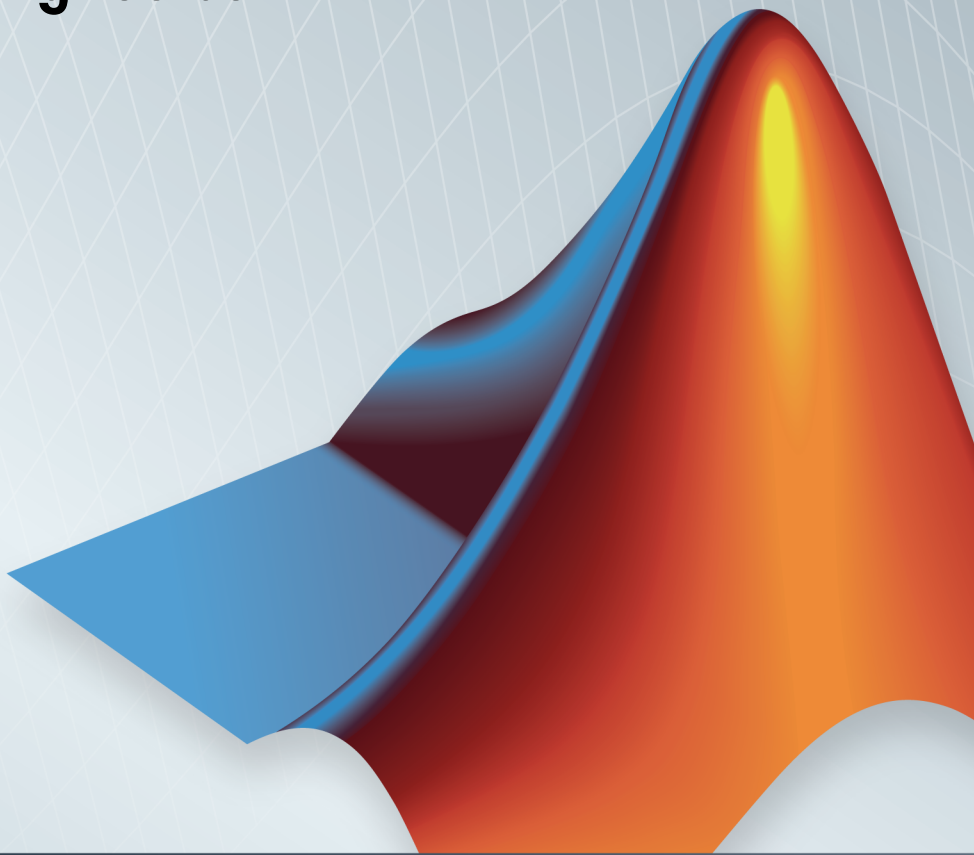


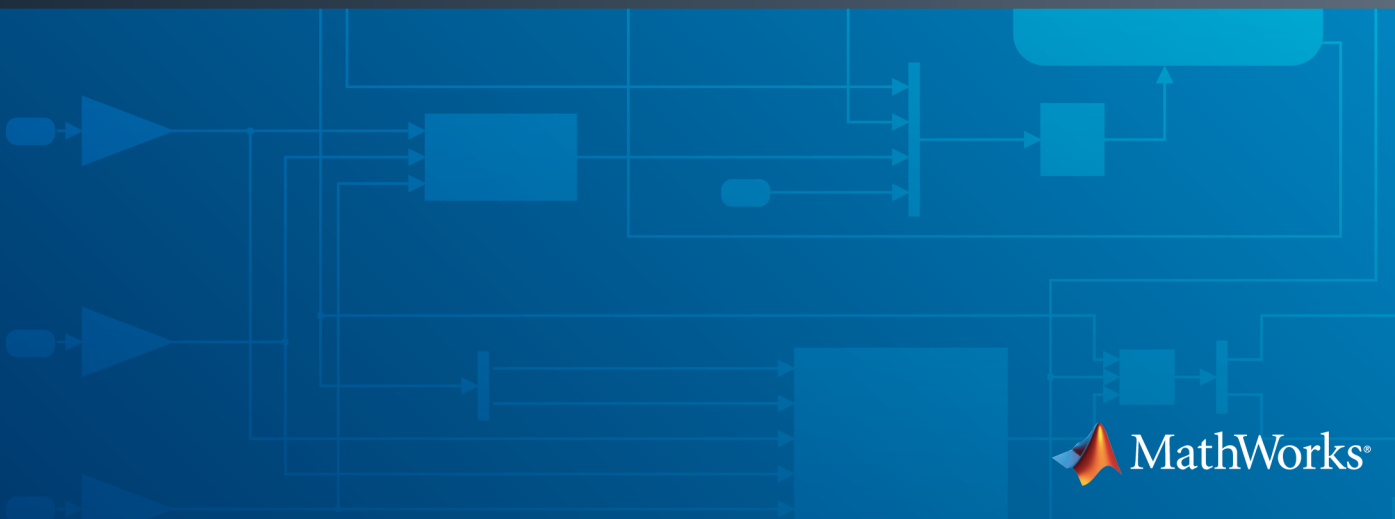
Image Processing Toolbox™

User's Guide

R2014b



MATLAB®



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Image Processing Toolbox™ User's Guide

© COPYRIGHT 1993–2014 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 and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see 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)
March 2007	Online only	Revised for Version 5.4 (Release 2007a)
September 2007	Online only	Revised for Version 6.0 (Release 2007b)
March 2008	Online only	Revised for Version 6.1 (Release 2008a)
October 2008	Online only	Revised for Version 6.2 (Release 2008b)
March 2009	Online only	Revised for Version 6.3 (Release 2009a)
September 2009	Online only	Revised for Version 6.4 (Release 2009b)
March 2010	Online only	Revised for Version 7.0 (Release 2010a)
September 2010	Online only	Revised for Version 7.1 (Release 2010b)
April 2011	Online only	Revised for Version 7.2 (Release 2011a)
September 2011	Online only	Revised for Version 7.3 (Release 2011b)
March 2012	Online only	Revised for Version 8.0 (Release 2012a)
September 2012	Online only	Revised for Version 8.1 (Release 2012b)
March 2013	Online only	Revised for Version 8.2 (Release 2013a)
September 2013	Online only	Revised for Version 8.3 (Release 2013b)
March 2014	Online only	Revised for Version 9.0 (Release 2014a)
October 2014	Online only	Revised for Version 9.1 (Release 2014b)

	Getting Started
1	
	<hr/>
	Image Processing Toolbox Product Description 1-2
	Key Features 1-2
	Configuration Notes 1-3
	Related Products 1-4
	Compilability 1-5
	Basic Image Import, Processing, and Export 1-6
	Basic Image Enhancement and Analysis Techniques 1-13
	Getting Help 1-25
	Product Documentation 1-25
	Image Processing Examples 1-25
	MATLAB Newsgroup 1-26
	Acknowledgments 1-27

	Introduction
2	
	<hr/>
	Images in MATLAB 2-2
	Expressing Image Locations 2-3
	Pixel Indices 2-3
	Spatial Coordinates 2-4

Image Types in the Toolbox	2-9
Overview of Image Types	2-9
Binary Images	2-10
Indexed Images	2-10
Grayscale Images	2-11
Truecolor Images	2-12
Converting Between Image Types	2-16
Converting Between Image Classes	2-18
Overview of Image Class Conversions	2-18
Losing Information in Conversions	2-18
Converting Indexed Images	2-18
Process Multi-Frame Image Arrays	2-20
Perform an Operation on a Sequence of Images	2-21
What is an Image Sequence?	2-22
Toolbox Functions That Work with Image Sequences	2-23
Image Arithmetic	2-26
Overview of Image Arithmetic Functions	2-26
Image Arithmetic Saturation Rules	2-26
Nesting Calls to Image Arithmetic Functions	2-27

Reading and Writing Image Data

3

Getting Information About Graphics Files	3-2
Reading Image Data	3-3
Writing Image Data to Files	3-5
Specify Format-Specific Parameters	3-5
Read and Write 1-Bit Binary Images	3-6
Determine Storage Class of Output Files	3-6
Convert Between Graphics File Formats	3-8

DICOM Files	3-9
Overview of DICOM Support	3-9
Read Metadata from a DICOM File	3-10
Read Image Data from DICOM Files	3-11
Write Data to DICOM File	3-12
Mayo Analyze 7.5 Files	3-18
Interfile Files	3-19
High Dynamic Range Images	3-20
Dynamic Range	3-20
Read High Dynamic Range Image	3-20
Create High Dynamic Range Image	3-21
Display High Dynamic Range Image	3-21
Write High Dynamic Range Image to File	3-23

Displaying and Exploring Images

4

Image Display and Exploration Overview	4-2
Displaying Images Using the imshow Function	4-4
Overview	4-4
Specifying the Initial Image Magnification	4-6
Controlling the Appearance of the Figure	4-7
Displaying Each Image in a Separate Figure	4-9
Displaying Multiple Images in the Same Figure	4-9
Using the Image Viewer App to Explore Images	4-13
Image Viewer App Overview	4-13
Opening the Image Viewer App	4-15
Specifying the Initial Image Magnification	4-15
Specifying the Colormap	4-16
Importing Image Data from the Workspace	4-17
Exporting Image Data to the Workspace	4-18
Saving Image Data Displayed in Image Viewer	4-19
Closing the Image Viewer App	4-20
Printing Images Displayed in Image Viewer App	4-21

Explore Very Large Images	4-22
Overview	4-22
Creating an R-Set File	4-22
Opening an R-Set File	4-22
Explore Images with the Image Viewer App	4-23
Explore Images Using the Overview Tool	4-23
Pan Images Displayed in Image Viewer App	4-26
Zoom Images in the Image Viewer App	4-26
Specify Image Magnification in Image Viewer	4-27
Get Pixel Information in Image Viewer App	4-30
Determine Individual Pixel Values in Image Viewer	4-30
Determine Pixel Values in an Image Region	4-31
Determine Image Display Range in Image Viewer	4-35
Measure Distance Between Pixels in Image Viewer	4-38
Determine Distance Between Pixels Using Distance Tool ..	4-38
Export Endpoint and Distance Data	4-40
Customize the Appearance of the Distance Tool	4-40
Get Image Information in Image Viewer App	4-42
Adjust Image Contrast In Image Viewer App	4-44
Understanding Contrast Adjustment	4-44
Open the Adjust Contrast Tool	4-45
Adjust Image Contrast Using the Histogram Window	4-48
Adjust Image Contrast Using Window/Level Tool	4-49
Make Contrast Adjustments Permanent	4-52
Crop Image Using Image Viewer App	4-54
View Image Sequences in Video Viewer App	4-59
View MRI Sequence Using Video Viewer App	4-59
Configure Video Viewer App	4-62
Specifying the Frame Rate	4-65
Specify Color Map	4-66
Get Information about an Image Sequence	4-66
View Image Sequence as Montage	4-68
Convert Multiframe Image to Movie	4-70

Display Different Image Types	4-71
Display Indexed Images	4-71
Display Grayscale Images	4-72
Display Binary Images	4-73
Display Truecolor Images	4-75
 Add Colorbar to Displayed Image	 4-77
 Print Images	 4-79
Handle Graphics Properties That Impact Printing	4-79
 Image Processing Toolbox Preferences	 4-80
Retrieve Values of Toolbox Preferences	4-80
Set Values of Toolbox Preferences	4-80

Building GUIs with Modular Tools

5

Modular Interactive Tools	5-2
 Displaying the Target Image	 5-9
 Creating the Modular Tools	 5-10
Overview	5-10
Associating Modular Tools with a Particular Image	5-11
Getting the Handle of the Target Image	5-13
Specifying the Parent of a Modular Tool	5-14
Positioning the Modular Tools in a GUI	5-18
Build a Pixel Information GUI	5-20
Adding Navigation Aids to a GUI	5-22
 Customizing Modular Tool Interactivity	 5-29
Overview	5-29
Build Image Comparison Tool	5-29
 Creating Your Own Modular Tools	 5-34
Overview	5-34
Create Angle Measurement Tool	5-35

Geometric Transformations

6

Resize an Image	6-2
Overview	6-2
Specify the Interpolation Method	6-3
Prevent Aliasing by Using Filters	6-3
Rotate an Image	6-5
Crop an Image	6-6
Translate an Image	6-8
2-D Geometric Transformations	6-11
Define the Geometric Transformation	6-12
Perform the Geometric Transformation	6-14
Understand Geometric Transformation	6-15
Specify Fill Values	6-15
Perform Simple 2-D Translation Transformation	6-18
N-Dimensional Spatial Transformations	6-22
Register Two Images Using Spatial Referencing to Enhance Display	6-24

Image Registration

7

Image Registration Techniques	7-2
Control Point Registration	7-3
Using <code>cpselect</code> in a Script	7-5
Register an Aerial Photograph to a Digital Orthophoto	7-6
Geometric Transformation Types	7-11

Control Point Selection Procedure	7-12
Start the Control Point Selection Tool	7-14
Find Visual Elements Common to Both Images	7-16
Using Scroll Bars to View Other Parts of an Image	7-16
Using the Detail Rectangle to Change the View	7-16
Panning the Image Displayed in the Detail Window	7-17
Zooming In and Out on an Image	7-17
Specifying the Magnification of the Images	7-18
Locking the Relative Magnification of the Moving and Fixed Images	7-19
Select Matching Control Point Pairs	7-20
Picking Control Point Pairs Manually	7-20
Using Control Point Prediction	7-22
Moving Control Points	7-25
Deleting Control Points	7-26
Export Control Points to the Workspace	7-27
Save Control Point Selection Session	7-28
Use Cross-Correlation to Improve Control Point Placement	7-29
Use Phase Correlation as Preprocessing Step in Registration	7-30
Intensity-Based Automatic Image Registration	7-36
Registering Multimodal MRI Images	7-39

Designing and Implementing Linear Filters for Image Data

8

What Is Image Filtering in the Spatial Domain?	8-2
Convolution	8-2
Correlation	8-3

Filter Images Using imfilter	8-5
How imfilter Handles Data Types	8-7
imfilter Boundary Padding Options	8-9
Filter Images Using imfilter with Convolution	8-13
Filter Images Using Predefined Filters	8-14
Filter Multidimensional Images with imfilter	8-17
What is Guided Image Filtering?	8-19
Perform Flash/No-flash Denoising with Guided Filter	8-20
Designing Linear Filters in the Frequency Domain	8-25
Transform 1-D FIR Filter to 2-D FIR Filter	8-25
Frequency Sampling Method	8-28
Windowing Method	8-29
Creating the Desired Frequency Response Matrix	8-30
Computing the Frequency Response of a Filter	8-31
Two-Dimensional Finite Impulse Response (FIR) Filters ..	8-33

Transforms

9

Fourier Transform	9-2
Definition of Fourier Transform	9-2
Discrete Fourier Transform	9-7
Applications of the Fourier Transform	9-10
Discrete Cosine Transform	9-15
DCT Definition	9-15
The DCT Transform Matrix	9-17
DCT and Image Compression	9-17
Radon Transform	9-19
Radon Transformation Definition	9-19

Plotting the Radon Transform	9-22
Viewing the Radon Transform as an Image	9-24
Detecting Lines Using the Radon Transform	9-25
The Inverse Radon Transformation	9-29
Inverse Radon Transform Definition	9-29
Reconstructing an Image from Parallel Projection Data	9-31
Fan-Beam Projection Data	9-35
Fan-Beam Projection Data Definition	9-35
Computing Fan-Beam Projection Data	9-36
Reconstructing an Image from Fan-Beam Projection Data	9-38
Reconstruct Image using Inverse Fanbeam Projection	9-39

Morphological Operations

10

Morphology Fundamentals: Dilation and Erosion	10-2
Understanding Dilation and Erosion	10-2
Understanding Structuring Elements	10-5
Dilating an Image	10-8
Eroding an Image	10-10
Combining Dilation and Erosion	10-11
Dilation- and Erosion-Based Functions	10-13
Morphological Reconstruction	10-16
Understanding Morphological Reconstruction	10-16
Understanding the Marker and Mask	10-18
Pixel Connectivity	10-19
Flood-Fill Operations	10-22
Finding Peaks and Valleys	10-25
Distance Transform	10-33
Labeling and Measuring Objects in a Binary Image	10-35
Understanding Connected-Component Labeling	10-35
Selecting Objects in a Binary Image	10-37
Finding the Area of the Foreground of a Binary Image	10-38
Finding the Euler Number of a Binary Image	10-38

Lookup Table Operations	10-40
Creating a Lookup Table	10-40
Using a Lookup Table	10-40

Analyzing and Enhancing Images

11

Pixel Values	11-2
Determine Values of Individual Pixels in Images	11-2
Intensity Profile of Images	11-4
Create an Intensity Profile of an Image	11-4
Create Intensity Profile of an RGB Image	11-5
Contour Plot of Image Data	11-8
Create Contour Plot of Image Data	11-8
Create Image Histogram	11-11
Image Mean, Standard Deviation, and Correlation	
Coefficient	11-14
Edge Detection	11-15
Detect Edges in Images	11-15
Corner Detection	11-18
Find Corners in Images	11-18
Boundary Tracing in Images	11-21
Trace Boundaries of Objects in Images	11-21
Select First Step and Direction for Tracing	11-26
Hough Transform	11-28
Detect Lines in Images	11-28
Quadtree Decomposition	11-33
Determine Image Homogeneity Using Quadtree	
Decomposition	11-33

Texture Analysis	11-35
Understanding How Texture Filter Functions Work	11-35
Detect Regions of Texture in Images	11-37
Gray-Level Co-Occurrence Matrix (GLCM)	11-38
Understanding a Gray-Level Co-Occurrence Matrix	11-38
Specify Offset Used in GLCM Calculation	11-39
Derive Statistics from GLCM and Plot Correlation	11-40
Contrast Adjustment	11-43
Adjust Intensity Values to Specified Range	11-44
Adjust Image Intensity Values Using Histogram Equalization	11-48
Adjust Contrast Using Contrast-Limited Adaptive Histogram Equalization (CLAHE)	11-49
Enhance Color Separation Using Decorrelation Stretching	11-50
Noise Removal	11-55
Remove Noise By Linear Filtering	11-55
Remove Noise Using Median Filtering	11-55
Remove Noise By Adaptive Filtering	11-58
Image Segmentation	11-61
Image Segmentation Using the Color Thesholder App ...	11-62
Open Image in Color Thesholder App	11-62
Segment Image Using Color Selector	11-67
Segment Image Using Color Component Controls	11-70
Create an Image Mask	11-74
Image Quality Metrics	11-80
Obtain Local Structural Similarity Index	11-80
Compare Image Quality at Various Compression Levels ..	11-82
Image Segmentation Using the Image Segmenter App ...	11-85
Open Image Segmenter App	11-85
Create the Initial Segmentation	11-88
Evolve and Refine the Segmentation	11-91
Image Region Properties	11-99
Calculate Region Properties Using Image Region Analyzer	11-100

Filter Images on Region Properties Using Image Region Analyzer App	11-108
---	---------------

ROI-Based Processing

12

Specifying a Region of Interest (ROI)	12-2
Overview of ROI Processing	12-2
Creating a Binary Mask	12-2
Creating an ROI Without an Associated Image	12-3
Creating an ROI Based on Color Values	12-4
Filtering an ROI	12-5
Overview of ROI Filtering	12-5
Filtering a Region in an Image	12-5
Specifying the Filtering Operation	12-6
Filling an ROI	12-8

Image Deblurring

13

Understanding Deblurring	13-2
Causes of Blurring	13-2
Deblurring Model	13-2
Deblurring Functions	13-4
Deblurring with the Wiener Filter	13-6
Refining the Result	13-6
Deblurring with a Regularized Filter	13-7
Refining the Result	13-8
Deblurring with the Lucy-Richardson Algorithm	13-9
Overview	13-9
Reducing the Effect of Noise Amplification	13-9
Accounting for Nonuniform Image Quality	13-10

Handling Camera Read-Out Noise	13-10
Handling Undersampled Images	13-10
Example: Using the deconvlucy Function to Deblur an Image	13-11
Refining the Result	13-13
Deblurring with the Blind Deconvolution Algorithm	13-14
Example: Using the deconvblind Function to Deblur an Image	13-14
Refining the Result	13-19
Creating Your Own Deblurring Functions	13-20
Avoiding Ringing in Deblurred Images	13-21

Color

14

Displaying Colors	14-2
Reducing the Number of Colors in an Image	14-4
Reducing Colors Using Color Approximation	14-4
Reducing Colors Using imapprox	14-9
Dithering	14-10
Convert from YIQ to RGB Color Space	14-12
Convert from YCbCr to RGB Color Space	14-13
Convert from HSV to RGB Color Space	14-14
Profile-Based Color Space Conversions	14-17
Read ICC Profiles	14-17
Write ICC Profile Information to a File	14-18
Convert RGB to CMYK Using ICC Profiles	14-18
What is Rendering Intent in Profile-Based Conversions? ..	14-20
Device-Independent Color Spaces	14-21
Convert Between Device-Independent Color Spaces	14-21
Color Space Data Encodings	14-22

Understanding Color Spaces and Color Space Conversion	14-24
Determine if L*a*b* value is in RGB gamut	14-25

Neighborhood and Block Operations

15

Neighborhood or Block Processing: An Overview	15-2
Performing Sliding Neighborhood Operations	15-3
Understanding Sliding Neighborhood Processing	15-3
Implementing Linear and Nonlinear Filtering as Sliding Neighborhood Operations	15-5
Performing Distinct Block Operations	15-7
Understanding Distinct Block Processing	15-7
Implementing Block Processing Using the <code>blockproc</code> Function	15-8
Applying Padding	15-9
Block Size and Performance	15-11
Using Parallel Block Processing on large Image Files	15-13
Working with Data in Unsupported Formats	15-15
Using Columnwise Processing to Speed Up Sliding Neighborhood or Distinct Block Operations	15-23
Understanding Columnwise Processing	15-23
Using Column Processing with Sliding Neighborhood Operations	15-23
Using Column Processing with Distinct Block Operations	15-24

Code Generation for Image Processing Toolbox Functions

16

Code Generation for Image Processing	16-2
List of Supported Functions with Usage Notes	16-3

Generate Code from Application Containing Image	
Processing Functions	16-11
Setup Your Compiler	16-11
Prepare Your MATLAB Code for Code Generation	16-12
Generate Code	16-18
Code Generation Using a Shared Library	16-25

GPU Computing with Image Processing Toolbox Functions

17

Image Processing on a GPU	17-2
List of Supported Functions with Limitations and Other Notes	17-4
Perform Thresholding and Morphological Operations on a GPU	17-7
Perform Element-wise Operations on a GPU	17-11

Getting Started

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

- “Image Processing Toolbox Product Description” on page 1-2
- “Configuration Notes” on page 1-3
- “Related Products” on page 1-4
- “Compilability” on page 1-5
- “Basic Image Import, Processing, and Export” on page 1-6
- “Basic Image Enhancement and Analysis Techniques” on page 1-13
- “Getting Help” on page 1-25
- “Acknowledgments” on page 1-27

Image Processing Toolbox Product Description

Perform image processing, analysis, and algorithm development

Image Processing Toolbox provides a comprehensive set of reference-standard algorithms, functions, and apps for image processing, analysis, visualization, and algorithm development. You can perform image analysis, image segmentation, image enhancement, noise reduction, geometric transformations, and image registration. Many toolbox functions support multicore processors, GPUs, and C-code generation.

Image Processing Toolbox supports a diverse set of image types, including high dynamic range, gigapixel resolution, embedded ICC profile, and tomographic. Visualization functions and apps let you explore images and videos, examine a region of pixels, adjust color and contrast, create contours or histograms, and manipulate regions of interest (ROIs). The toolbox supports workflows for processing, displaying, and navigating large images.

Key Features

- Image analysis, including segmentation, morphology, statistics, and measurement
- Image enhancement, filtering, and deblurring
- Geometric transformations and intensity-based image registration methods
- Image transforms, including FFT, DCT, Radon, and fan-beam projection
- Large image workflows, including block processing, tiling, and multiresolution display
- Visualization apps, including Image Viewer and Video Viewer
- Multicore- and GPU-enabled functions, and C-code generation support

Configuration Notes

To determine if the Image Processing Toolbox software 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 a list of the new features in this version of the toolbox, see the Release Notes documentation.

Many of the toolbox functions are MATLAB files with 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 toolbox by writing your own files, or by using the toolbox in combination with other toolboxes, such as the Signal Processing Toolbox™ software and the Wavelet Toolbox™ software.

For information about installing the toolbox, see the installation guide.

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).

Related Products

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

Compilability

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

- `cpselect`
- `implay`
- `imtool`

Basic Image Import, Processing, and Export

This example shows how to read an image into the workspace, adjust the contrast in the image, and then write the adjusted image to a file.

Step 1: Read and Display an Image

First, clear the workspace of any variables and close open figure windows.

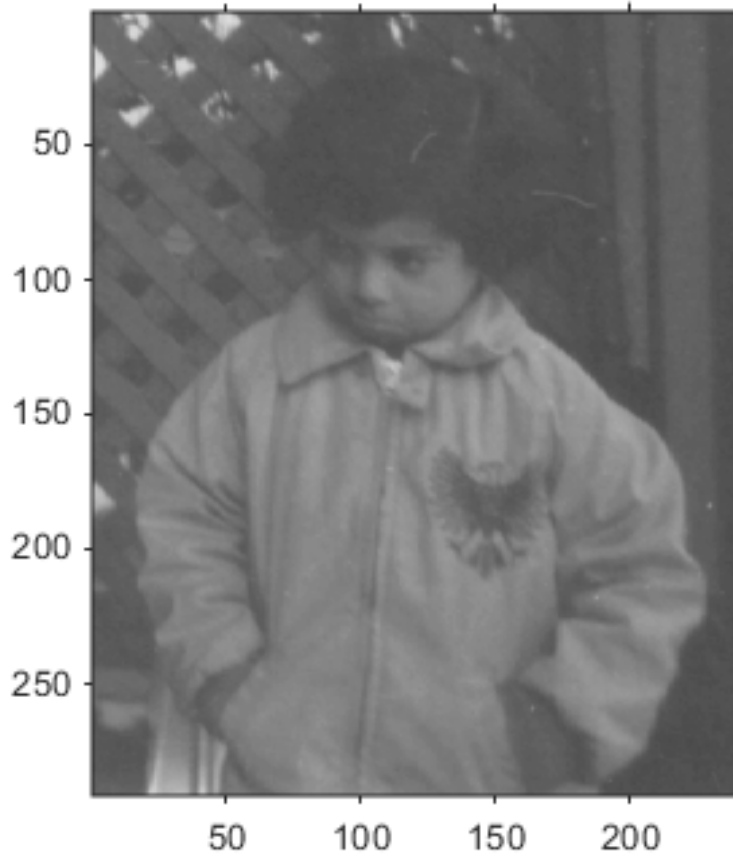
```
clear  
close all
```

Read an image into the workspace, using the `imread` command. The example reads one of the sample images included with the toolbox, an image of a young girl in a file named `pout.tif`, and stores it in an array named `I`. `imread` infers from the file that the graphics file format is Tagged Image File Format (TIFF).

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

Display the image, using the `imshow` function. You can also view an image in the Image Viewer app. The `imtool` function opens the Image Viewer app which presents an integrated environment for displaying images and performing some common image processing tasks. The Image Viewer app 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.

```
imshow(I)
```

Step 2: Check How the Image Appears in the Workspace

Check how the `imread` function stores the image data in the workspace, using the `whos` command. You can also check the variable in the Workspace Browser. The `imread` function returns the image data in the variable `I`, which is a 291-by-240 element array of `uint8` data.

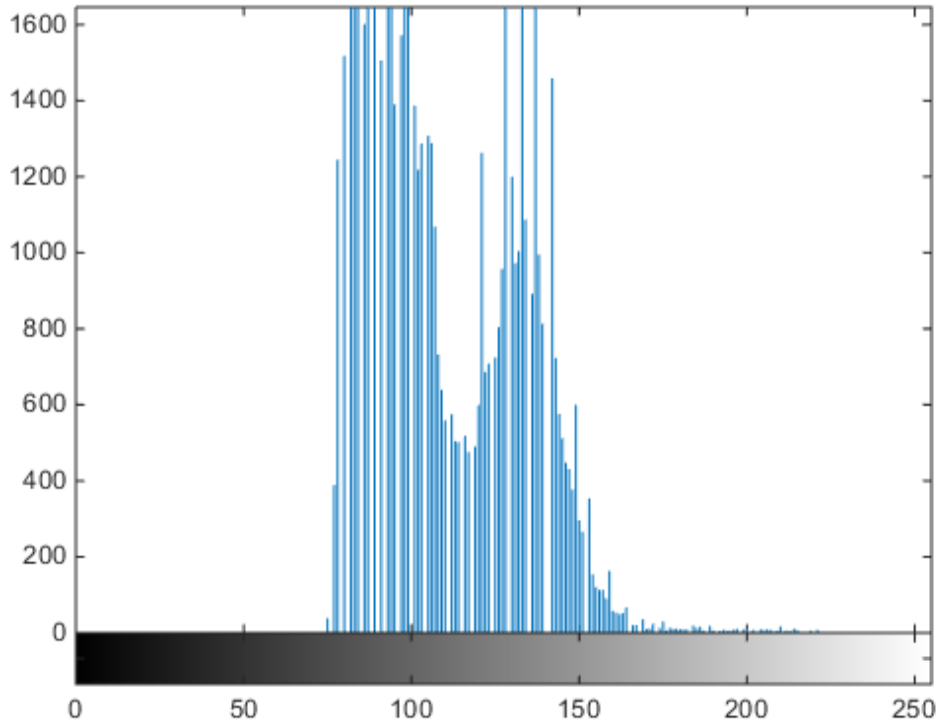
```
whos
```

Name	Size	Bytes	Class	Attributes
I	291x240	69840	uint8	

Step 3: Improve Image Contrast

View the distribution of image pixel intensities. The image `pout.tif` is a somewhat low contrast image. To see the distribution of intensities in the image, 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.) Notice how the histogram indicates that the intensity range of the image is rather narrow. The range does not cover the potential range of `[0, 255]`, and is missing the high and low values that would result in good contrast.

```
figure  
imhist(I)
```



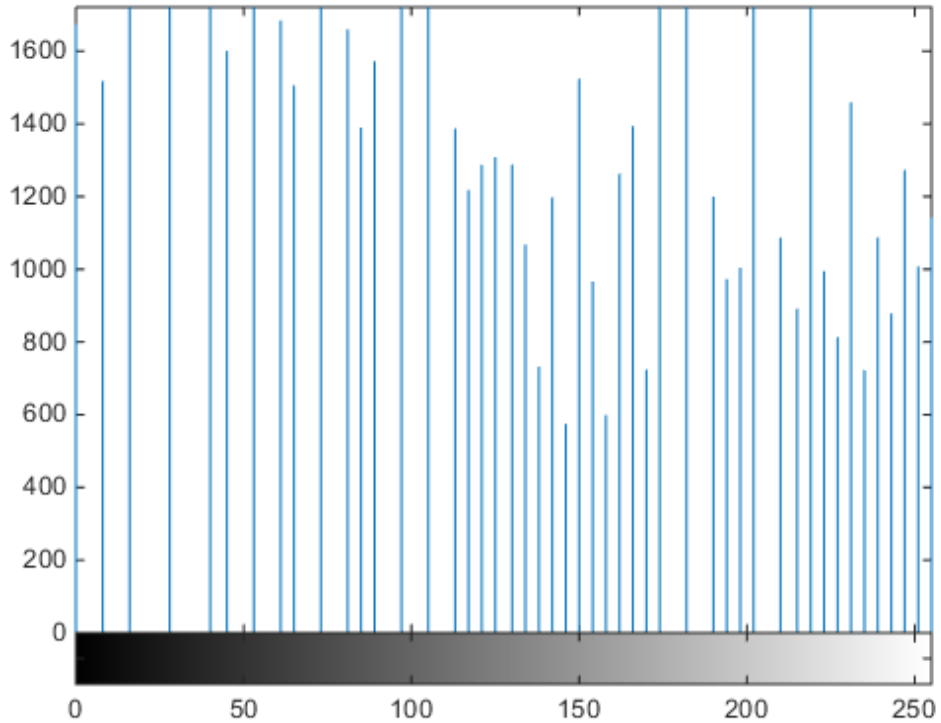
Improve the contrast in an image, using the `histeq` function. Histogram equalization spreads the intensity values over the full range of the image. Display the image. (The toolbox includes several other functions that perform contrast adjustment, including `imadjust` and `adaphisteq`, and interactive tools such as the Adjust Contrast tool, available in the Image Viewer.)

