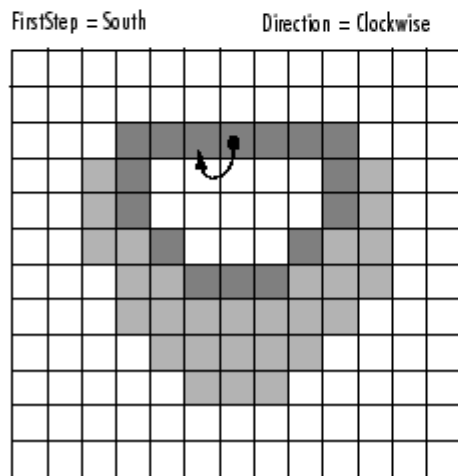
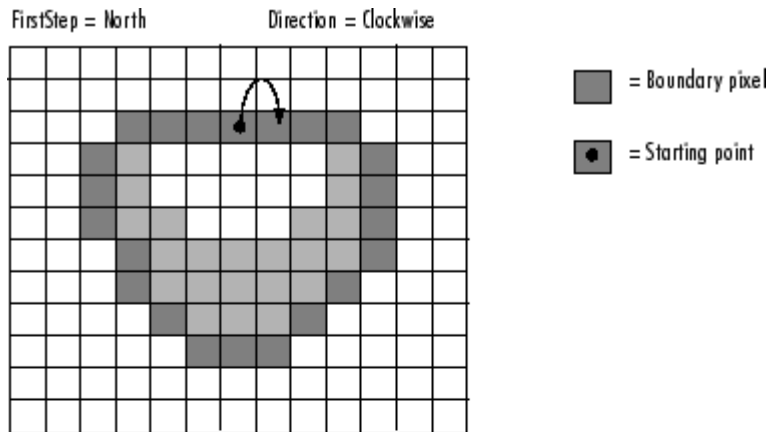
### Choosing the First Step and Direction for Boundary Tracing

For certain objects, you must take care when selecting the border pixel you choose as the starting point and the direction you choose for the first step parameter (north, south, etc.).

For example, if an object contains a hole and you select a pixel on a thin part of the object as the starting pixel, you can trace the outside border of the object or the inside border of the hole, depending on the direction you choose for the first step. For filled objects, the direction you select for the first step parameter is not as important.

To illustrate, this figure shows the pixels traced when the starting pixel is on a thin part of the object and the first step is set to north and south. The connectivity is set to 8 (the default).





### Impact of First Step and Direction Parameters on Boundary Tracing

## Detecting Lines Using the Hough Transform

The Image Processing Toolbox includes functions that support the Hough transform.

- hough
- houghpeaks
- houghlines

The `hough` function implements the Standard Hough Transform (SHT). The Hough transform is designed to detect lines, using the parametric representation of a line:

$$\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta)$$

The variable `rho` is the distance from the origin to the line along a vector perpendicular to the line. `theta` is the angle between the x-axis and this vector. The `hough` function generates a parameter space matrix whose rows and columns correspond to these `rho` and `theta` values, respectively.

The `houghpeaks` function finds peak values in this space, which represent potential lines in the input image.

The `houghlines` function finds the endpoints of the line segments corresponding to peaks in the Hough transform and it automatically fills in small gaps.

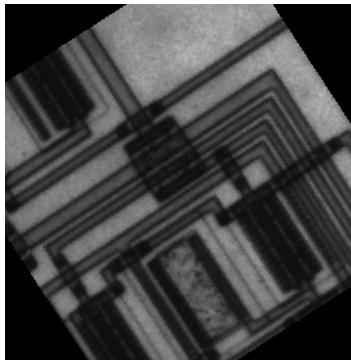
The following example shows how to use these functions to detect lines in an image.

**1** Read an image into the MATLAB workspace.

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

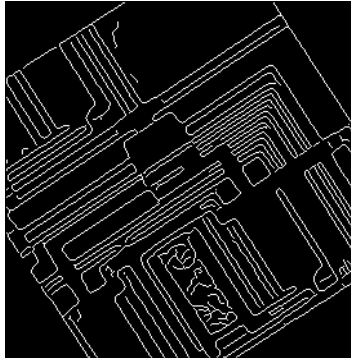
**2** For this example, rotate and crop the image.

```
rotI = imrotate(I,33,'crop');
```



- 3 Find the edges in the image.

```
BW = edge(rotI, 'canny');
```

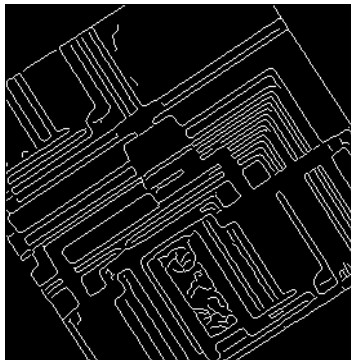


- 4 Compute the Hough transform of the image using the hough function.

```
[H, theta, rho] = hough(BW);
```

- 5 Display the transform.

```
imshow(H, [], 'XData', theta, 'YData', rho, ...  
       'InitialMagnification', 'fit');  
xlabel('\theta'), ylabel('\rho');  
axis on, axis normal, hold on;
```



- 6 Find the peaks in the Hough transform matrix, H, using the houghpeaks function.

```
P = houghpeaks(H,5,'threshold',ceil(0.3*max(H(:))));
```

**7** Plot the peaks.

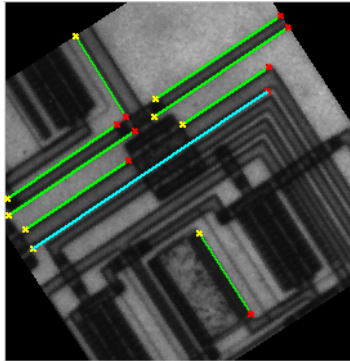
```
x = theta(P(:,2));  
y = rho(P(:,1));  
plot(x,y,'s','color','white');
```

**8** Find lines in the image.

```
lines = houghlines(BW,theta,rho,P,'FillGap',5,'MinLength',7);
```

**9** Create a plot that superimposes the lines on the original image.

```
figure, imshow(rotI), hold on  
max_len = 0;  
for k = 1:length(lines)  
    xy = [lines(k).point1; lines(k).point2];  
    plot(xy(:,1),xy(:,2),'LineWidth',2,'Color','green');  
  
    % Plot beginnings and ends of lines  
    plot(xy(1,1),xy(1,2),'x','LineWidth',2,'Color','yellow');  
    plot(xy(2,1),xy(2,2),'x','LineWidth',2,'Color','red');  
  
    % Determine the endpoints of the longest line segment  
    len = norm(lines(k).point1 - lines(k).point2);  
    if ( len > max_len)  
        max_len = len;  
        xy_long = xy;  
    end  
end  
  
% highlight the longest line segment  
plot(xy_long(:,1),xy_long(:,2),'LineWidth',2,'Color','cyan');
```



## Using 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 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 might have blocks of several different sizes.

### Example: Performing Quadtree Decomposition

To illustrate, this example performs quadtree decomposition on a 512-by-512 grayscale image. For an interactive demonstration of quadtree decomposition, run the demo `qtdemo`.

1 Read in the grayscale image.

```
I = imread('liftingbody.png');
```

- 2 Specify the test criteria used to determine the homogeneity of each block in the decomposition. For example, the criterion might be this threshold calculation.

```
max(block(:)) - min(block(:)) <= 0.2
```

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.

- 3 Perform this quadtree decomposition by calling the `qtdecomp` function, specifying the image and the threshold value as arguments.

```
S = qtdecomp(I,0.27)
```

You specify the threshold as a value between 0 and 1, regardless of the class of `I`. If `I` is `uint8`, `qtdecomp` multiplies the threshold value by 255 to determine the actual threshold to use. If `I` is `uint16`, `qtdecomp` multiplies the threshold value by 65535.

`qtdecomp` first divides the image into four 256-by-256 blocks and applies the test criterion to each block. If a block does not meet the criterion, `qtdecomp` subdivides it and applies the test criterion to each block. `qtdecomp` continues to subdivide blocks until all blocks meet the criterion. Blocks can be as small as 1-by-1, unless you specify otherwise.

`qtdecomp` returns `S` as a sparse matrix, the same size as `I`. The nonzero elements of `S` represent the upper left corners of the blocks; the value of each nonzero element indicates the block size.

The following figure shows the original image and a representation of its quadtree decomposition. (To see how this representation was created, see the example on the `qtdecomp` reference page.) 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.

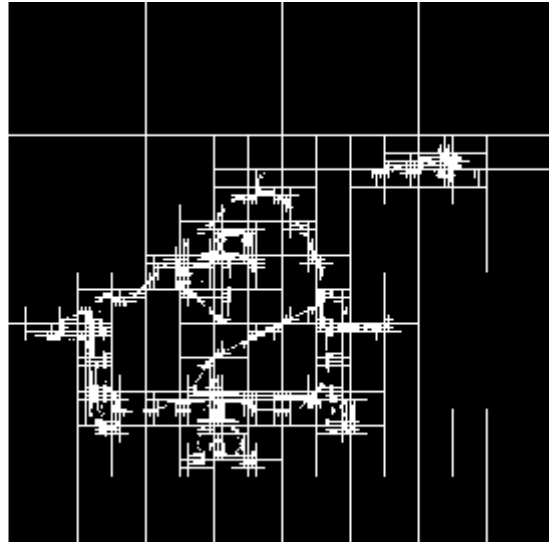
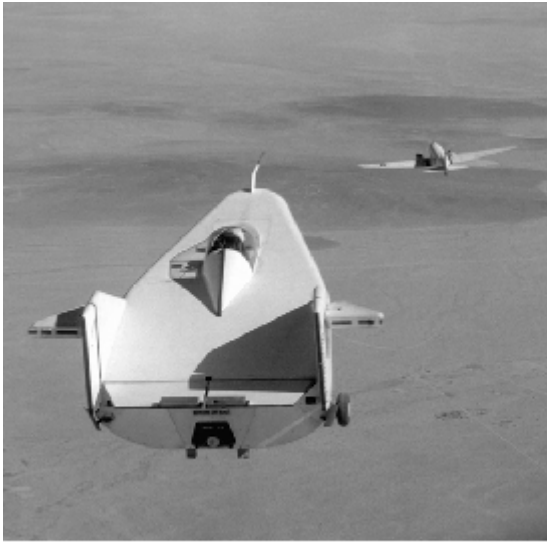


Image Courtesy of NASA

**Image and a Representation of Its Quadtree Decomposition**

## Analyzing the Texture of an Image

The toolbox supports a set of functions that you can use for texture analysis. Texture analysis refers to the characterization of regions in an image by their texture content. Texture analysis attempts to quantify intuitive qualities described by terms such as rough, smooth, silky, or bumpy as a function of the spatial variation in pixel intensities. In this sense, the roughness or bumpiness refers to variations in the intensity values, or gray levels.

Texture analysis is used in a variety of applications, including remote sensing, automated inspection, and medical image processing. Texture analysis can be used to find the texture boundaries, called texture segmentation. Texture analysis can be helpful when objects in an image are more characterized by their texture than by intensity, and traditional thresholding techniques cannot be used effectively.

The toolbox provides two types of texture functions:

- **Texture filter functions** — These functions use standard statistical measures to characterize the local texture of an image. See “Using Texture Filter Functions” on page 11-24 for more information.
- **Gray-level co-occurrence matrix** — These functions characterize the texture of an image by calculating how often pairs of pixel with specific values and in a specified spatial relationship occur in an image and then extracting statistical measures from this matrix. See “Using a Gray-Level Co-Occurrence Matrix (GLCM)” on page 11-28 for more information

### Using Texture Filter Functions

The toolbox includes three texture analysis functions that filter an image using standard statistical measures, such as range, standard deviation, and entropy. Entropy is a statistical measure of randomness. These statistics can characterize the texture of an image because they provide information about the local variability of the intensity values of pixels in an image.

For example, in areas with smooth texture, the range of values in the neighborhood around a pixel will be a small value; in areas of rough texture, the range will be larger. Similarly, calculating the standard deviation of pixels in a neighborhood can indicate the degree of variability of pixel values in that region.



The following sections provide additional information about the texture functions:

- “Understanding the Texture Filter Functions” on page 11-25
- “Example: Using the Texture Functions” on page 11-26

### Understanding the Texture Filter Functions

The three statistical texture filtering functions are

`rangefilt` -- Calculates the local range of an image

`stdfilt` -- Calculates the local standard deviation of an image

`entropyfilt` -- Calculates the local entropy of a grayscale image

The functions all operate in a similar way: they define a neighborhood around the pixel of interest and calculate the statistic for that neighborhood.

This example shows how the `rangefilt` function operates on a simple array.

```
A = [ 1 2 3 4 5; 6 7 8 9 10; 11 12 13 14 15; 16 17 18 19 20 ]
```

```
A =
```

```

     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
    16    17    18    19    20
```

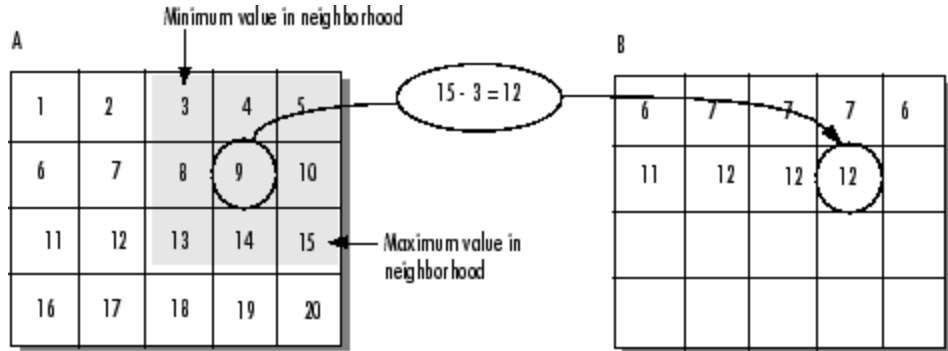
```
B = rangefilt(A)
```

```
B =
```

```

     6     7     7     7     6
    11    12    12    12    11
    11    12    12    12    11
     6     7     7     7     6
```

The following figure shows how the value of element B(2,4) was calculated from A(2,4). By default, the `rangefilt` function uses a 3-by-3 neighborhood but you can specify neighborhoods or different shapes and sizes.



### Determining Pixel Values in Range Filtered Output Image

The `stdfilt` and `entropyfilt` functions operate similarly, defining a neighborhood around the pixel of interest and calculating the statistic for the neighborhood to determine the pixel value in the output image. The `stdfilt` function calculates the standard deviation of all the values in the neighborhood.

The `entropyfilt` function calculates the entropy of the neighborhood and assigns that value to the output pixel. Note that, by default, the `entropyfilt` function defines a 9-by-9 neighborhood around the pixel of interest. To calculate the entropy of an entire image, use the `entropy` function.

### Example: Using the Texture Functions

The following example illustrates how the texture filter functions can detect regions of texture in an image. In the figure, the background is smooth; there is very little variation in the gray-level values. In the foreground, the surface contours of the coins exhibit more texture. In this image, foreground pixels have more variability and thus higher range values. Range filtering makes the edges and contours of the coins more visible.

To see an example of using filtering functions, view the [Texture Segmentation Using Texture Filters demo](#).

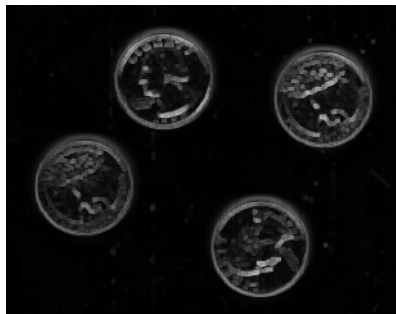
- 1 Read in the image and display it.

```
I = imread('eight.tif');  
imshow(I)
```



- 2 Filter the image with the `rangefilt` function and display the results. Note how range filtering highlights the edges and surface contours of the coins.

```
K = rangefilt(I);  
figure, imshow(K)
```



## Using a Gray-Level Co-Occurrence Matrix (GLCM)

The texture filter functions provide a statistical view of texture based on the image histogram. These functions can provide useful information about the texture of an image but cannot provide information about shape, i.e., the spatial relationships of pixels in an image.

Another statistical method that considers the spatial relationship of pixels is the gray-level co-occurrence matrix (GLCM), also known as the gray-level spatial dependence matrix. The toolbox provides functions to create a GLCM and derive statistical measurements from it.

This section includes the following topics.

- “Creating a Gray-Level Co-Occurrence Matrix” on page 11-28
- “Specifying the Offsets” on page 11-29
- “Deriving Statistics from a GLCM” on page 11-30
- “Example: Plotting the Correlation” on page 11-31

### Creating a Gray-Level Co-Occurrence Matrix

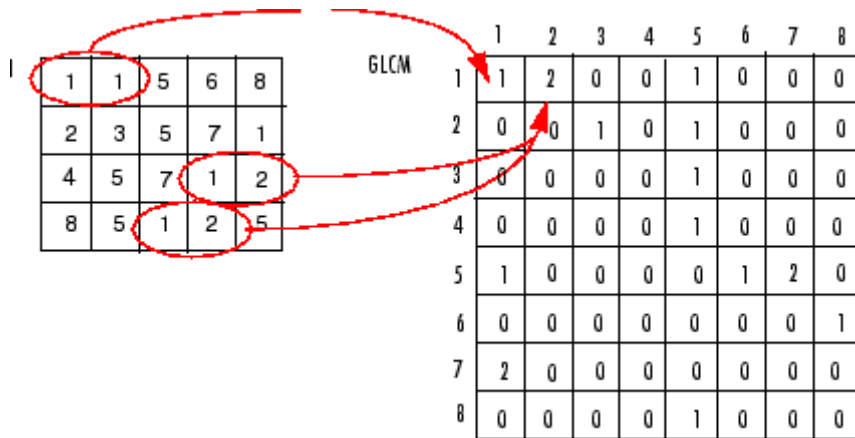
To create a GLCM, use the `graycomatrix` function. The `graycomatrix` function creates a gray-level co-occurrence matrix (GLCM) by calculating how often a pixel with the intensity (gray-level) value  $i$  occurs in a specific spatial relationship to a pixel with the value  $j$ . By default, the spatial relationship is defined as the pixel of interest and the pixel to its immediate right (horizontally adjacent), but you can specify other spatial relationships between the two pixels. Each element  $(i,j)$  in the resultant `glcm` is simply the sum of the number of times that the pixel with value  $i$  occurred in the specified spatial relationship to a pixel with value  $j$  in the input image.

The number of gray levels in the image determines the size of the GLCM. By default, `graycomatrix` uses scaling to reduce the number of intensity values in an image to eight, but you can use the `NumLevels` and the `GrayLimits` parameters to control this scaling of gray levels. See the `graycomatrix` reference page for more information.

The gray-level co-occurrence matrix can reveal certain properties about the spatial distribution of the gray levels in the texture image. For example, if

most of the entries in the GLCM are concentrated along the diagonal, the texture is coarse with respect to the specified offset. You can also derive several statistical measures from the GLCM. See “Deriving Statistics from a GLCM” on page 11-30 for more information.

To illustrate, the following figure shows how `graycomatrix` calculates the first three values in a GLCM. In the output GLCM, element (1,1) contains the value 1 because there is only one instance in the input image where two horizontally adjacent pixels have the values 1 and 1, respectively. `glcm(1,2)` contains the value 2 because there are two instances where two horizontally adjacent pixels have the values 1 and 2. Element (1,3) in the GLCM has the value 0 because there are no instances of two horizontally adjacent pixels with the values 1 and 3. `graycomatrix` continues processing the input image, scanning the image for other pixel pairs  $(i,j)$  and recording the sums in the corresponding elements of the GLCM.



### Process Used to Create the GLCM

### Specifying the Offsets

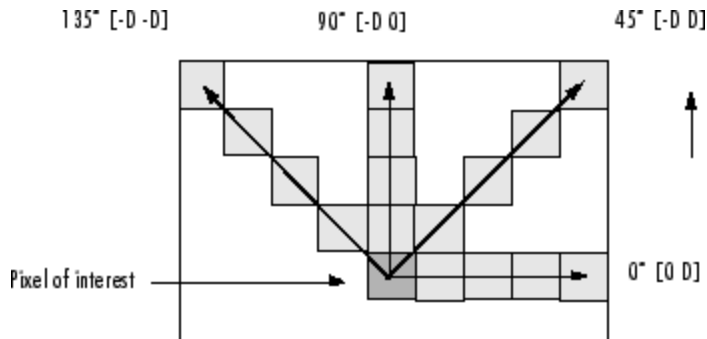
By default, the `graycomatrix` function creates a single GLCM, with the spatial relationship, or *offset*, defined as two horizontally adjacent pixels. However, a single GLCM might not be enough to describe the textural features of the input image. For example, a single horizontal offset might not be sensitive to texture with a vertical orientation. For this reason, `graycomatrix` can create multiple GLCMs for a single input image.

To create multiple GLCMs, specify an array of offsets to the `graycomatrix` function. These offsets define pixel relationships of varying direction and distance. For example, you can define an array of offsets that specify four directions (horizontal, vertical, and two diagonals) and four distances. In this case, the input image is represented by 16 GLCMs. When you calculate statistics from these GLCMs, you can take the average.

You specify these offsets as a  $p$ -by-2 array of integers. Each row in the array is a two-element vector, `[row_offset, col_offset]`, that specifies one offset. `row_offset` is the number of rows between the pixel of interest and its neighbor. `col_offset` is the number of columns between the pixel of interest and its neighbor. This example creates an offset that specifies four directions and 4 distances for each direction. For more information about specifying offsets, see the `graycomatrix` reference page.

```
offsets = [ 0 1; 0 2; 0 3; 0 4;...
           -1 1; -2 2; -3 3; -4 4;...
           -1 0; -2 0; -3 0; -4 0;...
           -1 -1; -2 -2; -3 -3; -4 -4];
```

The figure illustrates the spatial relationships of pixels that are defined by this array of offsets, where  $D$  represents the distance from the pixel of interest.



### Deriving Statistics from a GLCM

After you create the GLCMs, you can derive several statistics from them using the `graycoprops` function. These statistics provide information about the texture of an image. The following table lists the statistics you can derive. You specify the statistics you want when you call the `graycoprops` function.

For detailed information about these statistics, see the graycoprops reference page.

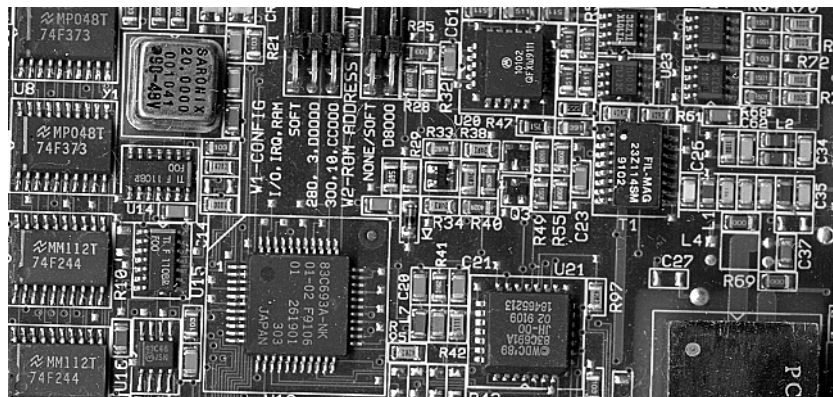
Statistic	Description
Contrast	Measures the local variations in the gray-level co-occurrence matrix.
Correlation	Measures the joint probability occurrence of the specified pixel pairs.
Energy	Provides the sum of squared elements in the GLCM. Also known as uniformity or the angular second moment.
Homogeneity	Measures the closeness of the distribution of elements in the GLCM to the GLCM diagonal.

### Example: Plotting the Correlation

This example shows how to create a set of GLCMs and derive statistics from them and illustrates how the statistics returned by graycoprops have a direct relationship to the original input image.

- 1 Read in a grayscale image and display it. The example converts the truecolor image to a grayscale image and then rotates it 90° for this example.

```
circuitBoard = rot90(rgb2gray(imread('board.tif')));
imshow(circuitBoard)
```



- 2 Define offsets of varying direction and distance. Because the image contains objects of a variety of shapes and sizes that are arranged in horizontal and vertical directions, the example specifies a set of horizontal offsets that only vary in distance.

```
offsets0 = [zeros(40,1) (1:40)'];
```

- 3 Create the GLCMs. Call the `graycomatrix` function specifying the offsets.

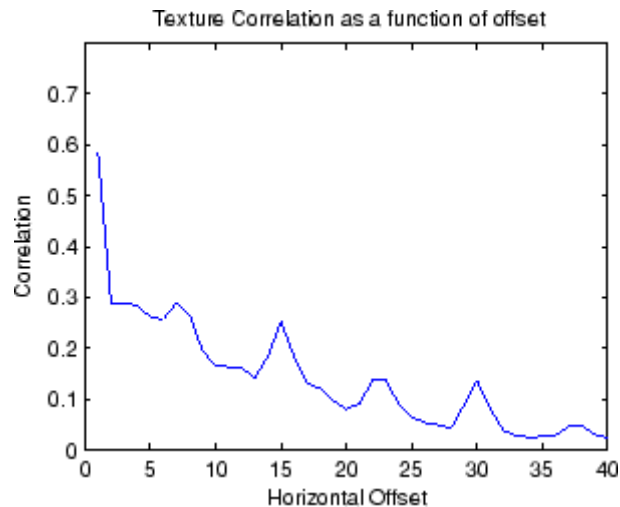
```
glcms = graycomatrix(circuitBoard,'Offset',offsets0)
```

- 4 Derive statistics from the GLCMs using the `graycoprops` function. The example calculates the contrast and correlation.

```
stats = graycoprops(glcms,'Contrast Correlation');
```

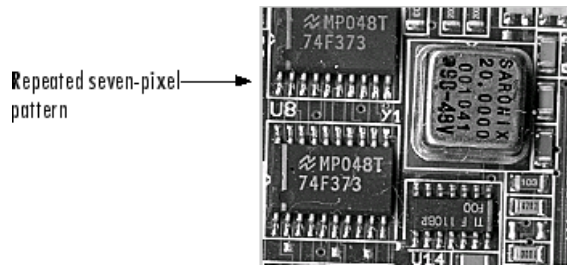
- 5 Plot correlation as a function of offset.

```
figure, plot([stats.Correlation]);  
title('Texture Correlation as a function of offset');  
xlabel('Horizontal Offset')  
ylabel('Correlation')
```





The plot contains peaks at offsets 7, 15, 23, and 30. If you examine the input image closely, you can see that certain vertical elements in the image have a periodic pattern that repeats every seven pixels. The following figure shows the upper left corner of the image and points out where this pattern occurs.

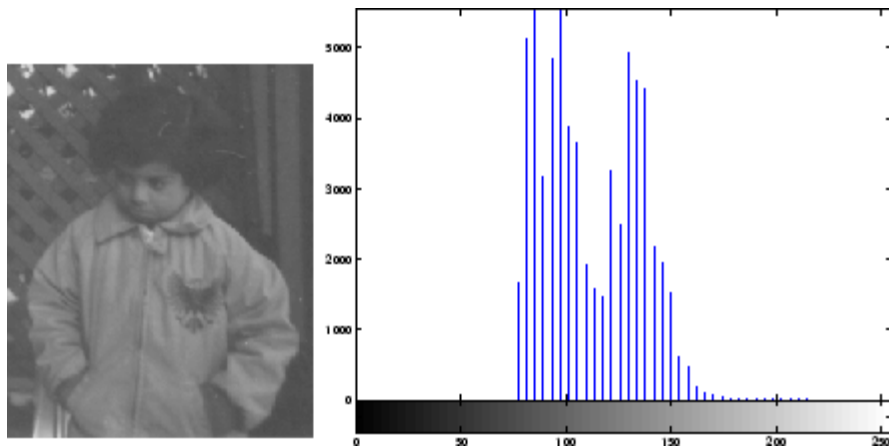


## Intensity Adjustment

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).

Intensity adjustment is an image enhancement technique that maps an image’s intensity values to a new range. To illustrate, this figure shows a low-contrast image with its histogram. Notice in the histogram of the image how all the values gather in the center of the range.

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



If you remap the data values to fill the entire intensity range [0, 255], you can increase the contrast of the image. The following sections describe several intensity adjustment techniques, including

- “Adjusting Intensity Values to a Specified Range” on page 11-35
- “Histogram Equalization” on page 11-39
- “Contrast-Limited Adaptive Histogram Equalization” on page 11-41
- “Decorrelation Stretching” on page 11-42

The functions described in this section apply primarily to grayscale 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.

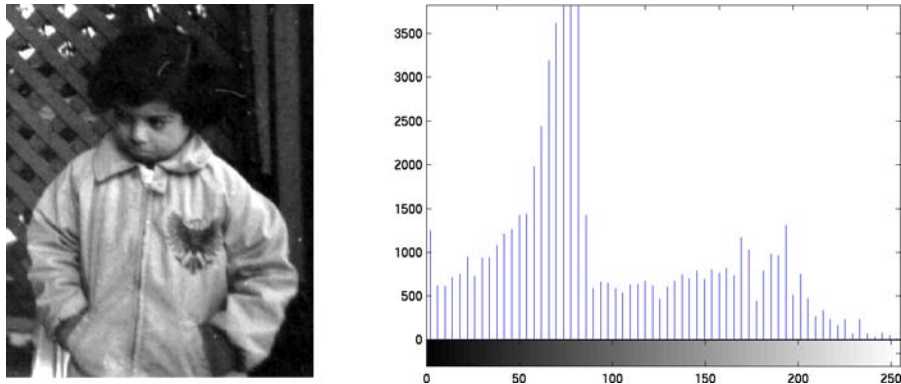
## Adjusting Intensity Values to a Specified Range

You can adjust the intensity values in an image using the `imadjust` function, where you specify the range of intensity values in the output image.

For example, this code increases the contrast in a low-contrast grayscale image by remapping the data values to fill the entire intensity range [0, 255].

```
I = imread('pout.tif');  
J = imadjust(I);  
imshow(J)  
figure, imhist(J,64)
```

This figure displays the adjusted image and its histogram. Notice the increased contrast in the image, and that the histogram now fills the entire range.



**Adjusted Image and Its Histogram**

## Specifying the Adjustment Limits

You can optionally specify the range of the input values and the output values using `imadjust`. You specify these ranges in two vectors that you pass to

`imadjust` as arguments. The first vector specifies the low- and high-intensity values that you want to map. The second vector specifies the scale over which you want to map them.

---

**Note** Note that you must specify the intensities 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. To learn about an alternative way to set these limits automatically, see “Setting the Adjustment Limits Automatically” on page 11-37.

---

For example, you can decrease the contrast of an image by narrowing the range of the data. In the example below, the man’s coat is too dark to reveal any detail. `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. Note, however, that because all values above 51 in the original image are mapped to 255 (white) in the adjusted image, the adjusted image appears washed out.

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



Image Courtesy of MIT

### Image After Remapping and Widening the Dynamic Range

## Setting the Adjustment Limits Automatically

To use `imadjust`, you must typically perform two steps:

- 1 View the histogram of the image to determine the intensity value limits.
- 2 Specify these limits as a fraction between 0.0 and 1.0 so that you can pass them to `imadjust` in the `[low_in high_in]` vector.

For a more convenient way to specify these limits, use the `stretchlim` function. (The `imadjust` function uses `stretchlim` for its simplest syntax, `imadjust(I)`.)

This function calculates the histogram of the image and determines the adjustment limits automatically. The `stretchlim` function returns these values as fractions in a vector that you can pass as the `[low_in high_in]` argument to `imadjust`; for example:

```
I = imread('rice.png');  
J = imadjust(I,stretchlim(I),[0 1]);
```

By default, `stretchlim` uses the intensity values that represent the bottom 1% (0.01) and the top 1% (0.99) of the range as the adjustment limits. By trimming the extremes at both ends of the intensity range, `stretchlim` makes more room in the adjusted dynamic range for the remaining intensities. But you can specify other range limits as an argument to `stretchlim`. See the `stretchlim` reference page for more information.

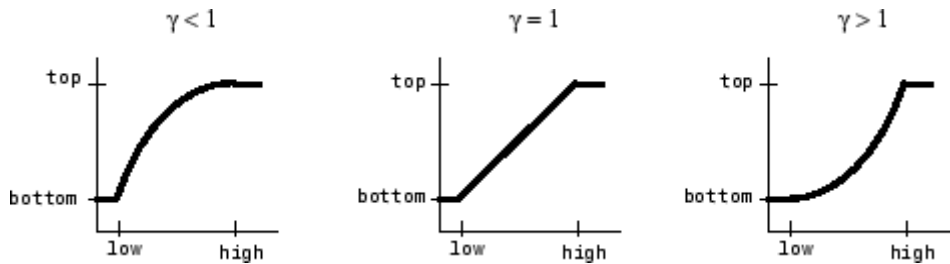
## Gamma Correction

`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 that specifies the *gamma correction* factor. Depending on the value of gamma, the mapping between values in the input and output images might be nonlinear. For example, the value halfway between low and high might 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.)



### 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)
```



Image Courtesy of Susan Cohen

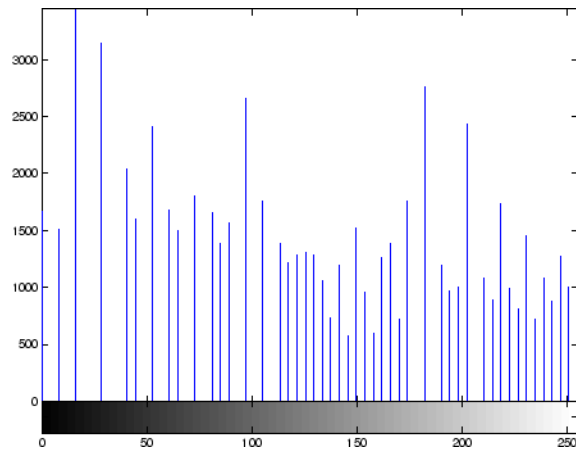
### Image Before and After Applying Gamma Correction

## 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 a grayscale 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(J)  
figure, imhist(J,64)
```

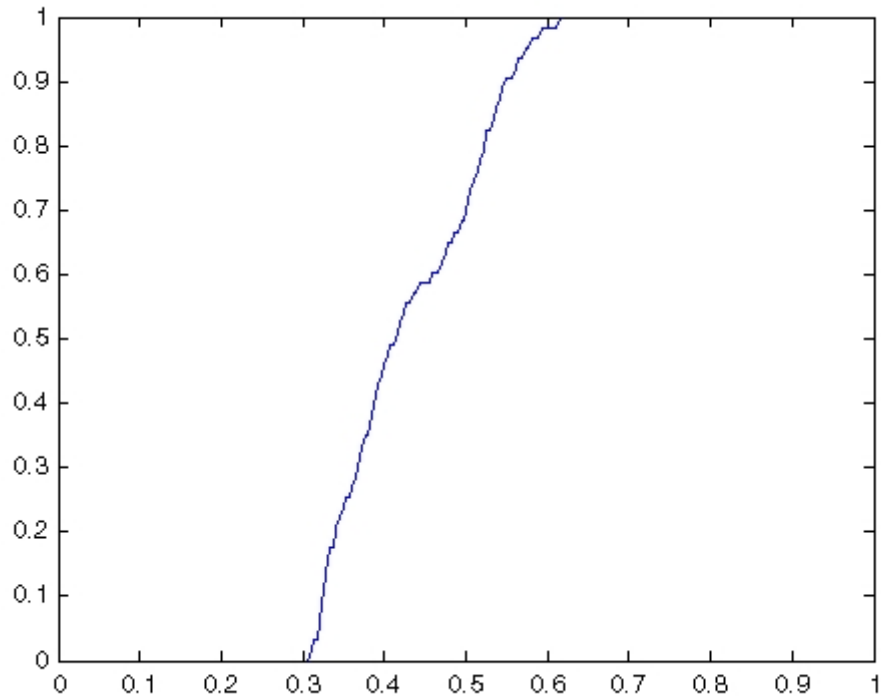


**Image After Histogram Equalization with Its Histogram**

`histeq` can return a 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 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`.

## Contrast-Limited Adaptive Histogram Equalization

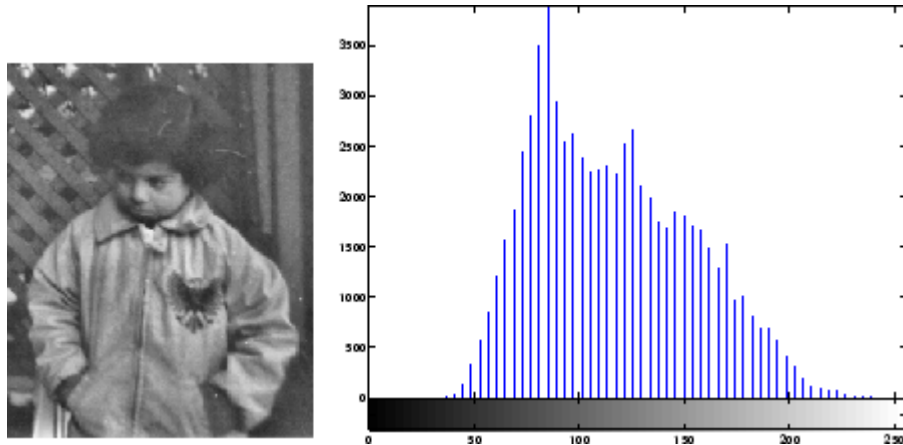
As an alternative to using `histeq`, you can perform contrast-limited adaptive histogram equalization (CLAHE) using the `adapthisteq` function. While `histeq` works on the entire image, `adapthisteq` operates on small regions in the image, called *tiles*. Each tile's contrast is enhanced, so that the histogram of the output region approximately matches a specified histogram. After

performing the equalization, `adapthisteq` combines neighboring tiles using bilinear interpolation to eliminate artificially induced boundaries.

To avoid amplifying any noise that might be present in the image, you can use `adapthisteq` optional parameters to limit the contrast, especially in homogeneous areas.

To illustrate, this example uses `adapthisteq` to adjust the contrast in a grayscale image. The original image has low contrast, with most values in the middle of the intensity range. `adapthisteq` produces an output image having values evenly distributed throughout the range.

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



**Image After CLAHE Equalization with Its Histogram**

## Decorrelation Stretching

Decorrelation stretching enhances the color separation of an image with significant band-band correlation. The exaggerated colors improve visual interpretation and make feature discrimination easier. You apply decorrelation stretching with the `decorrstretch` function. See “Adding a

Linear Contrast Stretch” on page 11-45 on how to add an optional linear contrast stretch to the decorrelation stretch.

The number of color bands, NBANDS, in the image is usually three. But you can apply decorrelation stretching regardless of the number of color bands.

The original color values of the image are mapped to a new set of color values with a wider range. The color intensities of each pixel are transformed into the color eigenspace of the NBANDS-by-NBANDS covariance or correlation matrix, stretched to equalize the band variances, then transformed back to the original color bands.

To define the bandwise statistics, you can use the entire original image or, with the subset option, any selected subset of it. See the decorrstretch reference page.

### Simple Decorrelation Stretching

You can apply decorrelation and stretching operations on the library of images available in the imdemos directory. The library includes a LANDSAT image of the Little Colorado River. In this example, you perform a simple decorrelation stretch on this image:

- 1 The image has seven bands, but just read in the three visible colors:

```
A = multibandread('littlecoriver.lan', [512, 512, 7], ...
    'uint8=>uint8', 128, 'bil', 'ieee-le', ...
    {'Band','Direct',[3 2 1]});
```

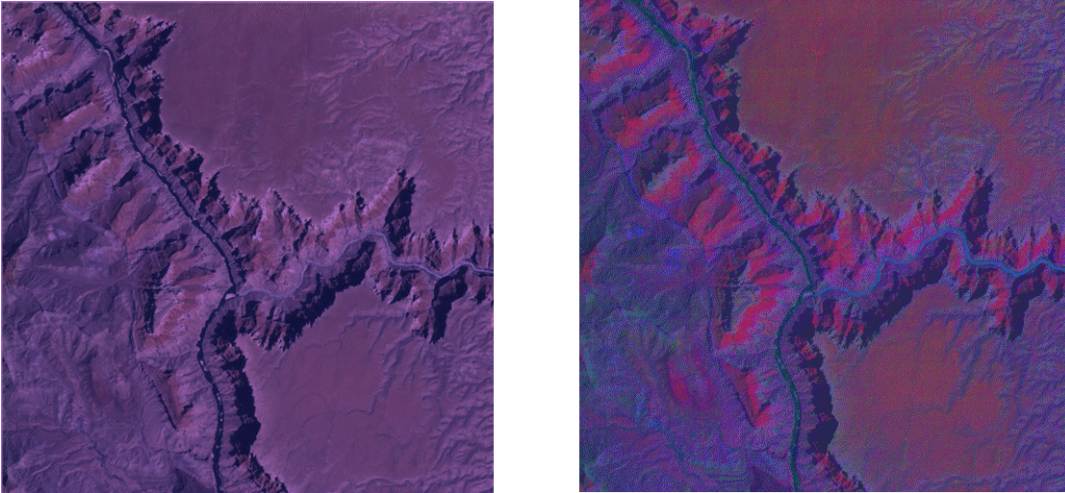
- 2 Then perform the decorrelation stretch:

```
B = decorrstretch(A);
```

- 3 Now view the results:

```
imshow(A); figure; imshow(B)
```

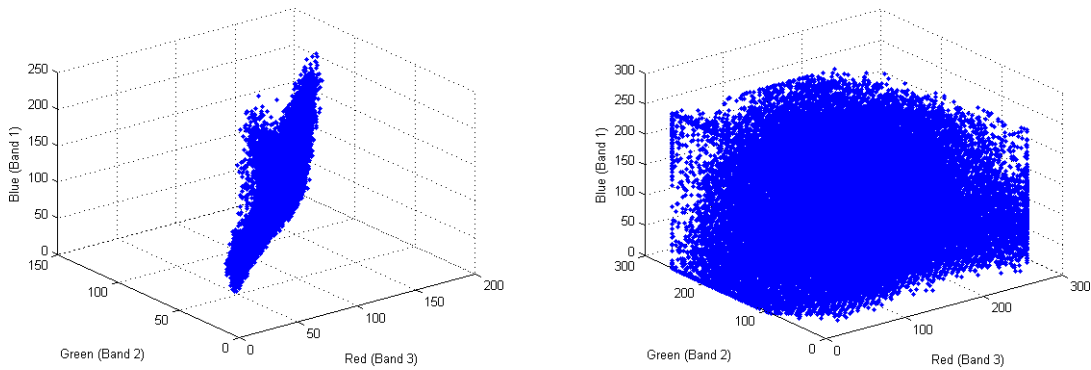
Compare the two images. The original has a strong violet (red-bluish) tint, while the transformed image has a somewhat expanded color range.



**Little Colorado River Before (left) and After (right) Decorrelation Stretch**

A color band scatterplot of the images shows how the bands are decorrelated and equalized:

```
rA = A(:,:,1); gA = A(:,:,2); bA = A(:,:,3);
figure, plot3(rA(:),gA(:),bA(:),'.'); grid('on')
xlabel('Red (Band 3)'); ylabel('Green (Band 2)'); ...
zlabel('Blue (Band 1)')
rB = B(:,:,1); gB = B(:,:,2); bB = B(:,:,3);
figure, plot3(rB(:),gB(:),bB(:),'.'); grid('on')
xlabel('Red (Band 3)'); ylabel('Green (Band 2)'); ...
zlabel('Blue (Band 1)')
```



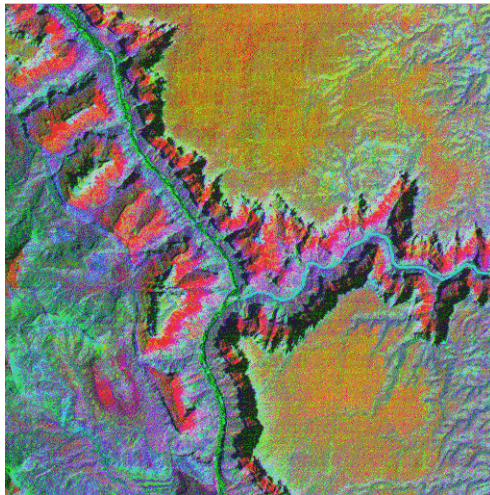
**Color Scatterplot Before (left) and After (right) Decorrelation Stretch**

### **Adding a Linear Contrast Stretch**

Now try the same transformation, but with a linear contrast stretch applied after the decorrelation stretch:

```
imshow(A); C = decorrstretch(A,'Tol',0.01); figure; imshow(C)
```

Compare the transformed image to the original.



**Little Colorado River After Decorrelation Stretch Followed by Linear Contrast Stretch**

Adding the linear contrast stretch enhances the resulting image by further expanding the color range. In this case, the transformed color range is mapped within each band to a normalized interval between 0.01 and 0.99, saturating 2%.

See the `stretchlim` function reference page for more about `Tol`. Without the `Tol` option, `decorrstretch` applies no linear contrast stretch.

---

**Note** You can apply a linear contrast stretch as a separate operation after performing a decorrelation stretch, using `stretchlim` and `imadjust`. This alternative, however, often gives inferior results for `uint8` and `uint16` images, because the pixel values must be clamped to `[0 255]` (or `[0 65535]`). The `Tol` option in `decorrstretch` circumvents this limitation.

---

## Noise Removal

Digital images are prone to a variety of types of noise. Noise is the result of errors in the image acquisition process that result in pixel values that do not reflect the true intensities of the real scene. 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

- “Using Linear Filtering” on page 11-47
- “Using Median Filtering” on page 11-48
- “Using Adaptive Filtering” on page 11-50

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.

### Using 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 8-2 for more information.

## Using 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.

---

**Note** 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.

---

The following example 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.

1 Read in the image and display it.

```
I = imread('eight.tif');  
imshow(I)
```



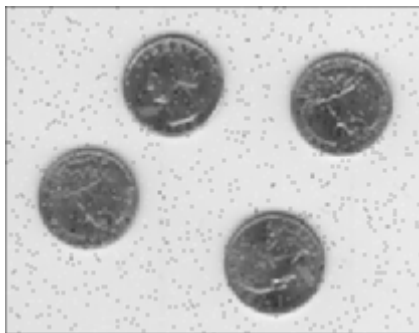


**2** Add noise to it.

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

**3** Filter the noisy image with an averaging filter and display the results.

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



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

```
L = medfilt2(J,[3 3]);  
figure, imshow(K)  
figure, imshow(L)
```



## Using Adaptive Filtering

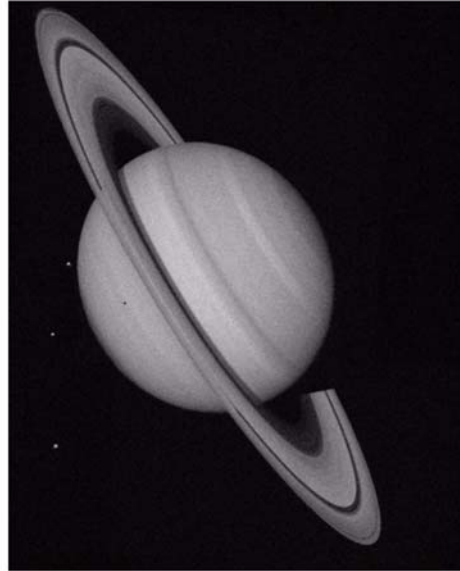
The `wiener2` function applies a Wiener filter (a type of linear filter) to an image *adaptively*, tailoring itself to the local image variance. Where the variance is large, `wiener2` performs little smoothing. Where the variance is small, `wiener2` performs more smoothing.

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

```
RGB = imread('saturn.png');  
I = rgb2gray(RGB);  
J = imnoise(I,'gaussian',0,0.005);
```

```
K = wiener2(J,[5 5]);  
imshow(J)  
figure, imshow(K)
```



Original Image Courtesy of NASA

**Noisy Version (left) and Filtered Version (right)**



# Region-Based Processing

---

This chapter describes operations that you can perform on a selected region of an image.

Specifying a Region of Interest  
(p. 12-2)

Describes how to specify a region of interest using the `roipoly` function

Filtering a Region (p. 12-5)

Describes how to apply a filter to a region using the `roifilt2` function

Filling a Region (p. 12-8)

Describes how to fill a region of interest using the `roifill` function

## 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.

You can define more than one region 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.

The following subsections discuss methods for creating binary masks:

- “Selecting a Polygon” on page 12-2
- “Other Selection Methods” on page 12-4 (using any binary mask or the `roicolor` function)

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

### 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 crosshairs when it is over the image displayed in the current axes. You can then specify the vertices of the polygon by clicking 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.

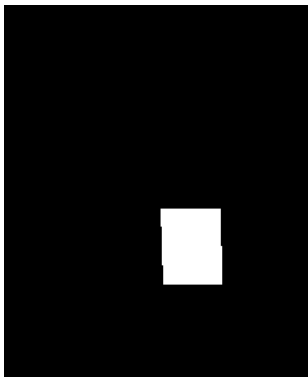
The example below illustrates using the interactive syntax of `roipoly` to create a binary mask. In the figure, the border of the selected region that was created using a mouse is shown in red.

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



**Polygonal Region of Interest Selected Using `roipoly`**

```
imshow(BW)
```



**Binary Mask Created for the Region Shown in the Preceding Figure**

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 grayscale 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 `poly2mask` function to create a binary mask. Unlike the `roipoly` function, `poly2mask` does not require an input image. For more information, see the `poly2mask` reference page.

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

Filtering a region is the process of applying a filter to a region of interest in an image, where a binary mask defines the region. For example, you can apply an intensity adjustment filter to certain regions of an image.

To filter a region in an image, use the `roifilt2` function. When you call `roifilt2`, you specify a grayscale 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*.

---

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

---

### Example: Filtering a Region in an Image

This example uses masked filtering to increase the contrast of a specific region of an image:

- 1 Read in the image.

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

- 2 Create the mask.

This example uses the mask `BW` created in “Selecting a Polygon” on page 12-2. The region of interest specified by the mask is the logo on the girl's jacket.

- 3 Create the filter.

```
h = fspecial('unsharp');
```

- 4 Call `roifilt2`, specifying the image to be filtered, the mask, and the filter.

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



**Image Before and After Using an Unsharp Filter on the Region of Interest**

## Specifying the Filtering Operation

`roifilt2` also enables you to specify your own function to operate on the region of interest. This example uses the `imadjust` function to lighten parts of an image:

- 1 Read in the image.

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

- 2 Create the mask. In this example, the mask is a binary image containing text. The mask image must be cropped to be the same size as the image to be filtered.

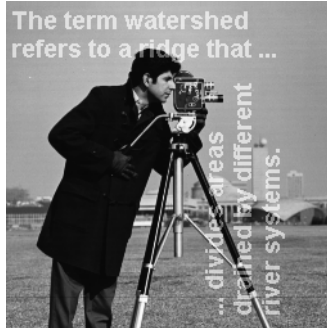
```
BW = imread('text.png');  
mask = BW(1:256,1:256);
```

- 3 Create the filter.

```
f = @(x) imadjust(x,[],[],0.3);
```

- 4 Call `roifilt2`, specifying the image to be filtered, the mask, and the filter. The resulting image, `I2`, has the text imprinted on it.

```
I2 = roifilt2(I,mask,f);  
imshow(I2)
```



**Image Brightened Using a Binary Mask Containing Text**

## Filling a Region

Filling is 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.

To fill a region of interest, you can use the `roifill` function. 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.

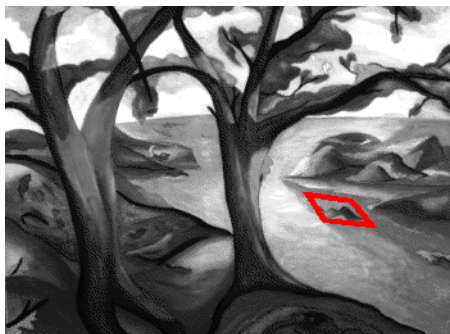
This example shows how to use `roifill` to fill a region in an image.

- 1 Read an image into the MATLAB workspace and display it. Because the image is an indexed image, the example uses `ind2gray` to convert it to a grayscale image.

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

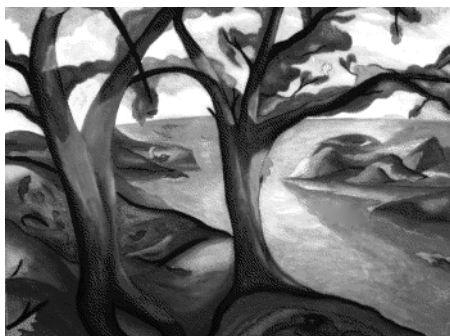
- 2 Call `roifill` and then use the mouse to select the region you want to fill (shown in red in the following figure). `roifill` returns the modified image in `I2`.

```
I2 = roifill;
```



- 3** Display the modified image. Note how the region of interest defined in the previous step has been filled.

```
imshow(I2)
```





# Image Deblurring

---

This chapter describes how to deblur an image using the toolbox deblurring functions.

Understanding Deblurring (p. 13-2)	Defines deblurring and deconvolution
Deblurring with the Wiener Filter (p. 13-6)	Using the <code>deconvwnr</code> function
Deblurring with a Regularized Filter (p. 13-8)	Using the <code>deconvreg</code> function
Deblurring with the Lucy-Richardson Algorithm (p. 13-10)	Using the <code>deconvlucy</code> function
Deblurring with the Blind Deconvolution Algorithm (p. 13-16)	Using the <code>deconvblind</code> function
Creating Your Own Deblurring Functions (p. 13-23)	Using the <code>otf2psf</code> and <code>psf2otf</code> functions
Avoiding Ringing in Deblurred Images (p. 13-24)	Using the <code>edgetaper</code> function to avoid "ringing" in deblurred images

## Understanding Deblurring

This section provides some background on deblurring techniques. The section includes these topics:

- “Causes of Blurring” on page 13-2
- “Deblurring Model” on page 13-2
- “Deblurring Functions” on page 13-4

### Causes of Blurring

The blurring, or degradation, of an image can be caused by many factors:

- Movement during the image capture process, by the camera or, when long exposure times are used, by the subject
- Out-of-focus optics, use of a wide-angle lens, atmospheric turbulence, or a short exposure time, which reduces the number of photons captured
- Scattered light distortion in confocal microscopy

### Deblurring Model

A blurred or degraded image can be approximately described by this equation  $\mathbf{g} = \mathbf{H}\mathbf{f} + \mathbf{n}$ , where

- g** The blurred image
- H** The distortion operator, also called the *point spread function* (PSF). In the spatial domain, the PSF describes the degree to which an optical system blurs (spreads) a point of light. The PSF is the inverse Fourier transform of the optical transfer function (OTF). In the frequency domain, the OTF describes the response of a linear, position-invariant system to an impulse. The OTF is the Fourier transform of the point spread function (PSF). The distortion operator, when convolved with the image, creates the distortion. Distortion caused by a point spread function is just one type of distortion.
- f** The original true image
- n** Additive noise, introduced during image acquisition, that corrupts the image



---

**Note** The image **f** really doesn't exist. This image represents what you would have if you had perfect image acquisition conditions.

---

### Importance of the PSF

Based on this model, the fundamental task of deblurring is to deconvolve the blurred image with the PSF that exactly describes the distortion. Deconvolution is the process of reversing the effect of convolution.

---

**Note** The quality of the deblurred image is mainly determined by knowledge of the PSF.

---

To illustrate, this example takes a clear image and deliberately blurs it by convolving it with a PSF. The example uses the `fspecial` function to create a PSF that simulates a motion blur, specifying the length of the blur in pixels, (`LEN=31`), and the angle of the blur in degrees (`THETA=11`). Once the PSF is created, the example uses the `imfilter` function to convolve the PSF with the original image, `I`, to create the blurred image, `Blurred`. (To see how deblurring is the reverse of this process, using the same images, see “Deblurring with the Wiener Filter” on page 13-6.)

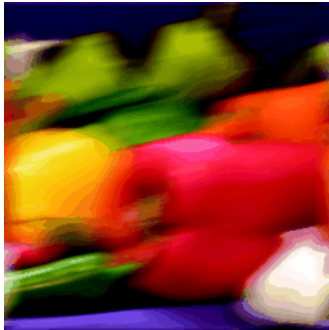
```
I = imread('peppers.png');  
I = I(60+[1:256],222+[1:256],:); % crop the image  
figure; imshow(I); title('Original Image');
```



```

LEN = 31;
THETA = 11;
PSF = fspecial('motion',LEN,THETA); % create PSF
Blurred = imfilter(I,PSF,'circular','conv');
figure; imshow(Blurred); title('Blurred Image');

```



## Deblurring Functions

The toolbox includes four deblurring functions, listed here in order of complexity:

deconvwnr	Implements deblurring using the Wiener filter
deconvreg	Implements deblurring using a regularized filter
deconvlucy	Implements deblurring using the Lucy-Richardson algorithm
deconvblind	Implements deblurring using the blind deconvolution algorithm

All the functions accept a PSF and the blurred image as their primary arguments. The `deconvwnr` function implements a least squares solution. The `deconvreg` function implements a constrained least squares solution, where you can place constraints on the output image (the smoothness requirement is the default). With either of these functions, you should provide some information about the noise to reduce possible noise amplification during deblurring.

The `deconvlucy` function implements an accelerated, damped Lucy-Richardson algorithm. This function performs multiple iterations, using optimization techniques and Poisson statistics. With this function, you do not need to provide information about the additive noise in the corrupted image.

The `deconvblind` function implements the blind deconvolution algorithm, which performs deblurring without knowledge of the PSF. When you call `deconvblind`, you pass as an argument your initial guess at the PSF. The `deconvblind` function returns a restored PSF in addition to the restored image. The implementation uses the same damping and iterative model as the `deconvlucy` function.

---

**Note** You might need to perform many iterations of the deblurring process, varying the parameters you specify to the deblurring functions with each iteration, until you achieve an image that, based on the limits of your information, is the best approximation of the original scene. Along the way, you must make numerous judgments about whether newly uncovered features in the image are features of the original scene or simply artifacts of the deblurring process.

---

For information about creating your own deblurring functions, see “Creating Your Own Deblurring Functions” on page 13-23. To avoid “ringing” in a deblurred image, you can use the `edgetaper` function to preprocess your image before passing it to the deblurring functions. See “Avoiding Ringing in Deblurred Images” on page 13-24 for more information.

## Deblurring with the Wiener Filter

Use the `deconvwnr` function to deblur an image using the Wiener filter. Wiener deconvolution can be used effectively when the frequency characteristics of the image and additive noise are known, to at least some degree. In the absence of noise, the Wiener filter reduces to the ideal inverse filter.

This example deblurs the blurred image created in “Deblurring Model” on page 13-2, specifying the same PSF function that was used to create the blur. This example illustrates the importance of knowing the PSF, the function that caused the blur. When you know the exact PSF, the results of deblurring can be quite effective.

- 1 Read an image into the MATLAB workspace. (To speed the deblurring operation, the example also crops the image.)

```
I = imread('peppers.png');  
I = I(10+[1:256],222+[1:256],:);  
figure;imshow(I);title('Original Image');
```



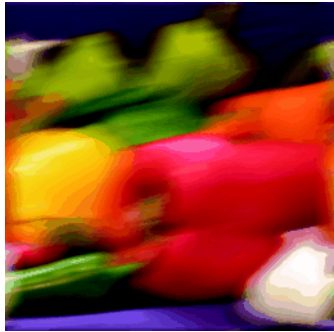
- 2 Create a PSF.

```
LEN = 31;  
THETA = 11;  
PSF = fspecial('motion',LEN,THETA);
```

- 3 Create a simulated blur in the image.

```
Blurred = imfilter(I,PSF,'circular','conv');
```

```
figure; imshow(Blurred);title('Blurred Image');
```



#### 4 Deblur the image.

```
wnr1 = deconvwnr(Blurred,PSF);  
figure;imshow(wnr1);  
title('Restored, True PSF');
```



## Refining the Result

You can affect the deconvolution results by providing values for the optional arguments supported by the `deconvwnr` function. Using these arguments you can specify the noise-to-signal power value and/or provide autocorrelation functions to help refine the result of deblurring. To see the impact of these optional arguments, view the Image Processing Toolbox deblurring demos.

## Deblurring with a Regularized Filter

Use the `deconvreg` function to deblur an image using a regularized filter. A regularized filter can be used effectively when limited information is known about the additive noise.

To illustrate, this example simulates a blurred image by convolving a Gaussian filter PSF with an image (using `imfilter`). Additive noise in the image is simulated by adding Gaussian noise of variance  $V$  to the blurred image (using `imnoise`):

- 1 Read an image into the MATLAB workspace. The example uses cropping to reduce the size of the image to be deblurred. This is not a required step in deblurring operations.

```
I = imread('tissue.png');  
I = I(125+[1:256],1:256,:);  
figure; imshow(I); title('Original Image');
```

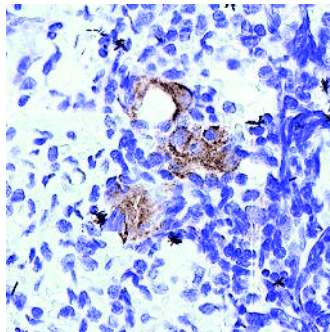


Image Courtesy Alan W. Partin

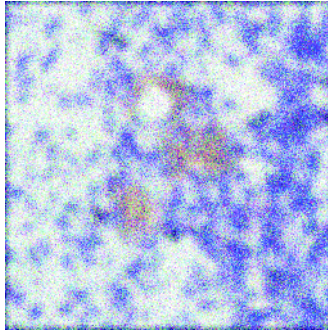
- 2 Create the PSF.

```
PSF = fspecial('gaussian',11,5);
```

- 3 Create a simulated blur in the image and add noise.

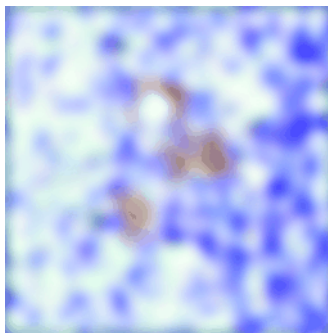
```
Blurred = imfilter(I,PSF,'conv');  
  
V = .02;
```

```
BlurredNoisy = imnoise(Blurred,'gaussian',0,V);  
figure;imshow(BlurredNoisy);title('Blurred and Noisy Image');
```



- 4 Use `deconvreg` to deblur the image, specifying the PSF used to create the blur and the noise power, NP.

```
NP = V*prod(size(I));  
[reg1 LAGRA] = deconvreg(BlurredNoisy,PSF,NP);  
figure,imshow(reg1),title('Restored Image');
```



## Refining the Result

You can affect the deconvolution results by providing values for the optional arguments supported by the `deconvreg` function. Using these arguments you can specify the noise power value, the range over which `deconvreg` should iterate as it converges on the optimal solution, and the regularization operator to constrain the deconvolution. To see the impact of these optional arguments, view the Image Processing Toolbox deblurring demos.

## Deblurring with the Lucy-Richardson Algorithm

Use the `deconvlucy` function to deblur an image using the accelerated, damped, Lucy-Richardson algorithm. The algorithm maximizes the likelihood that the resulting image, when convolved with the PSF, is an instance of the blurred image, assuming Poisson noise statistics. This function can be effective when you know the PSF but know little about the additive noise in the image.

The `deconvlucy` function implements several adaptations to the original Lucy-Richardson maximum likelihood algorithm that address complex image restoration tasks. Using these adaptations, you can

- Reduce the effect of noise amplification on image restoration
- Account for nonuniform image quality (e.g., bad pixels, flat-field variation)
- Handle camera read-out and background noise
- Improve the restored image resolution by subsampling

The following sections provide more information about each of these adaptations.

### Reducing the Effect of Noise Amplification

*Noise amplification* is a common problem of maximum likelihood methods that attempt to fit data as closely as possible. After many iterations, the restored image can have a speckled appearance, especially for a smooth object observed at low signal-to-noise ratios. These speckles do not represent any real structure in the image, but are artifacts of fitting the noise in the image too closely.

To control noise amplification, the `deconvlucy` function uses a damping parameter, `DAMPAR`. This parameter specifies the threshold level for the deviation of the resulting image from the original image, below which damping occurs. For pixels that deviate in the vicinity of their original values, iterations are suppressed.

Damping is also used to reduce *ringing*, the appearance of high-frequency structures in a restored image. Ringing is not necessarily the result of noise



amplification. See “Avoiding Ringing in Deblurred Images” on page 13-24 for more information.

## Accounting for Nonuniform Image Quality

Another complication of real-life image restoration is that the data might include bad pixels, or that the quality of the receiving pixels might vary with time and position. By specifying the WEIGHT array parameter with the `deconvlucy` function, you can specify that certain pixels in the image be ignored. To ignore a pixel, assign a weight of zero to the element in the WEIGHT array that corresponds to the pixel in the image.

The algorithm converges on predicted values for the bad pixels based on the information from neighborhood pixels. The variation in the detector response from pixel to pixel (the so-called flat-field correction) can also be accommodated by the WEIGHT array. Instead of assigning a weight of 1.0 to the good pixels, you can specify fractional values and weight the pixels according to the amount of the flat-field correction.

## Handling Camera Read-Out Noise

Noise in charge coupled device (CCD) detectors has two primary components:

- Photon counting noise with a Poisson distribution
- Read-out noise with a Gaussian distribution

The Lucy-Richardson iterations intrinsically account for the first type of noise. You must account for the second type of noise; otherwise, it can cause pixels with low levels of incident photons to have negative values.

The `deconvlucy` function uses the READOUT input parameter to handle camera read-out noise. The value of this parameter is typically the sum of the read-out noise variance and the background noise (e.g., number of counts from the background radiation). The value of the READOUT parameter specifies an offset that ensures that all values are positive.

## Handling Undersampled Images

The restoration of undersampled data can be improved significantly if it is done on a finer grid. The `deconvlucy` function uses the `SUBSMPL` parameter to specify the subsampling rate, if the PSF is known to have a higher resolution.

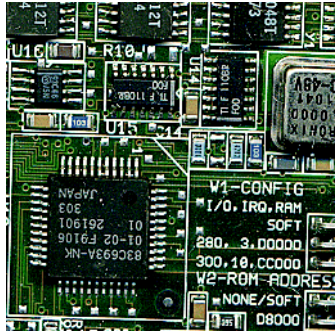
If the undersampled data is the result of camera pixel binning during image acquisition, the PSF observed at each pixel rate can serve as a finer grid PSF. Otherwise, the PSF can be obtained via observations taken at subpixel offsets or via optical modeling techniques. This method is especially effective for images of stars (high signal-to-noise ratio), because the stars are effectively forced to be in the center of a pixel. If a star is centered between pixels, it is restored as a combination of the neighboring pixels. A finer grid redirects the consequent spreading of the star flux back to the center of the star's image.

### Example: Using the `deconvlucy` Function to Deblur an Image

To illustrate a simple use of `deconvlucy`, this example simulates a blurred, noisy image by convolving a Gaussian filter PSF with an image (using `imfilter`) and then adding Gaussian noise of variance  $V$  to the blurred image (using `imnoise`):

- 1 Read an image into the MATLAB workspace. (The example uses cropping to reduce the size of the image to be deblurred. This is not a required step in deblurring operations.)

```
I = imread('board.tif');  
I = I(50+[1:256],2+[1:256],:);  
figure;imshow(I);title('Original Image');
```



**2** Create the PSF.

```
PSF = fspecial('gaussian',5,5);
```

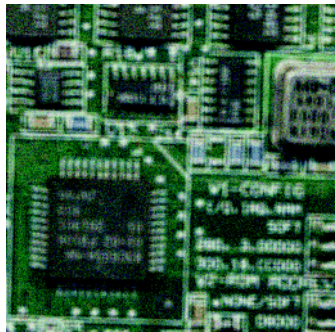
**3** Create a simulated blur in the image and add noise.

```
Blurred = imfilter(I,PSF,'symmetric','conv');
```

```
V = .002;
```

```
BlurredNoisy = imnoise(Blurred,'gaussian',0,V);
```

```
figure;imshow(BlurredNoisy);title('Blurred and Noisy Image');
```



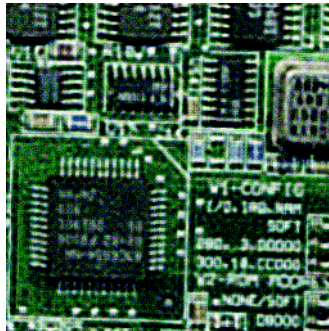
- 4 Use `deconvlucy` to restore the blurred and noisy image, specifying the PSF used to create the blur, and limiting the number of iterations to 5 (the default is 10).

---

**Note** The `deconvlucy` function can return values in the output image that are beyond the range of the input image.

---

```
luc1 = deconvlucy(BlurredNoisy,PSF,5);  
figure; imshow(luc1);  
title('Restored Image');
```



## Refining the Result

The `deconvlucy` function, by default, performs multiple iterations of the deblurring process. You can stop the processing after a certain number of iterations to check the result, and then restart the iterations from the point where processing stopped. To do this, pass in the input image as a cell array, for example, `{BlurredNoisy}`. The `deconvlucy` function returns the output image as a cell array that you can then pass as an input argument to `deconvlucy` to restart the deconvolution.

The output cell array contains these four elements:

Element	Description
<code>output{1}</code>	Original input image
<code>output{2}</code>	Image produced by the last iteration
<code>output{3}</code>	Image produced by the next to last iteration
<code>output{4}</code>	Internal information used by <code>deconvlucy</code> to know where to restart the process

The `deconvlucy` function supports several other optional arguments you can use to achieve the best possible result, such as specifying a damping parameter to handle additive noise in the blurred image. To see the impact of these optional arguments, view the Image Processing Toolbox deblurring demos.

## Deblurring with the Blind Deconvolution Algorithm

Use the `deconvblind` function to deblur an image using the blind deconvolution algorithm. The algorithm maximizes the likelihood that the resulting image, when convolved with the resulting PSF, is an instance of the blurred image, assuming Poisson noise statistics. The blind deconvolution algorithm can be used effectively when no information about the distortion (blurring and noise) is known. The `deconvblind` function restores the image and the PSF simultaneously, using an iterative process similar to the accelerated, damped Lucy-Richardson algorithm.

The `deconvblind` function, just like the `deconvlucy` function, implements several adaptations to the original Lucy-Richardson maximum likelihood algorithm that address complex image restoration tasks. Using these adaptations, you can

- Reduce the effect of noise on the restoration
- Account for nonuniform image quality (e.g., bad pixels)
- Handle camera read-out noise

For more information about these adaptations, see “Deblurring with the Lucy-Richardson Algorithm” on page 13-10. In addition, the `deconvblind` function supports PSF constraints that can be passed in through a user-specified function.

### **Example: Using the `deconvblind` Function to Deblur an Image**

To illustrate blind deconvolution, this example creates a simulated blurred image and then uses `deconvblind` to deblur it. The example makes two passes at deblurring the image to show the effect of certain optional parameters on the deblurring operation:

**1** Read an image into the MATLAB workspace.

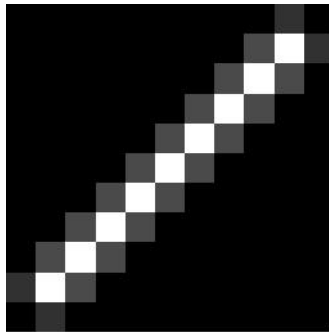
```
I = imread('cameraman.tif');  
figure; imshow(I); title('Original Image');
```



Image Courtesy of MIT

**2** Create the PSF.

```
PSF = fspecial('motion',13,45);  
figure; imshow(PSF,[],'notruesize'); title('Original PSF');
```



Original PSF

**3** Create a simulated blur in the image.

```
Blurred = imfilter(I,PSF,'circ','conv');  
figure; imshow(Blurred); title('Blurred Image');
```



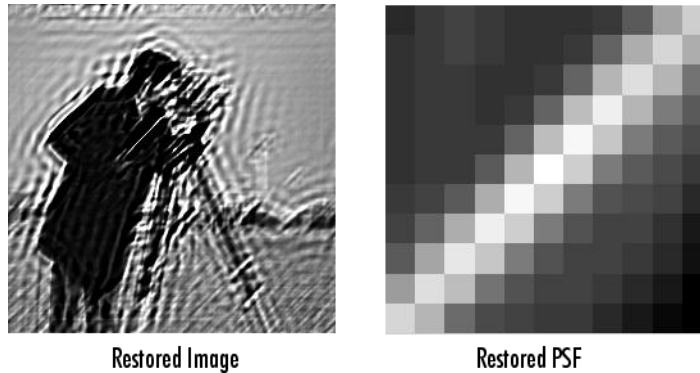
**4** Deblur the image, making an initial guess at the size of the PSF.

To determine the size of the PSF, examine the blurred image and measure the width of a blur (in pixels) around an obviously sharp object. In the sample blurred image, you can measure the blur near the contour of the man's sleeve. Because the size of the PSF is more important than the values it contains, you can typically specify an array of 1's as the initial PSF.

The following figure shows a restoration where the initial guess at the PSF is the same size as the PSF that caused the blur. In a real application, you might need to rerun `deconvblind`, experimenting with PSFs of different sizes, until you achieve a satisfactory result. The restored PSF returned by each deconvolution can also provide valuable hints at the optimal PSF size. See the Image Processing Toolbox deblurring demos for an example.

```
INITPSF = ones(size(PSF));  
[J P]= deconvblind(Blurred,INITPSF,30);  
figure; imshow(J); title('Restored Image');  
figure; imshow(P,[],'notruesize');  
title('Restored PSF');
```





Although `deconvblind` was able to deblur the image to a great extent, the ringing around the sharp intensity contrast areas in the restored image is unsatisfactory. (The example eliminated edge-related ringing by using the `'circular'` option with `imfilter` when creating the simulated blurred image in step 3.)

The next steps in the example repeat the deblurring process, attempting to achieve a better result by

- Eliminating high-contrast areas from the processing
  - Specifying a better PSF
- 5** Create a `WEIGHT` array to exclude areas of high contrast from the deblurring operation. This can reduce contrast-related ringing in the result.

To exclude a pixel from processing, you create an array of the same size as the original image, and assign the value 0 to the pixels in the array that correspond to pixels in the original image that you want to exclude from processing. (See for information about `WEIGHT` arrays.)

To create a `WEIGHT` array, the example uses a combination of edge detection and morphological processing to detect high-contrast areas in the image. Because the blur in the image is linear, the example dilates the image twice. (For more information about using these functions, see Chapter 10, “Morphological Operations”.) To exclude the image boundary pixels (a high-contrast area) from processing, the example uses `padarray` to assign the value 0 to all border pixels.

```
WEIGHT = edge(I,'sobel',.28);
se1 = strel('disk',1);
se2 = strel('line',13,45);
WEIGHT = ~imdilate(WEIGHT,[se1 se2]);
WEIGHT = padarray(WEIGHT(2:end-1,2:end-1),[2 2]);
figure; imshow(WEIGHT); title('Weight Array');
```



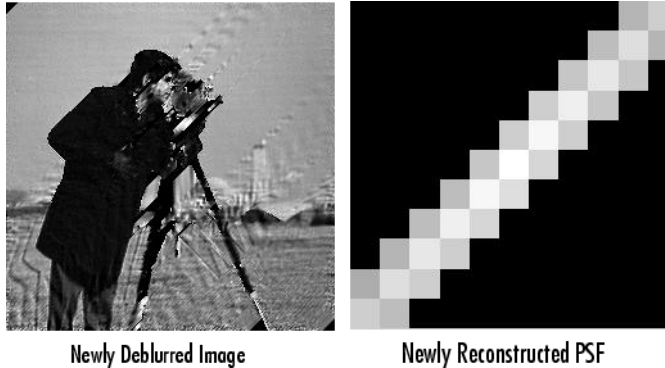
Weight Array

- 6** Refine the guess at the PSF. The reconstructed PSF  $P$  returned by the first pass at deconvolution shows a clear linearity, as shown in the figure in step 4. For the second pass, the example uses a new PSF,  $P_1$ , which is the same as the restored PSF but with the small amplitude pixels set to 0.

```
P1 = P;
P1(find(P1 < 0.01))=0;
```

- 7** Rerun the deconvolution, specifying the  $WEIGHT$  array and the modified PSF. Note how the restored image has much less ringing around the sharp intensity contrast areas than the result of the first pass (step 4).

```
[J2 P2] = deconvblind(Blurred,P1,50,[],WEIGHT);
figure; imshow(J2);
title('Newly Deblurred Image');
figure; imshow(P2,[],'notruesize');
title('Newly Reconstructed PSF');
```



## Refining the Result

The `deconvblind` function, by default, performs multiple iterations of the deblurring process. You can stop the processing after a certain number of iterations to check the result, and then restart the iterations from the point where processing stopped. To use this feature, you must pass in both the blurred image and the PSF as cell arrays, for example, `{Blurred}` and `{INITPSF}`.

The `deconvblind` function returns the output image and the restored PSF as cell arrays. The output image cell array contains these four elements:

Element	Description
<code>output{1}</code>	Original input image
<code>output{2}</code>	Image produced by the last iteration
<code>output{3}</code>	Image produced by the next to last iteration
<code>output{4}</code>	Internal information used by <code>deconvlucy</code> to know where to restart the process

The PSF output cell array contains similar elements.

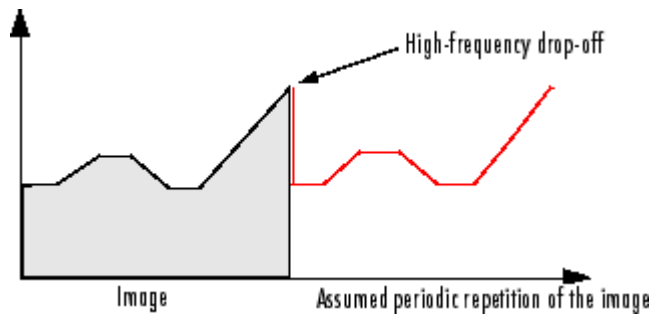
The `deconvblind` function supports several other optional arguments you can use to achieve the best possible result, such as specifying a damping parameter to handle additive noise in the blurred image. To see the impact of these optional arguments, as well as the functional option that allows you to place additional constraints on the PSF reconstruction, see the Image Processing Toolbox deblurring demos.

## Creating Your Own Deblurring Functions

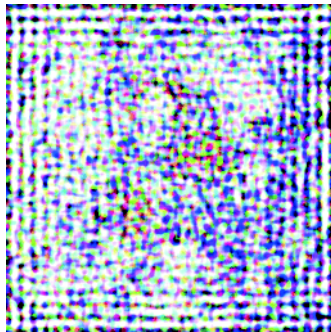
All the toolbox deblurring functions perform deconvolution in the frequency domain, where the process becomes a simple matrix multiplication. To work in the frequency domain, the deblurring functions must convert the PSF you provide into an optical transfer function (OTF), using the `psf2otf` function. The toolbox also provides a function to convert an OTF into a PSF, `otf2psf`. The toolbox makes these functions available in case you want to create your own deblurring functions. (In addition, to aid this conversion between PSFs and OTFs, the toolbox also makes the padding function `padarray` available.)

## Avoiding Ringing in Deblurred Images

The discrete Fourier transform (DFT), used by the deblurring functions, assumes that the frequency pattern of an image is periodic. This assumption creates a high-frequency drop-off at the edges of images. In the figure, the shaded area represents the actual extent of the image; the unshaded area represents the assumed periodicity.



This high-frequency drop-off can create an effect called *boundary related ringing* in deblurred images. In this figure, note the horizontal and vertical patterns in the image.



To avoid ringing, use the `edgetaper` function to preprocess your images before passing them to the deblurring functions. The `edgetaper` function removes the high-frequency drop-off at the edge of an image by blurring the entire image and then replacing the center pixels of the blurred image with the original image. In this way, the edges of the image taper off to a lower frequency.

# Color

---

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.

Working with Different Screen Bit Depths (p. 14-2)

Describes how to determine the screen bit depth of your system and provides recommendations if you can change the bit depth

Reducing the Number of Colors in an Image (p. 14-5)

Describes how to use `imapprox` and `rgb2ind` to reduce the number of colors in an image, including information about dithering

Converting Color Data Between Color Spaces (p. 14-14)

Defines the concept of image color space and describes how to convert images between color spaces

## 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 are displayed best on systems with 24-bit color, but usually look fine on 16-bit systems as well. (For additional information about how MATLAB handles color, see the MATLAB graphics documentation.)

This section

- Describes how to determine your system's screen bit depth
- Provides guidelines for choosing a screen bit depth

### Determining Screen Bit Depth

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

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

32

The integer MATLAB returns represents the number of bits per screen pixel:



Value	Screen Bit Depth
8	8-bit displays support 256 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 the available color slots.)
16	16-bit displays 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). Some systems use the extra bit to increase the number of levels of green that can be displayed. In this case, the number of different colors supported by a 16-bit display is actually 64,536 (i.e. $2^{16}$ ).
24	24-bit displays use 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 lifelike image.
32	32-bit displays use 24 bits to store color information and use the remaining 8 bits to store transparency data (alpha channel). For information about how MATLAB supports the alpha channel, see the section “Transparency” in the MATLAB 3-D Visualization documentation.

## Choosing a Screen Bit Depth

Depending on your system, you might be able to choose the screen bit depth you want to use. (There might be tradeoffs 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 might 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 might get better display results using 8-bit display mode, which provides up to 256 shades of gray.

For information about reducing the number of colors used by an image, see “Reducing the Number of Colors in an Image” on page 14-5.

## 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
<code>imapprox</code>	Reduces the number of colors used by an indexed image, enabling you to specify the number of colors in the new colormap.
<code>rgb2ind</code>	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, truecolor images can display up to 16,777,216 (i.e.,  $2^{24}$ ) colors. On systems with lower screen bit depths, truecolor images are still displayed reasonably well, because MATLAB automatically uses color approximation and dithering if needed. Color approximation is the process 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 — see “Color Approximation” on page 14-6.

Indexed images, however, might 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, might 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.

## Color Approximation

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

- Quantization
  - Uniform quantization
  - Minimum variance quantization
- Colormap mapping

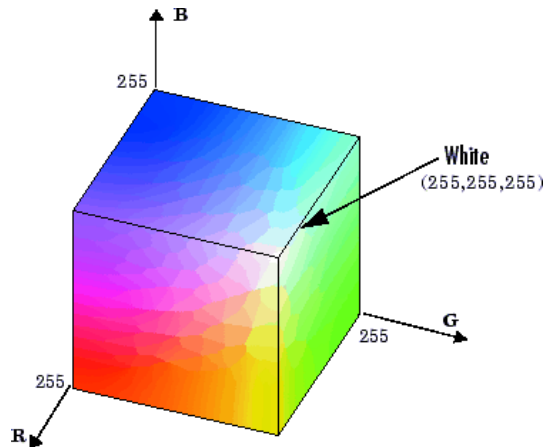
The quality of the resulting image depends on the approximation method you use, the range of colors in the input image, and whether or not you use dithering. Note that different methods work better for different images. See “Dithering” on page 14-12 for a description of dithering and how to enable or 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 a `uint8` RGB image appears 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). The following figure shows an RGB color cube for a `uint8` image.



### 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.

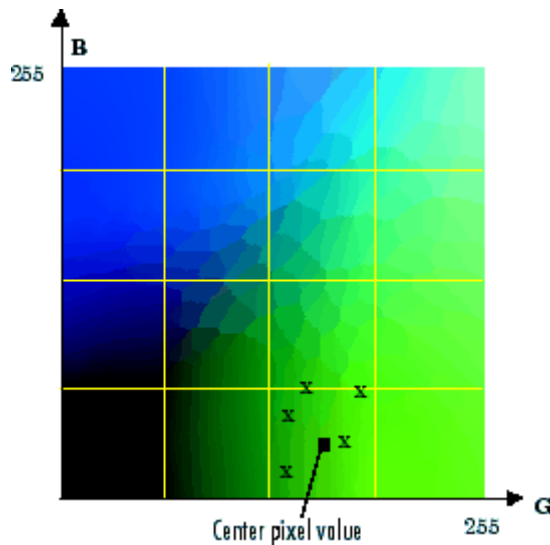
**Uniform Quantization.** To perform uniform quantization, call `rgb2ind` and specify a *tolerance*. The tolerance determines the size of the cube-shaped boxes into which the RGB color cube is divided. The allowable range for a tolerance setting is  $[0,1]$ . For example, if you specify a tolerance of 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('peppers.png');
[x,map] = rgb2ind(RGB, 0.1);
```

The following figure 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.



### 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 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('peppers.png');  
[X,map] = rgb2ind(RGB,185);
```

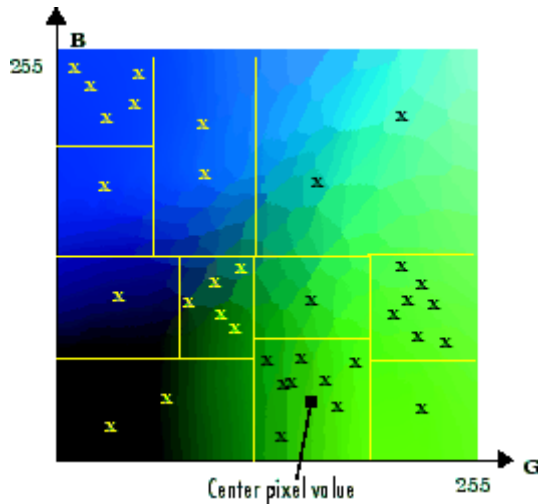
Minimum variance quantization works by associating pixels into groups based on the variance between their pixel values. For example, a set of blue pixels might be grouped together because they have a small variance from the center pixel of the group.

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.

The following figure shows the same two-dimensional slice of the color cube as shown in the preceding figure (demonstrating uniform quantization). Eleven boxes have been created using minimum variance quantization.



### 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('peppers.png');
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 4-48 or the reference page for `subimage`.

---

## Reducing Colors in an Indexed Image

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 14-12 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 might 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 orange pixels for which there is no exact match in the colormap. To create the appearance of this shade of orange, 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 orange. From a distance, the pixels appear to be the correct shade, but if you look up close at the image, you can see a blend of other shades. This example loads a 24-bit image, and then use `rgb2ind` to create two indexed images with just eight colors each:

- 1 Read image and display it.

```
rgb=imread('onion.png');  
imshow(rgb);
```



- 2 Create an indexed image with eight colors and without dithering.

```
[X_no_dither,map]= rgb2ind(rgb,8,'nodither');  
figure, imshow(X_no_dither,map);
```



**3** Create an indexed image using eight colors with dithering.

```
[X_dither,map]=rgb2ind(rgb,8,'dither');  
figure, imshow(X_dither,map);
```



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 can contain false contours.

## Converting Color Data Between 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. The various models are referred to as *color spaces* because most of them can be mapped into a 2-D, 3-D, or 4-D coordinate system; thus, a color specification is made up of coordinates in a 2-D, 3-D, or 4-D space.

The various color spaces exist because they present color information in ways that make certain calculations more convenient or because they provide a way to identify colors that is more intuitive. For example, the RGB color space defines a color as the percentages of red, green, and blue hues mixed together. Other color models describe colors by their hue (green), saturation (dark green), and luminance, or intensity.

The toolbox supports these color spaces by providing a means for converting color data from one color space to another through a mathematical transformation.

This section

- Describes how to convert color data between these color spaces
- Describes how to perform color space conversions using ICC profiles
- Describes some toolbox functions for converting between the RGB color space and three commonly used color spaces: YIQ, HSV, and YCbCr

### Converting Between Device-Independent Color Spaces

The standard terms used to describe colors, such as hue, brightness, and intensity, are subjective and make comparisons difficult.

In 1931, the International Commission on Illumination, known by the acronym CIE, for *Commission Internationale de l'Éclairage*, studied human color perception and developed a standard, called the CIE XYZ. This standard defined a three-dimensional space where three values, called tristimulus values, define a color. This standard is still widely used today.

In the decades since that initial specification, the CIE has developed several additional color space specifications that attempt to provide alternative color representations that are better suited to some purposes than XYZ. For example, in 1976, in an effort to get a perceptually uniform color space that could be correlated with the visual appearance of colors, the CIE created the  $L^*a^*b^*$  color space.

The toolbox supports conversions between members of the CIE family of device-independent color spaces. In addition, the toolbox also supports conversions between these CIE color spaces and the sRGB color space. This color space was defined by an industry group to describe the characteristics of a typical PC monitor.

This section

- Lists the supported device-independent color spaces
- Provides an example of how to perform a conversion
- Provides guidelines about data type support of the various conversions

## Supported Conversions

This table lists all the device-independent color spaces that the toolbox supports.

Color Space	Description	Supported Conversions
XYZ	The original, 1931 CIE color space specification.	$xyY$ , $uvL$ , $u'v'L$ , and $L^*a^*b^*$
$xyY$	CIE specification that provides normalized chromaticity values. The capital Y value represents luminance and is the same as in XYZ.	XYZ
$uvL$	CIE specification that attempts to make the chromaticity plane more visually uniform. $l$ is luminance and is the same as Y in XYZ.	XYZ
$u'v'L$	CIE specification in which $u$ and $v$ are rescaled to improve uniformity.	XYZ

Color Space	Description	Supported Conversions
$L^*a^*b^*$	CIE specification that attempts to make the luminance scale more perceptually uniform. $L^*$ is a nonlinear scaling of $L$ , normalized to a reference white point.	$XYZ$
$L^*ch$	CIE specification where $c$ is chroma and $h$ is hue. These values are a polar coordinate conversion of $a^*$ and $b^*$ in $L^*a^*b^*$ .	$L^*a^*b^*$
$sRGB$	Standard adopted by major manufacturers that characterizes the average PC monitor.	$XYZ$ and $L^*a^*b^*$

### Example: Performing a Color Space Conversion

To illustrate a conversion between two device-independent color spaces, this example reads an RGB color image into the MATLAB workspace and converts the color data to the XYZ color space:

- 1 Import color space data. This example reads an RGB color image into the MATLAB workspace.

```
I_rgb = imread('peppers.png');
```

- 2 Create a color transformation structure. A color transformation structure defines the conversion between two color spaces. You use the `makecform` function to create the structure, specifying a transformation type string as an argument.

This example creates a color transformation structure that defines a conversion from RGB color data to XYZ color data.

```
C = makecform('srgb2xyz');
```

- 3** Perform the conversion. You use the `applycform` function to perform the conversion, specifying as arguments the color data you want to convert and the color transformation structure that defines the conversion. The `applycform` function returns the converted data.

```
I_xyz = applycform(I_rgb,C);
whos
      Name          Size          Bytes  Class
      C              1x1              7744  struct array
      I_xyz          384x512x3         1179648  uint16 array
      I_rgb          384x512x3         589824  uint8 array
```

### Color Space Data Encodings

When you convert between two device-independent color spaces, the data type used to encode the color data can sometimes change, depending on what encodings the color spaces support. In the preceding example, the original image is `uint8` data. The XYZ conversion is `uint16` data. The XYZ color space does not define a `uint8` encoding. The following table lists the data types that can be used to represent values in all the device-independent color spaces.

Color Space	Encodings
XYZ	uint16 or double
xyY	double
uvL	double
u'v'L	double
L*a*b*	uint8, uint16, or double
L*ch	double
sRGB	double

As the table indicates, certain color spaces have data type limitations. For example, the XYZ color space does not define a `uint8` encoding. If you convert 8-bit CIE LAB data into the XYZ color space, the data is returned in `uint16` format. If you want the returned XYZ data to be in the same format as the input LAB data, you can use one of the following toolbox color space format conversion functions.

- `lab2double`
- `lab2uint8`
- `lab2uint16`
- `xyz2double`
- `xyz2uint16`

## Performing Profile-Based Color Space Conversions

The Image Processing Toolbox can perform color space conversions based on device profiles. This section includes the following topics:

- “Understanding Device Profiles” on page 14-18
- “Reading ICC Profiles” on page 14-19
- “Writing Profile Information to a File” on page 14-19
- “Example: Performing a Profile-Based Conversion” on page 14-20
- “Specifying the Rendering Intent” on page 14-21

## Understanding Device Profiles

If two colors have the same CIE colorimetry, they will match *if viewed under the same conditions*. However, because color images are typically produced for a wide variety of viewing environments, it is necessary to go beyond simple application of the CIE system.

For this reason, the International Color Consortium (ICC) has defined a Color Management System (CMS) that provides a means for communicating color information among input, output, and display devices. The CMS uses device *profiles* that contain color information specific to a particular device. Vendors that support CMS provide profiles that characterize the color reproduction of their devices, and methods, called Color Management Modules (CMM), that interpret the contents of each profile and perform the necessary image processing.

Device profiles contain the information that color management systems need to translate color data between devices. Any conversion between color spaces is a mathematical transformation from some domain space to a range space. With profile-based conversions, the domain space is often called the *source*



*space* and the range space is called the *destination space*. In the ICC color management model, profiles are used to represent the source and destination spaces.