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



Call the `imhist` function again to create a histogram of the equalized image `I2` . If you compare the two histograms, you can see that the histogram of `I2` is more spread out over the entire range than the histogram of `I` .

```
figure  
imhist(I2)
```



Step 4: Write the Adjusted Image to a Disk File

Write the newly adjusted image `I2` to a disk file, using the `imwrite` function. This example includes the filename extension `'.png'` in the file name, so 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');
```

Step 5: Check the Contents of the Newly Written File

View what `imwrite` wrote to the disk file, using the `imfinfo` function. The `imfinfo` function returns information about the image in the file, such as its format, size, width, and height.

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

```
ans =
```

```
      Filename: 'C:\TEMP\Bdoc14b_152206_6100\tp7d117baf_15bf_4...'
      FileModDate: '20-Sep-2014 00:42:07'
      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: '20 Sep 2014 04:42:07 +0000'
      Title: []
      Author: []
      Description: []
      Copyright: []
      CreationTime: []
      Software: []
      Disclaimer: []
      Warning: []
      Source: []
      Comment: []
      OtherText: []
```

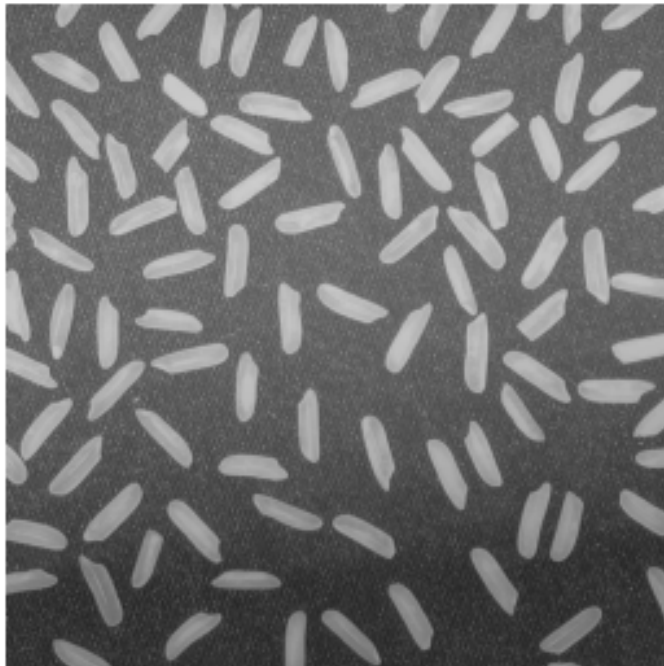
Basic Image Enhancement and Analysis Techniques

This example shows how to enhance an image as a preprocessing step before analysis. In this example, you correct the nonuniform background illumination and convert the image into a binary image so that you can perform analysis of the image foreground objects.

Step 1: Read the Image into the Workspace

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

```
I = imread('rice.png');  
imshow(I)
```



Step 2: Preprocess the Image to Enable Analysis

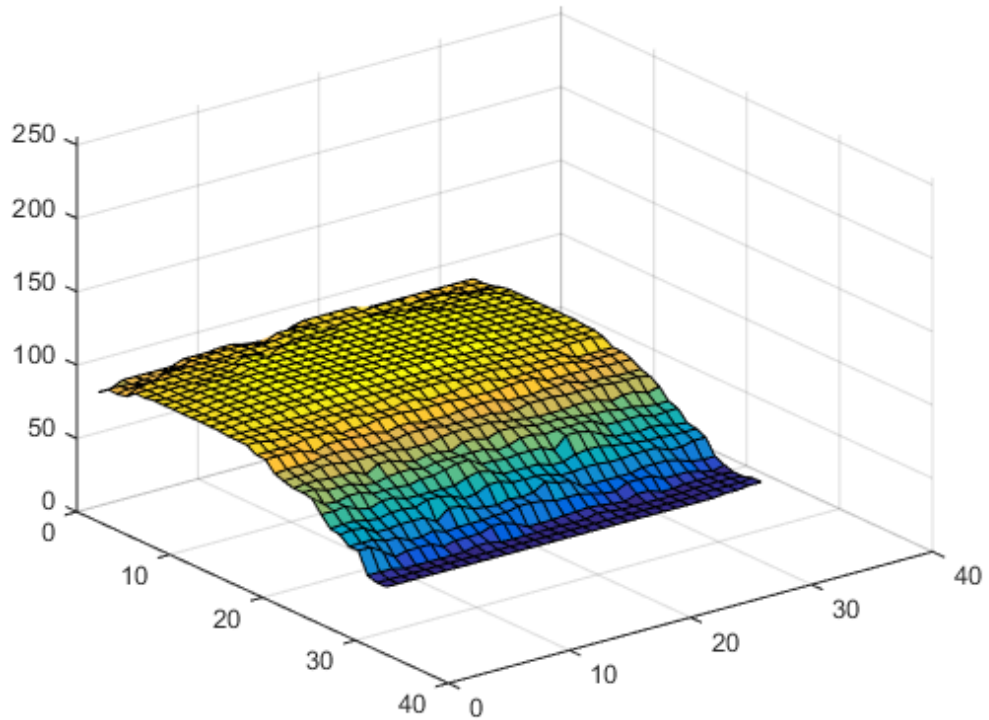
In the sample image, the background illumination is brighter in the center of the image than at the bottom. As a preprocessing step before analysis, make the background uniform and then convert the image into a binary image. To make the background illumination more uniform, create an approximation of the background as a separate image and then subtract this approximation from the original image.

As a first step to creating a background approximation image, remove all the foreground (rice grains) using morphological opening. The opening operation has the effect of removing objects that cannot completely contain the structuring element. 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. The example calls the `strel` function to create a disk-shaped structuring element with a radius of 15.

```
background = imopen(I,strel('disk',15));
```

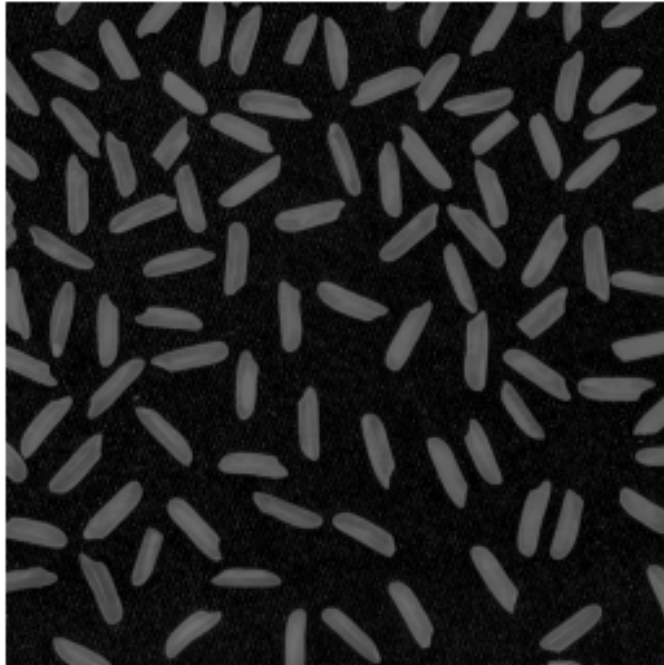
View the background approximation image as a surface to see where illumination varies. The `surf` command creates colored parametric surfaces that enable you to view mathematical functions over a rectangular region. Because the `surf` function requires data of class `double`, you first need to convert `background` using the `double` command. The example uses 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.

```
figure
surf(double(background(1:8:end,1:8:end))),zlim([0 255]);
set(gca,'ydir','reverse');
```

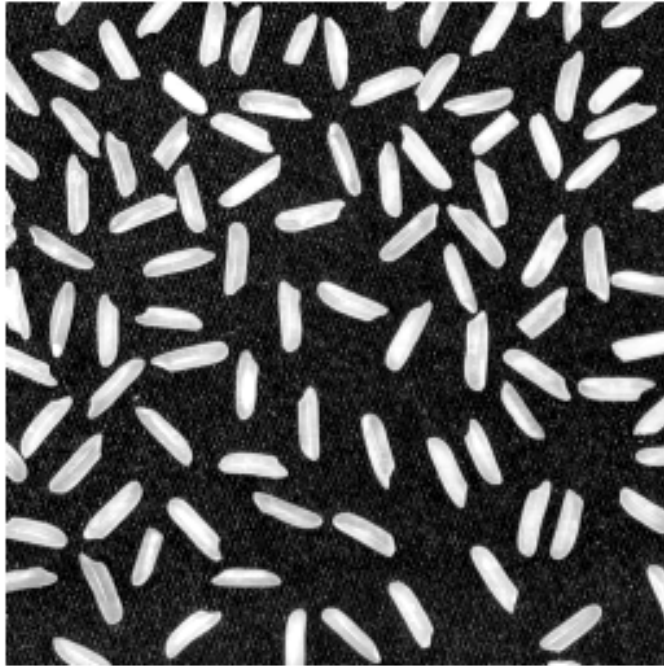
Subtract the background approximation image, `background`, from the original image, `I`, and view the resulting image. After subtracting the adjusted background image from the original image, the resulting image has a uniform background but is now a bit dark for analysis

```
I2 = I - background;  
imshow(I2)
```



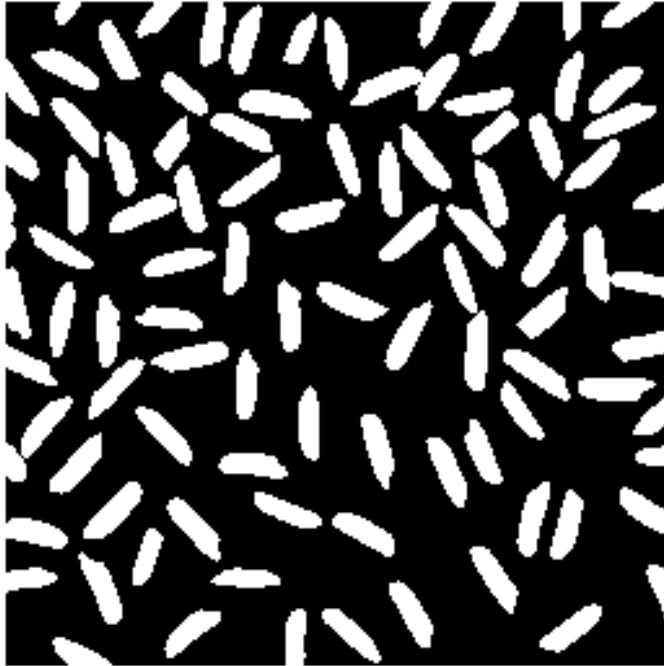
Use `imadjust` to increase the contrast of the processed image `I2` by saturating 1% of the data at both low and high intensities and by stretching the intensity values to fill the `uint8` dynamic range.

```
I3 = imadjust(I2);  
imshow(I3);
```



Create a binary version of the processed image so you can use toolbox functions for analysis. 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. Remove background noise with the `bwareaopen` function.

```
level = graythresh(I3);  
bw = im2bw(I3,level);  
bw = bwareaopen(bw, 50);  
imshow(bw)
```



Step 3: Perform Analysis of Objects in the Image

Now that you have created a binary version of the original image you can perform analysis of objects in the image.

Find all the connected components (objects) in the binary image. The accuracy of your results depends on the size of the objects, the connectivity parameter (4, 8, or arbitrary), and whether or not any objects are touching (in which case they could be labeled as one object). Some of the rice grains in the binary image `bw` are touching.

```
cc = bwconncomp(bw, 4)
```

```
cc.NumObjects
```

```
cc =
```

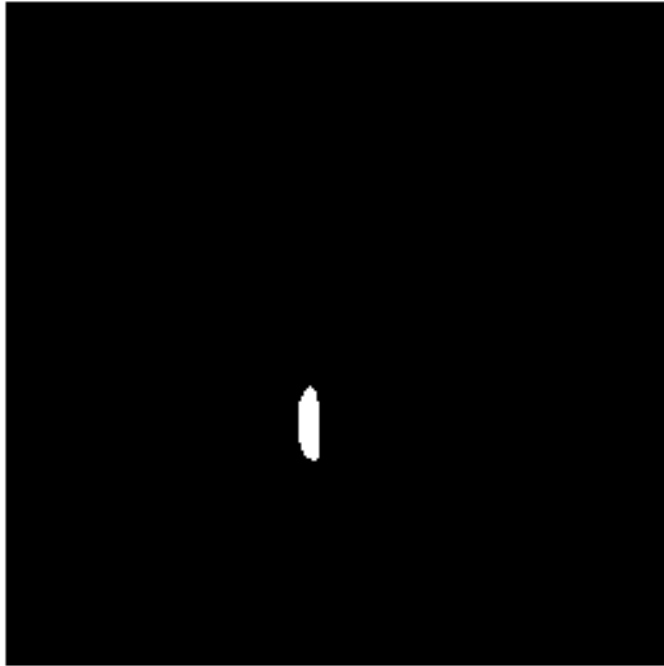
```
Connectivity: 4  
ImageSize: [256 256]  
NumObjects: 95  
PixelIdxList: {1x95 cell}
```

```
ans =
```

```
95
```

View the rice grain that is labeled 50 in the image.

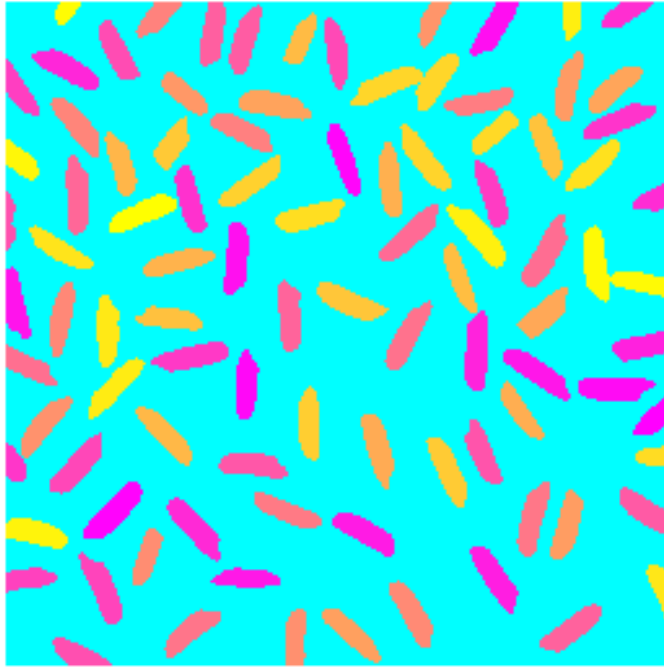
```
grain = false(size(bw));  
grain(cc.PixelIdxList{50}) = true;  
imshow(grain);
```



Visualize all the connected components in the image. First, create a label matrix, and then display the label matrix as a pseudocolor indexed image. Use `labelmatrix` to create a label matrix from the output of `bwconncomp`. Note that `labelmatrix` stores the label matrix in the smallest numeric class necessary for the number of objects. Since `bw` contains only 95 objects, the label matrix can be stored as `uint8`. In the pseudocolor image, the label identifying each object in the label matrix maps to a different color in an associated colormap. Use `label2rgb` to choose the colormap, the background color, and how objects in the label matrix map to colors in the colormap.

```
labeled = labelmatrix(cc);  
RGB_label = label2rgb(labeled, @spring, 'c', 'shuffle');
```

```
imshow(RGB_label)
```



Compute the area of each object in the image using `regionprops`. Each rice grain is one connected component in the `cc` structure.

```
graindata = regionprops(cc, 'basic')
```

```
graindata =
```

```
95x1 struct array with fields:
```

```
Area  
Centroid  
BoundingBox
```

Find the area of the 50th component, using dot notation to access the Area field in the 50th element of `graindata` .

```
graindata(50).Area
```

```
ans =
```

```
194
```

Create a vector `grain_areas` to hold the area measurement of each object (rice grain).

```
grain_areas = [graindata.Area];
```

Find the rice grain with the smallest area.

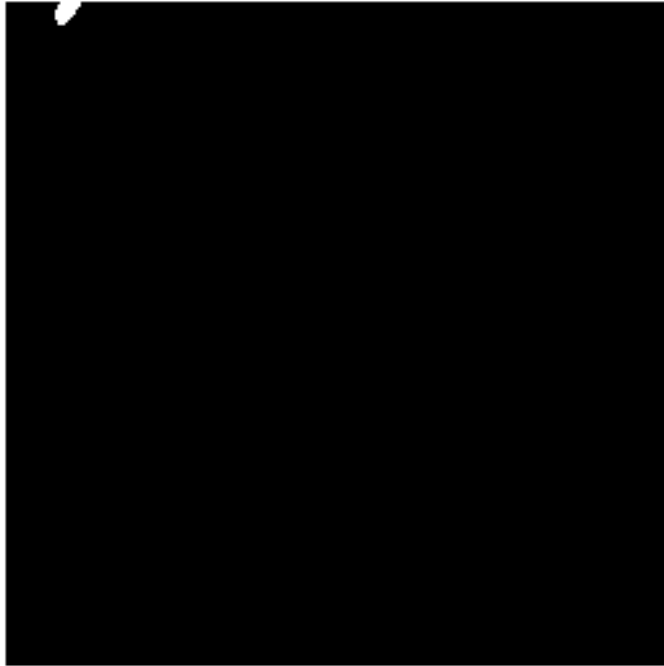
```
[min_area, idx] = min(grain_areas)  
grain = false(size(bw));  
grain(cc.PixelIdxList{idx}) = true;  
imshow(grain);
```

```
min_area =
```

```
61
```

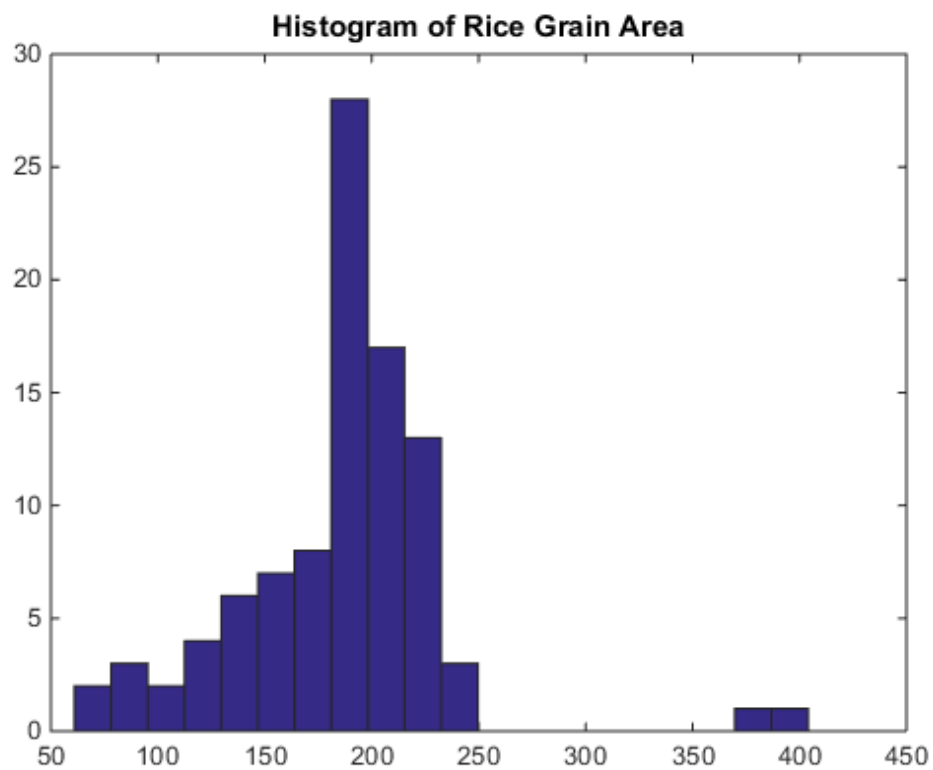
```
idx =
```

```
16
```

Using the `hist` command to create a histogram of rice grain areas.

```
nbins = 20;  
figure, hist(grain_areas, nbins)  
title('Histogram of Rice Grain Area');
```



Getting Help

In this section...

“Product Documentation” on page 1-25

“Image Processing Examples” on page 1-25

“MATLAB Newsgroup” on page 1-26

Product Documentation

The Image Processing Toolbox 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 (PDF) Documentation on the Web. (Note that to view the PDF help, you must have Adobe® Acrobat® Reader installed.)

For reference information about any of the Image Processing Toolbox functions, type in the MATLAB command window

```
help functionname
```

For example,

```
help imtool
```

Image Processing Examples

The Image Processing Toolbox software is supported by a full complement of example 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 examples, call the `iptdemos` function. This displays an HTML page in the MATLAB Help browser that lists all the examples.

The toolbox examples are located in the folder

`matlabroot\toolbox\images\imdata`

where *matlabroot* represents your MATLAB installation folder.

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 MATLAB files with other MATLAB users.

Acknowledgments

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

Image	Source
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.
tissue	Courtesy of Alan W. Partin, M.D., Ph.D., 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 software.

- “Images in MATLAB” on page 2-2
- “Expressing Image Locations” on page 2-3
- “Image Types in the Toolbox” on page 2-9
- “Converting Between Image Types” on page 2-16
- “Converting Between Image Classes” on page 2-18
- “Process Multi-Frame Image Arrays” on page 2-20
- “Perform an Operation on a Sequence of Images” on page 2-21
- “What is an Image Sequence?” on page 2-22
- “Toolbox Functions That Work with Image Sequences” on page 2-23
- “Image Arithmetic” on page 2-26

Images in MATLAB

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.

Expressing Image Locations

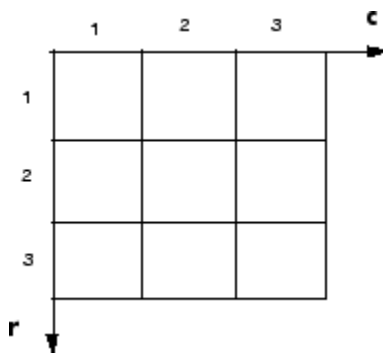
In this section...

“Pixel Indices” on page 2-3

“Spatial Coordinates” on page 2-4

Pixel Indices

Often, the most convenient method for expressing locations in an image is to use pixel indices. 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.



Pixel Indices

For pixel indices, the row increases downward, while the column increases to the right. Pixel indices are integer values, and range from 1 to the length of the row or column.

There is a one-to-one correspondence between pixel indices and subscripts for the first two matrix dimensions in MATLAB. 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**. Similarly, the MATLAB code

```
RGB(2,15,:)
```

returns the R, G, B values of the pixel at row 2, column 15 of the image RGB.

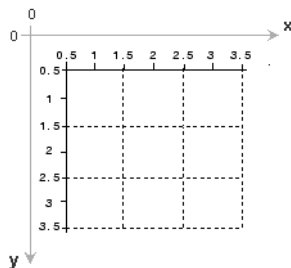
The correspondence between pixel indices and subscripts for the first two matrix dimensions in MATLAB makes the relationship between an image's data matrix and the way the image is displayed easy to understand.

Spatial Coordinates

Another method for expressing locations in an image is to use a system of continuously varying coordinates rather than discrete indices. This lets you consider an image as covering a square patch, for example. In a *spatial coordinate system* like this, locations in an image are positions on a plane, and they are described in terms of x and y (not row and column as in the pixel indexing system). From this Cartesian perspective, an (x,y) location such as $(3.2,5.3)$ is meaningful, and is distinct from pixel $(5,3)$.

Intrinsic Coordinates

By default, the toolbox uses a spatial coordinate system for an image that corresponds to the image's pixel indices. It's called the intrinsic coordinate system and is illustrated in the following figure. Notice that y increases downward, because this orientation is consistent with the way in which digital images are typically viewed.



Intrinsic Coordinate System

The intrinsic coordinates (x,y) of the center point of any pixel are identical to the column and row indices for that pixel. For example, the center point of the pixel in row 5, column 3 has spatial coordinates $x = 3.0$, $y = 5.0$. This correspondence simplifies many toolbox functions considerably. Be aware, however, that the order of coordinate specification $(3.0,5.0)$ is reversed in intrinsic coordinates relative to pixel indices $(5,3)$.

Several functions primarily work with spatial coordinates rather than pixel indices, but as long as you are using the default spatial coordinate system (intrinsic coordinates), you can specify locations in terms of their columns (x) and rows (y).

When looking at the intrinsic coordinate system, note that the upper left corner of the image is located at $(0.5, 0.5)$, not at $(0, 0)$, and the lower right corner of the image is located at $(\text{numCols} + 0.5, \text{numRows} + 0.5)$, where `numCols` and `numRows` are the number of rows and columns in the image. In contrast, the upper left pixel is pixel $(1, 1)$ and the lower right pixel is pixel $(\text{numRows}, \text{numCols})$. The center of the upper left pixel is $(1.0, 1.0)$ and the center of the lower right pixel is $(\text{numCols}, \text{numRows})$. In fact, the center coordinates of every pixel are integer valued. The center of the pixel with indices (r, c) — where r and c are integers by definition — falls at the point $x = c, y = r$ in the intrinsic coordinate system.

World Coordinates

In some situations, you might want to use a world coordinate system (also called a nondefault spatial coordinate system). For example, you could shift the origin by specifying that the upper left corner of an image is the point $(19.0, 7.5)$, rather than $(0.5, 0.5)$. Or, you might want to specify a coordinate system in which every pixel covers a 5-by-5 meter patch on the ground.

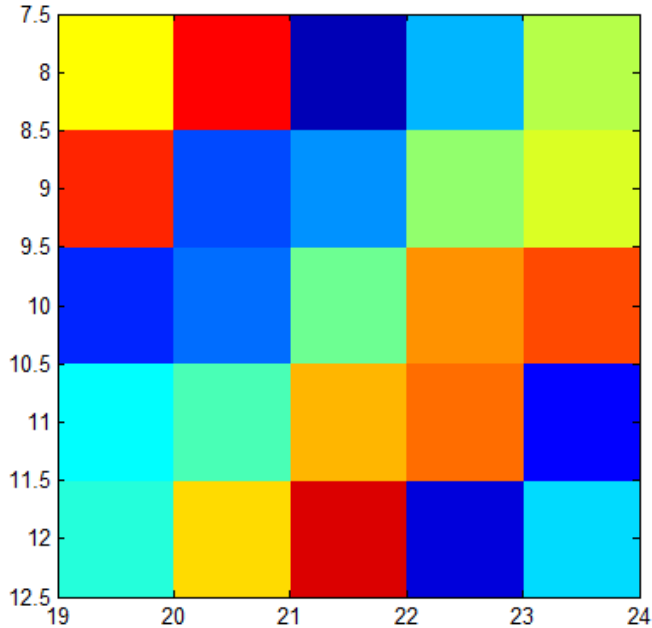
One way to define a world coordinate system for an image is to specify the `XData` and `YData` image properties for the image. The `XData` and `YData` image properties are two-element vectors that control the range of coordinates spanned by the image. When you do this, the MATLAB axes coordinates become identical to the world (nondefault) coordinates. If you do not specify `XData` and `YData`, the axes coordinates are identical to the intrinsic coordinates of the image. By default, for an image `A`, `XData` is `[1 size(A, 2)]`, and `YData` is `[1 size(A, 1)]`. With this default, the world coordinate system and intrinsic coordinate system coincide perfectly.

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 interval in X spanned by the image is `[0.5 200.5]`.

It's also possible to set `XData` or `YData` such that the x -axis or y -axis is reversed. You'd do this by placing the larger value first. (For example, set the `YData` to `[1000 1]`.) This is a common technique to use with geospatial data.

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))
```



Specifying Coordinate Information

To specify a world (nondefault spatial) coordinate system for an image, use the spatial referencing objects `imref2d` and `imref3d`. Spatial referencing objects let you define the location of the image in a world coordinate system and specify the image resolution, including nonsquare pixel shapes. These objects also support methods for converting between the world, intrinsic, and subscript coordinate systems. Several toolbox functions accept or return spatial referencing objects: `imwarp`, `imshow`, `imshowpair`, `imfuse`, `imregtform`, and `imregister`.

This example creates a spatial referencing object associated with a 2-by-2 image where the world extent is 4 units/pixel in the x direction and 2 units/pixel in the y direction. The example creates the object, specifying the pixels dimensions as arguments but does

not specify world limits in the **x** and **y** directions. You could specify other information when creating an object, see `imref2d` for more information.

```
I = [1 2; 3 4]
R = imref2d(size(I),4,2)
```

R =

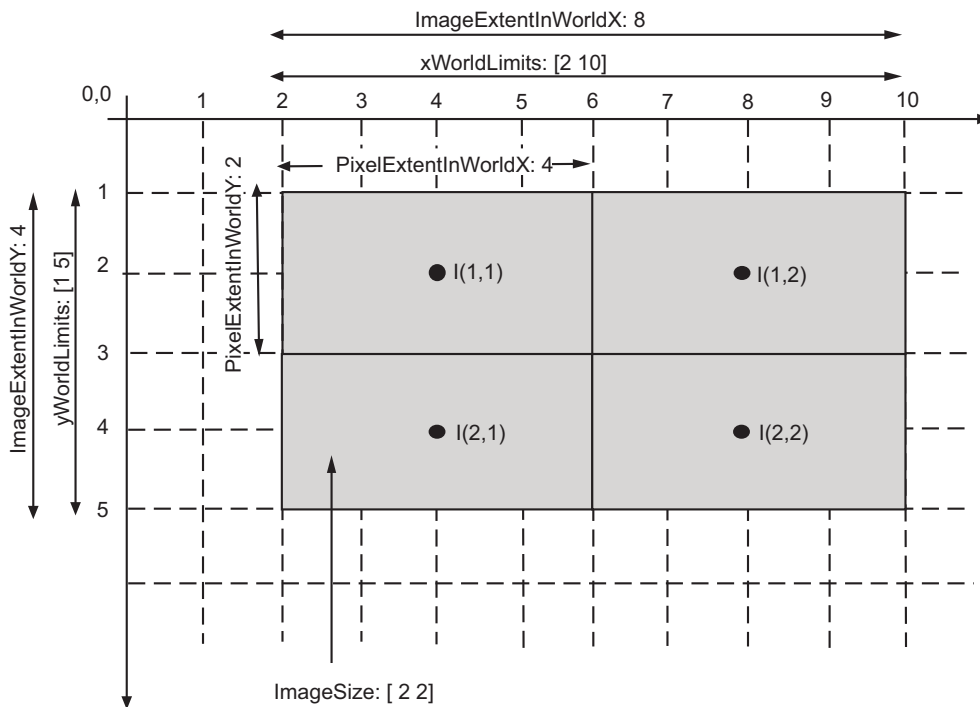
`imref2d` with properties:

```
    XWorldLimits: [2 10]
    YWorldLimits: [1 5]
    ImageSize: [2 2]
PixelExtentInWorldX: 4
PixelExtentInWorldY: 2
ImageExtentInWorldX: 8
ImageExtentInWorldY: 4
    XIntrinsicLimits: [0.5000 2.5000]
    YIntrinsicLimits: [0.5000 2.5000]
```

The `imref2d` object contains information about the image, some of it provided by you and some of it derived by the object. The following table provides descriptions of spatial referencing object fields.

Field	Description
<code>XWorldLimits</code>	Upper and lower bounds along the <i>X</i> dimension in world coordinates (nondefault spatial coordinates)
<code>YWorldLimits</code>	Upper and lower bounds along the <i>Y</i> dimension in world coordinates (nondefault spatial coordinates)
<code>ImageSize</code>	Size of the image, returned by the <code>size</code> function.
<code>PixelExtentInWorldX</code>	Size of pixel along the <i>X</i> dimension
<code>PixelExtentInWorldY</code>	Size of pixel along the <i>Y</i> dimension
<code>ImageExtentInWorldX</code>	Size of image along the <i>X</i> dimension
<code>ImageExtentInWorldY</code>	Size of image along the <i>Y</i> dimension
<code>XIntrinsicLimits</code>	Upper and lower bounds along <i>X</i> dimension in intrinsic coordinates (default spatial coordinates)
<code>YIntrinsicLimits</code>	Upper and lower bounds along <i>Y</i> dimension in intrinsic coordinates (default spatial coordinates).

The following figure illustrates how these properties map to elements of an image.



You can also use the `XData` and `YData` properties to define a world (nondefault spatial) coordinate system. Several toolbox functions accept this data as arguments and returns coordinates in the world coordinate system. These functions are: `bwselect`, `imcrop`, `impixel`, `roifill`, `roipoly`, and `imtransform`

Image Types in the Toolbox

In this section...
“Overview of Image Types” on page 2-9
“Binary Images” on page 2-10
“Indexed Images” on page 2-10
“Grayscale Images” on page 2-11
“Truecolor Images” on page 2-12

Overview of Image Types

The Image Processing Toolbox software 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. For information about converting between image types, see “Converting Between Image Types” on page 2-16.

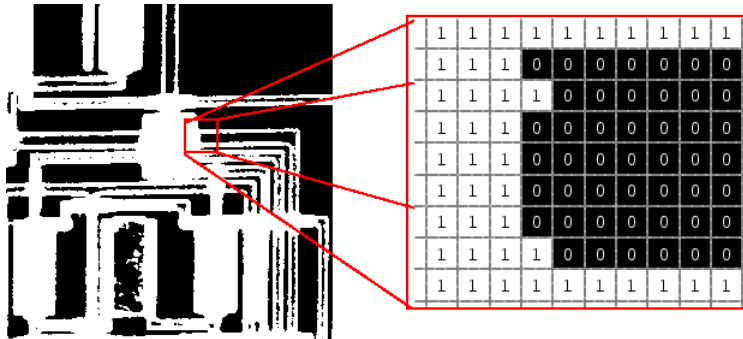
Image Type	Interpretation
Binary (Also known as a <i>bilevel</i> image)	Logical array containing only 0s and 1s, interpreted as black and white, respectively. See “Binary Images” on page 2-10 for more information.
Indexed (Also known as a <i>pseudocolor</i> image)	Array of class <code>logical</code> , <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 $[0, p-1]$. See “Indexed Images” on page 2-10 for more information.
Grayscale (Also known as an <i>intensity</i> , <i>gray scale</i> , or <i>gray level</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 $[-32768, 32767]$. See “Grayscale Images” on page 2-11 for more information.

Image Type	Interpretation
Truecolor (Also known as an <i>RGB</i> image)	<p>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.</p> <p>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]. See “Truecolor Images” on page 2-12 for more information.</p>

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.