For more information about color management systems, go to the International Color Consortium Web site, [www.color.org](http://www.color.org).

## Reading ICC Profiles

To read an ICC profile into the MATLAB workspace, use the `iccread` function. In this example, the function reads in the profile for the color space that describes color monitors.

```
P = iccread('sRGB.icm');
```

You can use the `iccfind` function to find ICC color profiles on your system, or to find a particular ICC color profile whose description contains a certain text string. To get the name of the directory that is the default system repository for ICC profiles, use `iccroot`.

`iccread` returns the contents of the profile in the structure `P`. All profiles contain a header, a tag table, and a series of tagged elements. The header contains general information about the profile, such as the device class, the device color space, and the file size. The tagged elements, or tags, are the data constructs that contain the information used by the CMM. For more information about the contents of a profile, see the `iccread` function reference page.

Using `iccread`, you can read both Version 2 (ICC.1:2001-04) or Version 4 (ICC.1:2001-12) ICC profile formats. For detailed information about these specifications and their differences, visit the ICC web site, [www.color.org](http://www.color.org).

## Writing Profile Information to a File

To export ICC profile information from the MATLAB workspace to a file, use the `iccwrite` function. This example reads a profile into the MATLAB workspace and then writes the profile information out to a new file.

```
P = iccread('sRGB.icm');  
P_new = iccwrite(P,'my_profile.icm');
```

`iccwrite` returns the profile it writes to the file in `P_new` because it can be different than the input profile `P`. For example, `iccwrite` updates the `Filename` field in `P` to match the name of the file specified as the second argument.

`iccwrite` checks the validity of the input profile structure. If any required fields are missing, `iccwrite` errors. To determine if a structure is a valid ICC profile, use the `isicc` function.

Using `iccwrite`, you can export profile information in both Version 2 (ICC.1:2001-04) or Version 4 (ICC.1:2001-12) ICC profile formats. The value of the `Version` field in the file profile header determines the format version. For detailed information about these specifications and their differences, visit the ICC web site, [www.color.org](http://www.color.org).

### **Example: Performing a Profile-Based Conversion**

To illustrate a profile-based color space conversion, this section presents an example that converts color data from the RGB space of a monitor to the CMYK space of a printer. This conversion requires two profiles: a monitor profile and a printer profile. The source color space in this example is monitor RGB and the destination color space is printer CMYK:

- 1 Import RGB color space data. This example imports an RGB color image into the MATLAB workspace.

```
I_rgb = imread('peppers.png');
```

- 2 Read ICC profiles. Read the source and destination profiles into the MATLAB workspace. This example uses the sRGB profile as the source profile. The sRGB profile is an industry-standard color space that describes a color monitor.

```
inprof = iccread('sRGB.icm');
```

For the destination profile, the example uses a profile that describes a particular color printer. The printer vendor supplies this profile. (The following profile and several other useful profiles can be obtained as downloads from [www.adobe.com](http://www.adobe.com).)

```
outprof = iccread('USSheetfedCoated.icc');
```

- 3 Create a color transformation structure. You must create a color transformation structure to define the conversion between the color spaces in the profiles. You use the `makecform` function to create the structure, specifying a transformation type string as an argument.

---

**Note** The color space conversion might involve an intermediate conversion into a device-independent color space, called the Profile Connection Space (PCS), but this is transparent to the user.

---

This example creates a color transformation structure that defines a conversion from RGB color data to CMYK color data.

```
C = makecform('icc',inprof,outprof);
```

- 4 Perform the conversion. You use the `applycform` function to perform the conversion, specifying as arguments the color data you want to convert and the color transformation structure that defines the conversion. The function returns the converted data.

```
I_cmyk = applycform(I_rgb,C);
```

- 5 Write the converted data to a file. To export the CMYK data, use the `imwrite` function, specifying the format as TIFF. If the format is TIFF and the data is an m-by-n-by-4 array, `imwrite` writes CMYK data to the file.

```
imwrite(I_cmyk,'pep_cmyk.tif','tif')
```

To verify that the CMYK data was written to the file, use `imfinfo` to get information about the file and look at the `PhotometricInterpretation` field.

```
info = imfinfo('pep_cmyk.tif');
info.PhotometricInterpretation
ans =
    'CMYK'
```

## Specifying the Rendering Intent

For most devices, the range of reproducible colors is much smaller than the range of colors represented by the PCS. It is for this reason that four

rendering intents (or gamut mapping techniques) are defined in the profile format. Each one has distinct aesthetic and color-accuracy tradeoffs.

When you create a profile-based color transformation structure, you can specify the rendering intent for the source as well as the destination profiles. For more information, see the `makecform` reference information.

## Converting Between Device-Dependent Color Spaces

The toolbox includes functions that you can use to convert RGB data to several common device-dependent color spaces, and vice versa:

- YIQ
- YCbCr
- Hue, saturation, value (HSV)

### YIQ Color Space

The National Television Systems Committee (NTSC) defines a color space known as YIQ. This 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 color space, 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 an RGB image to NTSC format.

```
RGB = imread('peppers.png');  
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 the 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. (YUV, another color space widely used for digital video, is very similar to YCbCr but not identical.)

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 an RGB image to YCbCr format.

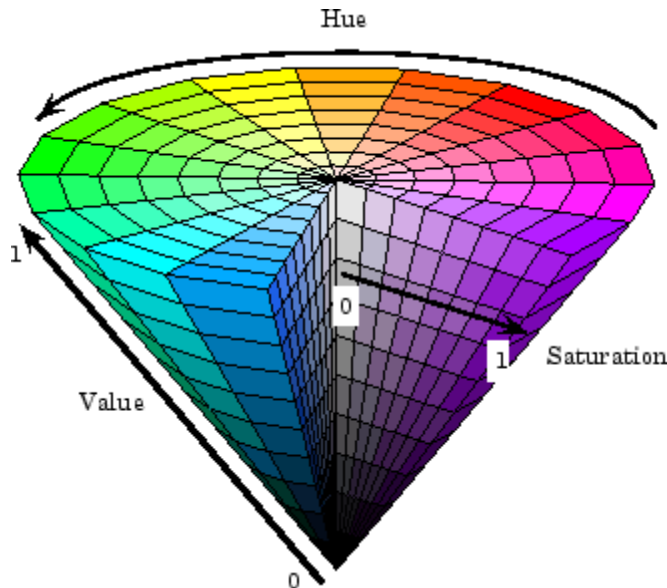
```
RGB = imread('peppers.png');  
YCBCR = rgb2ycbcr(RGB);
```

## 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, magenta, and 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.

The following figure illustrates the HSV color space.



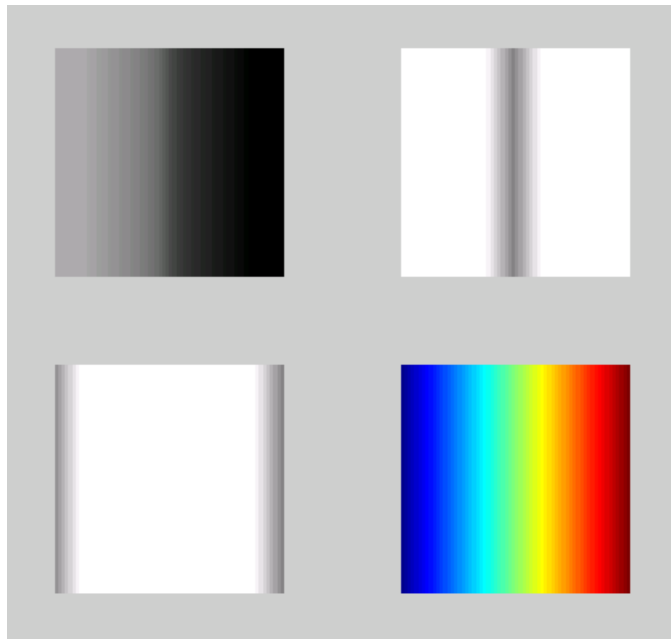
### Illustration of the HSV Color Space

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

```
RGB = imread('peppers.png');  
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);  
subplot(2,2,1), imshow(H)  
subplot(2,2,2), imshow(S)  
subplot(2,2,3), imshow(V)  
subplot(2,2,4), imshow(RGB)
```



**The Separated Color Planes of an HSV Image**

As the hue plane image in the preceding figure illustrates, hue values make a 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. (As stated previously, there are values of red on both ends of the hue scale. To avoid confusion, the 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 correspond to the brightest colors in the original image.



# Neighborhood and Block Operations

---

This chapter discusses these generic block processing functions. Topics covered include

Block Processing Operations (p. 15-2)	Provides an overview of the types of block processing operations supported by the toolbox
Sliding Neighborhood Operations (p. 15-4)	Defines sliding neighborhood operations and describes how you can use them to implement many types of filtering operations
Distinct Block Operations (p. 15-8)	Describes block operations
Column Processing Operations (p. 15-12)	Describes how to process sliding neighborhoods or distinct blocks as columns

## Block Processing Operations

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 `imdilate` function for image dilation. In addition, the toolbox provides more generic functions for processing an image in blocks. This section discusses these generic block processing functions.

To use these functions, you supply two pieces of information:

- The size of the blocks
- The function to use to process the blocks.

The block processing function does the work of breaking the input image into blocks, calling the specified function for each block, and reassembling the results into an output image.

### Types of Block Processing Operations

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

- Sliding neighborhood operations — 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 performed on a block of neighboring pixels. For more information, see “Sliding Neighborhood Operations” on page 15-4.
- Distinct block operations — 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. For more information, see “Distinct Block Operations” on page 15-8.

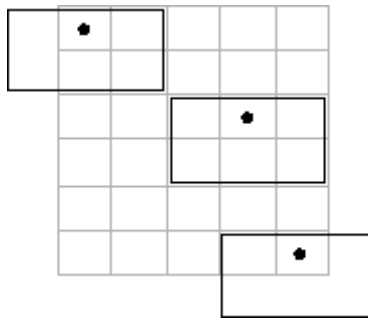
In addition, the toolbox provides functions for *column processing operations*. These operations are not actually distinct from block operations; instead, they are a way of speeding up block operations by rearranging blocks into matrix columns. For more information, see “Column Processing Operations” on page 15-12.

Note that even if you do not use these block processing functions, the information here might 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 morphological operations. See Chapter 8, “Linear Filtering and Filter Design” and Chapter 10, “Morphological 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 the application of an algorithm to the values of the corresponding input pixel's *neighborhood*. A pixel's neighborhood is some set of pixels, defined by their locations relative to that pixel, which is called the *center pixel*. The neighborhood is a rectangular block, and as you move from one element to the next in an image matrix, the neighborhood block slides in the same direction.

The following figure 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.



**Neighborhood Blocks in a 6-by-5 Matrix**

### Determining the Center Pixel

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

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

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

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

## Performing a Sliding Neighborhood Operation

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, the function might be an averaging operation that sums the values of the neighborhood pixels and then divides the result by the number of pixels in the neighborhood. The result of this calculation is the value of the output pixel.

## Padding Borders

As the neighborhood block slides over the image, some of the pixels in a neighborhood might be missing, especially if the center pixel is on the border of the image. For example, if the center pixel is the pixel in the upper left corner of the image, the 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.

## Implementing Linear and Nonlinear Filtering

You can use sliding neighborhood operations to implement many kinds of filtering operations. One example of a sliding neighborhood operation is convolution, which is used to implement linear filtering. MATLAB provides the `conv` and `filter2` functions for performing convolution, and the toolbox provides the `imfilter` function. See Chapter 8, “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).

```
I = imread('tire.tif');  
I2 = nlfilter(I,[3 3],'std2');
```

You can also 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`. The syntax `@myfun` is an example of a function handle.

```
I2 = nlfilter(I,[2 3],@myfun);
```

If you prefer not to write a M-file, you can use an anonymous function instead. This example converts the image to class double because the square root function is not defined for the uint8 datatype.

```
I = im2double(imread('tire.tif'));
f = @(x) sqrt(min(x(:)));
I2 = nlfiter(I,[2 2],f);
```

(For more information on function handles and anonymous functions, see `function_handle` in the MATLAB Function Reference documentation.)

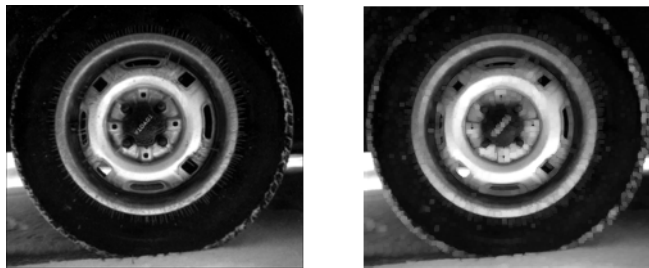
The following example uses `nlfiter` to set each pixel to the maximum value in its 3-by-3 neighborhood.

---

**Note** This example is only intended to illustrate the use of `nlfiter`. For a faster way to perform this local maximum operation, use `imdilate`.

---

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

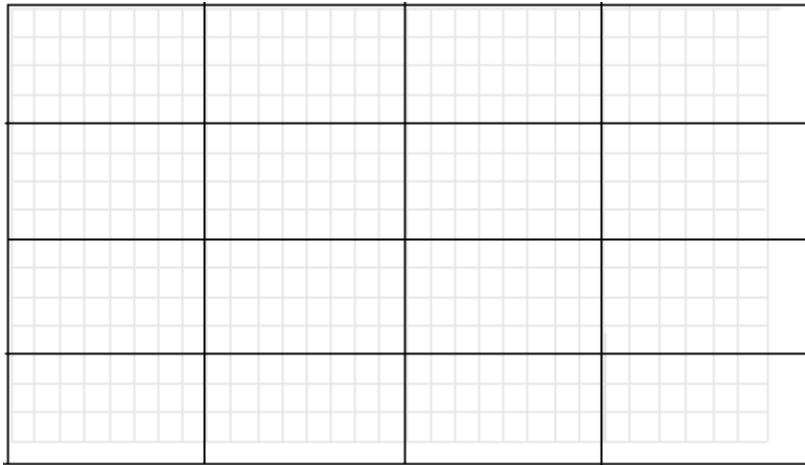


### Each Output Pixel Set to Maximum Input Neighborhood Value

Many operations that `nlfiter` 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`.

## Distinct Block Operations

*Distinct blocks* are rectangular partitions that divide a matrix into m-by-n sections. Distinct blocks overlay the image matrix starting in the upper left corner, with no overlap. If the blocks don't fit exactly over the image, the toolbox adds zero padding so that they do. The following figure shows a 15-by-30 matrix divided into 4-by-8 blocks.



### 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);
```



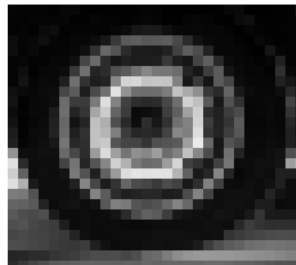
If you prefer not to create an M-file, you can specify the function as an anonymous function. For example:

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

(For more information about using function handles and anonymous functions, see `function_handle` in the MATLAB Function Reference documentation.)

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 = @(x) uint8(round(mean2(x)*ones(size(x))));  
I2 = blkproc(I,[8 8],f);  
imshow(I)  
figure, imshow(I2);
```

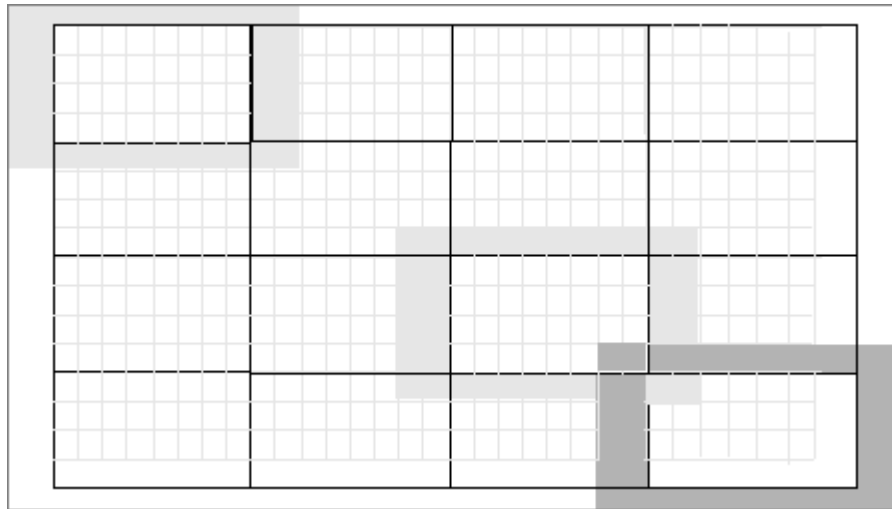


The anonymous function in the example computes the mean of the block and then multiplies the result by a matrix of ones, so that the output block is the same size as the input block. As a result, the output image is the same size as the input image. `blkproc` does not require that the images be the same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

## Specifying 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.

The following figure 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.



### 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 and Zero-padding**

Overlap often increases the amount of zero padding needed. For example, in the figure, 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 preceding figure, padding has been added to the left and top of the original image, not just to the right and bottom.

## Column Processing Operations

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

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

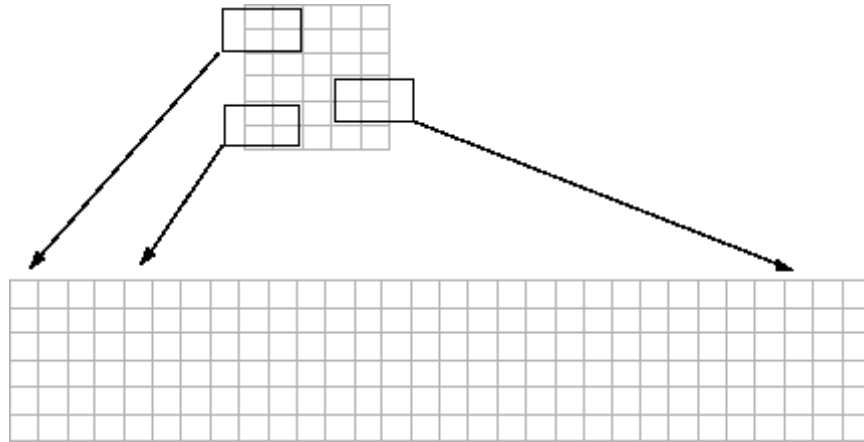
You can use the `colfilt` function to implement column processing. This function

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

### Sliding Neighborhoods

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

The following figure 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.



### **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.

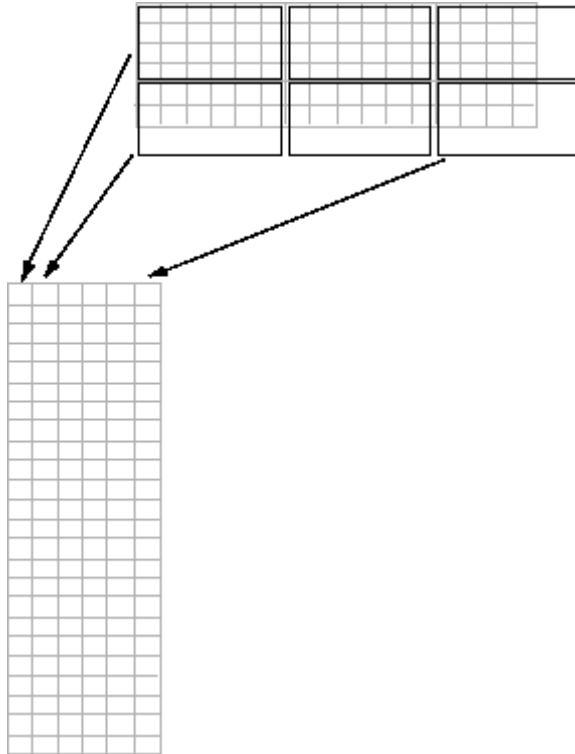
colfilt can produce the same results as nlfilt with faster execution time; however, it might use more memory. The example below sets each output pixel to the maximum value in the input pixel's neighborhood, producing the same result as the nlfilt example shown in “Implementing Linear and Nonlinear Filtering” on page 15-6.

```
I2 = colfilt(I,[3 3],'sliding',@max);
```

### **Using Column Processing with Distinct Block Operations**

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

The following figure illustrates this process. 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 six columns of 24 elements each.



### **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- $(\text{ceil}(mm/m)*\text{ceil}(nn/n))$ . After the function processes the temporary matrix, the output is rearranged 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 15-8.

```
I = im2double(imread('tire.tif'));
f = @(x) ones(64,1)*mean(x);
I2 = colfilt(I,[8 8],'distinct',f);
```

The anonymous function in the example 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`.





# Functions — By Category

---

Image Display and Exploration (p. 16-2)	Display, import, and export images
GUI Tools (p. 16-5)	Modular interactive tools and associated utility functions.
Spatial Transformation and Image Registration (p. 16-8)	Spatial transformation and image registration
Image Analysis and Statistics (p. 16-10)	Image analysis, texture analysis, view pixel values, and calculate image statistics
Image Arithmetic (p. 16-12)	Add, subtract, multiply, and divide images
Image Enhancement and Restoration (p. 16-13)	Enhance and restore images
Linear Filtering and Transforms (p. 16-15)	Linear filters, filter design, and image transforms
Morphological Operations (p. 16-17)	Morphological image processing
Region-Based, Neighborhood, and Block Processing (p. 16-20)	Region-based, neighborhood, and block operations
Colormap and Color Space Functions (p. 16-21)	Manipulate image color
Miscellaneous Functions (p. 16-23)	Array operations, demos, preferences and other toolbox utility functions

## Image Display and Exploration

Image Display and Exploration (p. 16-2)	Display and explore images
Image File I/O (p. 16-2)	Import and export images
Image Types and Type Conversions (p. 16-3)	Convert between the various image types

## Image Display and Exploration

colorbar	Display color bar
immovie	Make movie from multiframe image
imshow	Display image
imtool	Image Tool
montage	Display multiple image frames as rectangular montage
subimage	Display multiple images in single figure
warp	Display image as texture-mapped surface

## Image File I/O

analyze75info	Read metadata from header file of Analyze 7.5 data set
analyze75read	Read image data from image file of Analyze 7.5 data set
dicomanon	Anonymize DICOM file
dicomdict	Get or set active DICOM data dictionary

<code>dicominfo</code>	Read metadata from DICOM message
<code>dicomlookup</code>	Find attribute in DICOM data dictionary
<code>dicomread</code>	Read DICOM image
<code>dicomuid</code>	Generate DICOM unique identifier
<code>dicomwrite</code>	Write images as DICOM files
<code>interfileinfo</code>	Read metadata from Interfile file
<code>interfileread</code>	Read images in Interfile format

## Image Types and Type Conversions

<code>dither</code>	Convert image, increasing apparent color resolution by dithering
<code>double</code>	Convert data to double precision
<code>gray2ind</code>	Convert grayscale or binary image to indexed image
<code>grayslice</code>	Convert grayscale image to indexed image using multilevel thresholding
<code>graythresh</code>	Global image threshold using Otsu's method
<code>im2bw</code>	Convert image to binary image, based on threshold
<code>im2double</code>	Convert image to double precision
<code>im2int16</code>	Convert image to 16-bit signed integers
<code>im2java</code>	Convert image to Java image
<code>im2java2d</code>	Convert image to Java buffered image
<code>im2single</code>	Convert image to single precision

<code>im2uint16</code>	Convert image to 16-bit unsigned integers
<code>im2uint8</code>	Convert image to 8-bit unsigned integers
<code>ind2gray</code>	Convert indexed image to grayscale image
<code>ind2rgb</code>	Convert indexed image to RGB image
<code>label2rgb</code>	Convert label matrix into RGB image
<code>mat2gray</code>	Convert matrix to grayscale image
<code>rgb2gray</code>	Convert RGB image or colormap to grayscale
<code>rgb2ind</code>	Convert RGB image to indexed image
<code>uint16</code>	Convert data to unsigned 16-bit integers
<code>uint8</code>	Convert data to unsigned 8-bit integers

## GUI Tools

Modular Interactive Tools (p. 16-5)	Modular interactive tool creation functions
Navigational tools for Image Scroll Panel (p. 16-5)	Modular interactive navigational tools
Utility Functions for Interactive Tools (p. 16-6)	Modular interactive tool utility functions

### Modular Interactive Tools

<code>imageinfo</code>	Image Information tool
<code>imcontrast</code>	Adjust Contrast tool
<code>imdisplayrange</code>	Display Range tool
<code>imdistline</code>	Distance tool
<code>impixelinfo</code>	Pixel Information tool
<code>impixelinfoval</code>	Pixel Information tool without text label
<code>impixelregion</code>	Pixel Region tool
<code>impixelregionpanel</code>	Pixel Region tool panel

### Navigational tools for Image Scroll Panel

<code>immagbox</code>	Magnification box for scroll panel
<code>imoverview</code>	Overview tool for image displayed in scroll panel
<code>imoverviewpanel</code>	Overview tool panel for image displayed in scroll panel
<code>imscrollpanel</code>	Scroll panel for interactive image navigation

## Utility Functions for Interactive Tools

<code>axes2pix</code>	Convert axes coordinates to pixel coordinates
<code>getimage</code>	Image data from axes
<code>getimagemodel</code>	Image model object from image object
<code>imattributes</code>	Information about image attributes
<code>imgca</code>	Get handle to current axis containing image
<code>imgcf</code>	Get handle to current figure containing image
<code>imgetfile</code>	Open Image dialog box
<code>imhandles</code>	Get all image handles
<code>imline</code>	Create draggable, resizable line
<code>impoint</code>	Create draggable point
<code>imrect</code>	Create draggable rectangle
<code>iptaddcallback</code>	Add function handle to callback list
<code>iptcheckhandle</code>	Check validity of handle
<code>iptgetapi</code>	Get Application Programmer Interface (API) for handle
<code>iptGetPointerBehavior</code>	Retrieve pointer behavior from HG object
<code>ipticondir</code>	Directories containing IPT and MATLAB icons
<code>iptPointerManager</code>	Create pointer manager in figure
<code>iptremovecallback</code>	Delete function handle from callback list
<code>iptSetPointerBehavior</code>	Store pointer behavior structure in Handle Graphics object
<code>iptwindowalign</code>	Align figure windows

`makeConstrainToRectFcn`

Create rectangularly bounded drag  
constraint function

`trueSize`

Adjust display size of image

## Spatial Transformation and Image Registration

Spatial Transformations (p. 16-8)

Spatial transformation of images and multidimensional arrays

Image Registration (p. 16-9)

Align two images using control points

### Spatial Transformations

checkerboard

Create checkerboard image

findbounds

Find output bounds for spatial transformation

fliptform

Flip input and output roles of TFORM structure

imcrop

Crop image

imresize

Resize image

imrotate

Rotate image

imtransform

Apply 2-D spatial transformation to image

makeresampler

Create resampling structure

maketform

Create spatial transformation structure (TFORM)

tformarray

Apply spatial transformation to N-D array

tformfwd

Apply forward spatial transformation

tforminv

Apply inverse spatial transformation



## Image Registration

<code>cp2tform</code>	Infer spatial transformation from control point pairs
<code>cpcorr</code>	Tune control-point locations using cross correlation
<code>cpselect</code>	Control Point Selection Tool
<code>cpstruct2pairs</code>	Convert CPSTRUCT to valid pairs of control points
<code>normxcorr2</code>	Normalized 2-D cross-correlation

## Image Analysis and Statistics

Image Analysis (p. 16-10)	Trace boundaries, detect edges, and perform quadtree decomposition
Texture Analysis (p. 16-10)	Entropy, range, and standard deviation filtering; gray-level co-occurrence matrix
Pixel Values and Statistics (p. 16-11)	Create histograms, contour plots, and get statistics on image regions

### Image Analysis

<code>bwboundaries</code>	Trace region boundaries in binary image
<code>bwtraceboundary</code>	Trace object in binary image
<code>edge</code>	Find edges in grayscale image
<code>hough</code>	Hough transform
<code>houghlines</code>	Extract line segments based on Hough transform
<code>houghpeaks</code>	Identify peaks in Hough transform
<code>qtdecomp</code>	Quadtree decomposition
<code>qtgetblk</code>	Block values in quadtree decomposition
<code>qtsetblk</code>	Set block values in quadtree decomposition

### Texture Analysis

<code>entropy</code>	Entropy of grayscale image
<code>entropyfilt</code>	Local entropy of grayscale image

<code>graycomatrix</code>	Create gray-level co-occurrence matrix from image
<code>graycoprops</code>	Properties of gray-level co-occurrence matrix
<code>rangefilt</code>	Local range of image
<code>stdfilt</code>	Local standard deviation of image

## **Pixel Values and Statistics**

<code>corr2</code>	2-D correlation coefficient
<code>imcontour</code>	Create contour plot of image data
<code>imhist</code>	Display histogram of image data
<code>impixel</code>	Pixel color values
<code>improfile</code>	Pixel-value cross-sections along line segments
<code>mean2</code>	Average or mean of matrix elements
<code>pixval</code>	Display information about image pixels
<code>regionprops</code>	Measure properties of image regions (blob analysis)
<code>std2</code>	Standard deviation of matrix elements

## Image Arithmetic

<code>imabsdiff</code>	Absolute difference of two images
<code>imadd</code>	Add two images or add constant to image
<code>imcomplement</code>	Complement image
<code>imdivide</code>	Divide one image into another or divide image by constant
<code>imlincomb</code>	Linear combination of images
<code>immultiply</code>	Multiply two images or multiply image by constant
<code>imsubtract</code>	Subtract one image from another or subtract constant from image

## Image Enhancement and Restoration

Image Enhancement (p. 16-13)	Histogram equalization, decorrelation stretching, and 2-D filtering
Image Restoration (Deblurring) (p. 16-13)	Deconvolution for deblurring

### Image Enhancement

<code>adapthisteq</code>	Contrast-limited adaptive histogram equalization (CLAHE)
<code>decorrstretch</code>	Apply decorrelation stretch to multichannel image
<code>histeq</code>	Enhance contrast using histogram equalization
<code>imadjust</code>	Adjust image intensity values or colormap
<code>imnoise</code>	Add noise to image
<code>intlut</code>	Convert integer values using lookup table
<code>medfilt2</code>	2-D median filtering
<code>ordfilt2</code>	2-D order-statistic filtering
<code>stretchlim</code>	Find limits to contrast stretch image
<code>wiener2</code>	2-D adaptive noise-removal filtering

### Image Restoration (Deblurring)

<code>deconvblind</code>	Deblur image using blind deconvolution
<code>deconvlucy</code>	Deblur image using Lucy-Richardson method.

deconvreg

Deblur image using regularized filter

deconvwnr

Deblur image using Wiener filter

edgetaper

Taper discontinuities along image edges

otf2psf

Convert optical transfer function to point-spread function

psf2otf

Convert point-spread function to optical transfer function

# Linear Filtering and Transforms

Linear Filtering (p. 16-15)

Convolution, N-D filtering, and predefined 2-D filters

Linear 2-D Filter Design (p. 16-15)

2-D FIR filters

Image Transforms (p. 16-16)

Fourier, Discrete Cosine, Radon, and Fan-beam transforms

## Linear Filtering

`conv2`

2-D convolution

`convmtx2`

2-D convolution matrix

`convn`

N-D convolution

`filter2`

2-D linear filtering

`fspecial`

Create predefined 2-D filter

`imfilter`

N-D filtering of multidimensional images

## Linear 2-D Filter Design

`freqspace`

Determine frequency spacing for 2-D frequency response

`freqz2`

2-D frequency response

`fsamp2`

2-D FIR filter using frequency sampling

`ftrans2`

2-D FIR filter using frequency transformation

`fwind1`

2-D FIR filter using 1-D window method

`fwind2`

2-D FIR filter using 2-D window method

## Image Transforms

dct2	2-D discrete cosine transform
dctmtx	Discrete cosine transform matrix
fan2para	Convert fan-beam projections to parallel-beam
fanbeam	Fan-beam transform
fft2	2-D fast Fourier transform
fftn	N-D fast Fourier transform
fftshift	Shift zero-frequency component of fast Fourier transform to center of spectrum
idct2	2-D inverse discrete cosine transform
ifanbeam	Inverse fan-beam transform
ifft2	2-D inverse fast Fourier transform
ifftn	N-D inverse fast Fourier transform
iradon	Inverse Radon transform
para2fan	Convert parallel-beam projections to fan-beam
phantom	Create head phantom image
radon	Radon transform



## Morphological Operations

Intensity and Binary Images (p. 16-17)	Dilate, erode, reconstruct, and perform other morphological operations
Binary Images (p. 16-18)	Label, pack, and perform morphological operations on binary images
Structuring Element (STREL) Creation and Manipulation (p. 16-19)	Create and manipulate structuring elements for morphological operations

### Intensity and Binary Images

<code>conndef</code>	Create connectivity array
<code>imbothat</code>	Bottom-hat filtering
<code>imclearborder</code>	Suppress light structures connected to image border
<code>imclose</code>	Morphologically close image
<code>imdilate</code>	Dilate image
<code>imerode</code>	Erode image
<code>imextendedmax</code>	Extended-maxima transform
<code>imextendedmin</code>	Extended-minima transform
<code>imfill</code>	Fill image regions and holes
<code>imhmax</code>	H-maxima transform
<code>imhmin</code>	H-minima transform
<code>imimposemin</code>	Impose minima
<code>imopen</code>	Morphologically open image
<code>imreconstruct</code>	Morphological reconstruction
<code>imregionalmax</code>	Regional maxima

imregionalmin

Regional minima

imtophat

Top-hat filtering

watershed

Watershed transform

## Binary Images

applylut

Neighborhood operations on binary images using lookup tables

bwarea

Area of objects in binary image

bwareaopen

Morphologically open binary image (remove small objects)

bwdist

Distance transform of binary image

bweuler

Euler number of binary image

bwhitmiss

Binary hit-miss operation

bwlabel

Label connected components in binary image

bwlabeln

Label connected components in N-D binary image

bwmorph

Morphological operations on binary images

bwpack

Pack binary image

bwperim

Find perimeter of objects in binary image

bwselect

Select objects in binary image

bwulterode

Ultimate erosion

bwunpack

Unpack binary image

imregionalmin

Regional minima

imtophat

Top-hat filtering

makelut

Create lookup table for use with applylut

## Structuring Element (STREL) Creation and Manipulation

<code>getheight</code>	Height of structuring element
<code>getneighbors</code>	Structuring element neighbor locations and heights
<code>getnhood</code>	Structuring element neighborhood
<code>getsequence</code>	Sequence of decomposed structuring elements
<code>isflat</code>	True for flat structuring element
<code>reflect</code>	Reflect structuring element
<code>strel</code>	Create morphological structuring element (STREL)
<code>translate</code>	Translate structuring element (STREL)

## Region-Based, Neighborhood, and Block Processing

Region-Based Processing (p. 16-20)	Define regions of interest and perform operations on them
Neighborhood and Block Processing (p. 16-20)	Defining neighborhoods and blocks and processing them

### Region-Based Processing

<code>poly2mask</code>	Convert region-of-interest polygon to region mask
<code>roicolor</code>	Select region of interest (ROI) based on color
<code>roifill</code>	Fill in specified polygon in grayscale image
<code>roifilt2</code>	Filter region of interest in image
<code>roipoly</code>	Specify polygonal region of interest

### Neighborhood and Block Processing

<code>bestblk</code>	Determine optimal block size for block processing
<code>blkproc</code>	Distinct block processing for image
<code>col2im</code>	Rearrange matrix columns into blocks
<code>colfilt</code>	Columnwise neighborhood operations
<code>im2col</code>	Rearrange image blocks into columns
<code>nlfilter</code>	General sliding-neighborhood operations

## Colormap and Color Space Functions

Colormap Manipulation (p. 16-21)	Manipulate colormaps to brighten or change an image
Color Space Conversions (p. 16-21)	ICC profile-based device independent color space conversions and device-dependent color space conversions

### Colormap Manipulation

brighten	Brighten or darken colormap
cpermute	Rearrange colors in colormap
cmunique	Eliminate duplicate colors in colormap; convert grayscale or truecolor image to indexed image
imapprox	Approximate indexed image by one with fewer colors
rgbplot	Plot colormap

### Color Space Conversions

applycform	Apply color space transformation
hsv2rgb	Convert hue-saturation-value (HSV) values to RGB color space
iccfind	Search for ICC profiles
iccread	Read ICC profile
iccroot	Find system default ICC profile repository
iccwrite	Write ICC color profile to disk file
isicc	True for valid ICC color profile

<code>lab2double</code>	Convert $L^*a^*b^*$ data to double
<code>lab2uint16</code>	Convert $L^*a^*b^*$ data to uint16
<code>lab2uint8</code>	Convert $L^*a^*b^*$ data to uint8
<code>makecform</code>	Create color transformation structure
<code>ntsc2rgb</code>	Convert NTSC values to RGB color space
<code>rgb2hsv</code>	Convert RGB values to hue-saturation-value (HSV) color space
<code>rgb2ntsc</code>	Convert RGB color values to NTSC color space
<code>rgb2ycbcr</code>	Convert RGB color values to YCbCr color space
<code>whitepoint</code>	$XYZ$ color values of standard illuminants
<code>xyz2double</code>	Convert $XYZ$ color values to double
<code>xyz2uint16</code>	Convert $XYZ$ color values to uint16
<code>ycbcr2rgb</code>	Convert YCbCr color values to RGB color space

## Miscellaneous Functions

Toolbox Preferences (p. 16-23)	Set and determine the value of toolbox preferences
Toolbox Utility Functions (p. 16-23)	Check input arguments and perform other common programming tasks
Interactive Mouse Utility Functions (p. 16-24)	Retrieve the values of lines, points, and rectangles defined interactively using the mouse
Array Operations (p. 16-24)	Circularly shift pixel values and pad arrays
Demos (p. 16-24)	Launch Image Processing Toolbox demos
Performance (p. 16-24)	Check for presence of Intel Performance Primitives Library (IPPL)

### Toolbox Preferences

<code>iptgetpref</code>	Get value of Image Processing Toolbox preference
<code>iptsetpref</code>	Set Image Processing Toolbox preferences or display valid values

### Toolbox Utility Functions

<code>getrangefromclass</code>	Default display range of image based on its class
<code>iptcheckconn</code>	Check validity of connectivity argument
<code>iptcheckinput</code>	Check validity of array
<code>iptcheckmap</code>	Check validity of colormap

<code>iptchecknargin</code>	Check number of input arguments
<code>iptcheckstrs</code>	Check validity of option string
<code>iptnum2ordinal</code>	Convert positive integer to ordinal string

## Interactive Mouse Utility Functions

<code>getline</code>	Select polyline with mouse
<code>getpts</code>	Specify points with mouse
<code>getrect</code>	Specify rectangle with mouse

## Array Operations

<code>padarray</code>	Pad array
-----------------------	-----------

## Demos

<code>iptdemos</code>	Index of Image Processing Toolbox demos
-----------------------	---

## Performance

<code>ippl</code>	Check for presence of Intel Performance Primitives Library (IPPL)
-------------------	---



# Functions — Alphabetical List

---

# adapthisteq

---

**Purpose** Contrast-limited adaptive histogram equalization (CLAHE)

**Syntax**  
`J = adapthisteq(I)`  
`J = adapthisteq(I,param1,val1,param2,val2...)`

**Description** `J = adapthisteq(I)` enhances the contrast of the grayscale image `I` by transforming the values using contrast-limited adaptive histogram equalization (CLAHE).

CLAHE operates on small regions in the image, called *tiles*, rather than the entire image. Each tile's contrast is enhanced, so that the histogram of the output region approximately matches the histogram specified by the 'Distribution' parameter. The neighboring tiles are then combined using bilinear interpolation to eliminate artificially induced boundaries. The contrast, especially in homogeneous areas, can be limited to avoid amplifying any noise that might be present in the image.

`J = adapthisteq(I,param1,val1,param2,val2...)` specifies any of the additional parameter/value pairs listed in the following table. Parameter names can be abbreviated, and case does not matter.

Parameter	Value
'NumTiles'	Two-element vector of positive integers specifying the number of tiles by row and column, [M N]. Both M and N must be at least 2. The total number of tiles is equal to M*N. Default: [8 8]
'ClipLimit'	Real scalar in the range [0 1] that specifies a contrast enhancement limit. Higher numbers result in more contrast. Default: 0.01

Parameter	Value
'NBins'	<p>Positive integer scalar specifying the number of bins for the histogram used in building a contrast enhancing transformation. Higher values result in greater dynamic range at the cost of slower processing speed.</p> <p>Default: 256</p>
'Range'	<p>String specifying the range of the output image data.</p> <p>'original' — Range is limited to the range of the original image, <math>[\min(I(:)) \quad \max(I(:))]</math>.</p> <p>'full' — Full range of the output image class is used. For example, for uint8 data, range is [0 255].</p> <p>Default: 'full'</p>
'Distribution'	<p>String specifying the desired histogram shape for the image tiles.</p> <p>'uniform' — Flat histogram</p> <p>'rayleigh' — Bell-shaped histogram</p> <p>'exponential' — Curved histogram</p> <p>Default: 'uniform'</p>
'Alpha'	<p>Nonnegative real scalar specifying a distribution parameter.</p> <p>Default: 0.4</p> <hr/> <p><b>Note</b> Only used when 'Distribution' is set to either 'rayleigh' or 'exponential'.</p> <hr/>

## Remarks

- 'NumTiles' specifies the number of rectangular contextual regions (tiles) into which adapthisteq divides the image. adapthisteq calculates the contrast transform function for each of these regions individually. The optimal number of tiles depends on the type of the input image, and it is best determined through experimentation.
- 'ClipLimit' is a contrast factor that prevents over-saturation of the image specifically in homogeneous areas. These areas are characterized by a high peak in the histogram of the particular image tile due to many pixels falling inside the same gray level range. Without the clip limit, the adaptive histogram equalization technique could produce results that, in some cases, are worse than the original image.
- 'Distribution' specifies the distribution that adapthisteq uses as the basis for creating the contrast transform function. The distribution you select should depend on the type of the input image. For example, underwater imagery appears to look more natural when the Rayleigh distribution is used.

## Class Support

Grayscale image I can be of class uint8, uint16, int16, single, or double. The output image J has the same class as I.

## Examples

Apply Contrast-limited Adaptive Histogram Equalization (CLAHE) to an image and display the results.

```
I = imread('tire.tif');  
A = adapthisteq(I,'clipLimit',0.02,'Distribution','rayleigh');  
figure, imshow(I);  
figure, imshow(A);
```

Apply CLAHE to a color image.

```
[X MAP] = imread('shadow.tif');  
  
% Convert indexed image to true-color (RGB) format  
RGB = ind2rgb(X,MAP);
```

```
% Convert image to L*a*b* color space
cform2lab = makecform('srgb2lab');
LAB = applycform(IMG, cform2lab);

% Scale values to range from 0 to 1
L = LAB(:,:,1)/100;

% Perform CLAHE
LAB(:,:,1) = adapthisteq(L, 'NumTiles', ...
    [8 8], 'ClipLimit', 0.005)*100;

% Convert back to RGB color space
cform2srgb = makecform('lab2srgb');
J = applycform(LAB, cform2srgb);

% Display the results
figure, imshow(IMG);
figure, imshow(J);
```

**See Also** [histeq](#)

# analyze75info

---

**Purpose** Read metadata from header file of Analyze 7.5 data set

**Syntax**  
`info = analyze75info(filename)`  
`info = analyze75info(filename, 'ByteOrder', endian)`

**Description** `info = analyze75info(filename)` reads the header file of the Analyze 7.5 data set specified by the string `filename`. The function returns `info`, a structure whose fields contain information about the data set.

Analyze 7.5 is a 3-D biomedical image visualization and analysis product developed by the Biomedical Imaging Resource of the Mayo Clinic. An Analyze 7.5 data set is made of two files, a header file and an image file. The files have the same name with different file extensions. The header file has the file extension `.hdr` and the image file has the file extension `.img`. For more information about Analyze 7.5 format metadata returned in `info`, see the Mayo Clinic Web site.

`info = analyze75info(filename, 'ByteOrder', endian)` reads the Analyze 7.5 header file using the byte ordering specified by *endian*, where *endian* can have either of the following values:

Value	Meaning
'ieee-le'	Byte ordering is Little Endian
'ieee-be'	Byte ordering is Big Endian

If the specified *endian* value results in a read error, `analyze75info` issues a warning message and attempts to read the header file with the opposite `ByteOrder` format.

**Examples** Read an Analyze 7.5 header file. The file used in the example can be downloaded from <http://www.radiology.uiowa.edu/downloads/>.

```
info = analyze75info('CT_HAND.hdr');
```

Specify the byte ordering of the data set.

```
info = analyze75info('CT_HAND', 'ByteOrder', 'ieee-be');
```

## See Also

[analyze75read](#)

# analyze75read

---

**Purpose** Read image data from image file of Analyze 7.5 data set

**Syntax** `X = analyze75read(filename)`  
`X = analyze75read(info)`

**Description** `X = analyze75read(filename)` reads the image data from the image file of an Analyze 7.5 format data set specified by the string `filename`. The function returns the image data in `X`. For single-frame, grayscale images, `X` is an  $m$ -by- $n$  array. `analyze75read` uses a data type for `X` that is consistent with the data type specified in the data set header file.

Analyze 7.5 is a 3-D biomedical image visualization and analysis product developed by the Biomedical Imaging Resource of the Mayo Clinic. An Analyze 7.5 data set is made of two files, a header file and an image file. The files have the same name with different file extensions. The header file has the file extension `.hdr` and the image file has the file extension `.img`. For more information about the Analyze 7.5 format, see the Mayo Clinic Web site.

`X = analyze75read(info)` reads the image data from the image file specified in the metadata structure `info`. `info` must be a valid metadata structure returned by the `analyze75info` function.

---

**Note** `analyze75read` returns image data in radiological orientation (LAS). This is the default used by the Analyze 7.5 format.

---

## Examples

### Example 1

Read image data from an Analyze 7.5 image file. The file used in the example can be downloaded from <http://www.radiology.uiowa.edu/downloads/>.

```
X = analyze75read('CT_HAND');
```

Because Analyze 7.5 format uses radiological orientation (LAS), flip the data for correct image display in MATLAB.



```
X = flipdim(X,1);
```

Select frames 51 to 56 and use reshape to create an array for montage.

```
Y = reshape(X(:,:,51:56),[size(X,1) size(X,2) 1 6]);  
montage(Y);
```

## Example 2

Call analyze75read with the metadata obtained from the header file using analyze75info.

```
info = analyze75info('CT_HAND.hdr');  
X = analyze75read(info);
```

## Class Support

X can be logical, uint8, int16, int32, single, or double. Complex and RGB data types are not supported.

## See Also

analyze75info

# applycform

---

**Purpose** Apply color space transformation

**Syntax** `out = applycform(I,C)`

**Description** `out = applycform(I,C)` converts the color values in `I` to the color space specified in the color transformation structure `C`. The color transformation structure specifies various parameters of the transformation. See `makecform` for details.

If `I` is two-dimensional, each row is interpreted as a color. `I` typically has either three or four columns, depending on the input color space. `out` has the same number of rows and either three or four columns, depending on the output color space.

If `I` is three-dimensional, each row-column location is interpreted as a color, and `size(I,3)` is typically either three or four, depending on the input color space. `out` has the same number of rows and columns as `I`, and `size(out,3)` is either three or four, depending on the output color space.

**Class Support** `I` must be a real, nonsparse, finite array of class `uint8`, `uint16`, or `double`. The output array `out` has the same class and size as the input array, unless the output color space is `XYZ`. If the input is `XYZ` data of class `uint8`, the output is of class `uint16`, because there is no standard 8-bit encoding defined for `XYZ` color values.

**Examples** Read in a color image that uses the RGB color space.

```
I = imread('peppers.png');
```

Create a color transformation structure that defines an RGB to `XYZ` conversion.

```
C = makecform('srgb2xyz');
```

Perform the transformation with `applycform`.

```
I_xyz = applycform(I,C);
```

**See Also**

lab2double, lab2uint8, lab2uint16, makecform, whitepoint,  
xyz2double, xyz2uint16

For a full list of the toolbox color space conversion functions, see “Color Space Conversions” on page 16-21.

# applylut

---

**Purpose** 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.

**Class Support** `BW` can be numeric or logical, and it must be real, two-dimensional, and nonsparse. `LUT` can be numeric or logical, and it must be a real vector with 16 or 512 elements. If all the elements of `LUT` are 0 or 1, then `A` is logical. If all the elements of `LUT` are integers between 0 and 255, then `A` is `uint8`. For all other cases, `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.

$$\begin{array}{cc} 8 & 2 \\ 4 & 1 \end{array}$$

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 two 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.

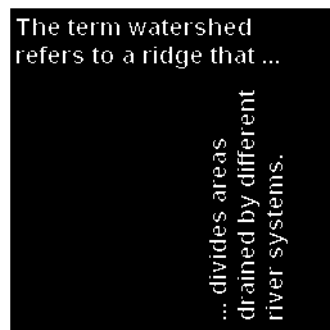
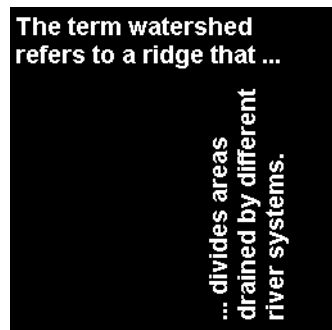
256	32	4
128	16	2
64	8	1

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.

### Examples

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);
BW = imread('text.png');
BW2 = applylut(BW,lut);
imshow(BW), figure, imshow(BW2)
```



# applylut

---

## See Also

`makelut`

<b>Purpose</b>	Convert axes coordinates to pixel coordinates
<b>Syntax</b>	<code>pixelx = axes2pix(dim, XDATA, AXESX)</code>
<b>Description</b>	<code>pixelx = axes2pix(dim, XDATA, AXESX)</code> converts an axes coordinate into a pixel coordinate. For example, if <code>pt = get(gca, 'CurrentPoint')</code> then <code>AXESX</code> could be <code>pt(1,1)</code> or <code>pt(1,2)</code> . <code>AXESX</code> must be in pixel coordinates. <code>XDATA</code> is a two-element vector returned by <code>get(image_handle, 'XData')</code> or <code>get(image_handle, 'YData')</code> . <code>dim</code> is the number of image columns for the $x$ coordinate, or the number of image rows for the $y$ coordinate.
<b>Class Support</b>	<code>dim</code> , <code>XDATA</code> , and <code>AXESX</code> can be double. The output is double.
<b>Note</b>	<code>axes2pix</code> performs minimal checking on the validity of <code>AXESX</code> , <code>DIM</code> , or <code>XDATA</code> . For example, <code>axes2pix</code> returns a negative coordinate if <code>AXESX</code> is less than <code>XDATA(1)</code> . The function calling <code>axes2pix</code> bears responsibility for error checking.
<b>Examples</b>	<p>Example with default <code>XData</code> and <code>YData</code>.</p> <pre>h = imshow('pout.tif'); [nrows,ncols] = size(get(h,'CData')); xdata = get(h,'XData') ydata = get(h,'YData') px = axes2pix(ncols,xdata,30) py = axes2pix(nrows,ydata,30)</pre> <p>Example with non-default <code>XData</code> and <code>YData</code>.</p> <pre>xdata = [10 100] ydata = [20 90] px = axes2pix(ncols,xdata,30) py = axes2pix(nrows,ydata,30)</pre>
<b>See Also</b>	<code>imixelinfo</code> , <code>bwselect</code> , <code>imfill</code> , <code>imixel</code> , <code>improfile</code> , <code>pixval</code> , <code>roipoly</code>

# bestblk

---

**Purpose** Determine optimal 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. The return value `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`.

**Examples**

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

siz =

    64    50
```

**See Also** `blkproc`



**Purpose**

Distinct block processing for image

**Syntax**

```
B = blkproc(A,[m n],fun)
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 0's if necessary. `fun` is a function handle that accepts an `m`-by-`n` matrix, `x`, and returns 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],[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- $(n+2*nborder)$ . The `blkproc` function pads the border with 0's, if necessary, on the edges of `A`. The function `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 0's if the class of `A` is `uint8` or `uint16`, or 1's if the class of `A` is `double`.

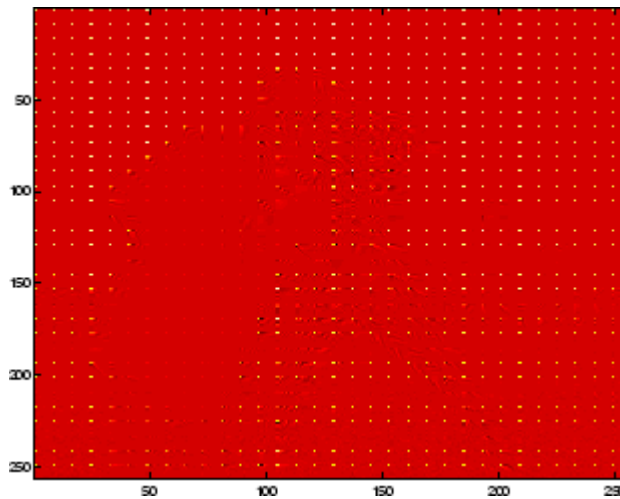
**Class Support**

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`.

## Examples

Compute the 2-D DCT of each 8-by-8 block to the standard deviation of the elements in that block. In this example, fun is specified as a function handle created using @.

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



Set the pixels in each 16-by-16 block to the standard deviation of the elements in that block. In this example, fun is specified as an anonymous function.

```
I = imread('liftingbody.png');  
fun = @(x) std2(x)*ones(size(x));  
I2 = blkproc(I,[32 32],fun);  
imshow(I), figure, imshow(I2,'DisplayRange',[])
```

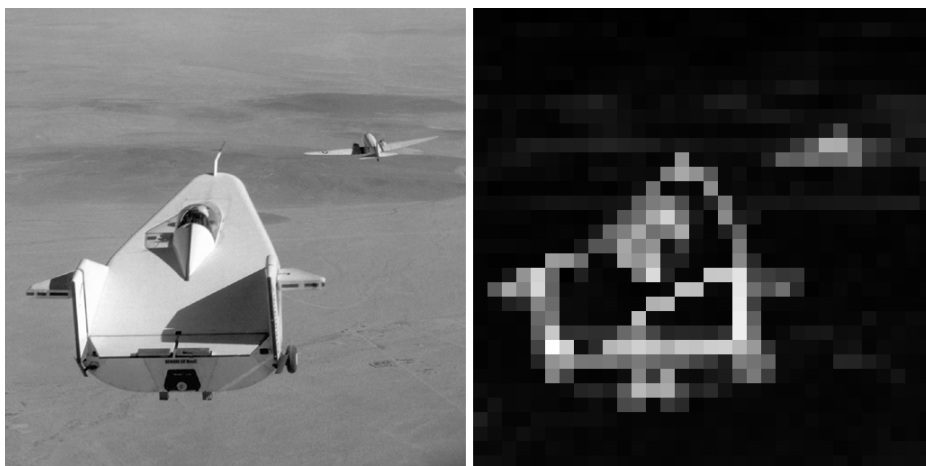


Image Courtesy of NASA

**See Also**

`bestblk`, `colfilt`, `nlfilter`, `function_handle`

# brighten

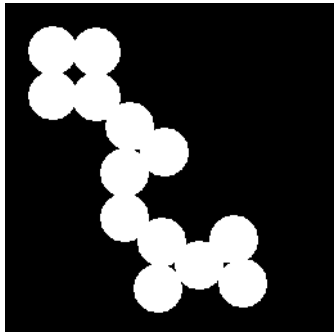
---

**Purpose**            Brighten or darken colormap

**Note**              brighten is a MATLAB function.

---

<b>Purpose</b>	Area of objects in binary image
<b>Syntax</b>	<code>total = bwarea(BW)</code>
<b>Description</b>	<code>total = bwarea(BW)</code> estimates the area of the objects in binary image BW. <code>total</code> is a scalar whose value corresponds roughly to the total number of on pixels in the image, but might not be exactly the same because different patterns of pixels are weighted differently.
<b>Class Support</b>	BW can be numeric or logical. For numeric input, any nonzero pixels are considered to be on. The return value <code>total</code> is of class <code>double</code> .
<b>Algorithm</b>	<p><code>bwarea</code> estimates the area of all of the on pixels in an image by summing the areas of each pixel in the image. The area of an individual pixel is determined by looking at its 2-by-2 neighborhood. There are six different patterns, each representing a different area:</p> <ul style="list-style-type: none"><li>• Patterns with zero on pixels (area = 0)</li><li>• Patterns with one on pixel (area = 1/4)</li><li>• Patterns with two adjacent on pixels (area = 1/2)</li><li>• Patterns with two diagonal on pixels (area = 3/4)</li><li>• Patterns with three on pixels (area = 7/8)</li><li>• Patterns with all four on pixels (area = 1)</li></ul> <p>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.</p>
<b>Examples</b>	<p>Compute the area in the objects of a 256-by-256 binary image.</p> <pre>BW = imread('circles.png'); imshow(BW);</pre>



```
bwarea(BW)
```

```
ans =
```

```
1.4187e+004
```

## See Also

bweuler, bwperim

## References

[1] Pratt, William K., *Digital Image Processing*, New York, John Wiley & Sons, Inc., 1991, p. 634.

**Purpose** Morphologically open binary image (remove small objects)

**Syntax**  
BW2 = bwareaopen(BW,P)  
BW2 = bwareaopen(BW,P,conn)

**Description** BW2 = bwareaopen(BW,P) removes from a binary image all connected components (objects) that have fewer than P pixels, producing another binary image, BW2. The default connectivity is 8 for two dimensions, 26 for three dimensions, and `conndef(ndims(BW),'maximal')` for higher dimensions.

BW2 = bwareaopen(BW,P,conn) specifies the desired connectivity. conn can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can be defined in a more general way for any dimension by using for conn a 3-by-3-by-...-by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood locations relative to the center element of conn. Note that conn must be symmetric about its center element.

**Class Support** BW can be a logical or numeric array of any dimension, and it must be nonsparse. The return value BW2 is of class logical.

## Algorithm

The basic steps are

- 1 Determine the connected components.

```
L = bwlabeln(BW, conn);
```

- 2 Compute the area of each component.

```
S = regionprops(L, 'Area');
```

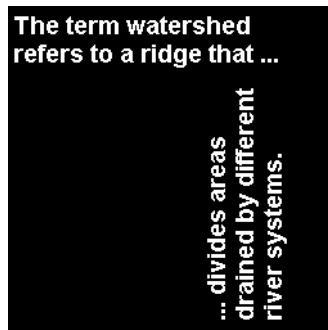
- 3 Remove small objects.

```
bw2 = ismember(L, find([S.Area] >= P));
```

## Examples

Read in the image and display it.

```
originalBW = imread('text.png');  
imshow(originalBW)
```



Remove all objects smaller than 50 pixels. Note the missing letters.

```
bwAreaOpenBW = bwareaopen(originalBW,50);  
figure, imshow(bwAreaOpenBW)
```





## See Also

`bwlabel`, `bwlabeln`, `conndef`, `regionprops`

# bwboundaries

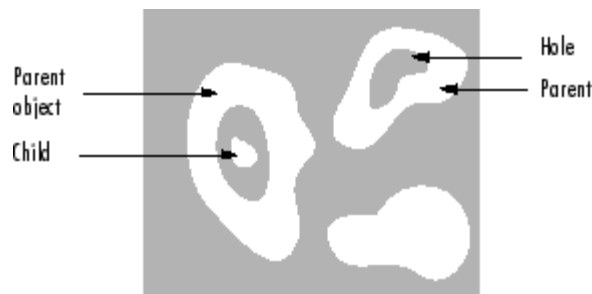
---

**Purpose** Trace region boundaries in binary image

**Syntax**

```
B = bwboundaries(BW)
B = bwboundaries(BW,conn)
B = bwboundaries(BW,conn,options)
[B L] = bwboundaries(...)
[B L N A] = bwboundaries()
```

**Description** `B = bwboundaries(BW)` traces the exterior boundaries of objects, as well as boundaries of holes inside these objects, in the binary image `BW`. `bwboundaries` also descends into the outermost objects (parents) and traces their children (objects completely enclosed by the parents). `BW` must be a binary image where nonzero pixels belong to an object and 0 pixels constitute the background. The following figure illustrates these components.



`bwboundaries` returns `B`, a `P`-by-1 cell array, where `P` is the number of objects and holes. Each cell in the cell array contains a `Q`-by-2 matrix. Each row in the matrix contains the row and column coordinates of a boundary pixel. `Q` is the number of boundary pixels for the corresponding region.

`B = bwboundaries(BW,conn)` specifies the connectivity to use when tracing parent and child boundaries. `conn` can have either of the following scalar values.

Value	Meaning
4	4-connected neighborhood
8	8-connected neighborhood. This is the default.

`B = bwboundaries(BW,conn,options)` specifies an optional argument, where `options` can have either of the following values:

Value	Meaning
'holes'	Search for both object and hole boundaries. This is the default.
'noholes'	Search only for object (parent and child) boundaries. This can provide better performance.

`[B,L] = bwboundaries(...)` returns the label matrix `L` as the second output argument. Objects and holes are labeled. `L` is a two-dimensional array of nonnegative integers that represent contiguous regions. The `k`th region includes all elements in `L` that have value `k`. The number of objects and holes represented by `L` is equal to `max(L(:))`. The zero-valued elements of `L` make up the background.

`[B,L,N,A] = bwboundaries(...)` returns `N`, the number of objects found, and `A`, an adjacency matrix. The first `N` cells in `B` are object boundaries. `A` represents the parent-child-hole dependencies. `A` is a square, sparse, logical matrix with side of length `max(L(:))`, whose rows and columns correspond to the positions of boundaries stored in `B`.

The boundaries enclosed by a `B{m}` as well as the boundary enclosing `B{m}` can both be found using `A` as follows:

```
enclosing_boundary = find(A(m,:));
enclosed_boundaries = find(A(:,m));
```

## Class Support

`BW` can be logical or numeric and it must be real, two-dimensional, and nonsparse. `L` and `N` are double. `A` is sparse logical.

## Examples

### Example 1

Read in and threshold an intensity image. Display the labeled objects using the jet colormap, on a gray background, with region boundaries outlined in white.

```
I = imread('rice.png');
BW = im2bw(I, graythresh(I));
[B,L] = bwboundaries(BW,'noholes');
imshow(label2rgb(L, @jet, [.5 .5 .5]))
hold on
for k = 1:length(B)
    boundary = B{k};
    plot(boundary(:,2), boundary(:,1), 'w', 'LineWidth', 2)
end
```

### Example 2

Read in and display a binary image. Overlay the region boundaries on the image. Display text showing the region number (based on the label matrix) next to every boundary. Additionally, display the adjacency matrix using the MATLAB spy function.

After the image is displayed, use the zoom tool to read individual labels.

```
BW = imread('blobs.png');
[B,L,N,A] = bwboundaries(BW);
figure, imshow(BW); hold on;
colors=['b' 'g' 'r' 'c' 'm' 'y'];
for k=1:length(B)
    boundary = B{k};
    cidx = mod(k,length(colors))+1;
    plot(boundary(:,2), boundary(:,1),...
        colors(cidx),'LineWidth',2);
    %randomize text position for better visibility
    rndRow = ceil(length(boundary)/(mod(rand*k,7)+1));
    col = boundary(rndRow,2); row = boundary(rndRow,1);
    h = text(col+1, row-1, num2str(L(row,col)));
    set(h,'Color',colors(cidx),...
```

```
        'FontSize',14,'FontWeight','bold');  
    end  
    figure; spy(A);
```

### Example 3

Display object boundaries in red and hole boundaries in green.

```
BW = imread('blobs.png');  
[B,L,N] = bwboundaries(BW);  
figure; imshow(BW); hold on;  
for k=1:length(B),  
    boundary = B{k};  
    if(k > N)  
        plot(boundary(:,2),...  
             boundary(:,1),'g','LineWidth',2);  
    else  
        plot(boundary(:,2),...  
             boundary(:,1),'r','LineWidth',2);  
    end  
end
```

### Example 4

Display parent boundaries in red (any empty row of the adjacency matrix belongs to a parent) and their holes in green.

```
BW = imread('blobs.png');  
[B,L,N,A] = bwboundaries(BW);  
figure; imshow(BW); hold on;  
for k=1:length(B),  
    if(~sum(A(k,:)))  
        boundary = B{k};  
        plot(boundary(:,2),...  
             boundary(:,1),'r','LineWidth',2);  
        for l=find(A(:,k))'  
            boundary = B{l};  
            plot(boundary(:,2),...  
                 boundary(:,1),'g','LineWidth',2);  
        end  
    end  
end
```

# bwboundaries

---

```
        end
    end
end
```

**See Also** [bwlabel](#), [bwlabeln](#), [bwperim](#), [bwtraceboundary](#)

**Purpose** Distance transform of binary image

**Syntax**

```
D = bwdist(BW)
[D,L] = bwdist(BW)
[D,L] = bwdist(BW,method)
```

**Description** `D = bwdist(BW)` computes the Euclidean distance transform of the binary image `BW`. For each pixel in `BW`, the distance transform assigns a number that is the distance between that pixel and the nearest nonzero pixel of `BW`. `bwdist` uses the Euclidean distance metric by default. `BW` can have any dimension. `D` is the same size as `BW`.

`[D,L] = bwdist(BW)` also computes the nearest-neighbor transform and returns it as label matrix `L`, which has the same size as `BW` and `D`. Each element of `L` contains the linear index of the nearest nonzero pixel of `BW`.

`[D,L] = bwdist(BW,method)` computes the distance transform, where `method` specifies an alternate distance metric. `method` can take any of these values:

Method	Description
'chessboard'	In 2-D, the chessboard distance between $(x_1, y_1)$ and $(x_2, y_2)$ is $\max( x_1 - x_2 ,  y_1 - y_2 )$
'cityblock'	In 2-D, the cityblock distance between $(x_1, y_1)$ and $(x_2, y_2)$ is $ x_1 - x_2  +  y_1 - y_2 $

Method	Description
'euclidean'	<p>In 2-D, the Euclidean distance between <math>(x_1, y_1)</math> and <math>(x_2, y_2)</math> is</p> $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ <p>This is the default method.</p>
'quasi-euclidean'	<p>In 2-D, the quasi-Euclidean distance between <math>(x_1, y_1)</math> and <math>(x_2, y_2)</math> is</p> $ x_1 - x_2  + (\sqrt{2} - 1) y_1 - y_2 ,  x_1 - x_2  >  y_1 - y_2 $ $(\sqrt{2} - 1) x_1 - x_2  +  y_1 - y_2 , \textit{otherwise}$

The *method* string can be abbreviated.

---

**Note** `bwdist` uses fast algorithms to compute the true Euclidean distance transform, especially in the 2-D case. The other methods are provided primarily for pedagogical reasons. However, the alternative distance transforms are sometimes significantly faster for multidimensional input images, particularly those that have many nonzero elements.

---

## Class Support

BW can be numeric or logical, and it must be nonsparse. D and L are double matrices with the same size as BW.

## Examples

Compute the Euclidean distance transform.

```
bw = zeros(5,5); bw(2,2) = 1; bw(4,4) = 1
bw =
    0    0    0    0    0
    0    1    0    0    0
    0    0    0    0    0
```



```

    0    0    0    1    0
    0    0    0    0    0

```

```
[D,L] = bwdist(bw)
```

```

D =
    1.4142    1.0000    1.4142    2.2361    3.1623
    1.0000         0    1.0000    2.0000    2.2361
    1.4142    1.0000    1.4142    1.0000    1.4142
    2.2361    2.0000    1.0000         0    1.0000
    3.1623    2.2361    1.4142    1.0000    1.4142

```

```

L =
     7     7     7     7     7
     7     7     7     7    19
     7     7     7    19    19
     7     7    19    19    19
     7    19    19    19    19

```

In the nearest-neighbor matrix L the values 7 and 19 represent the position of the nonzero elements using linear matrix indexing. If a pixel contains a 7, its closest nonzero neighbor is at linear position 7.

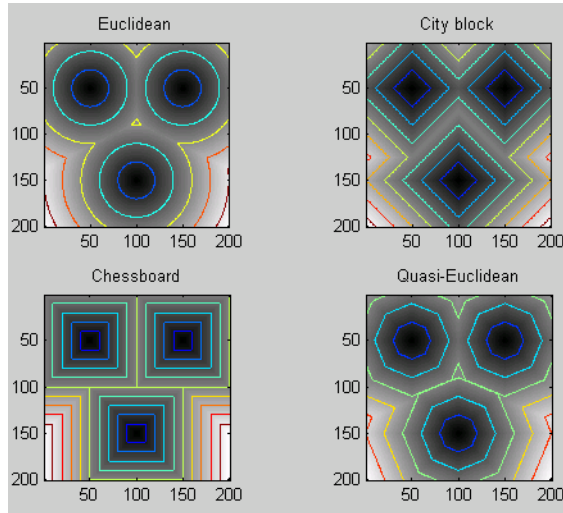
Compare the 2-D distance transforms for each of the supported distance methods. In the figure, note how the quasi-Euclidean distance transform best approximates the circular shape achieved by the Euclidean distance method.

```

bw = zeros(200,200); bw(50,50) = 1; bw(50,150) = 1;
bw(150,100) = 1;
D1 = bwdist(bw,'euclidean');
D2 = bwdist(bw,'cityblock');
D3 = bwdist(bw,'chessboard');
D4 = bwdist(bw,'quasi-euclidean');
figure
subplot(2,2,1), subimage(mat2gray(D1)), title('Euclidean')
hold on, imcontour(D1)
subplot(2,2,2), subimage(mat2gray(D2)), title('City block')

```

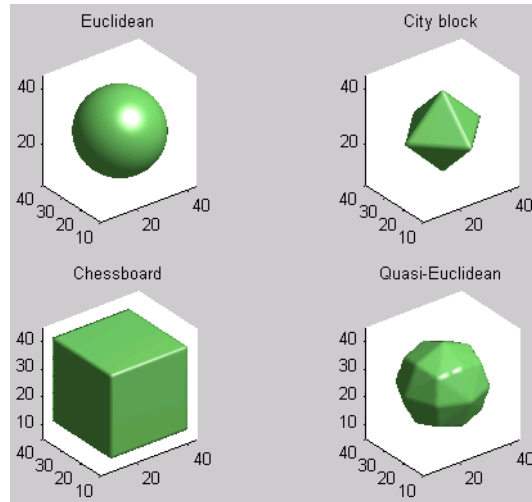
```
hold on, imcontour(D2)
subplot(2,2,3), subimage(mat2gray(D3)), title('Chessboard')
hold on, imcontour(D3)
subplot(2,2,4), subimage(mat2gray(D4)), title('Quasi-Euclidean')
hold on, imcontour(D4)
```



Compare isosurface plots for the distance transforms of a 3-D image containing a single nonzero pixel in the center.

```
bw = zeros(50,50,50); bw(25,25,25) = 1;
D1 = bwdist(bw);
D2 = bwdist(bw,'cityblock');
D3 = bwdist(bw,'chessboard');
D4 = bwdist(bw,'quasi-euclidean');
figure
subplot(2,2,1), isosurface(D1,15), axis equal, view(3)
camlight, lighting gouraud, title('Euclidean')
subplot(2,2,2), isosurface(D2,15), axis equal, view(3)
camlight, lighting gouraud, title('City block')
subplot(2,2,3), isosurface(D3,15), axis equal, view(3)
camlight, lighting gouraud, title('Chessboard')
```

```
subplot(2,2,4), isosurface(D4,15), axis equal, view(3)
camlight, lighting gouraud, title('Quasi-Euclidean')
```



## Algorithm

For two-dimensional Euclidean distance transforms, `bwdist` uses the second algorithm described in

[1] Brey, Heinz, Joseph Gil, David Kirkpatrick, and Michael Werman, "Linear Time Euclidean Distance Transform Algorithms," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 17, No. 5, May 1995, pp. 529-533.

For higher dimensional Euclidean distance transforms, `bwdist` uses a nearest-neighbor search on an optimized kd-tree, as described in

[1] Friedman, Jerome H., Jon Louis Bentley, and Raphael Ari Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Transactions on Mathematics Software*, Vol. 3, No. 3, September 1977, pp. 209-226.

For cityblock, chessboard, and quasi-Euclidean distance transforms, `bwdist` uses the two-pass, sequential scanning algorithm described in

[1] Rosenfeld, A. and J. Pfaltz, "Sequential operations in digital picture processing," *Journal of the Association for Computing Machinery*, Vol. 13, No. 4, 1966, pp. 471-494.

The different distance measures are achieved by using different sets of weights in the scans, as described in

[1] Paglieroni, David, "Distance Transforms: Properties and Machine Vision Applications," *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, Vol. 54, No. 1, January 1992, pp. 57-58.

## See Also

watershed

**Purpose** Euler number of binary image

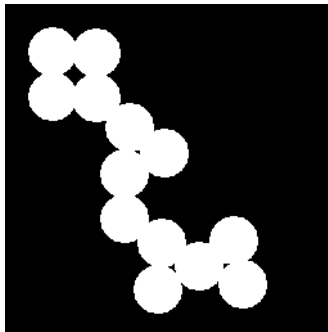
**Syntax** `eul = bweuler(BW,n)`

**Description** `eul = bweuler(BW,n)` returns the Euler number for the binary image `BW`. The return value `eul` is a scalar whose value is the total number of objects in the image minus the total number of holes in those objects. The argument `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 numeric or logical and it must be real, nonsparse, and two-dimensional. The return value `eul` is of class `double`.

**Examples**

```
BW = imread('circles.png');  
imshow(BW);
```



```
bweuler(BW)
```

```
ans =
```

```
-3
```

# bweuler

---

## 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.

<b>Purpose</b>	Binary hit-miss operation
<b>Syntax</b>	<pre>BW2 = bwhitmiss(BW1,SE1,SE2) BW2 = bwhitmiss(BW1,INTERVAL)</pre>
<b>Description</b>	<p><code>BW2 = bwhitmiss(BW1,SE1,SE2)</code> performs the hit-miss operation defined by the structuring elements <code>SE1</code> and <code>SE2</code>. The hit-miss operation preserves pixels whose neighborhoods match the shape of <code>SE1</code> and don't match the shape of <code>SE2</code>. <code>SE1</code> and <code>SE2</code> can be flat structuring element objects, created by <code>strel</code>, or neighborhood arrays. The neighborhoods of <code>SE1</code> and <code>SE2</code> should not have any overlapping elements. The syntax <code>bwhitmiss(BW1,SE1,SE2)</code> is equivalent to <code>imerode(BW1,SE1) &amp; imerode(~BW1,SE2)</code>.</p> <p><code>BW2 = bwhitmiss(BW1,INTERVAL)</code> performs the hit-miss operation defined in terms of a single array, called an <i>interval</i>. An interval is an array whose elements can contain 1, 0, or -1. The 1-valued elements make up the domain of <code>SE1</code>, the -1-valued elements make up the domain of <code>SE2</code>, and the 0-valued elements are ignored. The syntax <code>bwhitmiss(INTERVAL)</code> is equivalent to <code>bwhitmiss(BW1,INTERVAL == 1, INTERVAL == -1)</code>.</p>
<b>Class Support</b>	<p><code>BW1</code> can be a logical or numeric array of any dimension, and it must be nonsparse. <code>BW2</code> is always a logical array the same size as <code>BW1</code>. <code>SE1</code> and <code>SE2</code> must be flat STREL objects or they must be logical or numeric arrays containing 1's and 0's. <code>INTERVAL</code> must be an array containing 1's, 0's, and -1's.</p>
<b>Examples</b>	<p>Perform the hit-miss operation on a binary image using an interval.</p> <pre>bw = [0 0 0 0 0 0       0 0 1 1 0 0       0 1 1 1 1 0       0 1 1 1 1 0       0 0 1 1 0 0       0 0 1 0 0 0]</pre>

# bwhitmiss

---

```
interval = [0 -1 -1  
           1  1 -1  
           0  1  0];
```

```
bw2 = bwhitmiss(bw,interval)
```

```
bw2 =
```

```
    0    0    0    0    0    0  
    0    0    0    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
```

## See Also

`imdilate`, `imerode`, `strel`



---

<b>Purpose</b>	Label connected components in binary image
<b>Syntax</b>	<pre>L = bwlabel(BW,n) [L,num] = bwlabel(BW,n)</pre>
<b>Description</b>	<p><code>L = bwlabel(BW,n)</code> returns a matrix <code>L</code>, of the same size as <code>BW</code>, containing labels for the connected objects in <code>BW</code>. <code>n</code> can have a value of either 4 or 8, where 4 specifies 4-connected objects and 8 specifies 8-connected objects; if the argument is omitted, it defaults to 8.</p> <p>The elements of <code>L</code> 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.</p> <p><code>[L,num] = bwlabel(BW,n)</code> returns in <code>num</code> the number of connected objects found in <code>BW</code>.</p>
<b>Remarks</b>	<p><code>bwlabel</code> supports 2-D inputs only; <code>bwlabeln</code> supports inputs of any dimension. In some cases, you might prefer to use <code>bwlabeln</code> even for 2-D problems because it can be faster. If you have a 2-D input whose objects are relatively thick in the vertical direction, <code>bwlabel</code> is probably faster; otherwise <code>bwlabeln</code> is probably faster.</p>
<b>Class Support</b>	<p><code>BW</code> can be logical or numeric, and it must be real, two-dimensional, and nonsparse. <code>L</code> is of class <code>double</code>.</p>
<b>Remarks</b>	<p>You can use the MATLAB <code>find</code> function in conjunction with <code>bwlabel</code> to return vectors of indices for the pixels that make up a specific object. For example, to return the coordinates for the pixels in object 2,</p> <pre>[r,c] = find(bwlabel(BW)==2)</pre> <p>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. See <code>label2rgb</code> for more information.</p>

# bwlabel

---

## Examples

Label components 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
```

**Algorithm**

`bwlabel` uses the general procedure outlined in reference [1], pp. 40-48:

- 1** Run-length encode the input image.
- 2** Scan the runs, assigning preliminary labels and recording label equivalences in a local equivalence table.
- 3** Resolve the equivalence classes.
- 4** Relabel the runs based on the resolved equivalence classes.

**See Also**

`bweuler`, `bwlabeln`, `bwselect`, `label2rgb`

**Reference**

[1] Haralick, Robert M., and Linda G. Shapiro, *Computer and Robot Vision, Volume I*, Addison-Wesley, 1992, pp. 28-48.

# bwlabeln

---

**Purpose** Label connected components in N-D binary image

**Syntax**

```
L = bwlabeln(BW)
[L,NUM] = bwlabeln(BW)
[L,NUM] = bwlabeln(BW,conn)
```

**Description** L = bwlabeln(BW) returns a label matrix L containing labels for the connected components in BW. BW can have any dimension; L is the same size as BW. 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. The default connectivity is 8 for two dimensions, 26 for three dimensions, and conndef(ndims(BW), 'maximal') for higher dimensions.

[L,NUM] = bwlabeln(BW) returns in NUM the number of connected objects found in BW.

[L,NUM] = bwlabeln(BW,conn) specifies the desired connectivity. conn can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can also be defined in a more general way for any dimension by using for conn a 3-by-3-by-...-by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood locations relative to the center element of conn. Note that conn must be symmetric about its center element.

**Remarks**

bwlabel supports 2-D inputs only; bwlableIn supports inputs of any dimension. In some cases, you might prefer to use bwlableIn even for 2-D problems because it can be faster. If you have a 2-D input whose objects are relatively thick in the vertical direction, bwlabel is probably faster; otherwise bwlableIn is probably faster.

**Class Support**

BW can be numeric or logical, and it must be real and nonsparse. L is of class double.

**Examples**

```
BW = cat(3,[1 1 0; 0 0 0; 1 0 0],...  
[0 1 0; 0 0 0; 0 1 0],...  
[0 1 1; 0 0 0; 0 0 1])
```

```
bwlableIn(BW)
```

```
ans(:,:,1) =
```

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

```
ans(:,:,2) =
```

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

```
ans(:,:,3) =
```

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

# bwlabeln

---

## Algorithm

bwlabeln uses the following general procedure:

- 1** Scan all image pixels, assigning preliminary labels to nonzero pixels and recording label equivalences in a union-find table.
- 2** Resolve the equivalence classes using the union-find algorithm [1].
- 3** Relabel the pixels based on the resolved equivalence classes.

## See Also

bwlabel, label2rgb

## Reference

[1] Sedgewick, Robert, *Algorithms in C*, 3rd Ed., Addison-Wesley, 1998, pp. 11-20.

**Purpose** Morphological operations on binary images

**Syntax**  
 BW2 = bwmorph(BW,operation)  
 BW2 = bwmorph(BW,operation,n)

**Description** BW2 = bwmorph(BW,operation) applies a specific morphological operation to the binary image BW.  
 BW2 = bwmorph(BW,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.

Operation	Description
'bothat'	Performs <i>the morphological</i> “bottom hat” operation, which is <i>closing</i> (dilation followed by erosion) and subtracts the original image.
'bridge'	Bridges unconnected pixels, that is, sets 0-valued pixels to 1 if they have two nonzero neighbors that are not connected. For example:  <pre> 1 0 0      1 1 0 1 0 1  becomes 1 1 1 0 0 1      0 1 1 </pre>
'clean'	Removes <i>isolated</i> pixels (individual 1's that are surrounded by 0's), such as the center pixel in this pattern.  <pre> 0 0 0 0 1 0 0 0 0 </pre>
'close'	Performs morphological closing (dilation followed by erosion).

Operation	Description
'diag'	<p>Uses diagonal fill to eliminate 8-connectivity of the background. For example:</p> <pre> 0 1 0      0 1 0 1 0 0  becomes  1 1 0 0 0 0      0 0 0 </pre>
'dilate'	Performs dilation using the structuring element ones(3).
'erode'	Performs erosion using the structuring element ones(3).
'fill'	<p>Fills isolated interior pixels (individual 0's that are surrounded by 1's), such as the center pixel in this pattern.</p> <pre> 1 1 1 1 0 1 1 1 1 </pre>
'hbreak'	<p>Removes H-connected pixels. For example:</p> <pre> 1 1 1      1 1 1 0 1 0  becomes  0 0 0 1 1 1      1 1 1 </pre>
'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'	Performs morphological opening (erosion followed by dilation).
'remove'	Removes interior pixels. This option sets a pixel to 0 if all its 4-connected neighbors are 1, thus leaving only the boundary pixels on.



<b>Operation</b>	<b>Description</b>
'shrink'	With $n = \text{Inf}$ , 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 $n = \text{Inf}$ , 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:  <pre> 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 </pre>
'thicken'	With $n = \text{Inf}$ , thickens objects by adding pixels to the exterior of objects until doing so would result in previously unconnected objects being 8-connected. This option preserves the Euler number.
'thin'	With $n = \text{Inf}$ , thins objects to lines. It removes pixels so that an object without holes shrinks to a minimally connected stroke, and an object with holes shrinks to a connected ring halfway between each hole and the outer boundary. This option preserves the Euler number. See “Algorithm” on page 17-51 for more detail.
'tophat'	Performs morphological "top hat" operation, returning the image minus the morphological opening of the image.

**Class Support**

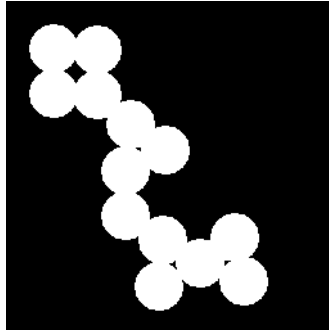
The input image BW can be numeric or logical. It must be 2-D, real and nonspare. The output image BW2 is of class logical.

# bwmorph

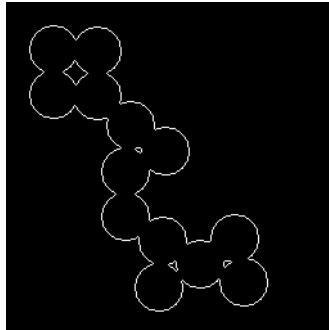
---

## Examples

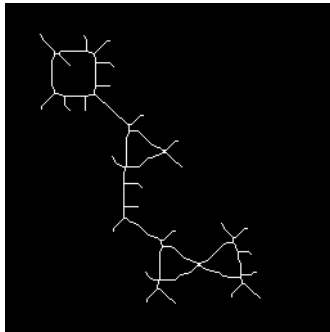
```
BW = imread('circles.png');  
imshow(BW);
```



```
BW2 = bwmorph(BW, 'remove');  
figure, imshow(BW2)
```



```
BW3 = bwmorph(BW, 'skel', Inf);  
figure, imshow(BW3)
```



## See Also

bweuler, bwperim, imdilate, imerode

## 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.

[3] Lam, L., Seong-Whan Lee, and Ching Y. Suen, "Thinning Methodologies-A Comprehensive Survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol 14, No. 9, September 1992, page 879, bottom of first column through top of second column.

See "Algorithm" on page 17-51 for more detail.

## Algorithm

When used with the 'thin' option, bwmorph uses the following algorithm (References [3]):

- 1 Divide the image into two distinct subfields in a checkerboard pattern.
- 2 In the first subiteration, delete pixel  $p$  from the first subfield if and only if the conditions  $G_1$ ,  $G_2$ , and  $G_3$  are all satisfied.
- 3 In the second subiteration, delete pixel  $p$  from the second subfield if and only if the conditions  $G_1$ ,  $G_2$ , and  $G_3'$  are all satisfied.

**Condition G1:**

$$X_H(p) = 1$$

where

$$X_H(p) = \sum_{i=1}^4 b_i$$

$$b_i = \begin{cases} 1 & \text{if } x_{2i-1} = 0 \text{ and } (x_{2i} = 1 \text{ or } x_{2i+1} = 1) \\ 0 & \text{otherwise} \end{cases}$$

$x_1, x_2, \dots, x_8$  are the values of the eight neighbors of  $p$ , starting with the east neighbor and numbered in counter-clockwise order.

**Condition G2:**

$$2 \leq \min\{n_1(p), n_2(p)\} \leq 3$$

where

$$n_1(p) = \sum_{k=1}^4 x_{2k-1} \vee x_{2k}$$

$$n_2(p) = \sum_{k=1}^4 x_{2k} \vee x_{2k+1}$$

**Condition G3:**

$$(x_2 \vee x_3 \vee \bar{x}_8) \wedge x_1 = 0$$

**Condition G3':**

$$(x_6 \vee x_7 \vee \bar{x}_4) \wedge x_5 = 0$$

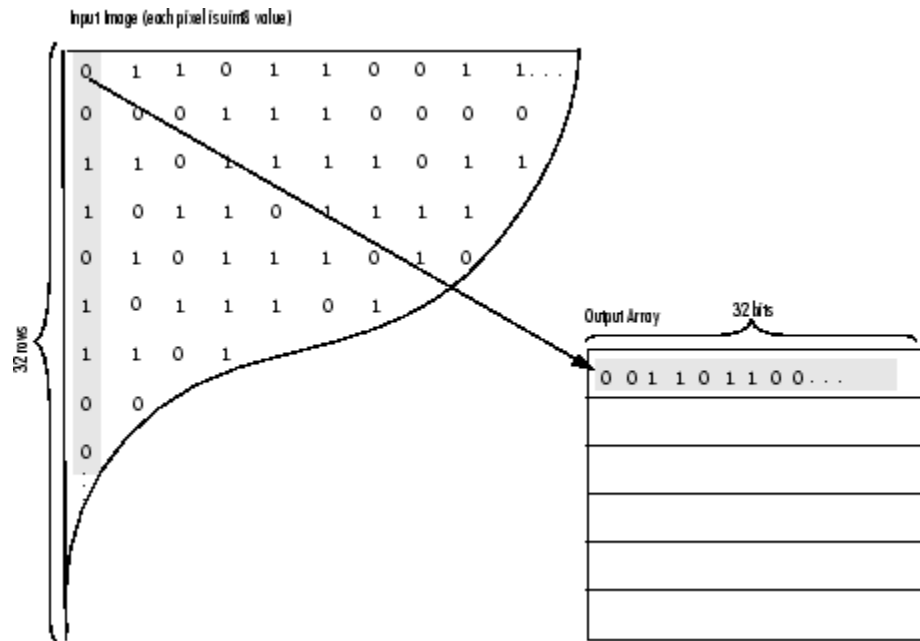
The two subiterations together make up one iteration of the thinning algorithm. When the user specifies an infinite number of iterations ( $n=\text{Inf}$ ), the iterations are repeated until the image stops changing. The conditions are all tested using `applylut` with precomputed lookup tables.

**Purpose** Pack binary image

**Syntax** BWP = bwpack(BW)

**Description** BWP = bwpack(BW) packs the uint8 binary image BW into the uint32 array BWP, which is known as a *packed binary image*. Because each 8-bit pixel value in the binary image has only two possible values, 1 and 0, bwpack can map each pixel to a single bit in the packed output image.

bwpack processes the image pixels by column, mapping groups of 32 pixels into the bits of a uint32 value. The first pixel in the first row corresponds to the least significant bit of the first uint32 element of the output array. The first pixel in the 32nd input row corresponds to the most significant bit of this same element. The first pixel of the 33rd row corresponds to the least significant bit of the second output element, and so on. If BW is M-by-N, then BWP is  $\text{ceil}(M/32)$ -by-N. This figure illustrates how bwpack maps the pixels in a binary image to the bits in a packed binary image.



Binary image packing is used to accelerate some binary morphological operations, such as dilation and erosion. If the input to `imdilate` or `imerode` is a packed binary image, the functions use a specialized routine to perform the operation faster.

`bwunpack` is used to unpack packed binary images.

## Class Support

BW can be logical or numeric, and it must be 2-D, real, and nonsparse. BWP is of class `uint32`.

## Examples

Pack, dilate, and unpack a binary image:

```
BW = imread('text.png');  
BWP = bwpack(BW);  
BWP_dilated = imdilate(BWP,ones(3,3),'ispacked');  
BW_dilated = bwunpack(BWP_dilated, size(BW,1));
```

**See Also**      `bwunpack, imdilate, imerode`

# bwperim

---

**Purpose** Find perimeter of objects in binary image

**Syntax**  
BW2 = bwperim(BW1)  
BW2 = bwperim(BW1,conn)

**Description** BW2 = bwperim(BW1) returns a binary image containing only the perimeter pixels of objects in the input image BW1. A pixel is part of the perimeter if it is nonzero and it is connected to at least one zero-valued pixel. The default connectivity is 4 for two dimensions, 6 for three dimensions, and conndef(ndims(BW), 'minimal') for higher dimensions.

BW2 = bwperim(BW1,conn) specifies the desired connectivity. conn can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

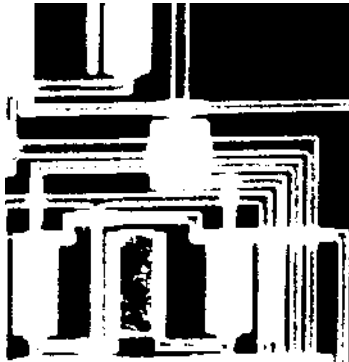
Connectivity can also be defined in a more general way for any dimension by using for conn a 3-by-3-by-...-by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood locations relative to the center element of conn. Note that conn must be symmetric about its center element.

**Class Support** BW1 must be logical or numeric, and it must be nonsparse. BW2 is of class logical.



**Examples**

```
BW1 = imread('circbw.tif');  
BW2 = bwperim(BW1,8);  
imshow(BW1)  
figure, imshow(BW2)
```

**See Also**

[bwarea](#), [bwboundaries](#), [bweuler](#), [bwtraceboundary](#), [conndef](#), [imfill](#)

# bwselect

---

**Purpose** Select objects in binary image

**Syntax**

```
BW2 = bwselect(BW,c,r,n)
BW2 = bwselect(BW,n)
[BW2,idx] = bwselect(...)