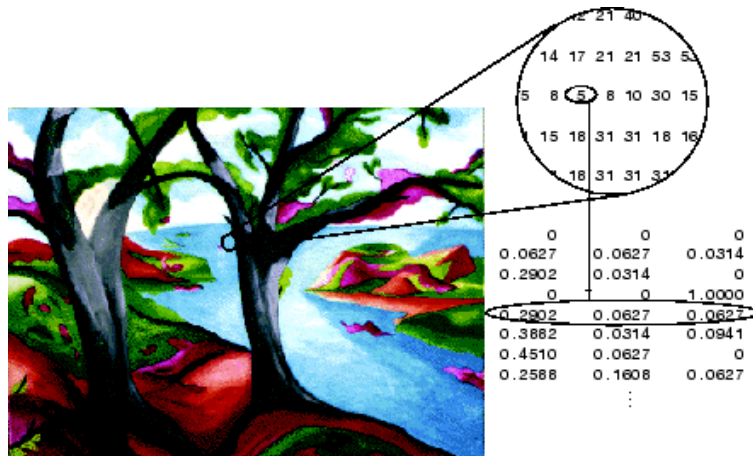


Image Courtesy of Susan Cohen

Pixel Values Index to Colormap Entries in Indexed Images

Grayscale Images

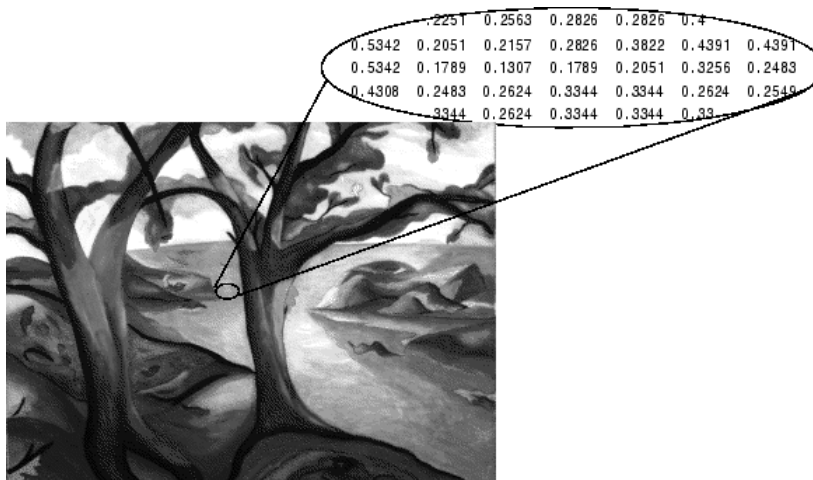
A grayscale image (also called gray-scale, gray scale, or gray-level) is a data matrix whose values represent intensities within some range. MATLAB stores a grayscale image

as an 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`.

	0.2235	0.1294	Blue	0.4198		
0.5804	0.2902	0.0627	0.2902	0.2902	0.4824	0.2235
0.5176	0.1922	0.0627	Green	0.1922	0.2588	0.2588
0.5176	0.1294	0.1608	0.1294	0.1294	0.2588	0.2588
0.5176	0.1608	0.0627	0.1608	0.1922	0.2588	0.2588
0.5490	0.2235	0.5490	Red	0.7412	0.7765	0.7765
0.5490	0.3882	0.5176	0.5804	0.5804	0.7765	0.7765
0.5490	0.2588	0.2902	0.2588	0.2235	0.4824	0.2235
0.2235	0.1608	0.2588	0.2588	0.1608	0.2588	0.2588
0.2588	0.1608	0.2588	0.2588	0.2588	0.2588	0.2588



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

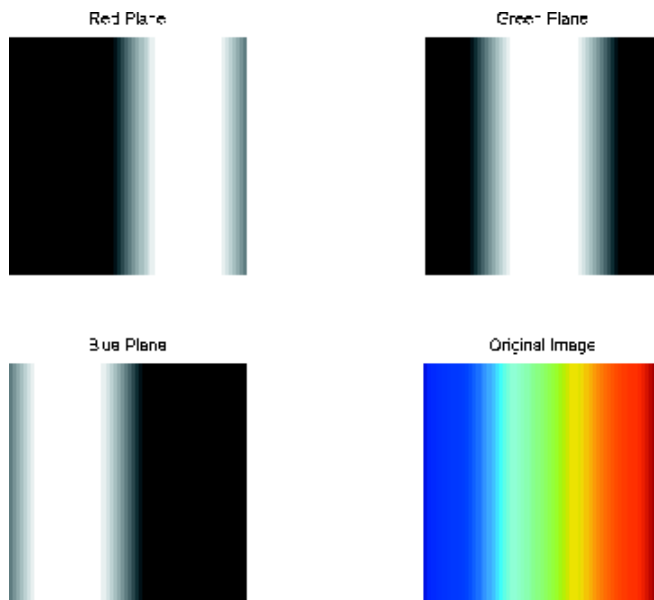
To further illustrate the concept of the three separate color planes used in a truecolor image, the code sample below creates a simple image containing uninterrupted areas

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

The toolbox includes many functions that you can use to convert an image from one type to another, listed in the following table. 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.

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 image type 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.

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 displays as shades of grays, not color.

Function	Description
<code>demosaic</code>	Convert Bayer pattern encoded image to truecolor (RGB) image.
<code>dither</code>	Use dithering to convert a grayscale image to a binary image or to convert a truecolor image to an indexed image.
<code>gray2ind</code>	Convert a grayscale image to an indexed image.
<code>grayscale</code>	Convert a grayscale image to an indexed image by using multilevel thresholding.
<code>im2bw</code>	Convert a grayscale image, indexed image, or truecolor image, to a binary image, based on a luminance threshold.

Function	Description
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	<p>Convert a truecolor image to a grayscale image.</p> <p>Note: To work with images that use other color spaces, such as HSV, first convert the image to RGB, process the image, and then convert it back to the original color space. For more information about color space conversion routines, see “Color”.</p>
rgb2ind	Convert a truecolor image to an indexed image.

Converting Between Image Classes

In this section...

“Overview of Image Class Conversions” on page 2-18

“Losing Information in Conversions” on page 2-18

“Converting Indexed Images” on page 2-18

Overview of Image Class Conversions

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 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 Using `imapprox`” on page 14-9 for more information.

Process 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, 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.

Perform an Operation on a Sequence of Images

This example shows how to perform an operation on a sequence of images. The example creates an array of images and passes the entire array to the `stdfilt` function to perform standard deviation filtering on each image in the sequence.)

Create an array of file names.

```
fileFolder = fullfile(matlabroot, 'toolbox', 'images', 'imdata');
dirOutput = dir(fullfile(fileFolder, 'AT3_im4_*.tif'));
fileNames = {dirOutput.name}
numFrames = numel(fileNames)
```

Preallocate an m -by- n -by- p array and read images into the array.

```
I = imread(fileNames{1});
sequence = zeros([size(I) numFrames], class(I));
sequence(:,:,1) = I;
%
for p = 2:numFrames
    sequence(:,:,p) = imread(fileNames{p});
end
```

Process each image in the sequence, performing standard deviation filtering, and view the results. Note that, to use `stdfilt` with an image sequence, you must specify the `nhood` argument, passing a 2-D neighborhood.

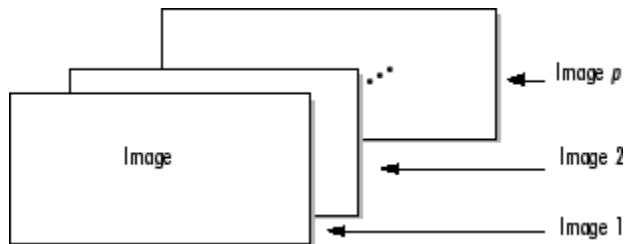
```
sequenceNew = stdfilt(sequence, ones(3));

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
```

What is an Image Sequence?

Some applications work with collections of images related by time, such as frames in a movie, or by 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*, *image stacks*, or videos.

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

The Image Viewer app and `imshow` can display one frame at a time, using standard MATLAB array indexing syntax. To animate an image sequence or provide navigation within the sequence, use the Video Viewer app (`implay`). The Video Viewer app provides playback controls that you can use to navigate among the frames in the sequence. To get a static view of all the frames in an image sequence at one time, use the `montage` function.

Toolbox Functions That Work with Image Sequences

Many toolbox functions can operate on multi-dimensional arrays and, consequently, can operate on image sequences. (For more information, see “What is an Image Sequence?” on page 2-22.) 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. For information about displaying image sequences, see “View Image Sequences in Video Viewer App”.

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

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

Function	Image Sequence Dimensions	Guideline When Used with an Image Sequence
<code>stdfilt</code>	m -by- n -by- p only	NHOOD argument must be 2-D.
<code>tformarray</code>	m -by- n -by- p or m -by- n -by-3-by- p	<ul style="list-style-type: none"> • T must be 2-D to 2-D (compatible with <code>imtransform</code>). • 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 F 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.
<code>watershed</code>	m -by- n -by- p only	Must use <code>watershed(I,conn)</code> syntax with a 2-D connectivity.

Image Arithmetic

In this section...
“Overview of Image Arithmetic Functions” on page 2-26
“Image Arithmetic Saturation Rules” on page 2-26
“Nesting Calls to Image Arithmetic Functions” on page 2-27

Overview of Image Arithmetic Functions

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 software 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 more information, see these additional topics:

Note On Intel[®] architecture processors, the image arithmetic functions can take advantage of the Intel Integrated Performance Primitives (Intel IPP) library, thus accelerating their execution time. The Intel IPP library 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 Graphics Files” on page 3-2
- “Reading Image Data” on page 3-3
- “Writing Image Data to Files” on page 3-5
- “Convert Between Graphics File Formats” on page 3-8
- “DICOM Files” on page 3-9
- “Mayo Analyze 7.5 Files” on page 3-18
- “Interfile Files” on page 3-19
- “High Dynamic Range Images” on page 3-20

Getting Information About Graphics Files

To obtain information about a graphics file and its contents, use the `imfinfo` function. 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 about an image displayed in the Image Tool — see “Get Image Information in Image Viewer App” on page 4-42.

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

Reading Image Data

To import an image from any supported graphics image file format, in any of the supported bit depths, use the `imread` function. This example reads a truecolor image into the MATLAB workspace as the variable `RGB`.

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

If the image file format uses 8-bit pixels, `imread` stores the data in the workspace as a `uint8` array. For file formats that support 16-bit data, such as PNG and TIFF, `imread` creates a `uint16` array.

`imread` uses two variables to store an indexed image in the workspace: one for the image and another for its associated colormap. `imread` always reads the colormap into a matrix of class `double`, even though the image array itself may be of class `uint8` or `uint16`.

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

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`. `imread` 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 `imread` and `imformats`.

If the graphics file contains multiple images, `imread` imports only the first image from the file. To import additional images, you must use `imread` with format-specific arguments to specify the image you want to import. In this example, `imread` imports 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 = zeros([128 128 1 27],'uint8'); % 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, you can call image processing algorithms directly. For more information, see “What is an Image Sequence?”.

If you are working with a large file, you may want to try block processing to reduce memory usage. For more information, see “Neighborhood or Block Processing: An Overview”.

Writing Image Data to Files

In this section...

“Specify Format-Specific Parameters” on page 3-5

“Read and Write 1-Bit Binary Images” on page 3-6

“Determine Storage Class of Output Files” on page 3-6

To export image data from the MATLAB workspace to a graphics file in one of the supported graphics file formats, use the `imwrite` function. When using `imwrite`, you specify the MATLAB variable name and the name of the file. If you include an extension in the filename, `imwrite` attempts to infer the desired file format from it. For example, the file extension `.jpg` infers the Joint Photographic Experts Group (JPEG) format. You can also specify the format explicitly as an argument to `imwrite`.

This example loads the indexed image `X` from a MAT-file, `clown.mat`, along with the associated colormap `map`, and then exports the image as a bitmap (BMP) file.

```
load clown
whos
```

Your output appears as shown:

Name	Size	Bytes	Class	Attributes
<code>X</code>	200x320	512000	double	
<code>caption</code>	2x1	4	char	
<code>map</code>	81x3	1944	double	

Export the image as a bitmap file:

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

Specify Format-Specific Parameters

When using `imwrite` with some graphics formats, you can specify additional format-specific parameters. For example, with PNG files, you can specify the bit depth. 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 these additional format-specific syntaxes, see the `imwrite` reference page.

Read and Write 1-Bit Binary Images

In certain file formats, such as TIFF, a binary image can be stored in a 1-bit format. When you read in a binary image in 1-bit format, `imread` stores the data in the workspace as a logical array. If the file format supports it, `imwrite` writes binary images as 1-bit images by default. This example reads in a binary image and writes it as a TIFF file.

```
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'.

Determine Storage Class of Output Files

`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	If the output image file format supports 1-bit images, <code>imwrite</code> creates a 1-bit image file. If the output image file format specified does not support 1-bit images, <code>imwrite</code> exports the image data as a <code>uint8</code> grayscale image.

Storage Class of Image	Storage Class of Output Image File
<code>uint8</code>	If the output image file format supports unsigned 8-bit images, <code>imwrite</code> creates an unsigned 8-bit image file.
<code>uint16</code>	If the output image file format supports unsigned 16-bit images (PNG or TIFF), <code>imwrite</code> creates an unsigned 16-bit image file. If the output image file format 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.
<code>int16</code>	Partially supported; depends on file format.
<code>single</code>	Partially supported; depends on file format.
<code>double</code>	MATLAB scales the image data to <code>uint8</code> and creates an 8-bit image file, because most image file formats use 8 bits.

Convert Between 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 TIFF format into the workspace and write the image data as JPEG format.

```
moon_tiff = imread('moon.tif');  
imwrite(moon_tiff, 'moon.jpg');
```

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`.

DICOM Files

In this section...

- “Overview of DICOM Support” on page 3-9
- “Read Metadata from a DICOM File” on page 3-10
- “Read Image Data from DICOM Files” on page 3-11
- “Write Data to DICOM File” on page 3-12

Overview of DICOM Support

The Digital Imaging and Communications in Medicine (DICOM) Standard is a joint project of the American College of Radiology (ACR) and the National Electrical Manufacturers Association (NEMA). The standard facilitates interoperability of medical imaging equipment by specifying a set of media storage services, a file format, and a medical directory structure to facilitate access to the images and related information stored on interchange media. For detailed information about the standard, see the official DICOM web site.

MATLAB provides the following support for working with files in the DICOM format:

- Reading any file that complies with the DICOM standard
- Writing three different types of DICOM files, or DICOM information objects (IOD), with validation:
 - Secondary capture (default)
 - Magnetic resonance
 - Computed tomography
- Writing many more types of DICOM files *without* validation, by setting the `createmode` flag to 'copy' when writing data to a file.

Note: MATLAB supports working with DICOM files. There is no support for working with DICOM network capabilities.

Read 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, and equipment settings used to capture the image. 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 “Read Image Data from DICOM Files” on page 3-11.

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: [1x89 char]
    FileModDate: '18-Dec-2000 11:06:43'
    FileSize: 525436
    Format: 'DICOM'
    FormatVersion: 3
    Width: 512
    Height: 512
    BitDepth: 16
    ColorType: 'grayscale'
    FileMetaInformationGroupLength: 192
    FileMetaInformationVersion: [2x1 uint8]
    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'
    .
    .
    .
```

Private DICOM Metadata

The DICOM specification defines many of these metadata fields, but files can contain additional fields, called private metadata. This private metadata is typically defined by equipment vendors to provide additional information about the data they provide.

When `dicominfo` encounters a private metadata field in a DICOM file, it returns the metadata creating a generic name for the field based on the group and element tags of

the metadata. For example, if the file contained private metadata at group 0009 and element 0006, `dicominfo` creates the name: `Private_0009_0006`. `dicominfo` attempts to interpret the private metadata, if it can. For example, if the metadata is a text string, `dicominfo` processes the data. If it can't interpret the data, `dicominfo` returns a sequence of bytes.

If you need to process a DICOM file created by a manufacturer that uses private metadata, and you prefer to view the correct name of the field as well as the data, you can create your own copy of the DICOM data dictionary and update it to include definitions of the private metadata. You will need information about the private metadata that vendors typically provide in DICOM compliance statements. For more information about updating DICOM dictionary, see “Create Your Own Copy of DICOM Dictionary” on page 3-11.

Create Your Own Copy of DICOM Dictionary

MathWorks uses a DICOM dictionary that contains definitions of thousands of standard DICOM metadata fields. If your DICOM file contains metadata that is not defined this dictionary, you can update the dictionary, creating your own copy that it includes these private metadata fields.

To create your own dictionary, perform this procedure:

- 1 Make a copy of the text version of the DICOM dictionary that is included with MATLAB. This file, called `dicom-dict.txt` is located in `matlabroot/toolbox/images/medformats` or `matlabroot/toolbox/images/iptformats` depending on which version of the Image Processing Toolbox software you are working with. Do not attempt to edit the MAT-file version of the dictionary, `dicom-dict.mat`.
- 2 Edit your copy of the DICOM dictionary, adding entries for the metadata. Insert the new metadata field using the group and element tag, type, and other information. Follow the format of the other entries in the file. The creator of the metadata (e.g., an equipment vendor) must provide you with the information.
- 3 Save your copy of the dictionary.
- 4 Set MATLAB to use your copy of the DICOM dictionary, `dicomdict` function.

Read Image Data from DICOM Files

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-MONO2-16-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);
```

View DICOM Images

To view the image data imported from a DICOM file, use one of the toolbox image display functions `imshow` or `imshow`. 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', [])
```



Write Data to 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, 'ankle.dcm')
```

Include Metadata with 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, '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 and compare it 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.

```
info2 = dicominfo('ankle.dcm');

info2.SOPInstanceUID
ans =
```

1.2.841.113411.2.1.2411.10311244477.365.1.63874544

Note that the instance ID in the newly created file differs from the ID in the original file.

Explicit Versus Implicit VR Attributes

DICOM attributes provide the length and then the data. When writing data to a file, you can include a two-letter value representation (VR) with the attribute or you can let DICOM infer the value representation from the data dictionary. When you specify the VR option with the value 'explicit', the `dicomwrite` function includes the VR in the attribute. The following figure shows the attributes with and without the VR.

Remove 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 “Create a New DICOM Series” on page 3-14 with a call to the `dicomanon` function.

Create a New DICOM 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 `imshow`. Because the DICOM image data is signed 16-bit data, you must use the autoscaling syntax.

```
imshow(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`.

Mayo Analyze 7.5 Files

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 “Interfile Files” on page 3-19.

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.

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

Interfile Files

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 “Mayo Analyze 7.5 Files” on page 3-18.

The following example calls the `interfileinfo` function to read the metadata from an 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 the Interfile Archive maintained by the Department of Medical Physics and Bioengineering, University College, London, UK.

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

High Dynamic Range Images

In this section...

“Dynamic Range” on page 3-20

“Read High Dynamic Range Image” on page 3-20

“Create High Dynamic Range Image” on page 3-21

“Display High Dynamic Range Image” on page 3-21

“Write High Dynamic Range Image to File” on page 3-23

Dynamic Range

Dynamic range refers to the range of brightness levels, from dark to light. The dynamic range of real-world scenes can be quite high. High Dynamic Range (HDR) images attempt to capture the whole tonal range of real-world scenes (called *scene-referred*), using 32-bit floating-point values to store each color channel. HDR images contain a high level of detail, close to the range of human vision. The toolbox includes functions for reading, creating, and writing HDR images, and a tone-map operator for displaying HDR images on a computer monitor.

Read High Dynamic Range Image

To read a high dynamic range image into the MATLAB workspace, use the `hdrread` function.

```
hdr_image = hdrread('office.hdr');
```

The output image `hdr_image` is an m-by-n-by-3 image of type `single`.

```
whos
  Name           Size           Bytes    Class    Attributes
  ----           -
  hdr_image     665x1000x3     7980000  single
```

Note, however, that before you can display a high dynamic range image, you must convert it to a dynamic range appropriate to a computer display, a process called *tone mapping*. Tone mapping algorithms scale the dynamic range down while attempting to preserve the appearance of the original image. For more information, see “Display High Dynamic Range Image” on page 3-21.

Create High Dynamic Range Image

To create a high dynamic range image from a group of low dynamic range images, use the `makehdr` function. Note that the low dynamic range images must be spatially registered and the image files must contain EXIF metadata. Specify the low-dynamic range images in a cell array.

```
hdr_image = makehdr(files);
```

Display High Dynamic Range Image

This example shows how to display a high dynamic range (HDR) image. To view an HDR image, you must first convert the data to a dynamic range that can be displayed correctly on a computer.

Read a high dynamic range (HDR) image, using `hdrread`. If you try to display the HDR image, notice that it does not display correctly.

```
hdr_image = hdrread('office.hdr');  
imshow(hdr_image)
```



Convert the HDR image to a dynamic range that can be viewed on a computer, using the `tonemap` function. This function converts the HDR image into an RGB image of class `uint8`.

```
rgb = tonemap(hdr_image);  
whos
```

Name	Size	Bytes	Class	Attributes
hdr_image	665x1000x3	7980000	single	
rgb	665x1000x3	1995000	uint8	

Display the RGB image.

```
imshow(rgb)
```




Write High Dynamic Range Image to File

To write a high dynamic range image from the MATLAB workspace into a file, use the `hdrwrite` function.

```
hdrwrite(hdr, 'filename');
```


Displaying and Exploring Images

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

- “Image Display and Exploration Overview” on page 4-2
- “Displaying Images Using the imshow Function” on page 4-4
- “Using the Image Viewer App to Explore Images” on page 4-13
- “Explore Very Large Images” on page 4-22
- “Explore Images with the Image Viewer App” on page 4-23
- “Get Pixel Information in Image Viewer App” on page 4-30
- “Measure Distance Between Pixels in Image Viewer” on page 4-38
- “Get Image Information in Image Viewer App” on page 4-42
- “Adjust Image Contrast In Image Viewer App” on page 4-44
- “Crop Image Using Image Viewer App” on page 4-54
- “View Image Sequences in Video Viewer App” on page 4-59
- “View Image Sequence as Montage” on page 4-68
- “Convert Multiframe Image to Movie” on page 4-70
- “Display Different Image Types” on page 4-71
- “Add Colorbar to Displayed Image” on page 4-77
- “Print Images” on page 4-79
- “Image Processing Toolbox Preferences” on page 4-80

Image Display and Exploration Overview

The Image Processing Toolbox software 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.

`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 “Displaying Images Using the `imshow` Function” on page 4-4. The `imshow` function is also a key building block for image applications you can create using the toolbox modular tools. For more information, see “Build Interactive Tools”

The other toolbox display function, `imtool`, launches the Image Viewer app, which presents an integrated environment for displaying images and performing some common image-processing tasks. The Image Viewer 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 Viewer App to Explore Images” on page 4-13.

In general, using the toolbox functions to display images is preferable to using MATLAB image display functions `image` and `imagesc` because the toolbox functions set certain Handle Graphics properties automatically to optimize the image display. The following table lists these properties and their settings for each image type. In the table, 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 X	Set to the data in I	Set to data in BW	Set to data in RGB

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

Displaying Images Using the `imshow` Function

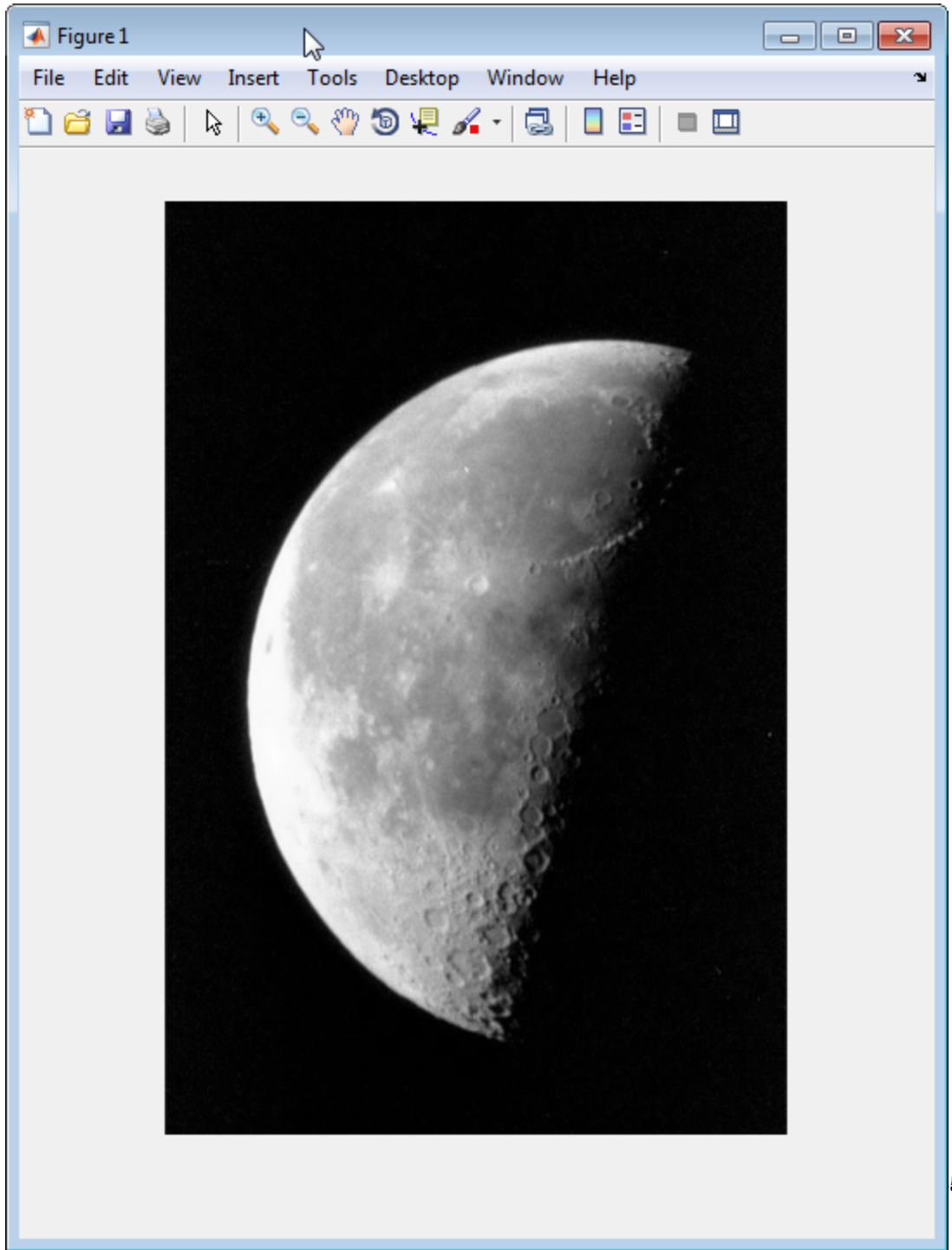
In this section...
“Overview” on page 4-4
“Specifying the Initial Image Magnification” on page 4-6
“Controlling the Appearance of the Figure” on page 4-7
“Displaying Each Image in a Separate Figure” on page 4-9
“Displaying Multiple Images in the Same Figure” on page 4-9

Overview

To display image data, use the `imshow` function. The following example reads an image into the MATLAB workspace and then displays the image 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.



You can also pass `imshow` the name of a file containing an image.

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

This syntax can be useful for scanning through images. Note, however, that when you use this syntax, `imread` does not store the image data 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. This example assigns the image data from `moon.tif` to the variable `moon`, if the figure window in which it is displayed is currently active.

```
moon = getimage;
```

For more information about using `imshow` to display the various image types supported by the toolbox, see “Display Different Image Types” on page 4-71.

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.

To change the default initial magnification behavior of `imshow`, set the `ImshowInitialMagnification` toolbox preference. To set the preference, open the Image Processing Toolbox Preferences dialog by calling `iptprefs` or by selecting **Preferences** from the MATLAB Desktop **File** menu.

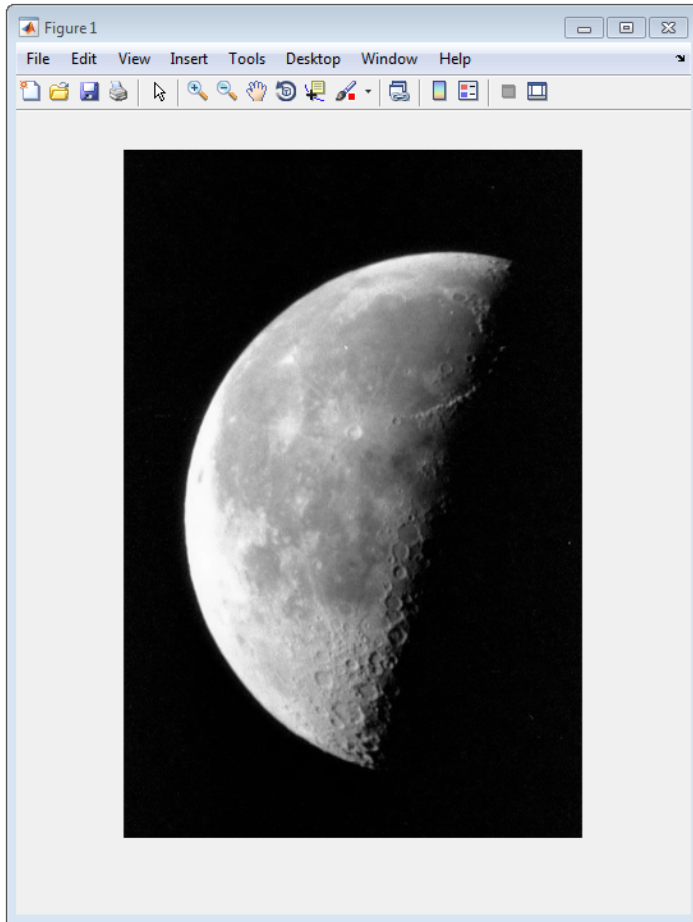
When `imshow` scales an image, it uses interpolation to determine the values for screen pixels that do not directly correspond to elements in the image matrix. For more information, see “Specify the Interpolation Method” on page 6-3.

Controlling the Appearance of the Figure

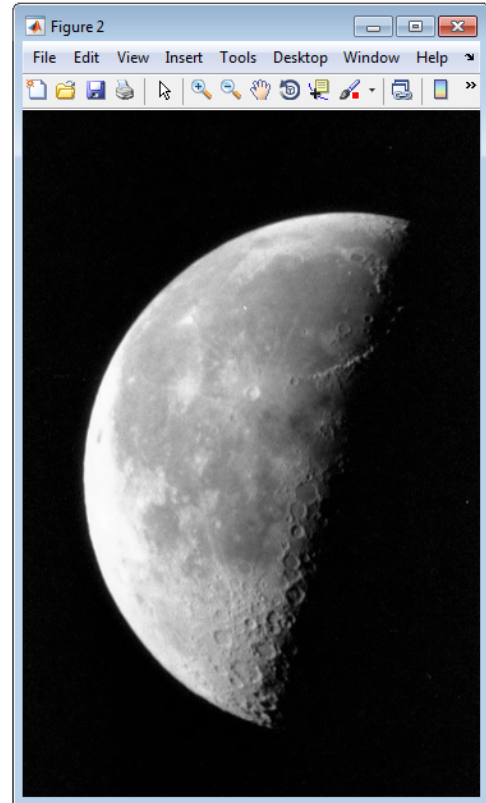
By default, when `imshow` displays an image in a figure, it surrounds the image with a gray border. You can change this default and suppress the border using the `'border'` parameter, as shown in the following example.

```
imshow('moon.tif','Border','tight')
```

The following figure shows the same image displayed with and without a border.



'Loose'



'Tight'

The `'border'` parameters affect only the image being displayed in the call to `imshow`. If you want all the images that you display using `imshow` to appear without the gray border, set the Image Processing Toolbox `'imshowborder'` preference to `'tight'`. You can also use preferences to include visible axes in the figure. For more information about preferences, see `iptprefs`.

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.

`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))
```

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. For additional options, see “What is an Image Sequence?”.

Note The Image Viewer app (`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)
```

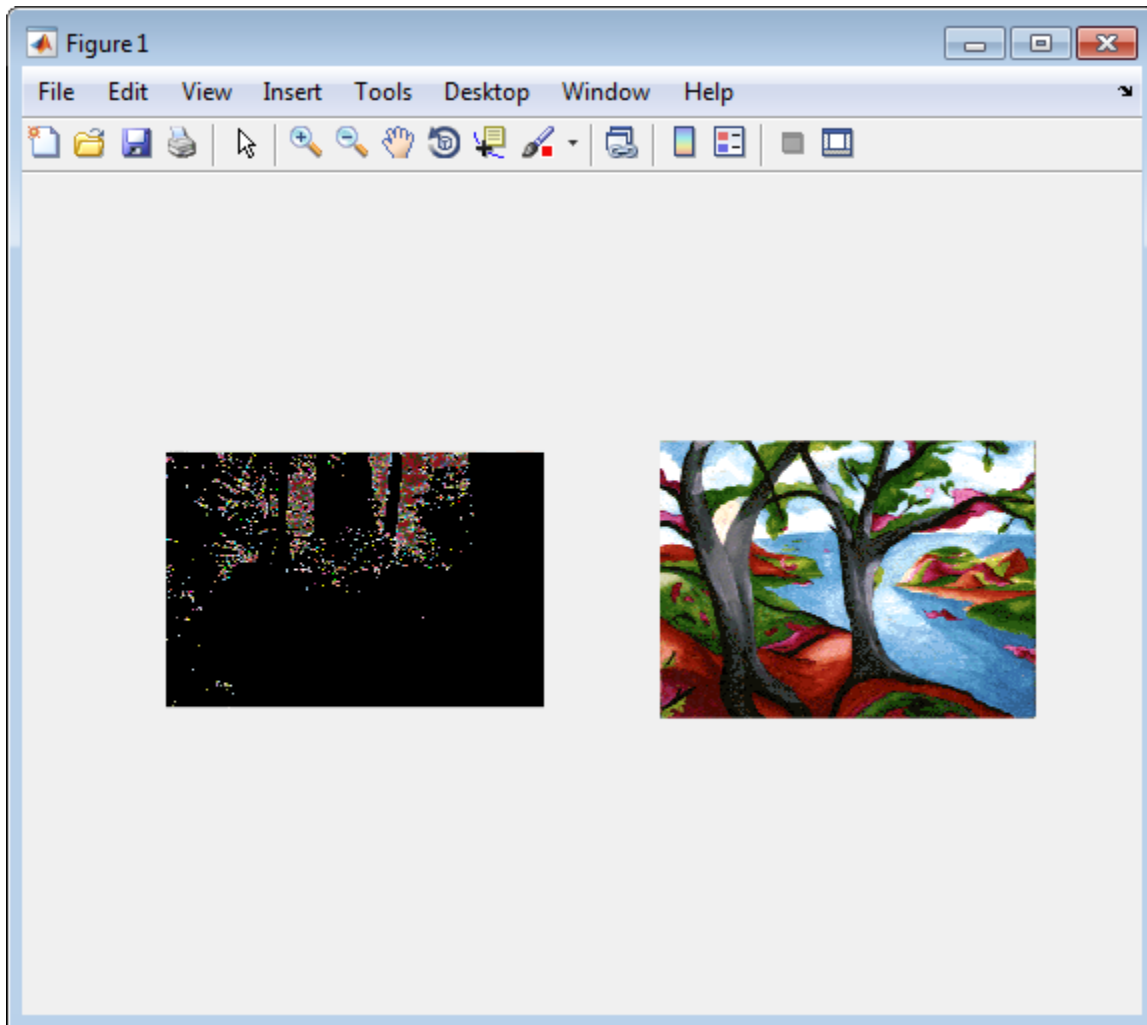
This syntax divides the figure into an `m`-by-`n` matrix of display regions and makes the `pth` display region active.