BW2 = bwselect(x,y,BW,xi,yi,n)
[x,y,BW2,idx,xi,yi] = bwselect(...)
```

**Description** `BW2 = bwselect(BW,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 pixels (i.e., pixels having a value of 1).

`BW2 = bwselect(BW,n)` displays the image `BW` on the screen and lets you select the  $(r,c)$  coordinates using the mouse. If you omit `BW`, `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,BW,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  $yi$ .

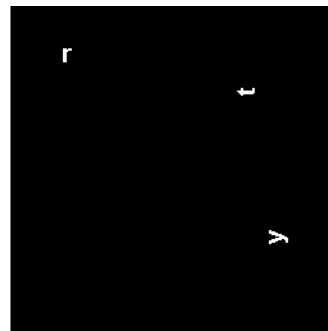
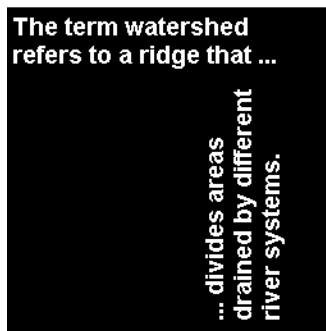
If `bwselect` is called with no output arguments, the resulting image is displayed in a new figure.

**Class Support**

The input image BW can be logical or numeric and must be 2-D and nonsparse. The output image BW2 is of class logical.

**Examples**

```
BW1 = imread('text.png');
c = [43 185 212];
r = [38 68 181];
BW2 = bwselect(BW1,c,r,4);
imshow(BW1), figure, imshow(BW2)
```



**See Also**

`bwlabel`, `imfill`, `impixel`, `roipoly`, `roifill`

# bwtraceboundary

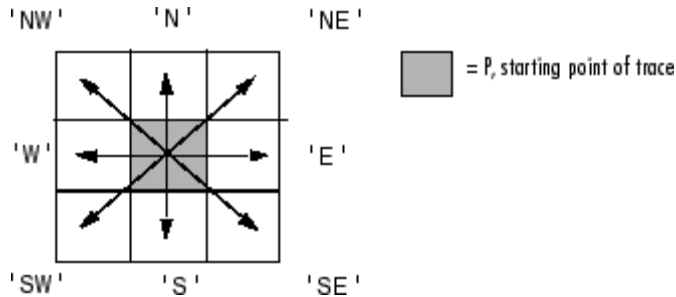
---

**Purpose** Trace object in binary image

**Syntax**  
B = bwtraceboundary(BW,P,fstep)  
B = bwtraceboundary(BW,P,fstep,conn)  
B = bwtraceboundary(...,N,dir)

**Description** B = bwtraceboundary(BW,P,fstep) traces the outline of an object in binary image bw. Nonzero pixels belong to an object and 0 pixels constitute the background. P is a two-element vector specifying the row and column coordinates of the point on the object boundary where you want the tracing to begin.

fstep is a string specifying the initial search direction for the next object pixel connected to P. You use strings such as 'N' for north, 'NE' for northeast, to specify the direction. The following figure illustrates all the possible values for fstep.



bwtraceboundary returns B, a Q-by-2 matrix, where Q is the number of boundary pixels for the region. B holds the row and column coordinates of the boundary pixels.

B = bwtraceboundary(bw,P,fstep,conn) specifies the connectivity to use when tracing the boundary. conn can have either of the following scalar values.

Value	Meaning
4	4-connected neighborhood  <b>Note</b> With this connectivity, <code>fstep</code> is limited to the following values: 'N', 'E', 'S', and 'W'.
8	8-connected neighborhood. This is the default.

`B = bwtraceboundary(...,N,dir)` specifies `n`, the maximum number of boundary pixels to extract, and `dir`, the direction in which to trace the boundary. When `N` is set to `Inf`, the default value, the algorithm identifies all the pixels on the boundary. `dir` can have either of the following values:

Value	Meaning
'clockwise'	Search in a clockwise direction. This is the default.
'counterclockwise'	Search in counterclockwise direction.

## Class Support

`BW` can be logical or numeric and it must be real, 2-D, and nonsparse. `B`, `P`, `conn`, and `N` are of class `double`. `dir` and `fstep` are strings.

## Examples

Read in and display a binary image. Starting from the top left, project a beam across the image searching for the first nonzero pixel. Use the location of that pixel as the starting point for the boundary tracing. Including the starting point, extract 50 pixels of the boundary and overlay them on the image. Mark the starting points with a green x. Mark beams that missed their targets with a red x.

```
BW = imread('blobs.png');
imshow(BW,[]);
s=size(BW);
for row = 2:55:s(1)
```

# bwtraceboundary

---

```
for col=1:s(2)
    if BW(row,col),
        break;
    end
end

contour = bwtraceboundary(BW, [row, col], 'W', 8, 50,...
    'counterclockwise');

if(~isempty(contour))
    hold on;
    plot(contour(:,2),contour(:,1),'g','LineWidth',2);
    hold on;
    plot(col, row,'gx','LineWidth',2);
else
    hold on; plot(col, row,'rx','LineWidth',2);
end
end
```

## See Also

bwboundaries, bwperim

**Purpose** Ultimate erosion

**Syntax**  
 BW2 = bwulterode(BW)  
 BW2 = bwulterode(BW,method,conn)

**Description** BW2 = bwulterode(BW) computes the ultimate erosion of the binary image BW. The ultimate erosion of BW consists of the regional maxima of the Euclidean distance transform of the complement of BW. The default connectivity for computing the regional maxima is 8 for two dimensions, 26 for three dimensions, and conndef(ndims(BW), 'maximal') for higher dimensions.

BW2 = bwulterode(BW,method,conn) specifies the distance transform method and the regional maxima connectivity. *method* can be one of the strings 'euclidean', 'cityblock', 'chessboard', and 'quasi-euclidean'.

conn can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can be defined in a more general way for any dimension by using for conn a 3-by-3-by... - by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood locations relative to the center element of conn. Note that conn must be symmetric about its center element.

# bwulterode

---

## **Class Support**

BW can be numeric or logical and it must be nonsparse. It can have any dimension. The return value BW2 is always a logical array.

## **Examples**

```
originalBW = imread('circles.png');  
imshow(originalBW)  
ultimateErosion = bwulterode(originalBW);  
figure, imshow(ultimateErosion)
```

## **See Also**

`bwdist`, `conndef`, `imregionalmax`



<b>Purpose</b>	Unpack binary image
<b>Syntax</b>	<code>BW = bwunpack(BWP,m)</code>
<b>Description</b>	<p><code>BW = bwunpack(BWP,m)</code> unpacks the packed binary image <code>BWP</code>. <code>BWP</code> is a <code>uint32</code> array. When it unpacks <code>BWP</code>, <code>bwunpack</code> maps the least significant bit of the first row of <code>BWP</code> to the first pixel in the first row of <code>BW</code>. The most significant bit of the first element of <code>BWP</code> maps to the first pixel in the 32nd row of <code>BW</code>, and so on. <code>BW</code> is <code>M</code>-by-<code>N</code>, where <code>N</code> is the number of columns of <code>BWP</code>. If <code>m</code> is omitted, its default value is <code>32*size(BWP,1)</code>.</p> <p>Binary image packing is used to accelerate some binary morphological operations, such as dilation and erosion. If the input to <code>imdilate</code> or <code>imerode</code> is a packed binary image, the functions use a specialized routine to perform the operation faster.</p> <p><code>bwpack</code> is used to create packed binary images.</p>
<b>Class Support</b>	<code>BWP</code> is of class <code>uint32</code> and must be real, 2-D, and nonsparse. The return value <code>BW</code> is of class <code>uint8</code> .
<b>Examples</b>	<p>Pack, dilate, and unpack a binary image.</p> <pre>bw = imread('text.png'); bwp = bwpack(bw); bwp_dilated = imdilate(bwp,ones(3,3),'ispacked'); bw_dilated = bwunpack(bwp_dilated, size(bw,1));</pre>
<b>See Also</b>	<code>bwpack</code> , <code>imdilate</code> , <code>imerode</code>

# checkerboard

---

**Purpose** Create checkerboard image

**Syntax**  
I = checkerboard  
I = checkerboard(n)  
I = checkerboard(n,p,q)

**Description** I = checkerboard creates an 8-by-8 square checkerboard image that has four identifiable corners. Each square has 10 pixels per side. The light squares on the left half of the checkerboard are white. The light squares on the right half of the checkerboard are gray.

I = checkerboard(n) creates a checkerboard image where each square has n pixels per side.

I = checkerboard(n,p,q) creates a rectangular checkerboard where p specifies the number of rows and q specifies the number of columns. If you omit q, it defaults to p and the checkerboard is square.

Each row and column is made up of tiles. Each tile contains four squares, n pixels per side, defined as

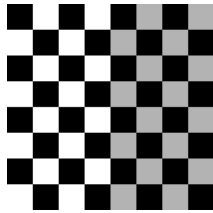
```
TILE = [DARK LIGHT; LIGHT DARK]
```



The light squares on the left half of the checkerboard are white. The light squares on the right half of the checkerboard are gray.

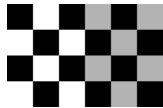
**Examples** Create a checkerboard where the side of every square is 20 pixels in length.

```
I = checkerboard(20);imshow(I)
```



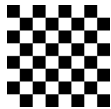
Create a rectangular checkerboard that is 2 tiles in height and 3 tiles wide.

```
J = checkerboard(10,2,3);  
figure, imshow(J)
```



Create a black and white checkerboard.

```
K = (checkerboard > 0.5);  
figure, imshow(K)
```



## See Also

[cp2tform](#), [imtransform](#), [maketform](#)

# cmpermute

---

**Purpose**

Rearrange colors in 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. The cmpermute function 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.

**Examples**

Order a colormap by luminance.

```
load trees
ntsc = rgb2ntsc(map);
[dum,index] = sort(ntsc(:,1));
[Y,newmap] = cmpermute(X,map,index);
figure, imshow(X,map)
figure, imshow(Y,newmap)
```

**See Also**

randperm, sort in the MATLAB Function Reference

**Purpose** Eliminate duplicate colors in colormap; convert grayscale or truecolor image to indexed 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. The cmunique function removes duplicate rows from the colormap and adjusts the indices in the image matrix accordingly.

[Y,newmap] = cmunique(RGB) converts the true-color image RGB to the indexed image Y and its associated colormap newmap. The return value newmap is the smallest possible colormap for the image, containing one entry for each unique color in RGB. (Note that newmap might 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 grayscale image I to an indexed image Y and its associated colormap newmap. The return value 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.

**Examples** Use the magic function to create a sample 4-by-4 image that uses every value in the range between 1 and 16.

```
X = magic(4)

X =

    16     2     3    13
```

# cmunique

---

```
5    11    10    8
9     7     6    12
4    14    15     1
```

Concatenate two 8-entry grayscale colormaps created using the `gray` function. The resultant colormap, `map`, has 16 entries. Entries 9 through 16 are duplicates of entries 1 through 8.

```
map = [gray(8); gray(8)]
size(map)
```

```
ans =
```

```
16     3
```

Use `cmunique` to eliminate duplicate entries in the colormap.

```
[Y, newmap] = cmunique(X, map);
size(newmap)
```

```
ans =
```

```
8     3
```

`cmunique` adjusts the values in the original image `X` to index the new colormap.

```
Y =
```

```
7     1     2     4
4     2     1     7
0     6     5     3
3     5     6     0
```

View both images to verify that their appearance is the same.

```
imshow(X, map, 'InitialMagnification', 'fit')
figure, imshow(Y, newmap, 'InitialMagnification', 'fit')
```

**See Also**      `gray2ind`, `rgb2ind`

# col2im

---

## Purpose

Rearrange matrix columns into blocks

## Syntax

```
A = col2im(B,[m n],[mm nn], 'distinct')  
A = col2im(B,[m n],[mm nn], 'sliding')
```

## Description

`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(:) A21(:) A12(:) 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`).

`col2im(B,[m n],[mm nn])` is the same as `col2im(B, [m n], [mm nn], 'sliding')`.

## Class Support

`B` can be logical or numeric. The return value `A` is of the same class as `B`.

## Examples

```
B = reshape(uint8(1:25),[5 5])'  
C = im2col(B,[1 5])  
A = col2im(C,[1 5],[5 5], 'distinct')
```

## See Also

`blkproc`, `colfilt`, `im2col`, `nlfilt`



**Purpose** Columnwise neighborhood operations

**Syntax**

```
B = colfilt(A,[m n],block_type,fun)
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 operations similar 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` must be a function handle. `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 that can have one of the values listed in this table.

Value	Description
'distinct'	Rearranges each <code>m</code> -by- <code>n</code> distinct block of <code>A</code> into a column in a temporary matrix, and then applies the function <code>fun</code> to this matrix. <code>fun</code> must return a matrix the same size as the temporary matrix. <code>colfilt</code> then rearranges the columns of the matrix returned by <code>fun</code> into <code>m</code> -by- <code>n</code> distinct blocks.
'sliding'	Rearranges each <code>m</code> -by- <code>n</code> sliding neighborhood of <code>A</code> into a column in a temporary matrix, and then applies the function <code>fun</code> to this matrix. <code>fun</code> must return a row vector containing a single value for each column in the temporary matrix. (Column compression functions such as <code>sum</code> return the appropriate type of output.) <code>colfilt</code> then rearranges the vector returned by <code>fun</code> into a matrix the same size as <code>A</code> .

`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 0's if the class of `A` is `uint8` or `uint16`, or 1's if the class of `A` is `double` or `single`.

---

**Note** To save memory, the `colfilt` function might divide `A` into subimages and process one subimage at a time. This implies that `fun` may be called multiple times, and that the first argument to `fun` may have a different number of columns each time.

---

## 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`.

## Examples

Set 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`, `function_handle`, `im2col`, `nlfilter`

**Purpose**      Display color bar

**Note**          colorbar is a MATLAB function.

# conndef

---

**Purpose** Create connectivity array

**Syntax** `conn = conndef(num_dims,type)`

**Description** `conn = conndef(num_dims,type)` returns the connectivity array defined by *type* for `num_dims` dimensions. *type* can have either of the values listed in this table.

Value	Description
'minimal'	Defines a neighborhood whose neighbors are touching the central element on an (N-1)-dimensional surface, for the N-dimensional case.
'maximal'	Defines a neighborhood including neighbors that touch the central element in any way; it is <code>ones(repmat(3,1,NUM_DIMS))</code> .

Several Image Processing Toolbox functions use `conndef` to create the default connectivity input argument.

## Examples

The minimal connectivity array for two dimensions includes the neighbors touching the central element along a line.

```
conn1 = conndef(2,'minimal')

conn1 =
     0     1     0
     1     1     1
     0     1     0
```

The minimal connectivity array for three dimensions includes all the neighbors touching the central element along a face.

```
conndef(3,'minimal')

ans(:,:,1) =
     0     0     0
```

```
      0     1     0
      0     0     0

ans(:,:,2) =
      0     1     0
      1     1     1
      0     1     0

ans(:,:,3) =
      0     0     0
      0     1     0
      0     0     0
```

The maximal connectivity array for two dimensions includes all the neighbors touching the central element in any way.

```
conn2 = conndef(2,'maximal')

conn2 =
      1     1     1
      1     1     1
      1     1     1
```

# conv2

---

**Purpose**            2-D convolution

**Note**                conv2 is a function in MATLAB.

**Purpose** 2-D 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 documentation

## convn

---

**Purpose** N-D convolution

**Note** convn is a MATLAB function.



**Purpose** 2-D correlation coefficient

**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 numeric or logical. The return value r is a scalar 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  $\bar{A} = \text{mean2}(A)$ , and  $\bar{B} = \text{mean2}(B)$ .

**See Also** `std2`  
`corrcoef` in the MATLAB Function Reference

## Purpose

Infer spatial transformation from control point pairs

## Syntax

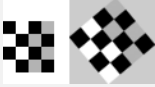



```
TFORM = cp2tform(input_points,base_points,transformtype)
TFORM = cp2tform(CPSTRUCT,transformtype)
TFORM = cp2tform(input_points,base_points,transformtype,parameter)
TFORM = cp2tform(CPSTRUCT,transformtype,parameter)
[TFORM,input_points,base_points] = cp2tform(CPSTRUCT,...)
[TFORM,input_points,base_points,input_points_bad,base_points_bad]
= cp2tform(...,'piecewise linear')
```



## Description

TFORM = cp2tform(input\_points,base\_points,transformtype) takes pairs of control points and uses them to infer a spatial transformation. input\_points is an m-by-2 double matrix containing the x- and y-coordinates of control points in the image you want to transform. base\_points is an m-by-2 double matrix containing the x- and y-coordinates of control points specified in the base image. The function returns a TFORM structure containing the spatial transformation.

TFORM = cp2tform(CPSTRUCT,transformtype) takes pairs of control points and uses them to infer a spatial transformation. CPSTRUCT is a structure that contains the control point matrices for the input and base images. You use the Control Point Selection Tool to create the CPSTRUCT.

transformtype specifies the type of spatial transformation to infer. This table lists all the transformation types supported by cp2tform in order of complexity. The 'lwm' and 'polynomial' transform types can each take an optional, additional parameter. See the syntax descriptions that follow for details.

Transformation Type	Description	Minimum Control Points	Example
'linear conformal'	Use this transformation when shapes in the input image are unchanged, but the image is distorted by some combination of translation, rotation, and scaling. Straight lines remain straight, and parallel lines are still parallel.	2 pairs	
'affine'	Use this transformation when shapes in the input image exhibit shearing. Straight lines remain straight, and parallel lines remain parallel, but rectangles become parallelograms.	3 pairs	
'projective'	Use this transformation when the scene appears tilted. Straight lines remain straight, but parallel lines converge toward vanishing points that might or might not fall within the image.	4 pairs	
'polynomial'	Use this transformation when objects in the image are curved. The higher the order of the polynomial, the better the fit, but the result can contain more curves than the base image.	6 pairs (order 2) 10 pairs (order 3) 15 pairs (order 4)	

Transformation Type	Description	Minimum Control Points	Example
'piecewise linear'	Use this transformation when parts of the image appear distorted differently.	4 pairs	
'lwm'	Use this transformation (local weighted mean), when the distortion varies locally and piecewise linear is not sufficient.	6 pairs (12 pairs recommended)	

---

**Note** When transformtype is 'linear conformal', 'affine', 'projective', or 'polynomial', and input\_points and base\_points (or CPSTRUCT) have the minimum number of control points needed for a particular transformation, cp2tform finds the coefficients exactly. If input\_points and base\_points include more than the minimum number of points, cp2tform uses a least squares solution. For more information, see mldivide.

---

TFORM =  
cp2tform(input\_points,base\_points,'polynomial',order) returns a TFORM structure specifying a 'polynomial' transformation, where order specifies the order of the polynomial to use. order can be the scalar value 2, 3, or 4. If you omit order, it defaults to 3.

TFORM = cp2tform(CPSTRUCT,'polynomial',order) same as the previous syntax except that the control points are specified in a CPSTRUCT.

TFORM = cp2tform(input\_points,base\_points,'lwm',N) returns a TFORM structure specifying a 'lwm' transformation, where N specifies the number of points used to infer each polynomial. The radius of influence extends out to the furthest control point used to infer that polynomial. The N closest points are used to infer a polynomial of order 2 for each

control point pair. If you omit  $N$ , it defaults to 12.  $N$  can be as small as 6, but making  $N$  small risks generating ill-conditioned polynomials.

`TFORM = cp2tform(CPSTRUCT, 'lwm', N)` same as the previous syntax except that the control points are specified in a `CPSTRUCT`.

`[TFORM, input_points, base_points] = cp2tform(CPSTRUCT, ...)` returns the control points that were actually used in the return values `input_points` and `base_points`. Unmatched and predicted points are not used. For more information, see `cpstruct2pairs`.

`[TFORM, input_points, base_points, input_points_bad, base_points_bad] = cp2tform(input_points, base_points, 'piecewise linear')` returns a `TFORM` structure specifying a 'piecewise linear' transformation. Returns the control points that were actually used in `input_points` and `base_points`, and returns the control points that were eliminated because they were middle vertices of degenerate fold-over triangles, in `input_points_bad` and `base_points_bad`.

`[TFORM, input_points, base_points, input_points_bad, base_points_bad] = cp2tform(CPSTRUCT, 'piecewise linear')` same as the previous syntax except that the control points are specified in a `CPSTRUCT`.

## Algorithms

`cp2tform` uses the following general procedure:

- 1 Use valid pairs of control points to infer a spatial transformation or an inverse mapping from output space  $(x,y)$  to input space  $(u,v)$  according to `transformtype`.
- 2 Return `TFORM` structure containing spatial transformation.

The procedure varies depending on the `transformtype`.

### Linear Conformal

Linear conformal transformations can include a rotation, a scaling, and a translation. Shapes and angles are preserved. Parallel lines remain parallel. Straight lines remain straight.

Let

```
sc = scale*cos(angle)
ss = scale*sin(angle)

[u v] = [x y 1] * [ sc -ss
                  ss  sc
                  tx  ty]
```

Solve for `sc`, `ss`, `tx`, `ty`.

```
t_lc = cp2tform(input_points,base_points,'linear conformal');
```

The coefficients of the inverse mapping are stored in `t_lc.tdata.Tinv`.

Since linear conformal transformations are a subset of affine transformations, `t_lc.forward_fcn` is `@affine_fwd` and `t_lc.inverse_fcn` is `@affine_inv`.

At least two control-point pairs are needed to solve for the four unknown coefficients.

## Affine

In an affine transformation, the `x` and `y` dimensions can be scaled or sheared independently and there can be a translation. Parallel lines remain parallel. Straight lines remain straight. Linear conformal transformations are a subset of affine transformations.

For an affine transformation,

```
[u v] = [x y 1] * Tinv
```

`Tinv` is a 3-by-2 matrix. Solve for the six elements of `Tinv`.

```
t_affine = cp2tform(input_points,base_points,'affine');
```

The coefficients of the inverse mapping are stored in `t_affine.tdata.Tinv`.

At least three control-point pairs are needed to solve for the six unknown coefficients.

**Projective**

In a projective transformation, quadrilaterals map to quadrilaterals. Straight lines remain straight. Affine transformations are a subset of projective transformations.

For a projective transformation

$$\begin{bmatrix} up & vp & wp \end{bmatrix} = \begin{bmatrix} x & y & w \end{bmatrix} * T_{inv}$$

where

$$\begin{aligned} u &= up/wp \\ v &= vp/wp \end{aligned}$$

$T_{inv}$  is a 3-by-3 matrix.

Assuming

$$\begin{aligned} T_{inv} &= \begin{bmatrix} A & D & G; \\ B & E & H; \\ C & F & I \end{bmatrix}; \\ u &= (Ax + By + C)/(Gx + Hy + I) \\ v &= (Dx + Ey + F)/(Gx + Hy + I) \end{aligned}$$

Solve for the nine elements of  $T_{inv}$ .

```
t_proj = cp2tform(input_points,base_points,'projective');
```

The coefficients of the inverse mapping are stored in `t_proj.tdata.Tinv`.

At least four control-point pairs are needed to solve for the nine unknown coefficients.

**Polynomial**

In a polynomial transformation, polynomial functions of  $x$  and  $y$  determine the mapping.

## Second-Order Polynomials

For a second-order polynomial transformation,

$$[u \ v] = [1 \ x \ y \ x*y \ x^2 \ y^2] * T_{inv}$$

Both  $u$  and  $v$  are second-order polynomials of  $x$  and  $y$ . Each second-order polynomial has six terms. To specify all coefficients,  $T_{inv}$  has size 6-by-2.

```
t_poly_ord2 = cp2tform(input_points, base_points, 'polynomial');
```

The coefficients of the inverse mapping are stored in `t_poly_ord2.tdata`.

At least six control-point pairs are needed to solve for the 12 unknown coefficients.

## Third-Order Polynomials

For a third-order polynomial transformation:

$$[u \ v] = [1 \ x \ y \ x*y \ x^2 \ y^2 \ y*x^2 \ x*y^2 \ x^3 \ y^3] * T_{inv}$$

Both  $u$  and  $v$  are third-order polynomials of  $x$  and  $y$ . Each third-order polynomial has ten terms. To specify all coefficients,  $T_{inv}$  has size 10-by-2.

```
t_poly_ord3 = cp2tform(input_points, base_points, 'polynomial',3);
```

The coefficients of the inverse mapping are stored in `t_poly_ord3.tdata`.

At least ten control-point pairs are needed to solve for the 20 unknown coefficients.



### Fourth-Order Polynomials

For a fourth-order polynomial transformation:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 1 & x & y & x*y & x^2 & y^2 & y*x^2 & x*y^2 & x^3 & y^3 & x^3*y & x^2*y^2 & x*y^3 & x^4 \\ y^4 \end{bmatrix} * T_{inv}$$

Both  $u$  and  $v$  are fourth-order polynomials of  $x$  and  $y$ . Each fourth-order polynomial has 15 terms. To specify all coefficients,  $T_{inv}$  has size 15-by-2.

```
t_poly_ord4 = cp2tform(input_points, base_points, 'polynomial',4);
```

The coefficients of the inverse mapping are stored in `t_poly_ord4.tdata`.

At least 15 control-point pairs are needed to solve for the 30 unknown coefficients.

### Piecewise Linear

In a piecewise linear transformation, linear (affine) transformations are applied separately to each triangular region of the image [1].

- 1 Find a Delaunay triangulation of the base control points.
- 2 Using the three vertices of each triangle, infer an affine mapping from base to input coordinates.

---

**Note** At least four control-point pairs are needed. Four pairs result in two triangles with distinct mappings.

---

### Local Weighted Mean

For each control point in `base_points`:

- 1 Find the  $N$  closest control points.
- 2 Use these  $N$  points and their corresponding points in `input_points` to infer a second-order polynomial.

- 3 Calculate the radius of influence of this polynomial as the distance from the center control point to the farthest point used to infer the polynomial (using `base_points`). [2]

---

**Note** At least six control-point pairs are needed to solve for the second-order polynomial. Ill-conditioned polynomials might result if too few pairs are used.

---

## Examples

```
I = checkerboard;
J = imrotate(I,30);
base_points = [11 11; 41 71];
input_points = [14 44; 70 81];
cpselect(J,I,input_points,base_points);

t = cp2tform(input_points,base_points,'linear conformal');

% Recover angle and scale by checking how a unit vector
% parallel to the x-axis is rotated and stretched.
u = [0 1];
v = [0 0];
[x, y] = tformfwd(t, u, v);
dx = x(2) - x(1);
dy = y(2) - y(1);
angle = (180/pi) * atan2(dy, dx)
scale = 1 / sqrt(dx^2 + dy^2)
```

## See Also

`cpcorr`, `cpselect`, `cpstruct2pairs`, `imtransform`

## References

[1] Goshtasby, Ardeshir, "Piecewise linear mapping functions for image registration," *Pattern Recognition*, Vol. 19, 1986, pp. 459-466.

[2] Goshtasby, Ardeshir, "Image registration by local approximation methods," *Image and Vision Computing*, Vol. 6, 1988, pp. 255-261.

---

<b>Purpose</b>	Tune control-point locations using cross correlation
<b>Syntax</b>	<code>input_points = cpcorr(input_points_in,base_points_in,input,base)</code>
<b>Description</b>	<p><code>input_points = cpcorr(input_points_in,base_points_in,input,base)</code> uses normalized cross-correlation to adjust each pair of control points specified in <code>input_points_in</code> and <code>base_points_in</code>.</p> <p><code>input_points_in</code> must be an M-by-2 double matrix containing the coordinates of control points in the input image. <code>base_points_in</code> is an M-by-2 double matrix containing the coordinates of control points in the base image.</p> <p><code>cpcorr</code> returns the adjusted control points in <code>input_points</code>, a double matrix the same size as <code>input_points_in</code>. If <code>cpcorr</code> cannot correlate a pair of control points, <code>input_points</code> contains the same coordinates as <code>input_points_in</code> for that pair.</p> <p><code>cpcorr</code> only moves the position of a control point by up to four pixels. Adjusted coordinates are accurate to one-tenth of a pixel. <code>cpcorr</code> is designed to get subpixel accuracy from the image content and coarse control-point selection.</p>

---

**Note** input and base images must have the same scale for `cpcorr` to be effective.

---

`cpcorr` cannot adjust a point if any of the following occur:

- Points are too near the edge of either image.
- Regions of images around points contain Inf or NaN.
- Region around a point in input image has zero standard deviation.
- Regions of images around points are poorly correlated.

## Class Support

The images `input` and `base` can be numeric and must contain finite values. The control-point pairs are of class `double`.

## Algorithm

`cpcorr` uses the following general procedure.

For each control-point pair,

- 1 Extract an 11-by-11 template around the input control point and a 21-by-21 region around the base control point.
- 2 Calculate the normalized cross-correlation of the template with the region.
- 3 Find the absolute peak of the cross-correlation matrix.
- 4 Use the position of the peak to adjust the coordinates of the input control point.

## Examples

Use `cpcorr` to fine-tune control points selected in an image. Note the difference in the values of the `input_points` matrix and the `input_points_adj` matrix.

```
input = imread('onion.png');
base = imread('peppers.png');
input_points = [127 93; 74 59];
base_points = [323 195; 269 161];
input_points_adj = cpcorr(input_points,base_points,...
                        input(:,:,1),base(:,:,1))

input_points_adj =

    127.0000    93.0000
    71.0000    59.6000
```

## See Also

`cp2tform`, `cpselect`, `imtransform`, `normxcorr2`

---

<b>Purpose</b>	Control Point Selection Tool
<b>Syntax</b>	<pre>cpselect(input,base) cpselect(input,base,CPSTRUCT_IN ) cpselect(input,base,xyinput_in,xybase_in) H = cpselect(input,base,...)</pre>
<b>Description</b>	<p><code>cpselect(input,base)</code> starts the Control Point Selection Tool, a graphical user interface that enables you to select control points in two related images. <code>input</code> is the image that needs to be warped to bring it into the coordinate system of the base image. <code>input</code> and <code>base</code> can be either variables that contain images or strings that identify files containing grayscale images. The Control Point Selection Tool returns the control points in a <code>CPSTRUCT</code> structure. (For more information, see “Using the Control Point Selection Tool” on page 7-13.)</p> <p><code>cpselect(input,base,CPSTRUCT_IN)</code> starts <code>cpselect</code> with an initial set of control points that are stored in <code>CPSTRUCT_IN</code>. This syntax allows you to restart <code>cpselect</code> with the state of control points previously saved in <code>CPSTRUCT_IN</code>.</p> <p><code>cpselect(input,base,xyinput_in,xybase_in)</code> starts <code>cpselect</code> with a set of initial pairs of control points. <code>xyinput_in</code> and <code>xybase_in</code> are <math>m</math>-by-2 matrices that store the input and base coordinates, respectively.</p> <p><code>H = cpselect(input,base,...)</code> returns a handle <code>H</code> to the tool. You can use the <code>close(H)</code> or <code>H.close</code> syntax to close the tool from the command line.</p>
<b>Class Support</b>	The input images can be of class <code>uint8</code> , <code>uint16</code> , <code>double</code> , or <code>logical</code> .
<b>Algorithm</b>	<p><code>cpselect</code> uses the following general procedure for control-point prediction.</p> <ol style="list-style-type: none"><li>1 Find all valid pairs of control points.</li></ol>

**2** Infer a spatial transformation between input and base control points using method that depends on the number of valid pairs, as follows:

2 pairs	Linear conformal
3 pairs	Affine
4 or more pairs	Projective

**3** Apply spatial transformation to the new point to generate the predicted point.

**4** Display predicted point.

## Notes

### Platform Support

cpselect requires Java and is not available on any platform that does not support Java.

### Memory Usage

You can increase the amount of memory available to cpselect by increasing the MATLAB Java Virtual Machine (JVM) memory allocation limit.

To increase this limit, create a file named `java.opts` and put it in your MATLAB startup directory. In this file, include the `-Xmx` option, specifying the amount of memory you want to give the JVM.

For example, to increase the JVM memory allocation limit to 128 MB, put the following text in the `java.opts` file:

```
-Xmx128m
```

---

**Note** To avoid virtual memory thrashing, never set the `-Xmx` option to more than 66% of the physical RAM available.

---

On UNIX systems, create the `java.opts` file in a directory where you intend to start MATLAB and move to that directory before starting MATLAB.

On Windows systems:

- 1 Create the `java.opts` file in a directory where you intend to start MATLAB.
- 2 Create a shortcut to MATLAB.
- 3 Right-click the shortcut and select **Properties**.

In the **Properties** dialog box, specify the name of the directory in which you created the `java.opts` file as the MATLAB startup directory.

## Examples

Start Control Point Selection tool with saved images.

```
aerial = imread('westconcordaerial.png');  
cpselect(aerial(:,:,1),'westconcordorthophoto.png')
```

Start Control Point Selection tool with images and points in the workspace.

```
I = checkerboard;  
J = imrotate(I,30);  
base_points = [11 11; 41 71];  
input_points = [14 44; 70 81];  
cpselect(J,I,input_points,base_points);
```

## See Also

`cpcorr`, `cp2tform`, `cpstruct2pairs`, `imtransform`

# cpstruct2pairs

---

**Purpose** Convert CPSTRUCT to valid pairs of control points

**Syntax** `[input_points, base_points] = cpstruct2pairs(CPSTRUCT)`

**Description** `[input_points, base_points] = cpstruct2pairs(CPSTRUCT)` takes a CPSTRUCT (produced by `cpselect`) and returns the arrays of coordinates of valid control point pairs in `input_points` and `base_points`. `cpstruct2pairs` eliminates unmatched points and predicted points.

**Examples** Start the Control Point Selection Tool, `cpselect`.

```
aerial = imread('westconcordaerial.png');  
cpselect(aerial(:,:,1), 'westconcordorthophoto.png')
```

Using `cpselect`, pick control points in the images. Select **Save to Workspace** from the **File** menu to save the points to the workspace. On the **Save** dialog box, check the **Structure with all points** check box and clear **Input points of valid pairs** and **Base points of valid pairs**. Click **OK**. Use `cpstruct2pairs` to extract the input and base points from the CPSTRUCT.

```
[input_points, base_points] = cpstruct2pairs(cpstruct);
```

**See Also** `cp2tform`, `cpselect`, `imtransform`



**Purpose** 2-D 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)` pads the matrix `A` with 0's to size `m`-by-`n` before transforming. If `m` or `n` is smaller than the corresponding dimension of `A`, `dct2` truncates `A`.

`B = dct2(A,[m n])` same as above.

**Class Support** `A` can be numeric or logical. 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

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

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

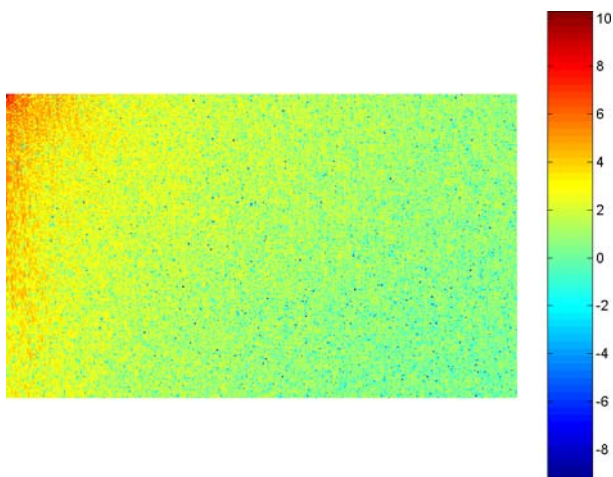
where  $M$  and  $N$  are the row and column size of `A`, respectively. If you apply the DCT to real data, the result is also real. The DCT tends to concentrate information, making it useful for image compression applications.

This transform can be inverted using `idct2`.

## Examples

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);  
imshow(I)  
figure, imshow(K,[0 255])
```

**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** 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 an integer 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.

**Examples**

```
A = im2double(imread('rice.png'));
D = dctmtx(size(A,1));
dct = D*A*D';
figure, imshow(dct)
```

**See Also** `dct2`

**Purpose**

Deblur image using blind deconvolution

**Syntax**

```
[J,PSF] = deconvblind(I,INITPSF)
[J,PSF] = deconvblind(I,INITPSF,NUMIT)
[J,PSF] = deconvblind(I,INITPSF,NUMIT,DAMPAR)
[J,PSF] = deconvblind(I,INITPSF,NUMIT,DAMPAR,WEIGHT)
[J,PSF] = deconvblind(I,INITPSF,NUMIT,DAMPAR,WEIGHT,READOUT)
[J,PSF] = deconvblind(...,FUN)
```

**Description**

[J,PSF] = deconvblind(I,INITPSF) deconvolves image I using the maximum likelihood algorithm, returning both the deblurred image J and a restored point-spread function PSF. The restored PSF is a positive array that is the same size as INITPSF, normalized so its sum adds up to 1. The PSF restoration is affected strongly by the size of the initial guess INITPSF and less by the values it contains. For this reason, specify an array of 1's as your INITPSF.

I can be a N-dimensional array.

To improve the restoration, deconvblind supports several optional parameters, described below. Use [] as a placeholder if you do not specify an intermediate parameter.

[J,PSF] = deconvblind(I,INITPSF,NUMIT) specifies the number of iterations (default is 10).

[J,PSF] = deconvblind(I,INITPSF,NUMIT,DAMPAR) specifies the threshold deviation of the resulting image from the input image I (in terms of the standard deviation of Poisson noise) below which damping occurs. The iterations are suppressed for the pixels that deviate within the DAMPAR value from their original value. This suppresses the noise generation in such pixels, preserving necessary image details elsewhere. The default value is 0 (no damping).

[J,PSF] = deconvblind(I,INITPSF,NUMIT,DAMPAR,WEIGHT) specifies which pixels in the input image I are considered in the restoration. By default, WEIGHT is a unit array, the same size as the input image. You can assign a value between 0.0 and 1.0 to elements in the WEIGHT array. The value of an element in the WEIGHT array determines how much the

# deconvblind

---

pixel at the corresponding position in the input image is considered. For example, to exclude a pixel from consideration, assign it a value of 0 in the WEIGHT array. You can adjust the weight value assigned to each pixel according to the amount of flat-field correction.

[J,PSF] = deconvblind(I,INITPSF,NUMIT,DAMPAR,WEIGHT,READOUT), where READOUT is an array (or a value) corresponding to the additive noise (e.g., background, foreground noise) and the variance of the read-out camera noise. READOUT has to be in the units of the image. The default value is 0.

[J,PSF] = deconvblind(...,FUN,P1,P2,...,PN), where FUN is a function describing additional constraints on the PSF. FUN must be a function handle.

FUN is called at the end of each iteration. FUN must accept the PSF as its first argument and can accept additional parameters P1, P2, ..., PN. The FUN function should return one argument, PSF, that is the same size as the original PSF and that satisfies the positivity and normalization constraints.

---

**Note** The output image J could exhibit ringing introduced by the discrete Fourier transform used in the algorithm. To reduce the ringing, use `I = edgetaper(I,PSF)` before calling `deconvblind`.

---

## Resuming Deconvolution

You can use `deconvblind` to perform a deconvolution that starts where a previous deconvolution stopped. To use this feature, pass the input image I and the initial guess at the PSF, INITPSF, as cell arrays: {I} and {INITPSF}. When you do, the `deconvblind` function returns the output image J and the restored point-spread function, PSF, as cell arrays, which can then be passed as the input arrays into the next `deconvblind` call. The output cell array J contains four elements:

J{1} contains I, the original image.

J{2} contains the result of the last iteration.

$J\{3\}$  contains the result of the next-to-last iteration.

$J\{4\}$  is an array generated by the iterative algorithm.

## Class Support

$I$  and  $INITPSF$  can be `uint8`, `uint16`, `int16`, `single`, or `double`. `DAMPAR` and `READOUT` must have the same class as the input image. Other inputs have to be `double`. The output image  $J$  (or the first array of the output cell) has the same class as the input image  $I$ . The output PSF is `double`.

## Examples

```
I = checkerboard(8);
PSF = fspecial('gaussian',7,10);
V = .0001;
BlurredNoisy = imnoise(imfilter(I,PSF),'gaussian',0,V);
WT = zeros(size(I));
WT(5:end-4,5:end-4) = 1;
INITPSF = ones(size(PSF));
[J P] = deconvblind(BlurredNoisy,INITPSF,20,10*sqrt(V),WT);
subplot(221);imshow(BlurredNoisy);
title('A = Blurred and Noisy');
subplot(222);imshow(PSF,[]);
title('True PSF');
subplot(223);imshow(J);
title('Deblurred Image');
subplot(224);imshow(P,[]);
title('Recovered PSF');
```

## See Also

`deconvlucy`, `deconvreg`, `deconvwnr`, `edgetaper`, `function_handle`, `imnoise`, `otf2psf`, `padarray`, `psf2otf`

# deconvlucy

---

**Purpose** Deblur image using Lucy-Richardson method.

**Syntax**

```
J = deconvlucy(I,PSF)
J = deconvlucy(I,PSF,NUMIT)
J = deconvlucy(I,PSF,NUMIT,DAMPAR)
J = deconvlucy(I,PSF,NUMIT,DAMPAR,WEIGHT)
J = deconvlucy(I,PSF,NUMIT,DAMPAR,WEIGHT,READOUT)
J = deconvlucy(I,PSF,NUMIT,DAMPAR,WEIGHT,READOUT,SUBSMPL)
```

**Description** J = deconvlucy(I,PSF) restores image I that was degraded by convolution with a point-spread function PSF and possibly by additive noise. The algorithm is based on maximizing the likelihood of the resulting image J's being an instance of the original image I under Poisson statistics.

I can be a N-dimensional array.

To improve the restoration, deconvlucy supports several optional parameters. Use [] as a placeholder if you do not specify an intermediate parameter.

J = deconvlucy(I,PSF,NUMIT) specifies the number of iterations the deconvlucy function performs. If this value is not specified, the default is 10.

J = deconvlucy(I,PSF,NUMIT,DAMPAR) specifies the threshold deviation of the resulting image from the image I (in terms of the standard deviation of Poisson noise) below which damping occurs. Iterations are suppressed for pixels that deviate beyond the DAMPAR value from their original value. This suppresses the noise generation in such pixels, preserving necessary image details elsewhere. The default value is 0 (no damping).

J = deconvlucy(I,PSF,NUMIT,DAMPAR,WEIGHT) specifies the weight to be assigned to each pixel to reflect its recording quality in the camera. A bad pixel is excluded from the solution by assigning it zero weight value. Instead of giving a weight of unity for good pixels, you can adjust their weight according to the amount of flat-field correction. The default is a unit array of the same size as input image I.



`J = deconvlucy(I,PSF,NUMIT,DAMPAR,WEIGHT,READOUT)` specifies a value corresponding to the additive noise (e.g., background, foreground noise) and the variance of the readout camera noise. `READOUT` has to be in the units of the image. The default value is 0.

`J = deconvlucy(I,PSF,NUMIT,DAMPAR,WEIGHT,READOUT,SUBSMPL)`, where `SUBSMPL` denotes subsampling and is used when the PSF is given on a grid that is `SUBSMPL` times finer than the image. The default value is 1.

---

**Note** The output image `J` could exhibit ringing introduced by the discrete Fourier transform used in the algorithm. To reduce the ringing, use `I = edgetaper(I,PSF)` before calling `deconvlucy`.

---

## Resuming Deconvolution

If `I` is a cell array, it can contain a single numerical array (the blurred image) or it can be the output from a previous run of `deconvlucy`.

When you pass a cell array to `deconvlucy` as input, it returns a 1-by-4 cell array `J`, where

`J{1}` contains `I`, the original image.

`J{2}` contains the result of the last iteration.

`J{3}` contains the result of the next-to-last iteration.

`J{4}` is an array generated by the iterative algorithm.

## Class Support

`I` and `PSF` can be `uint8`, `uint16`, `int16`, `double`, or `single`. `DAMPAR` and `READOUT` must have the same class as the input image. Other inputs have to be `double`. The output image `J` (or the first array of the output cell) has the same class as the input image `I`.

## Examples

```
I = checkerboard(8);
PSF = fspecial('gaussian',7,10);
V = .0001;
BlurredNoisy = imnoise(imfilter(I,PSF),'gaussian',0,V);
```

# deconvlucy

---

```
WT = zeros(size(I));
WT(5:end-4,5:end-4) = 1;
J1 = deconvlucy(BlurredNoisy,PSF);
J2 = deconvlucy(BlurredNoisy,PSF,20,sqrt(V));
J3 = deconvlucy(BlurredNoisy,PSF,20,sqrt(V),WT);

subplot(221);imshow(BlurredNoisy);
title('A = Blurred and Noisy');
subplot(222);imshow(J1);
title('deconvlucy(A,PSF)');
subplot(223);imshow(J2);
title('deconvlucy(A,PSF,NI,DP)');
subplot(224);imshow(J3);
title('deconvlucy(A,PSF,NI,DP,WT)');
```

## See Also

deconvblind, deconvreg, deconvwnr, otf2psf, padarray, psf2otf

**Purpose**

Deblur image using regularized filter

**Syntax**

```
J = deconvreg(I,PSF)
J = deconvreg(I,PSF,NOISEPOWER)
J = deconvreg(I,PSF,NOISEPOWER,LRANGE)
J = deconvreg(I,PSF,NOISEPOWER,LRANGE,REGOP)
[J, LAGRA] = deconvreg(I,PSF,...)
```

**Description**

`J = deconvreg(I,PSF)` deconvolves image `I` using the regularized filter algorithm, returning deblurred image `J`. The assumption is that the image `I` was created by convolving a true image with a point-spread function `PSF` and possibly by adding noise. The algorithm is a constrained optimum in the sense of least square error between the estimated and the true images under requirement of preserving image smoothness.

`I` can be a `N`-dimensional array.

`J = deconvreg(I,PSF,NOISEPOWER)` where `NOISEPOWER` is the additive noise power. The default value is 0.

`J = deconvreg(I,PSF,NOISEPOWER,LRANGE)` where `LRANGE` is a vector specifying range where the search for the optimal solution is performed. The algorithm finds an optimal Lagrange multiplier `LAGRA` within the `LRANGE` range. If `LRANGE` is a scalar, the algorithm assumes that `LAGRA` is given and equal to `LRANGE`; the `NP` value is then ignored. The default range is between `[1e-9 and 1e9]`.

`J = deconvreg(I,PSF,NOISEPOWER,LRANGE,REGOP)` where `REGOP` is the regularization operator to constrain the deconvolution. The default regularization operator is the Laplacian operator, to retain the image smoothness. The `REGOP` array dimensions must not exceed the image dimensions; any nonsingleton dimensions must correspond to the nonsingleton dimensions of `PSF`.

`[J, LAGRA] = deconvreg(I,PSF,...)` outputs the value of the Lagrange multiplier `LAGRA` in addition to the restored image `J`.

# deconvreg

---

---

**Note** The output image `J` could exhibit ringing introduced by the discrete Fourier transform used in the algorithm. To reduce the ringing, process the image with the `edgetaper` function prior to calling the `deconvreg` function. For example, `I = edgetaper(I,PSF)`.

---

## Class Support

`I` can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. Other inputs have to be of class `double`. `J` has the same class as `I`.

## Examples

```
I = checkerboard(8);
PSF = fspecial('gaussian',7,10);
V = .01;
BlurredNoisy = imnoise(imfilter(I,PSF),'gaussian',0,V);
NOISEPOWER = V*prod(size(I));
[J LAGRA] = deconvreg(BlurredNoisy,PSF,NOISEPOWER);

subplot(221); imshow(BlurredNoisy);
title('A = Blurred and Noisy');
subplot(222); imshow(J);
title('[J LAGRA] = deconvreg(A,PSF,NP)');
subplot(223); imshow(deconvreg(BlurredNoisy,PSF,[],LAGRA/10));
title('deconvreg(A,PSF,[],0.1*LAGRA)');
subplot(224); imshow(deconvreg(BlurredNoisy,PSF,[],LAGRA*10));
title('deconvreg(A,PSF,[],10*LAGRA)');
```

## See Also

`deconvblind`, `deconvlucy`, `deconvwnr`, `otf2psf`, `padarray`, `psf2otf`

**Purpose** Deblur image using Wiener filter

**Syntax**  
J = deconvwnr(I,PSF)  
J = deconvwnr(I,PSF,NSR)  
J = deconvwnr(I,PSF,NCORR,ICORR)

**Description** J = deconvwnr(I,PSF) restores image I that was degraded by convolution with a point-spread function PSF and possibly by additive noise. The algorithm is optimal in a sense of least mean square error between the estimated and the true image, and uses the correlation matrices of image and noise. In the absence of noise, the Wiener filter reduces to the ideal inverse filter.

I can be an N-dimensional array.

J = deconvwnr(I,PSF,NSR) where NSR is the noise-to-signal power ratio. NSR could be a scalar or an array of the same size as I. The default value is 0.

J = deconvwnr(I,PSF,NCORR,ICORR) where NCORR and ICORR are the autocorrelation functions of the noise and the original image, respectively. NCORR and ICORR can be of any size or dimension not exceeding the original image. An N-dimensional NCORR or ICORR array corresponds to the autocorrelation within each dimension. A vector NCORR or ICORR represents an autocorrelation function in the first dimension if PSF is a vector. If PSF is an array, the 1-D autocorrelation function is extrapolated by symmetry to all nonsingleton dimensions of PSF. A scalar NCORR or ICORR represents the power of the noise or the image.

---

**Note** The output image J could exhibit ringing introduced by the discrete Fourier transform used in the algorithm. To reduce the ringing, process the image with the edgetaper function prior to calling the deconvwnr function. For example, I = edgetaper(I,PSF)

---

## Class Support

I can be of class uint8, uint16, int16, single, or double. Other inputs have to be of class double. J has the same class as I.

## Examples

```
I = checkerboard(8);
noise = 0.1*randn(size(I));
PSF = fspecial('motion',21,11);
Blurred = imfilter(I,PSF,'circular');
BlurredNoisy = im2uint8(Blurred + noise);

% noise-to-power ratio
NSR = sum(noise(:).^2)/sum(I(:).^2);

% noise power
NP = abs(fftn(noise)).^2;
NPOW = sum(NP(:))/prod(size(noise));

% noise autocorrelation function, centered
NCORR = fftshift(real(ifftn(NP)));

% original image power
IP = abs(fftn(I)).^2;
IPOW = sum(IP(:))/prod(size(I));

% image autocorrelation function, centered
ICORR = fftshift(real(ifftn(IP)));
ICORR1 = ICORR(:,ceil(size(I,1)/2));

% noise to power ratio
NSR = NPOW/IPOW;

subplot(221);imshow(BlurredNoisy,[]);
title('A = Blurred and Noisy');
subplot(222);imshow(deconvwnr(BlurredNoisy,PSF,NSR),[]);
title('deconvwnr(A,PSF,NSR)');
subplot(223);imshow(deconvwnr(BlurredNoisy,PSF,NCORR,ICORR),[]);
title('deconvwnr(A,PSF,NCORR,ICORR)');
subplot(224);imshow(deconvwnr(BlurredNoisy,PSF,NPOW,ICORR1),[]);
```

```
title('deconvwnr(A,PSF,NPOW,ICORR_1_D)');
```

**See Also**

deconvblind, deconvlucy, deconvreg, otf2psf, padarray, psf2otf

# decorrstretch

---

**Purpose** Apply decorrelation stretch to multichannel image

**Syntax**  
S = decorrstretch(I)  
S = decorrstretch(I,TOL)

**Description** S = decorrstretch(I) applies a decorrelation stretch to a multichannel image I and returns the result in S. S has the same size and class as I. The mean and variance in each band are the same as in I.

S = decorrstretch(I,TOL) applies a contrast following the decorrelation stretch. The contrast stretch is controlled by TOL:

- TOL = [LOW\_FRACT HIGH\_FRACT] specifies the fraction of the image to saturate at low and high intensities.
- If TOL is a scalar, LOW\_FRACT = TOL, and HIGH\_FRACT = 1 - TOL, which saturates equal fractions at low and high intensities.

---

**Notes** The decorrelation stretch is normally applied to three band images (ordinary RGB images or RGB multispectral composite images), but decorrstretch works on an arbitrary number of bands.

The primary purpose of decorrelation stretch is visual enhancement. Small adjustments to TOL can strongly affect the visual appearance of the output.

---

**Class Support** The input image must be of class uint8, uint16, int16, single, or double.

**Examples**

```
[X, map] = imread('forest.tif');  
S = decorrstretch(ind2rgb(X, map), 'tol', 0.01);  
figure, imshow(X, map)  
figure, imshow(S)
```

**See Also** imadjust, stretchlim



**Purpose**

Anonymize DICOM file

**Syntax**

```
dicomanon(file_in,file_out)
dicomanon(...,'keep',FIELDS)
dicomanon(...,'update',ATTRS)
```

**Description**

`dicomanon(file_in,file_out)` removes confidential medical information from the DICOM file `file_in` and creates a new file `file_out` with the modified values. Image data and other attributes are unmodified.

`dicomanon(...,'keep',FIELDS)` modifies all of the confidential data except for those listed in `FIELDS`, which is a cell array of field names. This syntax is useful for keeping metadata that does not uniquely identify the patient but is useful for diagnostic purposes (e.g., `PatientAge`, `PatientSex`, etc.).

---

**Note** Keeping certain fields might compromise patient confidentiality.

---

`dicomanon(...,'update',ATTRS)` modifies the confidential data and updates particular confidential data. `ATTRS` is a structure whose fields are the names of the attributes to preserve. The structure values are the attribute values. Use this syntax to preserve the Study/Series/Image hierarchy or to replace a specific value with a more generic property (e.g., remove `PatientBirthDate` but keep a computed `PatientAge`).

For information about the fields that will be modified or removed, see DICOM Supplement 55 from <http://medical.nema.org/>.

**Examples**

Remove all confidential metadata from a file.

```
dicomanon('patient.dcm', 'anonymized.dcm')
```

# dicomanon

---

Create a training file.

```
dicomanon('tumor.dcm', 'tumor_anon.dcm', 'keep', ...
          {'PatientAge', 'PatientSex', 'StudyDescription'})
```

Anonymize a series of images, keeping the hierarchy.

```
values.StudyInstanceUID = dicomuid;
values.SeriesInstanceseriesUID = dicomuid;

d = dir('*.*.dcm');
for p = 1:numel(d)
    dicomanon(d(p).name, sprintf('anon%d.dcm', p), ...
              'update', values)
end
```

## See Also

[dicominfo](#), [dicomwrite](#)

**Purpose** Get or set active DICOM data dictionary

**Syntax**

```
dicomdict('set',dictionary)
dictionary = dicomdict('get')
dicomdict('factory')
```

**Description** `dicomdict('set',dictionary)` sets the Digital Imaging and Communications in Medicine (DICOM) data dictionary to the value stored in `dictionary`, a string containing the filename of the dictionary. DICOM-related functions use this dictionary by default, unless a different dictionary is provided at the command line.

`dictionary = dicomdict('get')` returns a string containing the filename of the stored DICOM data dictionary.

`dicomdict('factory')` resets the DICOM data dictionary to its default startup value.

---

**Note** The default data dictionary is a MAT-file, `dicom-dict.mat`. The toolbox also includes a text version of this default data dictionary, `dicom-dict.txt`. If you want to create your own DICOM data dictionary, open the `dicom-dict.txt` file in a text editor, modify it, and save it under another name.

---

**Examples**

```
dictionary = dicomdict('get')

dictionary =

dicom-dict.mat
```

**See Also** `dicominfo`, `dicomread`, `dicomwrite`

# dicominfo

---

**Purpose** Read metadata from DICOM message

**Syntax**  
`info = dicominfo(filename)`  
`info = dicominfo(filename, 'dictionary', D)`

**Description** `info = dicominfo(filename)` reads the metadata from the compliant Digital Imaging and Communications in Medicine (DICOM) file specified in the string `filename`.

`info = dicominfo(filename, 'dictionary', D)` uses the data dictionary file given in the string `D` to read the DICOM message. The file in `D` must be on the MATLAB search path. The default file is `dicom-dict.mat`.

## Examples

```
info = dicominfo('CT-MON02-16-ankle.dcm')
```

```
info =
```

```
          Filename: [1x62 char]
          FileModDate: '18-Dec-2000 11:06:43'
          FileSize: 525436
           Format: 'DICOM'
    FormatVersion: 3
           Width: 512
           Height: 512
          BitDepth: 16
        ColorType: 'grayscale'
    SelectedFrames: []
          FileStruct: [1x1 struct]
    StartOfPixelData: 1140
FileMetaInformationGroupLength: 192
  FileMetaInformationVersion: [2x1 uint8]
    MediaStorageSOPClassUID: '1.2.840.10008.5.1.4.1.1.7'
      .
      .
      .
```

**See Also**      `dicomdict`, `dicomread`, `dicomwrite`, `dicomuid`

# dicomlookup

---

**Purpose** Find attribute in DICOM data dictionary

**Syntax**  
`name = dicomlookup(group, element)`  
`[group,element] = dicomlookup(name)`

**Description**  
`name = dicomlookup(group, element)` looks into the current DICOM data dictionary for the attribute with the specified group and element tag and returns a string containing the name of the attribute. `group` and `element` can contain either a decimal value or hexadecimal string.  
`[group,element] = dicomlookup(name)` looks into the current DICOM data dictionary for the attribute specified byname and returns the group and element tags associated with the attribute. The values are returned as decimal values.

**Examples** Find the names of DICOM attributes using their tags.

```
name1 = dicomlookup('7FE0', '0010')
name2 = dicomlookup(40, 4)
```

Look up a DICOM attribute's tag (GROUP and ELEMENT) using its name.

```
[group, element] = dicomlookup('TransferSyntaxUID')
```

Examine the metadata of a DICOM file. This returns the same value even if the data dictionary changes.

```
metadata = dicominfo('CT-MONO2-16-ankle.dcm');
metadata.(dicomlookup('0028', '0004'))
```

**See Also** `dicomdict`, `dicominfo`

**Purpose**

Read DICOM image

**Syntax**

```
X = dicomread(filename)
X = dicomread(info)
[X,map] = dicomread(...)
[X,map,alpha] = dicomread(...)
[X,map,alpha,overlays] = dicomread(...)
[...] = dicomread(filename,'frames',v)
```

**Description**

`X = dicomread(filename)` reads the image data from the compliant Digital Imaging and Communications in Medicine (DICOM) file `filename`. For single-frame grayscale images, `X` is an M-by-N array. For single-frame true-color images, `X` is an M-by-N-by-3 array. Multiframe images are always 4-D arrays.

`X = dicomread(info)` reads the image data from the message referenced in the DICOM metadata structure `info`. The `info` structure is produced by the `dicominfo` function.

`[X,map] = dicomread(...)` returns the image `X` and the colormap `map`. If `X` is a grayscale or true-color image, `map` is empty.

`[X,map,alpha] = dicomread(...)` returns the image `X`, the colormap `map`, and an alpha channel matrix for `X`. The values of `alpha` are 0 if the pixel is opaque; otherwise they are row indices into `map`. The RGB value in `map` should be substituted for the value in `X` to use alpha. `alpha` has the same height and width as `X` and is 4-D for a multiframe image.

`[X,map,alpha,overlays] = dicomread(...)` returns the image `X`, the colormap `map`, an alpha channel matrix for `X`, and any overlays from the DICOM file. Each overlay is a 1-bit black and white image with the same height and width as `X`. If multiple overlays are present in the file, `overlays` is a 4-D multiframe image. If no overlays are in the file, `overlays` is empty.

`[...] = dicomread(filename,'frames',v)` reads only the frames in the vector `v` from the image. `v` must be an integer scalar, a vector of integers, or the string 'all'. The default value is 'all'.

# dicomread

---

## Class Support

X can be uint8, int8, uint16, or int16. map must be double. alpha has the same size and type as X. overlays is a logical array.

## Examples

Retrieve the data matrix X and colormap matrix map and create a montage.

```
[X, map] = dicomread('US-PAL-8-10x-echo.dcm');  
montage(X, map);
```

Call dicomread with the information retrieved from the DICOM file using dicominfo. Because a DICOM image is a 16-bit image, the example uses the imshow autoscaling syntax to display the image.

```
info = dicominfo('CT-MON02-16-ankle.dcm');  
Y = dicomread(info);  
figure, imshow(Y, 'DisplayRange',[]);
```

## See Also

dicomdict, dicominfo, dicomwrite



<b>Purpose</b>	Generate DICOM unique identifier
<b>Syntax</b>	UID = dicomuid
<b>Description</b>	<p>UID = dicomuid creates a string UID containing a new DICOM unique identifier.</p> <p>Multiple calls to dicomuid produce globally unique values. Two calls to dicomuid always return different values.</p>
<b>See Also</b>	dicominfo, dicomwrite

# dicomwrite

---

**Purpose** Write images as DICOM files

**Syntax**

```
dicomwrite(X, filename)
dicomwrite(X,map,filename)
dicomwrite(...,param1,value1,param2,value2,...)
dicomwrite(...,'ObjectType',IOD,...)
dicomwrite(...,'SOPClassUID',UID,...)
dicomwrite(...,meta_struct,...)
dicomwrite(...,info,...)
status = dicomwrite(...)
```

**Description** `dicomwrite(X, filename)` writes the binary, grayscale, or truecolor image `X` to the file `filename`, where `filename` is a string specifying the name of the Digital Imaging and Communications in Medicine (DICOM) file to create.

`dicomwrite(X,map,filename)` writes the indexed image `X` with colormap `map`.

`dicomwrite(...,param1,value1,param2,value2,...)` specifies optional metadata to write to the DICOM file or parameters that affect how the file is written. `param1` is a string containing the metadata attribute name or a `dicomwrite`-specific option. `value1` is the corresponding value for the attribute or option.

To find a list of the DICOM attributes that you can specify, see the data dictionary file, `dicom-dict.txt`, included with the Image Processing Toolbox. The following table lists the options that you can specify, in alphabetical order. Default values are enclosed in braces (`{}`).

Option Name	Description
'CompressionMode'	String specifying the type of compression to use when storing the image. Possible values:  { 'None' }  'JPEG lossless'  'JPEG lossy'  'RLE'
'CreateMode'	Specifies the method used for creating the data to put in the new file. Possible values:  { 'Create' } — Verify input values and generate missing data values.  'Copy' — Copy all values from the input and do not generate missing values.
'Dictionary'	String specifying the name of a DICOM data dictionary.
'Endian'	String specifying the byte ordering of the file.  'Big'  { 'Little' }  <hr/> <b>Note</b> If VR is set to 'Explicit', 'Endian' must be 'Big'. dicomwrite ignores this value if 'CompressionMode' or 'TransferSyntax' is set. <hr/>

Option Name	Description
'TransferSyntax'	<p>A DICOM UID specifying the 'Endian', 'VR', and 'CompressionMode' options.</p> <hr/> <p><b>Note</b> If specified, dicomwrite ignores any values specified for the 'Endian', 'VR', and 'CompressionMode' options. The TransferSyntax value encodes values for these options.</p> <hr/>
'VR'	<p>String specifying whether the two-letter value representation (VR) code should be written to the file.</p> <p>'explicit' — Write VR to file.</p> <p>{'implicit'} — Infer from data dictionary.</p> <hr/> <p><b>Note</b> If you specify the 'Endian' value 'Big', you must specify 'Explicit'.</p> <hr/>
'WritePrivate'	<p>Logical value indicating whether private data should be written to the file. Possible values: true — Write private data to file.</p> <p>{false} — Do not write private data.</p>

dicomwrite(..., 'ObjectType', IOD, ...) writes a file containing the necessary metadata for a particular type of DICOM Information Object (IOD). Supported IODs are

- 'Secondary Capture Image Storage' (default)
- 'CT Image Storage'
- 'MR Image Storage'

`dicomwrite(..., 'SOPClassUID', UID, ...)` provides an alternate method for specifying the IOD to create. UID is the DICOM unique identifier corresponding to one of the IODs listed above.

`dicomwrite(..., meta_struct, ...)` specifies optional metadata or file options in structure `meta_struct`. The names of fields in `meta_struct` must be the names of DICOM file attributes or options. The value of a field is the value you want to assign to the attribute or option.

`dicomwrite(..., info, ...)` specifies metadata in the metadata structure `info`, which is produced by the `dicominfo` function. For more information about this structure, see `dicominfo`.

`status = dicomwrite(...)` returns information about the metadata and the descriptions used to generate the DICOM file. This syntax can be useful when you specify an `info` structure that was created by `dicominfo` to the `dicomwrite` function. An `info` structure can contain many fields. If no metadata was specified, `dicomwrite` returns an empty matrix (`[]`).

The structure returned by `dicomwrite` contains these fields:

Field	Description
'BadAttribute'	The attribute's internal description is bad. It might be missing from the data dictionary or have incorrect data in its description.
'MissingCondition'	The attribute is conditional but no condition has been provided for when to use it.
'MissingData'	No data was provided for an attribute that must appear in the file.
'SuspectAttribute'	Data in the attribute does not match a list of enumerated values in the DICOM specification.

## Remarks

The DICOM format specification lists several Information Object Definitions (IODs) that can be created. These IODs correspond to

images and metadata produced by different real-world modalities (e.g., MR, X-ray, Ultrasound, etc.). For each type of IOD, the DICOM specification defines the set of metadata that must be present and possible values for other metadata.

dicomwrite fully implements a limited number of these IODs, listed above in the `ObjectType` syntax. For these IODs, dicomwrite verifies that all required metadata attributes are present, creates missing attributes if necessary, and specifies default values where possible. Using these supported IODs is the best way to ensure that the files you create conform to the DICOM specification. This is dicomwrite default behavior and corresponds to the `CreateMode` option value of 'Create'.

To write DICOM files for IODs that dicomwrite doesn't implement, use the 'Copy' value for the `CreateMode` option. In this mode, dicomwrite writes the image data to a file including the metadata that you specify as a parameter, shown above in the `info` syntax. The purpose of this option is to take metadata from an existing file of the same modality or IOD and use it to create a new DICOM file with different image pixel data.

---

**Note** Because dicomwrite copies metadata to the file without verification in 'copy' mode, it is possible to create a DICOM file that does not conform to the DICOM standard. For example, the file may be missing required metadata, contain superfluous metadata, or the metadata may no longer correspond to the modality settings used to generate the original image. When using 'Copy' mode, make sure that the metadata you use is from the same modality and IOD. If the copy you make is unrelated to the original image, use `dicomuid` to create new unique identifiers for series and study metadata. See the IOD descriptions in Part 3 of the DICOM specification for more information on appropriate IOD values.

---

## Examples

Read a CT image from the sample DICOM file included with the toolbox and then write the CT image to a file, creating a secondary capture image.

```
X = dicomread('CT-MONO2-16-ankle.dcm');  
dicomwrite(X, 'sc_file.dcm');
```

Write the CT image, *X*, to a DICOM file along with its metadata. Use the `dicominfo` function to retrieve metadata from a DICOM file.

```
metadata = dicominfo('CT-MONO2-16-ankle.dcm');  
dicomwrite(X, 'ct_file.dcm', metadata);
```

Copy all metadata from one file to another. In this mode, `dicomwrite` does not verify the metadata written to the file.

```
dicomwrite(X, 'ct_copy.dcm', metadata, 'CreateMode', 'copy');
```

## See Also

`dicomdict`, `dicominfo`, `dicomread`, `dicomuid`

# dither

---

<b>Purpose</b>	Convert image, increasing apparent color resolution by dithering
<b>Syntax</b>	<pre>X = dither(RGB,map) X = dither(RGB,map,Qm,Qe) BW = dither(I)</pre>
<b>Description</b>	<p><code>X = dither(RGB,map)</code> creates an indexed image approximation of the RGB image in the array <code>RGB</code> by dithering the colors in the colormap <code>map</code>. The colormap cannot have more than 65,536 colors.</p> <p><code>X = dither(RGB,map,Qm,Qe)</code> creates an indexed image from <code>RGB</code>, where <code>Qm</code> specifies the number of quantization bits to use along each color axis for the inverse color map, and <code>Qe</code> specifies the number of quantization bits to use for the color space error calculations. If <code>Qe &lt; Qm</code>, dithering cannot be performed, and an undithered indexed image is returned in <code>X</code>. If you omit these parameters, <code>dither</code> uses the default values <code>Qm = 5</code>, <code>Qe = 8</code>.</p> <p><code>BW = dither(I)</code> converts the grayscale image in the matrix <code>I</code> to the binary (black and white) image <code>BW</code> by dithering.</p>
<b>Class Support</b>	<p><code>RGB</code> can be <code>uint8</code>, <code>uint16</code>, <code>single</code>, or <code>double</code>. <code>I</code> can be <code>uint8</code>, <code>uint16</code>, <code>int16</code>, <code>single</code>, or <code>double</code>. All other input arguments must be <code>double</code>. <code>BW</code> is <code>logical</code>. <code>X</code> is <code>uint8</code>, if it is an indexed image with 256 or fewer colors; otherwise, it is <code>uint16</code>.</p>
<b>Algorithm</b>	<code>dither</code> increases the apparent color resolution of an image by applying Floyd-Steinberg's error diffusion dither algorithm.
<b>Examples</b>	<p>Convert intensity image to binary using dithering.</p> <pre>I = imread('cameraman.tif'); BW = dither(I); imshow(I), figure, imshow(BW)</pre>
<b>See Also</b>	<code>rgb2ind</code>



**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.

# double

---

**Purpose** Convert data to double precision

**Note** `double` is a MATLAB built-in function.

**Purpose**

Find edges in grayscale image

**Syntax**

```
BW = edge(I)

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**

`BW = edge(I)` takes a grayscale or a binary 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.

By default, `edge` uses the Sobel method to detect edges but the following provides a complete list of all the edge-finding methods supported by this function:

- 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 the 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 = edge(I, 'sobel', ..., options)` provides an optional string input. String 'nothinning' speeds up the operation of the algorithm by skipping the additional edge thinning stage. By default, or when 'thinning' string is specified, the algorithm applies edge thinning.

`[BW, thresh] = edge(I, 'sobel', ...)` returns the threshold value.

`[BW, thresh, gv, gh] = edge(I, 'sobel', ...)` returns vertical and horizontal edge responses to Sobel gradient operators. You can also use the following expressions to obtain gradient responses:

```
if ~(isa(I, 'double') || isa(I, 'single')); I = im2single(I); end
gh = imfilter(I, fspecial('sobel') /8, 'replicate');
gv = imfilter(I, fspecial('sobel') /8, 'replicate');
```

### **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 the 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, 'roberts')` specifies the Roberts method.

`BW = edge(I, 'roberts', 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 = edge(I, 'roberts', ..., options)` where options can be the text string 'thinning' or 'nothinning'. When you specify 'thinning', or don't specify a value, the algorithm applies edge thinning. Specifying the 'nothinning' option can speed up the operation of the algorithm by skipping the additional edge thinning stage.

`[BW, thresh] = edge(I, 'roberts', ...)` returns the threshold value.

`[BW, thresh, g45, g135] = edge(I, 'roberts', ...)` returns 45 degree and 135 degree edge responses to Roberts gradient operators. You can also use these expressions to obtain gradient responses:

```
if ~(isa(I, 'double') || isa(I, 'single'));
    I = im2single(I);
end
g45 = imfilter(I, [1 0; 0 -1]/2, 'replicate');
g135 = imfilter(I, [0 1; -1 0]/2, 'replicate');
```

## 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. If you specify a threshold of 0, the output image has closed contours, because it includes all the zero crossings in the input image.

`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. If you specify a threshold of 0, the output image has closed contours, because it includes all the zero crossings in the input image.

`[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` is a nonsparse numeric array. `BW` is of class `logical`.

## Remarks

For the gradient-magnitude methods (Sobel, Prewitt, Roberts), `thresh` is used to threshold the calculated gradient magnitude. For the zero-crossing methods, including Lap, `thresh` is used as a threshold for the zero-crossings; in other words, a large jump across zero is an edge, while a small jump isn't.

# edge

---

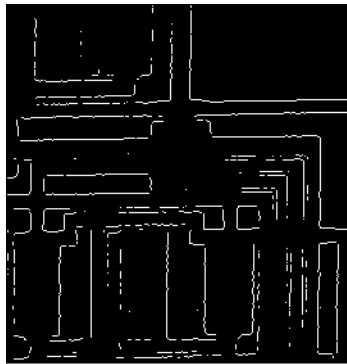
The Canny method applies two thresholds to the gradient: a high threshold for low edge sensitivity and a low threshold for high edge sensitivity. `edge` starts with the low sensitivity result and then grows it to include connected edge pixels from the high sensitivity result. This helps fill in gaps in the detected edges.

In all cases, the default threshold is chosen heuristically in a way that depends on the input data. The best way to vary the threshold is to run `edge` once, capturing the calculated threshold as the second output argument. Then, starting from the value calculated by `edge`, adjust the threshold higher (fewer edge pixels) or lower (more edge pixels).

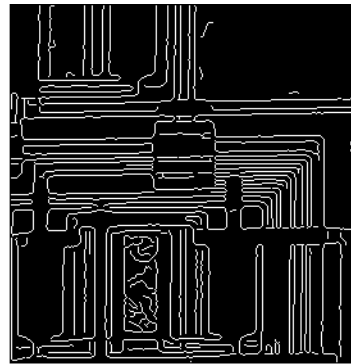
## Examples

Find the edges of an image using the Prewitt and Canny methods.

```
I = imread('circuit.tif');  
BW1 = edge(I, 'prewitt');  
BW2 = edge(I, 'canny');  
imshow(BW1);  
figure, imshow(BW2)
```



Prewitt Filter



Canny Filter

## See Also

`fspecial`



**References**

- [1] Canny, John, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-8, No. 6, 1986, pp. 679-698.
- [2] Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, pp. 478-488.
- [3] Parker, James R., *Algorithms for Image Processing and Computer Vision*, New York, John Wiley & Sons, Inc., 1997, pp. 23-29.

**Purpose** Taper discontinuities along image edges

**Syntax** `J = edgetaper(I,PSF)`

**Description** `J = edgetaper(I,PSF)` blurs the edges of the input image `I` using the point spread function `PSF`. The size of the `PSF` cannot exceed half of the image size in any dimension.

The output image `J` is the weighted sum of the original image `I` and its blurred version. The weighting array, determined by the autocorrelation function of `PSF`, makes `J` equal to `I` in its central region, and equal to the blurred version of `I` near the edges.

The `edgetaper` function reduces the ringing effect in image deblurring methods that use the discrete Fourier transform, such as `deconvwnr`, `deconvreg`, and `deconvlucy`.

**Class Support** `I` and `PSF` can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. `J` is of the same class as `I`.

**Examples**

```
original = imread('cameraman.tif');
PSF = fspecial('gaussian',60,10);
edgesTapered = edgetaper(original,PSF);
figure, imshow(original,[]);
figure, imshow(edgesTapered,[]);
```

**See Also** `deconvlucy`, `deconvreg`, `deconvwnr`, `otf2psf`, `padarray`, `psf2otf`

<b>Purpose</b>	Entropy of grayscale image
<b>Syntax</b>	<code>E = entropy(I)</code>
<b>Description</b>	<p><code>E = entropy(I)</code> returns <code>E</code>, a scalar value representing the entropy of grayscale image <code>I</code>. Entropy is a statistical measure of randomness that can be used to characterize the texture of the input image. Entropy is defined as</p> $-\text{sum}(p.*\log(p))$ <p>where <code>p</code> contains the histogram counts returned from <code>imhist</code>. By default, entropy uses two bins for logical arrays and 256 bins for <code>uint8</code>, <code>uint16</code>, or <code>double</code> arrays.</p> <p><code>I</code> can be a multidimensional image. If <code>I</code> has more than two dimensions, the entropy function treats it as a multidimensional grayscale image and not as an RGB image.</p>
<b>Class Support</b>	<code>I</code> can be <code>logical</code> , <code>uint8</code> , <code>uint16</code> , or <code>double</code> and must be real, nonempty, and nonsparse. <code>E</code> is <code>double</code> .
<b>Notes</b>	entropy converts any class other than <code>logical</code> to <code>uint8</code> for the histogram count calculation so that the pixel values are discrete and directly correspond to a bin value.
<b>Examples</b>	<pre>I = imread('circuit.tif'); J = entropy(I)</pre>
<b>See Also</b>	<code>imhist</code> , <code>entropyfilt</code> , <code>blkproc</code>
<b>References</b>	[1] Gonzalez, R.C., R.E. Woods, S.L. Eddins, <i>Digital Image Processing Using MATLAB</i> , New Jersey, Prentice Hall, 2003, Chapter 11.

# entropyfilt

---

**Purpose** Local entropy of grayscale image

**Syntax**  
`J = entropyfilt(I)`  
`J = entropyfilt(I,NHOOD)`

**Description** `J = entropyfilt(I)` returns the array `J`, where each output pixel contains the entropy value of the 9-by-9 neighborhood around the corresponding pixel in the input image `I`. `I` can have any dimension. If `I` has more than two dimensions, `entropyfilt` treats it as a multidimensional grayscale image and not as a truecolor (RGB) image. The output image `J` is the same size as the input image `I`.

For pixels on the borders of `I`, `entropyfilt` uses symmetric padding. In symmetric padding, the values of padding pixels are a mirror reflection of the border pixels in `I`.

`J = entropyfilt(I,NHOOD)` performs entropy filtering of the input image `I` where you specify the neighborhood in `NHOOD`. `NHOOD` is a multidimensional array of zeros and ones where the nonzero elements specify the neighbors. `NHOOD`'s size must be odd in each dimension.

By default, `entropyfilt` uses the neighborhood `true(9)`. `entropyfilt` determines the center element of the neighborhood by `floor((size(NHOOD) + 1)/2)`. To specify neighborhoods of various shapes, such as a disk, use the `strel` function to create a structuring element object and then use the `getnhood` function to extract the neighborhood from the structuring element object.

**Class Support** `I` can be logical, `uint8`, `uint16`, or `double`, and must be real and nonsparse. `NHOOD` can be logical or numeric and must contain zeros or ones. The output array `J` is of class `double`.

`entropyfilt` converts any class other than logical to `uint8` for the histogram count calculation so that the pixel values are discrete and directly correspond to a bin value.

**Examples**  
`I = imread('circuit.tif');`  
`J = entropyfilt(I);`

```
imshow(I), figure, imshow(J,[]);
```

**See Also**

entropy, imhist, rangefilt, stdfilt

**References**

[1] Gonzalez, R.C., R.E. Woods, S.L. Eddins, *Digital Image Processing Using MATLAB*, New Jersey, Prentice Hall, 2003, Chapter 11.

# fan2para

**Purpose** Convert fan-beam projections to parallel-beam

**Syntax**

```
P = fan2para(F,D)
P = fan2para(...,param1,val1,param2,val2,...)
[P,parallel_locations,parallel_rotation_angles] = fan2para(...)
```

**Description** P = fan2para(F,D) converts the fan-beam data F to the parallel-beam data P. D is the distance in pixels from the fan-beam vertex to the center of rotation that was used to obtain the projections.

P = fan2para(...,param1,val1,param2,val2,...) specifies parameters that control various aspects of the fan2para conversion, listed in the following table. Parameter names can be abbreviated, and case does not matter.

Parameter	Description
'FanCoverage'	String specifying the range through which the beams are rotated.  'cycle' — Rotate through the full range [0,360). This is the default.  'minimal' — Rotate the minimum range necessary to represent the object.
'FanRotationIncrement'	Positive real scalar specifying the increment of the rotation angle of the fan-beam projections, measured in degrees. Default value is 1.
'FanSensorGeometry'	String specifying how sensors are positioned.  'arc' — Sensors are spaced equally along a circular arc at distance D from the center of rotation. Default value is 'arc'  'line' — Sensors are spaced equally along a line, the closest point of which is distance D from the center of rotation.  See fanbeam for details.

Parameter	Description
'FanSensorSpacing'	<p>Positive real scalar specifying the spacing of the fan-beam sensors. Interpretation of the value depends on the setting of 'FanSensorGeometry'.</p> <p>If 'FanSensorGeometry' is set to 'arc' (the default), the value defines the angular spacing in degrees. Default value is 1.</p> <p>If 'FanSensorGeometry' is 'line', the value specifies the linear spacing. Default value is 1. See fanbeam for details.</p> <hr/> <p><b>Note</b> This linear spacing is measured on the <math>x'</math> axis. The <math>x'</math> axis for each column, col, of F is oriented at fan_rotation_angles(col) degrees counterclockwise from the x-axis. The origin of both axes is the center pixel of the image.</p>
'Interpolation'	<p>Text string specifying the type of interpolation used between the parallel-beam and fan-beam data.</p> <p>'nearest' — Nearest-neighbor</p> <p>{'linear'} — Linear</p> <p>'spline' — Piecewise cubic spline</p> <p>'pchip' — Piecewise cubic Hermite (PCHIP)</p> <p>'cubic' — Same as 'pchip'</p>
'ParallelCoverage'	<p>Text string specifying the range of rotation.</p> <p>'cycle' — Parallel data covers 360 degrees</p> <p>{'halfcycle'} — Parallel data covers 180 degrees</p>

# fan2para

Parameter	Description
'ParallelRotationIncrement'	<p>Positive real scalar specifying the parallel-beam rotation angle increment, measured in degrees. Parallel beam angles are calculated to cover <math>[0,180)</math> degrees with increment <code>PAR_ROT_INC</code>, where <code>PAR_ROT_INC</code> is the value of 'ParallelRotationIncrement'. <math>180/\text{PAR\_ROT\_INC}</math> must be an integer.</p> <p>If 'ParallelRotationIncrement' is not specified, the increment is assumed to be the same as the increment of the fan-beam rotation angles.</p>
'ParallelSensorSpacing'	<p>Positive real scalar specifying the spacing of the parallel-beam sensors in pixels. The range of sensor locations is implied by the range of fan angles and is given by</p> $[D \cdot \tan(\min(\text{FAN\_ANGLES})), \dots, D \cdot \tan(\max(\text{FAN\_ANGLES}))]$ <p>If 'ParallelSensorSpacing' is not specified, the spacing is assumed to be uniform and is set to the minimum spacing implied by the fan angles and sampled over the range implied by the fan angles.</p>

`[P,parallel_locations,parallel_rotation_angles] = fan2para(...)` returns the parallel-beam sensor locations in `parallel_locations` and rotation angles in `parallel_rotation_angles`.

## Class Support

The input arguments, `F` and `D`, can be double or single, and they must be nonsparse. All other numeric inputs are double. The output `P` is double.

## Examples

Create synthetic parallel-beam data, derive fan-beam data, and then use the fan-beam data to recover the parallel-beam data.



```

ph = phantom(128);
theta = 0:179;
[Psynthetic, xp] = radon(ph, theta);
imshow(Psynthetic, [], ...
        'XData', theta, 'YData', xp, 'InitialMagnification', 'fit')
axis normal
title('Synthetic Parallel-Beam Data')
xlabel('\theta (degrees)')
ylabel('x''')
colormap(hot), colorbar
Fsynthetic = para2fan(Psynthetic, 100, 'FanSensorSpacing', 1);

```

Recover original parallel-beam data.

```

[Precovered, Ploc, Pangles] = fan2para(Fsynthetic, 100, ...
                                       'FanSensorSpacing', 1, ...
                                       'ParallelSensorSpacing', 1);
figure
imshow(Precovered, [], 'XData', Pangles, ...
        'YData', Ploc, 'InitialMagnification', 'fit')
axis normal
title('Recovered Parallel-Beam Data')
xlabel('Rotation Angles (degrees)')
ylabel('Parallel Sensor Locations (pixels)')
colormap(hot), colorbar

```

## See Also

fanbeam, ifanbeam, iradon, para2fan, phantom, radon

# fanbeam

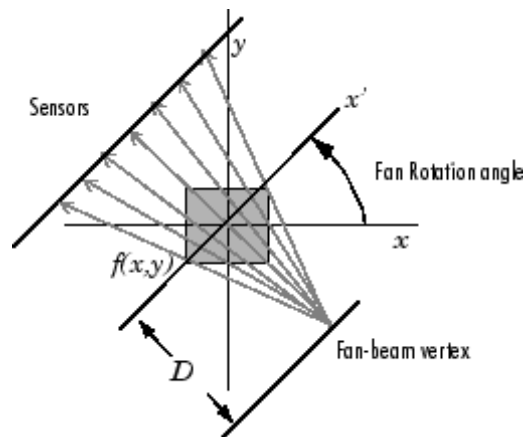
---

**Purpose** Fan-beam transform

**Syntax**  
`F = fanbeam(I,D)`  
`F = fanbeam(...,param1,val1,param1,val2,...)`  
`[F,sensor_positions,fan_rotation_angles] = fanbeam(...)`

**Description** `F = fanbeam(I,D)` computes the fan-beam data (sinogram) `F` from the image `I`. A sinogram is a special x-ray procedure that is done with contrast media (x-ray dye) to visualize any abnormal opening (sinus) in the body.

`D` is the distance in pixels from the fan-beam vertex to the center of rotation. The center of rotation is the center pixel of the image, defined as `floor(size(I)+1)/2`. `D` must be large enough to ensure that the fan-beam vertex is outside of the image at all rotation angles. See “Remarks” on page 17-149 for guidelines on specifying `D`. The following figure illustrates `D` in relation to the fan-beam vertex for one fan-beam geometry. See the `FanSensorGeometry` parameter for more information.



Each column of `F` contains the fan-beam sensor samples at one rotation angle. The number of columns in `F` is determined by the fan rotation increment. By default, the fan rotation increment is 1 degree so `F` has 360 columns.

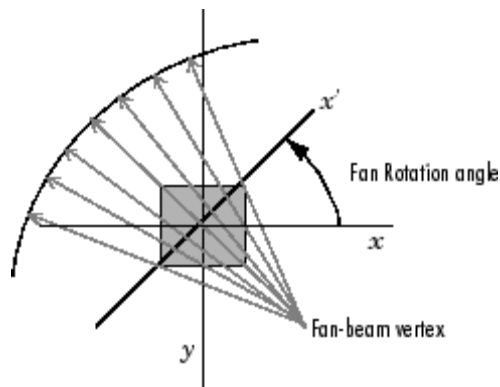
The number of rows in  $F$  is determined by the number of sensors. fanbeam determines the number of sensors by calculating how many beams are required to cover the entire image for any rotation angle.

For information about how to specify the rotation increment and sensor spacing, see the documentation for the FanRotationIncrement and FanSensorSpacing parameters, below.

$F = \text{fanbeam}(\dots, \text{param1}, \text{val1}, \text{param1}, \text{val2}, \dots)$  specifies parameters, listed below, that control various aspects of the fan-beam projections. Parameter names can be abbreviated, and case does not matter.

'FanRotationIncrement' -- Positive real scalar specifying the increment of the rotation angle of the fan-beam projections. Measured in degrees. Default value is 1.

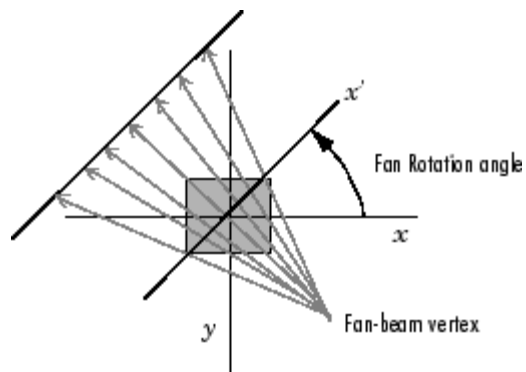
'FanSensorGeometry' -- Text string specifying how sensors are positioned. Valid values are 'arc' or 'line'. In the 'arc' geometry, sensors are spaced equally along a circular arc, as shown below. This is the default value.



In 'line' geometry, sensors are spaced equally along a line, as shown below.

# fanbeam

---



'FanSensorSpacing' -- Positive real scalar specifying the spacing of the fan-beam sensors. Interpretation of the value depends on the setting of 'FanSensorGeometry'. If 'FanSensorGeometry' is set to 'arc' (the default), the value defines the angular spacing in degrees. Default value is 1. If 'FanSensorGeometry' is 'line', the value specifies the linear spacing. Default value is 1.

---

**Note** This linear spacing is measured on the  $x'$  axis. The  $x'$  axis for each column, `col`, of `F` is oriented at `fan_rotation_angles(col)` degrees counterclockwise from the  $x$ -axis. The origin of both axes is the center pixel of the image.

---

`[F, fan_sensor_positions, fan_rotation_angles] = fanbeam(...)` returns the location of fan-beam sensors in `fan_sensor_positions` and the rotation angles where the fan-beam projections are calculated in `fan_rotation_angles`.

If 'FanSensorGeometry' is 'arc' (the default), `fan_sensor_positions` contains the fan-beam spread angles. If 'FanSensorGeometry' is 'line', `fan_sensor_positions` contains the fan-beam sensor positions along the  $x'$  axis. See 'FanSensorSpacing' for more information.

**Class Support** I can be logical or numeric. All other numeric inputs and outputs can be double. None of the inputs can be sparse.

**Remarks** As a guideline, try making D a few pixels larger than half the image diagonal dimension, calculated as follows

```
sqrt(size(I,1)^2 + size(I,2)^2)
```

The values returned in F are a numerical approximation of the fan-beam projections. The algorithm depends on the Radon transform, interpolated to the fan-beam geometry. The results vary depending on the parameters used. You can expect more accurate results when the image is larger, D is larger, and for points closer to the middle of the image, away from the edges.

**Examples** The following example computes the fan-beam projections for rotation angles that cover the entire image.

```
iptsetpref('ImshowAxesVisible','on')
ph = phantom(128);
imshow(ph)
[F,Fpos,Fangles] = fanbeam(ph,250);
figure
imshow(F,[],'XData',Fangles,'YData',Fpos,...
        'InitialMagnification','fit')
axis normal
xlabel('Rotation Angles (degrees)')
ylabel('Sensor Positions (degrees)')
colormap(hot), colorbar
```

The following example computes the Radon and fan-beam projections and compares the results at a particular rotation angle.

```
I = ones(100);
D = 200;
dtheta = 45;
```

```
% Compute fan-beam projections for 'arc' geometry
[Farc,FposArcDeg,Fangles] = fanbeam(I,D,...
    'FanSensorGeometry','arc',...
    'FanRotationIncrement',dtheta);
% Convert angular positions to linear distance
% along x-prime axis
FposArc = D*tan(FposArcDeg*pi/180);

% Compute fan-beam projections for 'line' geometry
[Fline,FposLine] = fanbeam(I,D,...
    'FanSensorGeometry','line',...
    'FanRotationIncrement',dtheta);

% Compute the corresponding Radon transform
[R,Rpos]=radon(I,Fangles);

% Display the three projections at one particular rotation
% angle. Note the three are very similar. Differences are
% due to the geometry of the sampling, and the numerical
% approximations used in the calculations.
figure
idx = find(Fangles==45);
plot(Rpos,R(:,idx),...
    FposArc,Farc(:,idx),...
    FposLine,Fline(:,idx))
legend('Radon','Arc','Line')
```

## See Also

fan2para, ifanbeam, iradon, para2fan, phantom, radon

## Reference

[1] Kak, A.C., & Slaney, M., *Principles of Computerized Tomographic Imaging*, IEEE Press, NY, 1988, pp. 92-93.

**Purpose**            2-D fast Fourier transform

**Note**              `fft2` is a function in MATLAB.

# fftn

---

**Purpose**            N-D fast Fourier transform

**Note**              `fftn` is a function in MATLAB.



**Purpose**            Shift zero-frequency component of fast Fourier transform to center of spectrum

**Note**              `fftshift` is a function in MATLAB.

# filter2

---

**Purpose**            2-D linear filtering

**Note**              `filter2` is a function in MATLAB.

<b>Purpose</b>	Find output bounds for spatial transformation
<b>Syntax</b>	<code>outbounds = findbounds(TFORM,inbounds)</code>
<b>Description</b>	<p><code>outbounds = findbounds(TFORM,inbounds)</code> estimates the output bounds corresponding to a given spatial transformation and a set of input bounds. TFORM is a spatial transformation structure as returned by <code>maketform</code>. <code>inbounds</code> is 2-by-<code>NUM_DIMS</code> matrix. The first row of <code>inbounds</code> specifies the lower bounds for each dimension, and the second row specifies the upper bounds. <code>NUM_DIMS</code> has to be consistent with the <code>ndims_in</code> field of TFORM.</p> <p><code>outbounds</code> has the same form as <code>inbounds</code>. It is an estimate of the smallest rectangular region completely containing the transformed rectangle represented by the input bounds. Since <code>outbounds</code> is only an estimate, it might not completely contain the transformed input rectangle.</p>
<b>Notes</b>	<p><code>imtransform</code> uses <code>findbounds</code> to compute the 'OutputBounds' parameter if the user does not provide it.</p> <p>If TFORM contains a forward transformation (a nonempty <code>forward_fcn</code> field), then <code>findbounds</code> works by transforming the vertices of the input bounds rectangle and then taking minimum and maximum values of the result.</p> <p>If TFORM does not contain a forward transformation, then <code>findbounds</code> estimates the output bounds using the Nelder-Mead optimization function <code>fminsearch</code>. If the optimization procedure fails, <code>findbounds</code> issues a warning and returns <code>outbounds = inbounds</code>.</p>
<b>Examples</b>	<pre>inbounds = [0 0; 1 1] tform = maketform('affine',[2 0 0; .5 3 0; 0 0 1]) outbounds = findbounds(tform, inbounds)</pre>
<b>See Also</b>	<code>cp2tform</code> , <code>imtransform</code> , <code>maketform</code> , <code>tformarray</code> , <code>tformfwd</code> , <code>tforminv</code>

# fliptform

---

**Purpose** Flip input and output roles of TFORM structure

**Syntax** TFLIP = fliptform(T)

**Description** TFLIP = fliptform(T) creates a new spatial transformation structure, a TFORM struct, by flipping the roles of the inputs and outputs in an existing TFORM struct.