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 4-11, 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.

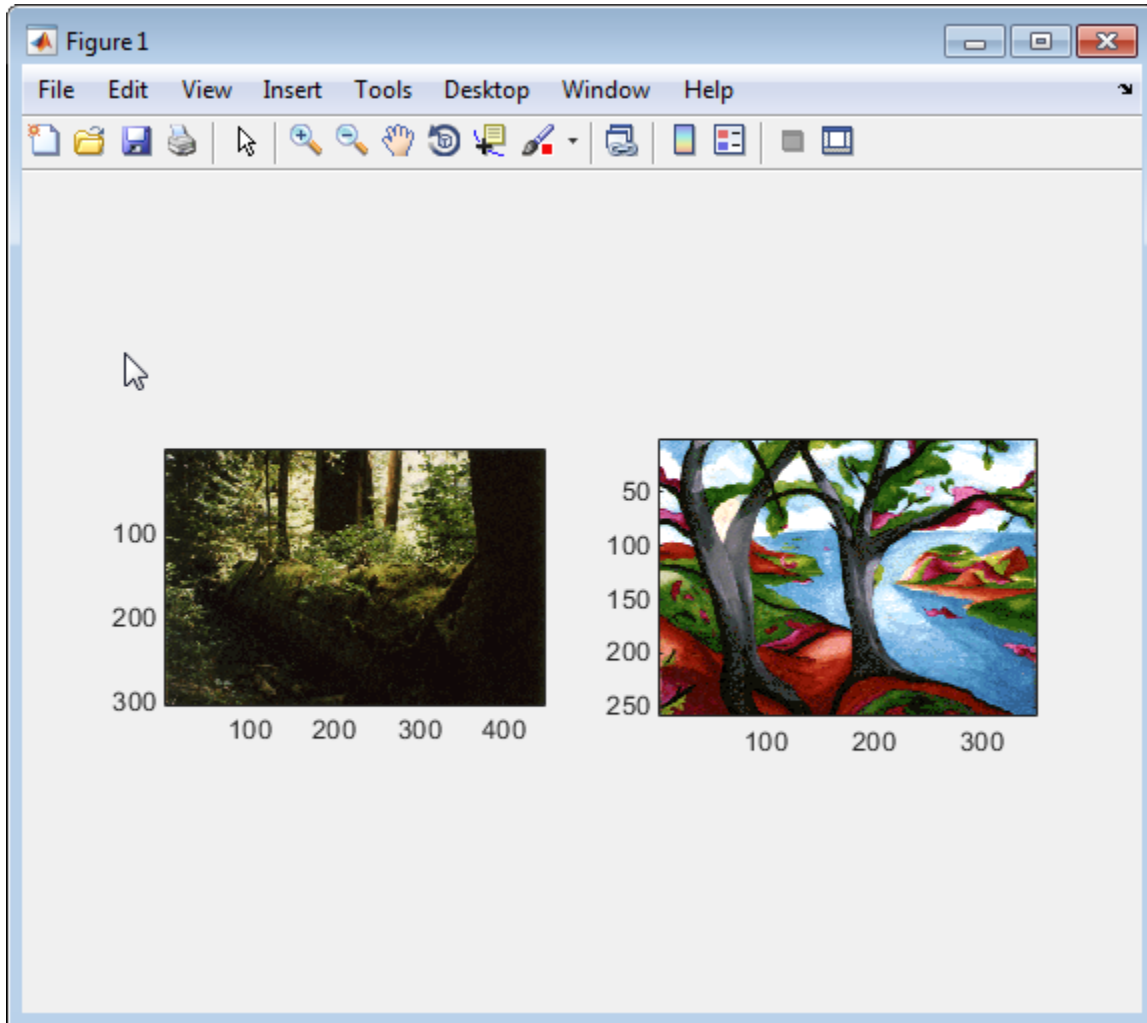


Using the `subimage` Function to Display Multiple Images

`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)
```



Using the Image Viewer App to Explore Images

In this section...

- “Image Viewer App Overview” on page 4-13
- “Opening the Image Viewer App” on page 4-15
- “Specifying the Initial Image Magnification” on page 4-15
- “Specifying the Colormap” on page 4-16
- “Importing Image Data from the Workspace” on page 4-17
- “Exporting Image Data to the Workspace” on page 4-18
- “Saving Image Data Displayed in Image Viewer” on page 4-19
- “Closing the Image Viewer App” on page 4-20
- “Printing Images Displayed in Image Viewer App” on page 4-21

Image Viewer App Overview

The Image Viewer app is an image display and exploration tool that presents an integrated environment for displaying images and performing common image-processing tasks. The Image Viewer provides access to several other tools:

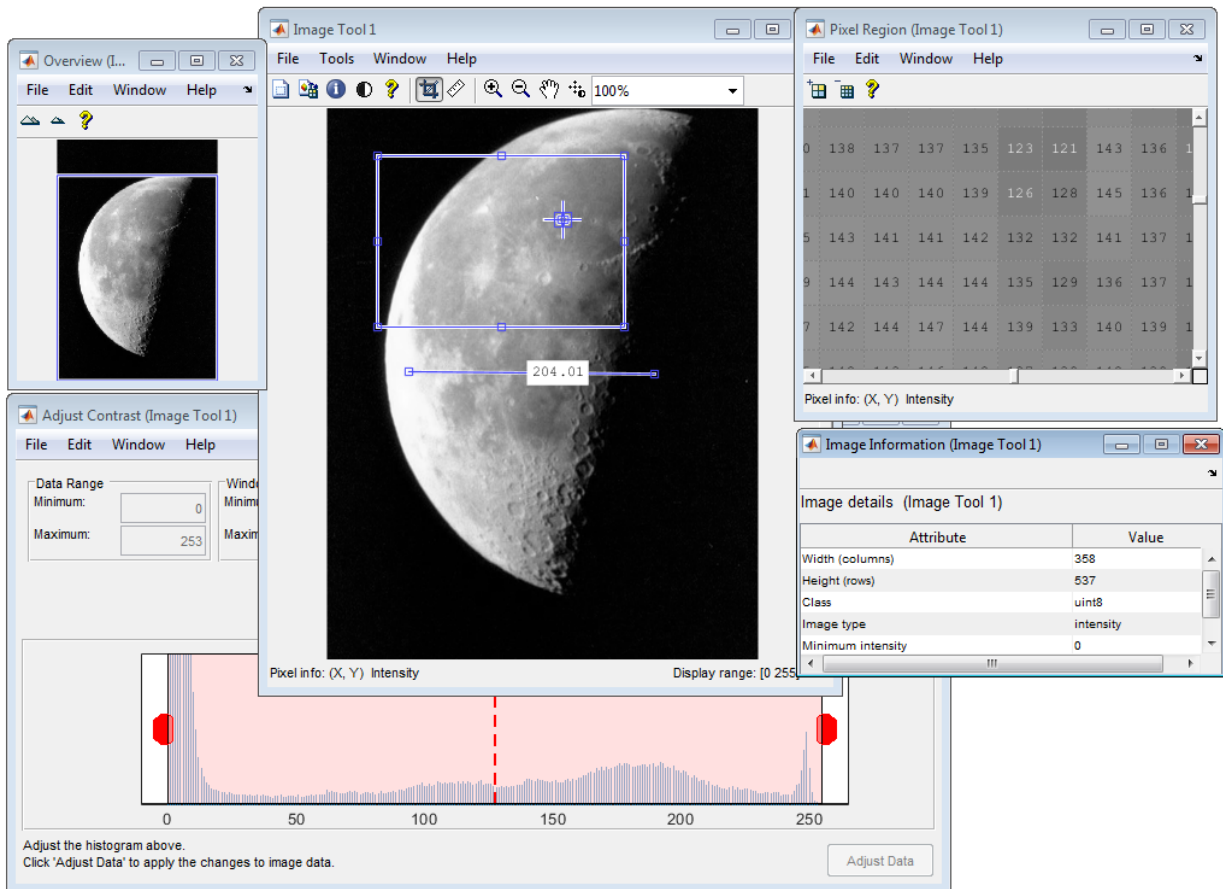
- Pixel Information tool — for getting information about the pixel under the pointer
- Pixel Region tool — for getting information about a group of pixels
- Distance tool — for measuring the distance between two pixels
- Image Information tool — for getting information about image and image file metadata
- Adjust Contrast tool and associated Window/Level tool — for adjusting the contrast of the image displayed in the Image Viewer and modifying the actual image data. You can save the adjusted data to the workspace or a file.
- Crop Image tool — for defining a crop region on the image and cropping the image. You can save the cropped image to the workspace or a file.
- Display Range tool — for determining the display range of the image data

In addition, the Image Viewer provides several navigation aids that can help explore large images:

4 Displaying and Exploring Images

- Overview tool — for determining what part of the image is currently visible in the Image Viewer and changing this view.
- Pan tool — for moving the image to view other parts of the image
- Zoom tool — for getting a closer view of any part of the image.
- Scroll bars — for navigating over the image.

The following figure shows the image displayed in the Image Viewer app with many of the related tools open and active.



Opening the Image Viewer App

To start the Image Viewer, click **Image Viewer** on the **Apps** tab, or use the `imtool` function. You can also start another Image Viewer from within an existing Image Viewer by using the **New** option from the **File** menu.

To bring image data into the Image Viewer, you can use either the **Open** or **Import from Workspace** options from the **File** menu — see “Importing Image Data from the Workspace” on page 4-17.

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 specify a file name, the image data is not stored in a MATLAB workspace variable. To bring the image displayed in the Image Viewer into the workspace, you must use the `getimage` function or the **Export to Workspace** option from the Image Viewer **File** menu — see “Exporting Image Data to the Workspace” on page 4-18.

Specifying the Initial Image Magnification

The Image Viewer attempts to display an image in its entirety at 100% magnification (one screen pixel for each image pixel) and always honors any magnification value you specify. If the image is too big to fit in a figure on the screen, the Image Viewer shows only a portion of the image, 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, the Image Viewer scales the image to fit, without issuing a warning. This is the default behavior, specified by the `'InitialMagnification'` parameter value `'adaptive'`.

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

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

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

Another way to change the default initial magnification behavior of the Image Viewer is to set the `ImtoolInitialMagnification` toolbox preference. The magnification value you specify remains in effect until you change it. To set the preference, use `iptsetpref` or open the Image Processing Preferences panel by calling `iptprefs` or by selecting **File** > **Preferences** in the Image Viewer menu. To learn more about toolbox preferences, see `iptprefs`.

When the Image Viewer 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 “Specify the Interpolation Method” on page 6-3.

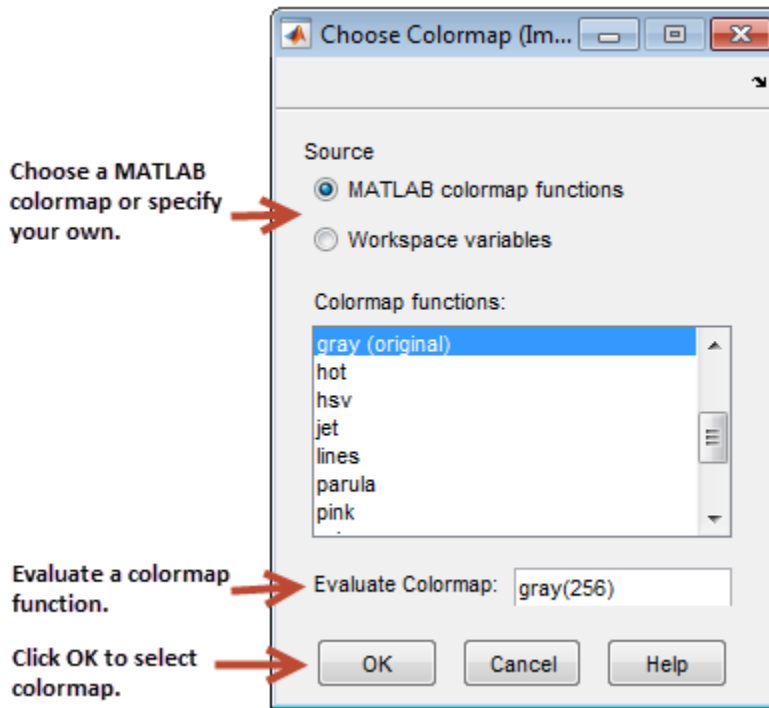
Specifying the Colormap

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 Viewer, select the **Choose Colormap** option on the **Tools** menu. This activates the Choose Colormap tool. 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 Viewer 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.

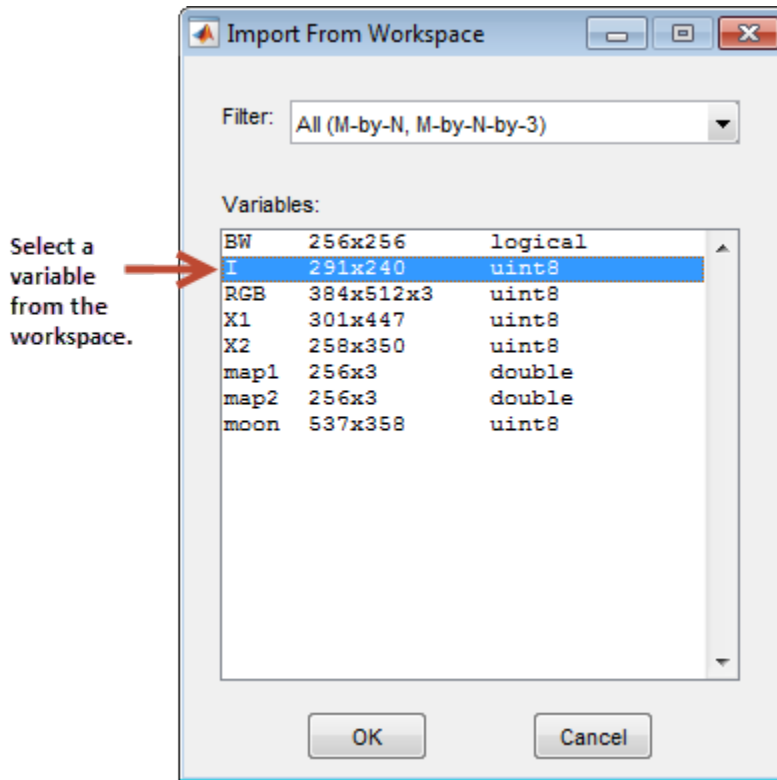
When you choose a colormap, the image updates to use the new map. If you click **OK**, the Image Viewer applies the colormap and closes the Choose Colormap tool. If you click **Cancel**, the image reverts to the previous colormap.



Importing Image Data from the Workspace

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

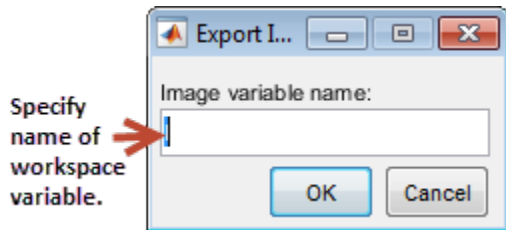
The following figure shows the Import from Workspace dialog box. You can use the Filter menu to limit the images included in the list to certain image types, i.e., binary, indexed, intensity (grayscale), or truecolor.



Exporting Image Data to the Workspace

To export the image displayed in the Image Viewer to the MATLAB workspace, you can use the **Export to Workspace** option on the Image Viewer **File** menu. (Note that when exporting data, changes to the display range will not be preserved.) In the dialog box, shown below, you specify the name you want to assign to the variable in the workspace. By default, the Image Viewer prefills the variable name field with **BW**, for binary images, **RGB**, for truecolor images, and **I** for grayscale or indexed images.

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



Using the `getimage` Function to Export Image Data

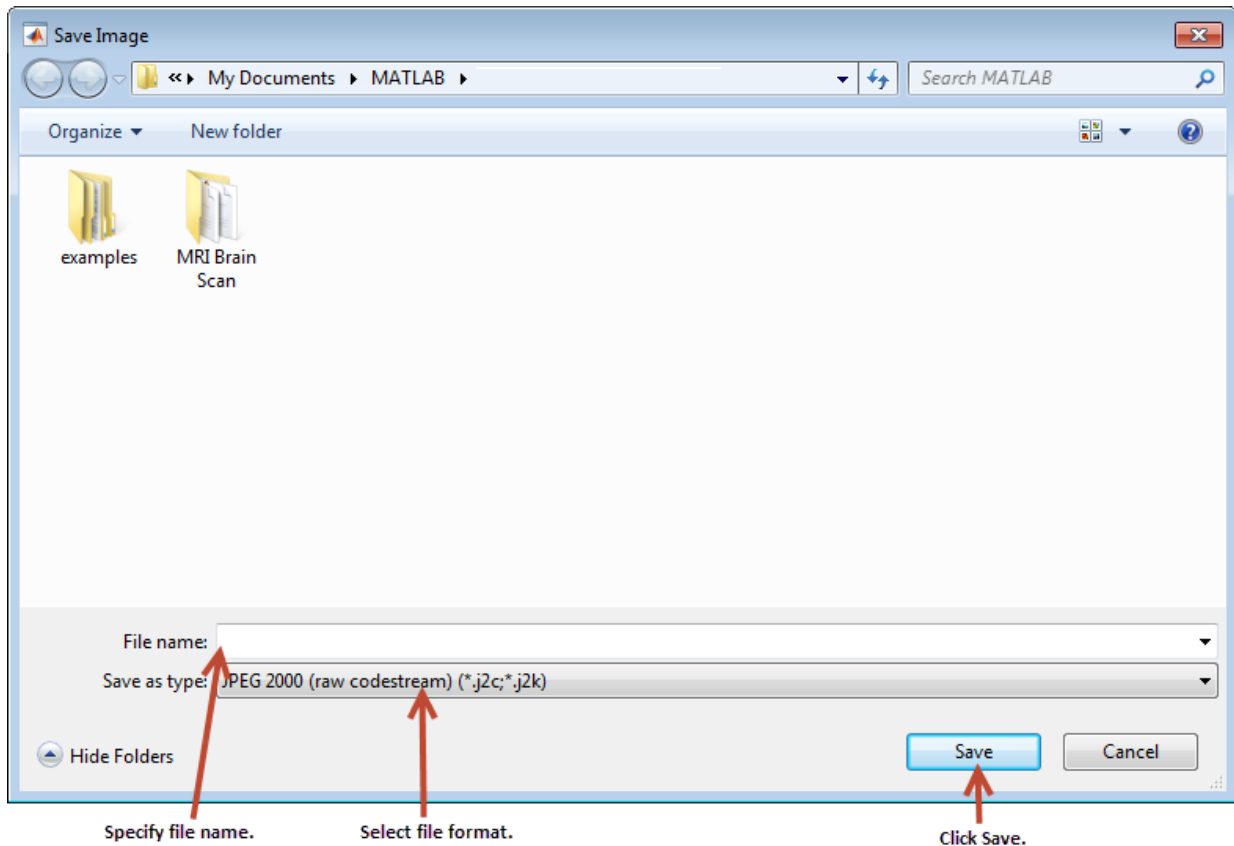
You can also use the `getimage` function to bring image data from the Image Viewer into the MATLAB workspace.

The `getimage` function retrieves the image data (`CData`) from the current Handle Graphics image object. Because, by default, the Image Viewer 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 Viewer. The following example assigns the image data from `moon.tif` to the variable `moon` if the figure window in which it is displayed is currently active.

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

Saving Image Data Displayed in Image Viewer

To save the image data displayed in the Image Viewer, select the **Save as** option from the Image Viewer **File** menu. The Image Viewer opens the Save Image dialog box, shown in the following figure. Use this dialog box to navigate your file system to determine where to save the image file and specify the name of the file. Choose the graphics file format you want to use from among many common image file formats listed in the Files of Type menu. If you do not specify a file name extension, the Image Viewer adds an extension to the file associated with the file format selected, such as `.jpg` for the JPEG format.



Note: Changes you make to the display range will not be saved. If you would like to preserve your changes, use `imcontrast`.

Closing the Image Viewer App

To close the Image Viewer, use the **Close** button in the window title bar or select the **Close** option from the Image Viewer **File** menu. If you used the `imtool` function to start the Image Viewer you can get a handle to the app. You can use this handle to close the app. When you close the Image Viewer, any related tools that are currently open also close.

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

```
imtool close all
```

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

Printing Images Displayed in Image Viewer App

To print the image displayed in the Image Viewer, select the **Print to Figure** option from the **File** menu. The Image Viewer 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 “Print Images” on page 4-79 for more information.

Explore Very Large Images

In this section...
“Overview” on page 4-22
“Creating an R-Set File” on page 4-22
“Opening an R-Set File” on page 4-22

Overview

If you are viewing a very large image, it might not load in Image Viewer, or it could load, but zooming and panning are slow. In either case, creating a reduced resolution data set (R-Set) can improve performance. Use the Image Viewer to navigate an R-Set image the same way you navigate a standard image.

Creating an R-Set File

To create an R-Set file, use the function `rsetwrite`. For example, to create an R-Set from a TIFF file called `'LargeImage.tif'`, enter the following:

```
rsetfile = rsetwrite('LargeImage.tif')
```

`rsetwrite` saves an R-Set file named `'LargeImage.rset'` in the current directory. Or, if you want to save the R-Set file under a different name, enter the following:

```
rsetfile = rsetwrite('LargeImage.tif', 'New_Name')
```

You can create an R-Set file directly from a TIFF or NITF file, or you can create one from another type of image file by first creating an Image Adapter object. See “Working with Data in Unsupported Formats” for more information.

The time required to create an R-Set varies, depending on the size of the initial file and the capability of your machine. A progress bar shows an estimate of time required. If you cancel the operation, processing stops, no file is written, and the `rsetfile` variable will be empty.

Opening an R-Set File

Open an R-Set file from the command line:

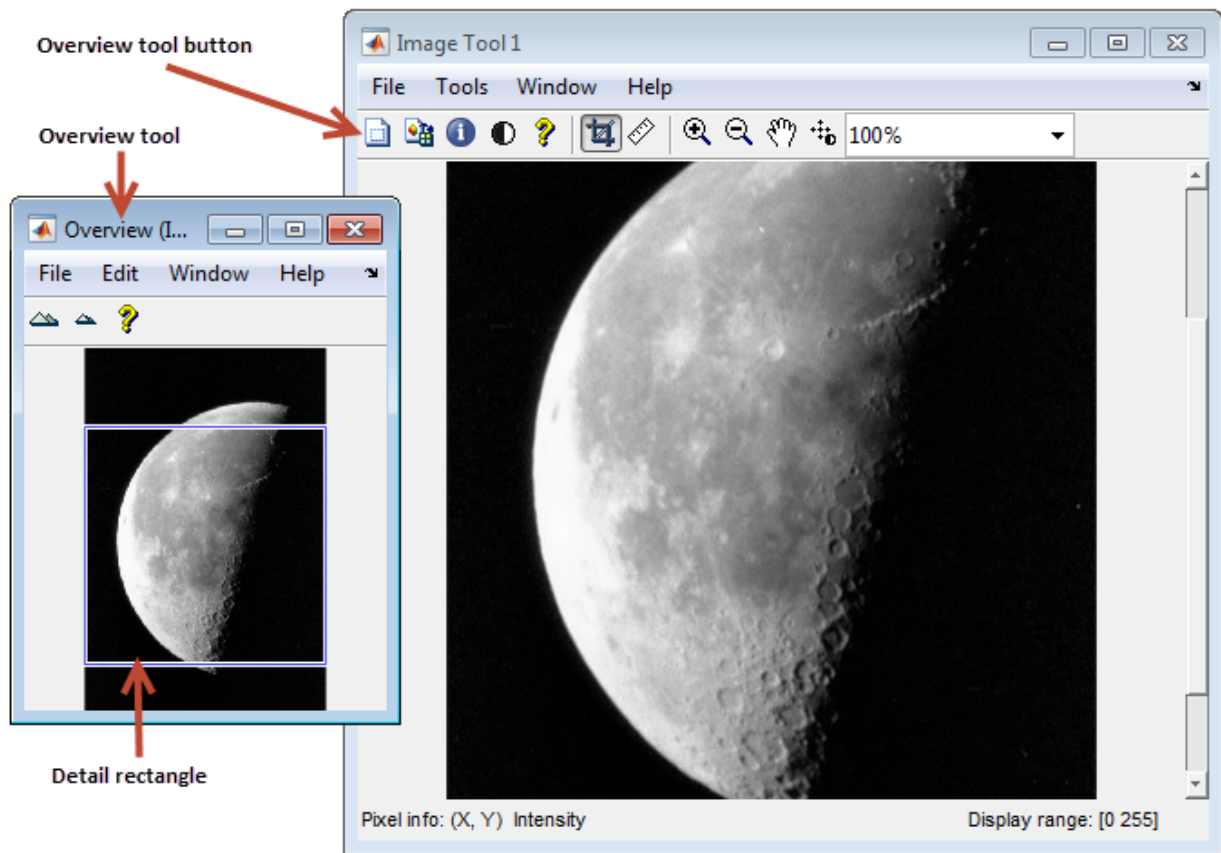
```
imtool('LargeImage.rset')
```


Explore Images with the Image Viewer App

In this section...
“Explore Images Using the Overview Tool” on page 4-23
“Pan Images Displayed in Image Viewer App” on page 4-26
“Zoom Images in the Image Viewer App” on page 4-26
“Specify Image Magnification in Image Viewer” on page 4-27

Explore Images Using the Overview Tool


If an image is large or viewed at a large magnification, the Image Viewer displays only a portion of the entire image and automatically includes scroll bars to allow navigation around the image. To determine which part of the image is currently visible in the Image Viewer, use the Overview tool. The Overview tool displays the entire image, scaled to fit. 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 Viewer. You can change the portion of the image visible in the Image Viewer by moving the detail rectangle over the image in the Overview tool.




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


- “Starting the Overview Tool” on page 4-25
- “Moving the Detail Rectangle to Change the Image View” on page 4-25
- “Specifying the Color of the Detail Rectangle” on page 4-25
- “Getting the Position and Size of the Detail Rectangle” on page 4-25
- “Printing the View of the Image in the Overview Tool” on page 4-26

Starting the Overview Tool

You can start the Overview tool by clicking the **Overview** button  in the Image Viewer toolbar or by selecting the **Overview** option from the Image Viewer **Tools** menu. You can also change the preferences, so the Overview tool will open automatically when you open the Image Viewer. For more information on setting preferences, see `iptprefs`.

Moving the Detail Rectangle to Change the Image View

- 1 Start the Overview tool by clicking the **Overview** button  in the Image Viewer 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.

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 pointer into the detail rectangle. The pointer changes to a fleur, .
- 3 Press and hold the mouse button to drag the detail rectangle anywhere on the image. The Image Viewer 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 can 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 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]`. You can paste this position vector into the MATLAB workspace or another application.

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 “Print Images” on page 4-79 for more information.

Pan Images Displayed in Image Viewer App

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

- 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 pointer over the image in the Image Viewer, using the mouse. The pointer changes to an open-hand shape .
- 3 Press and hold the mouse button and drag the image in the Image Viewer. When you drag the image, the pointer 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 Viewer, the Overview tool updates the position of the detail rectangle — see “Explore Images Using the Overview Tool” on page 4-23.

Zoom Images in the Image Viewer App

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 “Specify Image Magnification in Image Viewer” on page 4-27 or by using the **Ctrl+Plus** or **Ctrl+Minus** keys. Note that these are the **Plus(+)** and **Minus(-)** keys on the numeric keypad of your keyboard.)

- 1 Click the appropriate magnifying glass button in the Image Viewer 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.



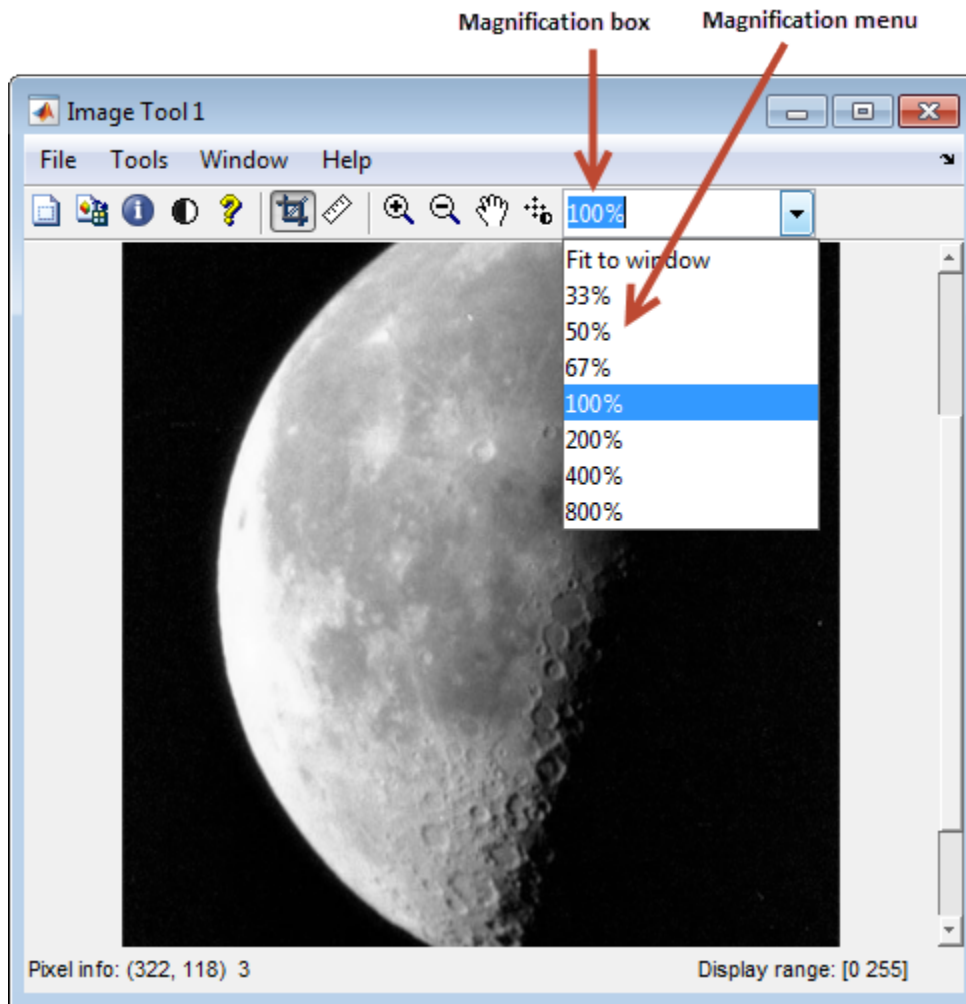
- 2 Move the pointer over the image you want to zoom in or out on, using the mouse. The pointer changes to the appropriate magnifying glass icon. With each click, the Image Viewer changes the magnification of the image, centering the new view of the image on the spot where you clicked.

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.

Specify Image Magnification in Image Viewer

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 “Zoom Images in the Image Viewer App” on page 4-26 for more information.)



To change the magnification of an image,

- 1 Move the pointer into the magnification edit box. The pointer changes to the text entry cursor.
- 2 Type a new value in the magnification edit box and press **Enter**. The Image Viewer 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 Viewer scales the image so that the entire image is visible.

Get Pixel Information in Image Viewer App

In this section...
“Determine Individual Pixel Values in Image Viewer” on page 4-30
“Determine Pixel Values in an Image Region” on page 4-31
“Determine Image Display Range in Image Viewer” on page 4-35

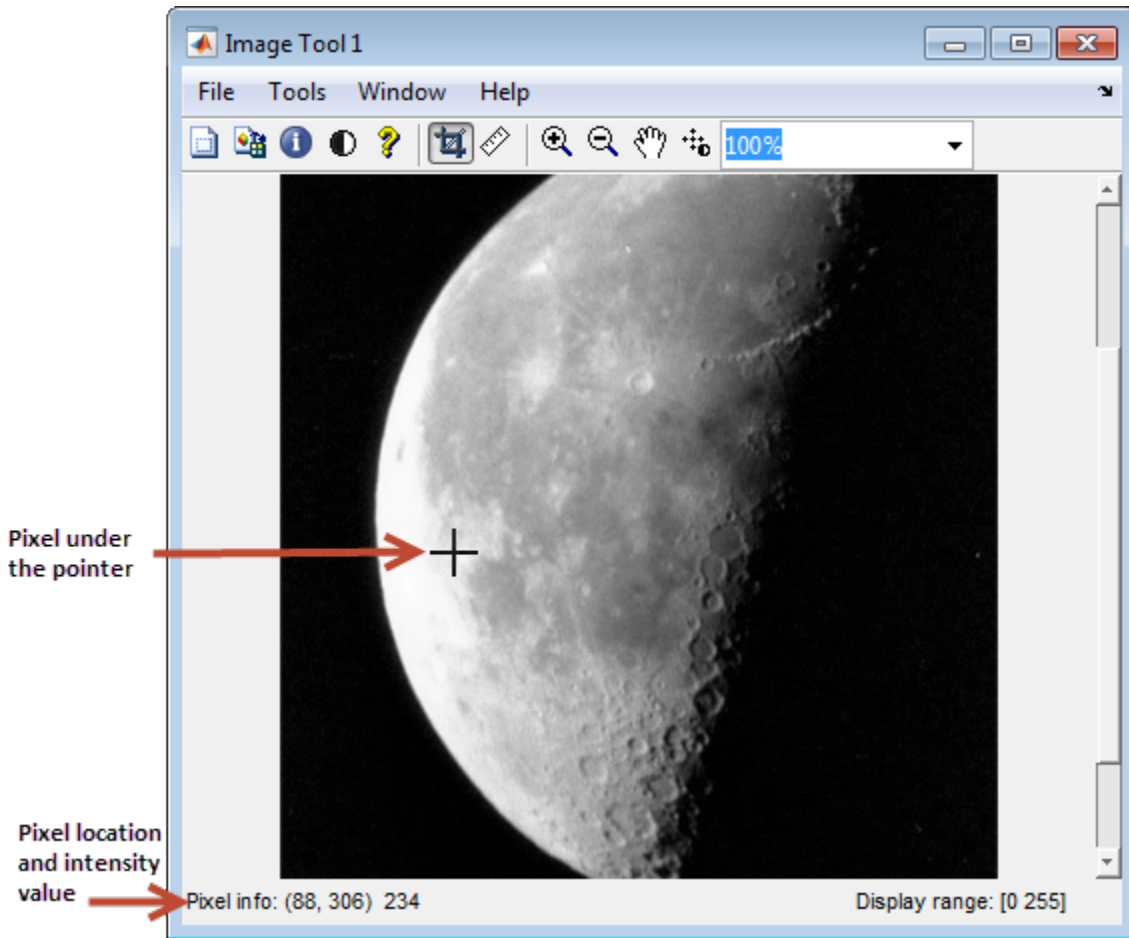
Determine Individual Pixel Values in Image Viewer

The Image Viewer displays information about the location and value of individual pixels in an image in the bottom left corner of the tool. (You can also obtain this information by opening a figure with `imshow` and then calling `impixelinfo` from the command line.) The pixel value and location information represent the pixel under the current location of the pointer. The Image Viewer updates this information as you move the pointer over the image.

For example, view an image in the Image Viewer.

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

The following figure shows the Image Viewer with pixel location and value displayed in the Pixel Information tool. For more information, see “Saving the Pixel Value and Location Information” on page 4-31.



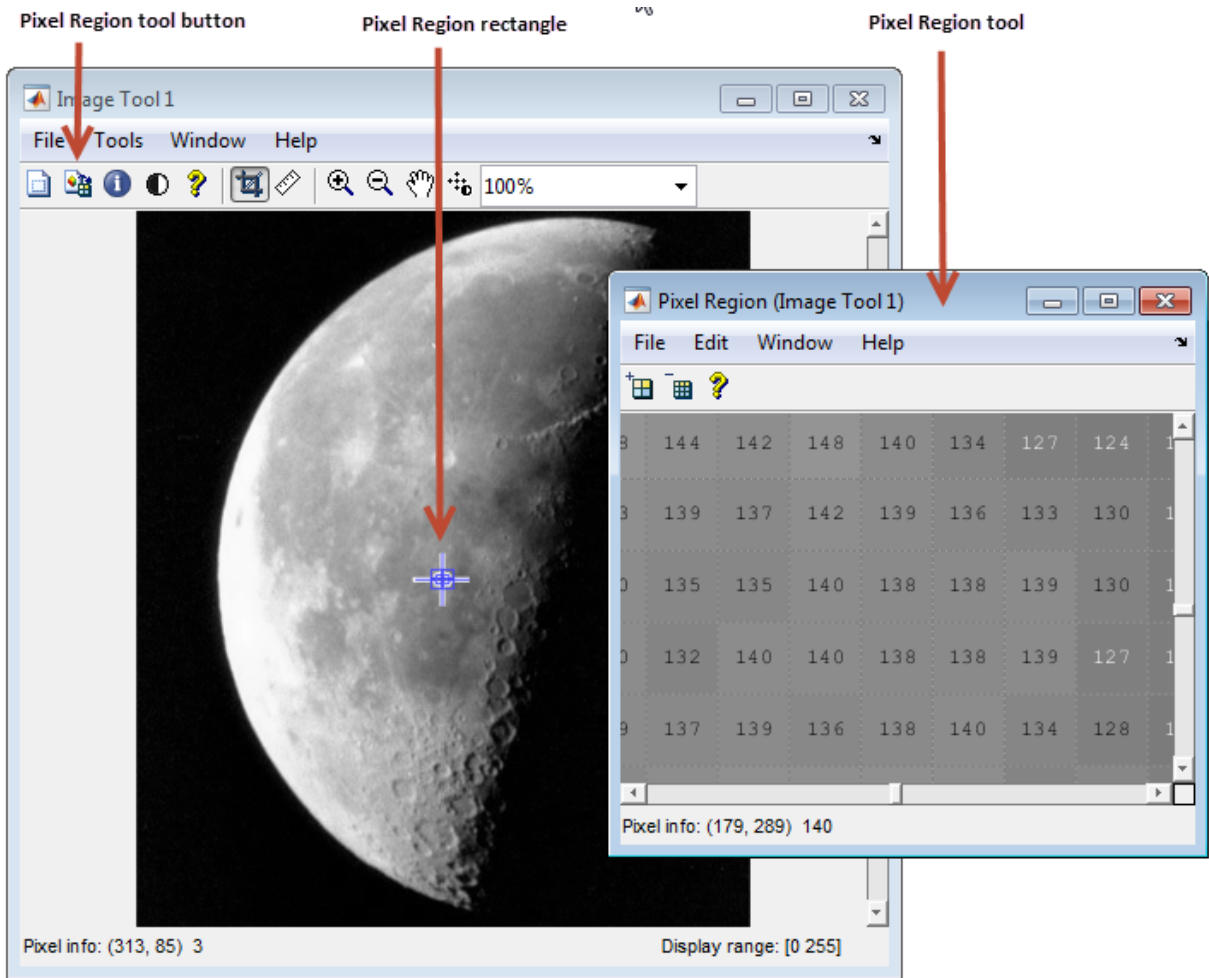
Saving the Pixel Value and Location Information

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 Viewer copies the x - and y -coordinates and the pixel value to the clipboard.

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

Determine Pixel Values in an Image Region



To view the values of pixels in a specific region of an image displayed in the Image Viewer, use the Pixel Region tool. The Pixel Region tool superimposes a rectangle, called the *pixel region rectangle*, over the image displayed in the Image Viewer. 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 Viewer with the Pixel Region tool. Note how the Pixel Region tool includes the value of each pixel in the display.




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

- “Selecting a Region” on page 4-33
- “Customizing the View” on page 4-33
- “Determining the Location of the Pixel Region Rectangle” on page 4-34
- “Printing the View of the Image in the Pixel Region Tool” on page 4-35

Selecting a Region

- 1 To start the Pixel Region tool, click the **Pixel Region** button  in the Image Viewer toolbar or select the **Pixel Region** option from the **Tools** menu. (Another option is to open a figure using `imshow` and then call `impixelregion` from the command line.) The Image Viewer 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 4-34.

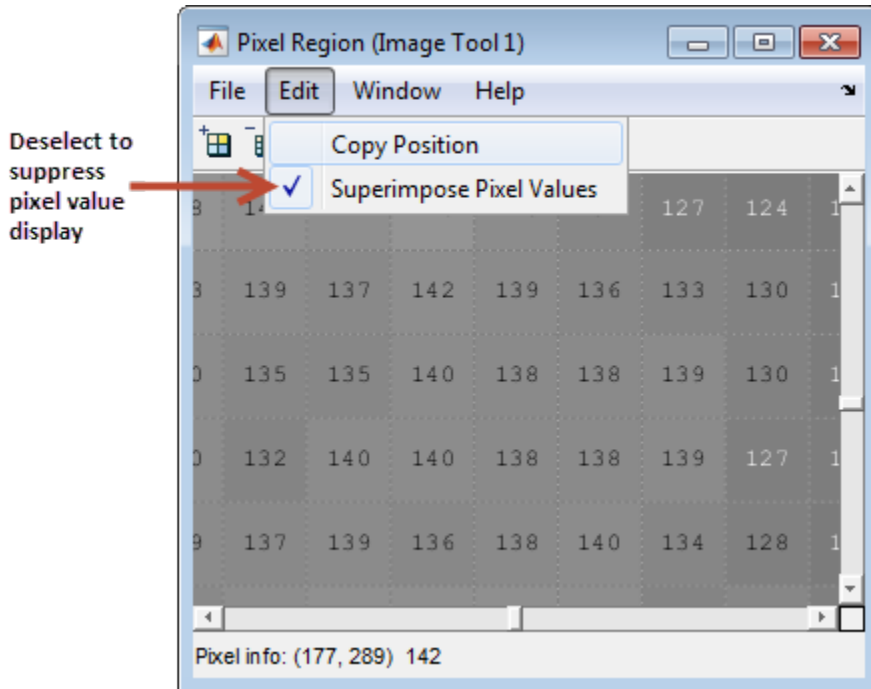
- 2 Using the mouse, position the pointer over the pixel region rectangle. The pointer changes to the fleur shape, .
- 3 Click the left mouse button and drag the pixel region rectangle to any part of the image. As you move the pixel region rectangle over the image, the Pixel Region tool updates the pixel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

Customizing the View

To get a closer view of image pixels, use the zoom buttons on the Pixel Region tool toolbar. As you zoom in, the size of the pixels displayed in the Pixel Region tool increase and fewer pixels are visible. As you zoom out, the size of the pixels in the Pixel Region tool decrease and more pixels are visible. To change the number of pixels displayed in the tool, without changing the magnification, resize the Pixel Region tool using the mouse.

As you zoom in or out, note how the size of the pixel region rectangle changes according to the magnification. You can resize the pixel region rectangle using the mouse. Resizing the pixel region rectangle changes the magnification of pixels displayed in the Pixel Region tool.

If the magnification allows, the Pixel Region tool overlays 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.



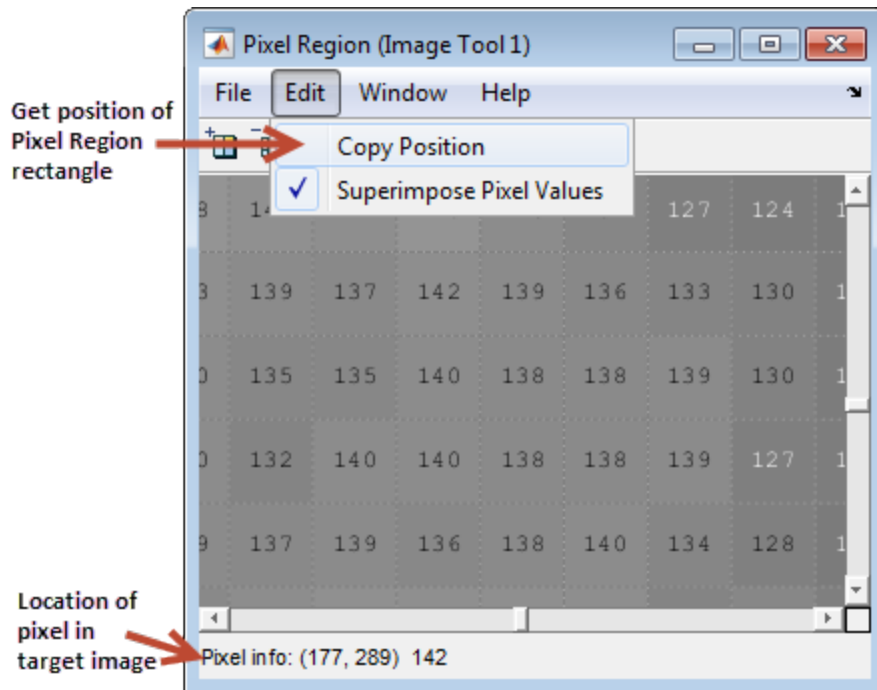
Determining the Location of the Pixel Region Rectangle

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.



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 “Print Images” on page 4-79 for more information.

Determine Image Display Range in Image Viewer

The Image Viewer 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

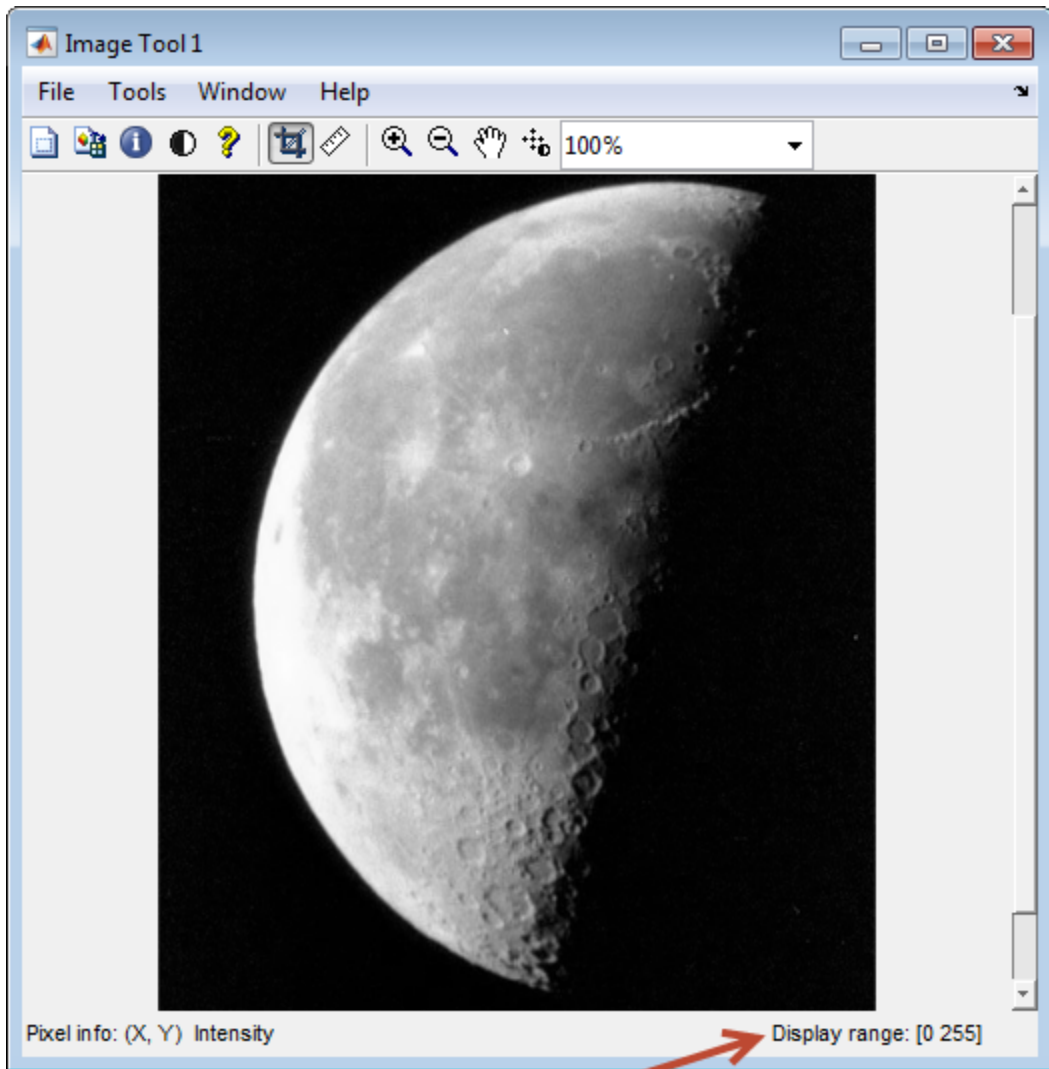
the `CData` value to map to the last color in the colormap (`cmax`). Data values in between are linearly scaled.

The Image Viewer displays this information in the Display Range tool at the bottom right corner of the window. The Image Viewer does not show the display range for indexed, truecolor, or binary images. (You can also obtain this information by opening a figure window with `imshow` and then calling `imshow_range` from the command line.)

For example, view an image in the Image Viewer.

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

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



Display Range tool

Measure Distance Between Pixels in Image Viewer

In this section...

“Determine Distance Between Pixels Using Distance Tool” on page 4-38



“Export Endpoint and Distance Data” on page 4-40

“Customize the Appearance of the Distance Tool” on page 4-40

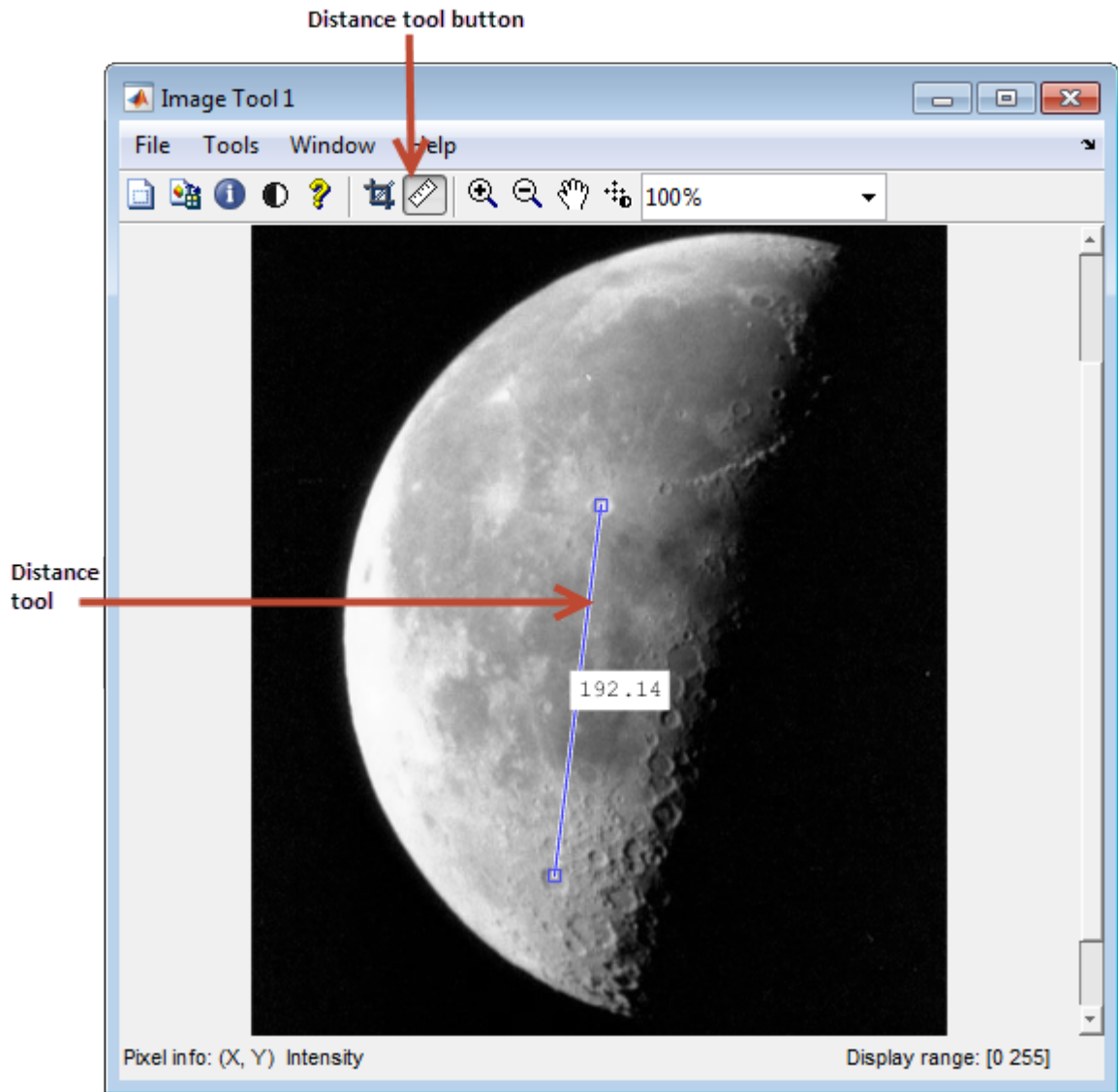
Determine Distance Between Pixels Using Distance Tool

- 1 Display an image in the Image Viewer.

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

- 2 Click the **Distance** tool button  in the Image Viewer toolbar or select **Measure Distance** from the **Tools** menu. The Distance tool allows you to measure distance with a click-and-drag approach. When you move the pointer over the image, the pointer changes to cross hairs . Position the cross hairs at the beginning of the region you wish to measure, hold down the mouse button, drag the cross hairs to the end of the region, and release the button.

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

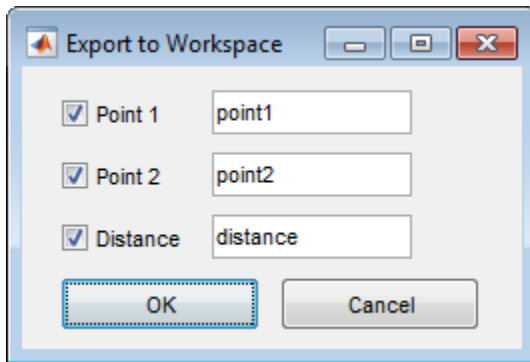


- 3 An alternative to using the Distance tool in Image Viewer is to open a figure window using `imshow` and then call `imdistanline`. The Distance tool appears as a horizontal

line displayed over the image. You can reposition the line or grab either of its endpoints to resize the tool.

Export Endpoint and Distance Data

To save the endpoint locations and distance information, right-click the Distance tool and choose the **Export to Workspace** 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  Attributes
  distance      1x1              8  double
  point1        1x2             16  double
  point2        1x2             16  double
```

Customize the Appearance of the Distance Tool

Using the Distance tool context menu, you can customize many aspects of the Distance tool appearance and behavior. Position the pointer over the line and right-click to access these context menu options.

- 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 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.

Get Image Information in Image Viewer App


To get information about the image displayed in the Image Viewer, 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.

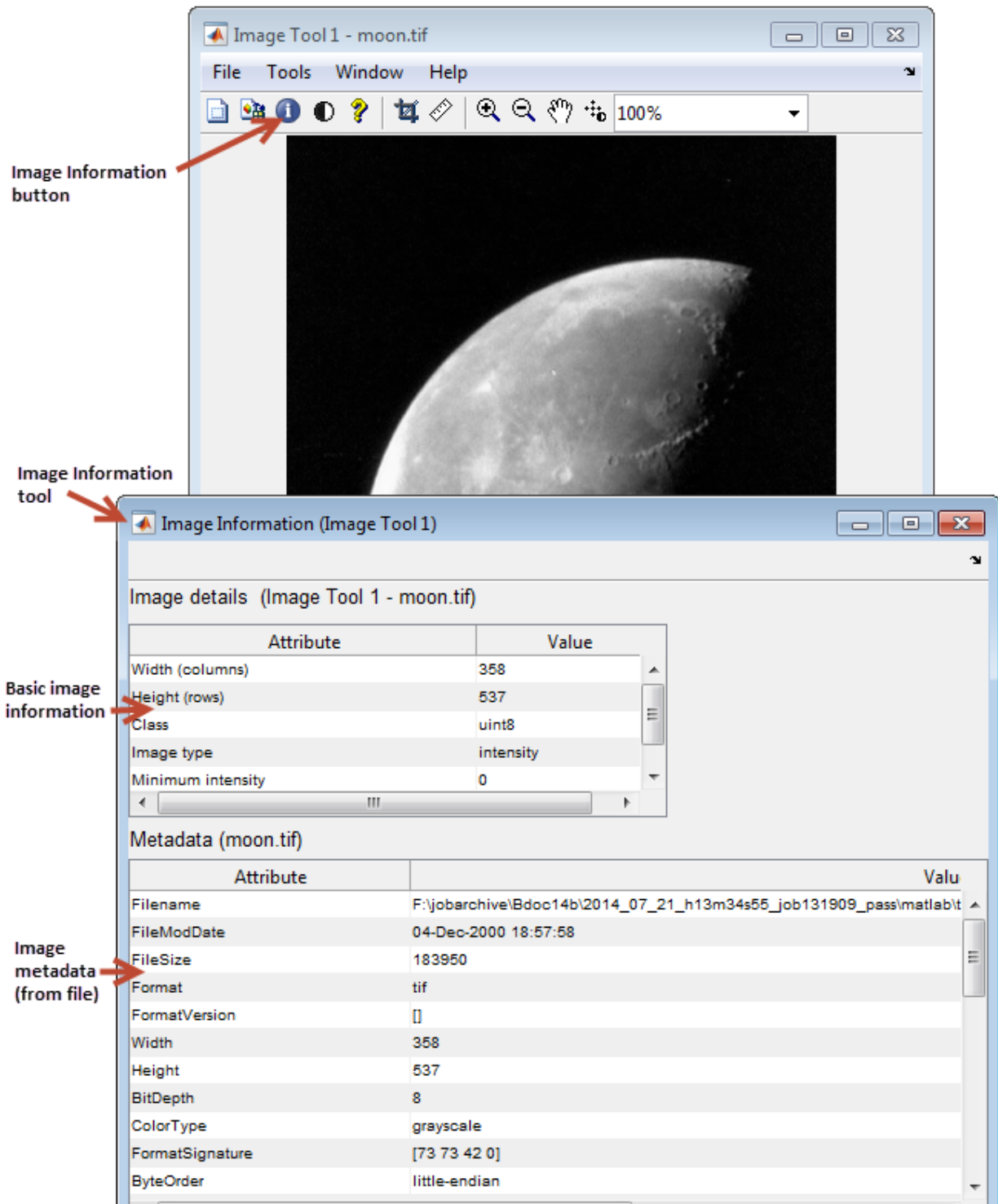
Note The Image Information tool can display image metadata only when you specify the filename containing the image to Image Viewer, e.g., `imtool('moon.tif')`.

For example, view an image in the Image Viewer.

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

Start the Image Information tool by clicking the Image Information button  in the Image Viewer toolbar or selecting the **Image Information** option from the **Tools** menu in the Image Viewer. (Another option is to open a figure window using `imshow` and then call `imageinfo` from the command line.)

The following figure shows the Image Viewer with the Image Information tool. If you specify a file name when you call the `imtool` function, the Image Information tool displays both basic image information and image metadata, as shown in the figure.



Adjust Image Contrast In Image Viewer App

In this section...

“Understanding Contrast Adjustment” on page 4-44

“Open the Adjust Contrast Tool” on page 4-45

“Adjust Image Contrast Using the Histogram Window” on page 4-48

“Adjust Image Contrast Using Window/Level Tool” on page 4-49

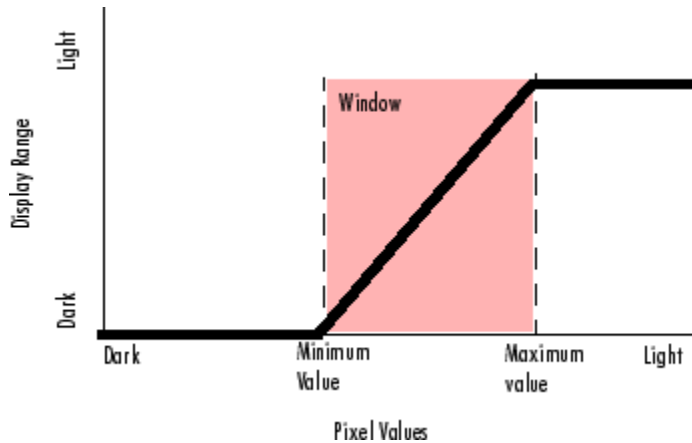
“Make Contrast Adjustments Permanent” on page 4-52

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 — see “Open the Adjust Contrast Tool” on page 4-45



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 Viewer 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.

Open the Adjust Contrast Tool

This section describes how to use the Adjust Contrast tool in the Image Viewer.

- 1 View an image in the Image Viewer.

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

You can also use the Adjust Contrast tool independent of the Image Viewer by calling the `imcontrast` function.

- 2 Click **Adjust Contrast**  in the Image Viewer toolbar, or select the **Adjust Contrast** option from the Image Viewer **Tools** menu. The Adjust Contrast tool

opens in a separate window containing a histogram of the image displayed in the Image Viewer. 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. The pixel values for the image are clustered in the middle of the display range. Adjusting the contrast spreads the pixel values across the display range, revealing much more detail in the image.

To adjust the contrast of the image, you can manipulate the red-tinted rectangular box, called a *window*, that the Adjust Contrast tool overlays on the histogram. By changing the size and position of this window using the mouse, you can modify the display range of the image and improve its contrast and brightness — see “Adjust Image Contrast Using the Histogram Window” on page 4-48.

Adjust Contrast tool button

Adjust Contrast tool

Adjust Contrast (Image Tool 1)

File Edit Window Help

Data Range
Minimum: 0
Maximum: 253

Window
Minimum: 0
Maximum: 248

Width: 248
Center: 124

Scale Display Range
 Match Data Range
 Eliminate outliers: 2 %
Apply


Adjust the histogram above.
Click 'Adjust Data' to apply the changes to image data.


Adjust Data

Contrast adjustment window

Image histogram

Click to change pixel values in image.

Note You can also use the Window/Level tool to adjust contrast and brightness using the mouse. (The name comes from medical applications.) Click **Window/Level**  in the Image Viewer toolbar or select the **Window/Level** option from the Image Viewer **Tools** menu. For more information about using the Window/Level tool, see “Adjust Image Contrast Using Window/Level Tool” on page 4-49.

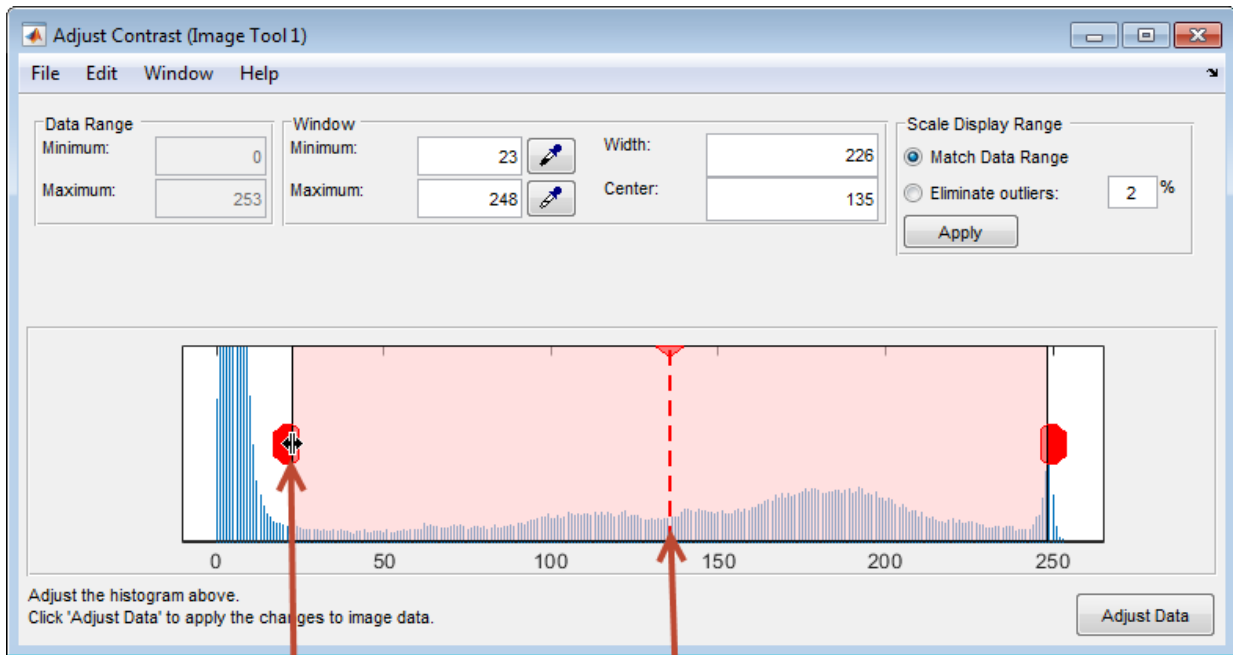
When you close the Adjust Contrast tool, the Window/Level tool remains active. To turn off the Window/Level tool, click the Window/Level button  or one of the navigation buttons in the Image Viewer toolbar.

Adjust Image Contrast Using the Histogram Window

To adjust image contrast using the Adjust Contrast tool, you manipulate the size of the red-tinted window displayed over the histogram, using any of the following methods.

- 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.
- 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 pointer becomes an eye dropper shape. Position the eye dropper pointer over the pixel in the image that you want to be the minimum (or maximum) value and click the mouse button.
- Specifying the size and position of the window in the **Width** and **Center** fields.
- Selecting the **Match data range** button in the Scale Display Range part of the tool. When you choose this option, the size of the window changes from the default display range to the data range of the image.
- 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 using the `stretchlim` function.)


The following figure shows these methods of adjusting contrast. The Image Viewer updates the display range values displayed in the lower right corner of the Image Viewer as you change the size of the window.



Drag either handle to resize the window.


Drag midline to move window.




Adjust Image Contrast Using Window/Level Tool

To start the Window/Level tool, click Window/Level  in the Image Viewer toolbar.

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. Note that any contrast adjustments you make using the Window/Level tool are reflected immediately in the Adjust Contrast tool. For example, if you increase the brightness, the window in the Adjust Contrast moves over the histogram.

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.

Mouse Motion		Effect
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 the **Window/Level** button in the Image Viewer toolbar, or click any of the navigation buttons in the toolbar.


Adjust Contrast with the Window/Level Tool

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

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


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

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

- 3 Click the **Window/Level** button  to start the tool, or select **Window/Level** from the **Tools** menu in the Image Viewer.