**Examples**

```
T = maketform('affine', [.5 0 0; .5 2 0; 0 0 1]);  
T2 = fliptform(T)
```

The following are equivalent:

```
x = tformfwd([-3 7],T)  
x = tforminv([-3 7],T2)
```

**See Also** maketform, tformfwd, tforminv

**Purpose** Determine frequency spacing for 2-D frequency response

**Note** freqspace is a function in MATLAB.

# freqz2

---

**Purpose** 2-D frequency response

**Syntax**

```
[H,f1,f2] = freqz2(h,n1,n2)
[H,f1,f2] = freqz2(h,[n2 n1])
[H,f1,f2] = freqz2(h)
[H,f1,f2] = freqz2(h,f1,f2)
[...] = 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  $\pi$  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  $\pi$  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.

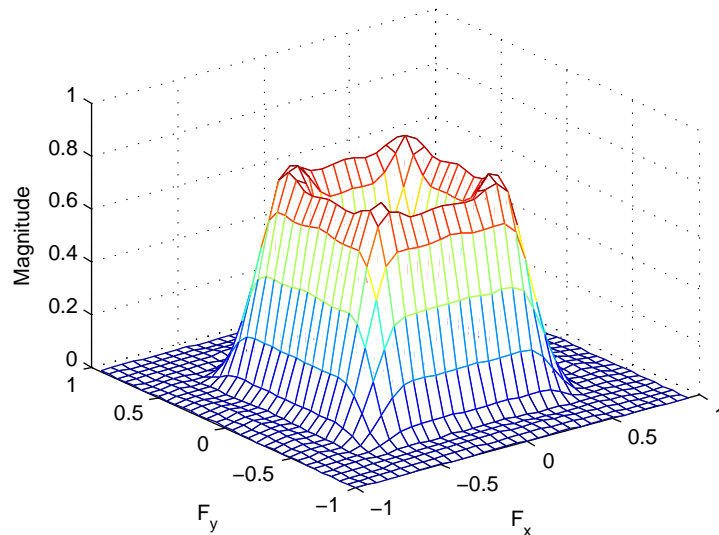
## Class Support

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`.

## Examples

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 documentation

# fsamp2

---

**Purpose** 2-D 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  $\pi$  radians.

$$\mathcal{I}_d(f_1, f_2) = H_d(\omega_1, \omega_2) \Big|_{\omega_1 = \pi f_1, \omega_2 = \pi f_2}$$

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  $\pi$  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`.

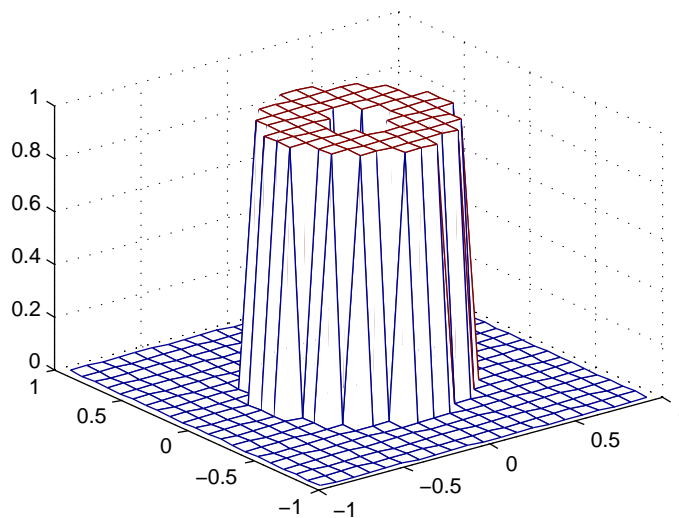
**Examples** Use `fsamp2` to design an approximately symmetric two-dimensional bandpass filter with passband between 0.1 and 0.5 (normalized



frequency, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians):

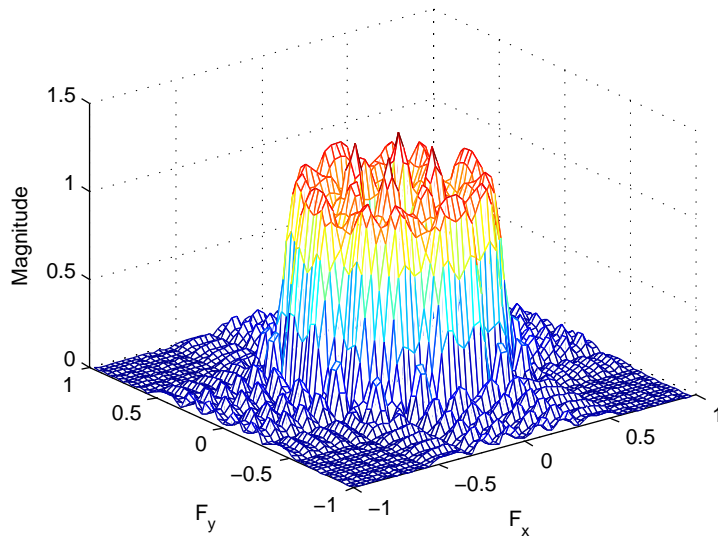
- 1 Create a matrix  $H_d$  that contains the desired bandpass response. Use `freqspace` to create the frequency range vectors  $f_1$  and  $f_2$ .

```
[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 2-D filter

**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 correlation kernel, which is the appropriate form to use with `imfilter`. *type* is a string having one of these values.

Value	Description
'average'	Averaging filter
'disk'	Circular averaging filter (pillbox)
'gaussian'	Gaussian lowpass filter
'laplacian'	Approximates the two-dimensional Laplacian operator
'log'	Laplacian of Gaussian filter
'motion'	Approximates the linear motion of a camera
'prewitt'	Prewitt horizontal edge-emphasizing filter
'sobel'	Sobel horizontal edge-emphasizing filter
'unsharp'	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('average',hsize)` returns an averaging filter `h` of size `hsize`. The argument `hsize` can be a vector specifying the number of rows and columns in `h`, or it can be a scalar, in which case `h` is a square matrix. The default value for `hsize` is `[3 3]`.

- `h = fspecial('disk',radius)` returns a circular averaging filter (pillbox) within the square matrix of side  $2*\text{radius}+1$ . The default radius is 5.
- `h = fspecial('gaussian',hsize,sigma)` returns a rotationally symmetric Gaussian lowpass filter of size `hsize` with standard deviation `sigma` (positive). `hsize` can be a vector specifying the number of rows and columns in `h`, or it can be a scalar, in which case `h` is a square matrix. The default value for `hsize` is `[3 3]`; the default value for `sigma` is 0.5.
- `h = fspecial('laplacian',alpha)` returns a 3-by-3 filter approximating the shape of the two-dimensional Laplacian operator. The parameter `alpha` controls the shape of the Laplacian and must be in the range 0.0 to 1.0. The default value for `alpha` is 0.2.
- `h = fspecial('log',hsize,sigma)` returns a rotationally symmetric Laplacian of Gaussian filter of size `hsize` with standard deviation `sigma` (positive). `hsize` can be a vector specifying the number of rows and columns in `h`, or it can be a scalar, in which case `h` is a square matrix. The default value for `hsize` is `[5 5]` and 0.5 for `sigma`.
- `h = fspecial('motion',len,theta)` returns a filter to approximate, once convolved with an image, the linear motion of a camera by `len` pixels, with an angle of `theta` degrees in a counterclockwise direction. The filter becomes a vector for horizontal and vertical motions. The default `len` is 9 and the default `theta` is 0, which corresponds to a horizontal motion of nine pixels.
- `h = fspecial('prewitt')` returns the 3-by-3 filter `h` (shown below) that emphasizes horizontal edges by approximating a vertical gradient. If you need to emphasize vertical edges, transpose the filter `h'`.

```
[ 1 1 1
 0 0 0
-1 -1 -1]
```

To find vertical edges, or for  $x$ -derivatives, use `h'`.

- `h = fspecial('sobel')` returns a 3-by-3 filter `h` (shown below) that emphasizes horizontal edges using the smoothing effect by approximating a vertical gradient. If you need to emphasize vertical edges, transpose the filter `h'`.

```
[ 1 2 1
 0 0 0
-1 -2 -1]
```

- `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 0.0 to 1.0. The default value for `alpha` is 0.2.

---

**Note** Do not be confused by the name of this filter: an unsharp filter is an image sharpening operator. The name comes from a publishing industry process in which an image is sharpened by subtracting a blurred (unsharp) version of the image from itself.

---

## Class Support

`h` is of class `double`.

## Examples

```
I = imread('cameraman.tif');
subplot(2,2,1);
imshow(I); title('Original Image');

H = fspecial('motion',20,45);
MotionBlur = imfilter(I,H,'replicate');
subplot(2,2,2);
imshow(MotionBlur);title('Motion Blurred Image');

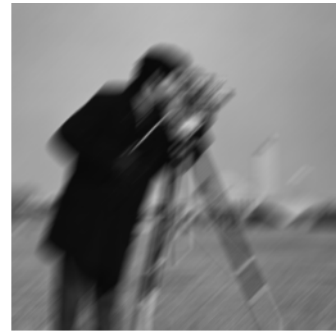
H = fspecial('disk',10);
blurred = imfilter(I,H,'replicate');
subplot(2,2,3);
```

```
imshow(blurred); title('Blurred Image');  
  
H = fspecial('unsharp');  
sharpened = imfilter(I,H,'replicate');  
subplot(2,2,4);  
imshow(sharpened); title('Sharpened Image');
```



Image Courtesy of MIT

**Original Image**



**Motion Blurred Image**



**Blurred Image**



**Sharpened Image**

**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

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

$$\nabla^2 = \frac{4}{(\alpha + 1)} \begin{bmatrix} \frac{\alpha}{4} & \frac{1-\alpha}{4} & \frac{\alpha}{4} \\ \frac{1-\alpha}{4} & -1 & \frac{1-\alpha}{4} \\ \frac{\alpha}{4} & \frac{1-\alpha}{4} & \frac{\alpha}{4} \end{bmatrix}$$

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

$$\text{ones}(n(1), n(2)) / (n(1) * n(2))$$

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, imfilter  
de12 in the MATLAB Function Reference



**Purpose** 2-D 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  $H(\omega_1, \omega_2)$ .

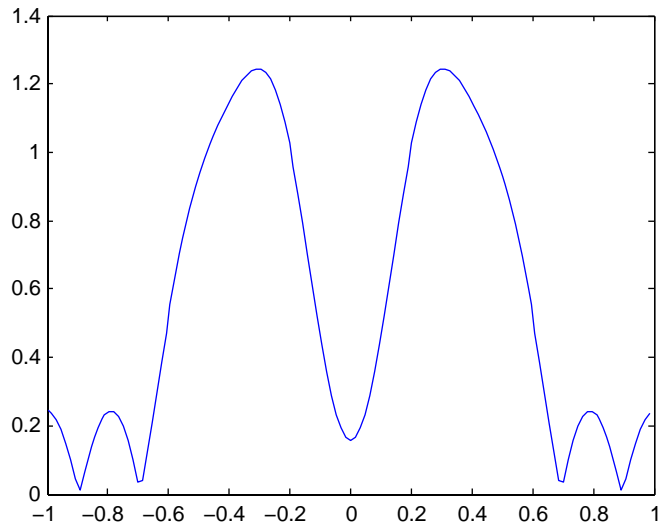
$$h(n_1, n_2) = \frac{1}{(2\pi)^2} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} H(\omega_1, \omega_2) e^{j\omega_1 n_1} e^{j\omega_2 n_2} d\omega_1 d\omega_2$$

## Examples

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  $\pi$  radians):

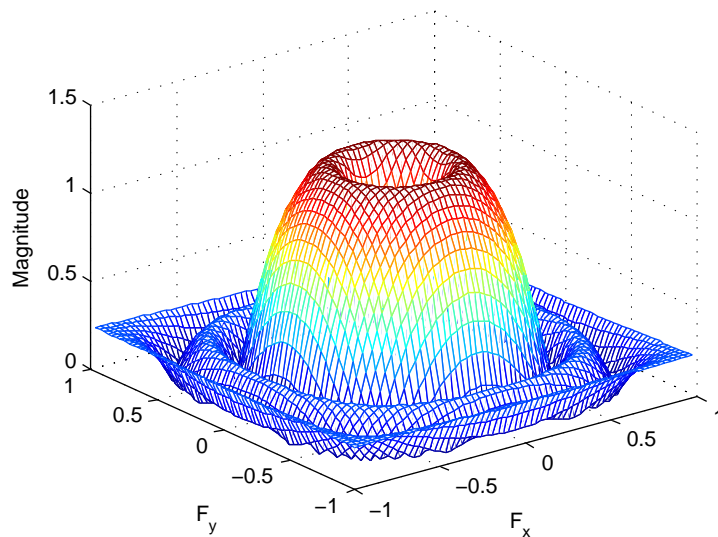
- 1 Since `ftrans2` transforms a one-dimensional FIR filter to create a two-dimensional filter, first design a one-dimensional FIR bandpass filter using the Signal Processing Toolbox function `remez`.

```
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.

# fwind1

---

**Purpose** 2-D FIR filter using 1-D 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  $\pi$  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  $\pi$  radians. The length of the windows controls the size of the resulting filter, as above.

## Class Support

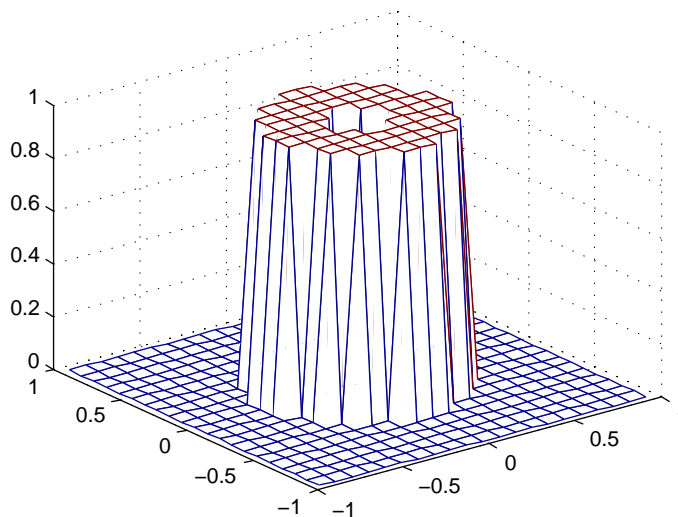
The input matrix `Hd` can be of class `double` or of any integer class. All other inputs to `fwind1` must be of class `double`. All outputs are of class `double`.

## Examples

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  $\pi$  radians):

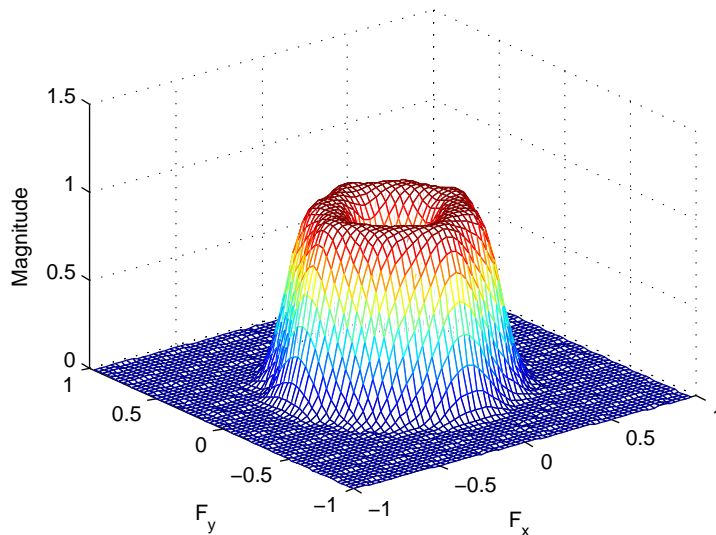
- 1 Create a matrix `Hd` that contains the desired bandpass response. Use `freqspace` to create the frequency range vectors `f1` and `f2`.

```
[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.

# fwind2

---

**Purpose** 2-D FIR filter using 2-D 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  $\pi$  radians. `h` is the same size as `win`.

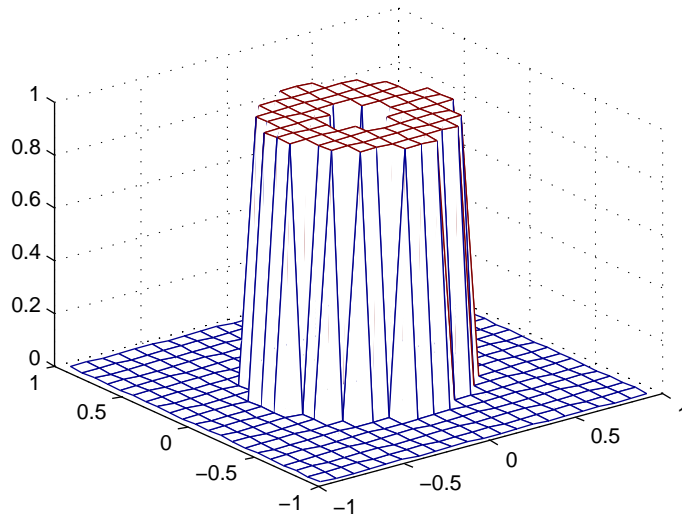
**Class Support** The input matrix `Hd` can be of class `double` or of any integer class. All other inputs to `fwind2` must be of class `double`. All outputs are of class `double`.

**Examples** 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  $\pi$  radians):

- 1 Create a matrix `Hd` that contains the desired bandpass response. Use `freqspace` to create the frequency range vectors `f1` and `f2`.

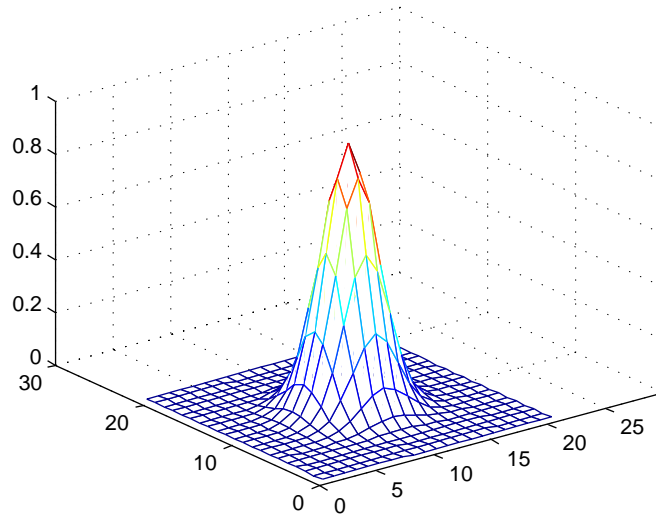


```
[f1,f2] = freqspace(21,'meshgrid');  
Hd = ones(21);  
r = sqrt(f1.^2 + f2.^2);  
Hd((r<0.1)|(r>0.5)) = 0;  
colormap(jet(64))  
mesh(f1,f2,Hd)
```



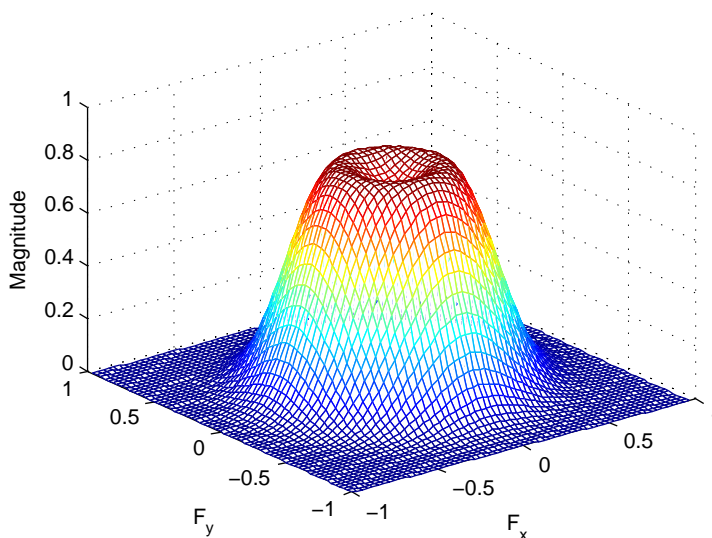
## 2 Create a two-dimensional Gaussian window using fspecial.

```
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  $w$ .

$$h_d(n_1, n_2) = \frac{1}{(2\pi)^2} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} H_d(\omega_1, \omega_2) e^{j\omega_1 n_1} e^{j\omega_2 n_2} d\omega_1 d\omega_2$$

$$h(n_1, n_2) = h_d(n_1, n_2)w(n_1, n_2)$$

## 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.

# getheight

---

<b>Purpose</b>	Height of structuring element
<b>Syntax</b>	<code>H = getheight(SE)</code>
<b>Description</b>	<code>H = getheight(SE)</code> returns an array the same size as <code>getnhood(SE)</code> containing the height associated with each of the structuring element neighbors. <code>H</code> is all zeros for a flat structuring element.
<b>Class Support</b>	<code>SE</code> is a STREL object. <code>H</code> is of class <code>double</code> .
<b>Examples</b>	<pre>se = strel(ones(3,3),magic(3)); getheight(se)</pre>
<b>See Also</b>	<code>strel</code> , <code>getnhood</code>

**Purpose** 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, or image. `A` is identical to the image CData; it contains the same values and is of the same class (uint8, uint16, double, or logical) as the image CData. If `h` is not an image or does not contain an image, `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	Indexed image
2	Intensity image with values in standard range ([0,1] for single and double arrays, [0,255] for uint8 arrays, [0,65535] for uint16 arrays)
3	Intensity data, but not in standard range
4	RGB image
5	Binary image

`[...] = getimage` returns information for the current axes object. It is equivalent to `[...] = getimage(gca)`.

# getimage

---

## Class Support

The output array A is of the same class as the image CData. All other inputs and outputs are of class double.

## Note

For `int16` and `single` images, the image data returned by `getimage` is of class `double`, not `int16` or `single`. This is because the `getimage` function gets the data from the image object's `CData` property and image objects store `int16` and `single` image data as class `double`.

For example, create an image object of class `int16`. If you retrieve the `CData` from the object and check its class, it returns `double`.

```
h = imshow(ones(10,'int16'));
class(get(h,'CData'))
```

Therefore, if you get the image data using the `getimage` function, the data it returns is also of class `double`. The `flag` return value is set to 3.

```
[img,flag] = getimage(h);
class(img)
```

The same is true for an image of class `single`. Getting the `CData` directly from the image object or by using `getimage`, the class of the returned data is `double`.

```
h = imshow(ones(10,'single'));
class(get(h,'CData'))
[img,flag] = getimage(h);
class(img)
```

For images of class `single`, the `flag` return value is set to 2 because `single` and `double` share the same dynamic range.

## Examples

After using `imshow` or `imtool` to display an image directly from a file, use `getimage` to get the image data into the workspace.

```
imshow rice.png
I = getimage;
```

```
imtool cameraman.tif  
I = getimage(imgca);
```

## See Also

imshow, imtool

# getimagemodel

---

**Purpose** Image model object from image object

**Syntax** `imgmodel = getimagemodel(himage)`

**Description** `imgmodel = getimagemodel(himage)` returns the image model object associated with `himage`. `himage` must be a handle to an image object or an array of handles to image objects.

The return value `imgmodel` is an image model object. If `himage` is an array of handles to image objects, `imgmodel` is an array of image models.

If `himage` does not have an associated image model object, `getimagemodel` creates one.

**Examples**

```
h = imshow('bag.png');  
imgmodel = getimagemodel(h);
```

**See Also** `imagemodel`



---

<b>Purpose</b>	Select polyline with mouse
<b>Syntax</b>	<pre>[x,y] = getline(fig) [x,y] = getline(ax) [x,y] = getline [x,y] = getline(...,'closed')</pre>
<b>Description</b>	<p>[x,y] = getline(fig) lets you select a polyline in the current axes of figure fig using the mouse. Coordinates of the polyline are returned in X and Y. Use normal button clicks to add points to the polyline. A shift-, right-, or double-click adds a final point and ends the polyline selection. Pressing <b>Return</b> or <b>Enter</b> ends the polyline selection without adding a final point. Pressing <b>Backspace</b> or <b>Delete</b> removes the previously selected point from the polyline.</p> <p>[x,y] = getline(ax) lets you select a polyline in the axes specified by the handle ax.</p> <p>[x,y] = getline is the same as [x,y] = getline(gcf).</p> <p>[x,y] = getline(...,'closed') animates and returns a closed polygon.</p>
<b>See Also</b>	getpts, getrect

# getneighbors

---

**Purpose** Structuring element neighbor locations and heights

**Syntax** `[offsets,heights] = getneighbors(SE)`

**Description** `[offsets,heights] = getneighbors(SE)` returns the relative locations and corresponding heights for each of the neighbors in the structuring element SE.

`offsets` is a P-by-N array where P is the number of neighbors in the structuring element and N is the dimensionality of the structuring element. Each row of `offsets` contains the location of the corresponding neighbor, relative to the center of the structuring element.

`heights` is a P-element column vector containing the height of each structuring element neighbor.

**Class Support** SE is a STREL object. The return values `offsets` and `heights` are arrays of double-precision values.

## Examples

```
se = strel([1 0 1],[5 0 -5])
[offsets,heights] = getneighbors(se)
se =
Nonflat STREL object containing 2 neighbors.
```

```
Neighborhood:
    1    0    1
```

```
Height:
    5    0   -5
```

```
offsets =
    0   -1
    0    1
heights =
    5   -5
```

**See Also** `strel`, `getnhood`, `getheight`

<b>Purpose</b>	Structuring element neighborhood
<b>Syntax</b>	<code>NHOOD = getnhood(SE)</code>
<b>Description</b>	<code>NHOOD = getnhood(SE)</code> returns the neighborhood associated with the structuring element SE.
<b>Class Support</b>	SE is a STREL object. NHOOD is a logical array.
<b>Examples</b>	<pre>se = strel(eye(5)); NHOOD = getnhood(se)</pre>
<b>See Also</b>	<code>strel</code> , <code>getneighbors</code>

# getpts

---

**Purpose** Specify points with mouse

**Syntax**  
`[x,y] = getpts(fig)`  
`[x,y] = getpts(ax)`  
`[x,y] = getpts`

**Description** `[x,y] = getpts(fig)` lets you choose a set of points in the current axes of figure `fig` using the mouse. Coordinates of the selected points are returned in `X` and `Y`.

Use normal button clicks to add points. A shift-, right-, or double-click adds a final point and ends the selection. Pressing **Return** or **Enter** ends the selection without adding a final point. Pressing **Backspace** or **Delete** removes the previously selected point.

`[x,y] = getpts(ax)` lets you choose points in the axes specified by the handle `ax`.

`[x,y] = getpts` is the same as `[x,y] = getpts(gcf)`.

**See Also** `getline`, `getrect`

<b>Purpose</b>	Default display range of image based on its class
<b>Syntax</b>	<code>range = getrangefromclass(I)</code>
<b>Description</b>	<code>range = getrangefromclass(I)</code> returns the default display range of the image <code>I</code> , based on its class type. The function returns <code>range</code> , a two-element vector specifying the display range in the form <code>[min max]</code> .
<b>Class Support</b>	<code>I</code> can be <code>uint8</code> , <code>uint16</code> , <code>int16</code> , <code>logical</code> , <code>single</code> , or <code>double</code> . <code>range</code> is of class <code>double</code> .
<b>Note</b>	For <code>single</code> and <code>double</code> data, <code>getrangefromclass</code> returns the range <code>[0 1]</code> to be consistent with the way <code>double</code> and <code>single</code> images are interpreted in MATLAB. For integer data, <code>getrangefromclass</code> returns the default display range of the class. For example, if the class is <code>uint8</code> , the dynamic range is <code>[0 255]</code> .
<b>Examples</b>	Read in the 16-bit DICOM image and get the default display range. <pre>CT = dicomread('CT-MON02-16-ankle.dcm'); r = getrangefromclass(CT) r =             -32768           32767</pre>
<b>See Also</b>	<code>intmin</code> , <code>intmax</code>

# getrect

---

**Purpose** Specify rectangle with mouse

**Syntax** `rect = getrect(fig)`  
`rect = getrect(ax)`

**Description** `rect = getrect(fig)` lets you select a rectangle in the current axes of figure `fig` using the mouse.

Use the mouse to click and drag the desired rectangle. `rect` is a four-element vector with the form `[xmin ymin width height]`. To constrain the rectangle to be a square, use a shift- or right-click to begin the drag.

`rect = getrect(ax)` lets you select a rectangle in the axes specified by the handle `ax`.

**See Also** `getline`, `getpts`

**Purpose** Sequence of decomposed structuring elements

**Syntax** SEQ = getsequence(SE)

**Description** SEQ = getsequence(SE) returns the array of structuring elements SEQ, containing the individual structuring elements that form the decomposition of SE. SE can be an array of structuring elements. SEQ is equivalent to SE, but the elements of SEQ have no decomposition.

**Class Support** SE and SEQ are arrays of STREL objects.

**Examples** The strel function uses decomposition for square structuring elements larger than 3-by-3. Use getsequence to extract the decomposed structuring elements.

```
se = strel('square',5)
se =
Flat STREL object containing 25 neighbors.
Decomposition: 2 STREL objects containing a total of 10 neighbors
```

Neighborhood:

```
  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
```

```
seq = getsequence(se)
seq =
2x1 array of STREL objects
```

Use imdilate with the 'full' option to see that dilating sequentially with the decomposed structuring elements really does form a 5-by-5 square:

```
imdilate(1,seq,'full')
```

# getsequence

---

**See Also**      imdilate, imerode, strel



**Purpose**

Convert grayscale or binary image to indexed image

**Syntax**

```
[X,map] = gray2ind(I,n)  
[X,map] = gray2ind(BW,n)
```

**Description**

`[X,map] = gray2ind(I,n)` converts the grayscale image `I` to an indexed image `X`. `n` specifies the size of the colormap, `gray(n)`. `n` must be an integer between 1 and 65536. If `n` is omitted, it defaults to 64.

`[X,map] = gray2ind(BW,n)` converts the binary image `BW` to an indexed image `X`. `n` specifies the size of the colormap, `gray(n)`. If `n` is omitted, it defaults to 2.

`gray2ind` scales and then rounds the intensity image to produce an equivalent indexed image.

**Class Support**

The input image `I` can be `logical`, `uint8`, `uint16`, `int16`, `single`, or `double` and must be a real and nonsparse. The image `I` can have any dimension. The class of the output image `X` is `uint8` if the colormap length is less than or equal to 256; otherwise it is `uint16`.

**Examples**

Convert a grayscale image into an indexed image and then view the result.

```
I = imread('cameraman.tif');  
[X, map] = gray2ind(I, 16);  
imshow(X, map);
```

**See Also**

`grayslice`, `ind2gray`, `mat2gray`

# graycomatrix

---

**Purpose** Create gray-level co-occurrence matrix from image

**Syntax**

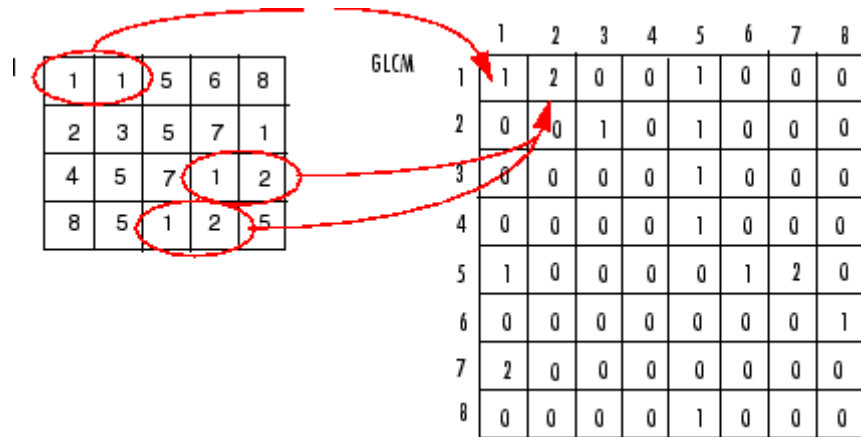
```
glcm = graycomatrix(I)
glcms = graycomatrix(I,param1,val1,param2,val2,...)
[glcms,SI] = graycomatrix(...)
```

**Description**

`glcm = graycomatrix(I)` creates a gray-level co-occurrence matrix (GLCM) from image `I`. `graycomatrix` creates the GLCM by calculating how often a pixel with gray-level (grayscale intensity) value  $i$  occurs horizontally adjacent to a pixel with the value  $j$ . (You can specify other pixel spatial relationships using the 'Offsets' parameter -- see Parameters.) Each element  $(i,j)$  in `glcm` specifies the number of times that the pixel with value  $i$  occurred horizontally adjacent to a pixel with value  $j$ .

`graycomatrix` calculates the GLCM from a scaled version of the image. By default, if `I` is a binary image, `graycomatrix` scales the image to two gray-levels. If `I` is an intensity image, `graycomatrix` scales the image to eight gray-levels. You can specify the number of gray-levels `graycomatrix` uses to scale the image by using the 'NumLevels' parameter, and the way that `graycomatrix` scales the values using the 'GrayLimits' parameter — see Parameters.

The following figure shows how `graycomatrix` calculates several values in the GLCM of the 4-by-5 image `I`. Element (1,1) in the GLCM contains the value 1 because there is only one instance in the image where two, horizontally adjacent pixels have the values 1 and 1. Element (1,2) in the GLCM contains the value 2 because there are two instances in the image where two, horizontally adjacent pixels have the values 1 and 2. `graycomatrix` continues this processing to fill in all the values in the GLCM.



`glcms = graycomatrix(I,param1,val1,param2,val2,...)` returns one or more gray-level co-occurrence matrices, depending on the values of the optional parameter/value pairs. Parameter names can be abbreviated, and case does not matter.

## Parameters

The following table lists these parameters in alphabetical order.

Parameter	Description	Default
'GrayLimits'	Two-element vector, [low high], that specifies how the grayscale values in I are linearly scaled into gray levels. Grayscale values less than or equal to low are scaled to 1. Grayscale values greater than or equal to high are scaled to NumLevels. If graylimits is set to [], graycomatrix uses the minimum and maximum grayscale values in the image as limits, [min(I(:)) max(I(:))].	Minimum and maximum specified by class, e.g. double [0 1] int16 [-32768 32767]

# graycomatrix

Parameter	Description	Default										
'NumLevels'	Integer specifying the number of gray-levels to use when scaling the grayscale values in I. For example, if NumLevels is 8, graycomatrix scales the values in I so they are integers between 1 and 8. The number of gray-levels determines the size of the gray-level co-occurrence matrix (glcm).	8 (numeric) 2 (binary)										
'Offset'	<p>p-by-2 array of integers specifying the distance between the pixel of interest and its neighbor. Each row in the array is a two-element vector, [row_offset, col_offset], that specifies the relationship, or <i>offset</i>, of a pair of pixels. row_offset is the number of rows between the pixel-of-interest and its neighbor. col_offset is the number of columns between the pixel-of-interest and its neighbor. Because the offset is often expressed as an angle, the following table lists the offset values that specify common angles, given the pixel distance D.</p> <table><thead><tr><th>Angle</th><th>Offset</th></tr></thead><tbody><tr><td>0</td><td>[0 D]</td></tr><tr><td>45</td><td>[-D D]</td></tr><tr><td>90</td><td>[-D 0]</td></tr><tr><td>135</td><td>[-D -D]</td></tr></tbody></table> <p>The figure illustrates the array: offset = [0 1; -1 1; -1 0; -1 -1]</p>	Angle	Offset	0	[0 D]	45	[-D D]	90	[-D 0]	135	[-D -D]	[0 1]
Angle	Offset											
0	[0 D]											
45	[-D D]											
90	[-D 0]											
135	[-D -D]											

Parameter	Description	Default
	<p>Diagram illustrating the offsets for the gray-level co-occurrence matrix (GLCM) calculation. The center pixel is shaded and labeled "Pixel-of-interest". The offsets are shown as arrows pointing to adjacent pixels: 0° [0 1] (right), 45° [-1 1] (up-right), 90° [-1 0] (up), and 135° [-1 -1] (up-left).</p>	
'Symmetric'	<p>A Boolean that creates a GLCM where the ordering of values in the pixel pairs is not considered. For example, when calculating the number of times the value 1 is adjacent to the value 2, graycomatrix counts both 1,2 and 2,1 pairings, if 'Symmetric' is set to true. When 'Symmetric' is set to false, graycomatrix only counts 1,2 or 2,1, depending on the value of 'offset'. The GLCM created when 'Symmetric' is set to true is symmetric across its diagonal, and is equivalent to the GLCM described by Haralick (1973). See Notes below for more information.</p>	false

[glcm, SI] = graycomatrix(...) returns the scaled image, SI, used to calculate the gray-level co-occurrence matrix. The values in SI are between 1 and NumLevels.

## Class Support

I can be numeric or logical but must be two-dimensional, real, and nonsparse. SI is a double matrix having the same size as I. glcms is a 'NumLevels'-by-'NumLevels'-by-P double array where P is the number of offsets in 'Offset'.

## Notes

Another name for a gray-level co-occurrence matrix is a gray-level spatial dependence matrix. Also, the word co-occurrence is frequently used in the literature without a hyphen, cooccurrence.

graycomatrix ignores pixel pairs if either of the pixels contains a NaN.

# graycomatrix

---

graycomatrix replaces positive Infs with the value NumLevels and replaces negative Infs with the value 1.

graycomatrix ignores border pixels, if the corresponding neighbor pixel falls outside the image boundaries.

The GLCM produced by the following syntax, with 'Symmetric' set to true

```
graycomatrix(I, 'offset', [0 1], 'Symmetric', true)
```

is equivalent to the sum of the two GLCMs produced by these statements where 'Symmetric' is set to false.

```
graycomatrix(I, 'offset', [0 1], 'Symmetric', false)
graycomatrix(I, 'offset', [0 -1], 'Symmetric', false)
```

## Examples

Calculate the gray-level co-occurrence matrix for a grayscale image.

```
I = imread('circuit.tif');
glcm = graycomatrix(I, 'Offset', [2 0]);
```

Calculate the gray-level co-occurrence matrix and return the scaled version of the image, SI, used by graycomatrix to generate the GLCM.

```
I = [ 1 1 5 6 8 8; 2 3 5 7 0 2; 0 2 3 5 6 7];
[glcm,SI] = graycomatrix(I, 'NumLevels',9, 'G', [])
```

## See Also

graycoprops

## References

Haralick, R.M., K. Shanmugan, and I. Dinstein, "Textural Features for Image Classification", IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-3, 1973, pp. 610-621.

Haralick, R.M., and L.G. Shapiro. Computer and Robot Vision: Vol. 1, Addison-Wesley, 1992, p. 459.

**Purpose** Properties of gray-level co-occurrence matrix

**Syntax** stats = graycoprops(glcm, properties)

**Description** stats = graycoprops(glcm, properties) calculates the statistics specified in properties from the gray-level co-occurrence matrix glcm. glcm is an *m-by-n-by-p* array of valid gray-level co-occurrence matrices. If glcm is an array of GLCMs, stats is an array of statistics for each glcm.

graycoprops normalizes the gray-level co-occurrence matrix (GLCM) so that the sum of its elements is equal to 1. Each element (r,c) in the normalized GLCM is the joint probability occurrence of pixel pairs with a defined spatial relationship having gray level values r and c in the image. graycoprops uses the normalized GLCM to calculate properties.

properties can be a comma-separated list of strings, a cell array containing strings, the string 'all', or a space separated string. The property names can be abbreviated and are not case sensitive.

Property	Description	Formula
'Contrast'	Returns a measure of the intensity contrast between a pixel and its neighbor over the whole image.  Range = [0 (size(GLCM,1)-1)^2]  Contrast is 0 for a constant image.	$\sum_{i,j}  i-j ^2 p(i,j)$

Property	Description	Formula
'Correlation'	Returns a measure of how correlated a pixel is to its neighbor over the whole image.  Range = [-1 1]  Correlation is 1 or -1 for a perfectly positively or negatively correlated image. Correlation is NaN for a constant image.	$\frac{\sum_{i,j} (i - \mu_i)(j - \mu_j) p(i, j)}{\sigma_i \sigma_j}$
'Energy'	Returns the sum of squared elements in the GLCM.  Range = [0 1]  Energy is 1 for a constant image.	$\sum_{i,j} p(i, j)^2$
'Homogeneity'	Returns a value that measures the closeness of the distribution of elements in the GLCM to the GLCM diagonal.  Range = [0 1]  Homogeneity is 1 for a diagonal GLCM.	$\sum_{i,j} \frac{p(i, j)}{1 +  i - j }$

stats is a structure with fields that are specified by properties. Each field contains a 1 x p array, where p is the number of gray-level co-occurrence matrices in GLCM. For example, if GLCM is an 8 x 8 x 3 array and properties is 'Energy', then stats is a structure containing the field Energy, which contains a 1 x 3 array.

## Notes

Energy is also known as uniformity, uniformity of energy, and angular second moment.

Contrast is also known as variance and inertia.



**Class Support**

glcm can be logical or numeric, and it must contain real, non-negative, finite, integers. stats is a structure.

**Examples**

```
GLCM = [0 1 2 3;1 1 2 3;1 0 2 0;0 0 0 3];  
stats = graycoprops(GLCM)
```

```
I = imread('circuit.tif');  
GLCM2 = graycomatrix(I,'Offset',[2 0;0 2]);  
stats = graycoprops(GLCM2,{'contrast','homogeneity'})
```

**See Also**

graycomatrix

# grayslice

---

**Purpose** Convert grayscale image to indexed image using multilevel thresholding

**Syntax** `X = grayslice(I,n)`  
`X = grayslice(I,v)`

**Description** `X = grayslice(I,n)` thresholds the intensity image `I` returning an indexed image in `X`. `grayslice` uses the threshold values:

$$\frac{1}{n}, \frac{2}{n}, \dots, \frac{n-1}{n}$$

`X = grayslice(I,v)` thresholds the intensity image `I` using the values of `v`, where `v` is a vector of values between 0 and 1, returning an indexed image in `X`.

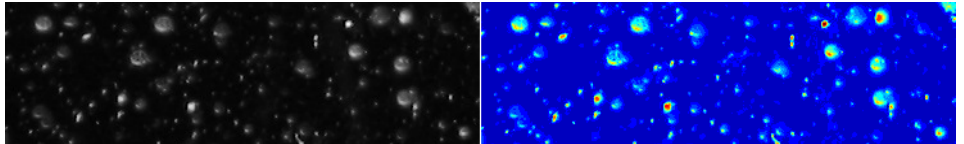
You can view the thresholded image using `imshow(X,map)` with a `colormap` of appropriate length.

**Class Support** The input image `I` can be of class `uint8`, `uint16`, `int16`, `single`, or `double`, and must be nonsparse. 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`.

## Examples

```
I = imread('snowflakes.png');  
X = grayslice(I,16);  
imshow(I)  
figure, imshow(X,jet(16))
```



## See Also

[gray2ind](#)

# graythresh

---

**Purpose** Global image threshold using Otsu's method

**Syntax** `level = graythresh(I)`  
`[level EM] = graythresh(I)`

**Description** `level = graythresh(I)` computes a global threshold (`level`) that can be used to convert an intensity image to a binary image with `im2bw`. `level` is a normalized intensity value that lies in the range `[0, 1]`.

The `graythresh` function uses Otsu's method, which chooses the threshold to minimize the intraclass variance of the black and white pixels.

Multidimensional arrays are converted automatically to 2-D arrays using `reshape`. The `graythresh` function ignores any nonzero imaginary part of `I`.

`[level EM] = graythresh(I)` returns the effectiveness metric, `EM`, as the second output argument. The effectiveness metric is a value in the range `[0 1]` that indicates the effectiveness of the thresholding of the input image. The lower bound is attainable only by images having a single gray level, and the upper bound is attainable only by two-valued images.

**Class Support** The input image `I` can be of class `uint8`, `uint16`, `int16`, `single`, or `double` and it must be nonsparse. The return value `level` is a double scalar. The effectiveness metric `EM` is a double scalar.

**Examples**

```
I = imread('coins.png');  
level = graythresh(I);  
BW = im2bw(I,level);  
imshow(BW)
```

**See Also** `im2bw`

**Reference** [1] Otsu, N., "A Threshold Selection Method from Gray-Level Histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 9, No. 1, 1979, pp. 62-66.

**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 grayscale transformation that maps gray levels in the 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`.

The `histeq` function 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 syntax that include an intensity image `I` as input, `I` can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. The output image `J` has the same class as `I`.

For syntax that include an indexed image `X` as input, `X` can be of class `uint8`, `single`, or `double`; the output colormap is always of class `double`. The optional output `T` (the gray-level transform) is always of class `double`.

## Examples

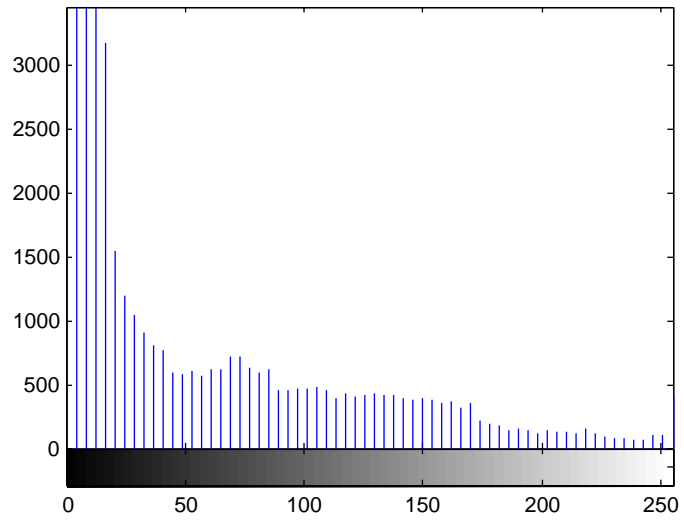
Enhance the contrast of an intensity image using histogram equalization.

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



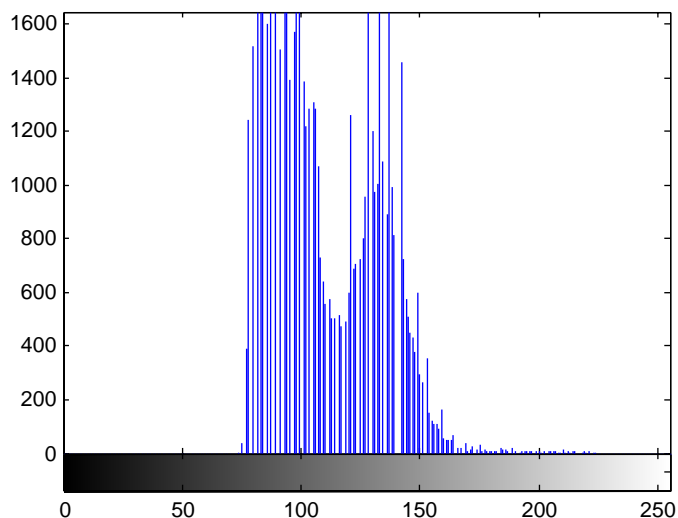
Display a histogram of the original image.

```
figure; imhist(I,64)
```



Compare it to a histogram of the processed image.

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



## Algorithm

When you supply a desired histogram `hgram`, `histeq` chooses the grayscale transformation  $T$  to minimize

$$|c_1(T(k)) - c_0(k)|$$

where  $c_0$  is the cumulative histogram of  $A$ ,  $c_1$  is the cumulative sum of `hgram` for all intensities  $k$ . This minimization is subject to the constraints that  $T$  must be monotonic and  $c_1(T(a))$  cannot overshoot  $c_0(a)$  by more than half the distance between the histogram counts at  $a$ . `histeq` uses this transformation to map the gray levels in  $X$  (or the colormap) to their new values.

$$b = T(a)$$

If you do not specify `hgram`, `histeq` creates a flat `hgram`,

$$\text{hgram} = \text{ones}(1,n) * \text{prod}(\text{size}(A)) / n;$$



and then applies the previous algorithm.

**See Also**

brighten, imadjust, imhist

# hough

---

**Purpose** Hough transform

**Syntax** [H, theta, rho] = hough(bw)

**Description** [H, theta, rho] = hough(BW) computes the Standard Hough Transform (SHT) of the binary image BW. You can use the hough function to detect lines in an image. The function returns H, the Hough transform matrix. theta (in degrees) and rho are the arrays of rho and theta values over which the Hough transform matrix was generated.

[H, theta, rho] = hough(BW,param1,val1,param2,val2) specifies parameter/value pairs, listed in the following table. Parameter names can be abbreviated, and case does not matter.

Parameter	Description
'ThetaResolution'	Real scalar value between 0 and 90, exclusive, that specifies the spacing (in degrees) of the Hough transform bins along the theta axis. Default: 1.
'RhoResolution'	Real scalar value between 0 and <code>norm(size(BW))</code> , exclusive, that specifies the spacing of the Hough transform bins along the rho axis. Default: 1

**Notes** The hough function implements the Standard Hough Transform (SHT). The SHT uses the parametric representation of a line:

$$\text{rho} = x \cdot \cos(\text{theta}) + y \cdot \sin(\text{theta})$$

The variable rho is the distance from the origin to the line along a vector perpendicular to the line. theta is the angle between the x-axis and this vector. The hough function generates a parameter space matrix whose rows and columns correspond to rho and theta values respectively. Peak values in this space represent potential lines in the input image.

The Hough transform matrix,  $H$ , is  $NRHO$ -by- $NTHETA$  where  $NRHO = 2 * \text{ceil}(\text{norm}(\text{size}(BW)) / \text{RhoResolution}) - 1$ , and  $NTHETA = 2 * \text{ceil}(90 / \text{ThetaResolution})$ . Theta angle values are in the range  $[-90, 90)$  degrees and rho values range from  $-DIAGONAL$  to  $DIAGONAL$  where  $DIAGONAL = \text{RhoResolution} * \text{ceil}(\text{norm}(\text{size}(BW)) / \text{RhoResolution})$ . Note that if  $90 / DTHETA$  is not an integer, the actual angle spacing will be  $90 / \text{ceil}(90 / DTHETA)$ .

## Class Support

$BW$  can be logical or numeric and it must be real, 2-D, and nonsparse.

## Examples

Compute and display the Hough transform of an image

```

RGB = imread('gantrycrane.png');
I = rgb2gray(RGB); % convert to intensity
BW = edge(I,'canny'); % extract edges
[H,T,R] = hough(BW,'RhoResolution',0.5,'ThetaResolution',0.5);

% display the original image
subplot(2,1,1);
imshow(RGB);
title('gantrycrane.png');

% display the hough matrix
subplot(2,1,2);
imshow(imadjust(mat2gray(H)), 'XData',T, 'YData',R,...
       'InitialMagnification','fit');
title('Hough transform of gantrycrane.png');
xlabel('\theta'), ylabel('\rho');
axis on, axis normal, hold on;
colormap(hot);

```

## See also

houghpeaks, houghlines

# houghlines

---

**Purpose** Extract line segments based on Hough transform

**Syntax**  
`lines = houghlines(BW,theta, rho, peaks)`  
`lines = houghlines(...,param1,val1,param2,val2)`

**Description** `lines = houghlines(BW,theta, rho, peaks)` extracts line segments in the image `BW` associated with particular bins in a Hough transform. `theta` and `rho` are vectors returned by function `hough`. `peaks` is a matrix returned by the `houghpeaks` function that contains the row and column coordinates of the Hough transform bins to use in searching for line segments.

The `houghlines` function returns `lines`, a structure array whose length equals the number of merged line segments found. Each element of the structure array has these fields:

Field	Description
<code>point1</code>	Two element vector [X Y] specifying the coordinates of the end-point of the line segment
<code>point2</code>	Two element vector [X Y] specifying the coordinates of the end-point of the line segment
<code>theta</code>	Angle in degrees of the Hough transform bin
<code>rho</code>	rho axis position of the Hough transform bin

`lines = houghlines(...,param1,val1,param2,val2)` specifies parameter/value pairs, listed in the following table. Parameter names can be abbreviated, and case does not matter.

Parameter	Description
'FillGap'	Positive real scalar value that specifies the distance between two line segments associated with the same Hough transform bin. When the distance between the line segments is less the value specified, the houghlines function merges the line segments into a single line segment. Default: 20
'MinLength'	Positive real scalar value that specifies whether merged lines should be kept or discarded. Lines shorter than the value specified are discarded. Default: 40

## Class Support

BW can be logical or numeric and it must be real, 2-D, and nonsparse.

## Examples

Search for line segments in an image and highlight the longest segment.

```
I = imread('circuit.tif');
rotI = imrotate(I,33,'crop');
BW = edge(rotI,'canny');
[H,T,R] = hough(BW);
imshow(H,[],'XData',T,'YData',R,...
        'InitialMagnification','fit');
xlabel('\theta'), ylabel('\rho');
axis on, axis normal, hold on;
P = houghpeaks(H,5,'threshold',ceil(0.3*max(H(:))));
x = T(P(:,2)); y = R(P(:,1));
plot(x,y,'s','color','white');
% Find lines and plot them
lines = houghlines(BW,T,R,P,'FillGap',5,'MinLength',7);
figure, imshow(rotI), hold on
max_len = 0;
for k = 1:length(lines)
    xy = [lines(k).point1; lines(k).point2];
    plot(xy(:,1),xy(:,2),'LineWidth',2,'Color','green');
```

# houghlines

---

```
% Plot beginnings and ends of lines
plot(xy(1,1),xy(1,2),'x','LineWidth',2,'Color','yellow');
plot(xy(2,1),xy(2,2),'x','LineWidth',2,'Color','red');

% Determine the endpoints of the longest line segment
len = norm(lines(k).point1 - lines(k).point2);
if ( len > max_len)
    max_len = len;
    xy_long = xy;
end
end

% highlight the longest line segment
plot(xy_long(:,1),xy_long(:,2),'LineWidth',2,'Color','cyan');
```

## See also

hough, houghpeaks

**Purpose** Identify peaks in Hough transform

**Syntax**

```
peaks = houghpeaks(H,numpeaks)
peaks = houghpeaks(...,param1,va11,param2,va12)
```

**Description** `peaks = houghpeaks(H,numpeaks)` locates peaks in the Hough transform matrix, H, generated by the `hough` function. `numpeaks` is a scalar value that specifies the maximum number of peaks to identify. If you omit `numpeaks`, it defaults to 1.

The function returns `peaks`, a Q-by-2 matrix, where Q can range from 0 to `numpeaks`. Q holds the row and column coordinates of the peaks.

`peaks = houghpeaks(...,param1,va11,param2,va12)` specifies parameter/value pairs, listed in the following table. Parameter names can be abbreviated, and case does not matter.

Parameter	Description
'Threshold'	Nonnegative scalar value that specifies the threshold at which values of H are considered to be peaks. Threshold can vary from 0 to Inf. Default is $0.5 * \max(H(:))$ .
'NHoodSize'	Two-element vector of positive odd integers: [M N]. 'NHoodSize' specifies the size of the suppression neighborhood. This is the neighborhood around each peak that is set to zero after the peak is identified. Default: smallest odd values greater than or equal to $\text{size}(H)/50$ .

**Class Support** H is the output of the `hough` function. `numpeaks` is a positive integer scalar.

**Examples** Locate and display two peaks in the Hough transform of a rotated image.

```
I = imread('circuit.tif');
BW = edge(imrotate(I,50,'crop'),'canny');
```

# houghpeaks

---

```
[H,T,R] = hough(BW);  
P = houghpeaks(H,2);  
imshow(H,[],'XData',T,'YData',R,'InitialMagnification','fit');  
xlabel('\theta'), ylabel('\rho');  
axis on, axis normal, hold on;  
plot(T(P(:,2)),R(P(:,1)),'s','color','white');
```

## See also

hough, houghlines



**Purpose** Convert hue-saturation-value (HSV) values to RGB color space

**Note** hsv2rgb is a function in MATLAB.

# iccfind

---

**Purpose** Search for ICC profiles

**Syntax**

```
P = iccfind(directory)
[P, descriptions] = iccfind(directory)
[...] = iccfind(directory, pattern)
```

**Description** `P = iccfind(directory)` searches for all of the ICC profiles found in the directory specified by `directory`. The function returns `P`, a cell array of structures containing profile information.

`[P, descriptions] = iccfind(directory)` searches for all of the ICC profiles in the specified directory and returns `P`, a cell array of structures containing profile information, and `descriptions`, a cell array of text strings, where each string describes the corresponding profile in `P`. Each text string is the value of the `Description.String` field in the profile information structure.

`[...] = iccfind(directory, pattern)` returns all of the ICC profiles in the specified directory with a given pattern in their `Description.String` fields. `iccfind` performs case-insensitive pattern matching.

---

**Note** To improve performance, `iccfind` caches copies of the ICC profiles in memory. Adding or modifying profiles might not change the results of `iccfind`. To clear the cache, use the `clear` functions command.

---

**Examples** Get all the ICC profiles in the default system directory where profiles are stored.

```
profiles = iccfind(iccroot);
```

Get a listing of all the ICC profiles with text strings that describe each profile.

```
[profiles, descriptions ] = iccfind(profiles);
```

Find the profiles whose descriptions contain the text string RGB.

```
[profiles, descriptions] = iccfind(iccroot, 'rgb');
```

## See Also

`iccread`, `iccroot`, `iccwrite`

# iccread

---

**Purpose** Read ICC profile

**Syntax** `P = iccread(filename)`

**Description** `P = iccread(filename)` reads the International Color Consortium (ICC) color profile information from the file specified by `filename`. The file can be either an ICC profile file or a TIFF file containing an embedded ICC profile. To determine if a TIFF file contains an embedded ICC profile, use the `imfinfo` function to get information about the file and look for the `ICCProfileOffset` field. `iccread` looks for the file in the current directory, a directory on the MATLAB path, or in the directory returned by `iccroot`, in that order.

`iccread` returns the profile information in the structure `P`, a 1-by-1 structure array whose fields contain the data structures (called tags) defined in the ICC specification. `iccread` can read profiles that conform with either Version 2 (ICC.1:2001-04) or Version 4 (ICC.1:2001-12) of the ICC specification. For more information about ICC profiles, visit the ICC web site, [www.color.org](http://www.color.org).

ICC profiles provide color management systems with the information necessary to convert color data between native device color spaces and device independent color spaces, called the Profile Connection Space (PCS). You can use the profile as the source or destination profile with the `makecform` function to compute color space transformations.

The number of fields in `P` depends on the profile class and the choices made by the profile creator. `iccread` returns all the tags for a given profile, both public and private. Private tags and certain public tags are left as encoded `uint8` data. The following table lists fields that are found in any profile structure generated by `iccread`, in the order they appear in the structure.

Field	Data Type	Description
Header	1-by-1 struct array	Profile header fields
TagTable	n-by-3 cell array	Profile tag table
Copyright	Text string	Profile copyright notice
Description	1-by-1 struct array	The String field in this structure contains a text string describing the profile.
MediaWhitepoint	double array	XYZ tristimulus values of the device's media white point
PrivateTags	m-by-2 cell array	Contents of all the private tags or tags not defined in the ICC specifications. The tag signatures are in the first column, and the contents of the tags are in the second column. Note that iccread leaves the contents of these tags in unsigned 8-bit encoding.
Filename	Text string	Name of the file containing the profile

Additionally, P might contain one or more of the following transforms:

- Three-component, matrix-based transform: A simple transform that is often used to transform between the RGB and XYZ color spaces. If this transform is present, P contains a field called MatTRC.
- N-component LUT-based transform: A transform that is used for transforming between color spaces that have a more complex relationship. This type of transform is found in any of the following fields in P:

AToB0	BToA0	Preview0
AToB1	BToA1	Preview1
AToB2	BToA2	Preview2
AToB3	BToA3	Gamut

## Notes

Portions of the implementation of iccread are derived from the RSA Data Security, Inc., MD5 Message-Digest Algorithm.

## Examples

The example reads the ICC profile that describes a typical PC computer monitor.

```
P = iccread('sRGB.icm')

P =

    Header: [1x1 struct]
    TagTable: {17x3 cell}
    Copyright: 'Copyright (c) 1999 Hewlett-Packard Company'
    Description: [1x1 struct]
    MediaWhitePoint: [0.9505 1 1.0891]
    MediaBlackPoint: [0 0 0]
    DeviceMfgDesc: [1x1 struct]
    DeviceModelDesc: [1x1 struct]
    ViewingCondDesc: [1x1 struct]
    ViewingConditions: [1x1 struct]
    Luminance: [76.0365 80 87.1246]
    Measurement: [1x36 uint8]
    Technology: [115 105 103 32 0 0 0 0 67 82 84 32]
    MatTRC: [1x1 struct]
    PrivateTags: {}
    Filename: 'sRGB.icm'
```

The profile header provides general information about the profile, such as its class, color space, and PCS. For example, to determine the source color space, view the `ColorSpace` field in the Header structure.

`P.Header.ColorSpace`

`ans =`

`RGB`

**See Also**

`applycform, iccfind, iccroot, iccwrite, isicc, makecform`

# iccroot

---

**Purpose** Find system default ICC profile repository

**Syntax** `rootdir = iccroot`

**Description** `rootdir = iccroot` returns the system directory containing ICC profiles. Additional profiles can be stored in other directories, but this is the default location used by the color management system.

---

**Note** Only Windows and Mac OS X platforms are supported.

---

**Examples** Return information on all the profiles in the root directory.

```
iccfind(iccroot)
```

**See Also** `iccfind`, `iccread`, `iccwrite`



**Purpose** Write ICC color profile to disk file

**Syntax** `P_new = iccwrite(P, filename)`

**Description** `P_new = iccwrite(P, filename)` writes the International Color Consortium (ICC) color profile data in structure `P` to the file specified by `filename`.

`P` is a structure representing an ICC profile in the data format returned by `iccread` and used by `makecform` and `applycform` to compute color-space transformations. `P` must contain all the tags and fields required by the ICC profile specification. Some fields may be inconsistent, however, because of interactive changes to the structure. For instance, the tag table may not be correct because tags may have been added, deleted, or modified since the tag table was constructed. `iccwrite` makes any necessary corrections to the profile structure before writing it to the file and returns this corrected structure in `P_new`.

---

**Note** Because some applications use the profile description string in the ICC profile to present choices to users, the ICC recommends modifying the profile description string in the ICC profile data before writing the data to a file. Each profile should have a unique description string. For more information, see the example.

---

`iccwrite` can write the color profile data using either Version 2 (ICC.1:2001-04) or Version 4 (ICC.1:2001-12) of the ICC specification, depending on the value of the `Version` field in the file profile header. If any required fields are missing, `iccwrite` errors. For more information about ICC profiles, visit the ICC web site, [www.color.org](http://www.color.org).

---

**Note** `iccwrite` does not perform automatic conversions from one version of the ICC specification to another. Such conversions have to be done manually, by adding fields or modifying fields. Use `isicc` to validate a profile.

---

## Notes

Portions of the implementation of `iccwrite` are derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm.

## Examples

Read a profile into the MATLAB workspace and export the profile data to a new file. The example changes the profile description string in the profile data before writing the data to a file.

```
P = iccread('monitor.icm');  
  
P.Description.String  
  
ans =  
  
sgC4_050102_d50.pf  
  
P.Description.String = 'my new description';  
  
pmon = iccwrite(P, 'monitor2.icm');
```

## See Also

`applycform`, `iccread`, `isicc`, `makecform`

**Purpose** 2-D 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)` pads `A` with 0's to size `m`-by-`n` before transforming. If `[m n] < size(A)`, `idct2` crops `A` before transforming.

`B = idct2(A,[m n])` same as above.

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 numeric class. The output matrix `B` is of class `double`.

**Algorithm** `idct2` computes the two-dimensional inverse DCT using

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

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

**Examples** Create a DCT matrix.

```
RGB = imread('autumn.tif');
I = rgb2gray(RGB);
J = dct2(I);
imshow(log(abs(J)),[]), colormap(jet), colorbar
```

Set values less than magnitude 10 in the DCT matrix to zero, then reconstruct the image using the inverse DCT function `idct2`.

# idct2

---

```
J(abs(J)<10) = 0;  
K = idct2(J);  
figure, imshow(I)  
figure, imshow(K,[0 255])
```

## See Also

dct2, dctmtx, fft2, ifft2

## References

- [1] Jain, A. K., *Fundamentals of Digital Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1989, pp. 150-153.
- [2] Pennebaker, W. B., and J. L. Mitchell, *JPEG: Still Image Data Compression Standard*, New York, Van Nostrand Reinhold, 1993.

**Purpose** Inverse fan-beam transform

**Syntax**  
`I = ifanbeam(F,D)`  
`I = ifanbeam(...,param1,val1,param2,val2,...)`  
`[I,H] = ifanbeam(...)`

**Description** `I = ifanbeam(F,D)` reconstructs the image `I` from projection data in the two-dimensional array `F`. Each column of `F` contains fan-beam projection data at one rotation angle. `ifanbeam` assumes that the center of rotation is the center point of the projections, which is defined as `ceil(size(F,1)/2)`.

The fan-beam spread angles are assumed to be the same increments as the input rotation angles split equally on either side of zero. The input rotation angles are assumed to be stepped in equal increments to cover `[0:359]` degrees.

`D` is the distance from the fan-beam vertex to the center of rotation.

`I = ifanbeam(...,param1,val1,param2,val2,...)` specifies parameters that control various aspects of the `ifanbeam` reconstruction, described in the following table. Parameter names can be abbreviated, and case does not matter. Default values are in braces (`{}`).

Parameter	Description
'FanCoverage'	String specifying the range through which the beams are rotated. {'cycle'} — Rotate through the full range <code>[0,360)</code> . 'minimal' — Rotate the minimum range necessary to represent the object.
'FanRotationIncrement'	Positive real scalar specifying the increment of the rotation angle of the fan-beam projections, measured in degrees. See <code>fanbeam</code> for details.

Parameter	Description
'FanSensorGeometry'	<p>String specifying how sensors are positioned.</p> <p>'arc' — Sensors are spaced equally along a circular arc at distance D from the center of rotation. Default value is 'arc'</p> <p>'line' — Sensors are spaced equally along a line, the closest point of which is distance D from the center of rotation.</p> <p>See fanbeam for details.</p>
'FanSensorSpacing'	<p>Positive real scalar specifying the spacing of the fan-beam sensors. Interpretation of the value depends on the setting of 'FanSensorGeometry'.</p> <p>If 'FanSensorGeometry' is set to 'arc' (the default), the value defines the angular spacing in degrees. Default value is 1.</p> <p>If 'FanSensorGeometry' is 'line', the value specifies the linear spacing. Default value is 1. See fanbeam for details.</p>
'Filter'	<p>String specifying the name of a filter. See iradon for details.</p>
'FrequencyScaling'	<p>Scalar in the range (0,1] that modifies the filter by rescaling its frequency axis. See iradon for details.</p>

Parameter	Description
'Interpolation'	<p>Text string specifying the type of interpolation used between the parallel-beam and fan-beam data.</p> <p>'nearest' — Nearest-neighbor</p> <p>{ 'linear' } — Linear</p> <p>'spline' — Piecewise cubic spline</p> <p>'pchip' — Piecewise cubic Hermite (PCHIP)</p> <p>'cubic' — Same as 'pchip'</p>
'OutputSize'	<p>Positive scalar specifying the number of rows and columns in the reconstructed image.</p> <p>If 'OutputSize' is not specified, ifanbeam determines the size automatically.</p> <p>If you specify 'OutputSize', ifanbeam reconstructs a smaller or larger portion of the image, but does not change the scaling of the data.</p> <hr/> <p><b>Note</b> If the projections were calculated with the fanbeam function, the reconstructed image might not be the same size as the original image.</p> <hr/>

[I,H] = ifanbeam(...) returns the frequency response of the filter in the vector H.

## Notes

`ifanbeam` converts the fan-beam data to parallel beam projections and then uses the filtered back projection 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.

## Class Support

The input arguments, `F` and `D`, can be double or single. All other numeric input arguments must be double. The output arguments are double.

## Examples

### Example 1

This example creates a fanbeam transformation of the phantom head image and then calls the `ifanbeam` function to recreate the phantom image from the fanbeam transformation.

```
ph = phantom(128);  
d = 100;  
F = fanbeam(ph,d);  
I = ifanbeam(F,d);  
imshow(ph), figure, imshow(I);
```

### Example 2

This example illustrates use of the `ifanbeam` function with the `'fancoverage'` option set to `'minimal'`.

```
ph = phantom(128);  
P = radon(ph);  
[F,obeta,otheta] = para2fan(P,100,...  
    'FanSensorSpacing',0.5,...  
    'FanCoverage','minimal',...  
    'FanRotationIncrement',1);  
phReconstructed = ifanbeam(F,100,...  
    'FanSensorSpacing',0.5,...  
    'Filter','Shepp-Logan',...  
    'OutputSize',128,...
```



```
        'FanCoverage','minimal',...  
        'FanRotationIncrement',1);  
imshow(ph), figure, imshow(phReconstructed)
```

**See Also**

fan2para, fanbeam, iradon, para2fan, phantom, radon

**References**

[1] Kak, A. C., and M. Slaney, *Principles of Computerized Tomographic Imaging*, New York, NY, IEEE Press, 1988.

# ifft2

---

**Purpose**            2-D inverse fast Fourier transform

**Note**             `ifft2` is a function in MATLAB.

**Purpose** N-D inverse fast Fourier transform

**Note** `ifftn` is a function in MATLAB.

# im2bw

---

**Purpose** Convert image to 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 uses thresholding to convert this grayscale image to binary. The output binary image `BW` has values of 1 (white) for all pixels in the input image with luminance greater than `level` and 0 (black) 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.

---

**Note** The function `graythresh` can be used to compute the `level` argument automatically.

---

**Class Support** The input image can be of class `uint8`, `uint16`, `single`, `int16`, or `double`, and must be nonsparse. The output image `BW` is of class `logical`.

**Examples**

```
load trees
BW = im2bw(X,map,0.4);
imshow(X,map), figure, imshow(BW)
```

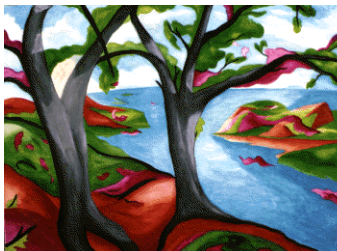


Image Courtesy of Susan Cohen

**See Also**

`graythresh`, `ind2gray`, `rgb2gray`

# im2col

**Purpose** Rearrange image blocks into columns

**Syntax**  
`B = im2col(A,[m n],block_type)`  
`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. The default value is enclosed in braces (`{}`).

Value	Description
'distinct'	Rearranges each <i>distinct</i> $m$ -by- $n$ block in the image $A$ into a column of $B$ . <code>im2col</code> pads $A$ with 0's, 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 $B = [A11(:) A12(:) A21(:) A22(:)]$ .
{'sliding'}	Converts each <i>sliding</i> $m$ -by- $n$ block of $A$ into a column of $B$ , with no zero padding. $B$ has $m*n$ rows and contains 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- $((mm-m+1)*(nn-n+1))$ .

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 Examples, 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 0's if the class of  $A$  is `uint8`, or 1's if the class of  $A$  is `double`.

**Class Support**

The input image A can be numeric or logical. The output matrix B is of the same class as the input image.

**See Also**

blkproc, col2im, colfilt, nlfilt

# im2double

---

**Purpose** Convert image to double precision

**Syntax**

```
I2 = im2double(I)
RGB2 = im2double(RGB)
I = im2double(BW)
X2 = im2double(X, '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 not `double`, `im2double` returns the equivalent image of class `double`, rescaling or offsetting the data as necessary.

`I2 = im2double(I)` converts the intensity image `I` to double precision, rescaling the data if necessary.

`RGB2 = im2double(RGB)` converts the truecolor image `RGB` to double precision, rescaling the data if necessary.

`I = im2double(BW)` converts the binary image `BW` to a double-precision intensity image.

`X2 = im2double(X, 'indexed')` converts the indexed image `X` to double precision, offsetting the data if necessary.

**Class Support** Intensity and truecolor images can be `uint8`, `uint16`, `double`, `logical`, `single`, or `int16`. Indexed images can be `uint8`, `uint16`, `double` or `logical`. Binary input images must be `logical`. The output image is `double`.

**Examples**

```
I1 = reshape(uint8(linspace(1,255,25)),[5 5])
I2 = im2double(I1)
```

**See Also** `double`, `im2single`, `im2int16`, `im2uint8`, `im2uint16`



<b>Purpose</b>	Convert image to 16-bit signed integers
<b>Syntax</b>	<pre>I2 = im2int16(I) RGB2 = im2int16(RGB) I = im2int16(BW)</pre>
<b>Description</b>	<p>im2int16 takes an image as input and returns an image of class int16. If the input image is of class int16, the output image is identical to it. If the input image is not of class int16, im2int16 returns the equivalent image of class int16, rescaling or offsetting the data as necessary.</p> <p>I2 = im2int16(I) converts the intensity image I to int16, rescaling the data if necessary.</p> <p>RGB2 = im2int16(RGB) converts the truecolor image RGB to int16, rescaling the data if necessary.</p> <p>I = im2int16(BW) converts the binary image BW to an int16 intensity image, changing false-valued elements to -32768 and true-valued elements to 32767.</p>
<b>Class Support</b>	Intensity and truecolor images can be uint8, uint16, int16, single, double, or logical. Binary input images must be logical. The output image is int16.
<b>Examples</b>	<pre>I = reshape(linspace(0,1,20),[5 4]) I2 = im2int16(I)</pre>
<b>See Also</b>	im2double, im2single, im2uint8, im2uint16, int16

# im2java

---

**Purpose** Convert image to Java image

**Note** `im2java` is a MATLAB function.

---

<b>Purpose</b>	Convert image to Java buffered image
<b>Syntax</b>	<pre>jimage = im2java2d(I) jimage = im2java2d(X,MAP)</pre>
<b>Description</b>	<p><code>jimage = im2java2d(I)</code> converts the image <code>I</code> to an instance of the Java image class <code>java.awt.image.BufferedImage</code>. The image <code>I</code> can be an intensity (grayscale), RGB, or binary image.</p> <p><code>jimage = im2java2d(X,MAP)</code> converts the indexed image <code>X</code> with colormap <code>MAP</code> to an instance of the Java class <code>java.awt.image.BufferedImage</code>.</p> <hr/> <p><b>Note</b> The <code>im2java2d</code> function works with the Java 2D API. The <code>im2java</code> function works with the Java Abstract Windowing Toolkit (AWT).</p> <hr/>
<b>Class Support</b>	Intensity, indexed, and RGB input images can be of class <code>uint8</code> , <code>uint16</code> , or <code>double</code> . Binary input images must be of class <code>logical</code> .
<b>Examples</b>	<p>Read an image into the MATLAB workspace and then use <code>im2java2d</code> to convert it into an instance of the Java class <code>java.awt.image.BufferedImage</code>.</p> <pre>I = imread('moon.tif'); javaImage = im2java2d(I); frame = javax.swing.JFrame; icon = javax.swing.ImageIcon(javaImage); label = javax.swing.JLabel(icon); frame.getContentPane.add(label); frame.pack frame.show</pre>

# im2single

---

**Purpose** Convert image to single precision

**Syntax**

```
I2 = im2single(I)
RGB2 = im2single(RGB)
I = im2single(BW)
X2 = im2single(X, 'indexed')
```

**Description** `im2single` takes an image as input and returns an image of class `single`. If the input image is of class `single`, the output image is identical to it. If the input image is not of class `single`, `im2single` returns the equivalent image of class `single`, rescaling or offsetting the data as necessary.

`I2 = im2single(I)` converts the intensity image `I` to `single`, rescaling the data if necessary.

`RGB2 = im2single(RGB)` converts the truecolor image `RGB` to `single`, rescaling the data if necessary.

`I = im2single(BW)` converts the binary image `BW` to a single-precision intensity image.

`X2 = im2single(X, 'indexed')` converts the indexed image `X` to single precision, offsetting the data if necessary.

**Class Support** Intensity and truecolor images can be `uint8`, `uint16`, `int16`, `single`, `double`, or `logical`. Indexed images can be `uint8`, `uint16`, `double` or `logical`. Binary input images must be `logical`. The output image is `single`.

**Examples**

```
I = reshape(uint8(linspace(1,255,25)), [5 5])
I2 = im2single(I)
```

**See Also** `im2double`, `im2int16`, `im2uint8`, `im2uint16`, `single`

<b>Purpose</b>	Convert image to 16-bit unsigned integers
<b>Syntax</b>	<pre>I2 = im2uint16(I) RGB2 = im2uint16(RGB) I = im2uint16(BW) X2 = im2uint16(X,'indexed')</pre>
<b>Description</b>	<p><code>im2uint16</code> takes an image as input and returns an image of class <code>uint16</code>. If the input image is of class <code>uint16</code>, the output image is identical to it. If the input image is not of class <code>uint16</code>, <code>im2uint16</code> returns the equivalent image of class <code>uint16</code>, rescaling or offsetting the data as necessary.</p> <p><code>I2 = im2uint16(I)</code> converts the intensity image <code>I</code> to <code>uint16</code>, rescaling the data if necessary.</p> <p><code>RGB2 = im2uint16(RGB)</code> converts the truecolor image <code>RGB</code> to <code>uint16</code>, rescaling the data if necessary.</p> <p><code>I = im2uint16(BW)</code> converts the binary image <code>BW</code> to a <code>uint16</code> intensity image, changing 1-valued elements to 65535.</p> <p><code>X2 = im2uint16(X,'indexed')</code> converts the indexed image <code>X</code> to <code>uint16</code>, offsetting the data if necessary. If <code>X</code> is of class <code>double</code>, <code>max(X(:))</code> must be 65536 or less.</p>
<b>Class Support</b>	Intensity and truecolor images can be <code>uint8</code> , <code>uint16</code> , <code>double</code> , <code>logical</code> , <code>single</code> , or <code>int16</code> . Indexed images can be <code>uint8</code> , <code>uint16</code> , <code>double</code> , or <code>logical</code> . Binary input images must be <code>logical</code> . The output image is <code>uint16</code> .
<b>Examples</b>	<pre>I = reshape(linspace(0,1,20),[5 4]) I2 = im2uint16(I)</pre>
<b>See Also</b>	<code>im2uint8</code> , <code>double</code> , <code>im2double</code> , <code>uint8</code> , <code>uint16</code> , <code>imapprox</code>

# im2uint8

---

**Purpose** Convert image to 8-bit unsigned integers

**Syntax**

```
I2 = im2uint8(I)
RGB2 = im2uint8(RGB)
I = im2uint8(BW)
X2 = im2uint8(X, '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 not of class `uint8`, `im2uint8` returns the equivalent image of class `uint8`, rescaling or offsetting the data as necessary.

`I2 = im2uint8(I)` converts the intensity image `I` to `uint8`, rescaling the data if necessary.

`RGB2 = im2uint8(RGB)` converts the truecolor image `RGB` to `uint8`, rescaling the data if necessary.

`I = im2uint8(BW)` converts the binary image `BW` to a `uint8` intensity image, changing 1-valued elements to 255

`X2 = im2uint8(X, 'indexed')` converts the indexed image `X` to `uint8`, offsetting the data if necessary. Note that it is not always possible to convert an indexed image to `uint8`. If `X` is of class `double`, the maximum value of `X` must be 256 or less; if `X` is of class `uint16`, the maximum value of `X` must be 255 or less.

**Class Support** Intensity and truecolor images can be `uint8`, `uint16`, `double`, `logical`, `single`, or `int16`. Indexed images can be `uint8`, `uint16`, `double`, or `logical`. Binary input images must be `logical`. The output image is `uint8`.

**Examples**

```
I = reshape(uint8(linspace(0,255,255)),[5 5])
I2 = im2uint8(I)
```

**See Also** `im2double`, `im2int16`, `im2single`, `im2uint16`, `uint8`

**Purpose** Absolute difference of two images

**Syntax** `Z = imabsdiff(X,Y)`

**Description** `Z = imabsdiff(X,Y)` subtracts each element in array `Y` from the corresponding element in array `X` and returns the absolute difference in the corresponding element of the output array `Z`. `X` and `Y` are real, nonsparse numeric arrays with the same class and size. `Z` has the same class and size as `X` and `Y`. If `X` and `Y` are integer arrays, elements in the output that exceed the range of the integer type are truncated.

If `X` and `Y` are double arrays, you can use the expression `abs(X-Y)` instead of this function.

---

**Note** On Intel architecture processors, `imabsdiff` can take advantage of the Intel Performance Primitives Library (IPPL), thus accelerating its execution time. IPPL is activated only if arrays `X`, `Y`, and `Z` are of class `logical`, `uint8`, or `single`, and are of the same class.

---

## Examples

Calculate the absolute difference between two `uint8` arrays. Note that the absolute value prevents negative values from being rounded to zero in the result, as they are with `imsubtract`.

```
X = uint8([ 255 10 75; 44 225 100]);
Y = uint8([ 50 50 50; 50 50 50 ]);
Z = imabsdiff(X,Y)
```

```
Z =
    205     40     25
     6    175     50
```

Display the absolute difference between a filtered image and the original.

```
I = imread('cameraman.tif');
J = uint8(filter2(fspecial('gaussian'), I));
```

# imabsdiff

---

```
K = imabsdiff(I,J);  
imshow(K,[]) % [] = scale data automatically
```

## See Also

[imadd](#), [imcomplement](#), [imdivide](#), [imlincomb](#), [immultiply](#), [imsubtract](#),  
[ippl](#)



**Purpose** Add two images or add constant to image

**Syntax** `Z = imadd(X,Y)`

**Description** `Z = imadd(X,Y)` adds each element in array `X` with the corresponding element in array `Y` and returns the sum in the corresponding element of the output array `Z`. `X` and `Y` are real, nonsparse numeric arrays with the same size and class, or `Y` is a scalar double. `Z` has the same size and class as `X`, unless `X` is logical, in which case `Z` is double.

If `X` and `Y` are integer arrays, elements in the output that exceed the range of the integer type are truncated, and fractional values are rounded.

---

**Note** On Intel architecture processors, `imadd` can take advantage of the Intel Performance Primitives Library (IPPL), thus accelerating its execution time. IPPL is activated if arrays `X`, `Y`, and `Z` are of class `logical`, `uint8`, or `single` and are of the same class. IPPL is also activated if `Y` is a double scalar and arrays `X` and `Z` are `uint8`, `int16`, or `single` and are of the same class.

---

**Examples** Add two `uint8` arrays. Note the truncation that occurs when the values exceed 255.

```
X = uint8([ 255 0 75; 44 225 100]);
Y = uint8([ 50 50 50; 50 50 50 ]);
Z = imadd(X,Y)
Z =
```

```
255    50    125
 94   255    150
```

Add two images together and specify an output class.

```
I = imread('rice.png');
J = imread('cameraman.tif');
```

# imadd

---

```
K = imadd(I,J,'uint16');  
imshow(K,[])
```

Add a constant to an image.

```
I = imread('rice.png');  
J = imadd(I,50);  
subplot(1,2,1), imshow(I)  
subplot(1,2,2), imshow(J)
```

## See Also

[imabsdiff](#), [imcomplement](#), [imdivide](#), [imlincomb](#), [immultiply](#),  
[imsubtract](#), [ippl](#)

**Purpose**

Adjust image intensity values or colormap

**Syntax**

```
J = imadjust(I)
J = imadjust(I,[low_in; high_in],[low_out; high_out])
J = imadjust(...,gamma)
newmap = imadjust(map,[low_in high_in],[low_out high_out],gamma)
RGB2 = imadjust(RGB1,...)
```

**Description**

`J = imadjust(I)` maps the intensity values in grayscale image `I` to new values in `J` such that 1% of data is saturated at low and high intensities of `I`. This increases the contrast of the output image `J`. This syntax is equivalent to `imadjust(I,stretchlim(I))`.

`J = imadjust(I,[low_in; high_in],[low_out; high_out])` maps the values in `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]`.

`J = imadjust(I,[low_in; high_in],[low_out; high_out],gamma)` maps the values in `I` to new values in `J`, where `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`, `int16`, `single`, 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`.

## Examples

Adjust a low-contrast grayscale image.

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



Adjust the grayscale image, specifying the contrast limits.

```
K = imadjust(I,[0.3 0.7],[,]);  
figure, imshow(K)
```

Adjust an RGB image.

```
RGB1 = imread('football.jpg');  
RGB2 = imadjust(RGB1,[.2 .3 0; .6 .7 1],[,]);  
imshow(RGB1), figure, imshow(RGB2)
```



### See Also

brighten, histeq, stretchlim

# imageinfo

---

**Purpose** Image Information tool

**Syntax**

```
imageinfo
imageinfo(h)
imageinfo(filename)
imageinfo(info)
imageinfo(himage, filename)
imageinfo(himage, info)
hfig = imageinfo(...)
```

**Description** `imageinfo` creates an Image Information tool associated with the image in the current figure. The tool appears in a separate figure information about the basic attributes of the target image. `imageinfo` gets the image attributes by querying the image object's CData.

The following table lists the basic image information included in the Image Information tool display. Note that the tool contains either four or six fields, depending on the type of image.

Attribute Name	Value
Width (columns)	Number of columns in the image
Height (rows)	Number of rows in the image
Class	Data type used by the image, such as <code>uint8</code> .  <b>Note</b> For single or <code>int16</code> images, <code>imageinfo</code> returns a class value of <code>double</code> , because image objects convert the CData of these classes to <code>double</code> .
Image type	One of the image types identified by the Image Processing Toolbox: <code>'intensity'</code> , <code>'truecolor'</code> , <code>'binary'</code> , or <code>'indexed'</code> .

Attribute Name	Value
Minimum intensity	<p>For intensity images, this value represents the lowest intensity value of any pixel.</p> <p>For indexed images, this value represents the lowest index value into a color map.</p> <p>Not included for 'binary' or 'truecolor' images.</p>
Maximum intensity	<p>For intensity images, this value represents the highest intensity value of any pixel.</p> <p>For indexed images, this value represents the highest index value into a color map.</p> <p>Not included for 'binary' or 'truecolor' images.</p>

`imageinfo(h)` creates an Image Information tool associated with `h`, where `h` is a handle to a figure, axes, or image object.

`imageinfo(filename)` creates an Image Information tool containing image metadata from the graphics file `filename`. The image does not have to be displayed in a figure window. `filename` can be any file type that has been registered with an information function in the file formats registry, `imformats`, so its information can be read by `imfinfo`. `filename` can also be a DICOM file with information readable by `dicominfo`.

`imageinfo(info)` creates an Image Information tool containing the image metadata in the structure `info`. `info` is a structure returned by the functions `imfinfo` or `dicominfo`, or `info` can be a user-created structure.

`imageinfo(himage, filename)` creates an Image Information tool containing information about the basic attributes of the image specified by the handle `himage` and the image metadata from the graphics file `filename`.

# imageinfo

---

`imageinfo(himage,info)` creates an Image Information tool containing information about the basic attributes of the image specified by the handle `himage` and the image metadata in the structure `info`.

`hfig=imageinfo(...)` returns a handle to the Image Information tool figure.

## Examples

```
imageinfo('peppers.png')
```

```
h = imshow('bag.png');  
info = imfinfo('bag.png');  
imageinfo(h,info);
```

```
imshow('trees.tif');  
imageinfo;
```

## See Also

`dicominfo`, `imattributes`, `imfinfo`, `imformats`, `imtool`



**Purpose** Image Model object

**Syntax** `imgmodel = imagemodel(himage)`

**Description** `imgmodel = imagemodel(himage)` create an image model object associated with the target image `himage`. `himage` is a handle to an image object or an array of handles to image objects.

`imagemodel` returns an image model object or, if `himage` is an array of image objects, an array of image model objects.

An image model object stores information about an image such as class, type, display range, width, height, minimum intensity value and maximum intensity value.

**API Functions** The image model object supports methods that you can use to access this information, get information about the pixels in an image, and perform special text formatting. The following lists these methods with a brief description. Use `methods(imgmodel)` to get a list of image model methods.

Method	Description
<code>getClassType</code>	Returns a string indicating the class of the image.  <code>str = getClassType(imgmodel)</code> where <code>imgmodel</code> is a valid image model and <code>str</code> is a text string, such as 'uint8'.
<code>getDisplayRange</code>	Returns a double array containing the minimum and maximum values of the display range for an intensity image. For image types other than intensity, the value returned is an empty array.  <code>disp_range = getDisplayRange(imgmodel)</code> where <code>imgmodel</code> is a valid image model and <code>disp_range</code> is an array of doubles, such as [0 255].

# imagemodel

---

Method	Description
<code>getImageHeight</code>	Returns a double scalar containing the number of rows.  <code>height = getImageHeight(imgmodel)</code> where <code>imgmodel</code> is a valid image model and <code>height</code> is a double scalar.
<code>getImageType</code>	Returns a text string indicating the image type.  <code>str = getImageType(imgmodel)</code> where <code>imgmodel</code> is a valid image model and <code>str</code> is one of the text strings 'intensity', 'truecolor', 'binary', or 'indexed'.
<code>getImageWidth</code>	Returns a double scalar containing the number of columns.  <code>width = getImageWidth(imgmodel)</code> where <code>imgmodel</code> is a valid image model and <code>width</code> is a double scalar.
<code>getMinIntensity</code>	Returns the minimum value in the image calculated as <code>min(Image(:))</code> . For an intensity image, the value returned is the minimum intensity. For an indexed image, the value returned is the minimum index. For any other image type, the value returned is an empty array. <code>minval = getMinIntensity(imgmodel)</code>  where <code>imgmodel</code> is a valid image model and <code>minval</code> is a numeric value. The class of <code>minval</code> depends on the class of the target image.

Method	Description
<p><code>getMaxIntensity</code></p>	<p>Returns the maximum value in the image calculated as <code>max(Image(:))</code>. For an intensity image, the value returned is the maximum intensity. For an indexed image, the value returned is the maximum index. For any other image type, the value returned is an empty array.</p> <pre>maxval = getMaxIntensity(imgmodel)</pre> <p>where <code>imgmodel</code> is a valid image model and <code>maxval</code> is a numeric value. The class of <code>maxval</code> depends on the class of the target image.</p>
<p><code>getNumberFormatFcn</code></p>	<p>Returns the handle to a function that converts a numeric value into a string.</p> <pre>fun = getNumberFormatFcn(imgmodel)</pre> <p>where <code>imgmodel</code> is a valid image model. <code>fun</code> is a handle to a function that accepts a numeric value and returns the value as a text string. For example, you can use this function to convert the numeric return value of the <code>getPixelValue</code> method into a text string.</p> <pre>str = fun(getPixelValue(imgmodel,100,100))</pre>
<p><code>getPixelInfoString</code></p>	<p>Returns a text string containing value of the pixel at the location specified by row and column.</p> <pre>str = getPixelInfoString(imgmodel,row,column)</pre> <p>where <code>imgmodel</code> is a valid image model and <code>row</code> and <code>column</code> are numeric scalar values. <code>str</code> is a character array. For example, for an RGB image, the method returns a text string such as <code>'[66 35 60]'</code>.</p>

# imagemodel

---

Method	Description
<code>getPixelRegionFormatFcn</code>	<p>Returns a handle to a function that formats the value of a pixel into a text string.</p> <pre>fun = getPixelRegionFormatFcn(imgmodel)</pre> <p>where <code>imgmodel</code> is a valid image model. <code>fun</code> is a handle to a function that accepts the location (<code>row</code>, <code>column</code>) of a pixel in the target image and returns the value of the pixel as a specially formatted text string. For example, when used with an RGB image, this function returns a text string of the form 'R:000 G:000 B:000' where 000 is the actual pixel value.</p> <pre>str = fun(100,100)</pre>
<code>getPixelValue</code>	<p>Returns the value of the pixel at the location specified by <code>row</code> and <code>column</code> as a numeric array.</p> <pre>val = getPixelValue(imgmodel, row, column)</pre> <p>where <code>imgmodel</code> is a valid image model and <code>row</code> and <code>column</code> are numeric scalar values. The class of <code>val</code> depends on the class of the target image.</p>
<code>getDefaultPixelInfoString</code>	<p>Returns a text string indicating the type of information returned in a pixel information string. This string can be used in place of actual pixel information values.</p> <pre>str = getDefaultPixelInfoString(imgmodel)</pre> <p>where <code>imgmodel</code> is a valid image model. Depending on the image type, <code>str</code> can be the text string 'Intensity', '[R G B]', 'BW', or '&lt;Index&gt; [R G B]'.</p>

Method	Description
<code>getDefaultPixelRegionString</code>	<p>Returns a text string indicating the type of information displayed in the Pixel Region tool for each image type. This string can be used in place of actual pixel values.</p> <pre>str = getDefaultPixelRegionString(imgmodel)</pre> <p>where <code>imgmodel</code> is a valid image model. Depending</p>
<code>getScreenPixelRGBValue</code>	<p>Returns the screen display value of the pixel at the location specified by <code>row</code> and <code>col</code> as a double array.</p> <pre>val = getScreenPixelRGBValue(imgmodel, row, col)</pre> <p>where <code>imgmodel</code> is a valid image model and <code>row</code> and <code>col</code> are numeric scalar values. <code>val</code> is an array of doubles, such as <code>[0.2 0.5 0.3]</code>.</p>

---

**Note** `imagemodel` works by querying the image object's `CData`. For a single or `int16` image, the image object converts its `CData` to `double`. For example, in the case of `h = imshow(int16(ones(10)))`, `class(get(h, 'CData'))` returns `'double'`. Therefore, `getClassType(imgmodel)` returns `'double'`.

---

## Examples

Create an image model.