Window/Level tool button




- 4 Move the pointer over the image. The pointer changes to the Window/Level cursor .
- 5 Click and drag the left (or right) mouse button and move the pointer horizontally to the left or right to adjust the contrast, or vertically up or down to change the brightness.

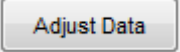
Make Contrast Adjustments Permanent

By default, the Adjust Contrast tool adjusts the values of the pixels used to display the image in the Image Viewer but does not change the actual image data. To modify pixel values in the image to reflect the contrast adjustments you made, you must click the **Adjust Data** button.

The following example illustrates this process.

- 1 Display an image in the Image Viewer. The example opens an image from a file.

```
imtool('moon.tif');
```
- 2 Start the Adjust Contrast tool by clicking the **Adjust contrast** button, , or by selecting **Adjust Contrast** from the **Tools** menu in the Image Viewer.
- 3 Adjust the contrast of the image. Use one of the mechanisms provided by Adjust Contrast tool, such as resizing the window over the histogram. See “Adjust Image Contrast Using the Histogram Window” on page 4-48. You can also adjust contrast using the Window/Level tool, moving the pointer over the image.
- 4 Adjust the image data to reflect the contrast adjustment you just made. Click the

 Adjust Data button in the Adjust Contrast Tool. When you click the Adjust Data button, the histogram will update. You can then adjust the contrast again, if necessary. If you have other interactive modular tool windows open, they will update automatically to reflect the contrast adjustment.

Note: The Adjust Data button is unavailable until you make a change to the contrast of the image.

Saving the Modified Image Data

By default, if you close the Image Viewer, it does not save the modified image data. To save these changed values, use the **Save As** option from the Image Viewer **File** menu

to store the modified data in a file or use the **Export to Workspace** option to save the modified data in a workspace variable.

Crop Image Using Image Viewer App


Cropping an image means creating a new image from a part of an original image. To crop an image using the Image Viewer, use the Crop Image tool. To use the Crop Image tool, follow this procedure.

By default, if you close the Image Viewer, it does not save the modified image data. To save the cropped image, you can use the **Save As** option from the Image Viewer **File** menu to store the modified data in a file or use the **Export to Workspace** option to save the modified data in the workspace variable.

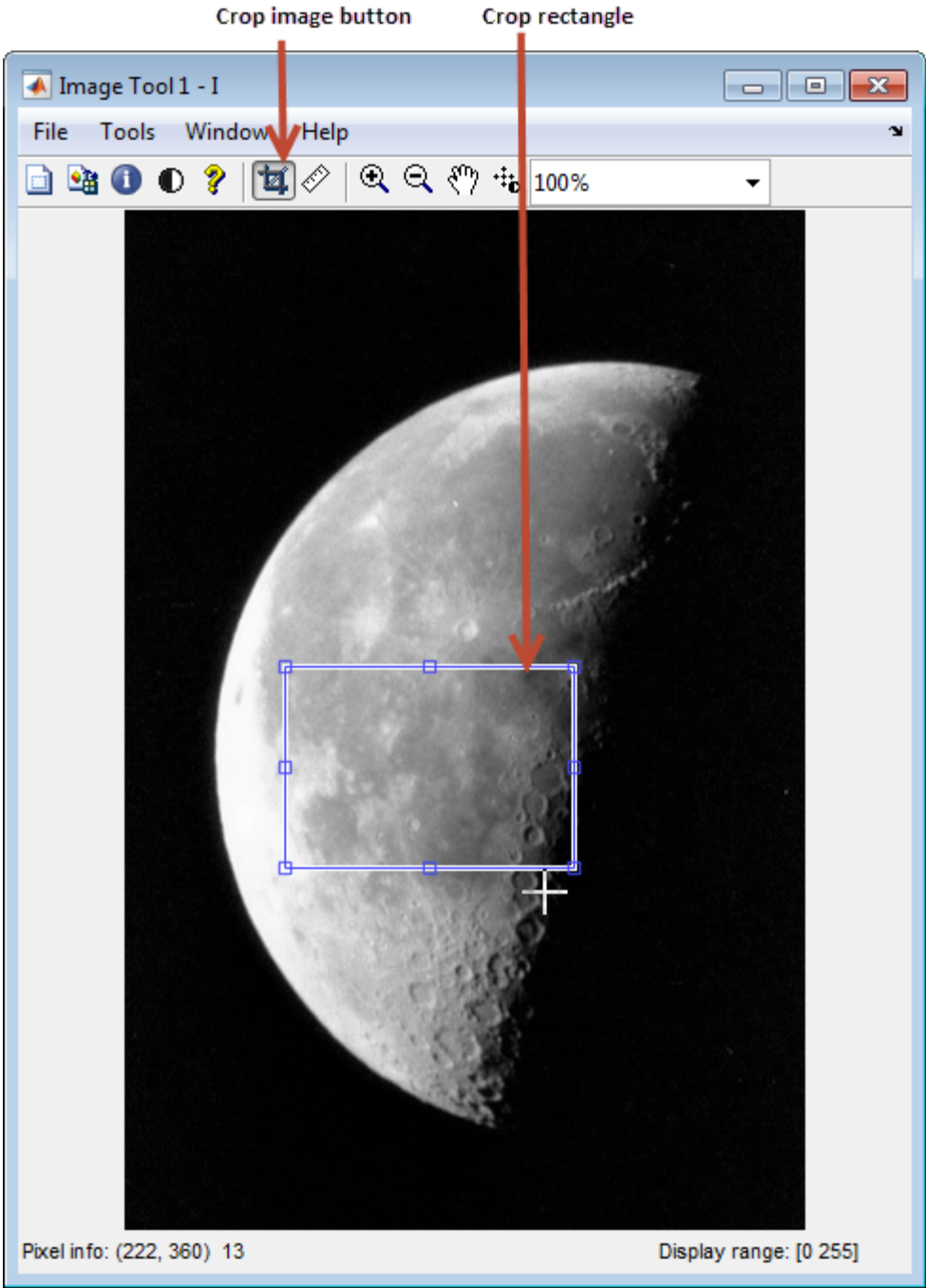
- 1 View an image in the Image Viewer.

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

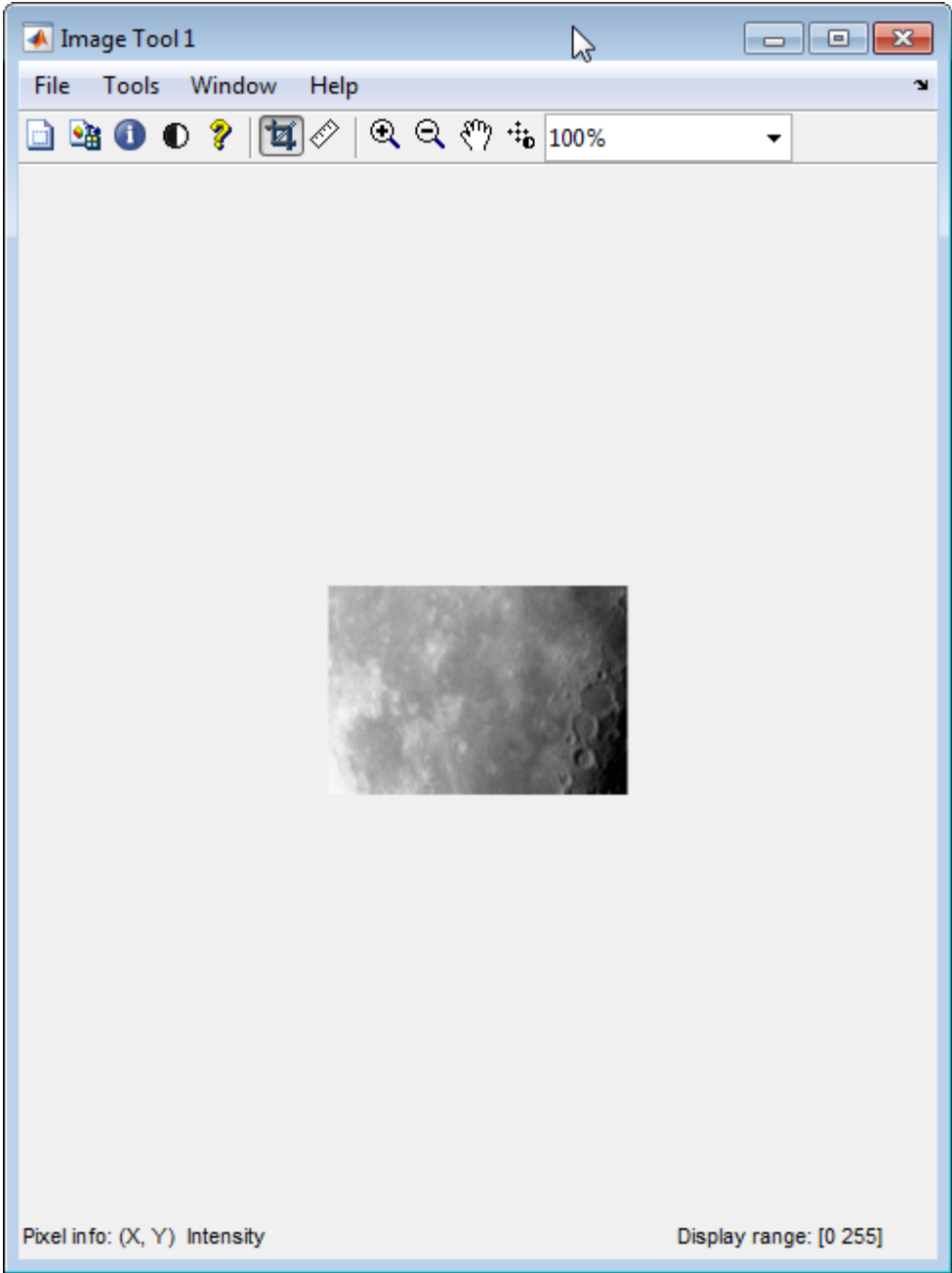
- 2 Start the Crop Image tool by clicking **Crop Image**  in the Image Viewer toolbar or selecting **Crop Image** from the Image Viewer **Tools** menu. (Another option is to open a figure window with `imshow` and call `imcrop` from the command line.) When

you move the pointer over the image, the pointer changes to cross hairs .

- 3 Define the rectangular crop region, by clicking and dragging the mouse over the image. You can fine-tune the crop rectangle by moving and resizing the crop rectangle using the mouse. Or, if you want to crop a different region, move to the new location and click and drag again. To zoom in or out on the image while the Crop Image tool is active, use **Ctrl+Plus** or **Ctrl+Minus** keys. Note that these are the **Plus(+)** and **Minus(-)** keys on the numeric keypad of your keyboard. The following figure shows a crop rectangle being defined using the Crop Image tool.



- 4 When you are finished defining the crop region, perform the crop operation. Double-click the left mouse button or right-click inside the region and select **Crop Image** from the context menu. The Image Viewer displays the cropped image. If you have other modular interactive tools open, they will update to show the newly cropped image.



- 5 To save the cropped image, use the **Save as** option or the **Export to Workspace** option on the Image Viewer **File** menu.

View Image Sequences in Video Viewer App

This section describes how to use the Video Viewer app to view image sequences and provides information about configuring the Video Viewer app.

In this section...

“View MRI Sequence Using Video Viewer App” on page 4-59

“Configure Video Viewer App” on page 4-62

“Specifying the Frame Rate” on page 4-65

“Specify Color Map” on page 4-66

“Get Information about an Image Sequence” on page 4-66

View MRI Sequence Using Video Viewer App

- 1 Load the image sequence into the MATLAB workspace. For this example, load the MRI data from the file `mrystack.mat`, which is included in the `imdata` folder.

```
load mrystack
```

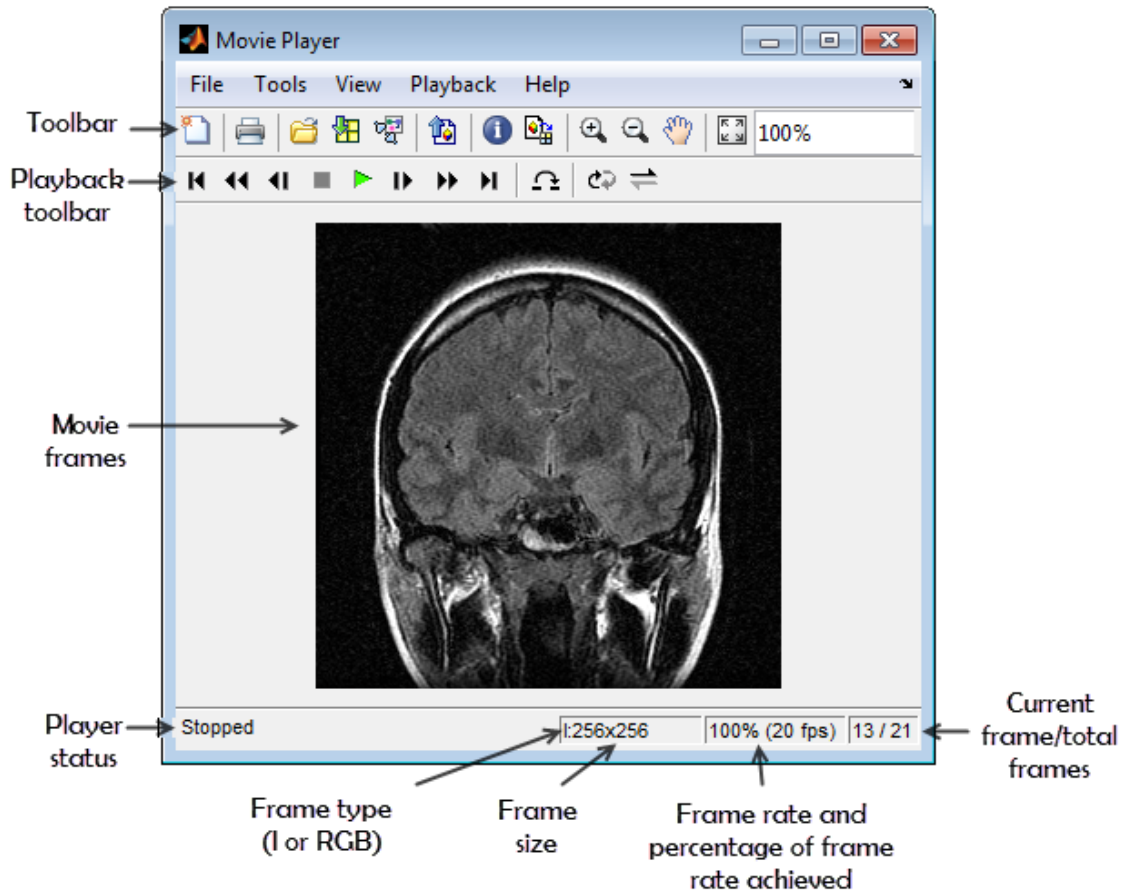
This places a variable named `mrystack` in your workspace. The variable is an array of 21 grayscale frames containing MRI images of the brain. Each frame is a 256-by-256 array of `uint8` data.

```
mrystack      256x256x21      1276256      uint8
```


- 2 Click the Video Viewer app in the apps gallery and select the **Import from workspace** option on the File menu. You can also call `implay`, specifying the name of the image sequence variable as an argument.






```
implay(mrystack)
```

The Video Viewer opens, displaying the first frame of the image sequence. Note how the Video Viewer displays information about the image sequence, such as the size of each frame and the total number of frames, at the bottom of the window.









3 Explore the image sequence using Video Viewer Playback controls.

To view the image sequence or video as an animation, click the Play button  in the Playback toolbar, select **Play** from the Playback menu, or press **P** or the Space bar. By default, the Video Viewer plays the image sequence forward, once in its entirety, but you can view the frames in the image sequence in many ways, described in this table. As you view an image sequence, note how the Video Viewer updates the Status Bar at the bottom of the window.

Viewing Option	Playback Control	Keyboard Shortcut
Specify the direction in which to play the image sequence.	Click the Playback mode button  in the Playback toolbar or select Playback Modes from the Playback menu. You can select forward, backward, or autoreverse. As you click the playback mode button, it cycles through these options and the appearance changes to indicate the current selection.	A
View the sequence repeatedly.	Click the Repeat button  in the Playback toolbar or select Playback Modes > Repeat from the Playback menu. You toggle this option on or off.	R
Jump to a specific frame in the sequence.	Click the Jump to button  in the Playback toolbar or select Jump to from the Playback menu. This option opens a dialog box in which you can specify the number of the frame.	J
Stop the sequence.	Click the Stop button  in the Playback toolbar or select Stop from the Playback menu. This button is only enabled when an image sequence is playing.	S
Step through the sequence, one frame at a time, or jump to the beginning or end of the sequence (rewind).	Click one of the navigation buttons  in the Playback toolbar, in the desired direction, or select an option, such as Fast Forward or Rewind from the Playback menu.	Arrow keys Page Up/Page Down L (last frame) F (first frame)

- 4 Change the view of the image sequence or examine a frame more closely.

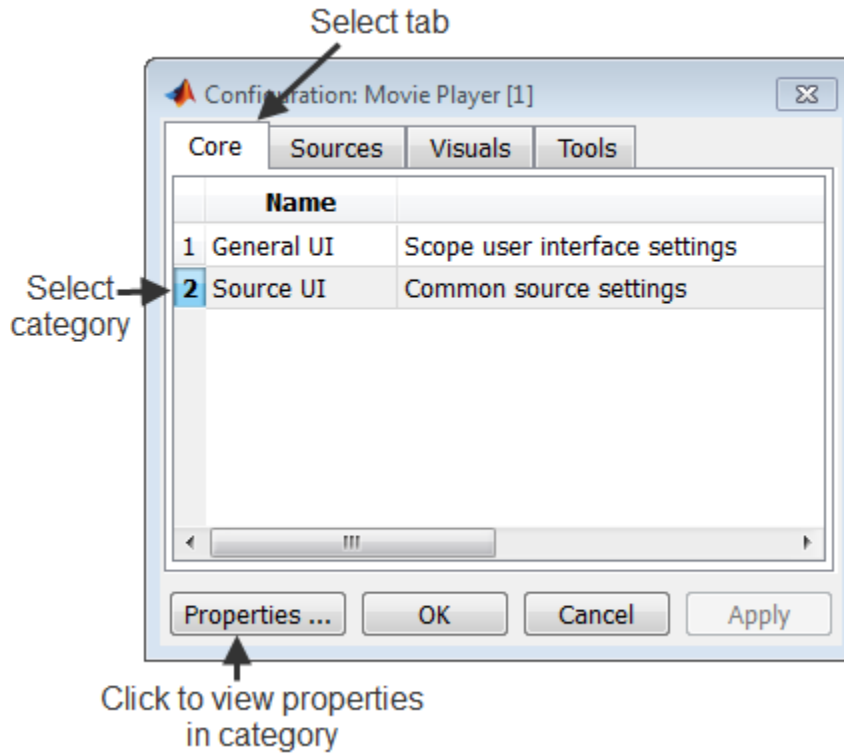
The Video Viewer supports several tools listed in the Tools menu and on the Toolbar that you can use to examine the frames in the image sequence more closely.

Viewing Option	Playback Control
Zoom in or out on the image, and pan to change the view.	Click one of the zoom buttons   in the toolbar or select Zoom In or Zoom Out from the Tools menu. Click the Pan button  in the toolbar or select Pan from the Tools menu. If you click Maintain fit to window button  in the toolbar or select Maintain fit to window or from the Tools menu, the zoom and pan buttons are disabled.
Examine an area of the current frame in detail.	Click the Pixel region button  in the Playback toolbar or select Pixel Region from the Tools menu.
Export frame to Image Viewer	Click the Export to Image Tool button  in the Playback toolbar or select Export to Image Tool from the File menu. The Video Viewer app opens an Image Viewer containing the current frame.

Configure Video Viewer App

The Video Viewer app Configuration dialog box enables you to change the appearance and behavior of the player. To open the Configuration dialog box, select **File > Configuration > Edit**. (To load a preexisting configuration set, select **File > Configuration > Load**.)

The Configuration dialog box contains four tabs: Core, Sources, Visuals, and Tools. On each tab, select a category and then click Properties to view configuration settings.



The following table lists the options that are available for each category on every pane.

Pane	Option Category	Option Descriptions
Core	General UI	<p>Display the full source path in the title bar check box — Select to display the full path to the video data source in the title bar. By default, Movie Player displays a shortened name in the title bar.</p> <p>Open message log menu — Specify when the Message log window opens. You can use the Message log window to debug issues with video playback. By default, the window only opens for failure messages.</p>

Pane	Option Category	Option Descriptions
Core	Source UI	<p>Keyboard commands respect playback mode check box — Select to make keyboard shortcut keys aware of your playback mode selection. If you clear this check box, the keyboard shortcut keys behave as if the playback mode is set to Forward play and Repeat is set to Off.</p> <p>Recently used sources list parameter — Specifies the number of sources listed in the File menu.</p>
Sources	Simulink	Select the Enabled check box to enable connections to Simulink models. You must have Simulink installed.
Sources	File	<p>Select the Enabled check box to enable connections to files (the default).</p> <p>Default open file path parameter — Specify the directory that is displayed in the Connect to File dialog box when you click File > Open.</p>
Sources	Workspace	Select the Enabled check box to enable connections to variables in the workspace (the default). There are no options associated with this selection.
Visuals	Video	Select the Enabled check box to use video visualization.
Tools	Image Tool	<p>Select the Enabled check box to include the Image Viewer.</p> <p>Open new Image Tool window for each export check box — Opens a new Image Viewer for each exported frame.</p>
Tools	Pixel Region	Select the Enabled check box to include the Pixel Region tool in the Video Viewer app (the default).
Tools	Image Navigation Tools	Select the Enabled check box to include the zoom and pan tools in the Video Viewer app (the default).
Tools	Instrumentation Sets	Select the Enabled check box to include instrumentation sets in the Video Viewer app. Provides a way to save your current configuration.

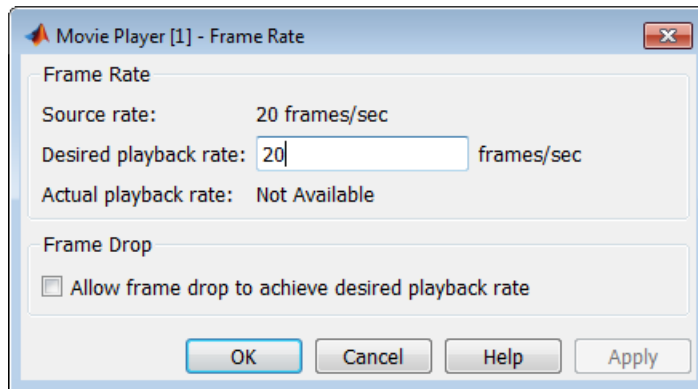
Save Image Viewer App Configuration Settings

To save your configuration settings for future use, select **File > Configuration Set > Save as**.

Note: By default, the Video Viewer uses the configuration settings from the file `implay.cfg`. If you want to store your configuration settings in this file, you should first create a backup copy of the file.

Specifying the Frame Rate

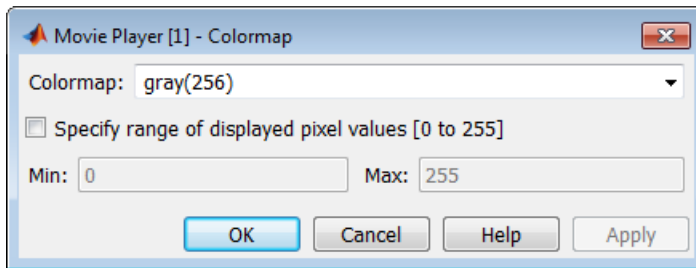
To decrease or increase the playback rate, select **Frame Rate** from the Playback menu, or use the keyboard shortcut **T**. The Frame Rate dialog box displays the frame rate of the source, lets you change the rate at which the Video Viewer app plays the image sequence or video, and displays the actual playback rate. The *playback rate* is the number of frames the Video Viewer processes per second.



If you want to increase the actual playback rate, but your system's hardware cannot keep up with the desired rate, select the **Allow frame drop to achieve desired playback rate** check box. This parameter enables the Video Viewer app to achieve the playback rate by dropping frames. When you select this option, the Frame Rate dialog box displays several additional options that you can use to specify the minimum and maximum refresh rates. If your hardware allows it, increase the refresh rate to achieve a smoother playback. However, if you specify a small range for the refresh rate, the computed frame replay schedule may lead to a choppy replay, and a warning will appear.

Specify Color Map

To specify the colormap to apply to the intensity values, select **Colormap** from the Tools menu, or use the keyboard shortcut **C**. The Video Viewer displays a dialog box that enables you to change the colormap.



Use the **Colormap** parameter to specify a particular colormap.

If you know that the pixel values do not use the entire data type range, you can select the **Specify range of displayed pixel values** check box and enter the range for your data. The dialog box automatically displays the range based on the data type of the pixel values.

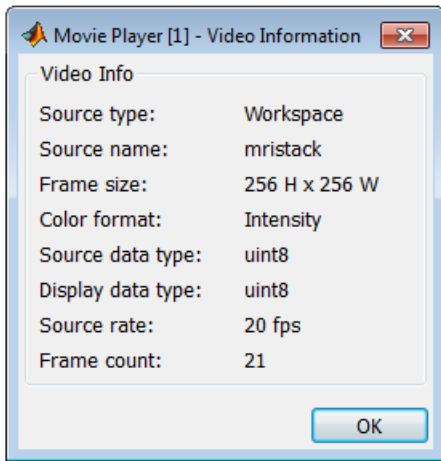
Get Information about an Image Sequence

To view basic information about the image data, click the Video Information button



in the Video Viewer toolbar or select **Video Information** from the Tools menu.

The Video Viewer displays a dialog box containing basic information about the image sequence, such as the size of each frame, the frame rate, and the total number of frames.



View Image Sequence as Montage

This example shows how to view multiple frames in a multiframe array at one time, use the `montage` function. `montage` displays all the image frames, arranging them into a rectangular grid. The montage of images is a single image object. The image frames can be grayscale, indexed, or truecolor images. If you specify indexed images, they all must use the same colormap.

Create an array of truecolor images.

```
onion = imread('onion.png');  
onionArray = repmat(onion, [ 1 1 1 4 ]);
```

Display all the images at once, in a montage. By default, the `montage` function displays the images in a grid. The first image frame is in the first position of the first row, the next frame is in the second position of the first row, and so on.

```
montage(onionArray);
```



To specify a different number of rows and columns, use the 'size' parameter. For example, to display the images in one horizontal row, specify the 'size' parameter with the value [1 NaN] . Using other `montage` parameters you can specify which images you want to display and adjust the contrast of the images displayed.

```
montage(onionArray, 'size', [1 NaN]);
```



Convert Multiframe Image to 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.

To play the movie, use the `implay` function.

```
implay(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);  
implay(mov);
```

Note To view a MATLAB movie, you must have MATLAB software installed. To make a movie that can be run outside the MATLAB environment, use the `VideoWriter` class to create a movie in a standard video format, such as, AVI.

Display Different Image Types

In this section...

“Display Indexed Images” on page 4-71

“Display Grayscale Images” on page 4-72

“Display Binary Images” on page 4-73

“Display Truecolor Images” on page 4-75

If you need help determining what type of image you are working with, see “Image Types in the Toolbox” on page 2-9.

Display 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.

Display Grayscale Images

To display a grayscale image, call the `imshow` function or open the Image Viewer app. This documentation uses the variable name `I` to represent a grayscale image in the workspace.

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 “Displaying Colors” on page 14-2 for a detailed explanation.)

Display Grayscale Images with Unconventional Ranges

In some cases, the image data you want to display as a grayscale image could 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 could 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

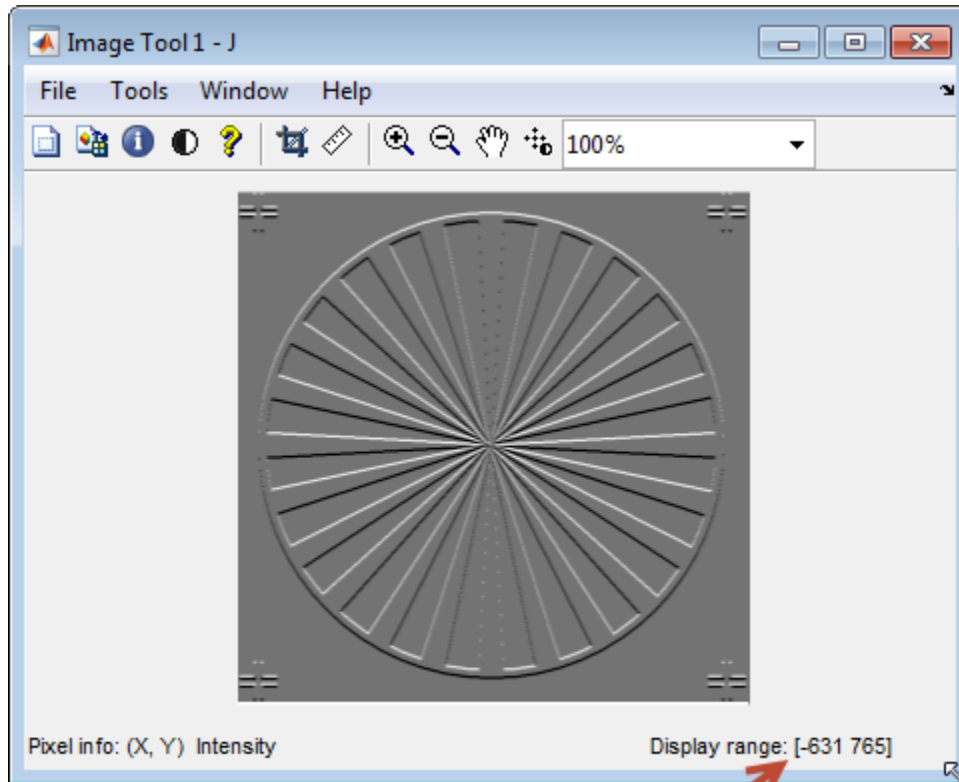
```
imtool(I, 'DisplayRange', [low high])
```

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 in the Image Viewer, using the automatic

scaling option. If you execute this example, note the display range specified in the lower right corner of the Image Viewer window.

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



Unconventional display range

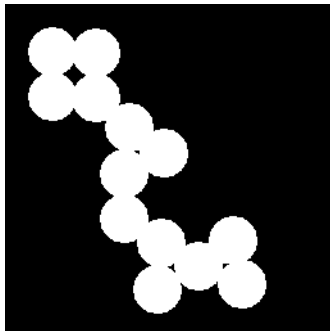
Display Binary Images

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, call the `imshow` function or open the Image Viewer app. 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

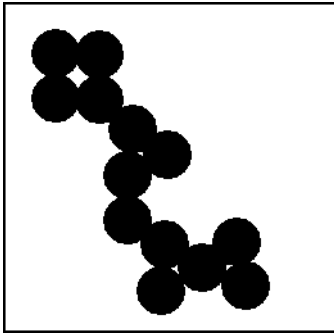
```
BW = imread('circles.png');  
imshow(BW)
```



Change Display Colors of Binary Image

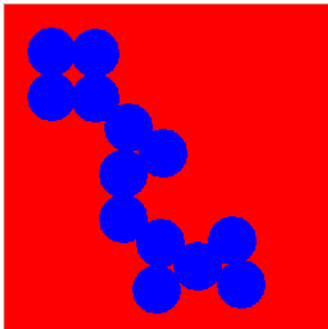
You might prefer to invert binary images when you display them, so that 0 values are displayed as white and 1 values are displayed as black. To do this, use the NOT (`~`) operator in MATLAB. (In this figure, a box is drawn around the image to show the image boundary.) For example:

```
imshow(~BW)
```



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])
```



Display 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, call the `imshow` function or open the Image Viewer app. 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)`



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 “Displaying Colors” 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 “Display Indexed Images” on page 4-71.

Add Colorbar to Displayed Image

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.

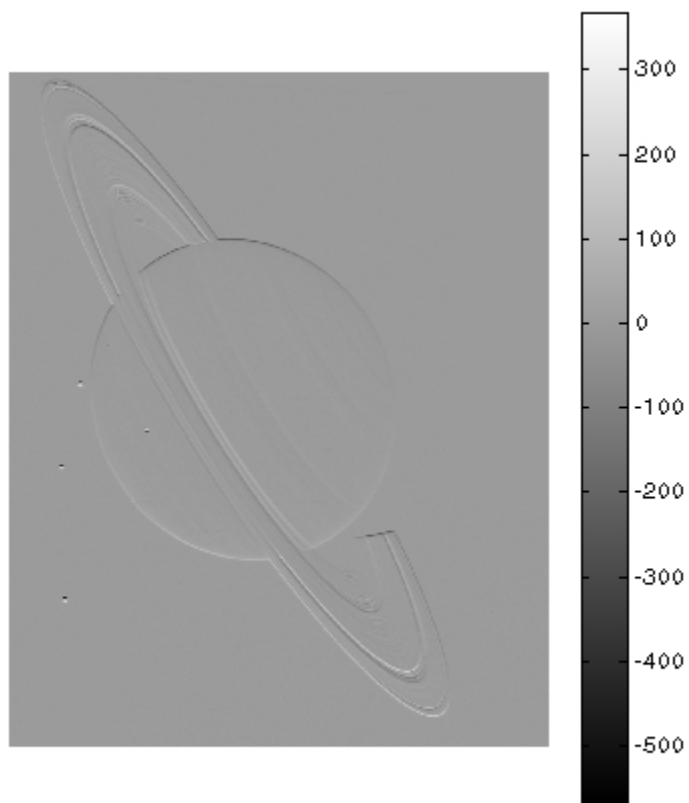
When you add a colorbar to an axes object that contains an image object, the colorbar indicates the data values that the different colors in the image correspond to.

If you want to add a colorbar to an image displayed in the Image Viewer, select the **Print to Figure** option from the **File** menu. The Image Viewer 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 “Display Grayscale Images with Unconventional Ranges” on page 4-72.

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',[0,255]), colorbar
```



Print 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 to Files” 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 Viewer, you must use the **Print to Figure** option on the **File** menu. When you choose this option, the Image Viewer 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.

Handle Graphics Properties That Impact Printing

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 could 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 “Overview of 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.

Image Processing Toolbox Preferences

In this section...

“Retrieve Values of Toolbox Preferences” on page 4-80

“Set Values of Toolbox Preferences” on page 4-80

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

Retrieve Values of Toolbox Preferences

To determine the current value of Image Processing Toolbox preferences, you can look in the Preferences dialog box or use the `iptgetpref` function.

To open the Preference dialog box, click **Preferences** in the **Home** tab in the MATLAB desktop. In the Preferences dialog box, select Image Processing Toolbox. You can also access Image Processing Toolbox preferences from the Image Viewer **File** menu, or by typing `iptprefs` at the command line.

To retrieve the values of Image Processing Toolbox preferences programmatically, type `iptgetpref` at the command prompt. The following example uses `iptgetpref` to retrieve the value to determine the value of the `ImtoolInitialMagnification` preference.

```
iptgetpref('ImtoolInitialMagnification')
```

```
ans =
```

```
100
```

Preference names are case insensitive and can be abbreviated. For a complete list of toolbox preferences, see the `iptprefs` reference page.

Set Values of Toolbox Preferences

To set the value of Image Processing Toolbox preferences, you can use the Preferences dialog box or use the `iptsetpref` function.

To open the Preference dialog box, click **Preferences** in the **Home** tab in the MATLAB desktop. In the Preferences dialog box, select Image Processing Toolbox. You can also access Image Processing Toolbox preferences from the Image Viewer **File** menu, or by typing `iptprefs` at the command line.

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 a table of the available preferences, see the `iptprefs` reference page.

Building GUIs with Modular Tools

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

- “Modular Interactive Tools” on page 5-2
- “Displaying the Target Image” on page 5-9
- “Creating the Modular Tools” on page 5-10
- “Customizing Modular Tool Interactivity” on page 5-29
- “Creating Your Own Modular Tools” on page 5-34

Modular Interactive Tools

The toolbox includes several 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. Using the tools typically involves the following steps.

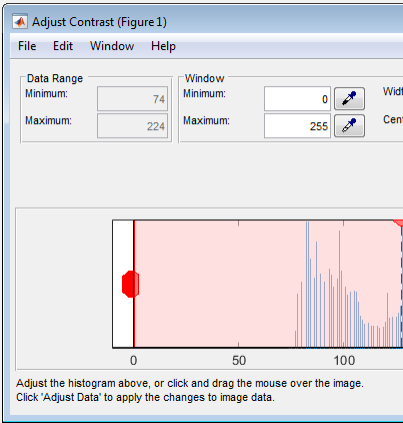
Step	Description	Notes
1	Display the image to be processed (called the <i>target image</i>) in a figure window.	Use the <code>imshow</code> function to display the target image, see “Displaying the Target Image” on page 5-9.
2	Create the modular tool, associating it with the target image.	<p>You use the modular tool creation functions to create the tools — see Summary of Modular Tools</p> <p>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 “Creating the Modular Tools” on page 5-10.</p> <p>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 5-14 for more information.</p> <p>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 5-18 for more information.</p>

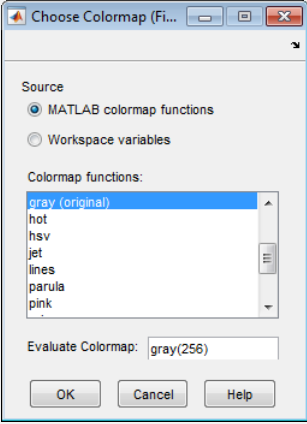
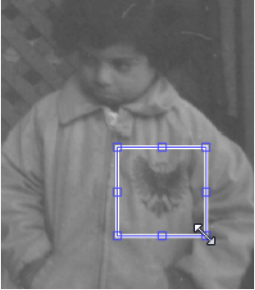
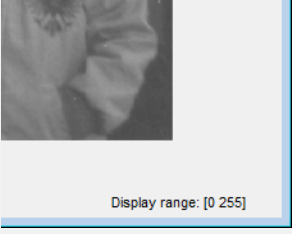
Step	Description	Notes
3	Set up interactivity between the tool and the target image. (Optional)	The modular tools all set up their interactive connections to the target image automatically. However, you can also specify custom connectivity using modular tool APIs. See “Customizing Modular Tool Interactivity” on page 5-29 for more information.


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.

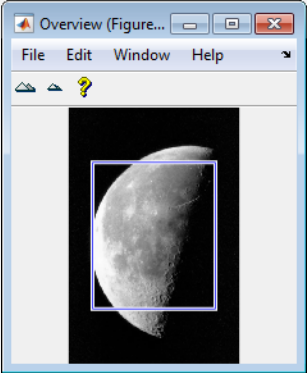
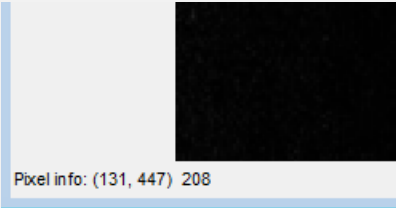
Note: The Image Processing Toolbox GUI, Image Tool, uses these modular tools — see “Using the Image Viewer App to Explore Images” on page 4-13.

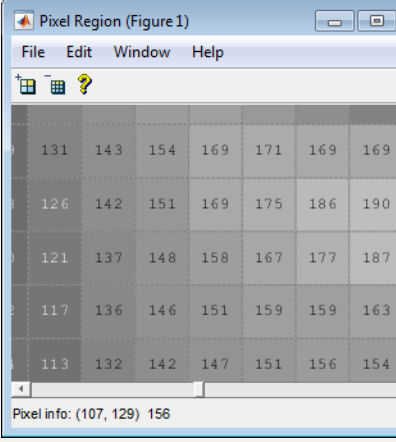
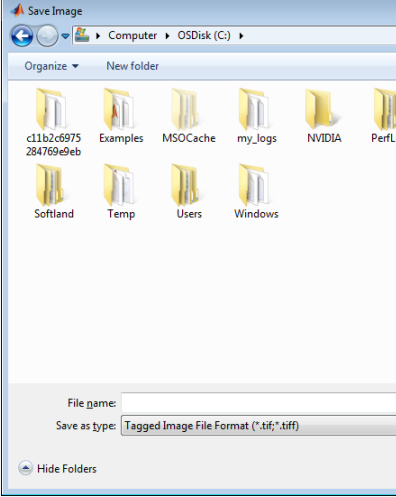
Summary of Modular Tools

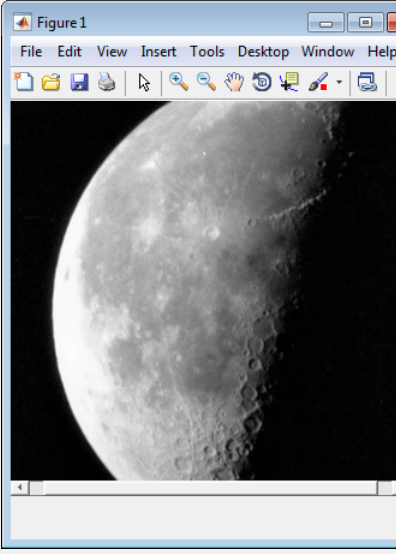
Modular Tool	Example	Description
Adjust Contrast tool		<p>Displays a histogram of the target image and enables interactive adjustment of contrast and brightness by manipulation of the display range.</p> <p>Use the <code>imcontrast</code> function to create the tool in a separate figure window and associate it with an image.</p>

Modular Tool	Example	Description
Choose Colormap tool		<p>Allows you to change the colormap of the target figure. You can select one of the MATLAB colormaps, select a colormap variable from the MATLAB workspace, or enter a custom MATLAB expression.</p> <p>Use the <code>imcolormaptool</code> function to launch the tool in a separate figure window.</p>
Crop Image tool		<p>Displays a draggable, resizable rectangle on an image. You can move and resize the rectangle to define the crop region. Double-click to perform the crop operation or select Crop Image from the context menu.</p> <p>Use the <code>imcrop</code> function to create the tool 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>imshow_range</code> function to create the tool, associate it with an image, and embed it in a figure or uipanel.</p>

Modular Tool	Example	Description
Distance tool		<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 <code>XData</code> and <code>YData</code> 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>
Image Information tool <code>im</code>		<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>
Magnification box		<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> <p>Note: The target image must be contained in a scroll panel.</p>

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 uipanel that can be embedded within another figure or uipanel.</p> <hr/> <p>Note: The target image must be contained in a scroll panel.</p>
<p>Pixel Information tool</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 uipanel.</p> <p>If you want to display only the pixel values, without the Pixel info label, use <code>impixelinfoval</code>.</p>

Modular Tool	Example	Description
Pixel Region tool		<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 uipanel that can be embedded within another figure or uipanel.</p>
Save Image tool		<p>Display the Save Image dialog box. Use this to specify the name of an output image and choose the file format used to store the image.</p> <p>Use <code>imsave</code> to create the tool in a separate figure window and associate it with an image.</p>

Modular Tool	Example	Description
Scroll Panel tool		<p>Display target image in a scrollable panel.</p> <p>Use <code>imshow</code> to add a scroll panel to an image displayed in a figure window.</p>

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.

Creating the Modular Tools

In this section...

“Overview” on page 5-10

“Associating Modular Tools with a Particular Image” on page 5-11

“Getting the Handle of the Target Image” on page 5-13

“Specifying the Parent of a Modular Tool” on page 5-14

“Positioning the Modular Tools in a GUI” on page 5-18

“Build a Pixel Information GUI” on page 5-20

“Adding Navigation Aids to a GUI” on page 5-22

Overview

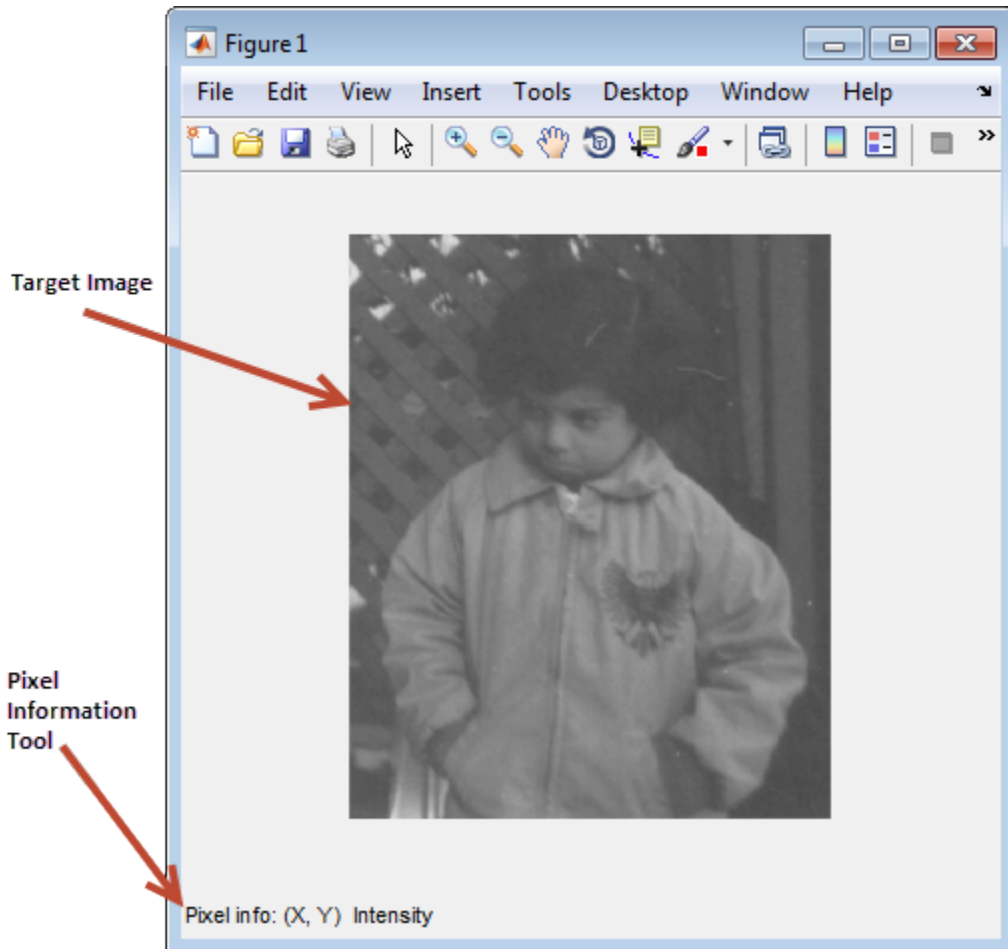
To associate a modular tool with a target image displayed in a MATLAB figure window, you must create the tool using the appropriate tool creation function. You specify a handle to the target image as an argument to the tool creation function. The function creates the tool and automatically sets up the interactivity connection between the tool and the 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 `imshowarray` 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 pointer over the image, the tool displays the x - and y -coordinates and value of the pixel under the pointer.



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.

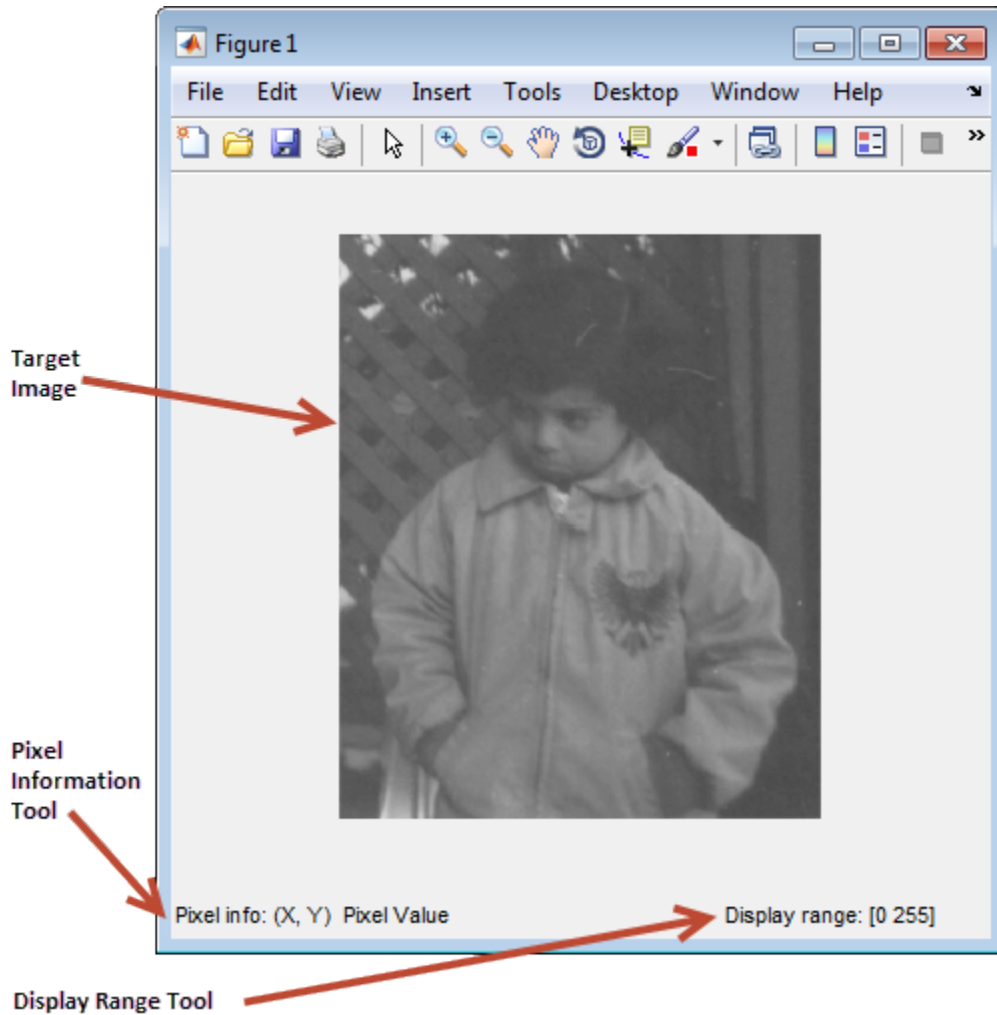
Continuing the example in the previous section, you might want to add the Display Range tool to the figure window that already contains the Pixel Information tool. To do this, call the `imshow_range` function, specifying the handle to the target image. You

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 = imdisplayrange(himage);
```

Note that the example retrieves handles to the uipanel objects created by the `impixelinfo` and `imdisplayrange` 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 5-18 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 pointer over the image (or images) in the figure, the Display Range tool shows the display range of the image.



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

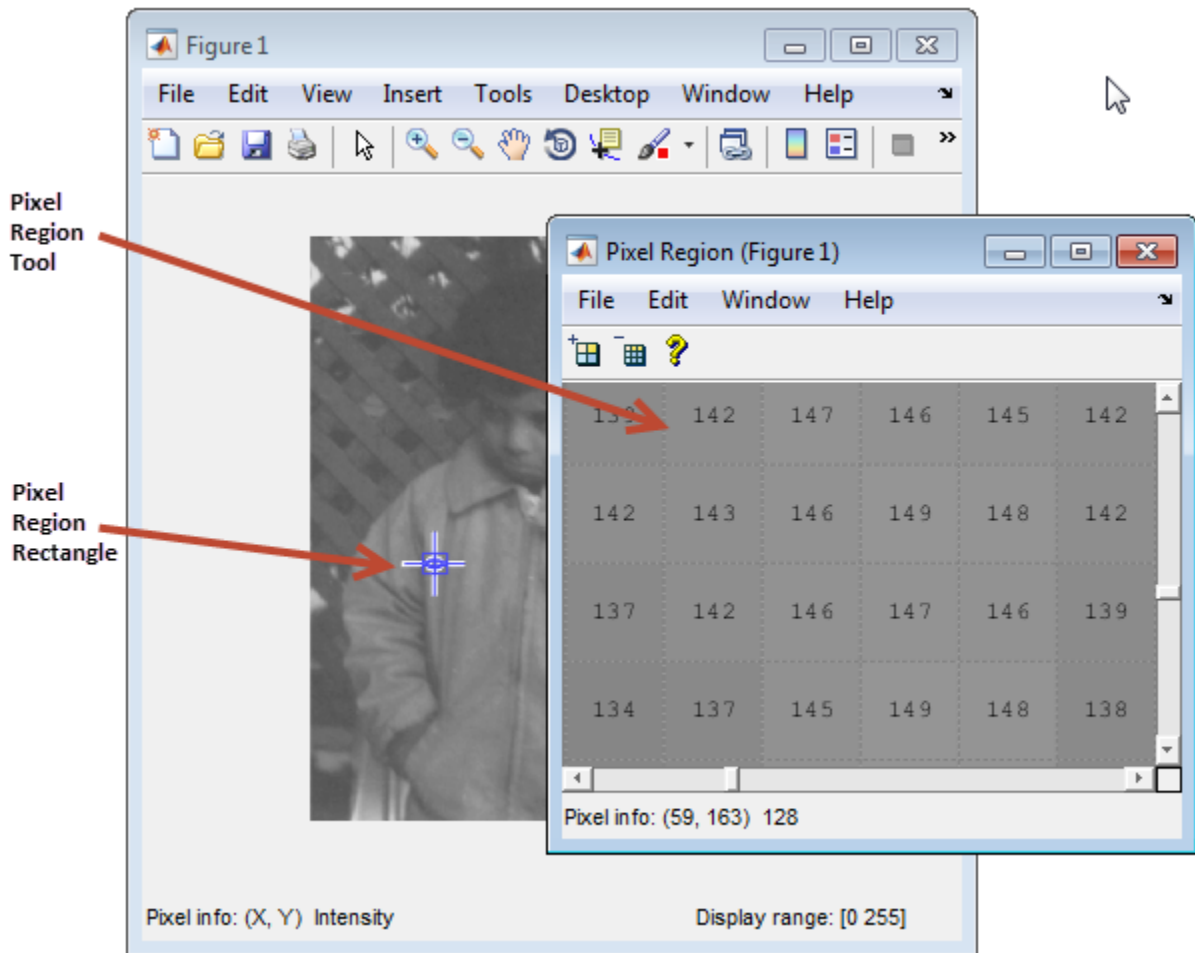
Two of the tools, the Pixel Region tool and the Overview tool, have a separate creation function to provide this capability. Their primary creation functions, `imoverview` and `impixelregion`, open the tools in a separate figure window. To specify a different parent, you must use the `imoverviewpanel`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



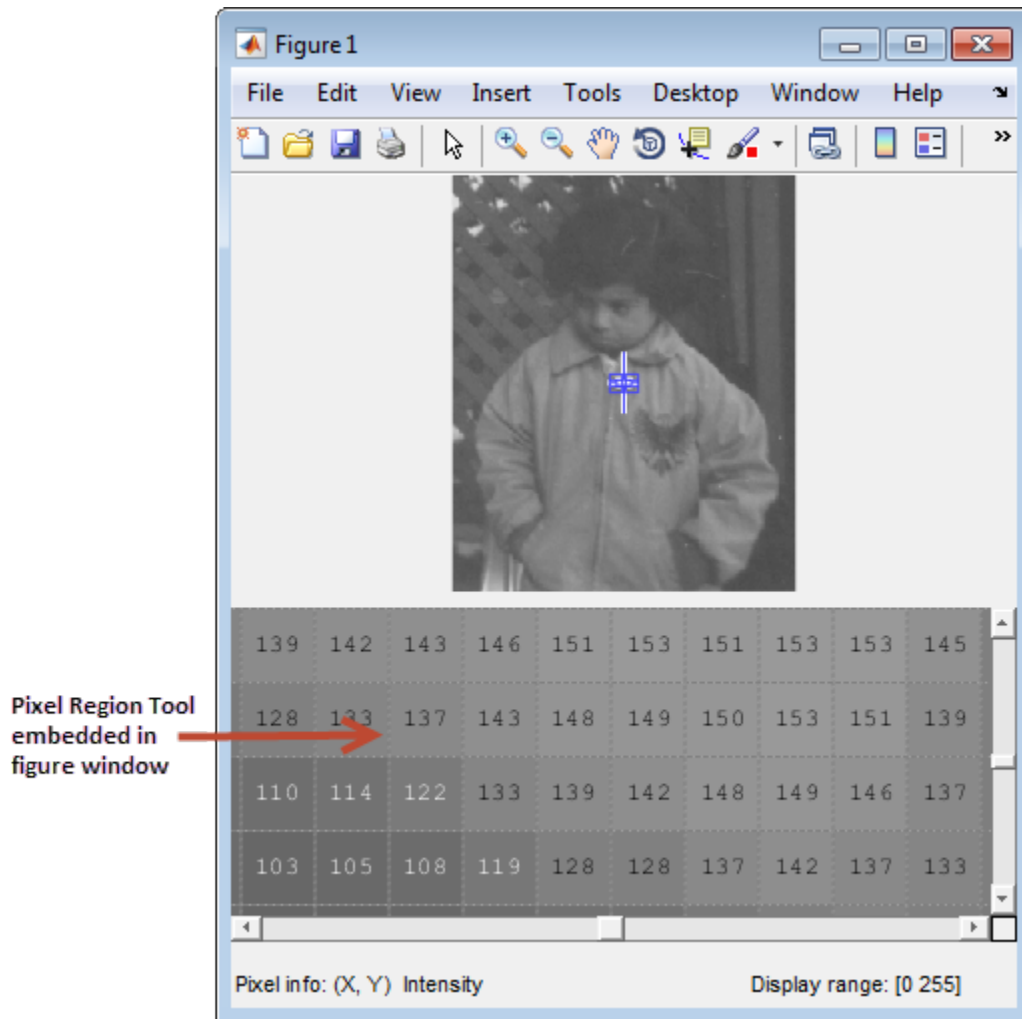
To embed the Pixel Region tool in the same window as the target image, you must specify the handle of the target image's parent figure as the parent of the Pixel Region tool when you create it.

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 5-18 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.



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 5-14, 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])
```

Build a Pixel Information GUI

This example shows how to use the tools to create a simple GUI that provides information about 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 5-22.

```
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
```

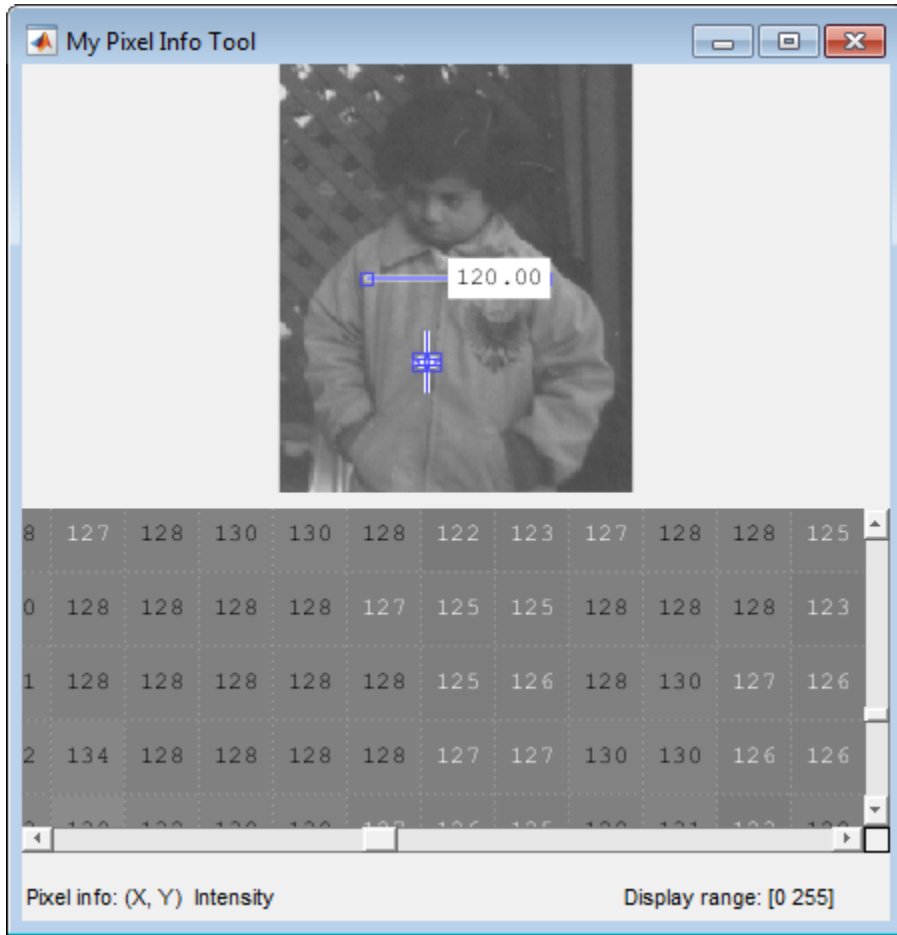


```
hpixreg = impixelregionpanel(hfig,himage);  
  
% Reposition the Pixel Region tool to fit in the figure  
% window, leaving room for the Pixel Information and  
% Display Range tools.  
set(hpixreg, 'units','normalized','position',[0 .08 1 .4])
```

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.



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.

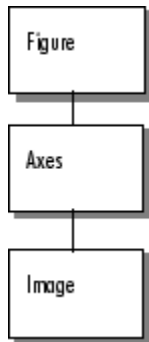
- “Understanding Scroll Panels” on page 5-23 — Adding a scroll panel to an image display changes the relationship of the graphics objects used in the display. This section provides some essential background.
- “Example: Building a Navigation GUI for Large Images” on page 5-26 — This section shows how 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.

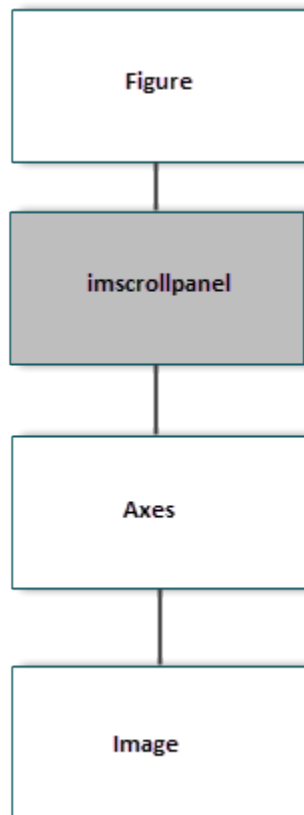


Object Hierarchy of Image Displayed in a Figure

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.

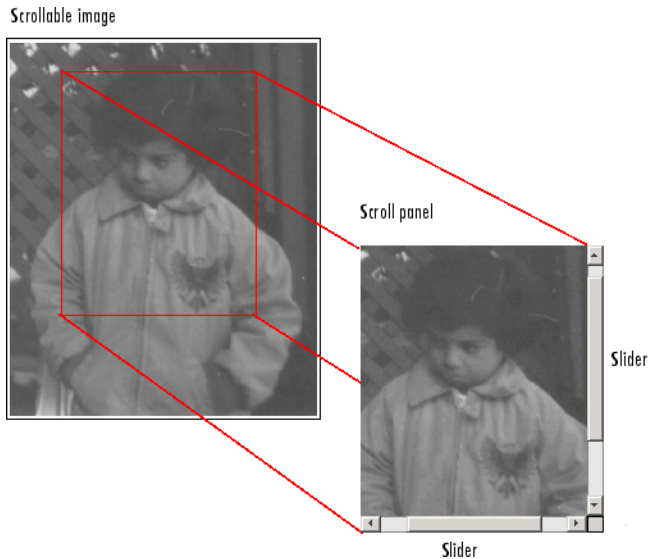
```
hfig = figure('Toolbar','none',...  
             'Menubar','none');  
himage = imshow('concordaerial.png');  
hpanel = imscrollpanel(hfig,himage);
```

The following figure shows the object hierarchy after the call to `imscrollpanel`. Note how `imscrollpanel` inserts a new object (shaded in gray) into the hierarchy between the figure object and the axes object containing the image. (To change the image data displayed in the scroll bar, use the `replaceImage` function in the `imscrollpanel` API.)



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.



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);

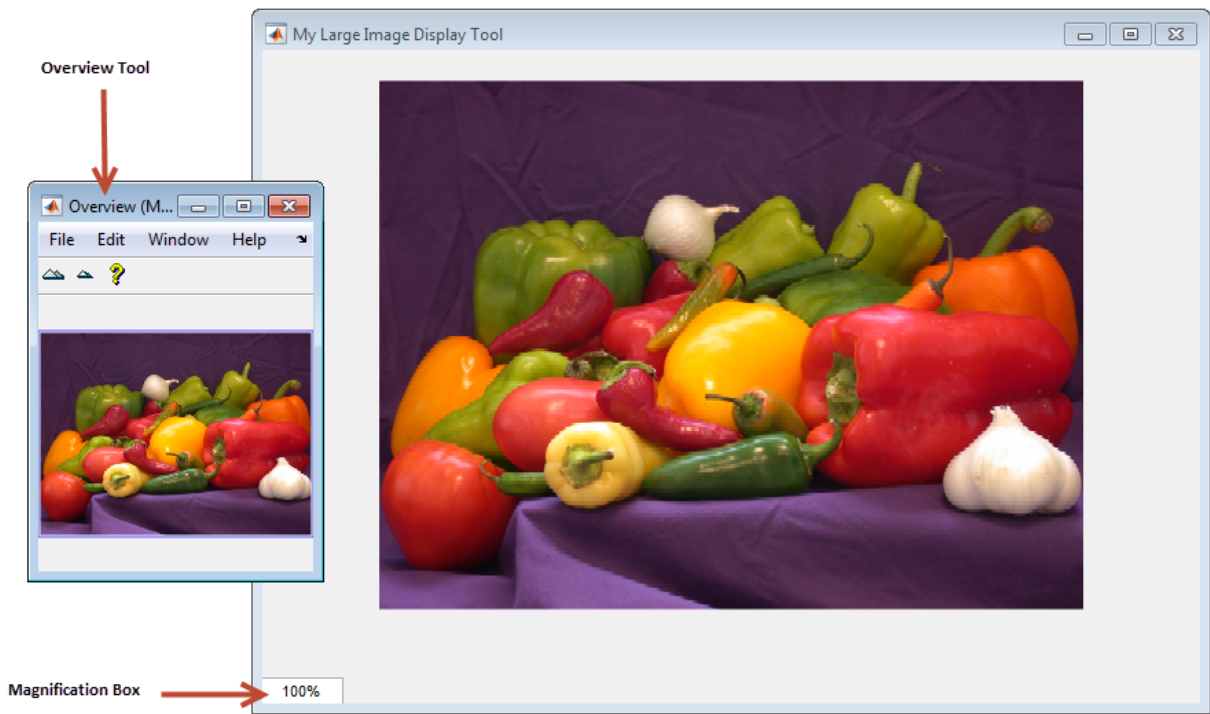
% Position the Magnification box
pos = get(hMagBox,'Position');
set(hMagBox,'Position',[0 0 pos(3) pos(4)]);

% 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.



Customizing Modular Tool Interactivity

In this section...

“Overview” on page 5-29

“Build Image Comparison Tool” on page 5-29

Overview

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 pointer.

As another example, 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 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. You can customize the connectivity of a modular tool by using the application programmer interface (API) associated with the tool to 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.