```
h = imshow('peppers.png');
im = imagemodel(h);

figure, subplot(1,2,1)
h1 = imshow('hestain.png');
subplot(1,2,2)
h2 = imshow('coins.png');
im = imagemodel([h1 h2]);
```

# imagemodel

---

## **See Also**

`getimagemodel`

**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  $(\text{floor}(1/\text{tol})+1)^3$  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. The default value is enclosed in braces ({}).

Value	Description
{'dither'}	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 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, Y is of class double.

**Algorithm** imapprox uses rgb2ind to create a new colormap that uses fewer colors.

# imapprox

---

## Examples

Approximate the indexed image `trees.tif` by another indexed image containing only 16 colors.

```
[X, map] = imread('trees.tif');  
[Y, newmap] = imapprox(X, map, 16);  
imshow(Y, newmap)
```

## See Also

`cmunique`, `dither`, `rgb2ind`



**Purpose** Information about image attributes

**Syntax**

```
attrs = imattributes
attrs = imattributes(himage)
attrs = imattributes(imgmodel)
```

**Description** `attrs = imattributes` returns information about an image in the current figure. If the current figure does not contain an image, `imattributes` returns an empty array.

`attrs = imattributes(himage)` returns information about the image specified by `himage`, a handle to an image object. `imattributes` gets the image attributes by querying the image object's `CData`.

`imattributes` returns image attribute information in `attrs`, a 4-by-2 or 6-by-2 cell array, depending on the image type. The first column of the cell array contains the name of the attribute as a text string. The second column contains the value of the attribute, also represented as a text string. The following table lists these attributes in the order they appear in the cell array.

Attribute Name	Value
Width (columns)	Number of columns in the image
Height (rows)	Number of rows in the image
Class	Data type used by the image, such as <code>uint8</code> .  <b>Note</b> For single or <code>int16</code> images, <code>imageinfo</code> returns a class value of <code>double</code> , because image objects convert <code>CData</code> of these classes to <code>double</code> .

# imattributes

---

Attribute Name	Value
Image type	One of the image types identified by the Image Processing Toolbox: 'intensity', 'truecolor', 'binary', or 'indexed'.
Minimum intensity	For intensity images, this value represents the lowest intensity value of any pixel. For indexed images, this value represents the lowest index value into a color map. Not included for 'binary' or 'truecolor' images.
Maximum intensity	For intensity images, this value represents the highest intensity value of any pixel. For indexed images, this value represents the highest index value into a color map. Not included for 'binary' or 'truecolor' images.

`attrs = imattributes(imgmodel)` returns information about the image represented by the image model object, `imgmodel`.

## Examples

Retrieve the attributes of a grayscale image.

```
h = imshow('liftingbody.png');
attrs = imattributes(h)
attrs =

    'Width (columns)'    '512'
    'Height (rows)'     '512'
    'Class'              'uint8'
    'Image type'         'intensity'
    'Minimum intensity'  '0'
    'Maximum intensity'  '255'
```

Retrieve the attributes of a truecolor image.

```
h = imshow('gantrycrane.png');
im = imagemodel(h);
attrs = imattributes(im)
attrs =

    'Width (columns)'    '400'
    'Height (rows)'     '264'
    'Class'              'uint8'
    'Image type'        'truecolor'
```

**See Also** [imagemodel](#)

# imbothat

---

**Purpose** Bottom-hat filtering

**Syntax** `IM2 = imbothat(IM,SE)`  
`IM2 = imbothat(IM,NHOOD)`

**Description** `IM2 = imbothat(IM,SE)` performs morphological bottom-hat filtering on the grayscale or binary input image, `IM`, returning the filtered image, `IM2`. The argument `SE` is a structuring element returned by the `strel` function. `SE` must be a single structuring element object, not an array containing multiple structuring element objects.

`IM2 = imbothat(IM,NHOOD)` performs morphological bottom-hat filtering where `NHOOD` is an array of 0's and 1's that specifies the size and shape of the structuring element. This is equivalent to `imbothat(IM,strel(NHOOD))`.

**Class Support** `IM` can be numeric or logical and must be nonsparse. The output image has the same class as the input image. If the input is binary (logical), then the structuring element must be flat.

**Examples** Top-hat filtering and bottom-hat filtering can be used together to enhance contrast in an image.

1 Read the image into the MATLAB workspace.

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



- 2 Create disk-shaped structuring element, needed for morphological processing.

```
se = strel('disk',3);
```

- 3 Add the original image  $I$  to the top-hat filtered image, and then subtract the bottom-hat filtered image.

```
J = imsubtract(imadd(I,imtophat(I,se)), imbothat(I,se));  
figure, imshow(J)
```

**See Also**

imtophat, strel

# imclearborder

---

**Purpose** Suppress light structures connected to image border

**Syntax**  
IM2 = imclearborder(IM)  
IM2 = imclearborder(IM,conn)

**Description** IM2 = imclearborder(IM) suppresses structures that are lighter than their surroundings and that are connected to the image border. IM can be a grayscale or binary image. The output image, IM2, is grayscale or binary, respectively. The default connectivity is 8 for two dimensions, 26 for three dimensions, and `conndef(ndims(BW), 'maximal')` for higher dimensions.

---

**Note** For grayscale images, `imclearborder` tends to reduce the overall intensity level in addition to suppressing border structures.

---

IM2 = imclearborder(IM,conn) specifies the desired connectivity. conn can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can also be defined in a more general way for any dimension by using for conn a 3-by-3-by- ... -by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood locations relative to the

center element of conn. Note that conn must be symmetric about its center element.

---

**Note** A pixel on the edge of the input image might not be considered to be a border pixel if a nondefault connectivity is specified. For example, if `conn = [0 0 0; 1 1 1; 0 0 0]`, elements on the first and last row are not considered to be border pixels because, according to that connectivity definition, they are not connected to the region outside the image.

---

## Class Support

IM can be a numeric or logical array of any dimension, and it must be nonsparse and real. IM2 has the same class as IM.

## Examples

The following examples use this simple binary image to illustrate the effect of `imclearborder` when you specify different connectivities.

```
BW =
    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
    1     0     0     1     1     1     0     0     0
    0     1     0     1     1     1     0     0     0
    0     0     0     1     1     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
```

Using a 4-connected neighborhood, the pixel at (5,2) is not considered connected to the border pixel (4,1), so it is not cleared.

```
BWc1 = imclearborder(BW,4)
BWc1 =
    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     1     1     1     0     0     0
```

# imclearborder

---

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

Using an 8-connected neighborhood, pixel (5, 2) is considered connected to pixel (4, 1) so both are cleared.

```
BWc2 = imclearborder(BW,8)
```

```
BWc2 =
```

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

## Algorithm

`imclearborder` uses morphological reconstruction where

- Mask image is the input image.
- Marker image is zero everywhere except along the border, where it equals the mask image.

## See Also

`conndef`

## Reference

[1] Soille, P., *Morphological Image Analysis: Principles and Applications*, Springer, 1999, pp. 164-165.



**Purpose** Morphologically close image

**Syntax**  
`IM2 = imclose(IM,SE)`  
`IM2 = imclose(IM,NHOOD)`

**Description** `IM2 = imclose(IM,SE)` performs morphological closing on the grayscale or binary image `IM`, returning the closed image, `IM2`. The structuring element, `SE`, must be a single structuring element object, as opposed to an array of objects.

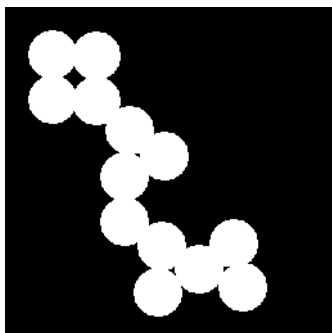
`IM2 = imclose(IM,NHOOD)` performs closing with the structuring element `strel(NHOOD)`, where `NHOOD` is an array of 0's and 1's that specifies the structuring element neighborhood.

**Class Support** `IM` can be any numeric or logical class and any dimension, and must be nonsparse. If `IM` is logical, then `SE` must be flat. `IM2` has the same class as `IM`.

**Examples** Use `imclose` to join the circles in the image together by filling in the gaps between them and by smoothing their outer edges.

- 1 Read the image into the MATLAB workspace and view it.

```
originalBW = imread('circles.png');  
imshow(originalBW);
```



# imclose

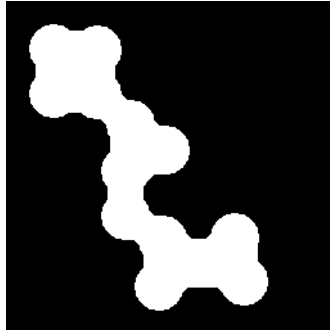
---

- 2 Create a disk-shaped structuring element. Use a disk structuring element to preserve the circular nature of the object. Specify a radius of 10 pixels so that the largest gap gets filled.

```
se = strel('disk',10);
```

- 3 Perform a morphological close operation on the image.

```
closeBW = imclose(originalBW,se);  
figure, imshow(closeBW)
```



## See Also

`imdilate`, `imerode`, `imopen`, `strel`

**Purpose** Complement image

**Syntax** `IM2 = imcomplement(IM)`

**Description** `IM2 = imcomplement(IM)` computes the complement of the image `IM`. `IM` can be a binary, grayscale, or RGB image. `IM2` has the same class and size as `IM`.

In the complement of a binary image, zeros become ones and ones become zeros; black and white are reversed. In the complement of an intensity or RGB image, each pixel value is subtracted from the maximum pixel value supported by the class (or 1.0 for double-precision images) and the difference is used as the pixel value in the output image. In the output image, dark areas become lighter and light areas become darker.

If `IM` is an grayscale or RGB image of class `double`, you can use the expression `1-IM` instead of this function. If `IM` is a binary image, you can use the expression `~IM` instead of this function.

**Examples** Create the complement of a `uint8` array.

```
X = uint8([ 255 10 75; 44 225 100]);
X2 = imcomplement(X)
X2 =
     0    245    180
    211     30    155
```

Reverse black and white in a binary image.

```
bw = imread('text.png');
bw2 = imcomplement(bw);
subplot(1,2,1), imshow(bw)
subplot(1,2,2), imshow(bw2)
```

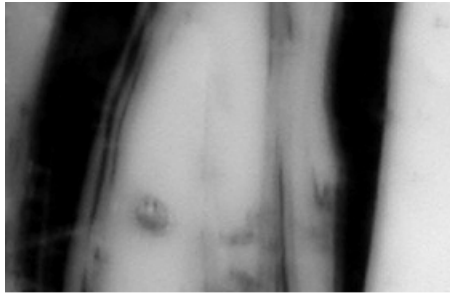
Create the complement of an intensity image.

```
I = imread('glass.png');
J = imcomplement(I);
```

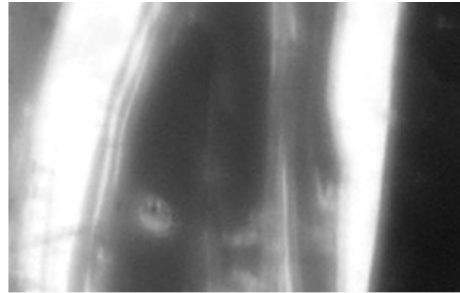
# imcomplement

---

`imshow(I), figure, imshow(J)`



Original Image



Complement Image

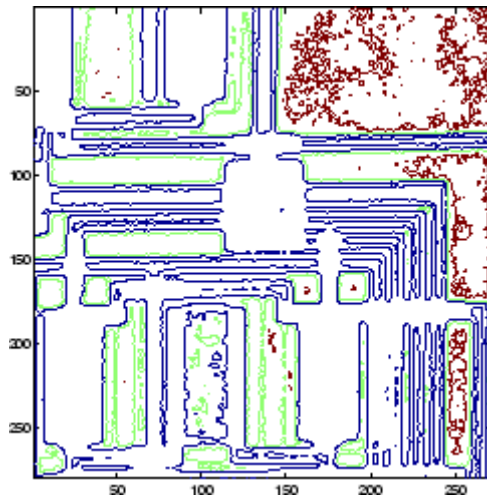
## See Also

`imabsdiff`, `imadd`, `imdivide`, `imlincomb`, `immultiply`, `imsubtract`

<b>Purpose</b>	Create contour plot of image data
<b>Syntax</b>	<pre>imcontour(I) imcontour(I,n) imcontour(I,v) imcontour(x,y,...) imcontour(...,LineStyleSpec) [C,h] = imcontour(...)</pre>
<b>Description</b>	<p><code>imcontour(I)</code> draws a contour plot of the grayscale image <code>I</code>, automatically setting up the axes so their orientation and aspect ratio match the image.</p> <p><code>imcontour(I,n)</code> draws a contour plot of the grayscale image <code>I</code>, automatically setting up the axes so their orientation and aspect ratio match the image. <code>n</code> is the number of equally spaced contour levels in the plot; if you omit the argument, the number of levels and the values of the levels are chosen automatically.</p> <p><code>imcontour(I,v)</code> draws a contour plot of <code>I</code> with contour lines at the data values specified in vector <code>v</code>. The number of contour levels is equal to <code>length(v)</code>.</p> <p><code>imcontour(x,y,...)</code> uses the vectors <code>x</code> and <code>y</code> to specify the <math>x</math>- and <math>y</math>-axis limits.</p> <p><code>imcontour(...,LineStyleSpec)</code> draws the contours using the line type and color specified by <code>LineStyleSpec</code>. Marker symbols are ignored.</p> <p><code>[C,h] = imcontour(...)</code> returns the contour matrix <code>C</code> and a vector of handles to the objects in the plot. (The objects are actually patches, and the lines are the edges of the patches.) You can use the <code>clabel</code> function with the contour matrix <code>C</code> to add contour labels to the plot.</p>
<b>Class Support</b>	The input image can be of class <code>uint8</code> , <code>uint16</code> , <code>int16</code> , <code>single</code> , <code>double</code> , or <code>logical</code> .
<b>Examples</b>	<pre>I = imread('circuit.tif'); imcontour(I,3)</pre>

# imcontour

---



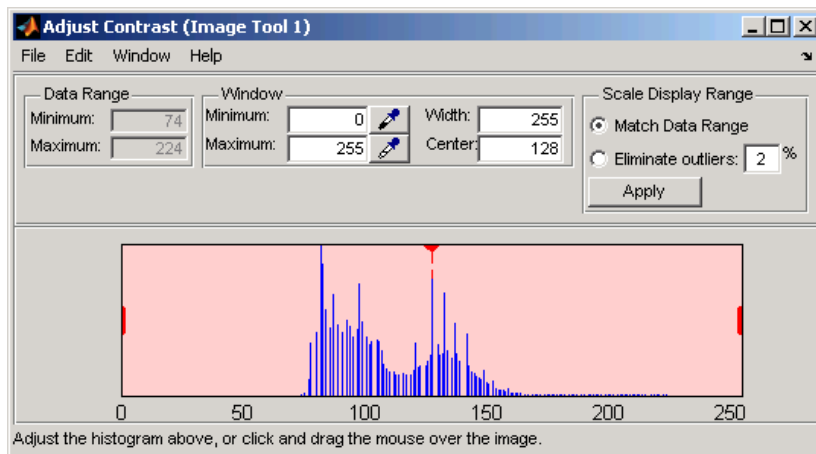
## See Also

`clabel`, `contour`, `LineSpec` in the MATLAB Function Reference

**Purpose** Adjust Contrast tool

**Syntax**  
`imcontrast`  
`imcontrast(h)`  
`hfigure = imcontrast(h)`

**Description** `imcontrast` creates an Adjust Contrast tool in a separate figure that is associated with the grayscale image in the current figure, called the target image. The Adjust Contrast tool is an interactive contrast and brightness adjustment tool, shown in the following figure, that you can use to adjust the black-to-white mapping used to display the image. The tool works by modifying the `CLim` property.



---

**Note** The Adjust Contrast tool can handle grayscale images of class `double` and `single` with data ranges beyond the default display range, which is `[0 1]`. For these images, `imcontrast` sets the histogram limits to fit the image data range, with padding at the upper and lower bounds.

---

# imcontrast

---

`imcontrast(h)` creates the Adjust Contrast tool associated with the image specified by the handle `h`. `h` can be a handle to a figure, axes, uipanel, or image object. If `h` is an axes or figure handle, `imcontrast` uses the first image returned by `findobj(H, 'Type', 'image')`.

`hfigure = imcontrast(...)` returns a handle to the Adjust Contrast tool figure.

## Remarks

The Adjust Contrast tool presents a scaled histogram of pixel values (overly represented pixel values are truncated for clarity). Dragging on the left red bar in the histogram display changes the minimum value. The minimum value (and any value less than the minimum) displays as black. Dragging on the right red bar in the histogram changes the maximum value. The maximum value (and any value greater than the maximum) displays as white. Values in between the red bars display as intermediate shades of gray.

Together the minimum and maximum values create a "window". Stretching the window reduces contrast. Shrinking the window increases contrast. Changing the center of the window changes the brightness of the image. It is possible to manually enter the minimum, maximum, width, and center values for the window. Changing one value automatically updates the other values and the image.

## Window/Level Interactivity

Clicking and dragging the mouse within the target image interactively changes the image's window values. Dragging the mouse horizontally from left to right changes the window width (i.e., contrast). Dragging the mouse vertically up and down changes the window center (i.e., brightness). Holding down the **Ctrl** key before clicking and dragging the mouse accelerates the rate of change; holding down the **Shift** key before clicking and dragging the mouse slows the rate of change.

## Examples

```
imshow('pout.tif')
imcontrast(gca)
```

## See Also

`imadjust`, `imtool`, `stretchlim`



**Purpose**

Crop 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 syntax 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 two- or three-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 two- or three-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 syntax

```
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

If you specify `RECT` as an input argument, the input image can be logical or numeric, and must be real and nonsparse. `RECT` is of class `double`.

If you do not specify `RECT` as an input argument, `imcrop` calls `imshow`. `imshow` expects `I` to be logical, `uint8`, `uint16`, `int16`, `single`, or `double`. `RGB` can be `uint8`, `int16`, `uint16`, `single`, or `double`. `X` can be logical, `uint8`, `uint16`, `single`, or `double`. The input image must be real and nonsparse.

If you specify an image as an input argument, the output image has the same class as the input image.

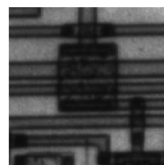
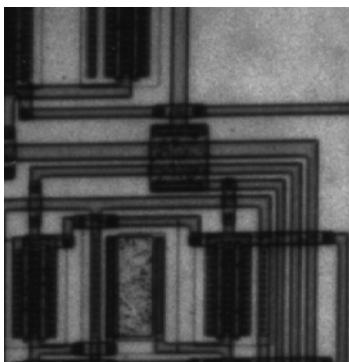
If you don't specify an image as an input argument, i.e., you call `imcrop` with no input arguments, the output image has the same class as the input image except for the `int16` or `single`. The output image is `double` if the input image is `int16` or `single`.

## 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.

## Examples

```
I = imread('circuit.tif');  
I2 = imcrop(I,[75 68 130 112]);  
imshow(I), figure, imshow(I2)
```



## See Also

`zoom`

# imdilate

---

**Purpose** Dilate image

**Syntax**  
`IM2 = imdilate(IM,SE)`  
`IM2 = imdilate(IM,NHOOD)`  
`IM2 = imdilate(IM,SE,PACKOPT)`  
`IM2 = imdilate(...,PADOPT)`

**Description** `IM2 = imdilate(IM,SE)` dilates the grayscale, binary, or packed binary image `IM`, returning the dilated image, `IM2`. The argument `SE` is a structuring element object, or array of structuring element objects, returned by the `strel` function.

If `IM` is logical and the structuring element is flat, `imdilate` performs binary dilation; otherwise, it performs grayscale dilation. If `SE` is an array of structuring element objects, `imdilate` performs multiple dilations of the input image, using each structuring element in `SE` in succession.

`IM2 = imdilate(IM,NHOOD)` dilates the image `IM`, where `NHOOD` is a matrix of 0's and 1's that specifies the structuring element neighborhood. This is equivalent to the syntax `imdilate(IM,strel(NHOOD))`. The `imdilate` function determines the center element of the neighborhood by `floor((size(NHOOD)+1)/2)`.

`IM2 = imdilate(IM,SE,PACKOPT)` or `imdilate(IM,NHOOD,PACKOPT)` specifies whether `IM` is a packed binary image. `PACKOPT` can have either of the following values. Default value is enclosed in braces (`{}`).

Value	Description
'ispacked'	IM is treated as a packed binary image as produced by <code>bwpack</code> . IM must be a 2-D <code>uint32</code> array and SE must be a flat 2-D structuring element. If the value of <code>PACKOPT</code> is 'ispacked', <code>PADOPT</code> must be 'same'.
{'notpacked'}	IM is treated as a normal array.

IM2 = imdilate(..., PADOPT) specifies the size of the output image. PADOPT can have either of the following values. Default value is enclosed in braces ({}).

Value	Description
{'same'}	Make the output image the same size as the input image. If the value of PACKOPT is 'ispacked', PADOPT must be 'same'.
'full'	Compute the full dilation.

PADOPT is analogous to the optional SHAPE argument to the conv2 and filter2 functions.

## Class Support

IM can be logical or numeric and must be real and nonsparse. It can have any dimension. If IM is logical, SE must be flat. The output has the same class as the input. If the input is packed binary, then the output is also packed binary.

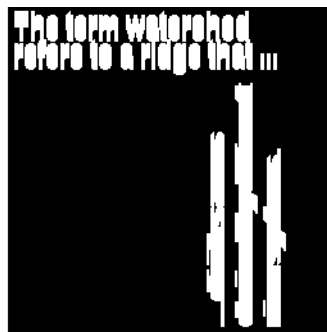
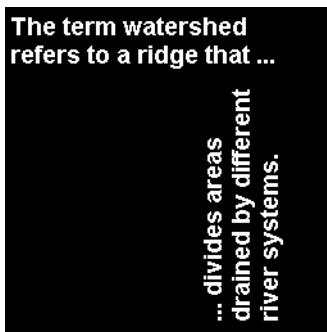
## Examples

Dilate a binary image with a vertical line structuring element.

```

bw = imread('text.png');
se = strel('line',11,90);
bw2 = imdilate(bw,se);
imshow(bw), title('Original')
figure, imshow(bw2), title('Dilated')

```



# imdilate

---

Dilate a grayscale image with a rolling ball structuring element.

```
I = imread('cameraman.tif');
se = strel('ball',5,5);
I2 = imdilate(I,se);
imshow(I), title('Original')
figure, imshow(I2), title('Dilated')
```



To determine the domain of the composition of two flat structuring elements, dilate the scalar value 1 with both structuring elements in sequence, using the 'full' option.

```
se1 = strel('line',3,0)
se1 =
```

Flat STREL object containing 3 neighbors.

Neighborhood:  
1 1 1

```
se2 = strel('line',3,90)
se2 =
```

Flat STREL object containing 3 neighbors.

Neighborhood:  
1  
1  
1

```
composition = imdilate(1,[se1 se2],'full')
composition =
    1     1     1
    1     1     1
    1     1     1
```

## Algorithm

`imdilate` automatically takes advantage of the decomposition of a structuring element object (if it exists). Also, when performing binary dilation with a structuring element object that has a decomposition, `imdilate` automatically uses binary image packing to speed up the dilation.

Dilation using bit packing is described in [2].

## See Also

`bwpack`, `bwunpack`, `conv2`, `filter2`, `imclose`, `imerode`, `imopen`, `strel`

## References

[1] Haralick, R.M., and L. G. Shapiro, *Computer and Robot Vision*, Vol. I, Addison-Wesley, 1992, pp. 158-205.

[2] van den Boomgaard and van Balen, "Image Transforms Using Bitmapped Binary Images," *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, Vol. 54, No. 3, May, 1992, pp. 254-258.

# imshow\_range

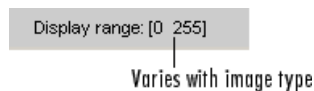
---

**Purpose** Display Range tool

**Syntax**  
`imshow_range`  
`imshow_range(h)`  
`imshow_range(hparent, himage)`  
`hpanel=imshow_range(...)`

**Description** `imshow_range` creates a Display Range tool in the current figure. The Display Range tool shows the display range of the intensity image or images in the figure.

The tool is a uipanel object, positioned in the lower-right corner of the figure. It contains the text string `Display range:` followed by the display range values for the image, as shown in the following figure.



For an indexed, truecolor, or binary image, the display range is not applicable and is set to empty (`[]`).

`imshow_range(h)` creates a Display Range tool in the figure specified by the handle `h`, where `h` is a handle to an image, axes, uipanel, or figure object. Axes, uipanel, or figure objects must contain at least one image object.

`imshow_range(hparent, himage)` creates a Display Range tool in `hparent` that shows the display range of `himage`. `himage` is a handle to an image or an array of image handles. `hparent` is a handle to the figure or uipanel object that contains the display range tool.

`hpanel=imshow_range(...)` returns a handle to the Display Range tool uipanel.

**Note** The Display Range tool can work with multiple images in a figure. When the cursor is not in an image in a figure, the Display Range tool displays the text string `[black white]`.



## Examples

Display an image and include the Display Range tool.

```
imshow('bag.png');  
imshow_range;
```

Import a 16-bit DICOM image and display it with its default range and scaled range in the same figure.

```
dcm = dicomread('CT-MON02-16-ankle.dcm');  
subplot(1,2,1), imshow(dcm);  
subplot(1,2,2), imshow(dcm,[]);  
imshow_range;
```

## See also

`imshow`

# imdistline

---

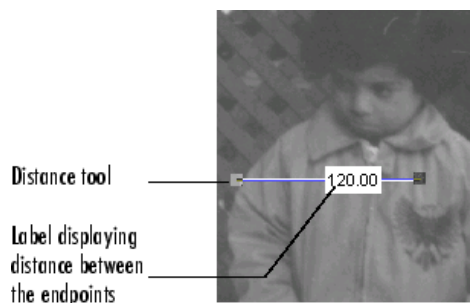
**Purpose** Distance tool

**Syntax**

```
h = imdistline
h = imdistline(himage)
h = imdistline(...,x,y)
```

**Description** `h = imdistline` creates a Distance tool on the current axes. The function returns `h`, a handle to the Distance tool, which is an `hggroup` object.

The Distance tool is a draggable, resizable line, superimposed on an axes, that measures the distance between the two endpoints of the line. Using the mouse, you can move and resize the Distance tool to measure the distance between any two points in an image. The Distance tool displays the distance in a text label superimposed over the line. The tools specifies the distance in data units determined by the `XData` and `YData` properties, which is pixels, by default. The following figure shows a Distance tool on an axes.



You can move the Distance tool over an image by dragging it with the mouse. You can also resize the Distance tool by selecting one of the endpoints with the mouse and dragging the endpoint.

`h = imdistline(hparent)` creates a Distance tool on the object specified by `hparent`. `hparent` specifies the Distance tool's parent, which is typically an axes object, but can also be any other object that can be the parent of an `hggroup` object.

`h = imdistline(...,x,y)` creates a Distance tool with endpoints located at the locations specified by the vectors `x` and `y`, where `x = [x1 x2]` and `y = [y1 y2]`.

**Context Menu**

The Distance tool has a context menu associated with it that allows you to

- Export endpoint and distance data to the workspace
- Toggle the distance label on/off
- Set the line color
- Specify horizontal and vertical drag constraints
- Delete the Distance tool object

Right-click to access the Distance tool context menu.

**API Functions**

The Distance tool contains a structure of function handles, called an API, that can be used to retrieve distance information and control other aspects of Distance tool behavior. To retrieve this structure from the Distance tool, use the `iptgetapi` function, where `h` is a handle to the Distance tool.

```
api = iptgetapi(h)
```

The following table lists the functions in the API, with their syntax and brief descriptions, in the order they appear in the structure.

Method	Description
<code>setPosition</code>	Sets the endpoint positions of the Distance tool.  <code>setPosition(X,Y)</code> <code>setPosition([X1 Y1; X2 Y2])</code>

# imdistline

---

Method	Description
getPosition	Returns the endpoint positions of the Distance tool, <pre>pos = api.getPosition()</pre> where <code>pos</code> is a 2-by-2 array [X1 Y1; X2 Y2].
delete	Deletes the Distance tool associated with the API. <pre>delete()</pre>
setColor	Sets the color used to draw the Distance tool, <pre>setColor(new_color)</pre> where <code>new_color</code> can be a three-element vector specifying an RGB triplet, or a text string specifying the long or short names of a predefined color, such as 'white' or 'w'. For a complete list of these predefined colors and their short names, see <code>ColorSpec</code> .
addNewPositionCallback	Adds the function handle <code>fcn</code> to the list of new-position callback functions. <pre>id = addNewPositionCallback(fcn)</pre> Whenever the Distance tool changes its position, each function in the list is called with the syntax <pre>fcn(pos)</pre> where <code>pos</code> is a 2-by-2 array [X1 Y1; X2 Y2]. The return value, <code>id</code> , is used only with <code>removeNewPositionCallback</code> .

Method	Description
removeNewPositionCallback	<p>Removes the corresponding function from the new-position callback list,</p> <pre>removeNewPositionCallback(id)</pre> <p>where <code>id</code> is the identifier returned by <code>addNewPositionCallback</code>.</p>
getDragConstraintFcn	<p>Returns the function handle of the current drag constraint function.</p> <pre>fcn = getDragConstraintFcn()</pre>
setDragConstraintFcn	<p>Sets the drag constraint function to be the specified function handle, <code>fcn</code>.</p> <pre>setDragConstraintFcn(fcn)</pre> <p>Whenever the Distance tool is moved or resized because of a mouse drag, the constraint function is called using the syntax</p> <pre>constrained_position = fcn(new_position)</pre> <p>where <code>new_position</code> is a 2-by-2 array [<code>X1 Y1; X2 Y2</code>].</p> <p>You can use the drag constraint function to control where the Distance tool can be moved and resized.</p>
getDistance	<p>Returns the distance between the endpoints of the Distance tool.</p> <pre>dist = getDistance()</pre>

# imdistline

Method	Description
<code>getAngleFromHorizontal</code>	Returns the angle in degrees between the line defined by the Distance tool and the horizontal axis. The angle returned is between 0 and 180 degrees. (For information about how this angle is calculated, see “Remarks” on page 17-294.)  <code>angle = getAngleFromHorizontal()</code>
<code>getLabelHandle</code>	Returns a handle to the Distance tool text label.  <code>hlabel = getLabelHandle()</code>
<code>getLabelTextFormatter</code>	Returns a character array specifying the format string used to display the distance label,  <code>str = getLabelTextFormatter()</code>  where <code>str</code> is a character array specifying a format string in the form expected by <code>sprintf</code> .
<code>setLabelTextFormatter</code>	Sets the format string used in displaying the distance label,  <code>setLabelTextFormatter(str)</code>  where <code>str</code> is a a character array specifying a format string in the form expected by <code>sprintf</code> .

## Remarks

If you use `imdistline` with an axis that contains an image object, and do not specify a drag constraint function, users can drag the point outside the extent of the image and lose the point. When used with an axis created by the `plot` function, the axis limits automatically expand to accommodate the movement of the point.

To understand how `imdistline` calculates the angle returned by `getAngleToHorizontal`, draw an imaginary horizontal vector from the bottom endpoint of the distance line, extending to the right. The value

returned by `getAngleToHorizontal` is the angle from this horizontal vector to the distance line, which can range from 0 to 180 degrees.

## Examples

### Example 1

Insert a Distance tool into an image. Use `makeConstrainToRectFcn` to specify a drag constraint function that prevents the Distance tool from being dragged outside the extent of the image.

```
figure, imshow('pout.tif');
h = imdistline(gca);
api = iptgetapi(h);
fcn = makeConstrainToRectFcn('imline',...
                             get(gca,'XLim'),get(gca,'YLim'));
api.setDragConstraintFcn(fcn);
```

Now, right-click the Distance tool and explore the context menu options.

Set the text formatter used to display the distance label.

```
close all, imshow('pout.tif');
h = imdistline;
api = iptgetapi(h);
api.setLabelTextFormatter('%02.1f pixels');
```

### Example 2

Position endpoints of the Distance tool at the specified locations.

```
close all, imshow('pout.tif');
h = imdistline(gca,[10 100],[10 100]);
```

Delete the Distance tool.

```
api = iptgetapi(h);
api.delete();
```

## Example 3

Use distance tool with XData and YData of associated image in non-pixel units.

```
hImg = imshow('boston.tif');

% Convert XData and YData to meters using conversion factor.
metersPerPixel = 620/139;
XDataInMeters = get(hImg,'XData')*metersPerPixel;
YDataInMeters = get(hImg,'YData')*metersPerPixel;

% Set XData and YData of image to reflect desired units.
set(hImg,'XData',XDataInMeters,'YData',YDataInMeters);
set(gca,'XLim',XDataInMeters,'YLim',YDataInMeters);

% Specify position of distance tool in terms of XData/YData units.
hline = imdistline(gca,[682 900],[1775 2356]);
api = iptgetapi(hline);
api.setLabelTextFormatter('%02.0f meters');
```

## See Also

[iptgetapi](#)



**Purpose** Divide one image into another or divide image by constant

**Syntax** `Z = imdivide(X,Y)`

**Description** `Z = imdivide(X,Y)` divides each element in the array `X` by the corresponding element in array `Y` and returns the result in the corresponding element of the output array `Z`. `X` and `Y` are real, nonsparse numeric arrays with the same size and class, or `Y` can be a scalar double. `Z` has the same size and class as `X` and `Y`.

If `X` is an integer array, elements in the output that exceed the range of integer type are truncated, and fractional values are rounded.

---

**Note** On Intel architecture processors, `imdivide` can take advantage of the Intel Performance Primitives Library (IPPL), thus accelerating its execution time. IPPL is activated only if arrays `X` and `Y` are of class `uint8`, `int16`, or `single` and are of the same size and class.

---

**Examples** Divide two `uint8` arrays. Note that fractional values greater than or equal to 0.5 are rounded up to the nearest integer.

```
X = uint8([ 255 10 75; 44 225 100]);
Y = uint8([ 50 20 50; 50 50 50 ]);
Z = imdivide(X,Y)
Z =
     5     1     2
     1     5     2
```

Estimate and divide out the background of the rice image.

```
I = imread('rice.png');
background = imopen(I,strel('disk',15));
Ip = imdivide(I,background);
imshow(Ip,[])
```

Divide an image by a constant factor.

# imdivide

---

```
I = imread('rice.png');  
J = imdivide(I,2);  
subplot(1,2,1), imshow(I)  
subplot(1,2,2), imshow(J)
```

## See Also

[imabsdiff](#), [imadd](#), [imcomplement](#), [imlincomb](#), [immultiply](#),  
[imsubtract](#), [ippl](#)

**Purpose** Erode image

**Syntax**

```
IM2 = imerode(IM,SE)
IM2 = imerode(IM,NHOOD)
IM2 = imerode(IM,SE,PACKOPT,M)
IM2 = imerode(...,PADOPT)
```

**Description** `IM2 = imerode(IM,SE)` erodes the grayscale, binary, or packed binary image `IM`, returning the eroded image `IM2`. The argument `SE` is a structuring element object or array of structuring element objects returned by the `strel` function.

If `IM` is logical and the structuring element is flat, `imerode` performs binary dilation; otherwise it performs grayscale erosion. If `SE` is an array of structuring element objects, `imerode` performs multiple erosions of the input image, using each structuring element in `SE` in succession.

`IM2 = imerode(IM,NHOOD)` erodes the image `IM`, where `NHOOD` is an array of 0's and 1's that specifies the structuring element neighborhood. This is equivalent to the syntax `imerode(IM,strel(NHOOD))`. The `imerode` function determines the center element of the neighborhood by `floor((size(NHOOD)+1)/2)`

`IM2 = imerode(IM,SE,PACKOPT,M)` or `imerode(IM,NHOOD,PACKOPT,M)` specifies whether `IM` is a packed binary image and, if it is, provides the row dimension `M` of the original unpacked image. `PACKOPT` can have either of the following values. Default value is enclosed in braces (`{}`).

Value	Description
'ispacked'	IM is treated as a packed binary image as produced by <code>bwpack</code> . IM must be a 2-D <code>uint32</code> array and SE must be a flat 2-D structuring element.
{ 'notpacked' }	IM is treated as a normal array.

If `PACKOPT` is 'ispacked', you must specify a value for `M`.

`IM2 = imerode(..., PADOPT)` specifies the size of the output image. `PADOPT` can have either of the following values. Default value is enclosed in braces (`{}`).

Value	Description
<code>{ 'same' }</code>	Make the output image the same size as the input image. If the value of <code>PACKOPT</code> is <code>'ispacked'</code> , <code>PADOPT</code> must be <code>'same'</code> .
<code>'full'</code>	Compute the full erosion.

`PADOPT` is analogous to the `SHAPE` input to the `CONV2` and `FILTER2` functions.

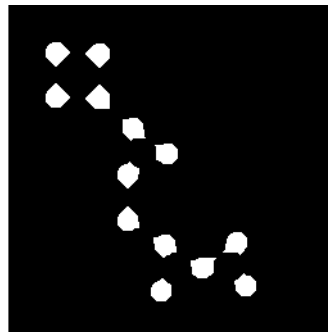
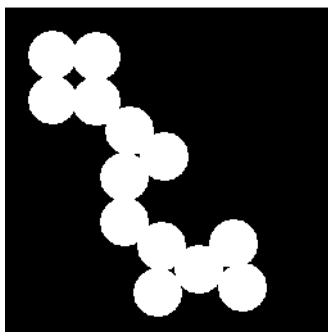
## Class Support

`IM` can be numeric or logical and it can be of any dimension. If `IM` is logical and the structuring element is flat, the output image is logical; otherwise the output image has the same class as the input. If the input is packed binary, then the output is also packed binary.

## Examples

Erode a binary image with a disk structuring element.

```
originalBW = imread('circles.png');  
se = strel('disk',11);  
erodedBW = imerode(originalBW,se);  
imshow(originalBW), figure, imshow(erodedBW)
```



Erode a grayscale image with a rolling ball.

```
I = imread('cameraman.tif');
se = strel('ball',5,5);
I2 = imerode(I,se);
imshow(I), title('Original')
figure, imshow(I2), title('Eroded')
```



## Algorithm Notes

`imerode` automatically takes advantage of the decomposition of a structuring element object (if a decomposition exists). Also, when performing binary dilation with a structuring element object that has a decomposition, `imerode` automatically uses binary image packing to speed up the dilation.

Erosion using bit packing is described in [2].

## See Also

`bwpack`, `bwunpack`, `conv2`, `filter2`, `imclose`, `imdilate`, `imopen`, `strel`

## References

[1] Haralick, Robert M., and Linda G. Shapiro, *Computer and Robot Vision*, Vol. I, Addison-Wesley, 1992, pp. 158-205.

[2] van den Boomgaard and van Balen, "Image Transforms Using Bitmapped Binary Images," *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, Vol. 54, No. 3, May, 1992, pp. 254-258.

# imextendedmax

---

**Purpose** Extended-maxima transform

**Syntax**  
BW = imextendedmax(I,H)  
BW = imextendedmax(I,H,conn)

**Description** BW = imextendedmax(I,H) computes the extended-maxima transform, which is the regional maxima of the H-maxima transform. H is a nonnegative scalar.

Regional maxima are connected components of pixels with a constant intensity value, and whose external boundary pixels all have a lower value.

By default, imextendedmax uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, imextendedmax uses conndef(ndims(I), 'maximal').

BW = imextendedmax(I,H,conn) computes the extended-maxima transform, where conn specifies the connectivity. conn can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

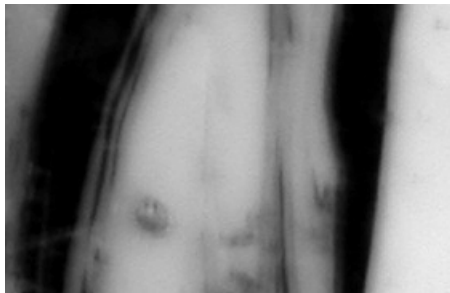
Connectivity can be defined in a more general way for any dimension by using for conn a 3-by-3-by- ...-by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood locations relative to the center element of conn. Note that conn must be symmetric about its center element.

## Class Support

I can be of any nonsparse numeric class and any dimension. BW has the same size as I and is always logical.

## Examples

```
I = imread('glass.png');  
BW = imextendedmax(I,80);  
imshow(I), figure, imshow(BW)
```



Original Image



Extended Maxima Image

## See Also

conndef, imextendedmin, imhmax, imreconstruct, imregionalmax

## Reference

[1] Soille, P., *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 170-171.

# imextendedmin

---

**Purpose** Extended-minima transform

**Syntax**  
BW = imextendedmin(I,h)  
BW = imextendedmin(I,h,conn)

**Description** BW = imextendedmin(I,h) computes the extended-minima transform, which is the regional minima of the H-minima transform. h is a nonnegative scalar.

Regional minima are connected components of pixels with a constant intensity value, and whose external boundary pixels all have a higher value.

By default, imextendedmin uses 8-connected neighborhoods for 2-D images, and 26-connected neighborhoods for 3-D images. For higher dimensions, imextendedmin uses conndef(ndims(I), 'maximal').

BW = imextendedmin(I,h,conn) computes the extended-minima transform, where conn specifies the connectivity. conn can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can be defined in a more general way for any dimension by using for conn a 3-by-3-by-...-by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood locations relative to the center element of conn. Note that conn must be symmetric about its center element.

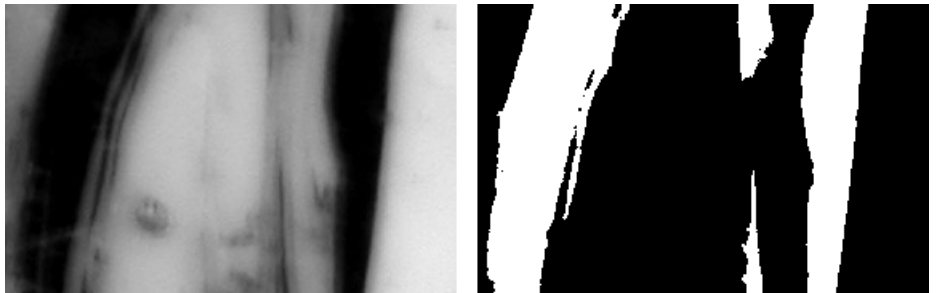


## Class Support

I can be of any nonsparse numeric class and any dimension. BW has the same size as I and is always logical.

## Examples

```
I = imread('glass.png');  
BW = imextendedmin(I,50);  
imshow(I), figure, imshow(BW)
```



Original Image

Extended Minima Image

## See Also

conndef, imextendedmax, imhmin, imreconstruct, imregionalmin

## Reference

[1] Soille, P., *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 170-171.

# imfill

---

**Purpose** Fill image regions and holes

**Syntax**

```
BW2 = imfill(BW,locations)
BW2 = imfill(BW,'holes')
I2 = imfill(I)
```

```
BW2 = imfill(BW)
[BW2 locations] = imfill(BW)
```

```
BW2 = imfill(BW,locations,conn)
BW2 = imfill(BW,conn,'holes')
I2 = imfill(I,conn)
```

**Description** `BW2 = imfill(BW,locations)` performs a flood-fill operation on background pixels of the binary image `BW`, starting from the points specified in `locations`. If `locations` is a P-by-1 vector, it contains the linear indices of the starting locations. If `locations` is a P-by-ndims (`BW`) matrix, each row contains the array indices of one of the starting locations.

`BW2 = imfill(BW,'holes')` fills holes in the binary image `BW`. A hole is a set of background pixels that cannot be reached by filling in the background from the edge of the image.

`I2 = imfill(I)` fills holes in the grayscale image `I`. In this case, a hole is an area of dark pixels surrounded by lighter pixels.

**Interactive Use** `BW2 = imfill(BW)` displays the binary image `BW` on the screen and lets you select the starting locations using the mouse. Click the mouse button 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 operation; pressing **Return** finishes the selection without adding a point.

---

**Note** `imfill` supports interactive use only for 2-D images.

---

`[BW2,locations] = imfill(BW)` lets you select the starting points selected using the mouse, returning the locations of points in `locations`. `locations` is a vector of linear indices into the input image.

## Specifying Connectivity

By default, `imfill` uses 4-connected background neighbors for 2-D inputs and 6-connected background neighbors for 3-D inputs. For higher dimensions the default background connectivity is determined by using `conndef(NUM_DIMS, 'minimal')`. You can override the default connectivity with these syntax:

```
BW2 = imfill(BW,locations,conn)
BW2 = imfill(BW,conn,'holes')
I2 = imfill(I,conn)
```

To override the default connectivity and interactively specify the starting locations, use this syntax:

```
BW2 = imfill(BW,0,conn)
```

`conn` can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can be defined in a more general way for any dimension by using for `conn` a 3-by-3-by- ... -by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood locations relative to the center element of `conn`. Note that `conn` must be symmetric about its center element.

# imfill

---

## Class Support

The input image can be numeric or logical, and it must be real and nonsparse. It can have any dimension. The output image has the same class as the input image.

## Examples

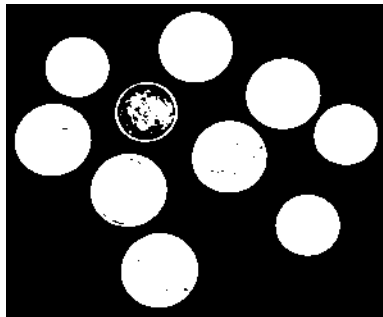
Fill in the background of a binary image from a specified starting location.

```
BW1 = logical([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]);
```

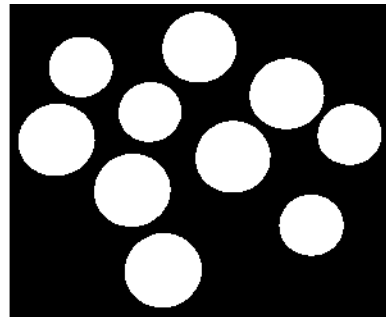
```
BW2 = imfill(BW1,[3 3],8)
```

Fill in the holes of a binary image.

```
BW4 = im2bw(imread('coins.png'));
BW5 = imfill(BW4,'holes');
imshow(BW4), figure, imshow(BW5)
```



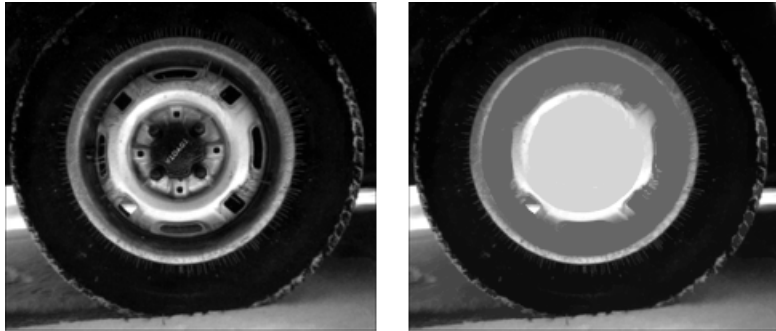
Original Image



Filled Image

Fill in the holes of a grayscale image.

```
I = imread('tire.tif');  
I2 = imfill(I,'holes');  
figure, imshow(I), figure, imshow(I2)
```



Original Image

Filled Image

**Algorithm**

imfill uses an algorithm based on morphological reconstruction [1].

**See Also**

bwselect, imreconstruct, roifill

**Reference**

[1] Soille, P., *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 173-174.

# imfilter

---

**Purpose** N-D filtering of multidimensional images

**Syntax**  
`B = imfilter(A,H)`  
`B = imfilter(A,H,option1,option2,...)`

**Description** `B = imfilter(A,H)` filters the multidimensional array `A` with the multidimensional filter `H`. The array `A` can be a nonsparse numeric array of any class and dimension. The result `B` has the same size and class as `A`.

Each element of the output `B` is computed using double-precision floating point. If `A` is an integer array, then output elements that exceed the range of the integer type are truncated, and fractional values are rounded.

`B = imfilter(A,H,option1,option2,...)` performs multidimensional filtering according to the specified options. Option arguments can have the following values.

## Boundary Options

Option	Description
<code>X</code>	Input array values outside the bounds of the array are implicitly assumed to have the value <code>X</code> . When no boundary option is specified, <code>imfilter</code> uses <code>X = 0</code> .
<code>'symmetric'</code>	Input array values outside the bounds of the array are computed by mirror-reflecting the array across the array border.
<code>'replicate'</code>	Input array values outside the bounds of the array are assumed to equal the nearest array border value.
<code>'circular'</code>	Input array values outside the bounds of the array are computed by implicitly assuming the input array is periodic.

### Output Size Options

Option	Description
'same'	The output array is the same size as the input array. This is the default behavior when no output size options are specified.
'full'	The output array is the full filtered result, and so is larger than the input array.

### Correlation and Convolution Options

Option	Description
'corr'	imfilter performs multidimensional filtering using correlation, which is the same way that filter2 performs filtering. When no correlation or convolution option is specified, imfilter uses correlation.
'conv'	imfilter performs multidimensional filtering using convolution.

N-D convolution is related to N-D correlation by a reflection of the filter matrix.

---

**Note** On Intel architecture processors, imfilter can take advantage of the Intel Performance Primitives Library (IPPL), thus accelerating its execution time. IPPL is activated only if A and H are both two-dimensional and A is of class uint8, int16, or single.

---

### Examples

Read a color image into the workspace and view it.

```
originalRGB = imread('peppers.png');
imshow(originalRGB)
```

Create a filter, h, that can be used to approximate linear camera motion.

# imfilter

---

```
h = fspecial('motion', 50, 45);
```

Apply the filter, using `imfilter`, to the image `rgb` to create a new image, `rgb2`.

```
filteredRGB = imfilter(originalRGB, h);  
figure, imshow(filteredRGB)
```

Note that `imfilter` is more memory efficient than some other filtering operations in that it outputs an array of the same data type as the input image array. In this example, the output is an array of `uint8`.

```
whos rgb2
```

Name	Size	Bytes	Class
h	37x37	10952	double array
rgb	384x512x3	589824	uint8 array
rgb2	384x512x3	589824	uint8 array

Specify the `replicate` boundary option.

```
boundaryReplicateRGB = imfilter(originalRGB, h, 'replicate');  
figure, imshow(boundaryReplicateRGB)
```

## See Also

`conv2`, `convn`, `filter2`, `fspecial`, `ippl`



**Purpose** Information about graphics file

**Note** `imfinfo` is a MATLAB function.

# imgca

---

**Purpose** Get handle to current axis containing image

**Syntax**  
`hax = imgca`  
`hax = imgca(hfig)`

**Description** `hax = imgca` returns the handle of the of the current axis that contains an image. The current axis may be in a regular figure window or in an Image Tool window.

If no figure contains an axis that contains an image, `imgca` creates a new axis.

`hax = imgca(hfig)` returns the handle to the current axis that contains an image in the specified figure (it need not be the current figure).

**Note** `imgca` can be useful in getting the handle to the Image Tool axis. Because the Image Tool turns graphics object handle visibility off, you cannot retrieve a handle to the tool figure using `gca`.

**Examples** Label the coins in the image, compute their centroids, and superimpose the centroid locations on the image. View the results using `imtool` and `imgca`.

```
I = imread('coins.png');  
figure, imshow(I)  
  
bw = im2bw(I, graythresh(getimage));  
figure, imshow(bw)  
  
bw2 = imfill(bw, 'holes');  
L = bwlabel(bw2);  
s = regionprops(L, 'centroid');  
centroids = cat(1, s.Centroid);
```

Display original image I and superimpose centroids.

```
imtool(I)  
hold(imgca, 'on')
```

```
plot(imgca,centroids(:,1), centroids(:,2), 'r*')  
hold(imgca,'off')
```

**See also**

gca,(gcf, imgcf)

# imgcf

---

**Purpose** Get handle to current figure containing image

**Syntax** `hfig = imgcf`

**Description** `hfig = imgcf` returns the handle of the most recent current figure that contains an image. The figure may be a regular figure window that contains at least one image or an Image Tool window.

If none of the figures currently open contains an image, `imgcf` creates a new figure.

**Note** `imgcf` can be useful in getting the handle to the Image Tool figure window. Because the Image Tool turns graphics object handle visibility off, you cannot retrieve a handle to the tool figure using `gcf`.

**Examples**

```
imtool rice.png
cmap = copper(256);
set(imgcf, 'Colormap', cmap)
```

**See also** `gca`, `gcf`, `imgca`

**Purpose** Open Image dialog box

**Syntax** `[filename, user_canceled] = imgetfile`

**Description** `[filename, user_canceled] = imgetfile` displays the Open Image dialog box for the user to fill in, and returns the full path to the file selected in filename. If the user presses the **Cancel** button, `user_canceled` will be TRUE. Otherwise, `user_canceled` will be FALSE.

The Open Image dialog box is modal; it blocks the MATLAB command line until the user responds. The file types listed in the dialog are all formats listed in `imformats` plus DICOM.

**See Also** `imformats`, `imtool`, `uigetfile`

# imhandles

---

**Purpose** Get all image handles

**Syntax** `himage = imhandles(h)`

**Description** `himage = imhandles(h)` takes a graphics handle `h` as an input and returns all of the image handles whose ancestor is `h`. `h` can be an array of valid figure, axes, image, or uipanel handles.

`himage` is an array of image handles.

`imhandles` ignores colorbars in `h` and does not include its handle in `himage`.

**Note** `imhandles` errors if the image objects in `himage` do not have the same figure as their parent.

**Examples** Return the handle to the image object in the current axes.

```
figure, imshow('moon.tif');  
himage = imhandles(gca)
```

Display two images in a figure and uses `imhandles` to get handles to both of the image objects in the figure.

```
subplot(1,2,1), imshow('autumn.tif');  
subplot(1,2,2), imshow('glass.png');  
himages = imhandles(gcf)
```

**See Also** `imgca`, `imgcf`

**Purpose** Display histogram of image data

**Syntax**

```
imhist(I,n)
imhist(X,map)
[counts,x] = imhist(...)
```

**Description** `imhist(I)` displays a histogram for the image `I` above a grayscale colorbar. The number of bins in the histogram is specified by the image type. If `I` is a grayscale image, `imhist` uses a default value of 256 bins. If `I` is a binary image, `imhist` uses two bins.

`imhist(I,n)` displays a histogram where `n` specifies the number of bins used in the histogram. `n` also specifies the length of the colorbar. If `I` is a binary image, `n` can only have the value 2.

`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, for intensity images that are not `int16`, the  $p$ th bin is the half-open interval  $A(p-1.5)/(n-1) \leq x < A(p-0.5)/(n-1)$ , where  $x$  is the intensity value. For `int16` intensity images, the  $p$ th bin is the half-open interval  $A(p-1.5)/(n-1) - 32768 \leq x < A(p-0.5)/(n-1) - 32768$ , where  $x$  is the intensity value. The scale factor  $A$  depends on the image class.  $A$  is 1 if the intensity image is `double` or `single`,  $A$  is 255 if the intensity image is `uint8`, and  $A$  is 65535 if the intensity image is `uint16` or `int16`.

---

# imhist

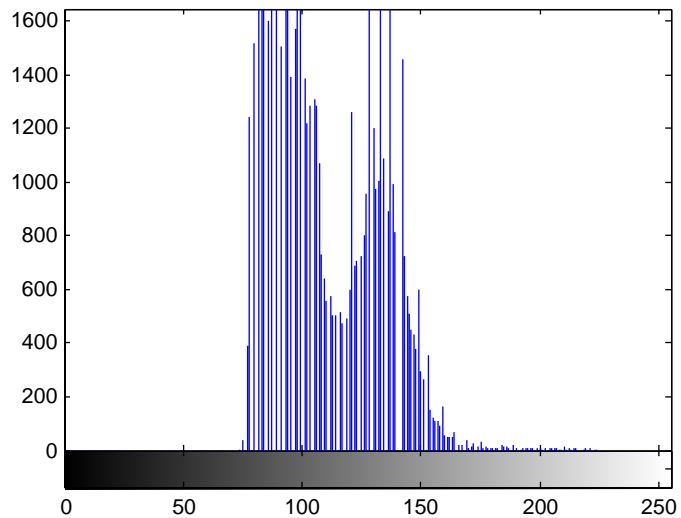
---

## Class Support

An input intensity image can be of class `uint8`, `uint16`, `int16`, `single`, `double`, or `logical`. An input indexed image can be of class `uint8`, `uint16`, `single`, `double`, or `logical`.

## Examples

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



## See Also

`histeq`

`hist` in the MATLAB Function Reference



**Purpose** H-maxima transform

**Syntax**  
`I2 = imhmax(I,h)`  
`I2 = imhmax(I,h,conn)`

**Description** `I2 = imhmax(I,h)` suppresses all maxima in the intensity image `I` whose height is less than `h`, where `h` is a scalar.

Regional maxima are connected components of pixels with a constant intensity value, and whose external boundary pixels all have a lower value.

By default, `imhmax` uses 8-connected neighborhoods for 2-D images, and 26-connected neighborhoods for 3-D images. For higher dimensions, `imhmax` uses `conndef(ndims(I), 'maximal')`.

`I2 = imhmax(I,h,conn)` computes the H-maxima transform, where `conn` specifies the connectivity. `conn` can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can be defined in a more general way for any dimension by using for `conn` a 3-by-3-by- ...-by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood locations relative to the center element of `conn`. Note that `conn` must be symmetric about its center element.

# imhmax

---

## Class Support

I can be of any nonsparse numeric class and any dimension. I2 has the same size and class as I.

## Examples

```
a = zeros(10,10);  
a(2:4,2:4) = 3; % maxima 3 higher than surround  
a(6:8,6:8) = 8; % maxima 8 higher than surround  
b = imhmax(a,4); % only the maxima higher than 4 survive.
```

## See Also

conndef, imextendedmax, imhmin, imreconstruct, imregionalmax

## Reference

[1] Soille, P., *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 170-171.

**Purpose** H-minima transform

**Syntax**  
`I2 = imhmin(I,h)`  
`I2 = imhmin(I,h,conn)`

**Description** `I2 = imhmin(I,h)` suppresses all minima in `I` whose depth is less than `h`. `I` is a grayscale image and `h` is a scalar.

Regional minima are connected components of pixels with a constant intensity value, and whose external boundary pixels all have a higher value.

By default, `imhmin` uses 8-connected neighborhoods for 2-D images, and 26-connected neighborhoods for 3-D images. For higher dimensions, `imhmin` uses `conndef(ndims(I), 'maximal')`.

`I2 = imhmin(I,h,conn)` computes the H-minima transform, where `conn` specifies the connectivity. `conn` can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can be defined in a more general way for any dimension by using for `conn` a 3-by-3-by-...-by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood locations relative to the center element of `conn`. Note that `conn` must be symmetric about its center element.

# imhmin

---

## **Class Support**

I can be of any nonsparse numeric class and any dimension. I2 has the same size and class as I.

## **Examples**

Create a sample image with two regional minima.

```
a = 10*ones(10,10);
```

```
a(2:4,2:4) = 7;
```

```
a(6:8,6:8) = 2
```

```
a =
```

```
10  10  10  10  10  10  10  10  10  10
10   7   7   7  10  10  10  10  10  10
10   7   7   7  10  10  10  10  10  10
10   7   7   7  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10   2   2   2  10  10
10  10  10  10  10   2   2   2  10  10
10  10  10  10  10   2   2   2  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
```

Suppress all minima below a specified value. Note how the region with pixel valued 7 disappears in the transformed image.

```
b = imhmin(a,4)
```

```
b =
```

```
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  6   6   6  10  10
10  10  10  10  10  6   6   6  10  10
10  10  10  10  10  6   6   6  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
```

**See Also**

conndef, imextendedmin, imhmax, imreconstruct, imregionalmin

**Reference**

[1] Soille, P., *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 170-171.

# imimposemin

---

**Purpose**            Impose minima

**Syntax**            `I2 = imimposemin(I,BW)`  
                      `I2 = imimposemin(I,BW,conn)`

**Description**        `I2 = imimposemin(I,BW)` modifies the intensity image `I` using morphological reconstruction so it only has regional minima wherever `BW` is nonzero. `BW` is a binary image the same size as `I`.

By default, `imimposemin` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imimposemin` uses `conndef(ndims(I),'minimum')`.

`I2 = imimposemin(I,BW,conn)` specifies the connectivity, where `conn` can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can also be defined in a more general way for any dimension by using for `conn` a 3-by-3-by-...-by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood locations relative to the center element of `conn`. Note that `conn` must be symmetric about its center element.

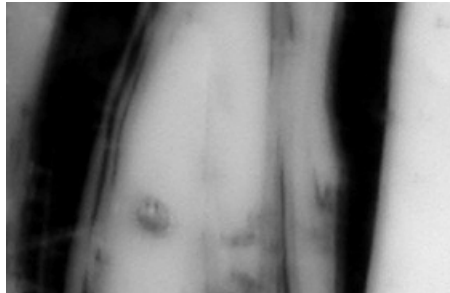
**Class Support**        `I` can be of any nonsparse numeric class and any dimension. `BW` must be a nonsparse numeric array with the same size as `I`. `I2` has the same size and class as `I`.

## Examples

Modify an image so that it only has regional minima at one location.

- 1 Read an image and display it. This image is called the *mask* image.

```
mask = imread('glass.png');  
imshow(mask)
```



- 2 Create the marker image that will be used to process the mask image.

The example creates a binary image that is the same size as the mask image and sets a small area of the binary image to 1. These pixels define the location in the mask image where a regional minimum will be imposed.

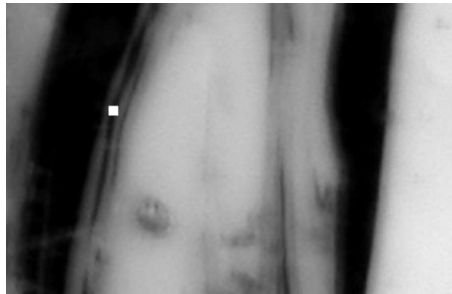
```
marker = false(size(mask));  
marker(65:70,65:70) = true;
```

To show where these pixels of interest fall on the original image, this code superimposes the marker over the mask. The small white square marks the spot. This code is not essential to the impose minima operation.

```
J = mask;  
J(marker) = 255;  
figure, imshow(J); title('Marker Image Superimposed on Mask');
```

# imimposemin

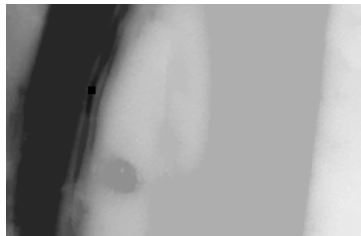
---



- 3 Impose the regional minimum on the input image using the `imimposemin` function.

The `imimposemin` function uses morphological reconstruction of the mask image with the marker image to impose the minima at the specified location. Note how all the dark areas of the original image, except the marked area, are lighter.

```
K = imimposemin(mask,marker);  
figure, imshow(K);
```

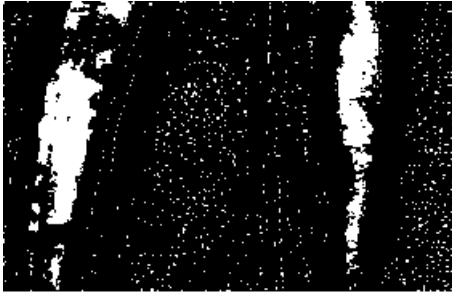


- 4 To illustrate how this operation removes all minima in the original image except the imposed minimum, compare the regional minima in the original image with the regional minimum in the processed image. These calls to `imregionalmin` return binary images that specify the locations of all the regional minima in both images.

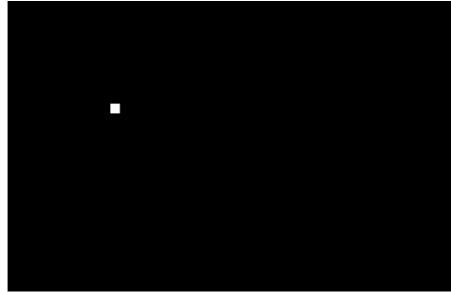
```
BW = imregionalmin(mask);  
figure, imshow(BW);
```



```
title('Regional Minima in Original Image');  
BW2 = imregionalmin(K);  
figure, imshow(BW2);  
title('Regional Minima After Processing');
```



Regional Minima in Original Image



Regional Minima After Processing

## Algorithm

`imimposemin` uses a technique based on morphological reconstruction.

## See Also

`conndef`, `imreconstruct`, `imregionalmin`

# imlincomb

---

**Purpose** Linear combination of images

**Syntax**  
`Z = imlincomb(K1,A1,K2,A2,...,Kn,An)`  
`Z = imlincomb(K1,A1,K2,A2,...,Kn,An,K)`  
`Z = imlincomb(..., output_class)`

**Description** `Z = imlincomb(K1,A1,K2,A2,...,Kn,An)` computes

$$K1*A1 + K2*A2 + \dots + Kn*An$$

where `K1`, `K2`, through `Kn` are real, double scalars and `A1`, `A2`, through `An` are real, nonsparse, numeric arrays with the same class and size. `Z` has the same class and size as `A1`.

`Z = imlincomb(K1,A1,K2,A2,...,Kn,An,K)` computes

$$K1*A1 + K2*A2 + \dots + Kn*An + K$$

where `imlincomb` adds `K`, a real, double scalar, to the sum of the products of `K1` through `Kn` and `A1` through `An`.

`Z = imlincomb(...,output_class)` lets you specify the class of `Z`. `output_class` is a string containing the name of a numeric class.

When performing a series of arithmetic operations on a pair of images, you can achieve more accurate results if you use `imlincomb` to combine the operations, rather than nesting calls to the individual arithmetic functions, such as `imadd`. When you nest calls to the arithmetic functions, and the input arrays are of an integer class, each function truncates and rounds the result before passing it to the next function, thus losing accuracy in the final result. `imlincomb` computes each element of the output `Z` individually, in double-precision floating point. If `Z` is an integer array, `imlincomb` truncates elements of `Z` that exceed the range of the integer type and rounds off fractional values.

On Intel architecture processors, `imlincomb` can take advantage of the Intel Performance Primitives Library (IPPL), thus accelerating its execution time. IPPL is activated only in the following cases:

$$Z = \text{imlincomb}(1.0, A1, 1.0, A2)$$

```
Z = imlincomb( 1.0, A1, -1.0, A2)
Z = imlincomb(-1.0, A1, 1.0, A2)
Z = imlincomb( 1.0 , A1, K)
```

where A1, A2, and Z are of class uint8, int16, or single and are of the same class.

## Examples

### Example 1

Scale an image by a factor of 2.

```
I = imread('cameraman.tif');
J = imlincomb(2,I);
imshow(J)
```

### Example 2

Form a difference image with the zero value shifted to 128.

```
I = imread('cameraman.tif');
J = uint8(filter2(fspecial('gaussian'), I));
K = imlincomb(1,I,-1,J,128); % K(r,c) = I(r,c) - J(r,c) + 128
figure, imshow(K)
```

### Example 3

Add two images with a specified output class.

```
I = imread('rice.png');
J = imread('cameraman.tif');
K = imlincomb(1,I,1,J,'uint16');
figure, imshow(K,[])
```

### Example 4

To illustrate how `imlincomb` performs all the arithmetic operations before truncating the result, compare the results of calculating the average of two arrays, X and Y, using nested arithmetic functions and then using `imlincomb`.

# imlincomb

---

In the version that uses nested arithmetic functions, `imadd` adds 255 and 50 and truncates the result to 255 before passing it to `imdivide`. The average returned in `Z(1,1)` is 128.

```
X = uint8([ 255 10 75; 44 225 100]);
Y = uint8([ 50 20 50; 50 50 50 ]);
Z = imdivide(imadd(X,Y),2)
Z =
    128    15    63
    47   128    75
```

`imlincomb` performs the addition and division in double precision and only truncates the final result. The average returned in `Z2(1,1)` is 153.

```
Z2 = imlincomb(.5,X,.5,Y)
Z2 =
    153    15    63
    47   138    75
```

## See Also

`imadd`, `imcomplement`, `imdivide`, `immultiply`, `imsubtract`

**Purpose** Create draggable, resizable line

**Syntax** `h = imline(himage,x,y)`

**Description** `h = imline(hparent,x,y)` creates a line on the object specified by `hparent`. The line can be dragged and resized interactively using the mouse. The function returns `h`, a handle to the line, which is an `hggroup` object.

`hparent` specifies the parent of the line, which is typically an axes object, but can be any object that can be the parent of an `hggroup`.

`x` and `y` specify the initial endpoint positions of the line in the form

$$X = [X1 \ X2], \ Y = [Y1 \ Y2]$$

The line has a context menu associated with it that allows you to copy the current endpoint positions to the clipboard in the form `[X1 Y1; X2 Y2]` and change the color of the line. Right-click the line to access the context menu.

## API Functions

The line `hggroup` object contains a structure of function handles, called an API, that can be used to get the current position of the line and control other aspects of its behavior and appearance. To access these functions, retrieve this API from the line using the `iptgetapi` function, as follows:

```
api = iptgetapi(h)
```

The following table lists these functions in the order they appear in the structure. The table includes the syntax for each function and a brief description. To see examples of their use, see the “Examples” on page 17-335 Examples section.

# inline

Method	Description
setPosition	Sets the endpoint positions of the line.  <code>setPosition(X,Y)</code> <code>setPosition([X1 Y1; X2 Y2])</code>
getPosition	Returns the endpoint positions of the line,  <code>pos = getPosition()</code>  where <code>pos</code> is a 2-by-2 array <code>[X1 Y1; X2 Y2]</code> .
delete	Deletes the line associated with the API.  <code>delete()</code>
setColor	Sets the color used to draw the line,  <code>setColor(new_color)</code>  where <code>new_color</code> can be a three-element vector specifying an RGB triplet, or a text string specifying the long or short names of a predefined color, such as 'white' or 'w'. For a complete list of these predefined colors and their short names, see <code>ColorSpec</code> .
addNewPositionCallback	Adds the function handle <code>fcn</code> to the list of new-position callback functions.  <code>id = addNewPositionCallback(fcn)</code>  Whenever the position of the line is changed, each function in the list is called with the syntax  <code>fcn(pos)</code>  where <code>pos</code> is a 2-by-2 array <code>[X1 Y1; X2 Y2]</code> .  The return value, <code>id</code> , is used only with the <code>removeNewPositionCallback</code> function.

<b>Method</b>	<b>Description</b>
removeNewPositionCallback	Removes the corresponding function from the new-position callback list,  removeNewPositionCallback(id)  where id is the identifier returned by the addNewPositionCallback function.
getDragConstraintFcn	Returns the function handle of the current drag constraint function.  fcn = getDragConstraintFcn()
setDragConstraintFcn	Sets the drag constraint function to be the specified function handle, fcn.  setDragConstraintFcn(fcn)  Whenever the line is moved or resized because of a mouse drag, the constraint function is called using the syntax  constrained_position = fcn(new_position)  where new_position is a 2-by-2 array [X1 Y1; X2 Y2].  You can use this function to control where the line can be moved and resized.

**Remarks**

If you use `imline` with an axis that contains an image object, and do not specify a drag constraint function, users can drag the line outside the extent of the image and lose the line. When used with an axis created by the `plot` function, the axis limits automatically expand to accommodate the movement of the line.

**Examples**

Create a line and specify a custom color for displaying the line.

```
figure, imshow('pout.tif');
```

# imline

---

```
h = imline(gca,[10 100], [100 100]);
api = iptgetapi(h);
api.setColor([0 1 0]);
```

To explore the context menu of the line, right-click the line.

Use the `addNewPositionCallback` function.

```
figure, imshow('pout.tif');
h = imline(gca,[10 100], [100 100]);
api = iptgetapi(h);
id = api.addNewPositionCallback(@(pos) title(mat2str(pos,3)));
```

Using the mouse, move the line. Note that the 2-by-2 position vector of the line is displayed as a title over the axes. To remove the callback use the `removeNewPositionCallback` function.

```
api.removeNewPositionCallback(id);
```

Use the `makeConstrainToRectFcn` function to prevent dragging line outside extent of image.

```
figure, imshow('pout.tif');
h = imline(gca,[10 100], [100 100]);
api = iptgetapi(h);
fcn = makeConstrainToRectFcn('imline',...
                             get(gca,'XLim'),get(gca,'YLim'));
api.setDragConstraintFcn(fcn);
```

## See Also

`iptgetapi`, `impoint`, `imrect`



**Purpose** Magnification box for scroll panel

**Syntax** `hbox = immagbox(hparent,himage)`

**Description** `hbox = immagbox(hparent,himage)` creates a Magnification box for the image displayed in a scroll panel created by `imscrollpanel`. `hparent` is a handle to the figure or `uipanel` object that will contain the Magnification box. `himage` is a handle to the target image (the image in the scroll panel). `immagbox` returns `hbox`, which is a handle to the Magnification box `uicontrol` object

A Magnification box is an editable text box `uicontrol` that contains the current magnification of the target image. When you enter a new value in the magnification box, the magnification of the target image changes. When the magnification of the target image changes for any reason, the magnification box updates the magnification value.

**API Functions** A Magnification box contains a structure of function handles, called an API. You can use the functions in this API to manipulate magnification box. To retrieve this structure, use the `iptgetapi` function.

```
api = iptgetapi(hbox)
```

The API for the Magnification box includes the following function.

Field	Description
<code>setMagnification</code>	<p>Sets the magnification in units of screen pixels per image pixel.</p> <pre>setMagnification(new_mag)</pre> <p>where <code>new_mag</code> is a scalar magnification factor. Multiply <code>new_mag</code> by 100 to get percent magnification. For example if you call <code>setMagnification(2)</code>, the magnification box will show the string '200%'.</p>

**Examples** Add a magnification box to a scrollable image. Because the toolbox scrollable navigation is incompatible with standard MATLAB figure

window navigation tools, the example suppresses the toolbar and menu bar in the figure window. The example positions the scroll panel in the figure window to allow room for the magnification box.

```
hFig = figure('Toolbar','none',...
             'Menubar','none');
hIm = imshow('pears.png');
hSP = imscrollpanel(hFig,hIm);
set(hSP,'Units','normalized',...
     'Position',[0 .1 1 .9])

hMagBox = immagbox(hFig,hIm);
pos = get(hMagBox,'Position');
set(hMagBox,'Position',[0 0 pos(3) pos(4)])
```

Change the magnification of the image in the scroll panel, using the scroll panel API function `setMagnification`. Notice how the magnification box updates.

```
apiSP = iptgetapi(hSP);
apiSP.setMagnification(2)
```

## See also

`imscrollpanel`, `iptgetapi`

<b>Purpose</b>	Make movie from multiframe image
<b>Syntax</b>	<pre>mov = immovie(X,map) mov = immovie(RGB)</pre>
<b>Description</b>	<p><code>mov = immovie(X,map)</code> returns the movie structure array <code>mov</code> from the images in the multiframe indexed image <code>X</code> with the colormap <code>map</code>. As it creates the movie array, <code>immovie</code> displays the movie frames on the screen. You can play the movie using the MATLAB <code>movie</code> function. For details about the movie structure array, see the reference page for <code>getframe</code>.</p> <p><code>X</code> comprises multiple indexed images, all having the same size and all using the colormap <code>map</code>. <code>X</code> is an <code>m-by-n-by-1-by-k</code> array, where <code>k</code> is the number of images.</p> <p><code>mov = immovie(RGB)</code> returns the movie structure array <code>mov</code> from the images in the multiframe, truecolor image <code>RGB</code>.</p> <p><code>RGB</code> comprises multiple truecolor images, all having the same size. <code>RGB</code> is an <code>m-by-n-by-3-by-k</code> array, where <code>k</code> is the number of images.</p>
<b>Remarks</b>	You can also use the MATLAB function <code>avifile</code> to make movies from images. The <code>avifile</code> function creates AVI files. To convert an existing MATLAB movie into an AVI file, use the <code>movie2avi</code> function.
<b>Class Support</b>	An indexed image can be <code>uint8</code> , <code>uint16</code> , <code>single</code> , <code>double</code> , or <code>logical</code> . A truecolor image can be <code>uint8</code> , <code>uint16</code> , <code>single</code> , or <code>double</code> . <code>mov</code> is a MATLAB movie structure.
<b>Examples</b>	<pre>load mri mov = immovie(D,map); movie(mov,3)</pre>
<b>See Also</b>	<code>avifile</code> , <code>getframe</code> , <code>montage</code> , <code>movie</code> , <code>movie2avi</code>

# immultiply

---

**Purpose** Multiply two images or multiply image by constant

**Syntax** `Z = immultiply(X,Y)`

**Description** `Z = immultiply(X,Y)` multiplies each element in array `X` by the corresponding element in array `Y` and returns the product in the corresponding element of the output array `Z`.

If `X` and `Y` are real numeric arrays with the same size and class, then `Z` has the same size and class as `X`. If `X` is a numeric array and `Y` is a scalar double, then `Z` has the same size and class as `X`.

If `X` is logical and `Y` is numeric, then `Z` has the same size and class as `Y`. If `X` is numeric and `Y` is logical, then `Z` has the same size and class as `X`.

`immultiply` computes each element of `Z` individually in double-precision floating point. If `X` is an integer array, then elements of `Z` exceeding the range of the integer type are truncated, and fractional values are rounded.

---

**Note** On Intel architecture processors, `immultiply` can take advantage of the Intel Performance Primitives Library (IPPL), thus accelerating its execution time. IPPL is activated only if arrays `X`, `Y`, and `Z` are of class `logical`, `uint8`, or `single`, and are of the same class.

---

## Examples

Multiply an image by itself. Note how the example converts the class of the image from `uint8` to `uint16` before performing the multiplication to avoid truncating the results.

```
I = imread('moon.tif');
I16 = uint16(I);
J = immultiply(I16,I16);
imshow(I), figure, imshow(J)
```

Scale an image by a constant factor:

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

```
J = immultiply(I,0.5);  
subplot(1,2,1), imshow(I)  
subplot(1,2,2), imshow(J)
```

## See also

[imabsdiff](#), [imadd](#), [imcomplement](#), [imdivide](#), [imlincomb](#), [imsubtract](#),  
[ippl](#)

# imnoise

---

**Purpose** Add noise to image

**Syntax**  
`J = imnoise(I,type)`  
`J = imnoise(I,type,parameters)`

**Description** `J = imnoise(I,type)` adds noise of a given type to the intensity image `I`. `type` is a string that can have one of these values.

Value	Description
'gaussian'	Gaussian white noise with constant mean and variance
'localvar'	Zero-mean Gaussian white noise with an intensity-dependent variance
'poisson'	Poisson noise
'salt & pepper'	On and off pixels
'speckle'	Multiplicative noise

`J = imnoise(I,type,parameters)` Depending on `type`, you can specify additional parameters to `imnoise`. All numerical parameters are normalized; they correspond to operations with images with intensities ranging from 0 to 1.

`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,'localvar',V)` adds zero-mean, Gaussian white noise of local variance `V` to the image `I`. `V` is an array of the same size as `I`.

`J = imnoise(I,'localvar',image_intensity,var)` adds zero-mean, Gaussian noise to an image `I`, where the local variance of the noise, `var`, is a function of the image intensity values in `I`. The `image_intensity` and `var` arguments are vectors of the same size, and `plot(image_intensity,var)` plots the functional relationship between

noise variance and image intensity. The `image_intensity` vector must contain normalized intensity values ranging from 0 to 1.

`J = imnoise(I, 'poisson')` generates Poisson noise from the data instead of adding artificial noise to the data. If `I` is double precision, then input pixel values are interpreted as means of Poisson distributions scaled up by  $1e12$ . For example, if an input pixel has the value  $5.5e-12$ , then the corresponding output pixel will be generated from a Poisson distribution with mean of 5.5 and then scaled back down by  $1e12$ . If `I` is single precision, the scale factor used is  $1e6$ . If `I` is `uint8` or `uint16`, then input pixel values are used directly without scaling. For example, if a pixel in a `uint8` input has the value 10, then the corresponding output pixel will be generated from a Poisson distribution with mean 10.

`J = imnoise(I, 'salt & pepper', d)` adds salt and pepper noise to the image `I`, where `d` is the noise density. This affects approximately  $d * \text{numel}(I)$  pixels. The default for `d` is 0.05.

`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.

---

**Note** The mean and variance parameters for 'gaussian', 'localvar', and 'speckle' noise types are always specified as if the image were of class `double` in the range [0, 1]. If the input image is of class `uint8` or `uint16`, the `imnoise` function converts the image to `double`, adds noise according to the specified type and parameters, and then converts the noisy image back to the same class as the input.

---

## Class Support

For most noise types, `I` can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. For Poisson noise, `int16` is not allowed. The output image `J` is of the same class as `I`. If `I` has more than two dimensions it is treated as a multidimensional intensity image and not as an RGB image.

# imnoise

---

## Examples

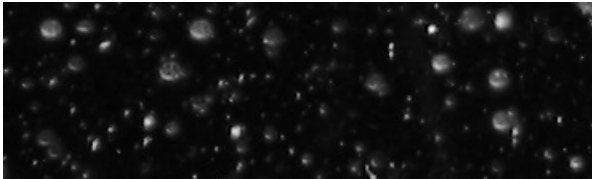
```
I = imread('eight.tif');  
J = imnoise(I,'salt & pepper',0.02);  
figure, imshow(I)  
figure, imshow(J)
```



## See Also

rand, randn in the MATLAB Function Reference

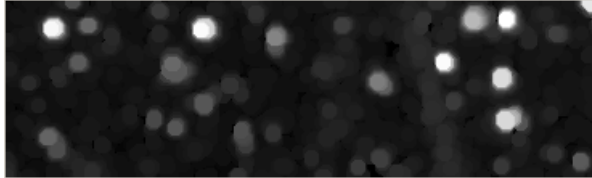


<b>Purpose</b>	Morphologically open image
<b>Syntax</b>	<pre>IM2 = imopen(IM,SE) IM2 = imopen(IM,NHOOD)</pre>
<b>Description</b>	<p><code>IM2 = imopen(IM,SE)</code> performs morphological opening on the grayscale or binary image <code>IM</code> with the structuring element <code>SE</code>. The argument <code>SE</code> must be a single structuring element object, as opposed to an array of objects.</p> <p><code>IM2 = imopen(IM,NHOOD)</code> performs opening with the structuring element <code>strel(NHOOD)</code>, where <code>NHOOD</code> is an array of 0's and 1's that specifies the structuring element neighborhood.</p>
<b>Class Support</b>	<code>IM</code> can be any numeric or logical class and any dimension, and must be nonsparse. If <code>IM</code> is logical, then <code>SE</code> must be flat. <code>IM2</code> has the same class as <code>IM</code> .
<b>Examples</b>	<p>Remove the smaller objects in an image.</p> <ol style="list-style-type: none"><li>1 Read the image into the MATLAB workspace and display it. <pre>I = imread('snowflakes.png'); imshow(I)</pre></li><li>2 Create a disk-shaped structuring element with a radius of 5 pixels. <pre>se = strel('disk',5);</pre></li><li>3 Remove snowflakes having a radius less than 5 pixels by opening it with the disk-shaped structuring element created in step 2.</li></ol>

# imopen

---

```
I_opened = imopen(I,se);  
figure, imshow(I_opened,[])
```



**See Also** `imclose`, `imdilate`, `imerode`, `strel`

**Purpose** Overview tool for image displayed in scroll panel

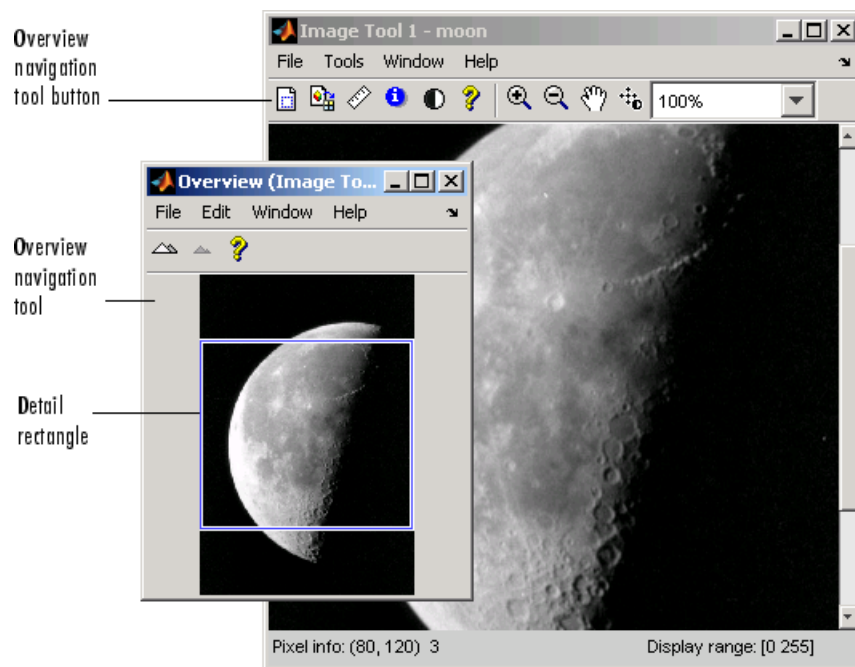
**Syntax**  
`imoverview(himage)`  
`hfig = imoverview(himage)`

**Description** `imoverview(himage)` creates an Overview tool associated with the image specified by the handle `himage`, called the target image. The target image must be contained in a scroll panel created by `imscrollpanel`.

The Overview tool is a navigation aid for images displayed in a scroll panel. `imoverview` creates the tool in a separate figure window that displays the target image in its entirety, scaled to fit. Over this scaled version of the image, the tool draws a rectangle, called the detail rectangle, that shows the portion of the target image that is currently visible in the scroll panel. To view portions of the image that are not currently visible in the scroll panel, move the detail rectangle in the Overview tool.

The following figure shows the Image Tool with the Overview tool.

# imoverview



`hfig = imoverview(...)` returns a handle to the Overview tool figure.

## Note

To create an Overview tool that can be embedded in an existing figure or uipanel object, use `imoverviewpanel`.

## Examples

Create a figure, disabling the toolbar and menubar, because the toolbox navigation tools are not compatible with the standard MATLAB zoom and pan tools. Then create a scroll panel in the figure and use scroll panel API functions to set the magnification.

```
hFig = figure('Toolbar','none',...  
             'Menubar','none');  
hIm = imshow('tape.png');  
hSP = imscrollpanel(hFig,hIm);  
api = iptgetapi(hSP);
```

```
api.setMagnification(2) % 2X = 200%  
imoverview(hIm)
```

**See Also**

imoverviewpanel, imscrollpanel

# imoverviewpanel

---

**Purpose** Overview tool panel for image displayed in scroll panel

**Syntax** `hpanel = imoverviewpanel(hparent,himage)`

**Description** `hpanel=imoverviewpanel(hparent,himage)` creates an Overview tool panel associated with the image specified by the handle `himage`, called the target image. `himage` must be contained in a scroll panel created by `imscrollpanel`. `hparent` is a handle to the figure or `uipanel` object that will contain the Overview tool panel. `imoverviewpanel` returns `hpanel`, a handle to the Overview tool `uipanel` object

The Overview tool is a navigation aid for images displayed in a scroll panel. `imoverviewpanel` creates the tool in a `uipanel` object that can be embedded in a figure or `uipanel` object. The tool displays the target image in its entirety, scaled to fit. Over this scaled version of image, the tool draws a rectangle, called the detail rectangle, that shows the portion of the target image that is currently visible in the scroll panel. To view portions of the image that are not currently visible in the scroll panel, move the detail rectangle in the Overview tool.

**Note** To create an Overview tool in a separate figure, use `imoverview`. When created using `imoverview`, the Overview tool includes zoom-in and zoom-out buttons.

**Examples** Create an Overview tool that is embedded in the same figure that contains the target image.

```
hFig = figure('Toolbar','none','Menubar','none');
hIm = imshow('tissue.png');
hSP = imscrollpanel(hFig,hIm);
set(hSP,'Units','normalized','Position',[0 .5 1 .5])
hOvPanel = imoverviewpanel(hFig,hIm);
set(hOvPanel,'Units','Normalized',...
'Position',[0 0 1 .5])
```

**See Also** `imoverview`, `imscrollpanel`

**Purpose**

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 syntax 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 syntax.

```
P = impixel(I,c,r)
```

# impixel

---

```
P = impixel(X,map,c,r)
P = impixel(RGB,c,r)
```

`r` and `c` are equal-length vectors specifying the coordinates of the pixels whose RGB values are returned in `P`. The  $k$ th row of `P` contains the RGB values for the pixel  $(r(k),c(k))$ .

If you supply three output arguments, `impixel` returns the coordinates of the selected pixels. For example,

```
[c,r,P] = impixel(...)
```

To specify a nondefault spatial coordinate system for the input image, use these syntax.

```
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`, `int16`, `single`, `double`, or `logical`. All other inputs are of class `double`.

If the input is `double`, the output `P` is `double`. For all other input classes the output is `single`. The rest of the outputs are `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.

## Examples

```
RGB = imread('peppers.png');
c = [12 146 410];
r = [104 156 129];
pixels = impixel(RGB,c,r)
```

```
pixels =
```

```
    62    34    63
   166    54    60
    59    28    47
```

## See Also

`improfile`, `pixval`

# impixelinfo

---

**Purpose** Pixel Information tool

**Syntax**

```
impixelinfo
impixelinfo(h)
impixelinfo(hparent,himage)
hpanel = impixelinfo(...)
```

**Description** `impixelinfo` creates a Pixel Information tool in the current figure. The Pixel Information tool displays information about the pixel in an image that the cursor is positioned over. The tool can display pixel information for all the images in a figure.

The Pixel Information tool is a uipanel object, positioned in the lower-left corner of the figure. The tool contains the text string `Pixel info:` followed by the pixel information. Before you move the cursor over the image, the tool contains the default pixel information text string `(X,Y) Pixel Value`. Once you move the cursor over the image, the information displayed varies by image type, as shown in the following table. If you move the cursor off the image, the pixel information tool displays the default pixel information string for that image type.

Image Type	Pixel Information	Example
Intensity	(X,Y) Intensity	(13,30) 82
Indexed	(X,Y) <index> [R G B]	(2,6) <4> [0.29 0.05 0.32]
Binary	(X,Y) BW	(12,1) 0
Truecolor	(X,Y) [R G B]	(19,10) [15 255 10]
Floating point image with <code>CDataMapping</code> property set to <code>direct</code>	(X,Y) value <index> [R G B]	(19,10) 82 <4> [15 255 10]

For example, for grayscale (intensity) images, the pixel information tool displays the  $x$  and  $y$  coordinates of the pixel and its value, as shown in the following figure.

X and Y coordinates	Pixel Value
Pixel info: (418, 261) 143	

If you want to display the pixel information without the “Pixel Info” label, use the `impixelinfoval` function.

`impixelinfo(h)` creates a Pixel Information tool in the figure specified by `h`, where `h` is a handle to an image, axes, uipanel, or figure object. Axes, uipanel, or figure objects must contain at least one image object.

`impixelinfo(hparent,himage)` creates a Pixel Information tool in `hparent` that provides information about the pixels in `himage`. `himage` is a handle to an image or an array of image handles. `hparent` is a handle to the figure or uipanel object that contains the pixel information tool.

`hpanel=impixelinfo(...)` returns a handle to the Pixel Information tool `uipanel`.

## Note

To copy the pixel information string to the clipboard, right-click while the cursor is positioned over a pixel. In the context menu displayed, choose **Copy pixel info**.

## Examples

Display an image and add a Pixel Information tool to the figure. The example shows how you can change the position of the tool in the figure using properties of the tool `uipanel` object.

```
h = imshow('hestain.png');
hp = impixelinfo;
set(hp,'Position',[150 290 300 20]);
```

Use the Pixel Information tool in a figure containing multiple images of different types.

# imshowinfo

---

```
figure
subplot(1,2,1), imshow('liftingbody.png');
subplot(1,2,2), imshow('autumn.tif');
imshowinfo;
```

## See Also

`imshowinfoval`, `imtool`

**Purpose** Pixel Information tool without text label

**Syntax** `hcontrol = impixelinfoval(hparent,himage)`

**Description** `hcontrol = impixelinfoval(hparent,himage)` creates a Pixel Information tool in `hparent` that provides information about the pixels in the image specified by `himage`. `hparent` is a handle to a figure or uipanel object. `himage` can be a handle to an image or an array of image handles.

The Pixel Information tool displays information about the pixel in an image that the cursor is positioned over. The tool displays pixel information for all the images in a figure.

When created with `impixelinfo`, the tool is a uipanel object, positioned in the lower-left corner of the figure, that contains the text label Pixel Info: followed by the *x*- and *y*-coordinates of the pixel and its value. When created with `impixelinfoval`, the tool is a uicontrol object positioned in the lower-left corner of the figure, that displays the pixel information without the text label, as shown in the following figure.

X and Y coordinates	Pixel Value
(167, 251)	114

The information displayed depends on the image type. See `impixelinfo` for details.

To copy the pixel value string to the Clipboard, right-click while the cursor is positioned over a pixel. In the context menu displayed, choose **Copy pixel info**.

## Examples

Add a Pixel Information tool to a figure. Note how you can change the style and size of the font used to display the value in the tool using standard Handle Graphics commands.

```
ankle = dicomread('CT-MONO2-16-ankle.dcm');
h = imshow(ankle,[]);
```

# impixelinfoval

---

```
hText = impixelinfoval(gcf,h);  
set(hText,'FontWeight','bold')  
set(hText,'FontSize',10)
```

**See also** [impixelinfo](#)

**Purpose** Pixel Region tool

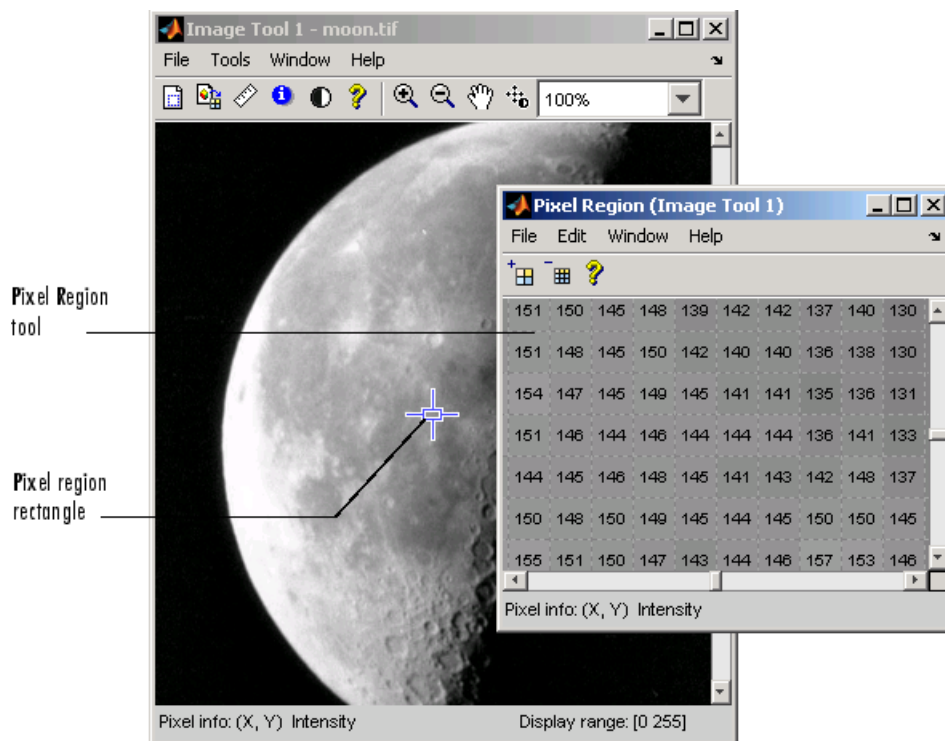
**Syntax**

```
impixelregion  
impixelregion(h)  
hfig = impixelregion(...)
```

**Description** `impixelregion` creates a Pixel Region display tool associated with the image displayed in the current figure, called the target image.

The Pixel Region tool opens a separate figure window containing an extreme close-up view of a small region of pixels in the target image, shown in the following figure. The tool superimposes the numeric value of the pixel over each pixel. To define the region being examined, the tool overlays a rectangle on the target image, called the pixel region rectangle. To view pixels in a different region, click and drag the rectangle over the target image.

# impixelregion



`impixelregion(h)` creates a Pixel Region tool associated with the object specified by the handle `h`. `h` can be a handle to a figure, axes, uipanel, or image object. If `h` is a handle to an axes or figure, `impixelregion` associates the tool with the first image found in the axes or figure.

`hfig=impixelregion(...)` returns `hfig`, a handle of the Pixel Region tool figure.

## Note

To create a Pixel Region tool that can be embedded in an existing figure window or uipanel, use `impixelregionpanel`.

## Examples

Display an image and then create a Pixel Region tool associated with the image.



```
imshow peppers.png  
impixelregion
```

**See Also**

`impixelinfo`, `impixelregionpanel`, `imtool`

# impixelregionpanel

---

**Purpose** Pixel Region tool panel

**Syntax** `hpanel = impixelregionpanel(hparent,himage)`

**Description** `hpanel = impixelregionpanel(hparent,himage)` creates a Pixel Region tool panel associated with the image specified by the handle `himage`, called the target image. This is the image whose pixels are to be displayed. `hparent` is the handle to the figure or `uipanel` object that will contain the Pixel Region tool panel. `hpanel` is the handle to the Pixel Region tool scroll panel.

The Pixel Region tool is a `uipanel` object that contains an extreme close-up view of a small region of pixels in the target image. The tool superimposes the numeric value of the pixel over each pixel. To define the region being examined, the tool overlays a rectangle on the target image, called the pixel region rectangle. To view pixels in a different region, click and drag the rectangle over the target image.

**Note** To create a Pixel Region tool in a separate figure window, use `impixelregion`.

**Examples**

```
himage = imshow('peppers.png');  
hfigure = figure;  
hpanel = impixelregionpanel(hfigure, himage);
```

Set the panel's position to the lower-left quadrant of the figure.

```
set(hpanel, 'Position', [0 0 .5 .5])
```

**See Also** `impixelregion`, `impositionrect`, `imtool`

**Purpose** Create draggable point

**Syntax** `h = impoint(hparent,x,y)`

**Description** `h = impoint(hparent,x,y)` creates a draggable point on the object specified by `hparent`. A draggable point is an `hggroup` object that can be moved interactively using the mouse. `hparent` specifies the `hggroup`'s parent, which is typically an axes object, but can also be any other object that can be the parent of an `hggroup`. `x` and `y` are both scalars that together specify the initial position of the point. The function returns `h`, a handle to the point `hggroup` object.

The draggable point has a context menu associated with it that allows you to copy the current position to the clipboard and change the color used to display the point.

**API Functions** The point `hggroup` object contains a structure, called an API. This structure contains function handles that support various operations on the point. To access these functions, use the `iptgetapi` function, passing it a handle to the point, as follows:

```
api = iptgetapi(h)
```

The following lists these functions in the order they appear in the API structure.

Method	Description
<code>setPosition</code>	Sets the draggable point to a new position. <code>setPosition(new_x, new_y)</code> <code>setPosition([new_x new_y])</code>
<code>getPosition</code>	Returns the current position of the point, <code>pos = getPosition()</code> where <code>pos</code> is a two-element vector <code>[x y]</code> .

Method	Description
delete	Deletes the draggable point associated with the API. <code>delete()</code>
setColor	Sets the color used to draw the draggable point, <code>setColor(new_color)</code> where <code>new_color</code> can be a three-element vector specifying an RGB triplet, or a text string specifying the long or short names of a predefined color, such as 'white' or 'w'. For a complete list of these predefined colors and their short names, see <code>ColorSpec</code> .
addNewPositionCallback	Adds the function handle <code>fcn</code> to the list of new-position callback functions. <code>id = addNewPositionCallback(fcn)</code> Whenever the point changes its position, each function in the list is called with the syntax <code>fcn(pos)</code> where <code>pos</code> is a two-element vector <code>[x y]</code> . The return value, <code>id</code> , is used only with the <code>removeNewPositionCallback</code> function.
removeNewPositionCallback	Removes the corresponding function from the new-position callback list, <code>removeNewPositionCallback(id)</code> where <code>id</code> is the identifier returned by the <code>addNewPositionCallback</code> function.
getDragConstraintFcn	Returns the function handle of the current drag constraint function. <code>fcn = getDragConstraintFcn()</code>

Method	Description
setDragConstraintFcn	<p>Sets the drag constraint function to be the specified function handle, fcn.</p> <pre>setDragConstraintFcn(fcn)</pre> <p>Whenever the draggable point is moved because of a mouse drag, the constraint function is called using the syntax</p> <pre>constrained_position = fcn(new_position)</pre> <p>where new_position is a two-element vector [new_x new_y].</p> <p>This allows a client, for example, to control where the point may be dragged.</p>
setString	<p>Sets the string for the optional text label,</p> <pre>setString(s)</pre> <p>where s is a string to be placed to the lower right of the point.</p>

## Remarks

If you use `impoint` with an axis that contains an image object, and do not specify a drag constraint function, users can drag the point outside the extent of the image and lose the point. When used with an axes created by the `plot` function, the axes limits automatically expand when the point is dragged outside the extent of the axes.

## Examples

### Example 1

Display updated position in the title. Specify a drag constraint function using `makeConstainToRectFcn` to keep the point inside the original `Xlim` and `Ylim` ranges.

```
figure, imshow rice.png
h = impoint(gca,100,200);
api = iptgetapi(h);
```

```
api.addNewPositionCallback(@(p) title(...
    ['(',mat2str(p(1)),',',mat2str(p(2)),']'));
fcn = makeConstrainToRectFcn('impoint',...
    get(gca,'XLim'),get(gca,'YLim'));
api.setDragConstraintFcn(fcn);
```

Now drag the point using the mouse.

## Example 2

Use a custom color for displaying the point. Specify a drag constraint function using `makeConstrainToRectFcn` to keep the point inside the original `Xlim` and `Ylim` ranges.

```
figure, plot(1:10)
h = impoint(gca,6,4);
api = iptgetapi(h);
api.setColor([1 0 0]);
fcn = makeConstrainToRectFcn('impoint',...
    get(gca,'XLim'),get(gca,'YLim'));
api.setDragConstraintFcn(fcn);
```

## Example 3

Add string labels to each point.

```
figure, plot(1:10)
h1 = impoint(gca,3,6);
api1 = iptgetapi(h1);
api1.setString('1')
h2 = impoint(gca,7,2);
api2 = iptgetapi(h2);
api2.setString('2')
```

## See Also

`iptgetapi`, `imline`, `imrect`

**Purpose** Create draggable position rectangle

**Syntax** `H = impositionrect(hparent,position)`

---

**Note** This function is obsolete and may be removed in future versions. Use `imrect` instead.

---

**Description** `H = impositionrect(hparent,position)` creates a position rectangle on the object specified by `hparent`. The function returns `H`, a handle to the position rectangle, which is an `hggroup` object. `hparent` specifies the `hggroup`'s parent, which is typically an axes object, but can also be any other object that can be the parent of an `hggroup`. `position` is a four-element position vector that specifies the initial location of the rectangle. `position` has the form `[XMIN YMIN WIDTH HEIGHT]`.

All measurements are in units specified by the `Units` property axes object. When you do not specify the `position` argument, `impositionrect` uses `[0 0 1 1]` as the default value.

**Remarks** A position rectangle can be dragged interactively using the mouse. When the position rectangle occupies a small number of screen pixels, its appearance changes to aid visibility.

The position rectangle has a context menu associated with it that you can use to copy the current position to the clipboard and change the color used to display the rectangle.

**API Function Syntaxes** A position rectangle contains a structure of function handles, called an API, that can be used to manipulate it. To retrieve this structure from the position rectangle, use the `iptgetapi` function.

`API = iptgetapi(H)`

The following lists the functions in the position rectangle API in the order they appear in the API structure.

# impositionrect

Function	Description
setPosition	<p>Sets the position rectangle to a new position.</p> <pre>api.setPosition(new_position)</pre> <p>where <code>new_position</code> is a four-element position vector.</p>
getPosition	<p>Returns the current position of the position rectangle.</p> <pre>position = api.getPosition()</pre> <p><code>position</code> is a four-element position vector.</p>
delete	<p>Deletes the position rectangle associated with the API.</p> <pre>api.delete()</pre>
setColor	<p>Sets the color used to draw the position rectangle.</p> <pre>api.setColor(new_color)</pre> <p>where <code>new_color</code> can be a three-element vector specifying an RGB triplet, or a text string specifying the long or short names of a predefined color, such as 'white' or 'w'. For a complete list of these predefined colors and their short names, see <code>ColorSpec</code>.</p>
addNewPositionCallback	<p>Adds the function handle <code>fun</code> to the list of new-position callback functions.</p> <pre>id = api.addNewPositionCallback(fun)</pre> <p>Whenever the position rectangle changes its position, each function in the list is called with the syntax:</p> <pre>fun(position)</pre> <p>The return value, <code>id</code>, is used only with <code>removeNewPositionCallback</code>.</p>



Function	Description
removeNewPositionCallback	<p>Removes the corresponding function from the new-position callback list.</p> <pre>api.removeNewPositionCallback(id)</pre> <p>where <code>id</code> is the identifier returned by <code>api.addNewPositionCallback</code></p>
setDragConstraintCallback	<p>Sets the drag constraint function to be the specified function handle, <code>fcn</code>.</p> <pre>api.setDragConstraintCallback(fcn)</pre> <p>Whenever the position rectangle is moved because of a mouse drag, the constraint function is called using the syntax:</p> <pre>constrained_position = fcn(new_position)</pre> <p>where <code>new_position</code> is a four-element position vector. This allows a client, for example, to control where the position rectangle may be dragged.</p>

## Examples

Display in the command window the updated position of the position rectangle as it moves in the axes.

```
close all, plot(1:10)
h = impositionrect(gca, [4 4 2 2]);
api = iptgetapi(h);
api.addNewPositionCallback(@(p) disp(p));
```

Constrain the position rectangle to move only up and down.

```
close all, plot(1:10)
h = impositionrect(gca, [4 4 2 2]);
api = getappdata(h, 'API');
```

# impositionrect

---

```
api.setDragConstraintCallback(@(p) [4 p(2:4)]);
```

Specify the color of the position rectangle.

```
close all, plot(1:10)
h = impositionrect(gca, [4 4 2 2]);
api = iptgetapi(h, 'API');
api.setColor([1 0 0]);
```

When the position rectangle occupies only a few pixels on the screen, the rectangle is drawn in a different style to increase its visibility.

```
close all, imshow cameraman.tif
h = impositionrect(gca, [100 100 10 10]);
```

## See Also

`iptgetapi`

**Purpose**

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 images and RGB images.

If you call `improfile` with one of these syntax, 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 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

# improfile

---

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 syntax.

```
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 syntax 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 syntax.

```
[...] = 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. The default value is enclosed in braces (`{}`).

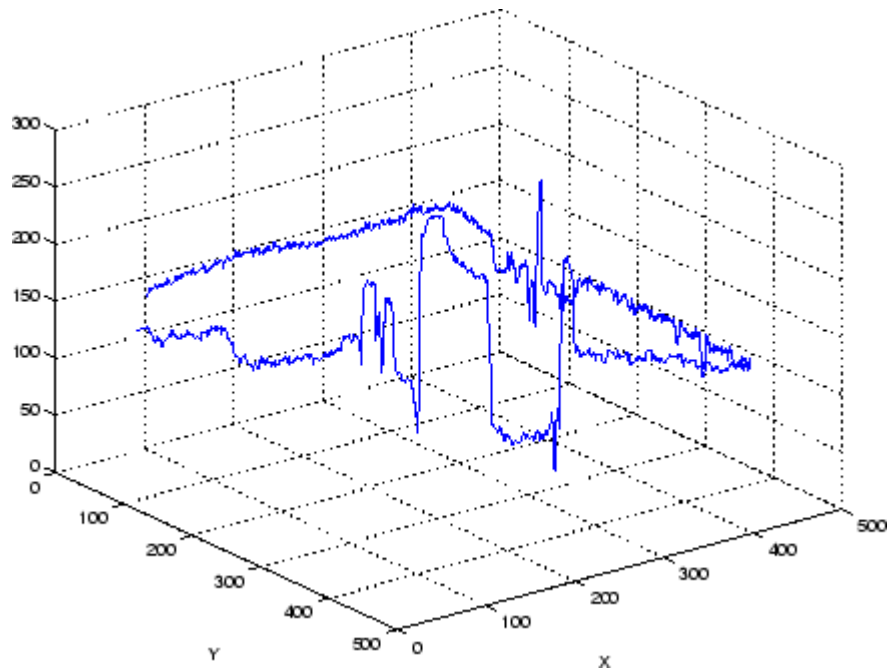
Value	Description
{'nearest'}	Nearest-neighbor interpolation
'bilinear'	Bilinear interpolation
'bicubic'	Bicubic interpolation

## Class Support

The input image can be uint8, uint16, int16, single, double, or logical. All other inputs and outputs must be double.

## Examples

```
I = imread('liftingbody.png');  
x = [19 427 416 77];  
y = [96 462 37 33];  
improfile(I,x,y),grid on;
```



# improfile

---

## See Also

`impixel`

`interp2` in the MATLAB Function Reference

**Purpose** Read image from graphics file

**Note** imread is a MATLAB function.

# imreconstruct

---

**Purpose** Morphological reconstruction

**Syntax**  
IM = imreconstruct(marker,mask)  
IM = imreconstruct(marker,mask,conn)

**Description** IM = imreconstruct(marker,mask) performs morphological reconstruction of the image marker under the image mask. marker and mask can be two intensity images or two binary images with the same size. The returned image IM is an intensity or binary image, respectively. marker must be the same size as mask, and its elements must be less than or equal to the corresponding elements of mask.

By default, imreconstruct uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, imreconstruct uses conndef(ndims(I), 'maximal').

IM = imreconstruct(marker,mask,conn) performs morphological reconstruction with the specified connectivity. conn can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can be defined in a more general way for any dimension by using for conn a 3-by-3-by- ... -by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood locations relative to the center element of conn. Note that conn must be symmetric about its center element.



Morphological reconstruction is the algorithmic basis for several other Image Processing Toolbox functions, including `imclearborder`, `imextendedmax`, `imextendedmin`, `imfill`, `imhmax`, `imhmin`, and `imimposemin`.

## **Class Support**

`marker` and `mask` must be nonsparse numeric or logical arrays with the same class and any dimension. `IM` is of the same class as `marker` and `mask`.

## **Algorithm**

`imreconstruct` uses the fast hybrid grayscale reconstruction algorithm described in [1].

## **See Also**

`imclearborder`, `imextendedmax`, `imextendedmin`, `imfill`, `imhmax`, `imhmin`, `imimposemin`

## **Reference**

[1] Vincent, L., "Morphological Grayscale Reconstruction in Image Analysis: Applications and Efficient Algorithms," *IEEE Transactions on Image Processing*, Vol. 2, No. 2, April, 1993, pp. 176-201.

# imrect

---

**Purpose** Create draggable rectangle

**Syntax** `h = imrect(hparent,position)`

**Description** `h = imrect(hparent,position)` creates a draggable rectangle on the object specified by `hparent`. A draggable rectangle is an `hggroup` object that can be moved interactively using the mouse. `hparent` specifies the `hggroup`'s parent, which is typically an axes object, but can also be any other object that can be the parent of an `hggroup`. `position` is a four-element position vector that specifies the initial size and location of the rectangle of the form:

```
[xmin ymin width height]
```

All measurements are in units specified by the `Units` property of the axes object. The function returns `h`, a handle to the rectangle, which is an `hggroup` object.

A rectangle can be dragged interactively using the mouse. When the rectangle occupies a small number of screen pixels, its appearance changes to aid visibility.

The draggable rectangle has a context menu associated with it that you can use to copy the current position to the clipboard and change the color used to display the rectangle.

**API Functions** A draggable rectangle contains a structure of function handles, called an API, that can be used to manipulate it. To retrieve this structure, use the `iptgetapi` function.

```
API = iptgetapi(H)
```

The following lists the functions in the rectangle API in the order they appear in the API structure.

Function	Description
setPosition	<p>Sets the rectangle to a new position,</p> <pre>setPosition(new_position)</pre> <p>where <code>new_position</code> is a four-element position vector.</p>
getPosition	<p>Returns the current position of the rectangle.</p> <pre>position = getPosition()</pre>
delete	<p>Deletes the rectangle associated with the API.</p> <pre>delete()</pre>
setColor	<p>Sets the color used to draw the rectangle,</p> <pre>setColor(new_color)</pre> <p>where <code>new_color</code> can be a three-element vector specifying an RGB triplet, or a text string specifying the long or short names of a predefined color, such as 'white' or 'w'. For a complete list of these predefined colors and their short names, see <code>ColorSpec</code>.</p>
addNewPositionCallback	<p>Adds the function handle <code>fcn</code> to the list of new-position callback functions.</p> <pre>id = addNewPositionCallback(fcn)</pre> <p>Whenever the rectangle changes its position, each function in the list is called with the syntax</p> <pre>fcn(position)</pre> <p>The return value, <code>id</code>, is used only with the <code>removeNewPositionCallback</code> function.</p>

Function	Description
removeNewPositionCallback	Removes the corresponding function from the new-position callback list,  <pre>removeNewPositionCallback(id)</pre> where <code>id</code> is the identifier returned by <code>api.addNewPositionCallback</code> .
getDragConstraintFcn	Returns the handle of the current drag constraint function.  <pre>fcn = getDragConstraintFcn()</pre>
setDragConstraintFcn	Sets the drag constraint function to be the specified function handle, <code>fcn</code> .  <pre>setDragConstraintFcn(fcn)</pre> Whenever the draggable rectangle is moved because of a mouse drag, the constraint function is called using the syntax  <pre>constrained_position = fcn(new_position)</pre> where <code>new_position</code> is a four-element position vector. This allows a client, for example, to control where the rectangle can be dragged.

## Remarks

If you use `imrect` with an axis that contains an image object, and do not specify a drag constraint function, users can drag the rectangle outside the extent of the image. When used with an axis created by the `plot` function, the axis limits automatically expand to accommodate the movement of the rectangle.

## Examples

Display updated position in the title. Specify a drag constraint function using `makeConstainToRectFcn` to keep the rectangle inside the original `Xlim` and `Ylim` ranges.

```
figure, imshow('cameraman.tif');
h = imrect(gca, [10 10 100 100]);
api = iptgetapi(h);
api.addNewPositionCallback(@(p) title(mat2str(p)));
fcn = makeConstrainToRectFcn('imrect',...
                             get(gca,'XLim'),get(gca,'YLim'));
api.setDragConstraintFcn(fcn);
```

Now drag the rectangle using the mouse.

Use a custom color for displaying the rectangle.

```
figure, imshow('cameraman.tif')
h = imrect(gca, [10 10 100 100]);
api = iptgetapi(h);
api.setColor([1 0 0]);
```

When the rectangle position occupies only a few pixels on the screen, the rectangle is drawn in a different style to increase its visibility.

```
figure, imshow cameraman.tif
h = imrect(gca, [100 100 10 10]);
```

## See Also

iptgetapi, imline, impoint

# imregionalmax

---

**Purpose** Regional maxima

**Syntax** BW = imregionalmax(I)  
BW = imregionalmax(I,conn)

**Description** BW = imregionalmax(I) finds the regional maxima of I. imregionalmax returns the binary image BW that identifies the locations of the regional maxima in I. BW is the same size as I. In BW, pixels that are set to 1 identify regional maxima; all other pixels are set to 0.

Regional maxima are connected components of pixels with a constant intensity value, and whose external boundary pixels all have a lower value.

By default, imregionalmax uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, imregionalmax uses conndef(ndims(I), 'maximal').

BW = imregionalmax(I,conn) computes the regional maxima of I using the specified connectivity. conn can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can be defined in a more general way for any dimension by using for conn a 3-by-3-by- ...-by-3 matrix of 0's and 1's. The 1-valued

elements define neighborhood locations relative to the center element of conn. Note that conn must be symmetric about its center element.

## Class Support

I can be any nonsparse, numeric class and any dimension. BW is logical.

## Examples

Create a sample image with several regional maxima.

```
A = 10*ones(10,10);
A(2:4,2:4) = 22;
A(6:8,6:8) = 33;
A(2,7) = 44;
A(3,8) = 45;
A(4,9) = 44;
A =
    10    10    10    10    10    10    10    10    10    10
    10    22    22    22    10    10    44    10    10    10
    10    22    22    22    10    10    10    45    10    10
    10    22    22    22    10    10    10    10    44    10
    10    10    10    10    10    10    10    10    10    10
    10    10    10    10    10    33    33    33    10    10
    10    10    10    10    10    33    33    33    10    10
    10    10    10    10    10    33    33    33    10    10
    10    10    10    10    10    10    10    10    10    10
    10    10    10    10    10    10    10    10    10    10
```

Find the regional maxima.

# imregionalmax

---

```
regmax = imregionalmax(A)
regmax =
    0     0     0     0     0     0     0     0     0     0
    0     1     1     1     0     0     0     0     0     0
    0     1     1     1     0     0     0     1     0     0
    0     1     1     1     0     0     0     0     0     0
    0     0     0     0     0     0     0     0     0     0
    0     0     0     0     0     1     1     1     0     0
    0     0     0     0     0     1     1     1     0     0
    0     0     0     0     0     1     1     1     0     0
    0     0     0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0     0     0
```

## See Also

conndef, imextendedmax, imhmax, imreconstruct, imregionalmin



**Purpose** Regional minima

**Syntax**  
 BW = imregionalmin(I)  
 BW = imregionalmin(I,conn)

**Description** BW = imregionalmin(I) computes the regional minima of I. The output binary image BW has value 1 corresponding to the pixels of I that belong to regional minima and 0 otherwise. BW is the same size as I.

Regional minima are connected components of pixels with a constant intensity value, and whose external boundary pixels all have a higher value.

By default, imregionalmin uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, imregionalmin uses conndef(ndims(I), 'maximal').

BW = imregionalmin(I,conn) specifies the desired connectivity. conn can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can be defined in a more general way for any dimension by using for conn a 3-by-3-by- ...-by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood locations relative to the center element of conn. Note that conn must be symmetric about its center element.

# imregionalmin

---

## Class Support

I can be any nonsparse, numeric class and any dimension. BW is logical.

## Examples

Create a 10-by-10 pixel sample image that contains two regional minima.

```
A = 10*ones(10,10);
```

```
A(2:4,2:4) = 2;
```

```
A(6:8,6:8) = 7;
```

```
A =
```

```
10 10 10 10 10 10 10 10 10 10
10 2 2 2 10 10 10 10 10 10
10 2 2 2 10 10 10 10 10 10
10 2 2 2 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 7 7 7 10 10
10 10 10 10 10 7 7 7 10 10
10 10 10 10 10 7 7 7 10 10
10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
```

Pass the sample image A to `imregionalmin`. The function returns a binary image, the same size as A, in which pixels with the value 1 represent the regional minima in A. `imregionalmin` sets all other pixels in to zero (0).

```
B = imregionalmin(A)
```

```
B =
```

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

## See Also

conndef, imextendedmin, imhmin, imimposemin, imreconstruct,  
imregionalmax

# imresize

---

**Purpose**            Resize image

**Syntax**

```
B = imresize(A,m)
B = imresize(A,m,method)
B = imresize(A,[mrows ncols],method)

B = imresize(...,method,n)
B = imresize(...,method,h)
```

**Description**    `B = imresize(A,m)` returns image `B` that is `m` times the size of `A`. The input image `A` can be an indexed, grayscale, RGB, or binary image. 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`. When resizing the image, `imresize` uses nearest-neighbor interpolation.

`B = imresize(A,m,method)` returns an image that is `m` times the size of `A` using the interpolation method specified by `method`. `method` is a string that can have one of these values. The default value is enclosed in braces (`{}`).

Value	Description
{'nearest'}	Nearest-neighbor interpolation
'bilinear'	Bilinear interpolation
'bicubic'	Bicubic interpolation

`B = imresize(A,[mrows ncols],method)` returns an image of the size specified by `[mrows ncols]`. If the specified size does not produce the same aspect ratio as the input image has, the output image is distorted.

`B = imresize(...,method,n)` 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. `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. If you do not specify a size, the default filter size is 11-by-11.

`B = imresize(...,method,h)` When the specified output size is smaller than the size of the input image, and `method` is 'bilinear' or 'bicubic', you can also specify a two-dimensional FIR filter, `h`, such as those returned by `ftrans2`, `fwind1`, `fwind2`, or `fsamp2`.

## **Class Support**

The input image can be numeric or logical and it must be nonsparse. The output image is of the same class as the input image.

## **See Also**

`imrotate`, `imtransform`, `tformarray`  
`interp2` in the MATLAB Function Reference

# imrotate

**Purpose** Rotate image

**Syntax**  
B = imrotate(A,angle)  
B = imrotate(A,angle,method)  
B = imrotate(A,angle,method,bbox)

**Description** B = imrotate(A,angle) rotates image A by angle degrees in a counterclockwise direction around its center point. To rotate the image clockwise, specify a negative value for angle. imrotate makes the output image B large enough to contain the entire rotated image. imrotate uses nearest neighbor interpolation, setting the values of pixels in B that are outside the rotated image to 0 (zero).

B = imrotate(A,angle,method) rotates image A, using the interpolation method specified by method. method is a text string that can have one of these values. The default value is enclosed in braces ({}).

Value	Description
{'nearest'}	Nearest-neighbor interpolation
'bilinear'	Bilinear interpolation
'bicubic'	Bicubic interpolation
	<b>Note</b> Bicubic interpolation can produce pixel values outside the original range.

B = imrotate(A,angle,method,bbox) rotates image A, where bbox specifies the size of the returned image. bbox is a text string that can have one of the following values. The default value is enclosed in braces ({}).

Value	Description
'crop'	Make output image B the same size as the input image A, cropping the rotated image to fit
{'loose'}	Make output image B large enough to contain the entire rotated image. B is generally larger than A.

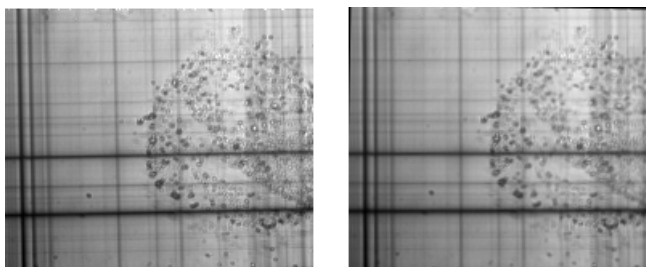
## Class Support

The input image can be numeric or logical. The output image is of the same class as the input image.

## Examples

Read a solar spectra image, stored in FITS format, and rotate the image to bring it into horizontal alignment. A rotation of -1 degree is all that is required.

```
I = fitsread('solarspectra.fits');
I = mat2gray(I);
J = imrotate(I,-1,'bilinear','crop');
figure, imshow(I)
figure, imshow(J)
```



Original Image

Rotated Image

## See Also

[imcrop](#), [imresize](#), [imtransform](#), [tformarray](#)

# imscrollpanel

---

**Purpose** Scroll panel for interactive image navigation

**Syntax** `hpanel = imscrollpanel(hparent, himage)`

**Description** `hpanel = imscrollpanel(hparent, himage)` creates a scroll panel containing the target image (the image to be navigated).. `himage` is a handle to the target image. `hparent` is a handle to the figure or uipanel that will contain the new scroll panel. `hpanel` is the handle to the scroll panel, which is a uipanel object.

A scroll panel makes an image scrollable. If the size or magnification makes an image too large to display in a figure on the screen, the scroll panel displays a portion of the image at 100% magnification (one screen pixel represents one image pixel). The scroll panel adds horizontal and vertical scroll bars to enable navigation around the image.

`imscrollpanel` changes the object hierarchy of the target image. Instead of the familiar figure->axes->image object hierarchy, `imscrollpanel` inserts several uipanel and uicontrol objects between the figure and the axes object.

## API Functions

A scroll panel contains a structure of function handles, called an API. You can use the functions in this API to manipulate the scroll panel. To retrieve this structure, use the `iptgetapi` function, as in the following example.

```
api = iptgetapi(hpanel)
```

This table lists the scroll panel API functions, in the order they appear in the structure.



Scroll Panel API Function	Description
setMagnification	<p>Sets the magnification in units of screen pixels per image pixel.</p> <pre>mag = api.getMagnification(new_mag)</pre> <p>where new_mag is a scalar magnification factor.</p>
getMagnification	<p>Returns the current magnification factor in units of screen pixels per image pixel.</p> <pre>mag = api.getMagnification()</pre> <p>Multiply mag by 100 to convert to percentage. For example if mag=2, that's equivalent to 200% magnification.</p>
setMagnificationAndCenter	<p>Changes the magnification and makes the point cx, cy appear in the center of the scroll panel. This operation is equivalent to a simultaneous zoom and recenter.</p> <pre>api.setMagnificationAndCenter(mag,cx,cy)</pre>
findFitMag	<p>Returns the magnification factor that would make the image just fit in the scroll panel.</p> <pre>mag = api.findFitMag()</pre>
setVisibleLocation	<p>Moves the target image so that the specified location is visible. Scrollbars update.</p> <pre>api.setVisibleLocation(xmin,ymin) api.setVisibleLocation([xmin ymin])</pre>

# imscrollpanel

Scroll Panel API Function	Description
<code>getVisibleLocation</code>	Returns the location of the currently visible portion of the target image.  <code>loc = api.getVisibleLocation()</code>  where <code>loc</code> is a vector <code>[xmin ymin]</code> .
<code>getVisibleImageRect</code>	Returns the current visible portion of the image.  <code>r = api.getVisibleImageRect()</code>  where <code>r</code> is a rectangle <code>[xmin ymin width height]</code> .
<code>addNewMagnificationCallback</code>	Adds the function handle <code>fcn</code> to the list of new-magnification callback functions.  <code>id = api.addNewMagnificationCallback(fcn)</code>  Whenever the scroll panel magnification changes, each function in the list is called with the syntax:  <code>fcn(mag)</code>  where <code>mag</code> is a scalar magnification factor.  The return value, <code>id</code> , is used only with <code>removeNewMagnificationCallback</code> .
<code>removeNewMagnificationCallback</code>	Removes the corresponding function from the new-magnification callback list.  <code>api.removeNewMagnificationCallback(id)</code>  where <code>id</code> is the identifier returned by <code>addNewMagnificationCallback</code> .

Scroll Panel API Function	Description
addNewLocationCallback	<p>Adds the function handle <code>fcn</code> to the list of new-location callback functions.</p> <pre>id = api.addNewLocationCallback(fcn)</pre> <p>Whenever the scroll panel location changes, each function in the list is called with the syntax:</p> <pre>fcn(loc)</pre> <p>where <code>loc</code> is <code>[xmin ymin]</code>.</p> <p>The return value, <code>id</code>, is used only with <code>removeNewLocationCallback</code>.</p>
removeNewLocationCallback	<p>Removes the corresponding function from the new-location callback list.</p> <pre>api.removeNewLocationCallback(id)</pre> <p>where <code>id</code> is the identifier returned by <code>addNewLocationCallback</code>.</p>

## Note

Scrollbar navigation as provided by `imscrollpanel` is incompatible with the default MATLAB figure navigation buttons (pan, zoom in, zoom out). The corresponding menu items and toolbar buttons should be removed in a custom GUI that includes a scrollable `uipanel` created by `imscrollpanel`.

When you run `imscrollpanel`, it appears to take over the entire figure because, by default, an `hpanel` object has `'Units'` set to `'normalized'` and `'Position'` set to `[0 0 1 1]`. If you want to see other children of `hparent` while using your new scroll panel, you must manually set the `'Position'` property of `hpanel`.

# imscrollpanel

---

## Examples

Create a scroll panel with a Magnification Box and an Overview tool.

### 1 Create a scroll panel.

```
hFig = figure('Toolbar','none',...
             'Menubar','none');
hIm = imshow('saturn.png');
hSP = imscrollpanel(hFig,hIm);
set(hSP,'Units','normalized',...
     'Position',[0 .1 1 .9])
```

### 2 Add a Magnification Box and an Overview tool.

```
hMagBox = immagbox(hFig,hIm);
pos = get(hMagBox,'Position');
set(hMagBox,'Position',[0 0 pos(3) pos(4)])
imoverview(hIm)
```

### 3 Get the scroll panel API to programmatically control the view.

```
api = iptgetapi(hSP);
```

### 4 Get the current magnification and position.

```
mag = api.getMagnification()
r = api.getVisibleImageRect()
```

### 5 View the top left corner of the image.

```
api.setVisibleLocation(0.5,0.5)
```

### 6 Change the magnification to the value that just fits.

```
api.setMagnification(api.findFitMag())
```

### 7 Zoom in to 1600% on the dark spot.

```
api.setMagnificationAndCenter(16,306,800)
```

## See Also

`immagbox`, `imoverview`, `imoverviewpanel`, `imtool`, `iptgetapi`

**Purpose**

Display image

**Syntax**

```
imshow(I)
imshow(I,[low high])
imshow(RGB)
imshow(BW)
imshow(X,map)
imshow(filename)
himage = imshow(...)
imshow(...,param1,val1,param2,val2)
```

**Description**

`imshow(I)` displays the grayscale image `I`.

`imshow(I,[low high])` displays the grayscale image `I`, specifying the display range for `I` in `[low high]`. The value `low` (and any value less than `low`) displays as black; the value `high` (and any value greater than `high`) displays as white. Values in between are displayed as intermediate shades of gray, using the default number of gray levels. If you use an empty matrix (`[]`) for `[low high]`, `imshow` uses `[min(I(:)) max(I(:))]`; that is, the minimum value in `I` is displayed as black, and the maximum value is displayed as white.

`imshow(RGB)` displays the truecolor image `RGB`.

`imshow(BW)` displays the binary image `BW`. `imshow` displays pixels with the value 0 (zero) as black and pixels with the value 1 as white.

`imshow(X,map)` displays the indexed image `X` with the colormap `map`. A color map matrix may have any number of rows, but it must have exactly 3 columns. Each row is interpreted as a color, with the first element specifying the intensity of red light, the second green, and the third blue. Color intensity can be specified on the interval 0.0 to 1.0.

`imshow(filename)` displays the image stored in the graphics file `filename`. The file must contain an image that can be read by `imread` or `dicomread`. `imshow` calls `imread` or `dicomread` to read the image from the file, but does not store the image data in the MATLAB workspace. If the file contains multiple images, the first one will be displayed. The file must be in the current directory or on the MATLAB path.

# imshow

`himage = imshow(...)` returns the handle to the image object created by `imshow`.

`imshow(...,param1,val1,param2,val2,...)` displays the image, specifying parameters and corresponding values that control various aspects of the image display. The following table lists all `imshow` parameters. Parameter names can be abbreviated, and case does not matter.

Parameter	Value
'DisplayRange'	<p>Two-element vector [LOW HIGH] that controls the display range of a grayscale image. See the <code>imshow(I,[low high])</code> syntax for more details about how to set this parameter.</p> <hr/> <p><b>Note</b> Including the parameter name is optional, except when the image is specified by a filename. The syntax <code>imshow(I,[LOW HIGH])</code> is equivalent to <code>imshow(I,'DisplayRange',[LOW HIGH])</code>. However, the 'DisplayRange' parameter must be specified when calling <code>imshow</code> with a filename, for example <code>imshow(filename,'DisplayRange',[LOW HIGH])</code>.</p> <hr/>

Parameter	Value
'InitialMagnification'	<p>A numeric scalar value, or the text string 'fit', that specifies the initial magnification used to display the image. When set to 100, imshow displays the image at 100% magnification (one screen pixel for each image pixel). When set to 'fit', imshow scales the entire image to fit in the window.</p> <p>On initial display, imshow always displays the entire image. If the magnification value is large enough that the image would be too big to display on the screen, imshow warns and displays the image at the largest magnification that fits on the screen.</p> <p>By default, the initial magnification parameter is set to the value returned by <code>iptgetpref('ImshowInitialMagnification')</code>.</p> <p>If the image is displayed in a figure with its 'WindowStyle' property set to 'docked', imshow warns and displays the image at the largest magnification that fits in the figure.</p>
'XData'	Two-element vector that establishes a nondefault spatial coordinate system by specifying the image XData. The value can have more than two elements, but only the first and last elements are actually used.
'YData'	Two-element vector that establishes a nondefault spatial coordinate system by specifying the image YData. The value can have more than two elements, but only the first and last elements are actually used.

## Class Support

A truecolor image can be `uint8`, `uint16`, `single`, or `double`. An indexed image can be `logical`, `uint8`, `single`, or `double`. A grayscale image can be `logical`, `uint8`, `int16`, `uint16`, `single`, or `double`. A binary image must be of class `logical`.

For all grayscale images having integer types, the default display range is `[intmin(class(I)) intmax(class(I))]`.

For grayscale images of class `single` or `double`, the default display range is `[0 1]`. If the data range of a `single` or `double` image is much larger or smaller than the default display range, you might need to experiment with setting the display range to see features in the image that would not be visible using the default display range.

If your image is `int16` or `single`, the `CData` in the resulting image object will be `double`. For all other classes, the `CData` matches the input image class.

## Related Toolbox Preferences

You can use the `iptsetpref` function to set several toolbox preferences that modify the behavior of `imshow`.

- `'ImshowBorder'` controls whether `imshow` displays the image with a border around it.
- `'ImshowAxesVisible'` controls whether `imshow` displays the image with the axes box and tick labels.
- `'ImshowInitialMagnification'` controls the initial magnification for image display, unless you override it in a particular call by specifying `imshow(..., 'InitialMagnification', initial_mag)`.

For more information about these preferences, see `iptsetpref`.

## Remarks

`imshow` is the toolbox's fundamental image display function, optimizing figure, axes, and image object property settings for image display. `imtool` provides all the image display capabilities of `imshow` but also provides access to several other tools for navigating and exploring images, such as the Pixel Region tool, Image Information tool, and the Adjust Contrast tool. `imtool` presents an integrated environment for displaying images and performing some common image processing tasks.



**Examples**

Display an image from a file

```
imshow('board.tif')
```

Display an indexed image

```
[X,map] = imread('trees.tif');  
imshow(X,map)
```

Display a grayscale image

```
I = imread('cameraman.tif');  
imshow(I)
```

Display an intensity image, adjusting the display range

```
h = imshow(I,[0 80]);
```

**See Also**

`imread`, `imtool`, `iptgetpref`, `iptsetpref`, `subimage`, `trueimage`, `warp`  
`image`, `imagesc` in the MATLAB Function Reference

# imsubtract

---

**Purpose** Subtract one image from another or subtract constant from image

**Syntax** `Z = imsubtract(X,Y)`

**Description** `Z = imsubtract(X,Y)` subtracts each element in array `Y` from the corresponding element in array `X` and returns the difference in the corresponding element of the output array `Z`. `X` and `Y` are real, nonsparse numeric arrays of the same size and class, or `Y` is a double scalar. The array returned, `Z`, has the same size and class as `X` unless `X` is logical, in which case `Z` is double.

If `X` is an integer array, elements of the output that exceed the range of the integer type are truncated, and fractional values are rounded.

---

**Note** On Intel architecture processors, `imsubtract` can take advantage of the Intel Performance Primitives Library (IPPL), thus accelerating its execution time. IPPL is activated only if array `X` is of class `uint8`, `int16`, or `single`.

---

**Examples** Subtract two `uint8` arrays. Note that negative results are rounded to 0.

```
X = uint8([ 255 10 75; 44 225 100]);
Y = uint8([ 50 50 50; 50 50 50 ]);
Z = imsubtract(X,Y)
Z =
```

```
    205     0    25
     0   175    50
```

Estimate and subtract the background of an image:

```
I = imread('rice.png');
background = imopen(I,strel('disk',15));
Ip = imsubtract(I,background);
imshow(Ip,[])
```

Subtract a constant value from an image:

```
I = imread('rice.png');  
Iq = imsubtract(I,50);  
figure, imshow(I), figure, imshow(Iq)
```

## See Also

`imabsdiff`, `imadd`, `imcomplement`, `imdivide`, `imlincomb`, `immultiply`,  
`ippl`

# imtool

---

**Purpose** Image Tool

**Syntax**

```
imtool
imtool(I)
imtool(I,[low high])
imtool(RGB)
imtool(BW)
imtool(X,map)
imtool(filename)
hfigure = imtool(...)
imtool close all
imtool(...,param1,val1,param2,val2,...)
```

**Description** `imtool` opens a new Image Tool in an empty state. Use the **File** menu options **Open** or **Import from Workspace** to choose an image for display.

`imtool(I)` displays the grayscale image `I`.

`imtool(I,[low high])` displays the grayscale image `I`, specifying the display range for `I` in the vector `[low high]`. The value `low` (and any value less than `low`) is displayed as black, the value `high` (and any value greater than `high`) is displayed as white. Values in between are displayed as intermediate shades of gray. `imtool` uses the default number of gray levels. If you use an empty matrix (`[]`) for `[low high]`, `imtool` uses `[min(I(:)) max(I(:))]`; the minimum value in `I` is displayed as black, and the maximum value is displayed as white.

`imtool(RGB)` displays the truecolor image `RGB`.

`imtool(BW)` displays the binary image `BW`. Pixels with the value 0 are displayed as black, and pixels with the value 1 are displayed as white.

`imtool(X,map)` displays the indexed image `X` with colormap `map`.

`imtool(filename)` displays the image contained in the graphics file `filename`. The file must contain an image that can be read by `imread` or `dicomread`. `imtool` calls `imread` or `dicomread` to read the image from the file, but the image data is not stored in the MATLAB workspace.

If the file contains multiple images, the first one is displayed. The file must be in the current directory or on the MATLAB path.

`hfigure = imtool(...)` returns `hfigure`, a handle to the figure created by `imtool`. `close(Hfigure)` closes the Image Tool.

`imtool close all` closes all image tools.

`imtool(...,param1,val1,param2,val2,...)` displays the image, specifying parameters and corresponding values that control various aspects of the image display. The following table lists all `imshow` parameters. Parameter names can be abbreviated, and case does not matter.

# imtool

Parameter	Value
'DisplayRange'	<p>Two-element vector [LOW HIGH] that controls the display range of a grayscale image. See the <code>imtool(I,[low high])</code> syntax for more details about how to set this parameter.</p> <hr/> <p><b>Note</b> Including the parameter name is optional, except when the image is specified by a filename. The syntax <code>imtool(I,[LOW HIGH])</code> is equivalent to <code>imtool(I,'DisplayRange',[LOW HIGH])</code>. However, the 'DisplayRange' parameter must be specified when calling <code>imtool</code> with a filename, as in the syntax <code>imtool(filename,'DisplayRange',[LOW HIGH])</code>.</p> <hr/>
'InitialMagnification'	<p>One of two text strings: 'adaptive' or 'fit' or a numeric scalar value that specifies the initial magnification used to display the image.</p> <p>When set to 'adaptive', the entire image is visible on initial display. If the image is too large to display on the screen, <code>imtool</code> displays the image at the largest magnification that fits on the screen.</p> <p>When set to 'fit', <code>imtool</code> scales the entire image to fit in the window.</p> <p>When set to a numeric value, the value specifies the magnification as a percentage. For example, if you specify 100, the image is displayed at 100% magnification (one screen pixel for each image pixel).</p> <p>By default, the initial magnification parameter is set to the value returned by <code>iptgetpref('ImtoolInitialMagnification')</code>.</p>

## Class Support

A truecolor image can be `uint8`, `uint16`, `single`, or `double`. An indexed image can be `logical`, `uint8`, `single`, or `double`. A grayscale image can be `uint8`, `uint16`, `int16`, `single`, or `double`. A binary image must be `logical`. A binary image is of class `logical`.

For all grayscale images having integer types, the default display range is `[intmin(class(I)) intmax(class(I))]`.

For grayscale images of class `single` or `double`, the default display range is `[0 1]`. If the data range of a `single` or `double` image is much larger or smaller than the default display range, you might need to experiment with setting the display range to see features in the image that would not be visible using the default display range.

## Related Toolbox Preferences

You can use the `'ImtoolInitialMagnification'` preference to control the initial magnification for image display. Use the `iptsetpref` function to set the this toolbox preference. You can override this toolbox preference by specifying the `'InitialMagnification'` parameter when you call `imtool`, as follows:

```
imtool(...,'InitialMagnification',initial_mag).
```

For more information about toolbox preferences, see the reference page for the `iptsetpref` function.

## Remarks

`imshow` is the toolbox's fundamental image display function, optimizing figure, axes, and image object property settings for image display. `imtool` provides all the image display capabilities of `imshow` but also provides access to several other tools for navigating and exploring images, such as the Pixel Region tool, Image Information tool, and the Adjust Contrast tool. `imtool` presents an integrated environment for displaying images and performing some common image processing tasks.

## Examples

Display an image from a file.

```
imtool('board.tif')
```

# imtool

---

Display an indexed image.

```
[X,map] = imread('trees.tif');  
imtool(X,map)
```

Display a grayscale image.

```
I = imread('cameraman.tif');  
imtool(I)
```

Display a grayscale image, adjusting the display range.

```
h = imtool(I,[0 80]);  
close(h)
```

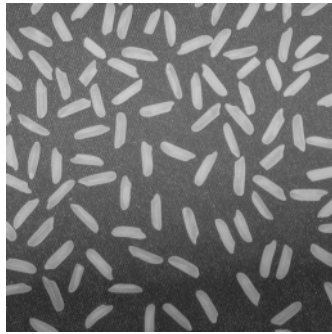
## See Also

getimage, imageinfo, imcontrast, imdisplayrange, imdistline,  
imgetfile, imoverview, impixelinfo, impixelregion, imread,  
imshow, iptgetpref, ipticondir, iptsetpref, iptwindowalign



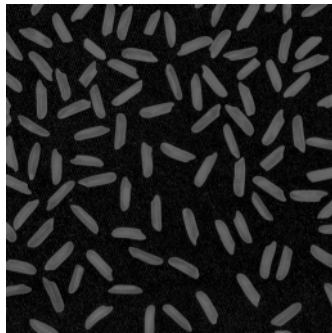
---

<b>Purpose</b>	Top-hat filtering
<b>Syntax</b>	<code>IM2 = imtophat(IM,SE)</code> <code>IM2 = imtophat(IM,NHOOD)</code>
<b>Description</b>	<p><code>IM2 = imtophat(IM,SE)</code> performs morphological top-hat filtering on the grayscale or binary input image <code>IM</code> using the structuring element <code>SE</code>, where <code>SE</code> is returned by <code>strel</code>. <code>SE</code> must be a single structuring element object, not an array containing multiple structuring element objects.</p> <p><code>IM2 = imtophat(IM,NHOOD)</code> where <code>NHOOD</code> is an array of 0's and 1's that specifies the size and shape of the structuring element, is the same as <code>imtophat(IM,strel(NHOOD))</code>.</p>
<b>Class Support</b>	<code>IM</code> can be numeric or logical and must be nonsparse. The output image <code>IM2</code> has the same class as the input image. If the input is binary (logical), the structuring element must be flat.
<b>Examples</b>	<p>You can use top-hat filtering to correct uneven illumination when the background is dark. This example uses top-hat filtering with a disk-shaped structuring element to remove the uneven background illumination from an image.</p> <ol style="list-style-type: none"><li>1 Read an image into the MATLAB workspace.</li></ol> <pre>I = imread('rice.png'); imshow(I)</pre>



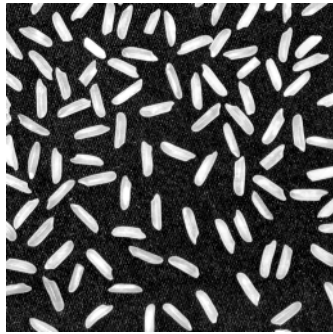
- 2 Create the structuring element and perform top-hat filtering of the image.

```
se = strel('disk',12);  
J = imtophat(I,se);  
figure, imshow(J)
```



- 3 Use `imadjust` to improve the visibility of the result.

```
K = imadjust(J);  
figure, imshow(K)
```



**See Also**

imbothat, strel

# imtransform

---

**Purpose** Apply 2-D spatial transformation to image

**Syntax**

```
B = imtransform(A,TFORM)
B = imtransform(A,TFORM,INTERP)
[B,XDATA,YDATA] = imtransform(...)
[B,XDATA,YDATA] = imtransform(...,param1,val1,param2,val2,...)
```

**Description** `B = imtransform(A,TFORM)` transforms the image `A` according to the 2-D spatial transformation defined by `TFORM`, which is a spatial transformation structure (`TFORM`) as returned by `maketform` or `cp2tform`. If `ndims(A) > 2`, such as for an RGB image, then the same 2-D transformation is automatically applied to all 2-D planes along the higher dimensions.

When you use this syntax, `imtransform` automatically shifts the origin of your output image to make as much of the transformed image visible as possible. If you are using `imtransform` to do image registration, this syntax is not likely to give you the results you expect; you might want to set `'XData'` and `'YData'` explicitly.

`B = imtransform(A,TFORM,INTERP)` specifies the form of interpolation to use. `INTERP` can have one of these values. The default value is enclosed in braces (`{}`).

Value	Description
'bicubic'	Bicubic interpolation
{'bilinear'}	Bilinear interpolation
'nearest'	Nearest-neighbor interpolation

Alternatively, `INTERP` can be a `RESAMPLER` structure returned by `makeresampler`. This option allows more control over how resampling is performed.

`[B,XDATA,YDATA] = imtransform(...)` returns the location of the output image `B` in the output X-Y space. `XDATA` and `YDATA` are

two-element vectors. The elements of XDATA specify the  $x$ -coordinates of the first and last columns of B. The elements of YDATA specify the  $y$ -coordinates of the first and last rows of B. Normally, imtransform computes XDATA and YDATA automatically so that B contains the entire transformed image A. However, you can override this automatic computation; see below.

`[B,XDATA,YDATA] = imtransform(...,param1,val1,param2,val2,...)` specifies parameters that control various aspects of the spatial transformation. This table lists all the parameters you can specify. Note that parameter names can be abbreviated and are not case sensitive.

Parameter	Description
'UData' 'VData'	Both of these parameters are two-element real vectors. 'UData' and 'VData' specify the spatial location of the image A in the 2-D input space U-V. The two elements of 'UData' give the $u$ -coordinates (horizontal) of the first and last columns of A, respectively. The two elements of 'VData' give the $v$ -coordinates (vertical) of the first and last rows of A, respectively. The default values for 'UData' and 'VData' are <code>[1 size(A,2)]</code> and <code>[1 size(A,1)]</code> , respectively.
'XData' 'YData'	Both of these parameters are two-element real vectors. 'XData' and 'YData' specify the spatial location of the output image B in the 2-D output space X-Y. The two elements of 'XData' give the $x$ -coordinates (horizontal) of the first and last columns of B, respectively. The two elements of 'YData' give the $y$ -coordinates (vertical) of the first and last rows of B, respectively. If 'XData' and 'YData' are not specified, imtransform estimates values for them that will completely contain the entire transformed output image.

# imtransform

---

Parameter	Description
'XYScale'	<p>A one- or two-element real vector. The first element of 'XYScale' specifies the width of each output pixel in X-Y space. The second element (if present) specifies the height of each output pixel. If 'XYScale' has only one element, then the same value is used for both width and height.</p> <p>If 'XYScale' is not specified but 'Size' is, then 'XYScale' is computed from 'Size', 'XData', and 'YData'. If neither 'XYScale' nor 'Size' is provided, then the scale of the input pixels is used for 'XYScale'.</p>
'Size'	<p>A two-element vector of nonnegative integers. 'Size' specifies the number of rows and columns of the output image B. For higher dimensions, the size of B is taken directly from the size of A. In other words, <math>\text{size}(B,k) = \text{size}(A,k)</math> for <math>k &gt; 2</math>. If 'Size' is not specified, then it is computed from 'XData', 'YData', and 'XYScale'.</p>

Parameter	Description
'FillValues'	<p>An array containing one or several fill values.</p> <p>Fill values are used for output pixels when the corresponding transformed location in the input image is completely outside the input image boundaries. If A is 2-D, 'FillValues' must be a scalar. However, if A's dimension is greater than two, then 'FillValues' can be an array whose size satisfies the following constraint: <code>size(fill_values,k)</code> must equal either <code>size(A,k+2)</code> or 1.</p> <p>For example, if A is a uint8 RGB image that is 200-by-200-by-3, then possibilities for 'FillValues' include</p> <p>0 Fill with black</p> <p>[0;0;0] Fill with black</p> <p>255 Fill with white</p> <p>[255;255;255] Fill with white</p> <p>[0;0;255] Fill with blue</p> <p>[255;255;0] Fill with yellow</p> <p>If A is 4-D with size 200-by-200-by-3-by-10, then 'FillValues' can be a scalar, 1-by-10, 3-by-1, or 3-by-10.</p>

## Notes

- When you do not specify the output-space location for B using 'XData' and 'YData', `imtransform` estimates them automatically using the function `findbounds`. For some commonly used transformations, such as affine or projective, for which a forward mapping is easily computable, `findbounds` is fast. For transformations that do not have a forward mapping, such as the polynomial ones computed by `cp2tform`, `findbounds` can take significantly longer. If you can specify 'XData' and 'YData' directly for such transformations, `imtransform` might run noticeably faster.

# imtransform

---

- The automatic estimate of 'XData' and 'YData' using `findbounds` is not guaranteed in all cases to completely contain all the pixels of the transformed input image.
- The output values `XDATA` and `YDATA` might not exactly equal the input 'XData' and 'YData' parameters. This can happen either because of the need for an integer number of rows and columns, or if you specify values for 'XData', 'YData', 'XYScale', and 'Size' that are not entirely consistent. In either case, the first element of `XDATA` and `YDATA` always equals the first element of 'XData' and 'YData', respectively. Only the second elements of `XDATA` and `YDATA` might be different.
- `imtransform` assumes spatial-coordinate conventions for the transformation `TFORM`. Specifically, the first dimension of the transformation is the horizontal or  $x$ -coordinate, and the second dimension is the vertical or  $y$ -coordinate. Note that this is the reverse of the array subscripting convention in MATLAB.
- `TFORM` must be a 2-D transformation to be used with `imtransform`. For arbitrary-dimensional array transformations, see `tformarray`.

## Class Support

The input image `A` can be of any nonsparse numeric class, real or complex, or it can be of class `logical`. The class of `B` is the same as the class of `A`.

## Examples

### Example 1

Apply a horizontal shear to an intensity image.

```
I = imread('cameraman.tif');
tform = maketform('affine',[1 0 0; .5 1 0; 0 0 1]);
J = imtransform(I,tform);
imshow(I), figure, imshow(J)
```

### Example 2

A projective transformation can map a square to a quadrilateral. In this example, set up an input coordinate system so that the input image fills the unit square and then transform the image into the quadrilateral



with vertices (0 0), (1 0), (1 1), (0 1) to the quadrilateral with vertices (-4 2), (-8 3), (-3 -5), (6 3). Fill with gray and use bicubic interpolation. Make the output size the same as the input size.

```
I = imread('cameraman.tif');
udata = [0 1]; vdata = [0 1]; % input coordinate system
tform = maketform('projective',[ 0 0; 1 0; 1 1; 0 1],...
                 [-4 2; -8 -3; -3 -5; 6 3]);
[B,xdata,ydata] = imtransform(I, tform, 'bicubic', ...
                              'udata', udata,...
                              'vdata', vdata,...
                              'size', size(I),...
                              'fill', 128);
subplot(1,2,1), imshow(udata,vdata,I), axis on
subplot(1,2,2), imshow(xdata,ydata,B), axis on
```

### Example 3

Register an aerial photo to an orthophoto.

Read in the aerial photo.

```
unregistered = imread('westconcordaerial.png');
figure, imshow(unregistered)
```

Read in the orthophoto.

```
figure, imshow('westconcordorthophoto.png')
```

Load control points that were previously picked.

```
load westconcordpoints
```

Create a transformation structure for a projective transformation.

```
t_concord = cp2tform(input_points,base_points,'projective');
```

Get the width and height of the orthophoto and perform the transformation.

# imtransform

---

```
info = imfinfo('westconcordorthophoto.png');  
  
registered = imtransform(unregistered,t_concord,...  
    'XData',[1 info.Width], 'YData',[1 info.Height]);  
figure, imshow(registered)
```

## See Also

checkerboard, cp2tform, imresize, imrotate, maketform,  
makesampler, tformarray

**Purpose**

Display image in the image tool

---

**Note** This function is obsolete and may be removed in future versions. Use `imtool` instead.

---

# imwrite

---

**Purpose** Write image to graphics file

**Note** `imwrite` is a function in MATLAB.

**Purpose** Convert indexed image to grayscale image

**Syntax** `I = ind2gray(X,map)`

**Description** `I = ind2gray(X,map)` converts the image `X` with colormap `map` to a grayscale 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`, `single`, or `double`. `map` is `double`. `I` is of the same class as `X`.

**Examples**

```
load trees
I = ind2gray(X,map);
imshow(X,map)
figure,imshow(I)
```

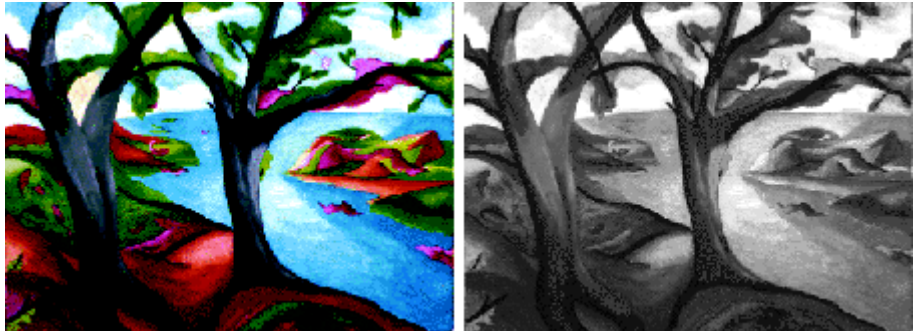


Image Courtesy of Susan Cohen

**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`, `imtool`, `rgb2ntsc`

# ind2rgb

---

<b>Purpose</b>	Convert indexed image to RGB image
<b>Syntax</b>	<code>RGB = ind2rgb(X,map)</code>
<b>Description</b>	<code>RGB = ind2rgb(X,map)</code> converts the matrix <code>X</code> and corresponding colormap <code>map</code> to RGB (truecolor) format.
<b>Class Support</b>	<code>X</code> can be of class <code>uint8</code> , <code>uint16</code> , or <code>double</code> . <code>RGB</code> is an <code>m-by-n-by-3</code> array of class <code>double</code> .
<b>See Also</b>	<code>ind2gray</code> , <code>rgb2ind</code>

**Purpose** Read metadata from Interfile file

**Syntax** `info = interfileinfo(filename)`

**Description** `info = interfileinfo(filename)` returns a structure whose fields contain information about an image in a Interfile file. `filename` is a string that specifies the name of the file. The file must be in the current directory or in a directory on the MATLAB path.

The Interfile file format was developed for the exchange of nuclear medicine data. In Interfile 3.3, metadata is stored in a header file, separate from the image data. The two files have the same name with different file extensions. The header file has the file extension `.hdr` and the image file has the file extension `.img`.

**Examples** Read metadata from an Interfile file. The following example uses an Interfile header file available at <http://www.keston.com/Phantoms/>.

```
info = interfileinfo('dyna3.hdr');
```

**See Also** `interfileread`

# interfileread

---

**Purpose** Read images in Interfile format

**Syntax** `A = interfileread(filename)`  
`A = interfileread(filename,window)`

**Description** `A = interfileread(filename)` reads the images in the first energy window of `filename` into `A`. The file must be in the current directory or in a directory on the MATLAB path.

`A = interfileread(filename,window)` reads the images in energy window of `filename` into `A`.

**Examples** Read image data from an Interfile file. The following example uses an Interfile file available at <http://www.keston.com/Phantoms/>.

```
info = interfileinfo('dyna3.hdr');
```

**See also** `interfileinfo`



---

<b>Purpose</b>	Convert integer values using lookup table
<b>Syntax</b>	<code>B = intlut(A, LUT)</code>
<b>Description</b>	<p><code>B = intlut(A, LUT)</code> converts values in array <code>A</code> based on lookup table <code>LUT</code> and returns these new values in array <code>B</code>.</p> <p>For example, if <code>A</code> is a vector whose <math>k</math>th element is equal to <code>alpha</code>, then <code>B(k)</code> is equal to the <code>LUT</code> value corresponding to <code>alpha</code>, i.e., <code>LUT(alpha+1)</code>.</p>
<b>Class Support</b>	<code>A</code> can be <code>uint8</code> , <code>uint16</code> , or <code>int16</code> . If <code>A</code> is <code>uint8</code> , <code>LUT</code> must be a <code>uint8</code> vector with 256 elements. If <code>A</code> is <code>uint16</code> or <code>int16</code> , <code>LUT</code> must be a vector with 65536 elements that has the same class as <code>A</code> . <code>B</code> has the same size and class as <code>A</code> .
<b>Examples</b>	<pre>A = uint8([1 2 3 4; 5 6 7 8; 9 10 11 12]) LUT = repmat(uint8([0 150 200 255]),1,64); B = intlut(A, LUT)</pre>
<b>See Also</b>	<code>ind2gray</code> , <code>rgb2ind</code>

**Purpose** Check for presence of Intel Performance Primitives Library (IPPL)

**Syntax** TF = ipp1  
[TF B] = ipp1

**Description** The Intel Performance Primitives Library (IPPL) provides a collection of basic functions used in signal and image processing. The IPPL takes advantage of the parallelism of the Single-Instruction, Multiple-Data (SIMD) instructions that make up the core of the MMX technology and Streaming SIMD Extensions. These instructions are available only on the Intel architecture processors. IPPL is used by some of the Image Processing Toolbox functions to accelerate their execution time.

TF = ipp1 returns true (1) if IPPL is available and false (0) otherwise.

[TF B] = ipp1 returns an additional column cell array B. Each row of B contains a string describing a specific IPPL module.

When IPPL is available, the Image Processing Toolbox image arithmetic functions (imabsdiff, imadd, imsubtract, imdivide, immultiply, and imlincomb) and the imfilter function take advantage of it. Toolbox functions that use these functions also benefit.

**Notes** IPPL is utilized only for some data types and only under specific conditions. See the help sections of the functions listed above for detailed information on when IPPL is activated.

To disable use of the IPPL, define a system environment variable called IPT\_IPPL\_OFF and set it to any value. To create an environment variable, right-click the My Computer icon on your desktop and click **Properties**. On the **Advanced** panel, click **Environment Variables**. In the System variables section, click **New**.

The ipp1 function is likely to change.

**See Also** imabsdiff, imadd, imdivide, imfilter, imlincomb, immultiply, imsubtract

<b>Purpose</b>	Add function handle to callback list
<b>Syntax</b>	<code>ID = iptaddcallback(h,callback,func_handle)</code>
<b>Description</b>	<p><code>ID = iptaddcallback(h,callback,func_handle)</code> adds the function handle <code>func_handle</code> to the list of functions to be called when the callback specified by <code>callback</code> executes. <code>callback</code> is a string specifying the name of a callback property of the Handle Graphics object specified by the handle <code>h</code>.</p> <p><code>iptaddcallback</code> returns a unique callback identifier, <code>ID</code>, that can be used with <code>iptremovecallback</code> to remove the function from the callback list.</p> <p><code>iptaddcallback</code> can be useful when you need to notify more than one tool about the same callback event for a single object.</p>
<b>Note</b>	<p>Callback functions that have already been added to an object using the <code>set</code> command continue to work after you call <code>iptaddcallback</code>. The first time you call <code>iptaddcallback</code> for a given object and <code>callback</code>, the function checks to see if a different callback function is already installed. If a callback is already installed, <code>iptaddcallback</code> replaces that callback function with the <code>iptaddcallback</code> callback processor, and then adds the preexisting callback function to the <code>iptaddcallback</code> list.</p>
<b>Examples</b>	<p>Create a figure and register two callback functions. Whenever MATLAB detects mouse motion over the figure, function handles <code>f1</code> and <code>f2</code> are called in the order in which they were added to the list.</p> <pre>h = figure; f1 = @(varargin) disp('Callback 1'); f2 = @(varargin) disp('Callback 2'); iptaddcallback(h, 'WindowButtonMotionFcn', f1); iptaddcallback(h, 'WindowButtonMotionFcn', f2);</pre>
<b>See Also</b>	<code>iptremovecallback</code>

# iptcheckconn

---

<b>Purpose</b>	Check validity of connectivity argument
<b>Syntax</b>	<code>iptcheckconn(conn, func_name, var_name, arg_pos)</code>
<b>Description</b>	<p><code>iptcheckconn(conn, func_name, var_name, arg_pos)</code> checks whether <code>conn</code> is a valid connectivity argument. If it is invalid, the function issues a formatted error message.</p> <p>A connectivity argument can be one of the following scalar values: 1, 4, 6, 8, 18, or 26. A connectivity argument can also be a 3-by-3-by- ... -by-3 array of 0's and 1s. The central element of a connectivity array must be nonzero and the array must be symmetric about its center.</p> <p><code>func_name</code> is a string that specifies the name used in the formatted error message to identify the function checking the connectivity argument.</p> <p><code>var_name</code> is a string that specifies the name used in the formatted error message to identify the argument being checked.</p> <p><code>arg_pos</code> is a positive integer that indicates the position of the argument being checked in the function argument list. <code>iptcheckconn</code> converts this value to an ordinal number and includes this information in the formatted error message.</p>
<b>Class Support</b>	<code>conn</code> must be of class <code>double</code> or <code>logical</code> and must be real and nonsparse.
<b>Examples</b>	<p>To trigger this error message, this example creates a 4-by-4 array and passes it as the connectivity argument.</p> <pre>iptcheckconn(eye(4), 'func_name', 'var_name', 2)</pre>
<b>See Also</b>	<code>iptnum2ordinal</code>

**Purpose** Check validity of handle

**Syntax** `iptcheckhandle(H,valid_types,func_name,var_name,arg_pos)`

**Description** `iptcheckhandle(H,valid_types,func_name,var_name,arg_pos)` checks the validity of the handle `H` and issues a formatted error message if the handle is invalid. `H` must be a handle to a single figure, `uipanel`, `hggroup`, `axes`, or `image` object.

`valid_types` is a cell array of strings specifying the set of Handle Graphics object types to which `H` is expected to belong. For example, if you specify `valid_types` as `{'uipanel','figure'}`, `H` can be a handle to either a `uipanel` object or a `figure` object.

`func_name` is a string that specifies the name used in the formatted error message to identify the function checking the handle.

`var_name` is a string that specifies the name used in the formatted error message to identify the argument being checked.

`arg_pos` is a positive integer that indicates the position of the argument being checked in the function argument list. `iptcheckhandle` converts this value to an ordinal number and includes this information in the formatted error message.

**Examples** To trigger the error message, create a figure that does not contain an `axes` object and then check for a valid `axes` handle.

```
fig = figure; % create figure without an axes
iptcheckhandle(fig,{'axes'},'my_function','my_variable',2)
```

The following shows the format of the error message and indicates which parts you can customize using `iptcheckhandle` arguments.

# iptcheckhandle

---

func\_name                      arg\_pos                      var\_name  
    ↓                                      ↓                                      ↓  
Function MY\_FUNCTION expected its second input argument, my\_variable,  
to be a handle of one of these types:  
  
axes                      ←————— valid\_types  
  
Instead, its type was: figure.

## See Also

iptcheckinput, iptcheckmap, iptchecknargin, iptcheckstrs,  
iptnum2ordinal

**Purpose**

Check validity of array

**Syntax**

```
iptcheckinput(A,classes,attributes,func_name,var_name,arg_pos)
```

**Description**

`iptcheckinput(A,classes,attributes,func_name,var_name,arg_pos)` checks the validity of the array `A` and issues a formatted error message if it is invalid.

`classes` is a cell array of strings specifying the set of classes to which `A` is expected to belong. For example, if you specify `classes` as `{'logical' 'cell'}`, `A` is required to be either a logical array or a cell array. The string `'numeric'` is interpreted as an abbreviation for the classes `uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`, `single`, and `double`.

`attributes` is a cell array of strings specifying the set of attributes that `A` must satisfy. For example, if `attributes` is `{'real' 'nonempty' 'finite'}`, `A` must be real and nonempty, and it must contain only finite values. The following table lists the supported attributes in alphabetical order.

<code>2d</code>	<code>nonemptyvector</code>	<code>odd</code>	<code>twod</code>
<code>column</code>	<code>nonnan</code>	<code>positive</code>	<code>vector</code>
<code>even</code>	<code>nonnegative</code>	<code>real</code>	
<code>finite</code>	<code>nonsparse</code>	<code>row</code>	
<code>integer</code>	<code>nonzero</code>	<code>scalar</code>	

`func_name` is a string that specifies the name used in the formatted error message to identify the function checking the input.

`var_name` is a string that specifies the name used in the formatted error message to identify the argument being checked.

`arg_pos` is a positive integer that indicates the position of the argument being checked in the function argument list. `iptcheckinput` converts this value to an ordinal number and includes this information in the formatted error message.

# iptcheckinput

---

## Examples

Create a three-dimensional array.

```
A = [ 1 2 3; 4 5 6 ];  
B = [ 7 8 9; 10 11 12];  
C = cat(3,A,B);  
iptcheckinput(F,{'numeric'},{'2d'},'func_name','var_name',2)
```

The following shows the format of the error message and indicates which parts you can customize using `iptcheckinput` arguments.

```
          func_name          arg_pos          var_name  
          |                  |                |  
          v                  v                v  
Function FUNC_NAME expected its second input, var_name, to be  
two-dimensional.  
          ^  
attributes
```

## See Also

`iptcheckhandle`, `iptcheckmap`, `iptchecknargin`, `iptcheckstrs`,  
`iptnum2ordinal`



**Purpose** Check validity of colormap

**Syntax** `iptcheckmap(map, func_name, var_name, arg_pos)`

**Description** `iptcheckmap(map, func_name, var_name, arg_pos)` checks the validity of the MATLAB colormap and issues an error message if it is invalid.

`func_name` is a string that specifies the name used in the formatted error message to identify the function checking the colormap.

`var_name` is a string that specifies the name used in the formatted error message to identify the argument being checked.

`arg_pos` is a positive integer that indicates the position of the argument being checked in the function argument list. `iptcheckmap` converts this value to an ordinal number and includes this information in the formatted error message.

## Examples

```
bad_map = ones(10);  
iptcheckmap(bad_map, 'func_name', 'var_name', 2)
```

The following shows the format of the error message and indicates which parts you can customize using `iptcheckmap` arguments.

```
func_name      arg_pos      var_name  
  ↓            ↓            ↓  
Function FUNC_NAME expected its second input argument, var_name,  
to be a valid colormap.  
Valid colormaps must be nonempty, double, 2-D matrices with 3  
columns.
```

## See Also

`iptcheckhandle`, `iptcheckinput`, `iptchecknargin`, `iptcheckstrs`, `iptnum2ordinal`

# iptchecknargin

---

**Purpose** Check number of input arguments

**Syntax** `iptchecknargin(low,high,num_inputs,func_name)`

**Description** `iptchecknargin(low,high,num_inputs,func_name)` checks whether `num_inputs` is in the range indicated by `low` and `high`. If not, `iptchecknargin` issues a formatted error message.

`low` should be a scalar nonnegative integer.

`high` should be a scalar nonnegative integer or `Inf`.

`func_name` is a string that specifies the name used in the formatted error message to identify the function checking the handle.

**Examples** Create a function and use `iptchecknargin` to check that the number of arguments passed to the function is within the expected range.

```
function test_function(varargin)
    iptchecknargin(1,3,nargin,mfilename);
```

Trigger the error message by executing the function at the MATLAB command line, specifying more than the expected number of arguments.

```
test_function(1,2,3,4)
```

**See Also** `iptcheckhandle`, `iptcheckinput`, `iptcheckmap`, `iptcheckstrs`, `iptnum2ordinal`

**Purpose** Check validity of option string

**Syntax** `out=iptcheckstrs(in,valid_strs,func_name,var_name,arg_pos)`

**Description** `out=iptcheckstrs(in,valid_strs,func_name,var_name,arg_pos)` checks the validity of the option string `in`. It returns the matching string in `valid_strs` in `out`. `iptcheckstrs` looks for a case-insensitive, nonambiguous match between `in` and the strings in `valid_strs`.

`valid_strs` is a cell array containing strings.

`func_name` is a string that specifies the name used in the formatted error message to identify the function checking the strings.

`var_name` is a string that specifies the name used in the formatted error message to identify the argument being checked.

`arg_pos` is a positive integer that indicates the position of the argument being checked in the function argument list. `iptcheckstrs` converts this value to an ordinal number and includes this information in the formatted error message.

**Examples** To trigger this error message, define a cell array of some text strings and pass in another string that isn't in the cell array.

```
iptcheckstrs('option3',{'option1','option2'},...  
            'func_name','var_name',2)
```

The following shows the format of the error message and indicates which parts you can customize using `iptcheckhandle` arguments.

# iptcheckstrs

---

func\_name



arg\_pos



var\_name



Function FUNC\_NAME expected its second input argument, var\_name, to match one of these strings:

option1, option2 ← valid\_strs

The input, 'option3', did not match any of the valid strings.

## See Also

iptcheckhandle, iptcheckinput, iptcheckmap, iptchecknargin, iptnum2ordinal

<b>Purpose</b>	Index of Image Processing Toolbox demos
<b>Syntax</b>	<code>iptdemos</code>
<b>Description</b>	<code>iptdemos</code> displays the HTML page that lists all the Image Processing demos. <code>iptdemos</code> displays the page in the MATLAB Help browser.

# iptgetapi

---

**Purpose** Get Application Programmer Interface (API) for handle

**Syntax** API = iptgetapi(h)

**Description** API = iptgetapi(h) returns the API structure associated with handle h if there is one. Otherwise, iptgetapi returns an empty array.

For more information about handle APIs, see the help for `immagbox`, `impositionrect`, or `imscrollpanel`.

**Examples**

```
hFig = figure('Toolbar','none',...
             'Menubar','none');
hIm = imshow('tape.png');
hSP = imscrollpanel(hFig,hIm);
api = iptgetapi(hSP);
api.setMagnification(2) % 2X = 200%
```

**See Also** `immagbox`, `imrect`, `imscrollpanel`

**Purpose** Retrieve pointer behavior from HG object

**Syntax** `pointerBehavior = iptGetPointerBehavior(h)`

**Description** `pointerBehavior = iptGetPointerBehavior(h)` returns the pointer behavior structure associated with the Handle Graphics object `h`. A pointer behavior structure contains function handles that interact with a figure's pointer manager (see `iptPointerManager`) to control what happens when the figure's mouse pointer moves over and then exits the object. See `iptSetPointerBehavior` for details.

If `h` does not contain a pointer behavior structure, `iptGetPointerBehavior` returns `[]`.

**See Also** `iptPointerManager`, `iptSetPointerBehavior`

# iptgetpref

---

**Purpose** Get value of Image Processing Toolbox preference

**Syntax**  
`prefs = iptgetpref`  
`value = iptgetpref(prefname)`

**Description** `prefs = iptgetpref` returns a structure containing all the Image Processing Toolbox preferences with their current values. Each field in the structure has the name of an Image Processing Toolbox preference. See `iptsetpref` for a list.

`value = iptgetpref(prefname)` returns the value of the Image Processing Toolbox preference specified by the string `prefname`. See `iptsetpref` for a complete list of valid preference names. Preference names are not case sensitive and can be abbreviated.

**Examples**

```
value = iptgetpref('ImshowAxesVisible')  
  
value =  
  
off
```

**See Also** `imshow`, `iptsetpref`



**Purpose** Directories containing IPT and MATLAB icons

**Syntax** [D1, D2] = ipticondir

**Description** [D1, D2] = ipticondir returns the names of the directories containing the Image Processing Toolbox icons (D1) and the MATLAB icons (D2).

**Examples**

```
[iptdir, MATLABdir] = ipticondir  
dir(iptdir)
```

**See Also** imtool

# iptnum2ordinal

---

**Purpose** Convert positive integer to ordinal string

**Syntax** `string = iptnum2ordinal(number)`

**Description** `string = iptnum2ordinal(number)` converts the positive integer number to the ordinal text string `string`.

**Examples** The following example returns the string 'fourth'.

```
str = iptnum2ordinal(4)
```

The following example returns the string '23rd'.

```
str = iptnum2ordinal(23)
```

**Purpose** Create pointer manager in figure

**Syntax**

```
iptPointerManager(hFigure)
iptPointerManager(hFigure, 'disable')
iptPointerManager(hFigure, 'enable')
```

**Description** `iptPointerManager(hFigure)` creates a pointer manager in the specified figure. The pointer manager controls pointer behavior for any Handle Graphics objects in the figure that contain pointer behavior structures. Use `iptSetPointerBehavior` to associate a pointer behavior structure with a particular object to define specific actions that occur when the mouse pointer moves over and then leaves the object. See `iptSetPointerBehavior` for more information.

`iptPointerManager(hFigure, 'disable')` disables the figure's pointer manager.

`iptPointerManager(hFigure, 'enable')` enables and updates the figure's pointer manager.

If the figure already contains a pointer manager, `iptPointerManager(hFigure)` does not create a new one. It has the same effect as `iptPointerManager(hFigure, 'enable')`.

**Examples** Plot a line. Create a pointer manager in the figure. Then, associate a pointer behavior structure with the line object in the figure that changes the mouse pointer into a fleur whenever the pointer is over it.

```
h = plot(1:10);
iptPointerManager(gcf);
enterFcn = @(hFigure, currentPoint)...
    set(hFigure, 'Pointer', 'fleur');
iptSetPointerBehavior(h, enterFcn);
```

**See Also** `iptGetPointerBehavior`, `iptSetPointerBehavior`

# iptremovecallback

---

**Purpose** Delete function handle from callback list

**Syntax** `iptremovecallback(h,callback,ID)`

**Description** `iptremovecallback(h,callback,ID)` deletes a callback from the list of callbacks created by `imaddcallback` for the object with handle `h` and the associated callback string `callback`. `ID` is the identifier of the callback to be deleted. This `ID` is returned by `iptaddcallback` when you add the function handle to the callback list.

**Examples** Register three callbacks and try them interactively.

```
h = figure;
f1 = @(varargin) disp('Callback 1');
f2 = @(varargin) disp('Callback 2');
f3 = @(varargin) disp('Callback 3');
id1 = iptaddcallback(h, 'WindowButtonMotionFcn', f1);
id2 = iptaddcallback(h, 'WindowButtonMotionFcn', f2);
id3 = iptaddcallback(h, 'WindowButtonMotionFcn', f3);
```

Remove one of the callbacks and then move the mouse over the figure again. Whenever MATLAB detects mouse motion over the figure, function handles `f1` and `f3` are called in that order.

```
iptremovecallback(h, 'WindowButtonMotionFcn', id2);
```

**See Also** `iptaddcallback`

**Purpose** Store pointer behavior structure in Handle Graphics object

**Syntax**

```
iptSetPointerBehavior(h, pointerBehavior)
iptSetPointerBehavior(h, [])
iptSetPointerBehavior(h, enterFcn)
```

**Description** `iptSetPointerBehavior(h, pointerBehavior)` stores the specified pointer behavior structure in the specified Handle Graphics object, `h`. If `h` is an array of objects, `iptSetPointerBehavior` stores the same structure in each object.

When used with a figure's pointer manager (see `iptPointerManager`), a pointer behavior structure controls what happens when the figure's mouse pointer moves over and then exits an object in the figure. For details about this structure, see "Pointer Behavior Structure" on page 17-445.

`iptSetPointerBehavior(h, [])` clears the pointer behavior from the Handle Graphics object or objects.

`iptSetPointerBehavior(h, enterFcn)` creates a pointer behavior structure for you, setting the `enterFcn` field to the function handle specified, and setting the `traverseFcn` and `exitFcn` fields to `[]`. See "Pointer Behavior Structure" on page 17-445 for details about these fields. This syntax is provided as a convenience because, for most common uses, only the `enterFcn` is necessary.

## Pointer Behavior Structure

A pointer behavior structure contains three fields: `enterFcn`, `traverseFcn`, and `exitFcn`. You set the value of these fields to function handles and use the `iptSetPointerBehavior` function to associate this structure with an HG object in a figure. If the figure has a pointer manager installed, the pointer manager calls these functions when the following events occur. If you set a field to `[]`, no action is taken.

# iptSetPointerBehavior

---

Function Handle	When Called
enterFcn	Called when the mouse pointer moves over the object.
traverseFcn	Called once when the mouse pointer moves over the object, and called again each time the mouse moves within the object.
exitFcn	Called when the mouse pointer leaves the object.

When the pointer manager calls the functions you create, it passes two arguments: a handle to the figure and the current position of the pointer.

## Examples

### Example 1

Change the mouse pointer to a fleur whenever it is over a specific object and restore the original pointer when the mouse pointer moves off the object. The example creates a patch object and associates a pointer behavior structure with the object. Because this scenario requires only an enterFcn, the example uses the `iptSetPointerBehavior(n, enterFcn)` syntax. The example then creates a pointer manager in the figure. Note that the pointer manager takes care of restoring the original figure pointer.

```
hPatch = patch([.25 .75 .75 .25 .25],...  
               [.25 .25 .75 .75 .25], 'r');  
xlim([0 1]);  
ylim([0 1]);  
  
enterFcn = @(figHandle, currentPoint)...  
           set(figHandle, 'Pointer', 'fleur');  
iptSetPointerBehavior(hPatch, enterFcn);  
iptPointerManager(gcf);
```

## Example 2

Change the appearance of the mouse pointer, depending on where it is within the object. This example sets up the pointer behavior structure, setting the `enterFcn` and `exitFcn` fields to `[]`, and setting `traverseFcn` to a function named `ipexOverMe` that handles the position-specific behavior. `ipexOverMe` is an example function (in `\toolbox\images\imdemos`) that varies the mouse pointer depending on the location of the mouse within the object. For more information, edit `ipexOverMe`.

```
hPatch = patch([.25 .75 .75 .25 .25],...
               [.25 .25 .75 .75 .25], 'r');
xlim([0 1])
ylim([0 1])

pointerBehavior.enterFcn = [];
pointerBehavior.exitFcn = [];
pointerBehavior.traverseFcn = @ipexOverMe;

iptSetPointerBehavior(hPatch, pointerBehavior);
iptPointerManager(gcf);
```

## Example 3

Change the figure's title when the mouse pointer is over the object. In this scenario, `enterFcn` and `exitFcn` are used to achieve the desired side effect, and `traverseFcn` is `[]`.

```
hPatch = patch([.25 .75 .75 .25 .25],...
               [.25 .25 .75 .75 .25], 'r');
xlim([0 1])
ylim([0 1])

pointerBehavior.enterFcn = ...
    @(figHandle, currentPoint)...
    set(figHandle, 'Name', 'Over patch');
pointerBehavior.exitFcn = ...
    @(figHandle, currentPoint) set(figHandle, 'Name', '');
```

# iptSetPointerBehavior

---

```
pointerBehavior.traverseFcn = [];  
  
iptSetPointerBehavior(hPatch, pointerBehavior);  
iptPointerManager(gcf)
```

## **See Also**

iptGetPointerBehavior, iptPointerManager



**Purpose** Set Image Processing Toolbox preferences or display valid values

**Syntax** `iptsetpref(prefname)`  
`iptsetpref(prefname,value)`

**Description** `iptsetpref(prefname)` displays the valid values for the Image Processing Toolbox preference specified by `prefname`.

`iptsetpref(prefname,value)` sets the Image Processing Toolbox preference specified by the string `prefname` to the value specified by `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. The default value is enclosed in braces (`{}`).

Preference Name	Description
'ImshowBorder'	<p>Controls whether <code>imshow</code> includes a border around the image in the figure window. Possible values:</p> <p>{ 'loose' } — Include a border between the image and the edges of the figure window, thus leaving room for axes labels, titles, etc.</p> <p>'tight' — Adjust the figure size so that the image entirely fills the figure.</p> <hr/> <p><b>Note</b> There can 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.</p> <hr/>

# iptsetpref

---

Preference Name	Description
'ImshowAxesVisible'	Controls whether imshow displays images with the axes box and tick labels. Possible values: 'on' — Include axes box and tick labels. { 'off' } — Do not include axes box and tick labels.

Preference Name	Description
'ImshowInitialMagnification'	<p>Controls the initial magnification of the image displayed by <code>imshow</code>. Possible values:</p> <p>Any numeric value — <code>imshow</code> interprets numeric values as a percentage. The default value is 100. One hundred percent magnification means that there should be one screen pixel for every image pixel.</p> <p>'fit' — Scale the image so that it fits into the window in its entirety.</p> <p>You can override this preference by specifying the 'InitialMagnification' parameter when you call <code>imshow</code>, or by calling the <code>truesize</code> function manually after displaying the image.</p>
'ImtoolInitialMagnification'	<p>Controls the initial magnification of the image displayed by <code>imtool</code>. Possible values:</p> <p>{ 'adaptive' } — Display the entire image. If the image is too large to display on the screen at 100% magnification, display the image at the largest magnification that fits on the screen. This is the default.</p> <p>Any numeric value — Specify the magnification as a percentage. One hundred percent magnification means that there should be one screen pixel for every image pixel.</p> <p>'fit' — Scale the image so that it fits into the window in its entirety.</p> <p>You can override this preference by specifying the 'InitialMagnification' parameter when you call <code>imtool</code>.</p>

# iptsetpref

---

## Examples

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

## See Also

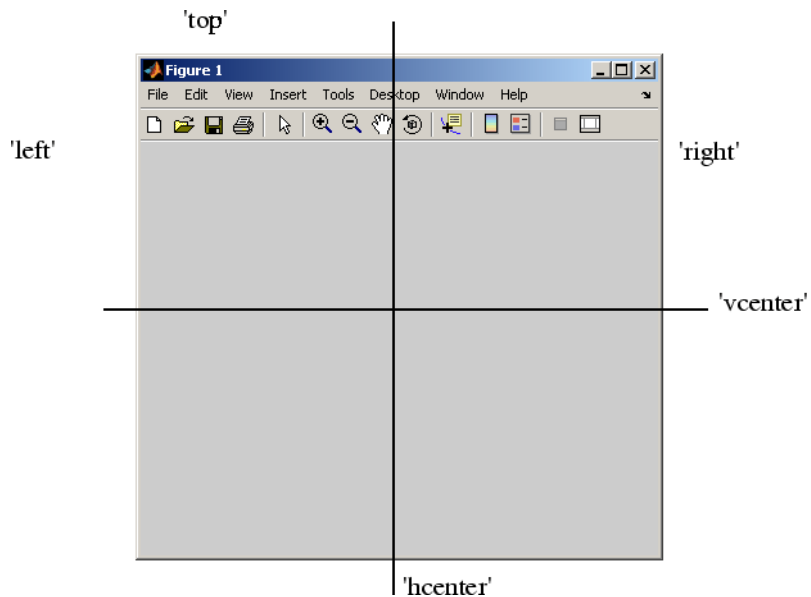
imshow, imtool, iptgetpref, truesize  
axis in the MATLAB Function Reference

**Purpose** Align figure windows

**Syntax** `iptwindowalign(fixed_fig, fixed_fig_edge, ...  
moving_fig, moving_fig_edge)`

**Description** `iptwindowalign(fixed_fig, fixed_fig_edge, moving_fig, ...  
moving_fig_edge)` moves the figure `moving_fig` to align it with the figure `fixed_fig`. `moving_fig` and `fixed_fig` are handles to figure objects.

`fixed_fig_edge` and `moving_fig_edge` describe the alignment of the figures in relation to their edges and can take any of the following values: 'left', 'right', 'hcenter', 'top', 'bottom', or 'vcenter'. 'hcenter' means center horizontally and 'vcenter' means center vertically. The following figure shows these alignments.



# iptwindowalign

---

## Notes

The two specified locations must be consistent in terms of their direction. For example, you cannot specify 'left' for `fixed_fig_edge` and 'bottom' for `moving_fig_edge`.

`iptwindowalign` constrains the position adjustment of `moving_fig` to keep it entirely visible on the screen.

## Examples

Create two figures: `fig1` and `fig2`.

```
fig1 = figure;  
fig2 = figure;
```

Move `fig2` so its left edge is aligned with the right edge of `fig1`.

```
iptwindowalign(fig1, 'right', fig2, 'left');
```

Move `fig2` so its top edge is aligned with `fig1`'s bottom edge, and then move it so the two figures are vertically centered.

```
iptwindowalign(fig1, 'bottom', fig2, 'top');  
iptwindowalign(fig1, 'vcenter', fig2, 'vcenter')
```

## See Also

`imtool`

---

<b>Purpose</b>	Inverse Radon transform
<b>Syntax</b>	<pre>I = iradon(R,theta) I = iradon(R,theta,interp,filter,frequency_scaling,output_size) [I,H] = iradon(...)</pre>
<b>Description</b>	<p><code>I = iradon(R,theta)</code> reconstructs the image <code>I</code> from projection data in the two-dimensional array <code>R</code>. The columns of <code>R</code> are parallel beam projection data. <code>iradon</code> assumes that the center of rotation is the center point of the projections, which is defined as <code>ceil(size(R,1)/2)</code>.</p> <p><code>theta</code> describes the angles (in degrees) at which the projections were taken. It can be either a vector containing the angles or a scalar specifying <code>D_theta</code>, the incremental angle between projections. If <code>theta</code> is a vector, it must contain angles with equal spacing between them. If <code>theta</code> is a scalar specifying <code>D_theta</code>, the projections were taken at angles <code>theta = m*D_theta</code>, where <code>m = 0,1,2,...,size(R,2)-1</code>. If the input is the empty matrix (<code>[]</code>), <code>D_theta</code> defaults to <code>180/size(R,2)</code>.</p> <p><code>iradon</code> uses the filtered back-projection 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.</p> <p><code>I = iradon(P,theta,interp,filter,frequency_scaling,output_size)</code> specifies parameters to use in the inverse Radon transform. You can specify any combination of the last four arguments. <code>iradon</code> uses default values for any of these arguments that you omit.</p> <p><code>interp</code> specifies the type of interpolation to use in the back projection. The available options are listed in order of increasing accuracy and computational complexity. The default value is enclosed in braces (<code>{}</code>).</p>

Value	Description
'nearest'	Nearest-neighbor interpolation
{'linear'}	Linear interpolation
'spline'	Spline interpolation

`filter` specifies the filter to use for frequency domain filtering. `filter` can be any of the strings that specify standard filters. The default value is enclosed in braces ({}).

Value	Description
{'Ram-Lak'}	Cropped Ram-Lak or ramp filter. The frequency response of this filter is $ f $ . Because this filter is sensitive to noise in the projections, one of the filters listed below might be preferable. These filters multiply the Ram-Lak filter by a window that deemphasizes high frequencies.
'Shepp-Logan'	Multiplies the Ram-Lak filter by a sinc function
'Cosine'	Multiplies the Ram-Lak filter by a cosine function
'Hamming'	Multiplies the Ram-Lak filter by a Hamming window
'Hann'	Multiplies the Ram-Lak filter by a Hann window

`frequency_scaling` is a scalar in the range (0,1] that modifies the filter by rescaling its frequency axis. The default is 1. If `frequency_scaling` is less than 1, the filter is compressed to fit into the frequency range  $[0, \text{frequency\_scaling}]$ , in normalized frequencies; all frequencies above `frequency_scaling` are set to 0.

`output_size` is a scalar that specifies the number of rows and columns in the reconstructed image. If `output_size` is not specified, the size is determined from the length of the projections.

$$n = 2 * \text{floor}(\text{size}(R,1) / (2 * \text{sqrt}(2)))$$



If you specify `output_size`, `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 might 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

`R` can be `double` or `single`. All other numeric input arguments must be of class `double`. `I` has the same class as `R`. `H` is `double`.

## Examples

```
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 back projection 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

`fan2para`, `fanbeam`, `ifanbeam`, `para2fan`, `phantom`, `radon`

## References

[1] Kak, A. C., and M. Slaney, *Principles of Computerized Tomographic Imaging*, New York, NY, IEEE Press, 1988.

# isbw

---

**Purpose** True for a binary image

**Syntax** `flag = isbw(A)`

---

**Note** This function is obsolete and may be removed in future versions. Use `islogical` instead.

---

**Description** `flag = isbw(A)` returns 1 if A is a binary image and 0 otherwise. The input image A is considered to be a binary image if it is a nonsparse logical array.

**Class Support** The input image A can be any MATLAB array.

**See Also** `isind`, `isgray`, `isrgb`

<b>Purpose</b>	True for flat structuring element
<b>Syntax</b>	TF = isflat(SE)
<b>Description</b>	TF = isflat(SE) returns true (1) if the structuring element SE is flat; otherwise it returns false (0). If SE is an array of STREL objects, then TF is the same size as SE.
<b>Class Support</b>	SE is a STREL object. TF is a double-precision value.
<b>See Also</b>	strel

# isgray

---

**Purpose** True for grayscale image

**Syntax** `flag = isgray(A)`

---

**Note** This function is obsolete and may be removed in future versions.

---

**Description** `flag = isgray(A)` returns 1 if A is a grayscale intensity image and 0 otherwise.

`isgray` uses these criteria to decide whether 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 grayscale images returns 0, not 1.

---

**Class Support** The input image A can be of class `logical`, `uint8`, `uint16`, or `double`.

**See Also** `isbw`, `isind`, `isrgb`

**Purpose** True for valid ICC color profile

**Syntax** TF = isicc(P)

**Description** TF = isicc(P) returns True if structure P is a valid ICC color profile; otherwise False.

isicc checks if P has a complete set of the tags required for an ICC profile. P must contain a Header field, which in turn must contain a Version field and a DeviceClass field. These fields, and others, are used to determine the set of required tags according to the ICC Profile Specification, either Version 2 (ICC.1:2001-04) or Version 4 (ICC.1:2001-12), which are available at [www.color.org](http://www.color.org). The set of required tags is given in Section 6.3 in either version.

**Examples** Read in a profile and test its validity.

```
P = iccread('sRGB.icm');
```

```
TF = isicc(P)
```

```
TF =
```

```
1
```

Create a MATLAB structure and test its validity.

```
S.name = 'Any Student';
```

```
S.score = 83;
```

```
S.grade = 'B+'
```

```
TF = isicc(S)
```

```
TF =
```

```
0
```

**See Also** applycform, iccread, iccwrite, makecform

# isind

---

**Purpose** True for indexed image

**Syntax** `flag = isind(A)`

---

**Note** This function is obsolete and may be removed in future versions.

---

**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` or `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 `logical`, `uint8`, `uint16`, or `double`.

**See Also** `isbw`, `isgray`, `isrgb`

**Purpose** True for RGB image

**Syntax** `flag = isrgb(A)`

---

**Note** This function is obsolete and may be removed in future versions.

---

**Description** `flag = isrgb(A)` returns 1 if A is an RGB truecolor image and 0 otherwise.

`isrgb` uses these criteria to determine whether 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 `logical`, `uint8`, `uint16`, or `double`.

**See Also** `isbw`, `isgray`, `isind`

# lab2double

**Purpose** Convert  $L^*a^*b^*$  data to double

**Syntax** `labd = lab2double(lab)`

**Description** `labd = lab2double(lab)` converts an M-by-3 or M-by-N-by-3 array of  $L^*a^*b^*$  color values to class double. The output array `labd` has the same size as `lab`.

The Image Processing Toolbox follows the convention that double-precision  $L^*a^*b^*$  arrays contain 1976 CIE  $L^*a^*b^*$  values.  $L^*a^*b^*$  arrays that are `uint8` or `uint16` follow the convention in the ICC profile specification (ICC.1:2001-4, [www.color.org](http://www.color.org)) for representing  $L^*a^*b^*$  values as unsigned 8-bit or 16-bit integers. The ICC encoding convention is illustrated by these tables.

Value ( $L^*$ )	uint8 Value	uint16 Value
0.0	0	0
100.0	255	65280
100.0 + (25500/65280)	None	65535

Value ( $a^*$ or $b^*$ )	uint8 Value	uint16 Value
-128.0	0	0
0.0	128	32768
127.0	255	65280
127.0 + (255/256)	None	65535

**Class Support** `lab` is a `uint8`, `uint16`, or `double` array that must be real and nonsparse. `labd` is `double`.

**Examples** Convert full intensity neutral color (white) from `uint8` to `double`.

```
lab2double(uint8([255 128 128]))
```



```
ans =  
    100     0     0
```

## See Also

`applycform`, `lab2uint8`, `lab2uint16`, `makecform`, `whitepoint`,  
`xyz2double`, `xyz2uint16`

For a full list of the toolbox color space conversion functions, see “Color Space Conversions” on page 16-21.

# lab2uint16

**Purpose** Convert  $L^*a^*b^*$  data to uint16

**Syntax** `lab16 = lab2uint16(lab)`

**Description** `lab16 = lab2uint16(lab)` converts an M-by-3 or M-by-N-by-3 array of  $L^*a^*b^*$  color values to uint16. `lab16` has the same size as `lab`.

The Image Processing Toolbox follows the convention that double-precision  $L^*a^*b^*$  arrays contain 1976 CIE  $L^*a^*b^*$  values.  $L^*a^*b^*$  arrays that are uint8 or uint16 follow the convention in the ICC profile specification (ICC.1:2001-4, [www.color.org](http://www.color.org)) for representing  $L^*a^*b^*$  values as unsigned 8-bit or 16-bit integers. The ICC encoding convention is illustrated by these tables.

Value ( $L^*$ )	uint8 Value	uint16 Value
0.0	0	0
100.0	255	65280
$100.0 + (25500/65280)$	None	65535

Value ( $a^*$ or $b^*$ )	uint8 Value	uint16 Value
-128.0	0	0
0.0	128	32768
127.0	255	65280
$127.0 + (255/256)$	None	65535

**Class Support** `lab` can be a uint8, uint16, or double array that must be real and nonsparse. `lab16` is of class uint16.

**Examples** Convert full intensity neutral color (white) from double to uint16.

```
lab2uint16(100 0 0)
ans =
```

65280 32768 32768

## See Also

applycform, lab2double, lab2uint8, makecform, whitepoint,  
xyz2double, xyz2uint16

For a full list of the toolbox color space conversion functions, see “Color Space Conversions” on page 16-21.

# lab2uint8

**Purpose** Convert  $L^*a^*b^*$  data to uint8

**Syntax** `lab8 = lab2uint8(lab)`

**Description** `lab8 = lab2uint8(lab)` converts an M-by-3 or M-by-N-by-3 array of  $L^*a^*b^*$  color values to uint8. `lab8` has the same size as `lab`.

The Image Processing Toolbox follows the convention that double-precision  $L^*a^*b^*$  arrays contain 1976 CIE  $L^*a^*b^*$  values.  $L^*a^*b^*$  arrays that are uint8 or uint16 follow the convention in the ICC profile specification (ICC.1:2001-4, [www.color.org](http://www.color.org)) for representing  $L^*a^*b^*$  values as unsigned 8-bit or 16-bit integers. The ICC encoding convention is illustrated by these tables.

Value ( $L^*$ )	uint8 Value	uint16 Value
0.0	0	0
100.0	255	65280
100.0 + (25500/65280)	None	65535

Value ( $a^*$ or $b^*$ )	uint8 Value	uint16 Value
-128.0	0	0
0.0	128	32768
127.0	255	65280
127.0 + (255/256)	None	65535

**Class Support** `lab` is a uint8, uint16, or double array that must be real and nonsparse. `lab8` is uint8.

**Examples** Convert full intensity neutral color (white) from double to uint8.

```
lab2uint8([100 0 0])
ans =
```

255 128 128

## See Also

`applycform`, `lab2double`, `lab2uint16`, `makecform`, `whitepoint`,  
`xyz2double`, `xyz2uint16`

For a full list of the toolbox color space conversion functions, see “Color Space Conversions” on page 16-21.

# label2rgb

---

**Purpose** Convert label matrix into RGB image

**Syntax**

```
RGB = label2rgb(L)
RGB = label2rgb(L,map)
RGB = label2rgb(L,map,zerocolor)
RGB = label2rgb(L,map,zerocolor,order)
```

**Description** `RGB = label2rgb(L)` converts a label matrix `L`, such as those returned by `bwlabel` or `watershed`, into an RGB color image for the purpose of visualizing the labeled regions. The `label2rgb` function determines the color to assign to each object based on the number of objects in the label matrix and range of colors in the colormap. The `label2rgb` function picks colors from the entire range.

`RGB = label2rgb(L,map)` defines the colormap `map` to be used in the RGB image. `map` can have any of the following values:

- $n$ -by-3 colormap matrix
- String containing the name of a MATLAB colormap function, such as 'jet' or 'gray' (See `colormap` for a list of supported colormaps.)
- Function handle of a colormap function, such as `@jet` or `@gray`

If you do not specify `map`, the default value is 'jet'.

`RGB = label2rgb(L,map,zerocolor)` defines the RGB color of the elements labeled 0 (zero) in the input label matrix `L`. As the value of `zerocolor`, specify an RGB triple or one of the strings listed in this table.

Value	Color
'b'	Blue
'c'	Cyan
'g'	Green
'k'	Black

Value	Color
'm'	Magenta
'r'	Red
'w'	White
'y'	Yellow

If you do not specify `zerocolor`, the default value for zero-labeled elements is `[1 1 1]` (white).

`RGB = label2rgb(L, map, zerocolor, order)` controls how `label2rgb` assigns colormap colors to regions in the label matrix. If `order` is `'noshuffle'` (the default), `label2rgb` assigns colormap colors to label matrix regions in numerical order. If `order` is `'shuffle'`, `label2rgb` assigns colormap colors pseudorandomly.

## Class Support

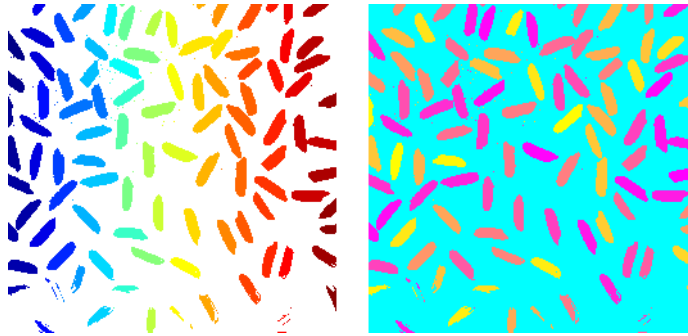
The input label matrix `L` can have any nonsparse, numeric class. It must contain finite, nonnegative integers. The output of `label2rgb` is of class `uint8`.

## Examples

```
I = imread('rice.png');  
figure, imshow(I), title('original image')  
BW = im2bw(I, graythresh(I));  
L = bwlabel(BW);  
RGB = label2rgb(L);  
RGB2 = label2rgb(L, 'spring', 'c', 'shuffle');  
imshow(RGB), figure, imshow(RGB2)
```

# label2rgb

---



## See Also

`bwlabel`, `colormap`, `ismember`, `watershed`



**Purpose** Create color transformation structure

**Syntax**

```
C = makecform(type)
C = makecform(type, 'whitepoint', WP)
C = makecform('icc', src_profile, dest_profile)
C = makecform('icc', src_profile, dest_profile,
    'SourceRenderingIntent', src_intent, 'DestRenderingIntent',
    dest_intent)
C = makecform('clut', profile, LUTtype)
C = makecform('mattrc', MatTrc, 'Direction', direction)
```

**Description**

C = makecform(type) creates the color transformation structure C that defines the color space conversion specified by type. To perform the transformation, pass the color transformation structure as an argument to the applycform function.

The type argument specifies one of the conversions listed in the following table. makecform supports conversions between members of the family of device-independent color spaces defined by the CIE, *Commission Internationale de l'Éclairage* (International Commission on Illumination). In addition, makecform supports conversions to and from the *sRGB* standard. For a list of the abbreviations used by the Image Processing Toolbox for each color space, see the Remarks section of this reference page.

Type	Description
'lab2lch'	Convert from $L^*a^*b^*$ to the $L^*ch$ color space.
'lab2srgb' <sup>1</sup>	Convert from $L^*a^*b^*$ to the <i>srgb</i> color space.
'lab2xyz' <sup>1</sup>	Convert from $L^*a^*b^*$ to the $XYZ$ color space.
'lch2lab'	Convert from $L^*ch$ to the $L^*a^*b^*$ color space.
'srgb2lab' <sup>1</sup>	Convert from <i>srgb</i> to the $L^*a^*b^*$ color space.
'srgb2xyz'	Convert from <i>srgb</i> to the $XYZ$ color space.
'upvp12xyz'	Convert from $u'v'L$ to the $XYZ$ color space.

# makecform

Type	Description
'uvl2xyz'	Convert from $uvL$ to the $XYZ$ color space.
'xyl2xyz'	Convert from $xyY$ to the $XYZ$ color space.
'xyz2lab' <sup>1</sup>	Convert from $XYZ$ to the $L^*a^*b^*$ color space.
'xyz2srgb'	Convert from $XYZ$ to the $srgb$ color space.
'xyz2upvpl'	Convert from $XYZ$ to the $u^*v^*L$ color space.
'xyz2uvl'	Convert from $XYZ$ to the $uvL$ color space.
'xyz2xyl'	Convert from $XYZ$ to the $xyY$ color space.

<sup>1</sup>For the 'xyz2lab', 'lab2xyz', 'srgb2lab', and 'lab2srgb' transforms, you can optionally specify the value of the reference illuminant, known as the white point. Use the syntax

```
C = makecform(type, 'WhitePoint', WP)
```

where WP is a 1-by-3 vector of XYZ values scaled so that Y = 1. The default is the CIE illuminant D50 as specified in the International Color Consortium specification ICC.1:2001-04 and ICC.1:2001-12. You can use the `whitepoint` function to create the WP vector.

`C = makecform('icc', src_profile, dest_profile)` creates a color transform based on two ICC profiles. `src_profile` and `dest_profile` are ICC profile structures returned by `iccread`.

`C = makecform('icc', src_profile, dest_profile, 'SourceRenderingIntent', src_intent, 'DestRenderingIntent', DEST_INTENT)` creates a color transform based on two ICC color profiles, `src_profile` and `dest_profile`, specifying rendering intent arguments for the source, `src_intent`, and the destination, `dest_intent`, profiles.

Rendering intents specify the style of reproduction that should be used when these profiles are combined. For most devices, the range of reproducible colors is much smaller than the range of colors represented by the PCS. Rendering intents define gamut mapping techniques.

Possible values for these rendering intents are listed below. Each rendering intent has distinct aesthetic and color-accuracy tradeoffs.

Value	Description
'AbsoluteColorimetric'	Maps all out-of-gamut colors to the nearest gamut surface while maintaining the relationship of all in-gamut colors. This absolute rendering contains color data that is relative to a perfectly reflecting diffuser.
'Perceptual' (default)	Employs vendor-specific gamut mapping techniques for optimizing the range of producible colors of a given device. The objective is to provide the most aesthetically pleasing result even though the relationship of the in-gamut colors might not be maintained. This media-relative rendering contains color data that is relative to the device's white point.
'RelativeColorimetric'	Maps all out-of-gamut colors to the nearest gamut surface while maintaining the relationship of all in-gamut colors. This media-relative rendering contains color data that is relative to the device's white point.
'Saturation'	Employs vendor-specific gamut mapping techniques for maximizing the saturation of device colors. This rendering is generally used for simple business graphics such as bar graphs and pie charts. This media-relative rendering contains color data that is relative to the device's white point.

`C = makecform('clut', profile, LUTtype)` creates the color transformation structure `C` based on a color lookup table (CLUT) contained in an ICC color profile. `profile` is an ICC profile structure returned by `iccread`. `LUTtype` specifies which `clut` in the profile structure is to be used. Each `LUTtype`, listed in the table below, contains the components of an 8-bit or 16-bit LUTtag that performs a transformation between device colors and PCS colors using a particular rendering.

# makecform

LUT Type	Description
'AToB0'	Device to PCS: perceptual rendering intent
'AToB1'	Device to PCS: media-relative colorimetric rendering intent
'AToB2'	Device to PCS: saturation rendering intent
'AToB3'	Device to PCS: ICC-absolute rendering intent
'BToA0'	PCS to device: perceptual rendering intent
'BToA1'	PCS to device: media-relative colorimetric rendering intent
'BToA2'	PCS to device: saturation rendering intent
'BToA3'	PCS to device: ICC-absolute rendering intent
'Gamut'	Determines which PCS colors are out of gamut for a given device
'Preview0'	PCS colors to the PCS colors available for soft proofing using the perceptual rendering
'Preview1'	PCS colors available for soft proofing using the media-relative colorimetric rendering.
'Preview2'	PCS colors to the PCS colors available for soft proofing using the saturation rendering.

`C = makecform('mattrc', MatTrc, 'Direction', direction)` creates the color transformation structure `C` based on a Matrix/Tone Reproduction Curve (MatTRC) model, contained in an ICC color profile. `direction` can be either 'forward' or 'inverse' and specifies whether the MatTRC is to be applied in the forward or inverse direction. For more information, see section 6.3.1.2 of the International Color Consortium specification ICC.1:2001-04 ([www.color.org](http://www.color.org)).

## Remarks

The Image Processing Toolbox uses the following abbreviations to represent color spaces.

Abbreviation	Description
xyz	1931 CIE XYZ tristimulus values (2° observer)
xy1	1931 CIE xyY chromaticity values (2° observer)
uv1	1960 CIE uvL values
upvp1	1976 CIE the $u'v'L$ values
lab	1976 CIE $L^*a^*b^*$ values
lch	Polar transformation of CIE $L^*a^*b^*$ values, where $c$ = chroma and $h$ = hue
srgb	Standard computer monitor RGB values, (IEC 61966-2-1)

## Examples

Convert RGB image to  $L^*a^*b^*$ , assuming input image is uint8.

```
rgb = imread('peppers.png');
cform = makecform('srgb2lab');
lab = applycform(rgb,cform);
```

Convert from a non-standard RGB color profile to the device-independent XYZ profile connection space. Note that the ICC input profile must include a MatTRC value.

```
InputProfile = iccread('myRGB.icc');
C = makecform('mattrc',InputProfile.MatTRC, ...
    'direction', 'forward');
```

## See Also

applycform, iccread, iccwrite, isicc, lab2double, lab2uint16, lab2uint8, whitepoint, xyz2double, xyz2uint16

For a full list of the toolbox color space conversion functions, see “Color Space Conversions” on page 16-21.

# makeConstrainToRectFcn

---

**Purpose** Create rectangularly bounded drag constraint function

**Syntax** `fcn = makeConstrainToRectFcn(type,xlim,ylim)`

**Description** `fcn = makeConstrainToRectFcn(type,xlim,ylim)` creates a drag constraint function for draggable tools of a given type, where type is the string: 'impoint', 'imline', or 'imrect'. The rectangular boundaries of the drag constraint function are described by the vectors `xlim` and `ylim` where `xlim = [xmin xmax]` and `ylim = [ymin ymax]`.

**Examples** Constrain drag of impoint within axis limits.

```
figure, plot(1:10);  
h = impoint(gca,2,6);  
api = iptgetapi(h);  
fcn = makeConstrainToRectFcn('impoint',...  
                             get(gca,'XLim'),get(gca,'YLim'));  
api.setDragConstraintFcn(fcn);
```

**See Also** `impoint`, `imrect`, `imline`, `imdistanline`

**Purpose** Create lookup table for use with applylut

**Syntax** lut = makelut(fun,n)

**Description** lut = makelut(fun,n) returns a lookup table for use with applylut. fun is a function that accepts an *n*-by-*n* matrix of 1's and 0's as input and return a scalar. n can be either 2 or 3. 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. fun must be a function handle.

**Class Support** lut is returned as a vector of class double.

**Examples** Construct a lookup table for 2-by-2 neighborhoods. In this example, the function passed to makelut returns TRUE if the number of 1's in the neighborhood is 2 or greater, and returns FALSE otherwise.

```
f = @(x) (sum(x(:)) >= 2);
lut = makelut(f,2)
lut =
    0
    0
    0
    1
    0
    1
    1
    1
    1
    0
    1
    1
    1
    1
```

# makelut

---

1  
1  
1

## See Also

`applylut`



**Purpose** Create resampling structure

**Syntax** `R = makeresampler(interpolant, padmethod)`

**Description** `R = makeresampler(interpolant, padmethod)` creates a separable resampler structure for use with `tformarray` and `imtransform`.

The `interpolant` argument specifies the interpolating kernel that the separable resampler uses. In its simplest form, `interpolant` can have any of the following strings as a value.

Interpolant	Description
'cubic'	Cubic interpolation
'linear'	Linear interpolation
'nearest'	Nearest-neighbor interpolation

If you are using a custom interpolating kernel, you can specify `interpolant` as a cell array in either of these forms:

{half_width, positive_half}	half_width is a positive scalar designating the half width of a symmetric interpolating kernel. positive_half is a vector of values regularly sampling the kernel on the closed interval [0 positive_half].
{half_width, interp_fcn}	interp_fcn is a function handle that returns interpolating kernel values, given an array of input values in the interval [0 positive_half].

To specify the interpolation method independently along each dimension, you can combine both types of interpolant specifications. The number of elements in the cell array must equal the number of transform dimensions. For example, if you specify this value for `interpolant`

```
{'nearest', 'linear', {2 KERNEL_TABLE}}
```

# makeresampler

---

the resampler uses nearest-neighbor interpolation along the first transform dimension, linear interpolation along the second dimension, and a custom table-based interpolation along the third.

The `padmethod` argument controls how the resampler interpolates or assigns values to output elements that map close to or outside the edge of the input array. The following table lists all the possible values of `padmethod`.

Pad Method	Description
'bound'	Assigns values from the fill value array to points that map outside the array and repeats border elements of the array for points that map inside the array (same as 'replicate'). When interpolant is 'nearest', this pad method produces the same results as 'fill'. 'bound' is like 'fill', but avoids mixing fill values and input image values.
'circular'	Pads array with circular repetition of elements within the dimension. Same as <code>padarray</code> .
'fill'	Generates an output array with smooth-looking edges (except when using nearest-neighbor interpolation). For output points that map near the edge of the input array (either inside or outside), it combines input image and fill values. When interpolant is 'nearest', this pad method produces the same results as 'bound'.
'replicate'	Pads array by repeating border elements of array. Same as <code>padarray</code> .
'symmetric'	Pads array with mirror reflections of itself. Same as <code>padarray</code> .

In the case of 'fill', 'replicate', 'circular', or 'symmetric', the resampling performed by `tformarray` or `imtransform` occurs in two logical steps:

- 1 Pad the array A infinitely to fill the entire input transform space.
- 2 Evaluate the convolution of the padded A with the resampling kernel at the output points specified by the geometric map.

Each nontransform dimension is handled separately. The padding is virtual, (accomplished by remapping array subscripts) for performance and memory efficiency. If you implement a custom resampler, you can implement these behaviors.

## Custom Resamplers

The syntax described above construct a resampler structure that uses the separable resampler function that ships with the Image Processing Toolbox. It is also possible to create a resampler structure that uses a user-written resampler by using this syntax:

```
R = makeresampler(PropertyName,PropertyValue,...)
```

The makeresampler function supports the following properties.

Property	Description
'Type'	Can have the value 'separable' or 'custom' and must always be supplied. If 'Type' is 'separable', the only other properties that can be specified are 'Interpolant' and 'PadMethod', and the result is equivalent to using the makeresampler(interpolant,padmethod) syntax. If 'Type' is 'custom', you must specify the 'NDims' and 'ResampleFcn' properties and, optionally, the 'CustomData' property.
'PadMethod'	See the padmethod argument for more information.
'Interpolant'	See the interpolant argument for more information.
'NDims'	Positive integer indicating the dimensionality the custom resampler can handle. Use a value of Inf to indicate that the custom resampler can handle any dimension. If 'Type' is 'custom', NDims is required.

# makeresampler

Property	Description
'ResampleFcn'	<p>Handle to a function that performs the resampling. The function is called with the following interface.</p> <pre>B = resample_fcn(A,M,TDIMS_A,TDIMS_B,FSIZE_A,FSIZE_B,F,R)</pre> <p>See the help for <code>tformarray</code> for information about the inputs <code>A</code>, <code>TDIMS_A</code>, <code>TDIMS_B</code>, and <code>F</code>. The argument <code>M</code> is an array that maps the transform subscript space of <code>B</code> to the transform subscript space of <code>A</code>. If <code>A</code> has <code>N</code> transform dimensions (<code>N = length(TDIMS_A)</code>) and <code>B</code> has <code>P</code> transform dimensions (<code>P = length(TDIMS_B)</code>), then <code>ndims(M) = P + 1</code>, if <code>N &gt; 1</code> and <code>P</code> if <code>N == 1</code>, and <code>size(M,P + 1) = N</code>.</p> <p>The first <code>P</code> dimensions of <code>M</code> correspond to the output transform space, permuted according to the order in which the output transform dimensions are listed in <code>TDIMS_B</code>. (In general <code>TDIMS_A</code> and <code>TDIMS_B</code> need not be sorted in ascending order, although such a limitation might be imposed by specific resamplers.) Thus, the first <code>P</code> elements of <code>size(M)</code> determine the sizes of the transform dimensions of <code>B</code>. The input transform coordinates to which each point is mapped are arrayed across the final dimension of <code>M</code>, following the order given in <code>TDIMS_A</code>. <code>M</code> must be double. <code>FSIZE_A</code> and <code>FSIZE_B</code> are the full sizes of <code>A</code> and <code>B</code>, padded with 1's as necessary to be consistent with <code>TDIMS_A</code>, <code>TDIMS_B</code>, and <code>size(A)</code>.</p>
'CustomData'	User-defined.

## Examples

Stretch an image in the  $y$ -direction using a separable resampler that applies cubic interpolation in the  $y$ -direction and nearest-neighbor interpolation in the  $x$ -direction. (This is equivalent to, but faster than, applying bicubic interpolation.)

```
A = imread('moon.tif');  
resamp = makeresampler({'nearest','cubic'},'fill');  
stretch = maketform('affine',[1 0; 0 1.3; 0 0]);  
B = imtransform(A,stretch,resamp);
```

**See Also** `imtransform`, `tformarray`

# maketform

---

**Purpose** Create spatial transformation structure (TFORM)

**Syntax** `T = maketform(transformtype,...)`

**Description** `T = maketform(transformtype,...)` creates a multidimensional spatial transformation structure (called a TFORM struct) that can be used with the `tformfwd`, `tforminv`, `fliptform`, `imtransform`, or `tformarray` functions.

`transformtype` can be any of the following spatial transformation types. `maketform` supports a special syntax for each transformation type. See the following sections for information about these syntax.

Transform Type	Description
'affine'	Affine transformation in 2-D or N-D
'projective'	Projective transformation in 2-D or N-D
'custom'	User-defined transformation that can be N-D to M-D
'box'	Independent affine transformation (scale and shift) in each dimension
'composite'	Composition of an arbitrary number of more basic transformations

## Transform Types

### Affine

`T = maketform('affine',A)` builds a TFORM struct `T` for an `N`-dimensional affine transformation. `A` is a nonsingular real  $(N+1)$ -by- $(N+1)$  or  $(N+1)$ -by- $N$  matrix. If `A` is  $(N+1)$ -by- $(N+1)$ , the last column of `A` must be `[zeros(N,1);1]`. Otherwise, `A` is augmented automatically, such that its last column is `[zeros(N,1);1]`. The matrix `A` defines a forward transformation such that `tformfwd(U,T)`, where `U` is a 1-by-`N` vector, returns a 1-by-`N` vector `X`, such that  $X = U * A(1:N,1:N) + A(N+1,1:N)$ . `T` has both forward and inverse transformations.

`T = maketform('affine',U,X)` builds a TFORM struct `T` for a two-dimensional affine transformation that maps each row of `U` to the corresponding row of `X`. The `U` and `X` arguments are each 3-by-2 and define the corners of input and output triangles. The corners cannot be collinear.

## Projective

`T = maketform('projective',A)` builds a TFORM struct for an `N`-dimensional projective transformation. `A` is a nonsingular real  $(N+1)$ -by- $(N+1)$  matrix.  $A(N+1, N+1)$  cannot be 0. The matrix `A` defines a forward transformation such that `tformfwd(U,T)`, where `U` is a 1-by-`N` vector, returns a 1-by-`N` vector `X`, such that  $X = W(1:N)/W(N+1)$ , where  $W = [U \ 1] * A$ . The transformation structure `T` has both forward and inverse transformations.

`T = maketform('projective',U,X)` builds a TFORM struct `T` for a two-dimensional projective transformation that maps each row of `U` to the corresponding row of `X`. The `U` and `X` arguments are each 4-by-2 and define the corners of input and output quadrilaterals. No three corners can be collinear.

## Custom

`T = maketform('custom',NDIMS_IN,NDIMS_OUT,...  
FORWARD_FCN, INVERSE_FCN, TDATA)` builds a custom TFORM struct `T` based on user-provided function handles and parameters. `NDIMS_IN` and `NDIMS_OUT` are the numbers of input and output dimensions. `FORWARD_FCN` and `INVERSE_FCN` are function handles to forward and inverse functions. Those functions must support the following syntax:

Forward function:  $X = \text{FORWARD\_FCN}(U, T)$

Inverse function:  $U = \text{INVERSE\_FCN}(X, T)$

where `U` is a `P`-by-`NDIMS_IN` matrix whose rows are points in the transformation's input space, and `X` is a `P`-by-`NDIMS_OUT` matrix whose rows are points in the transformation's output space. The

TDATA argument can be any MATLAB array and is typically used to store parameters of the custom transformation. It is accessible to FORWARD\_FCN and INVERSE\_FCN via the tdata field of T. Either FORWARD\_FCN or INVERSE\_FCN can be empty, although at least INVERSE\_FCN must be defined to use T with tformarray or imtransform.

## Box

T = maketform('box', tsize, LOW, HIGH) or  
T = maketform('box', INBOUNDS, OUTBOUNDS) builds an N-dimensional affine TFORM struct T. The tsize argument is an N-element vector of positive integers. LOW and HIGH are also N-element vectors. The transformation maps an input box defined by the opposite corners ones(1,N) and tsize or, alternatively, by corners INBOUNDS(1,:) and INBOUND(2,:) to an output box defined by the opposite corners LOW and HIGH or OUTBOUNDS(1,:) and OUTBOUNDS(2,:). LOW(K) and HIGH(K) must be different unless tsize(K) is 1, in which case the affine scale factor along the Kth dimension is assumed to be 1.0. Similarly, INBOUNDS(1,K) and INBOUNDS(2,K) must be different unless OUTBOUNDS(1,K) and OUTBOUNDS(2,K) are the same, and vice versa. The 'box' TFORM is typically used to register the row and column subscripts of an image or array to some world coordinate system.

## Composite

T = maketform('composite', T1, T2, ..., TL) or  
T = maketform('composite', [T1 T2 ... TL]) builds a TFORM struct T whose forward and inverse functions are the functional compositions of the forward and inverse functions of T1, T2, ..., TL.

For example, if L = 3, then tformfwd(U, T) is the same as tformfwd(tformfwd(tformfwd(U, T3), T2), T1). The components T1 through TL must be compatible in terms of the numbers of input and output dimensions. T has a defined forward transform function only if all the component transforms have defined forward transform functions. T has a defined inverse transform function only if all the component functions have defined inverse transform functions.



## Examples

Make and apply an affine transformation.

```
T = maketform('affine',[.5 0 0; .5 2 0; 0 0 1]);
tformfwd([10 20],T)
I = imread('cameraman.tif');
I2 = imtransform(I,T);
imshow(I), figure, imshow(I2)
```

## See Also

tformfwd, tforminv, fliptform, imtransform, tformarray

# mat2gray

---

**Purpose** Convert matrix to grayscale image

**Syntax**  
`I = mat2gray(A,[amin amax])`  
`I = mat2gray(A)`

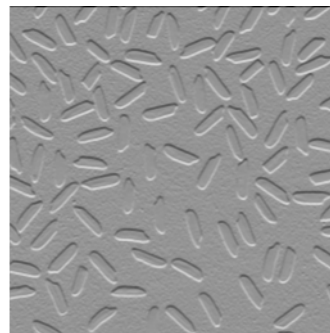
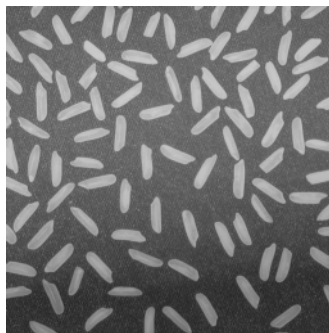
**Description** `I = mat2gray(A,[amin amax])` converts the matrix `A` to the intensity image `I`. The returned matrix `I` contains values in the range 0.0 (black) to 1.0 (full intensity or white). `amin` and `amax` are the values in `A` that correspond to 0.0 and 1.0 in `I`.

`I = mat2gray(A)` sets the values of `amin` and `amax` to the minimum and maximum values in `A`.

**Class Support** The input array `A` can be logical or numeric. The output image `I` is double.

**Examples**

```
I = imread('rice.png');  
J = filter2(fspecial('sobel'),I);  
K = mat2gray(J);  
imshow(I), figure, imshow(K)
```



**See Also** `gray2ind`

<b>Purpose</b>	Average or mean of matrix elements
<b>Syntax</b>	<code>B = mean2(A)</code>
<b>Description</b>	<code>B = mean2(A)</code> computes the mean of the values in A.
<b>Class Support</b>	The input image A can be numeric or logical. The output image B is a scalar of class double.
<b>Algorithm</b>	<code>mean2</code> computes the mean of an array A using <code>mean(A(:))</code> .
<b>See Also</b>	<code>std2</code> <code>mean</code> , <code>std</code> in the MATLAB Function Reference

# medfilt2

---

**Purpose** 2-D 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 0's on the edges, so the median values for the points within `[m n]/2` of the edges might 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 0's if the class of `A` is `uint8`, or 1's if the class of `A` is `double`.

**Class Support** The input image `A` can be of class `logical`, `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`.

---

**Note** For information about performance considerations, see `ordfilt2`.

---

**Remarks** If the input image `A` is of an integer class, all the output values are returned as integers. If the number of pixels in the neighborhood (i.e., `m*n`) is even, some of the median values might not be integers. In these cases, the fractional parts are discarded. Logical input is treated similarly.

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.

## Examples

Add salt and pepper noise to an image and then restore 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 rectangular montage

**Syntax**

```
montage(I)
montage(BW)
montage(X,map)
montage(RGB)
h = montage(...)
```

**Description** montage displays all 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.

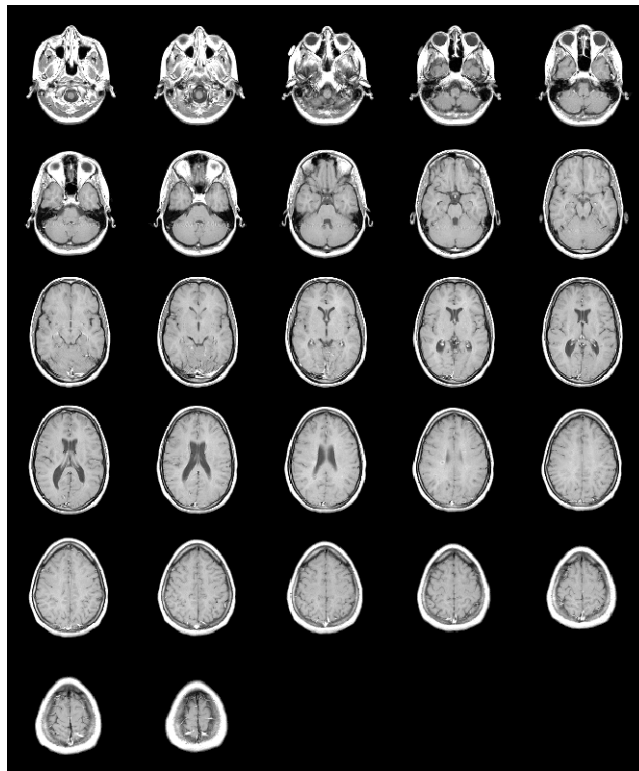
montage(RGB) displays the k frames of the truecolor image array RGB. RGB is m-by-n-by-3-by-k.

h = montage(...) returns the handle to the image object.

**Class Support** An intensity image can be logical, uint8, uint16, int16, single, or double. An indexed image can be logical, uint8, uint16, single, or double. The map must be double. A truecolor image can be uint8, uint16, single, or double. The output is a handle to the graphics objects produced by this function.

**Examples**

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



**See Also**

`imovie`

# nlfilter

---

<b>Purpose</b>	General sliding-neighborhood operations
<b>Syntax</b>	<pre>B = nlfilter(A,[m n],fun) B = nlfilter(A,'indexed',...)</pre>
<b>Description</b>	<p><code>B = nlfilter(A,[m n],fun)</code> applies the function <code>fun</code> to each <code>m</code>-by-<code>n</code> sliding block of <code>A</code>. <code>fun</code> is a function that accepts an <code>m</code>-by-<code>n</code> matrix as input and returns a scalar result.</p> <pre>c = fun(x)</pre> <p><code>fun</code> must be a function handle.</p> <p><code>c</code> is the output value for the center pixel in the <code>m</code>-by-<code>n</code> block <code>x</code>. <code>nlfilter</code> calls <code>fun</code> for each pixel in <code>A</code>. <code>nlfilter</code> zero-pads the <code>m</code>-by-<code>n</code> block at the edges, if necessary.</p> <p><code>B = nlfilter(A,'indexed',...)</code> processes <code>A</code> as an indexed image, padding with 1's if <code>A</code> is of class <code>double</code> and 0's if <code>A</code> is of class <code>uint8</code>.</p>
<b>Class Support</b>	The input image <code>A</code> can be of any class supported by <code>fun</code> . The class of <code>B</code> depends on the class of the output from <code>fun</code> .
<b>Remarks</b>	<code>nlfilter</code> can take a long time to process large images. In some cases, the <code>colfilt</code> function can perform the same operation much faster.
<b>Examples</b>	<p>This example produces the same result as calling <code>medfilt2</code> with a 3-by-3 neighborhood.</p> <pre>A = imread('cameraman.tif'); fun = @(x) median(x(:)); B = nlfilter(A,[3 3],fun); imshow(A), figure, imshow(B)</pre>
<b>See Also</b>	<code>blkproc</code> , <code>colfilt</code> , <code>function_handle</code>



<b>Purpose</b>	Normalized 2-D cross-correlation
<b>Syntax</b>	<code>C = normxcorr2(TEMPLATE,A)</code>
<b>Description</b>	<code>C = normxcorr2(TEMPLATE,A)</code> computes the normalized cross-correlation of the matrices <code>TEMPLATE</code> and <code>A</code> . The matrix <code>A</code> must be larger than the matrix <code>TEMPLATE</code> for the normalization to be meaningful. The values of <code>TEMPLATE</code> cannot all be the same. The resulting matrix <code>C</code> contains the correlation coefficients, which can range in value from -1.0 to 1.0.
<b>Class Support</b>	The input matrices can be of class <code>uint8</code> , <code>uint16</code> , or <code>double</code> .
<b>Algorithm</b>	<code>normxcorr2</code> uses the following general procedure: <ol style="list-style-type: none"><li>1 Calculate cross-correlation in the spatial or the frequency domain, depending on size of images.</li><li>2 Calculate local sums by precomputing running sums. [1]</li><li>3 Use local sums to normalize the cross-correlation to get correlation coefficients. [2]</li></ol>
<b>Examples</b>	<pre>T = .2*ones(11); % make light gray plus on dark gray background T(6,3:9) = .6; T(3:9,6) = .6; BW = T&gt;0.5;      % make white plus on black background imshow(BW), title('Binary') figure, imshow(T), title('Template')  % make new image that offsets template T T_offset = .2*ones(21); offset = [3 5]; % shift by 3 rows, 5 columns T_offset( 1:size(T,1)+offset(1), (1:size(T,2))+offset(2) ) = T; imshow(T_offset), title('Offset Template')</pre>

# normxcorr2

---

```
% cross-correlate BW and T_offset to recover offset
cc = normxcorr2(BW,T_offset);
[max_cc, imax] = max(abs(cc(:)));
[ypeak, xpeak] = ind2sub(size(cc),imax(1));
corr_offset = [ (ypeak-size(T,1)) (xpeak-size(T,2)) ];
isequal(corr_offset,offset) % 1 means offset was recovered
```

## See Also

corrcoef

## References

- [1] Lewis, J. P., "Fast Normalized Cross-Correlation," *Industrial Light & Magic*
- [2] Haralick, Robert M., and Linda G. Shapiro, *Computer and Robot Vision*, Volume II, Addison-Wesley, 1992, pp. 316-317.

**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`.

**Examples** Convert RGB image to NTSC and back.

```
RGB = imread('board.tif');
NTSC = rgb2ntsc(RGB);
RGB2 = ntsc2rgb(NTSC);
```

**See Also** `rgb2ntsc`, `rgb2ind`, `ind2rgb`, `ind2gray`

For a full list of the toolbox color space conversion functions, see “Color Space Conversions” on page 16-21.

# ordfilt2

---

## Purpose

2-D 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 `orderth` 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 0's at the boundaries. If `padopt` is 'symmetric', `A` is symmetrically extended at the boundaries.

## Class Support

The class of `A` can be `logical`, `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.

## Performance Considerations

When working with large domain matrices that do not contain any zero-valued elements, `ordfilt2` can achieve higher performance if `A` is in an integer data format (`uint8`, `int8`, `uint16`, `int16`). The gain in speed is larger for `uint8` and `int8` than for the 16-bit data types. For 8-bit data formats, the domain matrix must contain seven or more rows. For 16-bit data formats, the domain matrix must contain three or more rows and 520 or more elements.

## Examples

This examples uses a maximum filter with a [5 5] neighborhood. This is equivalent to `imdilate(image, strel('square',5))`.

```
A = imread('snowflakes.png');
B = ordfilt2(A,25,true(5));
figure, imshow(A), figure, imshow(B)
```

## See Also

`medfilt2`

## Reference

[1] Haralick, Robert M., and Linda G. Shapiro, *Computer and Robot Vision*, Volume I, Addison-Wesley, 1992.

[2] Huang, T.S., G.J.Yang, and G.Y.Tang. "A fast two-dimensional median filtering algorithm.", *IEEE transactions on Acoustics, Speech and Signal Processing*, Vol ASSP 27, No. 1, February 1979

# otf2psf

---

**Purpose** Convert optical transfer function to point-spread function

**Syntax** PSF = otf2psf(OTF)  
PSF = otf2psf(OTF,OUTSIZE)

**Description** PSF = otf2psf(OTF) computes the inverse Fast Fourier Transform (IFFT) of the optical transfer function (OTF) array and creates a point-spread function (PSF), centered at the origin. By default, the PSF is the same size as the OTF.

PSF = otf2psf(OTF,OUTSIZE) converts the OTF array into a PSF array, where OUTSIZE specifies the size of the output point-spread function. The size of the output array must not exceed the size of the OTF array in any dimension.

To center the PSF at the origin, otf2psf circularly shifts the values of the output array down (or to the right) until the (1,1) element reaches the central position, then it crops the result to match dimensions specified by OUTSIZE.

Note that this function is used in image convolution/deconvolution when the operations involve the FFT.

**Class Support** OTF can be any nonsparse, numeric array. PSF is of class double.

**Examples**

```
PSF = fspecial('gaussian',13,1);
OTF = psf2otf(PSF,[31 31]); % PSF --> OTF
PSF2 = otf2psf(OTF,size(PSF)); % OTF --> PSF2
subplot(1,2,1); surf(abs(OTF)); title('|OTF|');
axis square; axis tight
subplot(1,2,2); surf(PSF2); title('Corresponding PSF');
axis square; axis tight
```

**See Also** psf2otf, circshift, padarray

**Purpose** Pad array

**Syntax**

```
B = padarray(A,padsiz)
B = padarray(A,padsiz,padval)
B = padarray(A,padsiz,padval,direction)
```

**Description** `B = padarray(A,padsiz)` pads array `A` with 0's (zeros). `padsiz` is a vector of positive integers that specifies both the amount of padding to add and the dimension along which to add it. The value of an element in the vector specifies the amount of padding to add. The order of the element in the vector specifies the dimension along which to add the padding.

For example, a `padsiz` value of `[2 3]` means add 2 elements of padding along the first dimension and 3 elements of padding along the second dimension. By default, `padarray` adds padding before the first element and after the last element along the specified dimension.

`B = padarray(A,padsiz,padval)` pads array `A` where `padval` specifies the value to use as the pad value. `padarray` uses the value 0 (zero) as the default. `padval` can be a scalar that specifies the pad value directly or one of the following text strings that specifies the method `padarray` uses to determine the values of the elements added as padding.

Value	Meaning
'circular'	Pad with circular repetition of elements within the dimension.
'replicate'	Pad by repeating border elements of array.
'symmetric'	Pad array with mirror reflections of itself.

`B = padarray(A,padsiz,padval,direction)` pads `A` in the direction specified by the string `direction`. `direction` can be one of the following strings. The default value is enclosed in braces (`{}`).

# padarray

Value	Meaning
{ 'both' }	Pads before the first element and after the last array element along each dimension. This is the default.
'post'	Pad after the last array element along each dimension.
'pre'	Pad before the first array element along each dimension.

## Class Support

When padding with a constant value, A can be numeric or logical. When padding using the 'circular', 'replicate', or 'symmetric' methods, A can be of any class. B is of the same class as A.

## Examples

### Example 1

Add three elements of padding to the beginning of a vector. The padding elements, indicated by the gray shading, contain mirror copies of the array elements.

```
a = [ 1 2 3 4 ];  
b = padarray(a,[0 3],'symmetric','pre')  
b ==  
  3  2  1  1  2  3  4
```

### Example 2

Add three elements of padding to the end of the first dimension of the array and two elements of padding to the end of the second dimension. The example uses the value of the last array element as the padding value.

```
A = [1 2; 3 4];  
B = padarray(A,[3 2],'replicate','post')  
  1  2  2  2  
  3  4  4  4  
  3  4  4  4  
  3  4  4  4  
  3  4  4  4  
B =
```



### Example 3

Add three elements of padding to the vertical and horizontal dimensions of a three-dimensional array. Use default values for the pad value and direction.

```
A = [ 1 2; 3 4];
```

```
B = [ 5 6; 7 8];
```

```
C = cat(3,A,B)
```

```
C(:,:,1) =
```

```
    1    2
    3    4
```

```
C(:,:,2) =
```

```
    5    6
    7    8
```

```
D = padarray(C,[3 3])
```

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	1	2	0	0	0
0	0	0	3	4	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

```
D(:,:,1) ==
```

# padarray

---

```
D(:,:,2) ==  
  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  5  6  0  0  0  
  0  0  0  7  8  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
```

## See Also

`circshift`, `imfilter`

**Purpose** Convert parallel-beam projections to fan-beam

**Syntax**

```
F = para2fan(P,D)
I = para2fan(...,param1,val1,param2,val2,...)
[F,fan_positions,fan_rotation_angles] = fan2para(...)
```

**Description** F = para2fan(P,D) computes the fan-beam data (sinogram) F from the parallel-beam data (sinogram) P. Each column of P contains the parallel-beam sensor samples at one rotation angle. D is the distance in pixels from the center of rotation to the center of the sensors.

The sensors are assumed to have a one-pixel spacing. The parallel-beam rotation angles are assumed to be spaced equally to cover [0,180] degrees. The calculated fan-beam rotation angles cover [0,360) with the same spacing as the parallel-beam rotation angles. The calculated fan-beam angles are equally spaced with the spacing set to the smallest angle implied by the sensor spacing.

I = para2fan(...,param1,val1,param2,val2,...) specifies parameters that control various aspects of the para2fan conversion. Parameter names can be abbreviated, and case does not matter. Default values are enclosed in braces like this: {default}. Parameters include

Parameter	Description
'FanCoverage'	String specifying the range through which the beams are rotated.  Possible values: {'cycle'} or 'minimal'  See ifanbeam for details.

# para2fan

---

Parameter	Description
'FanRotationIncrement'	<p>Positive real scalar specifying the rotation angle increment of the fan-beam projections in degrees.</p> <p>If 'FanCoverage' is 'cycle', 'FanRotationIncrement' must be a factor of 360.</p> <p>If 'FanRotationIncrement' is not specified, then it is set to the same spacing as the parallel-beam rotation angles.</p>
'FanSensorGeometry'	<p>Text string specifying how sensors are positioned.</p> <p>Possible values: { 'arc' } or 'line'</p> <p>See fanbeam for details.</p>

Parameter	Description
'FanSensorSpacing'	<p>Positive real scalar specifying the spacing of the fan beams. Interpretation of the value depends on the setting of 'FanSensorGeometry':</p> <p>If 'FanSensorGeometry' is 'arc', the value defines the angular spacing in degrees. Default value is 1.</p> <p>If 'FanSensorGeometry' is 'line', the value defines the linear spacing in pixels.</p> <p>If 'FanSensorSpacing' is not specified, the default is the smallest value implied by 'ParallelSensorSpacing' such that</p> <p>If 'FanSensorGeometry' is 'arc',  'FanSensorSpacing' is</p> $180/\text{PI} * \text{ASIN}(\text{PSPACE}/D)$ <p>where PSPACE is the value of 'ParallelSensorSpacing'.</p> <p>If 'FanSensorGeometry' is 'line',  'FanSensorSpacing' is</p> $D * \text{ASIN}(\text{PSPACE}/D)$
'Interpolation'	<p>Text string specifying the type of interpolation used between the parallel-beam and fan-beam data.</p> <p>'nearest' — Nearest-neighbor</p> <p>{'linear'} — Linear</p> <p>'spline' — Piecewise cubic spline</p> <p>'pchip' — Piecewise cubic Hermite (PCHIP)</p> <p>'cubic' — Same as 'pchip'</p>

# para2fan

Parameter	Description
'ParallelCoverage'	Text string specifying the range of rotation. 'cycle' -- Parallel data covers 360 degrees { 'halfcycle' } — Parallel data covers 180 degrees
'ParallelRotationIncrement'	Positive real scalar specifying the parallel-beam rotation angle increment, measured in degrees. Parallel beam angles are calculated to cover [0,180) degrees with increment PAR_ROT_INC, where PAR_ROT_INC is the value of 'ParallelRotationIncrement'. 180/PAR_ROT_INC must be an integer.  If 'ParallelRotationIncrement' is not specified, the increment is assumed to be the same as the increment of the fan-beam rotation angles.
'ParallelSensorSpacing'	Positive real scalar specifying the spacing of the parallel-beam sensors in pixels. The range of sensor locations is implied by the range of fan angles and is given by $[D*\sin(\min(\text{FAN\_ANGLES})), D*\sin(\max(\text{FAN\_ANGLES}))]$  If 'ParallelSensorSpacing' is not specified, the spacing is assumed to be uniform and is set to the minimum spacing implied by the fan angles and sampled over the range implied by the fan angles.

[F, fan\_positions, fan\_rotation\_angles] = fan2para(...) returns the fan-beam sensor measurement *angles* in fan\_positions, if 'FanSensorGeometry' is 'arc'. If 'FanSensorGeometry' is 'line', fan\_positions contains the fan-beam sensor *positions* along the line of sensors. fan\_rotation\_angles contains rotation angles.

**Class Support**

P and D can be double or single, and must be nonsparse. The other numeric input arguments must be double. The output arguments are double.

**Examples**

Generate parallel-beam projections

```
ph = phantom(128);
theta = 0:180;
[P, xp] = radon(ph, theta);
imshow(theta, xp, P, [], 'n'), axis normal
title('Parallel-Beam Projections')
xlabel('\theta (degrees)')
ylabel('x''')
colormap(hot), colorbar
```

Convert to fan-beam projections

```
[F, Fpos, Fangles] = para2fan(P, 100);
figure, imshow(Fangles, Fpos, F, [], 'n'), axis normal
title('Fan-Beam Projections')
xlabel('\theta (degrees)')
ylabel('Sensor Locations (degrees)')
colormap(hot), colorbar
```

**See Also**

fan2para, fanbeam, iradon, ifanbeam, phantom, radon

# phantom

---

**Purpose** Create 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' — Test image used widely by researchers in tomography
- 'Modified Shepp-Logan' (default) — 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 semiaxis of the ellipse
Column 3	b	Length of the vertical semiaxis of the ellipse



Column	Parameter	Meaning
Column 4	x0	$x$ -coordinate of the center of the ellipse
Column 5	y0	$y$ -coordinate of the center of the ellipse
Column 6	phi	Angle (in degrees) between the horizontal semiaxis 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 and all outputs must be 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 can have values outside the range  $[0,1]$ .

## Examples

```
P = phantom('Modified Shepp-Logan',200);
imshow(P)
```

# phantom

---



## 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)  
pixval(ax,option)  
pixval(H,option)
```

---

**Note** This function is obsolete and may be removed in a future version of the Image Processing Toolbox. Instead, use `impixelinfo` for pixel reporting and use `imdistline` for measuring distance.

---

## Description

`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 the image and hold down the mouse button while you move the cursor, `pixval` also displays the Euclidean distance between the point you clicked 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 it and dragging it to another place in the figure.

`pixval on` turns interactive display on in the current 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

---

`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 a string containing 'on' or 'off'.

`pixval(ax,option)` applies the `pixval` command to the figure that contains the axes `ax`. `option` is a string containing 'on' or 'off'.

`pixval(H,option)` applies the `pixval` command to the figure that contains the image object `H`. `option` is a string containing 'on' or 'off'.

## Examples

```
figure, imshow peppers.png  
pixval
```

## See Also

`impixel`, `improfile`, `pixval`

**Purpose** Convert region-of-interest polygon to region mask

**Syntax** `BW = poly2mask(x,y,m,n)`

**Description** `BW = poly2mask(x,y,m,n)` computes a binary region-of-interest mask, BW, from a region-of-interest polygon, represented by the vectors X and Y. The size of BW is M-by-N. `poly2mask` sets pixels in BW that are inside the polygon (X,Y) to 1 and sets pixels outside the polygon to 0.

`poly2mask` closes the polygon automatically if it isn't already closed.

**Class Support** The class of BW is `logical`

**Examples**

```
x = [63 186 54 190 63];
y = [60 60 209 204 60];
bw = poly2mask(x,y,256,256);
imshow(bw)
hold on
plot(x,y,'b','LineWidth',2)
hold off
```

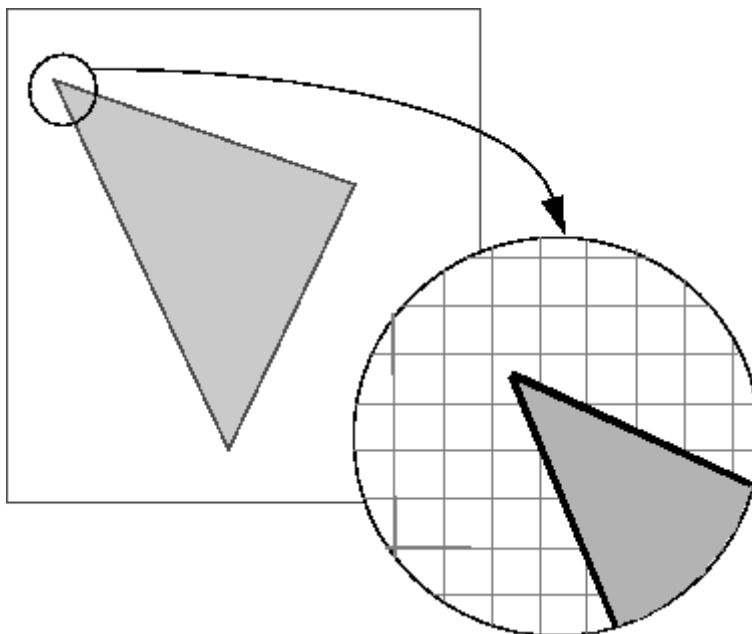
Create a mask using random points.

```
x = 256*rand(1,4);
y = 256*rand(1,4);
x(end+1) = x(1);
y(end+1) = y(1);
bw = poly2mask(x,y,256,256);
imshow(bw)
hold on
plot(x,y,'b','LineWidth',2)
hold off
```

**See Also** `roipoly`

## Algorithm

When creating a region-of-interest mask, `poly2mask` must determine which pixels are included in the region. This determination can be difficult when pixels on the edge of a region are only partially covered by the border line. The following figure illustrates a triangular region of interest, examining in close-up one of the vertices of the ROI. The figure shows how pixels can be partially covered by the border of a region-of-interest.



### **Pixels on the Edge of an ROI Are Only Partially Covered by Border**

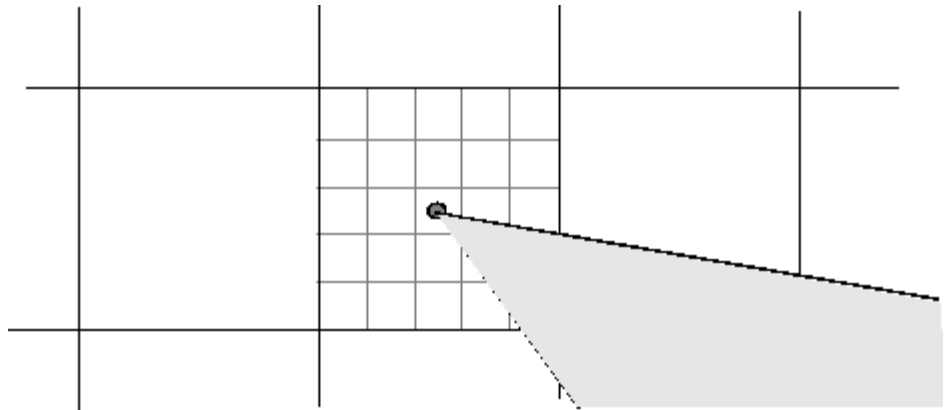
To determine which pixels are in the region, `poly2mask` uses the following algorithm:

- 1 Divide each pixel (unit square) into a 5-by-5 grid. See “Dividing Pixels into a 5-by-5 Subpixel Grid” on page 17-519 for an illustration.

- 2 Adjust the position of the vertices to be on the intersections of the subpixel grid. See “Adjusting the Vertices to the Subpixel Grid” on page 17-519 for an illustration.
- 3 Draw a path from each adjusted vertex to the next, following the edges of the subpixel grid. See “Drawing a Path Between the Adjusted Vertices” on page 17-520 for an illustration.
- 4 Determine which border pixels are inside the polygon using this rule: if a pixel’s central subpixel is inside the boundaries defined by the path between adjusted vertices, the pixel is considered inside the polygon. See “Determining Which Pixels Are in the Region” on page 17-521 for an illustration.

### Dividing Pixels into a 5-by-5 Subpixel Grid

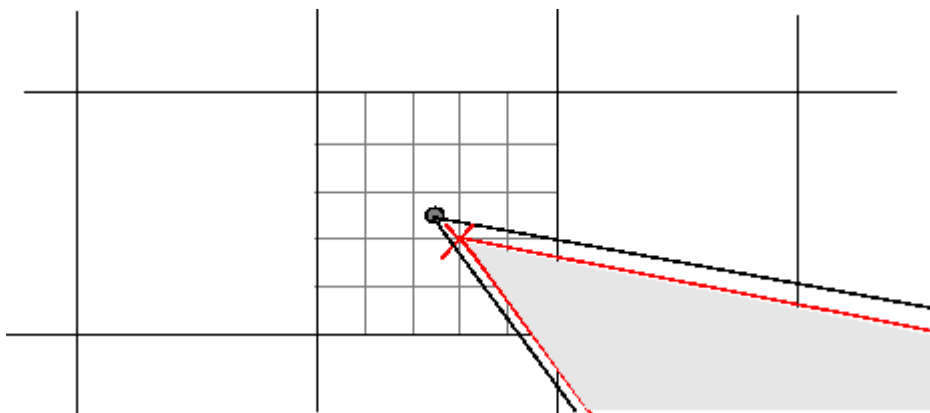
The following figure shows the pixel that contains the vertex of the ROI shown previously with this 5-by-5 subpixel grid.



### Adjusting the Vertices to the Subpixel Grid

poly2mask adjusts each vertex of the polygon so that the vertex lies on the subpixel grid. Note how poly2mask rounds up  $x$  and  $y$  coordinates to find the nearest grid corner. This creates a second, modified polygon,

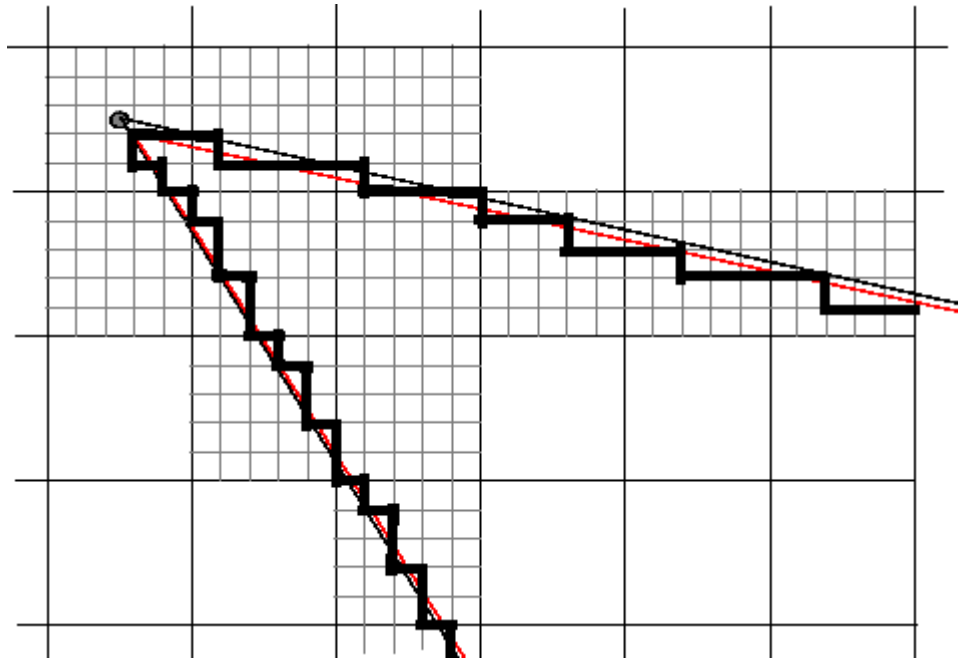
slightly smaller, in this case, than the original ROI. A portion is shown in the following figure.



### **Drawing a Path Between the Adjusted Vertices**

poly2mask forms a path from each adjusted vertex to the next, following the edges of the subpixel grid. In the following figure, a portion of this modified polygon is shown by the thick dark lines.

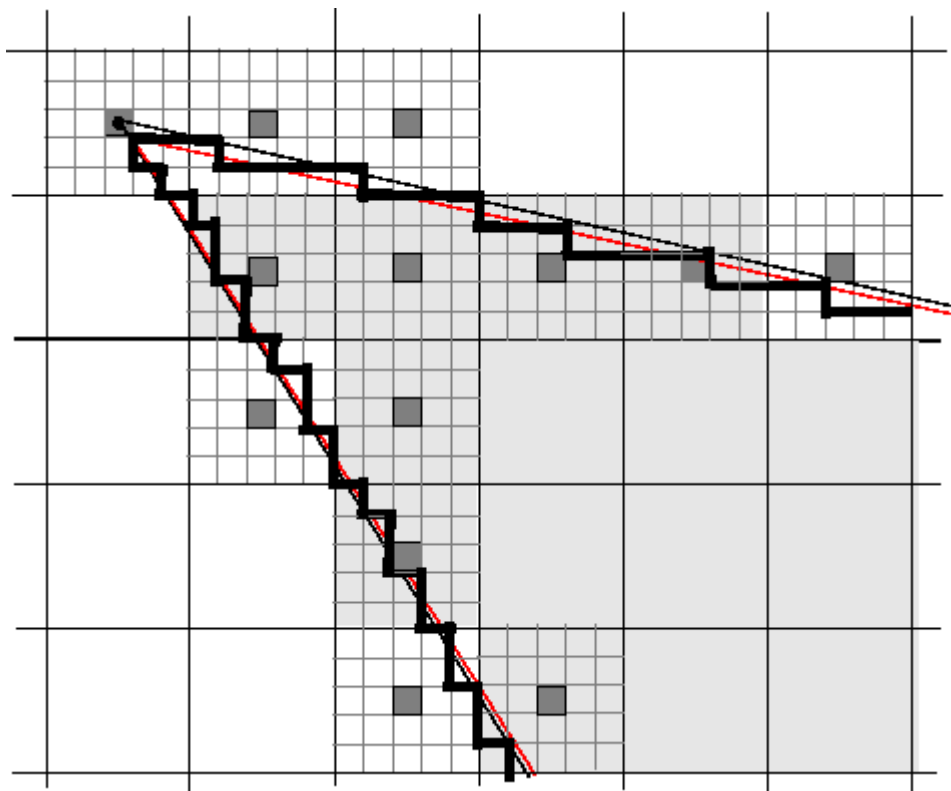




### Determining Which Pixels Are in the Region

poly2mask uses the following rule to determine which border pixels are inside the polygon: if the pixel's central subpixel is inside the modified polygon, the pixel is inside the region.

In the following figure, the central subpixels of pixels on the ROI border are shaded a dark gray color. Pixels inside the polygon are shaded a lighter gray. Note that the pixel containing the vertex is not part of the ROI because its center pixel is not inside the modified polygon.



---

<b>Purpose</b>	Convert point-spread function to optical transfer function
<b>Syntax</b>	<pre>OTF = psf2otf(PSF) OTF = psf2otf(PSF,OUTSIZE)</pre>
<b>Description</b>	<p><code>OTF = psf2otf(PSF)</code> computes the fast Fourier transform (FFT) of the point-spread function (PSF) array and creates the optical transfer function array, <code>OTF</code>, that is not influenced by the PSF off-centering. By default, the <code>OTF</code> array is the same size as the PSF array.</p> <p><code>OTF = psf2otf(PSF,OUTSIZE)</code> converts the PSF array into an <code>OTF</code> array, where <code>OUTSIZE</code> specifies the size of the <code>OTF</code> array. <code>OUTSIZE</code> cannot be smaller than the PSF array size in any dimension.</p> <p>To ensure that the <code>OTF</code> is not altered because of PSF off-centering, <code>psf2otf</code> postpads the PSF array (down or to the right) with 0's to match dimensions specified in <code>OUTSIZE</code>, then circularly shifts the values of the PSF array up (or to the left) until the central pixel reaches (1,1) position.</p> <p>Note that this function is used in image convolution/deconvolution when the operations involve the FFT.</p>
<b>Class Support</b>	PSF can be any nonsparse, numeric array. <code>OTF</code> is of class <code>double</code> .
<b>Examples</b>	<pre>PSF = fspecial('gaussian',13,1); OTF = psf2otf(PSF,[31 31]); % PSF --&gt; OTF subplot(1,2,1); surf(PSF); title('PSF'); axis square; axis tight subplot(1,2,2); surf(abs(OTF)); title('Corresponding  OTF '); axis square; axis tight</pre>
<b>See Also</b>	<code>otf2psf</code> , <code>circshift</code> , <code>padarray</code>

**Purpose**            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)
```

**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 can 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` returns a logical `k`-element vector, whose values are 1 if the corresponding block should be split, and 0 otherwise. (For example, if `k(3)` is 0, the third `m-by-m` block should not be split.) `fun` must be a function\_handle.

## Class Support

For the syntax that do not include a function, the input image can be of class `logical`, `uint8`, `uint16`, `int16`, `single`, or `double`. For the syntax 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.

## Examples

```
I = uint8([1 1 1 2 3 6 6;...
          1 1 2 1 4 5 6 8;...
          1 1 1 1 7 7 7 7;...
          1 1 1 1 6 6 5 5;...
          20 22 20 22 1 2 3 4;...
          20 22 22 20 5 4 7 8;...
          20 22 20 20 9 12 40 12;...
          20 22 20 20 13 14 15 16]);
```

```
S = qtdecomp(I, .05);
disp(full(S));
```

View the block representation of quadtree decomposition.

```
I = imread('liftingbody.png');
S = qtdecomp(I,.27);
blocks = repmat(uint8(0),size(S));

for dim = [512 256 128 64 32 16 8 4 2 1];
    numblocks = length(find(S==dim));
    if (numblocks > 0)
        values = repmat(uint8(1),[dim dim numblocks]);
        values(2:dim,2:dim,:) = 0;
        blocks = qtsetblk(blocks,S,dim,values);
    end
end

blocks(end,1:end) = 1;
blocks(1:end,end) = 1;

imshow(I), figure, imshow(blocks,[])
```

The following figure shows the original image and a representation of the quadtree decomposition of the image.

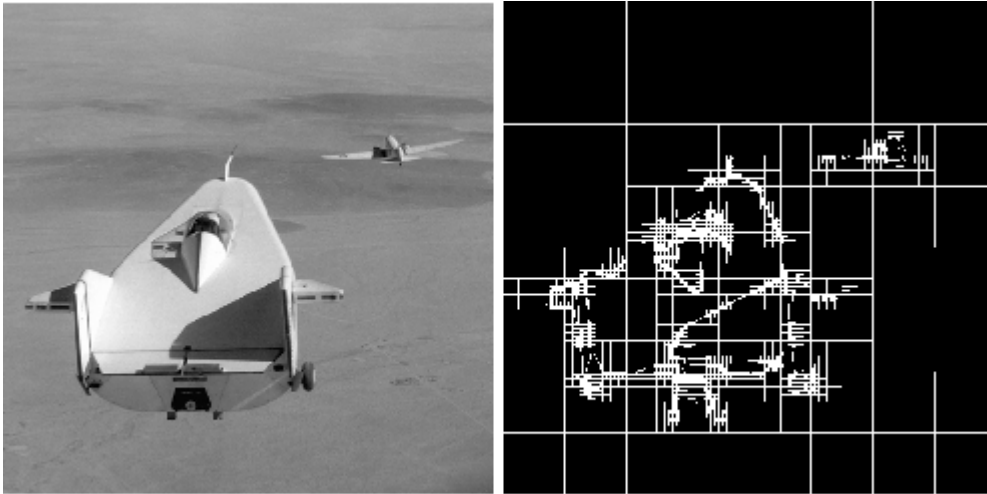


Image Courtesy of NASA

**See Also** `function_handle`, `qtgetblk`, `qtsetblk`

# qtgetblk

---

**Purpose** 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 `logical`, `uint8`, `uint16`, `int16`, `single`, 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.



**Examples**

```
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);
```

```
[vals,r,c] = qtgetblk(I,S,4)
```

**See Also**

qtdecomp, qtsetblk

# 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 `logical`, `uint8`, `uint16`, `int16`, `single`, 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.

## Examples

```
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];
```

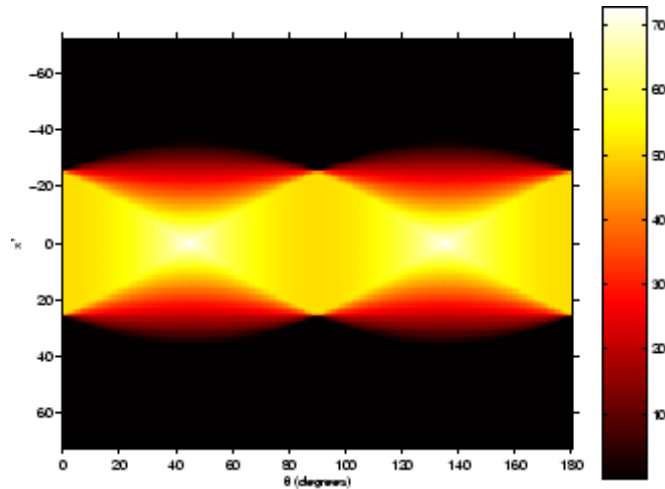
```
S = qtdecomp(I,5);
```

```
newvals = cat(3,zeros(4),ones(4));
J = qtsetblk(I,S,4,newvals)
```

**See Also** `qtdecomp`, `qtgetblk`

<b>Purpose</b>	Radon transform
<b>Syntax</b>	<pre>R = radon(I,theta) [R, xp] = radon(...)</pre>
<b>Description</b>	<p><code>R = radon(I, theta)</code> returns the Radon transform <math>R</math> of the intensity image <math>I</math> for the angle <math>\theta</math> degrees.</p> <p>The Radon transform is the projection of the image intensity along a radial line oriented at a specific angle. If <math>\theta</math> is a scalar, <math>R</math> is a column vector containing the Radon transform for <math>\theta</math> degrees. If <math>\theta</math> is a vector, <math>R</math> is a matrix in which each column is the Radon transform for one of the angles in <math>\theta</math>. If you omit <math>\theta</math>, it defaults to 0:179.</p> <p><code>[R, xp] = radon(...)</code> returns a vector <math>xp</math> containing the radial coordinates corresponding to each row of <math>R</math>.</p> <p>The radial coordinates returned in <math>xp</math> are the values along the <math>x'</math>-axis, which is oriented at <math>\theta</math> degrees counterclockwise from the <math>x</math>-axis. The origin of both axes is the center pixel of the image, which is defined as</p> $\text{floor}((\text{size}(I)+1)/2)$ <p>For example, in a 20-by-30 image, the center pixel is (10,15).</p>
<b>Class Support</b>	$I$ can be of class <code>double</code> , <code>logical</code> , or any integer class. All other inputs and outputs are of class <code>double</code> .
<b>Examples</b>	<pre>iptsetpref('ImshowAxesVisible','on') I = zeros(100,100); I(25:75, 25:75) = 1; theta = 0:180; [R, xp] = radon(I,theta); imshow(R,[],'Xdata',theta,'Ydata',xp,...         'InitialMagnification','fit') xlabel('\theta (degrees)') ylabel('x''')</pre>

```
colormap(hot), colorbar
```



## See Also

fan2para, fanbeam, ifanbeam, iradon, para2fan, phantom

## References

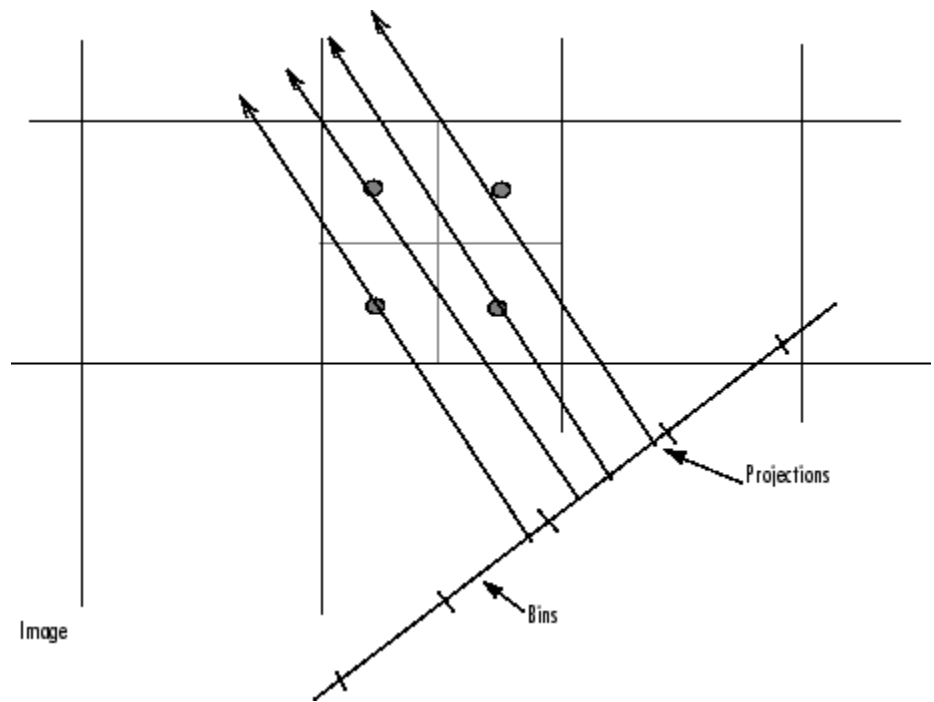
Bracewell, Ronald N., *Two-Dimensional Imaging*, Englewood Cliffs, NJ, Prentice Hall, 1995, pp. 505-537.

Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, pp. 42-45.

## Algorithm

The Radon transform of an image is the sum of the Radon transforms of each individual pixel, the superposition principle.

The algorithm first divides pixels in the image into four parts and projects each subdivision separately, as shown in the following figure.



Each pixel's contribution is proportionally split into the two nearest bins, according to the distance between the projected location and the bin centers. If the projection hits the center point of a bin, the bin on the axes gets the full .25 % weight of the pixel. If the projection hits the border between two bins, the bins share half the .25 %.

# rangefilt

---

**Purpose** Local range of image

**Syntax**  
`J = rangefilt(I)`  
`J = rangefilt(I,NHOOD)`

**Description** `J = rangefilt(I)` returns the array `J`, where each output pixel contains the range value (maximum value - minimum value) of the 3-by-3 neighborhood around the corresponding pixel in the input image `I`. `I` can have any dimension. The output image `J` is the same size as the input image `I`.

`J = rangefilt(I,NHOOD)` performs range filtering of the input image `I` where you specify the neighborhood in `NHOOD`. `NHOOD` is a multidimensional array of zeros and ones where the nonzero elements specify the neighborhood for the range filtering operation. `NHOOD`'s size must be odd in each dimension.

By default, `rangefilt` uses the neighborhood `true(3)`. `rangefilt` determines the center element of the neighborhood by `floor((size(NHOOD) + 1)/2)`. For information about specifying neighborhoods, see Notes.

**Class Support** `I` can be logical or numeric and must be real and nonsparse. `NHOOD` can be logical or numeric and must contain zeros or ones.

The output image `J` is the same class as `I`, except for signed integer data types. The output class for signed data types is the corresponding unsigned integer data type. For example, if the class of `I` is `int8`, then the class of `J` is `uint8`.

**Notes** `rangefilt` uses the morphological functions `imdilate` and `imerode` to determine the maximum and minimum values in the specified neighborhood. Consequently, `rangefilt` uses the padding behavior of these morphological functions.

In addition, to specify neighborhoods of various shapes, such as a disk, use the `strel` function to create a structuring element object and

then use the `getnhood` function to extract the neighborhood from the structuring element object.

## Examples

(2-D) Identify the two flying objects by measuring the local range.

```
I = imread('liftingbody.png');
J = rangefilt(I);
imshow(I), figure, imshow(J);
```

(3-D) Quantify land cover changes in an RGB image. The example first converts the image to  $L^*a^*b^*$  colorspace to separate intensity information into a single plane of the image, and then calculates the local range in each layer.

```
I = imread('autumn.tif');
cform = makecform('srgb2lab');
LAB = applycform(I, cform);
rLAB = rangefilt(LAB);
imshow(I);
figure, imshow(rLAB(:,:,1),[]);
figure, imshow(rLAB(:,:,2),[]);
figure, imshow(rLAB(:,:,3),[]);
```

## See Also

`entropyfilt`, `getnhood`, `imdilate`, `imerode`, `stdfilt`, `strel`

# reflect

---

**Purpose** Reflect structuring element

**Syntax** SE2 = reflect(SE)

**Description** SE2 = reflect(SE) reflects a structuring element through its center. The effect is the same as if you rotated the structuring element's domain 180 degrees around its center (for a 2-D structuring element). If SE is an array of structuring element objects, then reflect(SE) reflects each element of SE, and SE2 has the same size as SE.

**Class Support** SE and SE2 are STREL objects.

**Examples**

```
se = strel([0 0 1; 0 0 0; 0 0 0])
se2 = reflect(se)
se =
Flat STREL object containing 1 neighbor.
```

```
Neighborhood:
    0    0    1
    0    0    0
    0    0    0
```

```
se2 =
Flat STREL object containing 1 neighbor.
```

```
Neighborhood:
    0    0    0
    0    0    0
    1    0    0
```

**See Also** strel



**Purpose** Measure properties of image regions (blob analysis)

**Syntax** STATS = regionprops(L,properties)

**Description** STATS = regionprops(L,properties) measures a set of properties 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. The return value STATS is a structure array of length `max(L(:))`. The fields of the structure array denote different measurements for each region, as specified by properties.

properties can be a comma-separated list of strings, a cell array containing strings, the single string 'all', or the string 'basic'. This table lists the set of valid property strings. Property strings are case insensitive and can be abbreviated.

'Area'	'EulerNumber'	'Orientation'
'BoundingBox'	'Extent'	'Perimeter'
'Centroid'	'Extrema'	'PixelIdxList'
'ConvexArea'	'FilledArea'	'PixelList'
'ConvexHull'	'FilledImage'	'Solidity'
'ConvexImage'	'Image'	'SubarrayIdx'
'Eccentricity'	'MajorAxisLength'	
'EquivDiameter'	'MinorAxisLength'	

If properties is the string 'all', regionprops computes all the preceding measurements. If properties is not specified or if it is the string 'basic', regionprops computes only the 'Area', 'Centroid', and 'BoundingBox' measurements.

**Definitions** 'Area' — Scalar; the actual number of pixels in the region. (This value might differ slightly from the value returned by `bwarea`, which weights different patterns of pixels differently.)

# regionprops

---

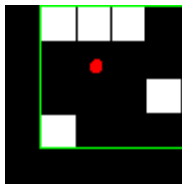
'BoundingBox' — 1-by-ndims(L)\*2 vector; the smallest rectangle containing the region. BoundingBox is [ul\_corner width], where

ul\_corner is in the form [x y z ...] and specifies the upper left corner of the bounding box

width is in the form [x\_width y\_width ...] and specifies the width of the bounding box along each dimension

'Centroid' — 1-by-ndims(L) vector; the center of mass of the region. Note that the first element of Centroid is the horizontal coordinate (or  $x$ -coordinate) of the center of mass, and the second element is the vertical coordinate (or  $y$ -coordinate). All other elements of Centroid are in order of dimension.

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.



'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. This property is supported only for 2-D input label matrices.

'ConvexImage' — Binary image (logical); 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, regionprops 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. This property is supported only for 2-D input label matrices.

'ConvexArea' — Scalar; the number of pixels in 'ConvexImage'. This property is supported only for 2-D input label matrices.

'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.) This property is supported only for 2-D input label matrices.

'EquivDiameter' — Scalar; the diameter of a circle with the same area as the region. Computed as  $\sqrt{4 \cdot \text{Area} / \pi}$ . This property is supported only for 2-D input label matrices.

'EulerNumber' — Scalar; equal to the number of objects in the region minus the number of holes in those objects. This property is supported only for 2-D input label matrices.

'Extent' — Scalar; the proportion of the pixels in the bounding box that are also in the region. Computed as the Area divided by the area of the bounding box. This property is supported only for 2-D input label matrices.

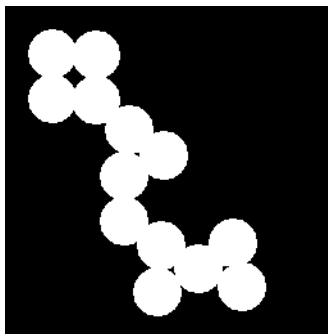
'Extrema' — 8-by-2 matrix; the extrema points in the region. Each row of the matrix contains the  $x$ - and  $y$ -coordinates of one of the points. The format of the vector is [top-left top-right right-top right-bottom bottom-right bottom-left left-bottom left-top]. This property is supported only for 2-D input label matrices.

This figure illustrates the extrema of two different regions. In the region on the left, each extrema point is distinct; in the region on the right, certain extrema points (e.g., top-left and left-top) are identical.



'FilledArea' — Scalar; the number of on pixels in FilledImage.

'FilledImage' — Binary image (logical) of the same size as the bounding box of the region. The on pixels correspond to the region, with all holes filled in.



Original Image, Containing a Single Region



Image Returned

'Image' — Binary image (logical) of the same size as the bounding box of the region; the on pixels correspond to the region, and all other pixels are off.

'MajorAxisLength' — Scalar; the length (in pixels) of the major axis of the ellipse that has the same normalized second central moments as the region. This property is supported only for 2-D input label matrices.

'MinorAxisLength' — Scalar; the length (in pixels) of the minor axis of the ellipse that has the same normalized second central moments as the region. This property is supported only for 2-D input label matrices.

'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 property is supported only for 2-D input label matrices.

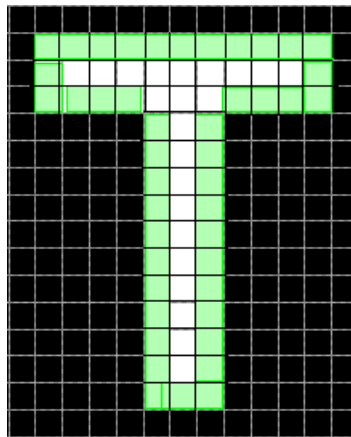
This figure illustrates the axes and orientation of the ellipse. The left side of the figure shows an image region and its corresponding ellipse. The right side shows the same ellipse, with features indicated graphically; the solid blue lines are the axes, the red dots are the foci,

and the orientation is the angle between the horizontal dotted line and



the major axis

'perimeter' —  $p$ -element vector containing the distance around the boundary of each contiguous region in the image, where  $p$  is the number of regions. `regionprops` computes the perimeter by calculating the distance between each adjoining pair of pixels around the border of the region. If the image contains discontinuous regions, `regionprops` returns unexpected results. The following figure shows the pixels included in the perimeter calculation for this object



'PixelIdxList' —  $p$ -element vector containing the linear indices of the pixels in the region.

'PixelList' —  $p$ -by- $\text{ndims}(L)$  matrix; the actual pixels in the region. Each row of the matrix has the form  $[x \ y \ z \ \dots]$  and specifies the coordinates of one pixel in the region.

'Solidity' — Scalar; the proportion of the pixels in the convex hull that are also in the region. Computed as  $\text{Area}/\text{ConvexArea}$ . This property is supported only for 2-D input label matrices.

## Class Support

The input label matrix `L` can have any numeric class.

## Remarks

### Using the Comma-Separated List Syntax

The comma-separated list syntax for structure arrays is very useful when you work with the output of `regionprops`. For example, for a field that contains a scalar, you can use 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 the following 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 = regionprops(L, 'Area');  
allArea = [stats.Area];
```

`allArea` is a vector of the same length as the structure array `stats`.

### Selecting Regions Based on Certain Criteria

The function `ismember` is useful in conjunction with `regionprops` for selecting regions based on certain criteria. For example, these commands create a binary image containing only the regions whose area is greater than 80.

```
idx = find([stats.Area] > 80);
```

```
BW2 = ismember(L,idx);
```

### Performance Considerations

Most of the measurements take very little time to compute. The exceptions are these, which can 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 `regionprops` takes advantage of intermediate computations used in both computations. Therefore, it is fastest to compute all the desired measurements in a single call to `regionprops`.

### Converting a Binary Image into a Label Matrix

You must convert a binary image into a label matrix before calling `regionprops`. There are two common ways to convert a binary image into a label matrix:

- Using the `bwlabel` function

```
L = bwlabel(BW);
```

- Using the `double` function

```
L = double(BW);
```

Note, however, that these functions produce different but equally valid label matrices from the same binary image.

For example, given the following logical matrix, `BW`,

# regionprops

---

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

`bwlabel` creates a label matrix containing two contiguous regions labeled by the integer values 1 and 2.

```
mylabel = bwlabel(BW)
```

```
mylabel =
```

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

The `double` function creates a label matrix containing one discontinuous region labeled by the integer value 1.

```
mylabel2 = double(BW)
```

```
mylabel2 =
```

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

Because each result is legitimately desirable in certain situations, `regionprops` does not attempt to perform either type of conversion on binary images and instead requires that you convert them using either method.



## Examples

Compute the centroids of all the labelled regions in an image and then superimpose a plot of the centroids over the original image.

```
bw = imread('text.png');  
L = bwlabel(bw);  
s = regionprops(L, 'centroid');  
centroids = cat(1, s.Centroid);  
imshow(bw)  
hold on  
plot(centroids(:,1), centroids(:,2), 'b*')  
hold off
```

## See Also

[bwlabel](#), [bwlabeln](#), [ismember](#), [watershed](#)  
[ismember](#) (MATLAB function)

# rgb2gray

---

<b>Purpose</b>	Convert RGB image or colormap to grayscale
<b>Syntax</b>	<pre>I = rgb2gray(RGB) newmap = rgb2gray(map)</pre>
<b>Description</b>	<p>rgb2gray converts RGB images to grayscale by eliminating the hue and saturation information while retaining the luminance.</p> <p>I = rgb2gray(RGB) converts the truecolor image RGB to the grayscale intensity image I.</p> <p>newmap = rgb2gray(map) returns a grayscale colormap equivalent to map.</p>
<b>Class Support</b>	<p>If the input is an RGB image, it can be of class <code>uint8</code>, <code>uint16</code>, <code>single</code>, or <code>double</code>. 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 <code>double</code>.</p>
<b>Examples</b>	<p>Convert an RGB image to a grayscale image.</p> <pre>I = imread('board.tif'); J = rgb2gray(I); figure, imshow(I), figure, imshow(J);</pre> <p>Convert the colormap to a grayscale colormap.</p> <pre>[X,map] = imread('trees.tif'); gmap = rgb2gray(map); figure, imshow(X,map), figure, imshow(X,gmap);</pre>
<b>Algorithm</b>	<p>rgb2gray converts the RGB values to NTSC coordinates, sets the hue and saturation components to zero, and then converts back to RGB color space.</p>
<b>See Also</b>	<code>ind2gray</code> , <code>ntsc2rgb</code> , <code>rgb2ind</code> , <code>rgb2ntsc</code>

**Purpose** Convert RGB values to hue-saturation-value (HSV) color space

**Note** rgb2hsv is a MATLAB function.

# rgb2ind

---

**Purpose** Convert RGB image to indexed image

**Syntax**

```
[X,map] = rgb2ind(RGB,n)
X = rgb2ind( RGB,map)
[X,map] = rgb2ind( RGB,tol)
[...] = rgb2ind(...,dither_option)
```

**Description** `rgb2ind` converts RGB images to indexed images using one of three different methods:

- Uniform quantization
- Minimum variance quantization
- Colormap approximation

For all these methods, `rgb2ind` also dithers the image unless you specify 'nodither' for `dither_option`.

`[X,map] = rgb2ind( RGB,n)` converts the RGB image to an indexed image `X` using minimum variance quantization. `map` contains at most `n` colors. `n` must be less than or equal to 65536.

`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`. `size( map,1)` must be less than or equal to 65536.

`[X,map] = rgb2ind( RGB,tol)` converts the RGB image to an indexed image `X` using uniform quantization. `map` contains at most  $(\text{floor}(1/\text{tol})+1)^3$  colors. `tol` must be between 0.0 and 1.0.

`[...] = 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.

---

**Note** The values in the resultant image  $X$  are indexes into the colormap map and cannot be used in mathematical processing, such as filtering operations.

---

## Class Support

The input image can be of class `uint8`, `uint16`, `single`, or `double`. If the length of map is less than or equal to 256, the output image is of class `uint8`. Otherwise, the output image is of class `uint16`.

## Remarks

If you specify `tol`, `rgb2ind` uses uniform quantization to convert the image. This method involves cutting the RGB color cube into smaller cubes of length `tol`. For example, if you specify a `tol` of 0.1, the edges of the cubes are one-tenth the length of the RGB cube. The total number of small cubes is

$$n = (\text{floor}(1/\text{tol})+1)^3$$

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 can 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.

## Examples

```
RGB = imread('peppers.png');  
[X,map] = rgb2ind(RGB,128);  
imshow(X,map)
```



**See Also**

`cmunique`, `dither`, `imapprox`, `ind2rgb`, `rgb2gray`

**Purpose** Convert RGB color values to NTSC color space

**Syntax** `yiormap = rgb2ntsc(rgbmap)`  
`YIQ = rgb2ntsc(RGB)`

**Description** `yiormap = rgb2ntsc(rgbmap)` converts the m-by-3 RGB values in `rgbmap` to NTSC color space. `yiormap` 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.

**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.

`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** RGB can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. `RGBMAP` can be `double`. The output is `double`.

**See Also** `ntsc2rgb`, `rgb2ind`, `ind2rgb`, `ind2gray`

For a full list of the toolbox color space conversion functions, see “Color Space Conversions” on page 16-21.

# rgb2ycbcr

---

**Purpose** Convert RGB color values to YCbCr color space

**Syntax** `ycbcrmap = rgb2ycbcr(map)`  
`YCBCR = rgb2ycbcr(RGB)`

**Description** `ycbcrmap = rgb2ycbcr(map)` converts the RGB values in `map` to the YCbCr color space. `map` must be an M-by-3 array. `ycbcrmap` is an M-by-3 matrix that contains the YCbCr luminance (*Y*) and chrominance (*Cb* and *Cr*) color values as columns. Each row in `ycbcrmap` represents the equivalent color to the corresponding row in the RGB colormap, `map`.

`YCBCR = rgb2ycbcr(RGB)` converts the truecolor image RGB to the equivalent image in the YCbCr color space. `RGB` must be a M-by-N-by-3 array.

If the input is `uint8`, `YCBCR` is `uint8`, where *Y* is in the range [16 235], and *Cb* and *Cr* are in the range [16 240]. If the input is a `double`, *Y* is in the range [16/255 235/255] and *Cb* and *Cr* are in the range [16/255 240/255]. If the input is `uint16`, *Y* is in the range [4112 60395] and *Cb* and *Cr* are in the range [4112 61680].

**Class Support** If the input is an RGB image, it can be of class `uint8`, `uint16`, or `double`. If the input is a colormap, it must be `double`. The output image is of the same class as the input image.

**Examples** Convert RGB image to YCbCr.

```
RGB = imread('board.tif');  
YCBCR = rgb2ycbcr(RGB);
```

Convert RGB color space to YCbCr.

```
map = jet(256);  
newmap = rgb2ycbcr(map);
```

**See Also** `ntsc2rgb`, `rgb2ntsc`, `ycbcr2rgb`



For a full list of the toolbox color space conversion functions, see “Color Space Conversions” on page 16-21.

**References**

[1] Poynton, C. A. *Technical Introduction to Digital Video*, John Wiley & Sons, Inc., 1996, p. 175.

[2] Rec. ITU-R BT.601-5, *Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-screen 16:9 Aspect Ratios*, (1982-1986-1990-1992-1994-1995), Section 3.5.

# rgbplot

---

**Purpose**            Plot colormap

**Note**             rgbplot is a MATLAB function.

**Purpose** Select region of interest (ROI) 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]`.

$$BW = (A \geq low) \ \& \ (A \leq high)$$

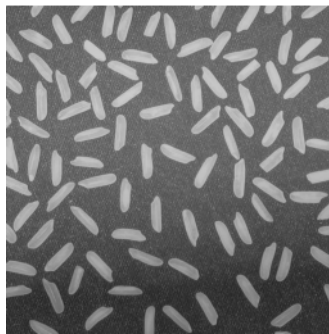
`BW` is a binary image with 0's outside the region of interest and 1's inside.

`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 image `A` must be numeric. The output image `BW` is of class `logical`.

**Examples**

```
I = imread('rice.png');  
BW = roicolor(I,128,255);  
imshow(I);  
figure, imshow(BW)
```



## See Also

`roifilt2`, `roipoly`

**Purpose** Fill in specified polygon in grayscale image

**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 a grayscale image. `roifill` 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 be of class `uint8`, `uint16`, `int16`, `single`, or `double`. The input binary mask `BW` can be any numeric class or `logical`. The output binary mask `BW` is always `logical`. The output image `J` is of the same class as `I`. All other inputs and outputs are of class `double`.

## Examples

```
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`

**Purpose**

Filter region of interest in image

**Syntax**

```
J = roifilt2(h,I,BW)
J = roifilt2(I,BW,fun)
```

**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` must be a function handle.

**Class Support**

For the syntax that includes a filter `h`, the input image can be logical or numeric, and the output array `J` has the same class as the input image. 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`.

**Examples**

This example continues the `roipoly` example, filtering the region of the image `I` specified by the mask `BW`. The `roifilt2` function returns the filtered image `J`, shown in the following figure.

```
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);
figure, imshow(I), figure, imshow(J)
```

## roifilt2

---



### See Also

`imfilter`, `filter2`, `function_handle`, `roipoly`



<b>Purpose</b>	Specify polygonal region of interest
<b>Syntax</b>	<pre>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(...)</pre>
<b>Description</b>	<p>Use <code>roipoly</code> to specify a polygonal region of interest within an image. <code>roipoly</code> returns a binary image that you can use as a mask for masked filtering.</p> <p><code>BW = roipoly(I,c,r)</code> returns the region of interest selected by the polygon described by vectors <code>c</code> and <code>r</code>. <code>BW</code> is a binary image the same size as <code>I</code> with 0's outside the region of interest and 1's inside.</p> <p><code>BW = roipoly(I)</code> displays the image <code>I</code> on the screen and lets you specify the polygon using the mouse. If you omit <code>I</code>, <code>roipoly</code> operates on the image in the current axes. Use normal button clicks to add vertices to the polygon. Pressing <b>Backspace</b> or <b>Delete</b> 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 <b>Return</b> finishes the selection without adding a vertex.</p> <p><code>BW = roipoly(x,y,I,xi,yi)</code> uses the vectors <code>x</code> and <code>y</code> to establish a nondefault spatial coordinate system. <code>xi</code> and <code>yi</code> are equal-length vectors that specify polygon vertices as locations in this coordinate system.</p> <p><code>[BW,xi,yi] = roipoly(...)</code> returns the polygon coordinates in <code>xi</code> and <code>yi</code>. Note that <code>roipoly</code> always produces a closed polygon. If the points specified describe a closed polygon (i.e., if the last pair of coordinates is identical to the first pair), the length of <code>xi</code> and <code>yi</code> is equal to the number of points specified. If the points specified do not describe a closed polygon, <code>roipoly</code> adds a final point having the same coordinates as the first point. (In this case the length of <code>xi</code> and <code>yi</code> is one greater than the number of points specified.)</p>

`[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`, `int16`, `single`, or `double`. The output image `BW` is of class `logical`. All other inputs and outputs are of class `double`.

## Remarks

For any of the `roipoly` syntax, 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.

## Examples

Use `roipoly` to create a mask image, `BW`, the same size as the input image, `I`. The example in `roifilt2` continues this example, filtering the specified region in the image.

```
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`, `poly2mask`

# std2

---

<b>Purpose</b>	Standard deviation of matrix elements
<b>Syntax</b>	<code>b = std2(A)</code>
<b>Description</b>	<code>b = std2(A)</code> computes the standard deviation of the values in A.
<b>Class Support</b>	A can be numeric or logical. B is a scalar of class double.
<b>Algorithm</b>	<code>std2</code> computes the standard deviation of the array A using <code>std(A(:))</code> .
<b>See Also</b>	<code>corr2</code> , <code>mean2</code> <code>std</code> , <code>mean</code> in the MATLAB Function Reference

---

<b>Purpose</b>	Local standard deviation of image
<b>Syntax</b>	<pre>J = stdfilt(I) J = stdfilt(I,NHOOD)</pre>
<b>Description</b>	<p><code>J = stdfilt(I)</code> returns the array <code>J</code>, where each output pixel contains the standard deviation of the 3-by-3 neighborhood around the corresponding pixel in the input image <code>I</code>. <code>I</code> can have any dimension. The output image <code>J</code> is the same size as the input image <code>I</code>.</p> <p>For pixels on the borders of <code>I</code>, <code>stdfilt</code> uses symmetric padding. In symmetric padding, the values of padding pixels are a mirror reflection of the border pixels in <code>I</code>.</p> <p><code>J = stdfilt(I,NHOOD)</code> calculates the local standard deviation of the input image <code>I</code>, where you specify the neighborhood in <code>NHOOD</code>. <code>NHOOD</code> is a multidimensional array of zeros and ones where the nonzero elements specify the neighbors. <code>NHOOD</code>'s size must be odd in each dimension.</p> <p>By default, <code>stdfilt</code> uses the neighborhood <code>ones(3)</code>. <code>stdfilt</code> determines the center element of the neighborhood by <code>floor((size(NHOOD) + 1)/2)</code>.</p>
<b>Class Support</b>	<code>I</code> can be logical or numeric and must be real and nonsparse. <code>NHOOD</code> can be logical or numeric and must contain zeros and/or ones. <code>J</code> is of class <code>double</code> .
<b>Notes</b>	To specify neighborhoods of various shapes, such as a disk, use the <code>strel</code> function to create a structuring element object and then use the <code>getnhood</code> function to extract the neighborhood from the structuring element object.
<b>Examples</b>	<pre>I = imread('circuit.tif'); J = stdfilt(I); imshow(I); figure, imshow(J,[]);</pre>
<b>See also</b>	<code>entropyfilt</code> , <code>getnhood</code> , <code>rangefilt</code> , <code>std2</code> , <code>strel</code>

# strel

**Purpose** Create morphological structuring element (STREL)


**Syntax** SE = strel(shape,parameters)

**Description** SE = strel(shape,parameters) creates a structuring element, SE, of the type specified by shape. This table lists all the supported shapes. Depending on shape, strel can take additional parameters. See the syntax descriptions that follow for details about creating each type of structuring element.

Flat Structuring Elements	
'arbitrary'	'pair'
'diamond'	'periodicline'
'disk'	'rectangle'
'line'	'square'
'octagon'	

Nonflat Structuring Elements	
'arbitrary'	'ball'

SE = strel('arbitrary',NHOOD) creates a flat structuring element where NHOOD specifies the neighborhood. NHOOD is a matrix containing 1's and 0's; the location of the 1's defines the neighborhood for the morphological operation. The center (or *origin*) of NHOOD is its center element, given by  $\text{floor}((\text{size}(\text{NHOOD})+1)/2)$ . You can omit the 'arbitrary' string and just use strel(NHOOD).

SE =   
NHOOD = [ 1 0 0; 1 0 0; 1 0 1];

`SE = strel('arbitrary',NHOOD,HEIGHT)` creates a nonflat structuring element, where `NHOOD` specifies the neighborhood. `HEIGHT` is a matrix the same size as `NHOOD` containing the height values associated with each nonzero element of `NHOOD`. The `HEIGHT` matrix must be real and finite valued. You can omit the 'arbitrary' string and just use `strel(NHOOD,HEIGHT)`.

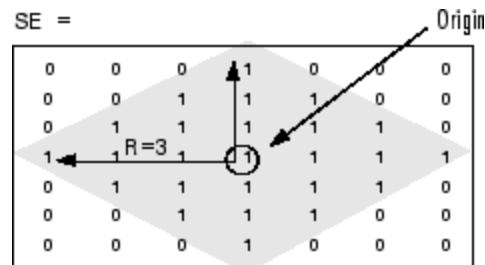
`SE = strel('ball',R,H,N)` creates a nonflat, ball-shaped structuring element (actually an ellipsoid) whose radius in the X-Y plane is `R` and whose height is `H`. Note that `R` must be a nonnegative integer, `H` must be a real scalar, and `N` must be an even nonnegative integer. When `N` is greater than 0, the ball-shaped structuring element is approximated by a sequence of `N` nonflat, line-shaped structuring elements. When `N` equals 0, no approximation is used, and the structuring element members consist of all pixels whose centers are no greater than `R` away from the origin. The corresponding height values are determined from the formula of the ellipsoid specified by `R` and `H`. If `N` is not specified, the default value is 8.

---

**Note** Morphological operations run much faster when the structuring element uses approximations ( $N > 0$ ) than when it does not ( $N = 0$ ).

---

`SE = strel('diamond',R)` creates a flat, diamond-shaped structuring element, where `R` specifies the distance from the structuring element origin to the points of the diamond. `R` must be a nonnegative integer scalar.

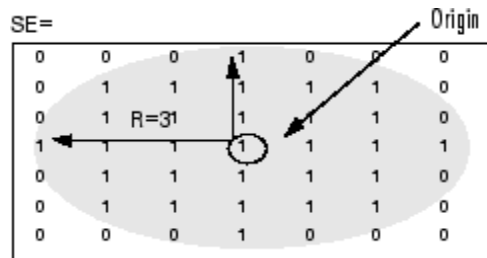


`SE = strel('disk',R,N)` creates a flat, disk-shaped structuring element, where `R` specifies the radius. `R` must be a nonnegative integer. `N` must be 0, 4, 6, or 8. When `N` is greater than 0, the disk-shaped structuring element is approximated by a sequence of `N` periodic-line structuring elements. When `N` equals 0, no approximation is used, and the structuring element members consist of all pixels whose centers are no greater than `R` away from the origin. If `N` is not specified, the default value is 4.

---

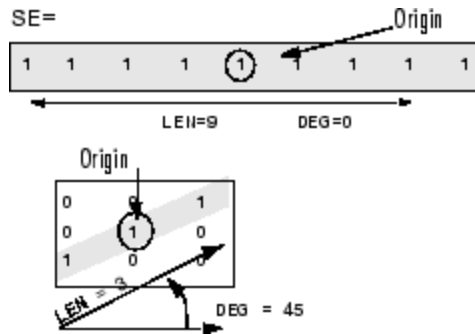
**Note** Morphological operations run much faster when the structuring element uses approximations ( $N > 0$ ) than when it does not ( $N = 0$ ). However, structuring elements that do not use approximations ( $N = 0$ ) are not suitable for computing granulometries. Sometimes it is necessary for `strel` to use two extra line structuring elements in the approximation, in which case the number of decomposed structuring elements used is  $N + 2$ .

---

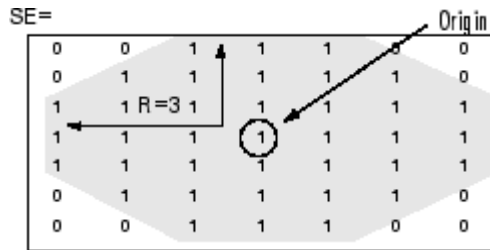


`SE = strel('line',LEN,DEG)` creates a flat, linear structuring element, where `LEN` specifies the length, and `DEG` specifies the angle (in degrees) of the line, as measured in a counterclockwise direction from the horizontal axis. `LEN` is approximately the distance between the centers of the structuring element members at opposite ends of the line.

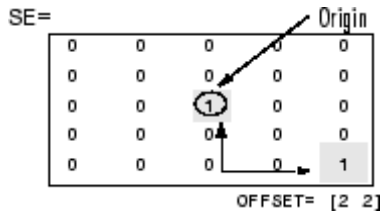




SE = strel('octagon',R) creates a flat, octagonal structuring element, where R specifies the distance from the structuring element origin to the sides of the octagon, as measured along the horizontal and vertical axes. R must be a nonnegative multiple of 3.

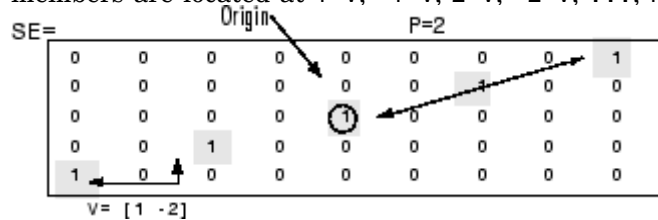


SE = strel('pair',OFFSET) creates a flat structuring element containing two members. One member is located at the origin. The second member's location is specified by the vector OFFSET. OFFSET must be a two-element vector of integers.

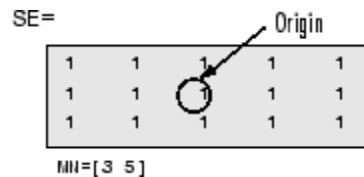


# strel

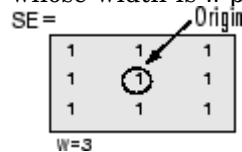
`SE = strel('periodicline',P,V)` creates a flat structuring element containing  $2*P+1$  members.  $V$  is a two-element vector containing integer-valued row and column offsets. One structuring element member is located at the origin. The other members are located at  $1*V, -1*V, 2*V, -2*V, \dots, P*V, -P*V$ .



`SE = strel('rectangle',MN)` creates a flat, rectangle-shaped structuring element, where  $MN$  specifies the size.  $MN$  must be a two-element vector of nonnegative integers. The first element of  $MN$  is the number of rows in the structuring element neighborhood; the second element is the number of columns.



`SE = strel('square',W)` creates a square structuring element whose width is  $W$  pixels.  $W$  must be a nonnegative integer scalar.



## Notes

For all shapes except 'arbitrary', structuring elements are constructed using a family of techniques known collectively as *structuring element decomposition*. The principle is that dilation by some large structuring elements can be computed faster by dilation with a sequence of smaller structuring elements. For example, dilation by an 11-by-11 square

structuring element can be accomplished by dilating first with a 1-by-1 structuring element and then with an 11-by-1 structuring element. This results in a theoretical performance improvement of a factor of 5.5, although in practice the actual performance improvement is somewhat less. Structuring element decompositions used for the 'disk' and 'ball' shapes are approximations; all other decompositions are exact.

## Methods

This table lists the methods supported by the STREL object.

Method	Description
getheight	Get height of structuring element
getneighbors	Get structuring element neighbor locations and heights
getnhood	Get structuring element neighborhood
getsequence	Extract sequence of decomposed structuring elements
isflat	Return true for flat structuring element
reflect	Reflect structuring element
translate	Translate structuring element

## Examples

```
se1 = strel('square',11)      % 11-by-11 square
se2 = strel('line',10,45)    % length 10, angle 45 degrees
se3 = strel('disk',15)      % disk, radius 15
se4 = strel('ball',15,5)    % ball, radius 15, height 5
```

## Algorithm

The method used to decompose diamond-shaped structuring elements is known as "logarithmic decomposition" [1].

The method used to decompose disk structuring elements is based on the technique called "radial decomposition using periodic lines" [2], [3]. For details, see the MakeDiskStrel subfunction in toolbox/images/images/@strel/strel.m.

The method used to decompose ball structuring elements is the technique called "radial decomposition of sphere" [2].

## See Also

imdilate, imerode

## References

[1] van den Boomgard, Rein, and Richard van Balen, "Methods for Fast Morphological Image Transforms Using Bitmapped Images," *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, Vol. 54, No. 3, May 1992, pp. 252-254.

[2] Adams, Rolf, "Radial Decomposition of Discs and Spheres," *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, Vol. 55, No. 5, September 1993, pp. 325-332.

[3] Jones, Ronald, and Pierre Soille, "Periodic lines: Definition, cascades, and application to granulometrie," *Pattern Recognition Letters*, Vol. 17, 1996, pp. 1057-1063.

**Purpose** Find limits to contrast stretch image

**Syntax**

```
LOW_HIGH = stretchlim(I)
LOW_HIGH = stretchlim(I,TOL)
LOW_HIGH = stretchlim(RGB,TOL)
```

**Description** `LOW_HIGH = stretchlim(I)` returns `LOW_HIGH`, a two-element vector of pixel values that specify lower and upper limits that can be used for contrast stretching image `I`. By default, values in `LOW_HIGH` specify the bottom 1% and the top 1% of all pixel values. The gray values returned can be used by the `imadjust` function to increase the contrast of an image.

`LOW_HIGH = stretchlim(I,TOL)` where `TOL` is a two-element vector [`LOW_FRACT HIGH_FRACT`] that specifies the fraction of the image to saturate at low and high pixel values.

If `TOL` is a scalar, `LOW_FRACT = TOL`, and `HIGH_FRACT = 1 - LOW_FRACT`, which saturates equal fractions at low and high pixel values.

If you omit the argument, `TOL` defaults to `[0.01 0.99]`, saturating 2%.

If `TOL = 0`, `LOW_HIGH = [min(I(:)); max(I(:))]`.

`LOW_HIGH = stretchlim(RGB,TOL)` returns a 2-by-3 matrix of intensity pairs to saturate each plane of the RGB image. `TOL` specifies the same fractions of saturation for each plane.

---

**Note** If `TOL` is too big, such that no pixels would be left after saturating low and high pixel values, `stretchlim` returns `[0 1]`.

---

**Class Support** The input image can be of class `uint8`, `uint16`, `int16`, `double`, or `single`. The output limits returned, `LOW_HIGH`, are of class `double` and have values between 0 and 1.

# stretchlim

---

## Examples

```
I = imread('pout.tif');  
J = imadjust(I,stretchlim(I),[]);  
imshow(I), figure, imshow(J)
```



## See Also

brighten, histeq, imadjust

**Purpose** Display multiple images in single 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 true color 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 `logical`, `uint8`, `uint16`, or `double`.

**Examples**

```
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

# tformarray

---

**Purpose** Apply spatial transformation to N-D array

**Syntax** `B = tformarray(A,T,R,TDIMS_A,TDIMS_B,TSIZE_B,TMAP_B,F)`

**Description** `B = tformarray(A,T,R,TDIMS_A,TDIMS_B,TSIZE_B,TMAP_B,F)` applies a spatial transformation to array A to produce array B. The `tformarray` function is like `imtransform`, but is intended for problems involving higher-dimensioned arrays or mixed input/output dimensionality, or requiring greater user control or customization. (Anything that can be accomplished with `imtransform` can be accomplished with a combination of `maketform`, `makeresampler`, `findbounds`, and `tformarray`; but for many tasks involving 2-D images, `imtransform` is simpler.)

This table provides a brief description of all the input arguments. See the following section for more detail about each argument. (Click an argument in the table to move to the appropriate section.)

Argument	Description
A	Input array or image
T	Spatial transformation structure, called a TFORM, typically created with <code>maketform</code>
R	Resampler structure, typically created with <code>makeresampler</code>
TDIMS_A	Row vector listing the input transform dimensions
TDIMS_B	Row vector listing the output transform dimensions
TSIZE_B	Output array size in the transform dimensions
TMAP_B	Array of point locations in output space; can be used as an alternative way to specify a spatial transformation
F	Array of fill values

A can be any nonsparse numeric array, and can be real or complex.



T is a TFORM structure that defines a particular spatial transformation. For each location in the output transform subscript space (as defined by TDIMS\_B and TSIZE\_B), tformarray uses T and the function tforminv to compute the corresponding location in the input transform subscript space (as defined by TDIMS\_A and size(A)).

If T is empty, tformarray operates as a direct resampling function, applying the resampler defined in R to compute values at each transform space location defined in TMAP\_B (if TMAP\_B is nonempty), or at each location in the output transform subscript grid.

R is a structure that defines how to interpolate values of the input array at specified locations. R is usually created with makeresampler, which allows fine control over how to interpolate along each dimension, as well as what input array values to use when interpolating close to the edge of the array.

TDIMS\_A and TDIMS\_B indicate which dimensions of the input and output arrays are involved in the spatial transformation. Each element must be unique, and must be a positive integer. The entries need not be listed in increasing order, but the order matters. It specifies the precise correspondence between dimensions of arrays A and B and the input and output spaces of the transformer T. length(TDIMS\_A) must equal T.ndims\_in, and length(TDIMS\_B) must equal T.ndims\_out.

For example, if T is a 2-D transformation, TDIMS\_A = [2 1], and TDIMS\_B = [1 2], then the column dimension and row dimension of A correspond to the first and second transformation input-space dimensions, respectively. The row and column dimensions of B correspond to the first and second output-space dimensions, respectively.

TSIZE\_B specifies the size of the array B along the output-space transform dimensions. Note that the size of B along nontransform dimensions is taken directly from the size of A along those dimensions. If, for example, T is a 2-D transformation, size(A) = [480 640 3 10], TDIMS\_B is [2 1], and TSIZE\_B is [300 200], then size(B) is [200 300 3].

TMAP\_B is an optional array that provides an alternative way of specifying the correspondence between the position of elements of B

and the location in output transform space. `TMAP_B` can be used, for example, to compute the result of an image warp at a set of arbitrary locations in output space. If `TMAP_B` is not empty, then the size of `TMAP_B` takes the form

$$[D1 \ D2 \ D3 \ \dots \ DN \ L]$$

where `N` equals `length(TDIMS_B)`. The vector `[D1 D2 ... DN]` is used in place of `TSIZE_B`. If `TMAP_B` is not empty, then `TSIZE_B` should be `[]`.

The value of `L` depends on whether or not `T` is empty. If `T` is not empty, then `L` is `T.ndims_out`, and each `L`-dimension point in `TMAP_B` is transformed to an input-space location using `T`. If `T` is empty, then `L` is `length(TDIMS_A)`, and each `L`-dimensional point in `TMAP_B` is used directly as a location in input space.

`F` is a double-precision array containing fill values. The fill values in `F` can be used in three situations:

- When a separable resampler is created with `makeresampler` and its `padmethod` is set to either `'fill'` or `'bound'`.
- When a custom resampler is used that supports the `'fill'` or `'bound'` pad methods (with behavior that is specific to the customization).
- When the map from the transform dimensions of `B` to the transform dimensions of `A` is deliberately undefined for some points. Such points are encoded in the input transform space by NaNs in either `TMAP_B` or in the output of `TFORMINV`.

In the first two cases, fill values are used to compute values for output locations that map outside or near the edges of the input array. Fill values are copied into `B` when output locations map well outside the input array. See `makeresampler` for more information about `'fill'` and `'bound'`.

`F` can be a scalar (including NaN), in which case its value is replicated across all the nontransform dimensions. `F` can also be a nonscalar, whose size depends on `size(A)` in the nontransform dimensions. Specifically,

if  $K$  is the  $J$ th nontransform dimension of  $A$ , then  $\text{size}(F, J)$  must be either  $\text{size}(A, K)$  or 1. As a convenience to the user, `tformarray` replicates  $F$  across any dimensions with unit size such that after the replication  $\text{size}(F, J)$  equals  $\text{size}(A, K)$ .

For example, suppose  $A$  represents 10 RGB images and has size 200-by-200-by-3-by-10,  $T$  is a 2-D transformation, and `TDIMS_A` and `TDIMS_B` are both `[1 2]`. In other words, `tformarray` will apply the same 2-D transform to each color plane of each of the 10 RGB images. In this situation you have several options for  $F$ :

- $F$  can be a scalar, in which case the same fill value is used for each color plane of all 10 images.
- $F$  can be a 3-by-1 vector, `[R G B]'`. Then  $R$ ,  $G$ , and  $B$  are used as the fill values for the corresponding color planes of each of the 10 images. This can be interpreted as specifying an RGB fill color, with the same color used for all 10 images.
- $F$  can be a 1-by-10 vector. This can be interpreted as specifying a different fill value for each of 10 images, with that fill value being used for all three color planes.
- $F$  can be a 3-by-10 matrix, which can be interpreted as supplying a different RGB fill color for each of the 10 images.

## Class Support

$A$  can be any nonsparse numeric array, and can be real or complex. It can also be of class `logical`.

## Examples

Create a 2-by-2 checkerboard image where each square is 20 pixels wide, then transform it with a projective transformation. Use a `pad` method of `'circular'` when creating a resampler, so that the output appears to be a perspective view of an infinite checkerboard. Swap the output dimensions. Specify a 100-by-100 output image. Leave `TMAP_B` empty, since `TSIZE_B` is specified. Leave the fill value empty, since it won't be needed.

```
I = checkerboard(20,1,1);
figure; imshow(I)
```

# tformarray

---

```
T = maketform('projective',[1 1; 41 1; 41 41; 1 41],...
              [5 5; 40 5; 35 30; -10 30]);
R = makesampler('cubic','circular');
J = tformarray(I,T,R,[1 2],[2 1],[100 100],[],[]);
figure; imshow(J)
```

## See Also

findbounds, imtransform, makesampler, maketform

**Purpose**

Apply forward spatial transformation

**Syntax**

```
[X,Y] = tformfwd(T,U,V)
[X1,X2,X3,...] = tformfwd(T,U1,U2,U3,...)
X = tformfwd(T,U)
[X1,X2,X3,...] = tformfwd(T,U)
X = tformfwd(T,U1,U2,U3,...)
```

**Description**

`[X,Y] = tformfwd(T,U,V)` applies the 2D-to-2D spatial transformation defined in `T` to coordinate arrays `U` and `V`, mapping the point `[U(k) V(k)]` to the point `[X(k) Y(k)]`.

`T` is a `TFORM` struct created with `maketform`, `fliptform`, or `cp2tform`. Both `T.ndims_in` and `T.ndims_out` must equal 2. `U` and `V` are typically column vectors matching in length. In general, `U` and `V` can have any dimensionality, but must have the same size. In any case, `X` and `Y` will have the same size as `U` and `V`.

`[X1,X2,X3,...] = tformfwd(T,U1,U2,U3,...)` applies the `ndims_in`-to-`ndims_out` spatial transformation defined in `TFORM` structure `T` to the coordinate arrays `U1,U2,...,UNDIMS_IN` (where `NDIMS_IN = T.ndims_in` and `NDIMS_OUT = T.ndims_out`). The number of output arguments must equal `NDIMS_OUT`. The transformation maps the point

$$[U1(k) \ U2(k) \ \dots \ UNDIMS\_IN(k)]$$

to the point

$$[X1(k) \ X2(k) \ \dots \ XNDIMS\_OUT(k)].$$

`U1,U2,U3,...` can have any dimensionality, but must be the same size.

`X1,X2,X3,...` must have this size also.

`X = tformfwd(T,U)` applies the `ndims_in`-to-`ndims_out` spatial transformation defined in `TFORM` structure `T` to each row of `U`, where `U` is an `M`-by-`NDIMS_IN` matrix. It maps the point `U(k,:)` to the point `X(k,:)`. `X` is an `M`-by-`NDIMS_OUT` matrix.

# tformfwd

---

$X = \text{tformfwd}(T,U)$ , where  $U$  is an  $(N+1)$ -dimensional array, maps the point  $U(k_1, k_2, \dots, k_N, :)$  to the point  $X(k_1, k_2, \dots, k_N, :)$ .  $\text{size}(U, N+1)$  must equal  $\text{NDIMS\_IN}$ .  $X$  is an  $(N+1)$ -dimensional array, with  $\text{size}(X, I)$  equal to  $\text{size}(U, I)$  for  $I = 1, \dots, N$  and  $\text{size}(X, N+1)$  equal to  $\text{NDIMS\_OUT}$ .

$[X_1, X_2, X_3, \dots] = \text{tformfwd}(T, U)$  maps an  $(N+1)$ -dimensional array to  $\text{NDIMS\_OUT}$  equally sized  $N$ -dimensional arrays.

$X = \text{tformfwd}(T, U_1, U_2, U_3, \dots)$  maps  $\text{NDIMS\_IN}$   $N$ -dimensional arrays to one  $(N+1)$ -dimensional array.

## Note

$X = \text{tformfwd}(U, T)$  is an older form of the two-argument syntax that remains supported for backward compatibility.

## Examples

Create an affine transformation that maps the triangle with vertices  $(0,0)$ ,  $(6,3)$ ,  $(-2,5)$  to the triangle with vertices  $(-1,-1)$ ,  $(0,-10)$ ,  $(4,4)$ .

```
u = [ 0  6 -2]';  
v = [ 0  3  5]';  
x = [-1  0  4]';  
y = [-1 -10  4]';  
tform = maketform('affine', [u v], [x y]);
```

Validate the mapping by applying `tformfwd`. Results should equal  $[x, y]$

```
[xm, ym] = tformfwd(tform, u, v)
```

## See Also

`cp2tform`, `fliptform`, `maketform`, `tforminv`

**Purpose** Apply inverse spatial transformation

**Syntax** `U = tforminv(X,T)`

**Description** `[U,V] = tforminv(T,X,Y)` applies the 2D-to-2D inverse transformation defined in TFORM structure T to coordinate arrays X and Y, mapping the point `[X(k) Y(k)]` to the point `[U(k) V(k)]`. Both `T.ndims_in` and `T.ndims_out` must equal 2. X and Y are typically column vectors matching in length. In general, X and Y can have any dimensionality, but must have the same size. In any case, U and V will have the same size as X and Y.

`[U1,U2,U3,...] = tforminv(T,X1,X2,X3,...)` applies the NDIMS\_OUT-to-NDIMS\_IN inverse transformation defined in TFORM structure T to the coordinate arrays `X1,X2,...,XNDIMS_OUT` (where `NDIMS_IN = T.ndims_in` and `NDIMS_OUT = T.ndims_out`). The number of output arguments must equal NDIMS\_IN. The transformation maps the point

$$[X1(k) \ X2(k) \ \dots \ XNDIMS\_OUT(k)]$$

to the point

$$[U1(k) \ U2(k) \ \dots \ UNDIMS\_IN(k)].$$

`X1,X2,X3,...` can have any dimensionality, but must be the same size.

`U1,U2,U3,...` have this size also.

`U = tforminv(T,X)` applies the NDIMS\_OUT-to-NDIMS\_IN inverse transformation defined in TFORM structure T to each row of X, where X is an M-by-NDIMS\_OUT matrix. It maps the point `X(k,:) = [X(k,1) X(k,2) ... X(k,NDIMS_OUT)]` to the point `U(k,:) = [U(k,1) U(k,2) ... U(k,NDIMS_IN)]`. U is an M-by-NDIMS\_IN matrix.

`U = tforminv(T,X)`, where X is an (N+1)-dimensional array, maps the point `X(k1,k2,...,kN,:) = [X(k1,k2,...,kN,1) X(k1,k2,...,kN,2) ... X(k1,k2,...,kN,N+1)]` to the point `U(k1,k2,...,kN,:) = [U(k1,k2,...,kN,1) U(k1,k2,...,kN,2) ... U(k1,k2,...,kN,NDIMS_IN)]`. `size(X,N+1)` must equal NDIMS\_OUT. U is an (N+1)-dimensional array, with `size(U,I)` equal to `size(X,I)` for `I = 1,...,N` and `size(U,N+1)` equal to NDIMS\_IN.

# tforminv

---

`[U1,U2,U3,...] = tforminv(T,X)` maps an (N+1)-dimensional array to NDIMS\_IN equally-sized N-dimensional arrays.

`U = tforminv(T,X1,X2,X3,...)` maps NDIMS\_OUT N-dimensional arrays to one (N+1)-dimensional array.

## Note

`U = tforminv(X,T)` is an older form of the two-argument syntax that remains supported for backward compatibility.

## Examples

Create an affine transformation that maps the triangle with vertices (0,0), (6,3), (-2,5) to the triangle with vertices (-1,-1), (0,-10), (4,4).

```
u = [ 0  6 -2]';  
v = [ 0  3  5]';  
x = [-1  0  4]';  
y = [-1 -10  4]';  
tform = maketform('affine',[u v],[x y]);
```

Validate the mapping by applying `tforminv`. Results should equal `[u, v]`.

```
[um, vm] = tforminv(tform, x, y)
```

## See Also

`cp2tform`, `tforminv`, `maketform`, `fliptform`



---

<b>Purpose</b>	Translate structuring element (STREL)
<b>Syntax</b>	<code>SE2 = translate(SE,V)</code>
<b>Description</b>	<p><code>SE2 = translate(SE,V)</code> translates the structuring element SE in N-D space. SE is an array of structuring elements, created using the <code>strel</code> function.</p> <p>V is an N-element vector that specifies the offsets of the desired translation in each dimension, relative to the structuring element's origin. If you specify an array, <code>translate</code> reshapes the array into a vector.</p> <p>SE2 is an array of structuring elements the same size as SE. Each individual structuring element in SE2 is the translation of the corresponding structuring element in SE.</p>
<b>Class Support</b>	SE and SE2 are STREL objects; V is a vector of doubles that must contain only integers.
<b>Examples</b>	<p>Translate a 3-by-3 structuring element.</p> <pre>se = strel(ones(3))  se2 = translate(se,[-2 -2])</pre> <p>The following figure shows the original structuring element and the translated structuring element.</p>



---

<b>Purpose</b>	Adjust display size of image
<b>Syntax</b>	<code>truesize(fig,[mrows mcols])</code> <code>truesize(fig)</code>
<b>Description</b>	<p><code>truesize(fig,[mrows mcols])</code> adjusts the display size of an image. <code>fig</code> is a figure containing a single image or a single image with a colorbar. <code>[MROWS MCOLS]</code> is a 1-by-2 vector that specifies the requested screen area (in pixels) that the image should occupy.</p> <p><code>truesize(fig)</code> uses the image height and width for <code>[MROWS MCOLS]</code>. This results in the display having one screen pixel for each image pixel.</p> <p>If you do not specify a figure, <code>truesize</code> uses the current figure.</p>
<b>Examples</b>	<p>Fit image to figure window.</p> <pre>imshow(checkerboard,'InitialMagnification','fit')</pre> <p>Resize image and figure to show image at its 80-by-80 pixel size.</p> <pre>truesize</pre>
<b>See Also</b>	<code>imshow</code> , <code>iptsetpref</code> , <code>iptgetpref</code>

# uint16

---

**Purpose** Convert data to unsigned 16-bit integers

**Note** `uint16` is a MATLAB built-in function.

**Purpose** Convert data to unsigned 8-bit integers

**Note** `uint8` is a MATLAB built-in function.

# uintlut

---

**Purpose** Compute new values of A based on lookup table (LUT)

**Syntax** B = uintlut(A,LUT)

---

**Note** uintlut is an obsolete version of intlut and may be removed in a future version of the toolbox.

---

**Class Support** A must be uint8 or uint16. If A is uint8, then LUT must be a uint8 vector with 256 elements. If A is uint16, then LUT must be a uint16 vector with 65536 elements. B has the same size and class as A.

**Examples**

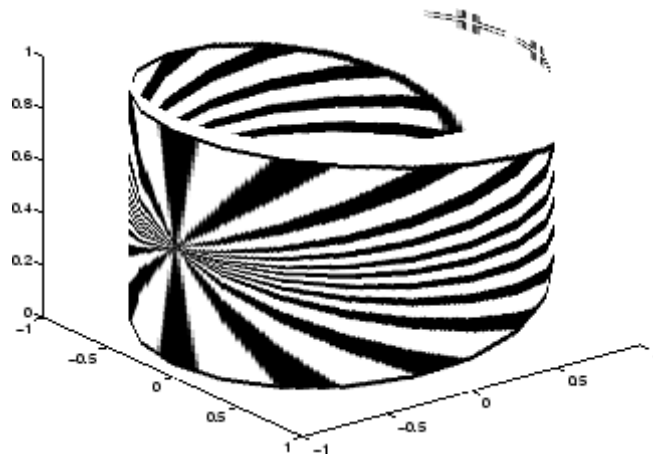
```
A = uint8([1 2 3 4; 5 6 7 8;9 10 11 12]);
LUT = repmat(uint8([0 150 200 255]),1,64);
B = uintlut(A,LUT);
imshow(A), figure, imshow(B);
```

**See Also** `impixel`, `improfile`

<b>Purpose</b>	Display image as texture-mapped surface
<b>Syntax</b>	<pre>warp(X,map) warp(I,n) warp(BW) warp(RGB) warp(z,...) warp(x,y,z,...) h = warp(...)</pre>
<b>Description</b>	<p><code>warp(X,map)</code> displays the indexed image <code>X</code> with colormap <code>map</code> as a texture map on a simple rectangular surface.</p> <p><code>warp(I,n)</code> displays the intensity image <code>I</code> with grayscale colormap of length <code>n</code> as a texture map on a simple rectangular surface.</p> <p><code>warp(BW)</code> displays the binary image <code>BW</code> as a texture map on a simple rectangular surface.</p> <p><code>warp(RGB)</code> displays the RGB image in the array <code>RGB</code> as a texture map on a simple rectangular surface.</p> <p><code>warp(z,...)</code> displays the image on the surface <code>z</code>.</p> <p><code>warp(x,y,z,...)</code> displays the image on the surface <code>(x,y,z)</code>.</p> <p><code>h = warp(...)</code> returns a handle to a texture-mapped surface.</p>
<b>Class Support</b>	The input image can be of class <code>logical</code> , <code>uint8</code> , <code>uint16</code> , or <code>double</code> .
<b>Remarks</b>	Texture-mapped surfaces are generally rendered more slowly than images.
<b>Examples</b>	Map an image of a test pattern onto a cylinder. <pre>[x,y,z] = cylinder; I = imread('testpat1.png'); warp(x,y,z,I);</pre>

# warp

---



## See Also

`imshow`

`image`, `imagesc`, `surf` in the MATLAB Function Reference



**Purpose** Watershed transform

**Syntax**  
`L = watershed(A)`  
`L = watershed(A,conn)`

**Description** `L = watershed(A)` computes a label matrix identifying the watershed regions of the input matrix `A`, which can have any dimension. The elements of `L` are integer values greater than or equal to 0. The elements labeled 0 do not belong to a unique watershed region. These are called *watershed pixels*. The elements labeled 1 belong to the first watershed region, the elements labeled 2 belong to the second watershed region, and so on.

By default, `watershed` uses 8-connected neighborhoods for 2-D inputs and 26-connected neighborhoods for 3-D inputs. For higher dimensions, `watershed` uses the connectivity given by `conndef(ndims(A), 'maximal')`.

`L = watershed(A,conn)` specifies the connectivity to be used in the watershed computation. `conn` can have any of the following scalar values.

Value	Meaning
<b>Two-dimensional connectivities</b>	
4	4-connected neighborhood
8	8-connected neighborhood
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can be defined in a more general way for any dimension by using for `conn` a 3-by-3-by- ...-by-3 matrix of 0's and 1's. The 1-valued

elements define neighborhood locations relative to the center element of conn. Note that conn must be symmetric about its center element.

## Class Support

A can be a numeric or logical array of any dimension, and it must be nonsparse. The output array L is of class double.

## Examples

### 2-D Example

- 1 Make a binary image containing two overlapping circular objects.

```
center1 = -10;
center2 = -center1;
dist = sqrt(2*(2*center1)^2);
radius = dist/2 * 1.4;
lims = [floor(center1-1.2*radius)...
        ceil(center2+1.2*radius)];
[x,y] = meshgrid(lims(1):lims(2));
bw1 = sqrt((x-center1).^2 + (y-center1).^2) <= radius;
bw2 = sqrt((x-center2).^2 + (y-center2).^2) <= radius;
bw = bw1 | bw2;
figure, imshow(bw,'InitialMagnification','fit');
title('bw')
```

- 2 Compute the distance transform of the complement of the binary image.

```
D = bwdist(~bw);
figure, imshow(D,[],'InitialMagnification','fit')
title('Distance transform of ~bw')
```

- 3 Complement the distance transform, and force pixels that don't belong to the objects to be at -Inf.

```
D = -D;
D(~bw) = -Inf;
```

- 4 Compute the watershed transform and display it as an indexed image.

```
L = watershed(D);
rgb = label2rgb(L,'jet',[.5 .5 .5]);
figure, imshow(rgb,'InitialMagnification','fit')
title('Watershed transform of D')
```

### 3-D Example

- 1 Make a 3-D binary image containing two overlapping spheres.

```
center1 = -10;
center2 = -center1;
dist = sqrt(3*(2*center1)^2);
radius = dist/2 * 1.4;
lims = [floor(center1-1.2*radius) ceil(center2+1.2*radius)];
[x,y,z] = meshgrid(lims(1):lims(2));
bw1 = sqrt((x-center1).^2 + (y-center1).^2 + ...
          (z-center1).^2) <= radius;
bw2 = sqrt((x-center2).^2 + (y-center2).^2 + ...
          (z-center2).^2) <= radius;
bw = bw1 | bw2;
figure, isosurface(x,y,z,bw,0.5), axis equal, title('BW')
xlabel x, ylabel y, zlabel z
xlim(lims), ylim(lims), zlim(lims)
view(3), camlight, lighting gouraud
```

- 2 Compute the distance transform.

```
D = bwdist(~bw);
figure, isosurface(x,y,z,D,radius/2), axis equal
title('Isosurface of distance transform')
xlabel x, ylabel y, zlabel z
xlim(lims), ylim(lims), zlim(lims)
view(3), camlight, lighting gouraud
```

- 3 Complement the distance transform, force nonobject pixels to be -Inf, and then compute the watershed transform.

```
D = -D;
```

# watershed

---

```
D(~bw) = -Inf;
L = watershed(D);
figure, isosurface(x,y,z,L==2,0.5), axis equal
title('Segmented object')
xlabel x, ylabel y, zlabel z
xlim(lims), ylim(lims), zlim(lims)
view(3), camlight, lighting gouraud
figure, isosurface(x,y,z,L==3,0.5), axis equal
title('Segmented object')
xlabel x, ylabel y, zlabel z
xlim(lims), ylim(lims), zlim(lims)
view(3), camlight, lighting gouraud
```

## Algorithm

watershed uses a variation of the Vincent and Soille algorithm [1]. For details of the variation, look at the filetoolbox/images/images/private/watershed\_vs.h, included with the Image Processing Toolbox.

## See Also

bwlabel, bwlabeln, bwdist, regionprops

## Reference

[1] Vincent, Luc, and Pierre Soille, "Watersheds in Digital Spaces: An Efficient Algorithm Based on Immersion Simulations," *IEEE Transactions of Pattern Analysis and Machine Intelligence*, Vol. 13, No. 6, June 1991, pp. 583-598.

**Purpose** `XYZ` color values of standard illuminants

**Syntax**  
`xyz = whitepoint`  
`xyz = whitepoint(string)`

**Description** `xyz = whitepoint(string)` returns `xyz`, a three-element row vector of **XYZ** values scaled so that  $Y = 1$ . `string` specifies the white reference illuminant. The following table lists all the possible values for `string`. The default value is enclosed in braces (`{}`).

Value	Description
'a'	CIE standard illuminant A
'c'	CIE standard illuminant C
'd50'	CIE standard illuminant D50
'd55'	CIE standard illuminant D55
'd65'	CIE standard illuminant D65
{'icc'}	ICC standard profile connection space illuminant; a 16-bit fractional approximation of D50

`xyz = whitepoint` is the same as `xyz = whitepoint('icc')`.

**Class Support** `string` is a character array. `xyz` is of class `double`.

**Examples** Return the *XYZ* color space representation of the default white reference illuminant `'icc'`.

```
wp_icc = whitepoint

wp_icc =

    0.9642    1.0000    0.8249
```

## **See Also**

`applycform`, `lab2double`, `lab2uint8`, `lab2uint16`, `makecform`,  
`xyz2double`, `xyz2uint16`

For a full list of the toolbox color space conversion functions, see “Color Space Conversions” on page 16-21.

<b>Purpose</b>	2-D adaptive noise-removal filtering
<b>Syntax</b>	<pre>J = wiener2(I,[m n],noise) [J,noise] = wiener2(I,[m n])</pre>
<b>Description</b>	<p>wiener2 lowpass-filters a grayscale image that has been degraded by constant power additive noise. wiener2 uses a pixelwise adaptive Wiener method based on statistics estimated from a local neighborhood of each pixel.</p> <p><code>J = wiener2(I,[m n],noise)</code> filters the image <code>I</code> using pixelwise adaptive Wiener filtering, using neighborhoods of size <code>m</code>-by-<code>n</code> to estimate the local image mean and standard deviation. If you omit the <code>[m n]</code> argument, <code>m</code> and <code>n</code> default to 3. The additive noise (Gaussian white noise) power is assumed to be <code>noise</code>.</p> <p><code>[J,noise] = wiener2(I,[m n])</code> also estimates the additive noise power before doing the filtering. <code>wiener2</code> returns this estimate in <code>noise</code>.</p>
<b>Class Support</b>	The input image <code>I</code> is a two-dimensional image of class <code>uint8</code> , <code>uint16</code> , <code>int16</code> , <code>single</code> , or <code>double</code> . The output image <code>J</code> is of the same size and class as <code>I</code> .
<b>Examples</b>	For an example, see “Using Adaptive Filtering” on page 11-50.
<b>Algorithm</b>	<p><code>wiener2</code> estimates the local mean and variance around each pixel,</p>

$$\mu = \frac{1}{NM} \sum_{n_1, n_2 \in \eta} a(n_1, n_2)$$

$$\sigma^2 = \frac{1}{NM} \sum_{n_1, n_2 \in \eta} a^2(n_1, n_2) - \mu^2$$

where  $\eta$  is the  $N$ -by- $M$  local neighborhood of each pixel in the image  $A$ . `wiener2` then creates a pixelwise Wiener filter using these estimates,

## wiener2

---

$$b(n_1, n_2) = \mu + \frac{\sigma^2 - v^2}{\sigma^2} (a(n_1, n_2) - \mu)$$

where  $v^2$  is the noise variance. If the noise variance is not given, `wiener2` uses the average of all the local estimated variances.

### See Also

`filter2`, `medfilt2`

### Reference

Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, p. 548, equations 9.44 -- 9.46.



**Purpose** Convert *XYZ* color values to double

**Syntax** `xyzd = xyz2double(XYZ)`

**Description** `xyxd = xyz2double(XYZ)` converts an M-by-3 or M-by-N-by-3 array of *XYZ* color values to double. `xyzd` has the same size as *XYZ*.

The Image Processing Toolbox follows the convention that double-precision *XYZ* arrays contain 1931 CIE *XYZ* values. *XYZ* arrays that are `uint16` follow the convention in the ICC profile specification (ICC.1:2001-4, [www.color.org](http://www.color.org)) for representing *XYZ* values as unsigned 16-bit integers. There is no standard representation of *XYZ* values as unsigned 8-bit integers. The ICC encoding convention is illustrated by this table.

Value (X, Y, or Z)	uint16 Value
0.0	0
1.0	32768
1.0 + (32767/32768)	65535

**Class Support** `xyz` is a `uint16` or `double` array that must be real and nonsparse. `xyzd` is of class `double`.

**Examples** Convert `uint16`-encoded *XYZ* values to double.

```
xyz2double(uint16([100 32768 65535]))
ans =

    3.0518e-003    1.0000e+000    2.0000e+000
```

**See Also** `applycform`, `lab2double`, `lab2uint16`, `lab2uint8`, `makecform`, `whitepoint`, `xyz2uint16`

For a full list of the toolbox color space conversion functions, see “Color Space Conversions” on page 16-21.

# xyz2uint16

---

**Purpose** Convert *XYZ* color values to uint16

**Syntax** xyz16 = xyz2uint16(xyz)

**Description** xyz16 = xyz2uint16(xyz) converts an M-by-3 or M-by-N-by-3 array of *XYZ* color values to uint16. xyz16 has the same size as xyz.

The Image Processing Toolbox follows the convention that double-precision *XYZ* arrays contain 1931 CIE *XYZ* values. *XYZ* arrays that are uint16 follow the convention in the ICC profile specification (ICC.1:2001-4, [www.color.org](http://www.color.org)) for representing *XYZ* values as unsigned 16-bit integers. There is no standard representation of *XYZ* values as unsigned 8-bit integers. The ICC encoding convention is illustrated by this table.

Value (X, Y, or Z)	uint16 Value
0.0	0
1.0	32768
1.0 + (32767/32768)	65535

**Class Support** xyz is a uint16 or double array that must be real and nonsparse. xyz16 is uint8.

**Examples** Convert *XYZ* values to uint16 encoding.

```
xyz2uint16([0.1 0.5 1.0])
ans =

    3277 16384 32768
```

**See Also** applycform, lab2double, lab2uint16, lab2uint8, makecform, whitepoint, xyz2double

For a full list of the toolbox color space conversion functions, see “Color Space Conversions” on page 16-21.

---

<b>Purpose</b>	Convert YCbCr color values to RGB color space
<b>Syntax</b>	<pre>rgbmap = ycbcr2rgb(ycbcrmap) RGB = ycbcr2rgb(YCBCR)</pre>
<b>Description</b>	<p><code>rgbmap = ycbcr2rgb(ycbcrmap)</code> converts the YCbCr values in the colormap <code>ycbcrmap</code> to the RGB color space. If <code>ycbcrmap</code> is M-by-3 and contains the YCbCr luminance (<i>Y</i>) and chrominance (<i>Cb</i> and <i>Cr</i>) color values as columns, <code>rgbmap</code> is returned as an M-by-3 matrix that contains the red, green, and blue values equivalent to those colors.</p> <p><code>RGB = ycbcr2rgb(YCBCR)</code> converts the YCbCr image <code>YCBCR</code> to the equivalent truecolor image <code>RGB</code>.</p>
<b>Class Support</b>	If the input is a YCbCr image, it can be of class <code>uint8</code> , <code>uint16</code> , or <code>double</code> . The output image is of the same class as the input image. If the input is a colormap, the input and output colormaps are both of class <code>double</code> .
<b>Remarks</b>	The YUV color space used by many video devices is similar, but not identical, to the YCbCr color space. If you need to convert YUV data to the RGB color space, try the <code>ycbcr2rgb</code> function.
<b>Examples</b>	Convert image from RGB space to YCbCr space and back. <pre>rgb = imread('board.tif'); ycbcr = rgb2ycbcr(rgb); rgb2 = ycbcr2rgb(ycbcr);</pre>
<b>See Also</b>	<code>ntsc2rgb</code> , <code>rgb2ntsc</code> , <code>rgb2ycbcr</code> For a full list of the toolbox color space conversion functions, see “Color Space Conversions” on page 16-21.
<b>References</b>	[1] Poynton, C. A. <i>A Technical Introduction to Digital Video</i> , John Wiley & Sons, Inc., 1996, p. 175.

[2] Rec. ITU-R BT.601-5, *Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-screen 16:9 Aspect Ratios*, (1982-1986-1990-1992-1994-1995), Section 3.5.

**Purpose**

Zoom in and out on image  
zoom is a MATLAB function.



# Examples

---

Use this list to find examples in the documentation.

## Introductory Examples

“Example 1 — Some Basic Concepts” on page 1-4

“Example 2 — Advanced Topics” on page 1-10

“Reading Image Data” on page 3-3

## Image Display

“Displaying Indexed Images” on page 4-51

“Displaying Grayscale Images” on page 4-52

“Displaying Binary Images” on page 4-54

“Displaying Truecolor Images” on page 4-56

“Special Display Techniques” on page 4-58

## Modular Tools

“Example: Embedding the Pixel Region Tool in an Existing Figure” on page 5-13

“Example: Building a Pixel Information GUI” on page 5-17

“Example: Building a Navigation GUI for Large Images” on page 5-23

“Example: Building an Image Comparison Tool” on page 5-26

## Morphology Examples

“Creating a Structuring Element” on page 10-7

“Dilating an Image” on page 10-10

“Eroding an Image” on page 10-11

“Combining Dilation and Erosion” on page 10-13

“Filling Holes” on page 10-26

“Finding Peaks and Valleys” on page 10-27

“Viewing a Label Matrix” on page 10-41

“Selecting Objects in a Binary Image” on page 10-42

“Finding the Area of the Foreground of a Binary Image” on page 10-42



“Finding the Euler Number of a Binary Image” on page 10-43

## **Image Analysis**

“Detecting Edges” on page 11-11

“Using Quadtree Decomposition” on page 11-21

## **Image Enhancement**

“Adjusting Intensity Values to a Specified Range” on page 11-35

“Contrast-Limited Adaptive Histogram Equalization” on page 11-41

“Decorrelation Stretching” on page 11-42

“Noise Removal” on page 11-47

“Using Median Filtering” on page 11-48

“Using Adaptive Filtering” on page 11-50

## **Working with Regions of Interest**

“Selecting a Polygon” on page 12-2

“Filtering a Region” on page 12-5

## **Working with Color**

“Determining Screen Bit Depth” on page 14-2

“Reducing the Number of Colors in an Image” on page 14-5

“Reducing Colors in an Indexed Image” on page 14-11

“Dithering” on page 14-12

“Example: Performing a Color Space Conversion” on page 14-16

“Example: Performing a Profile-Based Conversion” on page 14-20



1-bit image files 3-6

16-bit image files

    creating 3-7

    reading 3-3

24-bit image files 2-11

4-bit image files 3-5

8-bit image files

    creating 3-7

    reading 3-3

## A

adapthisteq 17-2

    increasing contrast 11-42

    using 11-41

adaptive filters 11-50

Adjust Contrast tool

    creating 5-2

    overview 4-36

    using 4-40

    using the dropper button 4-39

affine transformations

    definition 7-11

    using imtransform 6-12

aliasing

    definition 6-7

alpha channel 14-3

analyze75info 17-6

analyze75read 17-8

analyzing images

    edge detection 17-132

    intensity profiles 17-371

    pixel values 17-351

    quadtree decomposition 17-524

antialiasing 6-7

applycform 17-10

applylut 17-12

approximation

    in indexed images 14-11

    of an image background 1-11

    of colors 14-6

area

    of binary images 10-42

    of image regions 11-10

arrays

    storing images 2-2

averaging filter

    creating 17-163

    example 8-5

axes2pix 17-15

## B

background

    of an grayscale image 1-11

basicimageinfo 17-265

bestblk 17-16

bicubic interpolation 6-3

bilinear interpolation 6-3

binary images 2-8

    applying a lookup table 17-12

    calculating the Euler number 10-43

    changing the display colors of 4-54

    constructing lookup tables 17-479

    displaying 4-54

    feature measurement 10-40

    flood fill operation 10-24

    image area 10-42

    labeling connected components 17-41

    labeling connected components

        labeling 10-40

    lookup table operations 10-44

    morphological operations 10-3

    object selection 17-58

    perimeter determination 10-17

    selecting objects in 10-42

    using neighborhood operations 17-12

binary masks

    creating 12-2

bit depth

- 1-bit images 3-6
- screen bit depth 14-2
- blind deconvolution algorithm
  - used for deblurring 13-16
- blkproc 17-17
- block processing 15-2
  - block size 17-16
  - column processing 15-12
  - distinct blocks 15-8
  - padding borders in block processing 15-5
  - sliding neighborhoods 15-4
  - using nlfilt 17-496
- border padding
  - in block processing 15-5
- border replication
  - in image processing 8-10
- borders
  - controlling in figure window 4-7
- boundary padding, *see* border padding
- boundary ringing
  - in image deblurring 13-24
- boundary tracing 11-13
- bounding box
  - finding for a region 11-10
- brightness
  - adjusting interactively 4-36
  - adjusting the window/level 4-43
- brightness adjustment 11-36
- bwarea 17-15 17-21
  - using 10-42
- bwareaopen 17-23
- bwboundaries 17-26
- bwdist 17-31
  - using 10-37
- bweuler 17-37
- bwhitmiss 17-39
- bwlabel 17-41
- bwlabeln 17-44
- bwmorph 17-47
  - skeletonization example 10-16

- bwpack 17-53
- bwperim 17-56
- bwselect 17-58
- bwtraceboundary 17-60
  - using 11-14
- bwulterode 17-63
- bwunpack 17-65

## C

- camera read-out noise 13-11
- Canny edge detector 17-132
  - using the edge function 11-11
- center of mass
  - calculating for region 11-10
- center pixel
  - calculating 15-4
  - in sliding neighborhood operations 15-4
- checkerboard 17-66
- Choose Colormap tool
  - using 4-13
- chrominance
  - in CIE color spaces 14-15
  - in NTSC color space 14-22
  - in YCbCr color space 14-23
- CIE color spaces 14-14
- CIELAB color space 14-14
- closing 17-47
  - morphology 10-13
- cmpermute 17-68
- cmunique 17-69
- col2im 17-72
- colfilt 17-73
- color
  - approximation 14-6
  - dithering 14-12
  - quantization 14-6
  - quantization performed by rgb2ind 17-549
  - reducing number of colors 14-5
- color approximation

- performed by `rgb2ind` 17-549
- color cube
  - description of 14-6
  - quantization of 14-7
- color planes
  - of an HSV image 14-25
  - of an RGB image 2-13
  - slice of color cube 14-8
- color reduction 14-5
- color spaces
  - converting among 14-14
  - converting between 17-499 17-551
  - data encodings 14-17
  - device-independent color spaces 14-15
  - HSV 14-24
  - NTSC 14-22 17-499 17-551
  - YCbCr 14-23
  - YIQ 14-22
- colorbars
  - adding 4-58
- colorcube 14-11
- colormap mapping 14-10
- colormaps
  - brightening 17-20
  - choosing in Image Tool 4-13
  - creating a colormap using
    - colorcube 14-11
  - darkening 17-20
  - rearranging colors in 17-68
  - removing duplicate entries in 17-69
- column processing 17-73
  - in neighborhood operations 15-12
  - reshaping columns into blocks 17-72
- composite transformations 6-12
- conformal transformations 7-11
- `conndef` 17-76
- connected components
  - labeling 10-40
  - using `bwlabel` 17-41
- connectivity
  - overview 10-22
  - specifying custom 10-24
- constant component, *see* zero-frequency component
- contour plots
  - text labels 11-8
- contrast
  - adjusting interactively 4-36
  - adjusting the window/level 4-43
- contrast adjustment
  - decreasing contrast 11-36
  - increasing contrast 11-34
  - specifying limits automatically 11-37
- contrast stretching 11-34
  - understanding 4-45
  - with decorrelation stretching 11-45
  - See also* contrast adjustment
- contrast-limited adaptive histogram equalization (CLAHE) 11-41
- Control Point Selection Tool
  - appearance of control point symbols 7-25
  - changing view of images 7-16
  - saving a session 7-29
  - saving control points 7-28
  - specifying control points 7-21
  - starting 7-15
  - using 7-13
  - using point prediction 7-23
- control points
  - appearance of 7-25
  - prediction 7-23
  - saving 7-28
  - selecting 7-13
  - specifying 7-21
- `conv2`
  - compared to `imfilter` 8-13
- conversions between image types 2-15
- `convmtx2` 17-79
- `convn`
  - compared to `imfilter` 8-13

- convolution
  - convolution matrix 17-79
  - definition 8-2
  - Fourier transform and 9-11
  - two-dimensional 8-2
  - using sliding neighborhood operations 15-6
  - with `imfilter` 8-7
- convolution kernel
  - definition 8-2
- coordinate systems
  - pixel coordinates 2-3
  - spatial coordinates 2-4
- `corr2` 17-81
  - calculating summary statistics 11-10
- correlation
  - definition 8-4
  - Fourier transform 9-12
  - with `imfilter` 8-7
- correlation coefficient 17-81
- correlation kernel
  - definition 8-4
- `cp2tform` 17-82
  - using 7-11
- `cpcorr` 17-91
  - example 7-30
- `cpselect` 17-93
  - using 7-6
- `cpstruct2pairs` 17-96
- cropping an image 6-10
- cross-correlation
  - improving control point selection 7-30
  
- D**
- damping
  - for noise reduction 13-10
- data types
  - converting between 3-7
  - double-precision (`double`) 3-7
  - in image filtering 8-5
- DC component, *see* zero-frequency component
- `dct2` 17-97
  - equations 9-15
- `dctmtx` 17-100
  - using 9-16
- deblurring
  - avoiding boundary ringing 13-24
  - conceptual model 13-2
  - overview 13-2
  - overview of functions 13-4
  - use of frequency domain 13-23
  - using the blind deconvolution algorithm 13-16
  - using the Lucy-Richardson algorithm 13-10
  - with regularized filter 13-8
  - with the Wiener filter 13-6
- decomposition of structuring elements
  - example 10-8
  - getting sequence 17-191
- `deconvblind` 17-101
  - example 13-16
- `deconvlucy` 17-104
  - example 13-10
- `deconvreg` 17-107
  - example 13-8
- `deconvwnr` 17-109
  - example 13-6
- decorrelation stretching 11-42
  - See also* contrast adjustment
- `decorrstretch` 17-112
- detail rectangle
  - getting position of 4-20
  - in Control Point Selection Tool 7-15
  - specifying color in Image Tool 4-20
- DICOM files
  - reading and writing 3-9
- DICOM unique identifier
  - generating 3-15

- dicomanon 17-113
  - dicomdict 17-115
  - dicominfo 17-116
  - dicomlookup 17-118
  - dicomread 17-119
  - dicomuid 17-121
  - dicomwrite 17-122
  - Digital Imaging and Communications in Medicine, *see* DICOM files
  - dilation 10-3
    - grayscale 17-500
  - discrete cosine transform 9-15
    - image compression 9-17
    - transform matrix 9-16 17-100
  - discrete Fourier transform 9-7
  - disk filter 17-163
  - display depth 14-2
    - See also* screen color resolution
  - display range
    - getting information about 4-26
  - Display Range tool
    - creating 5-2
    - in Image Tool 4-26
    - overview 4-24
  - display techniques
    - displaying images at true size 17-587
    - multiple images 17-575
  - displaying images
    - adding a colorbar 4-58
    - binary 4-54
    - binary images with different colors 4-54
    - comparison of functions 4-3
    - controlling figure window 4-7
    - grayscale images 4-52
    - indexed images 4-51
    - initial size 4-6
    - multiple images 4-47
    - multiple images in figure 4-48
    - special techniques 4-58
    - texture mapping 4-62
    - toolbox preferences for 4-66
    - truecolor 4-56
    - unconventional ranges of data 4-52
    - using imshow 4-5
    - using the Image Tool 4-9
  - distance
    - between pixels 4-31
    - Euclidean 4-31
  - Distance tool
    - creating 5-3
  - distance transform 10-37
  - distinct block operations 15-8
    - overlap 15-10
    - zero padding 15-8
  - distortion operator 13-2
  - dither 17-128
  - dithering 14-12
    - example 14-12
  - dropper button
    - in Adjust Contrast tool 4-39
- E**
- edge 17-131
    - example 11-12
  - edge detection 11-11
    - Canny method 11-11
    - example 11-12
    - methods 17-132
    - Sobel method 11-12
  - edgetaper 17-138
    - avoiding boundary ringing 13-24
  - enhancing images
    - decorrelation stretching 11-42
    - intensity adjustment 11-35
    - noise removal 11-47
  - entropy 17-139
  - entropyfilt 17-140
  - erosion 10-3
    - grayscale 17-500

- Euclidean distance 4-31 17-515
- Euler number
  - calculating 10-43
- exploring images
  - using Image Tool 4-18
- exporting data
  - in Image Tool 4-16
- eye dropper button
  - in Adjust Contrast tool 4-39

## F

- fan-beam projection data
  - arc geometry 9-36
  - computing 9-35
  - line geometry 9-37
  - reconstructing image from 9-38
- fan2para 17-142
- fanbeam 17-146
  - using 9-35
- fast Fourier transform 9-7
  - zero padding 9-9
  - See also* Fourier transform
- feature measurement 1-19
  - area 11-10
  - binary images 10-40
  - bounding box 11-10
  - center of mass 11-10
- fft 9-7
- fft2 9-7
  - example 9-8
  - using 9-10
- fftn 9-7
- fftshift
  - example 8-17
  - using 9-10
- figure
  - controlling borders 4-7
- files
  - displaying images from disk 4-5
- filling a region 12-8
- filling holes in images 10-26
- filter design 8-15
  - frequency sampling method 8-18 17-160
  - frequency transformation method 8-16 17-169
  - windowing method 8-19 17-176
- filter2
  - compared to imfilter 8-13
  - example 11-49
- filtering
  - a region 12-5
  - masked filtering 12-5
- filters
  - adaptive 11-50
  - averaging 17-163
  - binary masks 12-5
  - computing frequency response 9-10
  - designing 8-15
  - disk 17-163
  - finite impulse response (FIR) 8-16
  - frequency response 8-22
  - Gaussian 17-163
  - imfilter 8-5
  - Infinite Impulse Response (IIR) 8-16
  - Laplacian 17-163
  - linear 8-2
  - Log 17-163
  - median 11-48 17-492
  - motion 17-163
  - multidimensional 8-11
  - order-statistic 17-500
  - Prewitt 17-163
  - Sobel 17-163
  - unsharp 17-163
  - unsharp masking 8-13
- findbounds 17-155
- FIR filters 8-16
  - transforming from one-dimensional to two-dimensional 8-16



- flat-field correction 13-11
  - fliptform 17-156
  - flood-fill operation 10-24
  - Fourier transform 9-2
    - applications of the Fourier transform 9-10
    - centering the zero-frequency coefficient 9-10
    - computing frequency response 9-10
    - convolution and 9-11
    - correlation 9-12
    - DFT coefficients 9-8
    - examples of transform on simple shapes 9-5
    - fast convolution with 9-11
    - for performing correlation 9-12
    - frequency domain 9-2
    - increasing resolution 9-9
    - padding before computation 9-9
    - two-dimensional 9-2
    - zero-frequency component 9-2
  - freqspace 8-21
    - example 8-18
  - frequency domain 9-2
  - frequency response
    - computing 8-22 17-158
    - desired response matrix 8-21
    - of linear filters 9-10
  - frequency sampling method (filter design) 8-18
    - using fsamp2 17-160
  - frequency transformation method (filter design) 8-16
    - using ftrans2 17-169
  - freqz
    - example 8-17
  - freqz2 17-158
    - computing frequency response of linear filters 9-10
    - example 8-18
  - fsamp2 17-160
    - example 8-18
  - fspecial 17-163
    - creating predefined filters 8-13
  - ftrans2 17-169
    - example 8-17
  - fwind1 17-172
    - example 8-20
  - fwind2 17-176
    - example 8-19
- ## G
- gamma correction 11-37
  - Gaussian convolution kernel
    - frequency response of 9-10
  - Gaussian filter 17-163
  - Gaussian noise 11-50
    - adding 17-342
  - geocoded images 7-4
  - geometric operations
    - cropping 6-10
    - interpolation 6-3
    - resizing 6-5
  - georegistered images 7-4
  - getheight 17-180
  - getimage 17-181
    - example 4-6
    - using with Image Tool 4-16
  - getimagemodel 17-184
  - getline 17-185
  - getneighbors 17-186
  - getnhood 17-187
  - getpts 17-188
  - getrangefromclass 17-189
  - getrect 17-190
  - getsequence 17-191
  - graphical user interfaces (GUIs)
    - building 5-2
  - graphics card 14-4
  - graphics file formats
    - converting from one format to another 3-8

- writing data 3-5
- gray2ind 17-193
- grayscale images
  - displaying 4-52
  - flood-fill operation 10-24
- grayscale morphological operations 17-500
- grayscale 17-202
- graythresh 17-204
  - thresholding image values 1-15

## H

- histeq 17-205
  - example 11-40
  - increase contrast example 11-39
- histogram equalization 11-39 17-205
- histograms 11-9
- holes
  - filling 10-24
  - tracing boundaries 11-13
- hough 17-210
  - using 11-19
- houghlines 17-212
  - using 11-20
- houghpeaks 17-215
  - using 11-19
- HSV color space 14-24
  - color planes of 14-25
- hsv2rgb 14-24
- hue
  - in HSV color space 14-24
  - in NTSC color space 14-22

## I

- ICC profiles
  - processing 14-18
- iccfind 17-218
- iccread 17-220
- iccroot 17-224

- iccwrite 17-225
- idct2 17-227
- ifanbeam 17-229
  - using 9-38
- ifft 9-7
- ifft2 9-7
- ifftn 9-7
- IIR filters 8-16
- im2bw 17-236
- im2col 17-238
- im2double 17-240
- im2int16 17-241 17-244
- im2java2d 17-243
- im2single 17-244
- im2uint16 17-245
- im2uint8 17-246
- imabsdiff 17-247
- imadd 17-249
- imadjust 17-251
  - brightening example 11-36
  - example 11-35
  - gamma correction and 11-37
  - gamma correction example 11-38
  - increase contrast example 11-35
  - setting limits automatically 11-37
- image analysis
  - contour plots 11-7
  - edge detection 11-11
  - histograms 11-9
  - intensity profiles 11-3
  - overview 11-11
  - pixel values 11-2
  - quadtree decomposition 11-21
  - summary statistics 11-10
- image arithmetic
  - combining functions 2-26
  - overview 2-25
  - truncation rules 2-25
- image display
  - special techniques 4-58

- image editing 12-8
- image filtering
  - data types 8-5
  - unsharp masking 8-13
  - with `imfilter` 8-5
- Image Information tool
  - creating 5-3
  - using 4-34
- image metadata
  - viewing 4-34
- image properties
  - set by `imshow` 4-4
- image registration
  - fine-tuning point placement 7-30
  - overview 7-2
  - procedure 7-2
  - selecting control points 7-13
  - specifying control point pairs 7-21
  - types of transformations 7-11
  - using control point prediction 7-23
- image rotation 6-8
  - specifying interpolation method 6-8
  - specifying size of output image 6-8
- Image Tool
  - choosing a colormap 4-13
  - closing 4-17
  - compared to `imshow` 4-3
  - controlling initial magnification 4-12
  - exploring images 4-18
  - exporting data 4-16
  - importing data 4-15
  - opening 4-11
  - overview 4-9
  - panning images 4-21
  - printing 4-17
  - specifying image magnification 4-22
  - using the Pixel Region tool 4-24
  - zoom tools 4-22
- image transformations
  - affine 7-11
    - custom 6-12
    - local weighted mean 7-12
    - piecewise linear 7-12
    - polynomial 7-12
    - projective 7-12
    - supported by `cp2tform` 7-11
    - types of 7-11
    - using `imtransform` 6-12
  - image types 2-7
    - binary 2-8
    - converting 2-15
    - grayscale 2-10
    - indexed 2-8
    - interpolation and 6-4
    - multiframe images 2-19
    - supported by the toolbox 2-7
    - truecolor 2-11
    - See also* binary images, grayscale images, indexed images, multiframe images, truecolor images
- `imageinfo` 17-254
- `imagemodel` 17-257
- images
  - adjusting contrast 4-36
  - analyzing 11-2
  - arithmetic operations 2-25
  - brightness control 4-36
  - causes of blurring 13-2
  - creating movies 17-339
  - data types 3-7
  - displaying multiple images 4-47 17-575
  - displaying multiple images in figure 4-48
  - feature measurement 1-19
  - filling holes in 10-26
  - finding image minima and maxima 10-27
  - getting data from axes 17-181
  - getting information about 4-34
  - getting information about display
    - range 4-26
  - how MATLAB stores 2-2

- image types 2-7
- improving contrast 1-14
- printing 4-64
- reducing number of colors 14-5
- registering 7-2
- restoring blurred images 13-2
- returning information about 3-2
- statistical analysis of 1-21
- using imshow 4-5
- viewing as a surface plot 1-12
- viewing metadata 4-34
- imapprox 17-263
  - example 14-11
- imbothat 17-268
- imclearborder 17-270
- imclose 17-273
  - using 10-13
- imcomplement 17-275
- imcontour 17-277
  - example 11-8
- imcontrast 17-279
- imcrop 17-281
  - example 6-10
- imdilate 17-284
- imdisplayrange 17-288
- imdistsline 17-290 17-363
- imdivide 17-297
- imerode 17-299
  - closure example 10-14
- imextendedmax 17-302
  - example 10-31
- imextendedmin 17-304
- imfill 17-306
  - example 10-26
- imfilter 17-310
  - compared to other filtering functions 8-13
  - convolution option 8-7
  - correlation option 8-7
  - padding options 8-8
  - using 8-5
- imfinfo 3-2
  - example 3-6
- imgca 17-314
- imgcf 17-316
- imgetfile 17-317
- imhandles 17-318
- imhist 17-319
  - creating a histogram 11-9
- imhmax 17-321
- imhmin 17-323
- imimposemin 17-326
- imlincomb 17-330
  - example 2-26
- imline 17-333
- immagbox 17-337
- immovie 17-339
  - example 4-61
- immultiply 17-340
- imnoise 17-342
  - example 11-50
  - salt & pepper example 11-49
- imopen 17-345
  - using 10-13
- imoverview 17-347
- imoverviewpanel 17-350
- impixel 17-351
  - example 11-2
- impixelinfo 17-354
- impixelinfoval 17-357
- impixelregion 17-359
- impixelregionpanel 17-362
- importing data
  - in Image Tool 4-15
- impositionrect 17-367 17-378
- improfile 17-371
  - example 11-5
  - grayscale example 11-4
- imread 3-3
  - example for multiframe image 3-4
- imreconstruct 17-376

- example 10-19
- imregionalmax 17-382
- imregionalmin 17-385
- imresize 17-388
  - using 6-5
- imrotate 17-390
  - specifying interpolation method 6-8
  - specifying size of output image 6-8
  - using 6-8
- imscrollpanel 17-392
- imshow 17-397
  - compared to Image Tool 4-3
  - displaying images 4-5
  - displaying unconventional range data 4-53
  - example for binary images 4-54
  - example for grayscale images 4-52
  - example for truecolor images 4-56
  - used with indexed images 4-51
- imsubtract 17-402
- imtool 17-404
  - compared to imshow 4-3
  - displaying unconventional range data 4-53
  - example for binary images 4-54
  - example for grayscale images 4-52
  - example for truecolor images 4-56
  - overview 4-9
  - used with indexed images 4-51
- imtophat 17-409
- imtransform 17-412
  - using 6-12
- imview 17-419
- imwrite
  - example 3-6
- ind2gray 17-421
- ind2rgb 17-422
- indexed images 2-8
  - converting from intensity 17-193
  - converting from RGB 17-548
  - displaying 4-51
  - reducing number of colors in 14-11
- infinite impulse response (IIR) filter 8-16
- Intel Performance Primitives Library 17-426
- intensity adjustment 11-35
  - gamma correction 11-37
  - histogram equalization 11-39
  - specifying limits automatically 11-37
  - See also* contrast adjustment
- intensity images
  - converting from matrices 17-490
  - converting from RGB 17-546
  - converting to indexed 17-193
  - , *see* grayscale images
- intensity profiles 11-3 17-371
- interfileinfo 17-423
- interfileread 17-424
- interpolation 6-3
  - bicubic 6-3
  - bilinear 6-3
  - intensity profiles 11-3
  - nearest-neighbor 6-3
  - tradeoffs between methods 6-3
  - within a region of interest 12-8
- intlut 17-425
- inverse Radon transform 9-27
  - example 9-33
  - filtered backprojection algorithm 9-29
- ippl 17-426
- iptaddcallback 17-427
- iptcheckconn 17-428
- iptcheckhandle 17-429
- iptcheckinput 17-431
- iptcheckmap 17-433
- iptcheckmargin 17-434
- iptcheckstrs 17-435
- iptcondir 17-441
- iptdemos 17-437
- iptgetapi 17-438
- iptGetPointerBehavior 17-439
- iptgetpref 17-440
  - using 4-67

- iptnum2ordinal 17-442
- iptPointerManager 17-443
- iptremovecallback 17-444
- iptSetPointerBehavior 17-445
- iptsetpref 17-449
  - using 4-68
- iptwindowalign 17-453
- iradon 9-27 17-455
  - example 9-27
- isbw 17-458
- isflat 17-459
- isgray 17-460
- isicc 17-461
- isind 17-462
- isrgb 17-463

## J

- JPEG compression
  - discrete cosine transform and 9-17

## L

- lab2double 17-464
- lab2uint16 17-466
- lab2uint8 17-468
- label matrix
  - creating 10-40
  - viewing as pseudocolor image 10-41
- label2rgb 17-470
- labeling
  - connected components 10-40
  - levels of contours 11-8
- Laplacian filter 17-163
- Laplacian of Gaussian edge detector 17-132
- line detection 9-24
- line segment
  - pixel values along 11-3
- linear conformal transformations 7-11
- linear filtering 8-2

- convolution 8-2
- filter design 8-15
- FIR filters 8-16
- IIR filters 8-16
- noise removal and 11-47
  - using sliding neighborhood operations 15-6
- local weighted mean transformations 7-12
- Log filters 17-163
- lookup tables
  - constructing 17-479
  - using 10-44
- Lucy-Richardson algorithm
  - used for deblurring 13-10
- luminance
  - in NTSC color space 14-22
  - in YCbCr color space 14-23

## M

- magnification
  - specifying in Image Tool 4-22
  - specifying initial value in Image Tool 4-12
- Magnification box
  - creating 5-3
- magnifying, *see* resizing images
- makecform 17-473
- makeConstrainToRectFcn 17-478
- makelut 17-479
- makeresampler 17-481
- maketform 17-486
- marker image
  - creating 10-34
  - definition 10-18
- mask image
  - definition 10-18
- masked filtering 12-5
- mat2gray 17-490
- matrices
  - converting to intensity images 17-490

- storing images in 2-2
- maxima
  - finding in images 10-29
  - imposing 10-33
  - suppressing 10-31
- McClellan transform 17-169
- mean2 11-10 17-491
- medfilt2 17-492
  - example 11-50
  - using 11-48
- median filtering 11-48 17-492
- metadata
  - viewing 4-34
- minima
  - finding in images 10-29
  - imposing 10-33
  - suppressing 10-31
- minimum variance quantization 14-9
- modular tools
  - connecting 5-25
  - creating your own 5-31
  - embedding in existing figures 5-12
  - navigation aids 5-19
  - overview 5-2
  - positioning in a GUI 5-15
  - specifying parent of 5-12
  - using 5-6
  - utility functions 5-31
- moiré patterns 6-7
- montage 17-494
  - example 4-60
- morphological operations 10-3
  - closing 17-47
  - diagonal fill 17-48
  - dilation 10-3
  - erosion 10-3
  - grayscale 17-500
  - opening 10-13
  - overview 10-1
  - predefined operations 10-15

- shrinking objects 17-49
  - skeletonization 10-16
- morphological reconstruction
  - finding peaks and valleys 10-27
  - overview 10-18
- morphology 10-18
  - See also* morphological reconstruction
- motion filters 17-163
- movies
  - creating from images 4-61 17-339
  - playing 4-61
- multidimensional filters 8-11
- multiframe images
  - about 2-19
  - displaying 17-494
- multilevel thresholding 17-202

## N

- navigation aids
    - creating 5-19
  - nearest-neighbor interpolation 6-3
  - neighborhoods
    - binary image operations 17-12
    - neighborhood operations 15-2
  - nlfilter 17-496
  - noise
    - adding to an image 17-342
  - noise amplification
    - reducing 13-10
  - noise removal 11-47
    - adaptive filtering (Weiner) and 11-50
    - Gaussian noise 11-50
    - grain noise 11-47
    - linear filtering and 11-47
    - median filter and 11-48
    - salt and pepper noise 11-48
  - nonimage data
    - displaying as image 4-52
  - nonlinear filtering

- using sliding neighborhood operations 15-6
- normalized cross-correlation 7-30
- normxcorr2 17-497
- NTSC color space 14-22 17-499 17-551
- ntsc2rgb 14-22 17-499

## O

- object selection 17-58
- objects
  - tracing boundaries 11-13
- observed image
  - in image registration 7-13
- opening
  - morphology 10-13
- optical transfer function (OTF) 13-2
- order-statistic filtering 17-500
- ordfilt2 17-500
- orthonormal matrix 9-17
- orthophoto
  - defined 7-4
- orthorectified image 7-4
- otf2psf 17-502
  - use of 13-23
- overlap
  - in distinct block operations 15-8
- Overview tool
  - creating 5-4
  - customizing 4-20
  - getting position of detail rectangle 4-20
  - printing image in 4-21
  - starting in Image Tool 4-20
  - using 4-18

## P

- padding borders
  - in block processing 15-5
- options with imfilter 8-8
- panning images
  - using the Image Tool 4-21
- para2fan 17-507
- parallel beam projections 9-28
- perimeter determination
  - in binary images 10-17
- phantom 9-30 17-512
- piecewise linear transformations 7-12
- Pixel Information tool
  - creating 5-4
  - overview 4-24
- Pixel Region tool
  - creating 5-5
  - customizing 4-29
  - determining location of cursor 4-29
  - overview 4-24
  - printing contents 4-30
  - selecting a region 4-28
  - using 4-24
  - using in Image Tool 4-27
- pixel regions
  - viewing 4-27
- pixel values
  - along a line segment 11-3
  - using impixel 17-351
  - using pixval 17-515
  - using the Pixel Region tool 4-24
- pixels
  - correcting for bad pixels 13-11
  - defining connectivity 10-22
  - definition 2-2
  - Euclidean distance between 4-31
  - getting pixel value information in Image Tool 4-25
  - getting value of using Pixel Information tool 4-24
  - getting values of 11-2
  - selecting 11-2
  - viewing values of pixel regions 4-27



pixval 17-515  
     using 11-2  
 PNG (Portable Network Graphics) files  
     writing as 16-bit 3-5  
 point mapping  
     for image registration 7-2  
 point spread function (PSF) 13-2  
     importance of in deblurring 13-3  
 Poisson noise  
     adding 17-342  
 poly2mask 17-517  
 polygon  
     pixels inside 12-2  
     selecting a polygonal region of  
         interest 12-2  
 polynomial transformations 7-12  
 predicting control point locations  
     in image registration 7-23  
 preferences  
     getting value of 4-67  
     Image Processing Toolbox display  
         preferences 4-66  
     ImshowAxesVisible 4-66  
     ImshowBorder 4-66  
     ImshowInitialMagnification 4-67  
     ImtoolInitialMagnification 4-67  
     setting value of 4-68  
 Prewitt edge detector 17-132  
 Prewitt filters 17-163  
 printing  
     contents of Overview tool 4-21  
     contents of Pixel Region tool 4-30  
 printing images 4-64  
     from Image Tool 4-17  
 profiles  
     reading ICC color profiles 14-18  
 projections  
     parallel beam 9-28  
 projective transformations 6-12 7-12  
 psf2otf 17-523

use of 13-23

## Q

qtdecomp 17-524  
     example 11-22  
 qtgetblk 17-528  
 qtsetblk 17-530  
 quadtree decomposition 11-21 17-524  
     getting block values 17-528  
     setting block values 17-530  
 quantization 14-6  
     minimum variance 14-9  
     performed by rgb2ind 17-549  
     tradeoffs between using minimum  
         variance and uniform quantization  
         methods 14-10

## R

radon 17-531  
     example 9-22  
 Radon transform  
     center pixel 9-21  
     description of 9-19  
     example 9-30  
     inverse Radon transform 9-27  
     line detection example 9-24  
     of the Shepp-Logan Head phantom 9-31  
     relationship to Hough transform 9-24  
 rangefilt 17-534  
 rank filtering 11-48  
     *See also* order-statistic filtering  
 read-out noise  
     correcting 13-11  
 real orthonormal matrix 9-17  
 reconstruction  
     morphological 10-18  
 reference image  
     in image registration 7-13

- reflect 17-536
  - region labeling 10-40
  - region of interest
    - based on color or intensity 12-4
    - binary masks 12-2
    - filling 12-8
    - filtering 12-5
    - polygonal 12-2
    - selecting 12-2
    - using arbitrary binary masks 12-4
  - region property measurement 11-10
  - regional maxima
    - definition 10-29
    - imposing 10-33
    - suppressing 10-31
  - regional minima
    - definition 10-29
    - imposing 10-33
    - suppressing 10-31
  - regionprops 17-537
    - using 1-19 11-10
  - registering an image 7-2
  - regularized filter
    - used for deblurring 13-8
  - replication
    - to avoid border effect 8-10
  - resizing images 6-5
    - antialiasing 6-7
  - resolution
    - screen color resolution 14-2
    - See also* bit depth 14-2
  - RGB color cube
    - description of 14-6
    - quantization of 14-7
  - RGB images 2-11
    - converting to indexed 17-548
    - converting to intensity 17-546
    - See also* truecolor images
  - rgb2gray 17-546
  - rgb2hsv 14-24
    - converting RGB to HSV color space 14-24
    - example 14-25
  - rgb2ind 17-548
    - colormap mapping example 14-11
    - example 14-11
    - in minimum variance quantization 14-9
    - minimum variance quantization
      - example 14-9
    - specifying a colormap to use 14-10
    - uniform quantization example 14-8
    - used in dithering 14-12
  - rgb2ntsc 17-551
    - example 14-22
  - rgb2ycbcr 17-552
    - example 14-23
  - Richardson-Lucy algorithm, *see* Lucy-Richardson
  - ringing
    - in image deblurring 13-24
  - Roberts edge detector 17-132
  - roicolor 12-4 17-555
  - roifill 17-557
    - example 12-8
  - roifilt2 17-559
    - contrast example 12-6
  - roipoly 17-561
    - example 12-2
  - rotation
    - of images 6-8
- S**
- salt and pepper noise 11-48
    - adding 17-342
  - sampling
    - handling undersampled images 13-12
  - saturation
    - in HSV color space 14-24
    - in NTSC color space 14-22
  - screen bit depth 14-2

screen color resolution 14-2  
 ScreenDepth 14-2  
 Scroll Panel tool  
     creating 5-5  
 scroll panels  
     understanding 5-20  
 Shepp-Logan head phantom 9-30  
 shrinking, *see* resizing images  
 Signal Processing Toolbox  
     hamming function 8-20  
 skeletonization 10-16  
 sliding neighborhood operations 15-4  
     center pixel in 15-4  
     padding in 15-5  
 Sobel edge detector 17-132  
 Sobel filters 17-163  
 spatial coordinates 2-4  
 speckle noise  
     adding 17-342  
 statistical properties  
     mean 17-491  
     of image objects 1-21  
     standard deviation 17-564  
 std2 11-10 17-564  
 stdfilt 17-565  
 storage classes  
     converting between 2-17  
 strel 17-566  
 stretchlim 17-573  
     adjusting image contrast 1-14  
     using 11-37  
 structuring elements 10-6  
     creating 10-7  
     decomposition of 10-8  
     decomposition sequence 17-191  
     determining composition 17-286  
 subimage 17-575  
 subtraction  
     of one image from another 1-14  
 surf

viewing images 1-12

## T

target images  
     definition 5-2  
     displaying 5-7  
     getting handle to 5-11  
     specifying 5-8  
 template matching 9-12  
 texture mapping 4-62  
 tformarray 17-576  
 tformfwd 17-581  
 tforminv 17-583  
 thresholding  
     to create a binary image 1-15  
     to create indexed image from intensity  
         image 17-202  
 tomography 9-27  
 tools  
     creating your own 5-31  
     modular 5-2  
 tracing boundaries 11-13  
 transformation matrix 8-16  
 transforms 9-1  
     discrete cosine 9-15 17-97  
     discrete Fourier transform 9-7  
     Fourier 9-2  
     inverse Radon 9-27  
     Radon 9-19  
     two-dimensional Fourier transform 9-2  
 translate 17-585  
 transparency 14-3  
 truecolor images  
     displaying 4-56  
     measuring the intensities of each color  
         plane 11-5  
     reducing number of colors 14-5  
 truesize 17-587  
 truncation rules

for image arithmetic operators 2-25

## U

uint16

storing images in 3-3

uint8

storing images in 3-3

uintlut 17-590

undersampling

correcting 13-12

uniform quantization, *see* quantization

unsharp filters 17-163

unsharp masking 8-13

## W

warp 17-591

example 4-62

watershed 17-593

weight array

in deblurring 13-11

whitepoint 17-597

Wiener filter

deblurring with 13-6

wiener2 17-599

adaptive filtering 11-50

using 11-50

window/level

adjusting 4-43

windowing method (filter design) 8-19 17-176

## X

X-ray absorption tomography 9-27

XYZ color space 14-14

xyz2double 17-601

xyz2uint16 17-602

## Y

YCbCr color space 14-23

ycbcr2rgb 17-603

using 14-23

YIQ color space 14-22

## Z

zero padding 9-12

and the fast Fourier transform 9-9

image boundaries 8-8

zero-cross edge detector 17-132

zero-frequency component 9-2

zooming

Control Point Selection Tool 7-18

in Image Tool 4-22