Build 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);

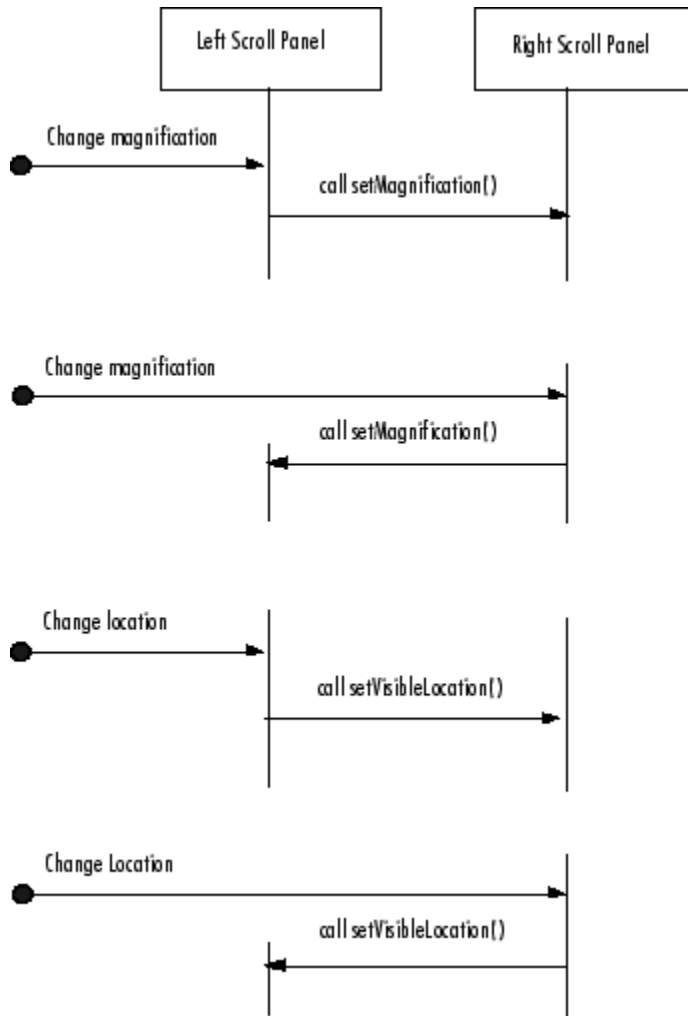
% When magnification changes on right scroll panel,
% tell left scroll panel
apiR.addNewMagnificationCallback(apiL.setMagnification);

% 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.

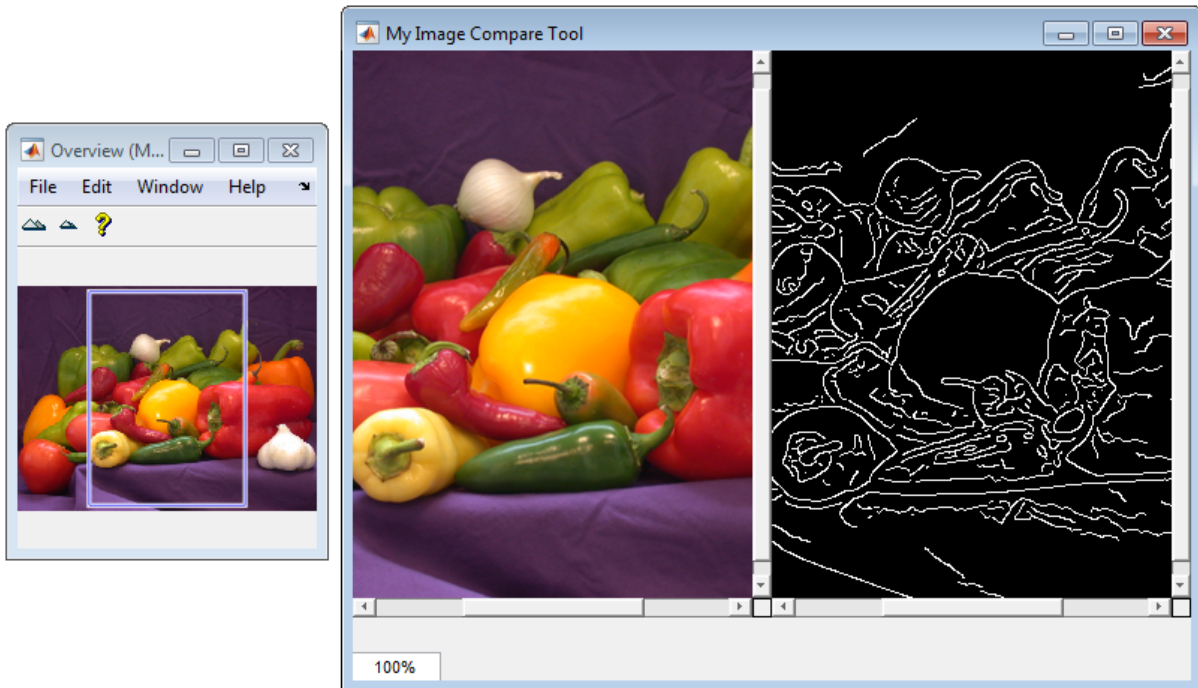


Scroll Panel Connections in Custom Image Comparison Tool

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.



Creating Your Own Modular Tools

In this section...

“Overview” on page 5-34

“Create Angle Measurement Tool” on page 5-35

Overview

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, and align figure windows in relation to a fixed window. The toolbox also provides a set of functions that you can use to define a region-of-interest of various shapes, including points, lines, rectangles, ellipses, polygons, and freehand shapes — see “Create Angle Measurement Tool” on page 5-35 for more information.

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>imellipse</code>	Create draggable, resizable ellipse
<code>imfreehand</code>	Create draggable freehand region
<code>imgca</code>	Get handle to current axes containing an image
<code>imgcf</code>	Get handle to most recent current figure containing an image
<code>imgetfile</code>	Display Open Image dialog box

Utility Function	Description
<code>imhandles</code>	Get all image handles
<code>imline</code>	Create draggable, resizable line
<code>impoint</code>	Create draggable point
<code>impoly</code>	Create draggable, resizable polygon
<code>imputfile</code>	Display Save Image dialog box
<code>imrect</code>	Create draggable, resizable rectangle
<code>iptaddcallback</code>	Add function handle to a callback list
<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

Create Angle Measurement Tool

The toolbox includes a set of functions that you can use to enable users of your image processing GUI to define a region-of-interest (ROI) on the target image. The functions implement drawing of various shapes of ROI, such as rectangles, ellipses, and polygons, and returning information about the coordinates that define the ROI. These ROI objects support methods that you can use to control aspects of its appearance and behavior.

To illustrate how to use these ROI tools, this example creates a simple angle measurement tool. This custom tool uses `impoly` to create a two-segment polyline on an image and displays the angle created by the two line segments in a title in the figure. Users of the tool can move the polyline anywhere on the image and view the angle formed by the two line segments.

```
function my_angle_measurement_tool(im)
% Create figure, setting up properties
figure('Name','My Angle Measurement Tool',...
       'NumberTitle','off',...
       'IntegerHandle','off');
% Display image in the axes
imshow(im)
% Get size of image.
m = size(im,1);
n = size(im,2);
% Get center point of image for initial positioning.
midy = ceil(m/2);
midx = ceil(n/2);
% Position first point vertically above the middle.
firstx = midx;
firsty = midy - ceil(m/4);
lastx = midx + ceil(n/4);
lasty = midy;
% Create a two-segment right-angle polyline centered in the image.
h = impoly(gca,[firstx,firsty;midx,midy;lastx,lasty],'Closed',false);
api = iptgetapi(h);
initial_position = api.getPosition()
% Display initial position
updateAngle(initial_position)
% set up callback to update angle in title.
api.addNewPositionCallback(@updateAngle);
fcn = makeConstrainToRectFcn('impoly',get(gca,'XLim'),get(gca,'YLim'));
api.setPositionConstraintFcn(fcn);
%

% Callback function that calculates the angle and updates the title.
% Function receives an array containing the current x,y position of
% the three vertices.
function updateAngle(p)
% Create two vectors from the vertices.
% v1 = [x1 - x2, y1 - y2]
% v2 = [x3 - x2, Y3 - y2]
v1 = [p(1,1)-p(2,1), p(1,2)-p(2,2)];
```

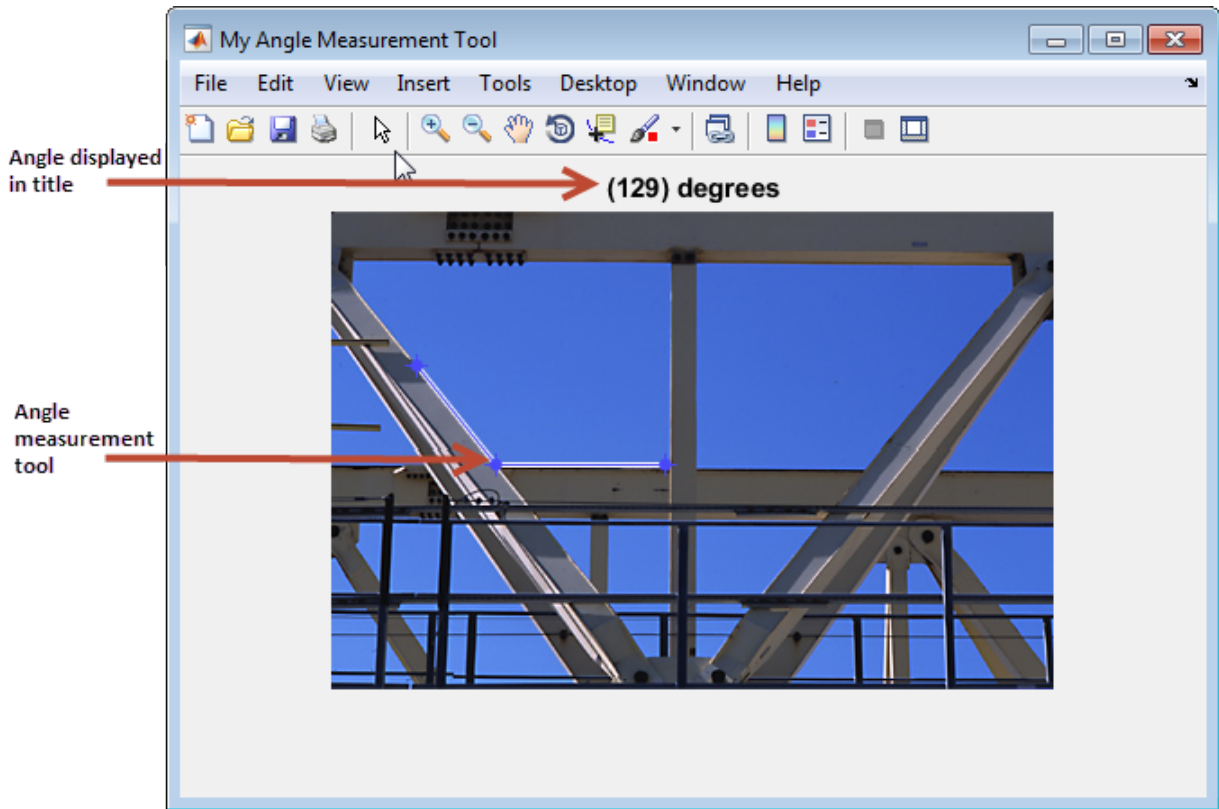


```
v2 = [p(3,1)-p(2,1), p(3,2)-p(2,2)];  
% Find the angle.  
theta = acos(dot(v1,v2)/(norm(v1)*norm(v2)));  
% Convert it to degrees.  
angle_degrees = (theta * (180/pi));  
% Display the angle in the title of the figure.  
title(sprintf('%1.0f degrees',angle_degrees))
```

To use the angle measurement tool, pass it an image.

```
I = imread('gantrycrane.png');  
my_angle_measurement_tool(I);
```

The tool opens a figure window, displaying the image with the angle measure tool centered over the image in a right angle. Move the pointer over any of the vertices of the tool to measure any angle in the image. In the following figure, the tool is measuring an angle in the image. Note the size of the angle displayed in the title of the figure.



Geometric Transformations

A geometric transformation (also known as a spatial transformation) modifies the spatial relationship between pixels in an image, mapping pixel locations in an moving image to new locations in an output image. The toolbox includes functions that perform certain specialized geometric 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 geometric transformations, including custom transformations.

- “Resize an Image” on page 6-2
- “Rotate an Image” on page 6-5
- “Crop an Image” on page 6-6
- “Translate an Image” on page 6-8
- “2-D Geometric Transformations” on page 6-11
- “Perform Simple 2-D Translation Transformation” on page 6-18
- “N-Dimensional Spatial Transformations” on page 6-22
- “Register Two Images Using Spatial Referencing to Enhance Display” on page 6-24

Resize an Image

In this section...

“Overview” on page 6-2

“Specify the Interpolation Method” on page 6-3

“Prevent Aliasing by Using Filters” on page 6-3

Overview

To resize an image, use the `imresize` function. When you resize an image, you specify the image to be resized and 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)
```

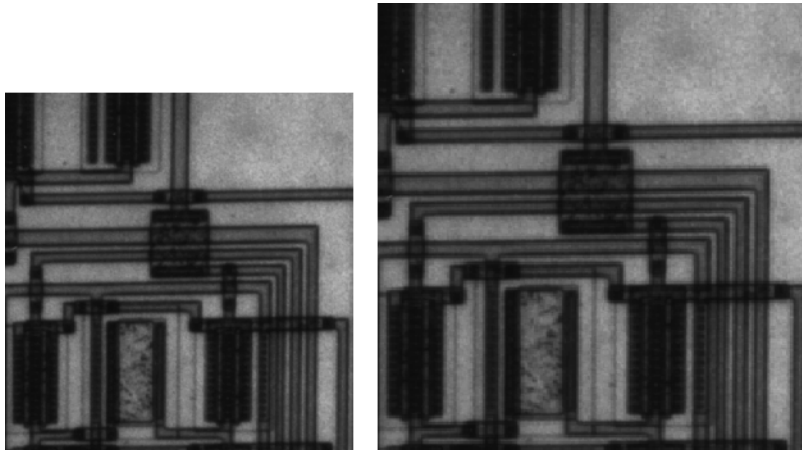


Image Courtesy of Steve Decker and Shujaat Nadeem

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. If you specify one of the elements in the vector as NaN, `imresize` calculates the value for that dimension to preserve the aspect ratio of the image.

This example creates an output image with 100 rows and 150 columns.

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

To perform the resizing required for multiresolution processing, use the `impyramid` function.

Specify the Interpolation Method

Interpolation is the process used to estimate an image value at a location in between image pixels. When `imresize` enlarges an image, the output image contains more pixels than the original image. The `imresize` function uses interpolation to determine the values for the additional pixels.

Interpolation methods determine the value for an interpolated pixel by finding the point in the input image that corresponds to a pixel in the output image and then calculating the value of 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.

By default, `imresize` uses bicubic interpolation to determine the values of pixels in the output image, but you can specify other interpolation methods and interpolation kernels. In the following example, `imresize` uses the bilinear interpolation method. See the `imresize` reference page for a complete list of interpolation methods and interpolation kernels available. You can also specify your own custom interpolation kernel.

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

Prevent Aliasing by Using Filters

When you reduce the size of an image, you lose some of the original pixels because there are fewer pixels in the output image and this can cause *aliasing*. 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.

By default, `imresize` uses antialiasing to limit the impact of aliasing on the output image for all interpolation types except nearest neighbor. To turn off antialiasing, specify the `'Antialiasing'` parameter and set the value to `false`.

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

For more information, see the reference page for `imresize`.

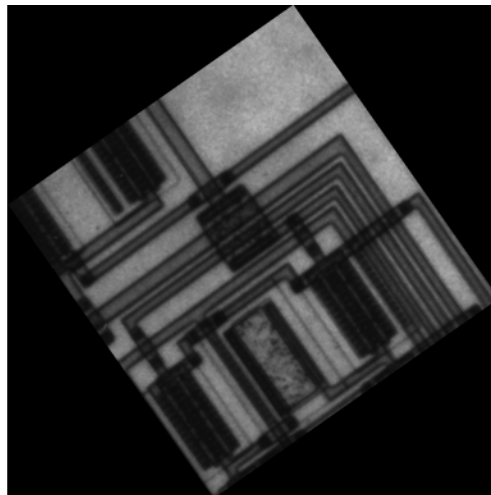
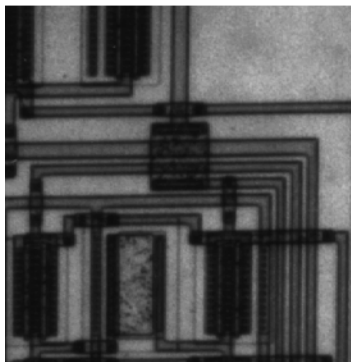
Rotate an Image

To rotate an image, use the `imrotate` function. When you rotate an image, you specify the image to be rotated and the rotation angle, in degrees. If you specify a positive rotation angle, `imrotate` rotates the image counterclockwise; if you specify a negative rotation angle, `imrotate` rotates the image clockwise.

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. You can, however, specify that the output image be the same size as the input image, using the `'crop'` argument. Similarly, `imrotate` uses nearest-neighbor interpolation by default to determine the value of pixels in the output image, but you can specify other interpolation methods. See the `imrotate` reference page for a list of supported interpolation methods.

This example rotates an image 35° counterclockwise and specifies bilinear interpolation.

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




Crop an Image

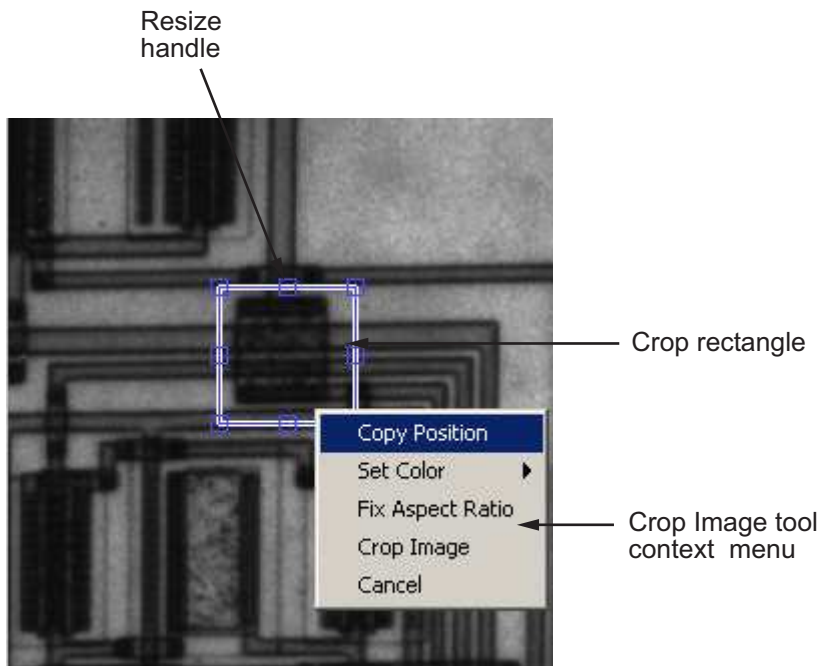
Note: You can also crop an image interactively using the Image Tool — see “Crop Image Using Image Viewer App” on page 4-54.

To extract a rectangular portion of an image, use the `imcrop` function. Using `imcrop`, you can specify the crop region interactively using the mouse or programmatically by specifying the size and position of the crop region.

This example illustrates an interactive syntax. The example reads an image into the MATLAB workspace and calls `imcrop` specifying the image as an argument. `imcrop` displays the image in a figure window and waits for you to draw the crop rectangle on the image. When you move the pointer over the image, the shape of the pointer changes

to cross hairs . Click and drag the pointer to specify the size and position of the crop rectangle. You can move and adjust the size of the crop rectangle using the mouse. When you are satisfied with the crop rectangle, double-click to perform the crop operation, or right-click inside the crop rectangle and select **Crop Image** from the context menu. `imcrop` returns the cropped image in `J`.

```
I = imread('circuit.tif')
J = imcrop(I);
```

You can also specify the size and position of the crop rectangle as parameters when you call `imcrop`. Specify the crop rectangle as a four-element position vector, `[xmin ymin width height]`.

In this example, you call `imcrop` specifying the image to crop, `I`, and the crop rectangle. `imcrop` returns the cropped image in `J`.

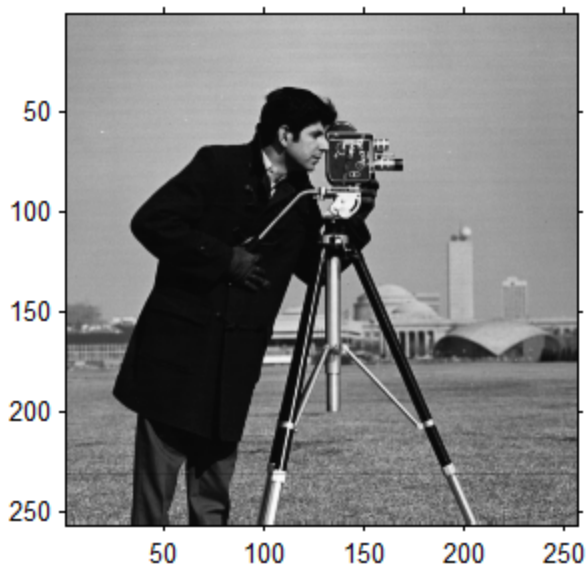
```
I = imread('circuit.tif');  
J = imcrop(I,[60 40 100 90]);
```

Translate an Image

This example shows how to perform a translation operation on an image using the `imtranslate` function. A translation operation shifts an image by a specified number of pixels in either the x or y direction, or both.

Read image. The size of the image is 256-by-256 pixels. By default, `imshow` displays the image with the corner at 0,0.

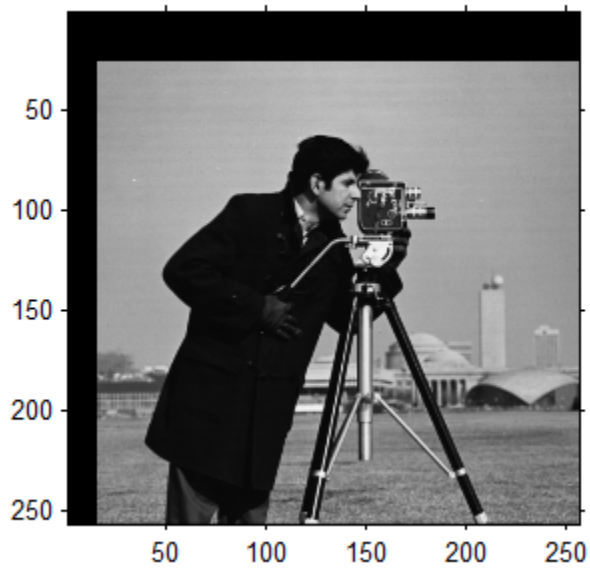
```
I = imread('cameraman.tif');  
  
imshow(I)
```



Translate the image, shifting the image by 15 pixels in the x direction and 25 pixels in the y direction. Note that, by default, `imtranslate` displays the translated image within the boundaries (or limits) of the original 256-by-256 image. This results in some of the translated image being clipped. To see the entire translated image, use the 'OutputView' parameter (shown in the following step).

```
J = imtranslate(I,[15, 25]);
```

```
figure;  
imshow(J);
```



Use the 'OutputView' parameter set to 'full' to adjust the display to show the entire translated image.

```
J = imtranslate(I,[15, 25], 'OutputView', 'full');  
figure;  
imshow(J);
```



2-D Geometric Transformations

In this section...

“Define the Geometric Transformation” on page 6-12

“Perform the Geometric Transformation” on page 6-14

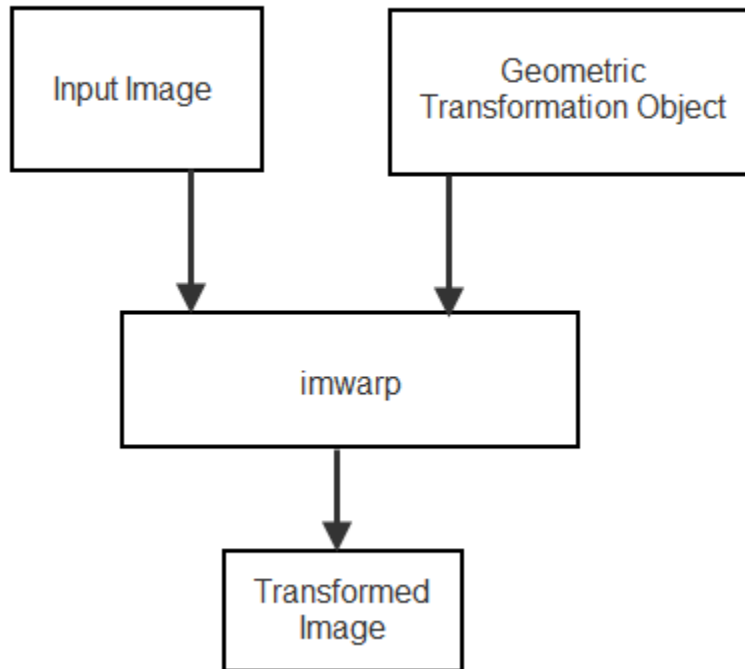
“Understand Geometric Transformation” on page 6-15

“Specify Fill Values” on page 6-15

To perform a general 2-D geometric transformation:

- 1 Define the parameters of the geometric transformation, creating a geometric transformation object. The toolbox defines several geometric transformation objects for different types of transformations, such as affine and projective.
- 2 Perform the transformation. To do this, you pass the image to be transformed and the geometric transformation object to the `imwarp` function.

The following figure illustrates this process.



Define the Geometric Transformation





Before you can perform a geometric transformation, you must first define the parameters of the transformation in a geometric transformation object. The following sections describe two ways you can do this:

- “Using a Transformation Matrix” on page 6-12
- “Using Sets of Points” on page 6-14

Using a Transformation Matrix

If you know the transformation matrix for the geometric transformation you want to perform, you can create one of the geometric transformation objects directly, passing the transformation matrix as a parameter.

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]`). The following table lists affine transformations 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.

The following example defines the transformation matrix for a translation and creates an `affine2d` geometric transformation object.

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

tform_translate = affine2d(xform);

tform_translate =
```

```
affine2d with properties:
    T: [3x3 double]
    Dimensionality: 2
```

Using Sets of Points

You can create a geometric transformation object by passing two sets of control point pairs to the `fitgeotrans` function. The `fitgeotrans` function estimates the transformation from these points and returns one of the geometric transformation objects.

Different transformations require a varying number of points. For example, affine transformations require three non-collinear points in each image (a triangle) and projective transformations require four points (a quadrilateral).

This example passes two sets of control points to `fitgeotrans`, which returns an affine geometric transformation object.

```
movingPoints = [11 11;21 11; 21 21];
fixedPoints = [51 51;61 51;61 61];

tform = fitgeotrans(movingPoints, fixedPoints, 'affine')

tform =

    affine2d with properties:
        T: [3x3 double]
        Dimensionality: 2
```

Perform the Geometric Transformation

Once you specify the transformation in a geometric transformation object, you can perform the transformation by calling the `imwarp` function, passing it the image to be transformed and a geometric transformation object. The `imwarp` function performs the specified transformation on the coordinates of the input image and creates an output image.

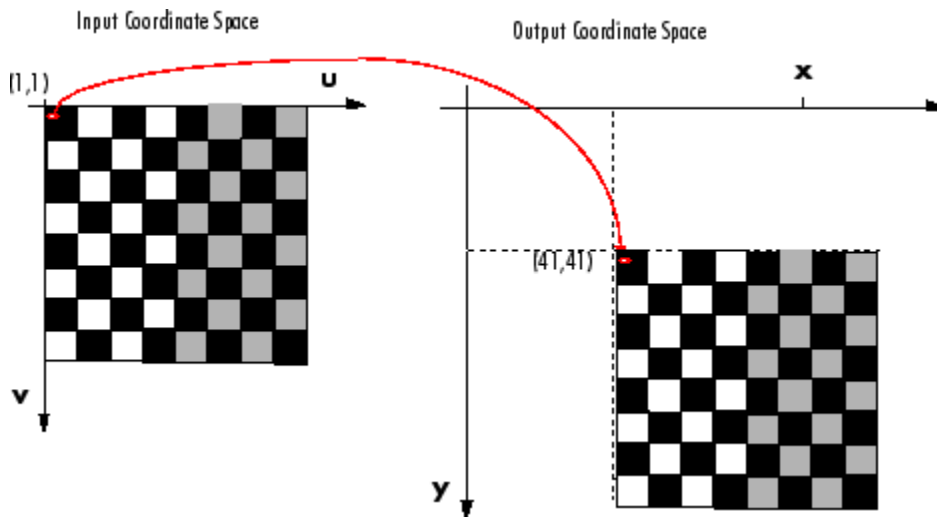
For example, to perform a transformation of the checkerboard image using the geometric transformation object created in “Define the Geometric Transformation” on page 6-12, you would call `imwarp` this way:


```
cb_trans = imwarp(cb, tform_translate);
```

Understand Geometric Transformation

`imwarp` 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). `imwarp` interpolates within the input image to compute the output pixel value. See `imwarp` for more information about supported interpolation methods.

The following figure illustrates a translation transformation. 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 `imwarp` 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. (For more information about the distinction between spatial coordinates and pixel coordinates, see “Expressing Image Locations” on page 2-3.)



Input Image Translated

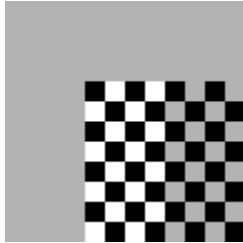
Specify Fill Values

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, `imwarp` sets these pixels to zero and they are displayed as black. Using the `FillValues` parameter with the `imwarp` function, you can specify a different color.

If the image being transformed is a grayscale image, you must specify a scalar value that specifies a shade of gray. For example, if you are translating the checkerboard image, you can specify a fill value that matches the color of the gray squares, rather than the default black color.

```
cb_fill = imwarp(cb, tform_translate, 'FillValues', .7 );  
figure, imshow(cb_fill)
```



Translated Image with Gray Fill Value

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, `imwarp` uses that shade of gray for each plane of the RGB image. If you specify a 1-by-3 vector, `imwarp` 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 = affine2d(xform);  
cb_rgb = imwarp(rgb, tform_translate, 'FillValues', [187;192;57]);  
  
figure, imshow(cb_rgb)
```



Translated RGB Image with Color Fill Value

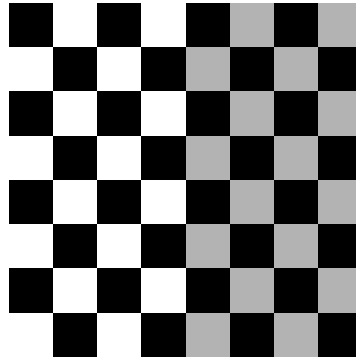
If you are transforming multiple RGB images, you can specify different fill values for each RGB image. For example, if you want to transform a series of 10 RGB images, a 4-D array with dimensions 200-by-200-by-3-by-10, you can specify a 3-by-10 array containing RGB color values for each plane.

Perform Simple 2-D Translation Transformation

This example shows how to perform 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. (You can also use the `imtranslate` function to perform translation.)

Read the image to be transformed. This example creates a checkerboard image using the checkerboard function.

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



Get spatial referencing information about the image. This information is useful when you want to display the result of the transformation.

```
Rcb = imref2d(size(cb))
```

```
Rcb =
```

```
imref2d with properties:
```

```

    XWorldLimits: [0.5000 80.5000]
    YWorldLimits: [0.5000 80.5000]
    ImageSize: [80 80]
    PixelExtentInWorldX: 1
    PixelExtentInWorldY: 1
    ImageExtentInWorldX: 80
    ImageExtentInWorldY: 80
    XIntrinsicLimits: [0.5000 80.5000]
    YIntrinsicLimits: [0.5000 80.5000]
```

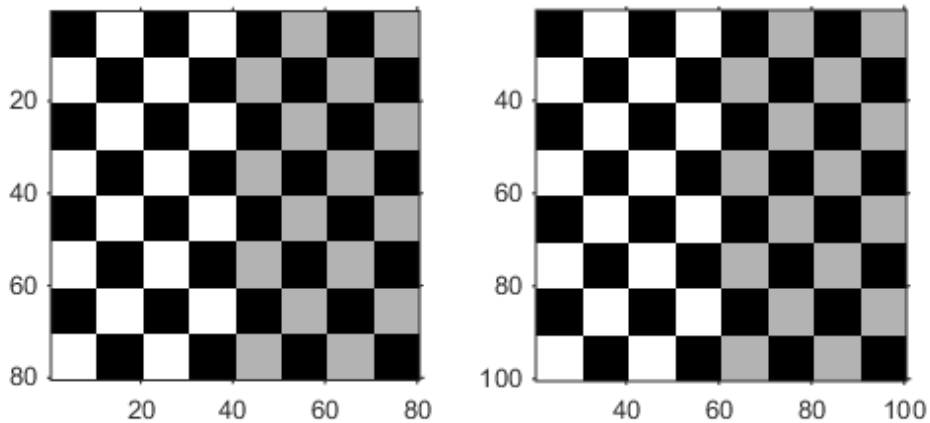
Create a geometric transformation object that defines the translation you want to perform. A translation transformation is a type of 2-D affine transformation so it uses the `affine2d` geometric transformation object. You can use a 3-by-3 transformation matrix to define the transformation. In this matrix, `A(3,1)` specifies the number of pixels to shift the image in the horizontal direction and `A(3,2)` specifies the number of pixels to shift the image in the vertical direction. Now create a geometric transformation object using the transformation matrix.

```
tform = affine2d([1 0 0; 0 1 0; 20 20 1]);
```

Perform the transformation. Call the `imwarp` function specifying the image you want to transform and the geometric transformation object. `imwarp` returns the transformed image. This example gets the optional spatial referencing object return value which contains information about the transformed image. View the original and the transformed image side-by-side using the `subplot` function in conjunction with `imshow`. When viewing the translated image, it might appear that the transformation had no effect. The transformed image looks identical to the original image. The reason that no change is apparent in the visualization is because `imwarp` sizes the output image to be just large enough to contain the entire transformed image but not the entire output coordinate space. Notice, however, that the coordinate values have been changed by the transformation.

```
[out,Rout] = imwarp(cb,tform);
figure;
subplot(1,2,1);
```

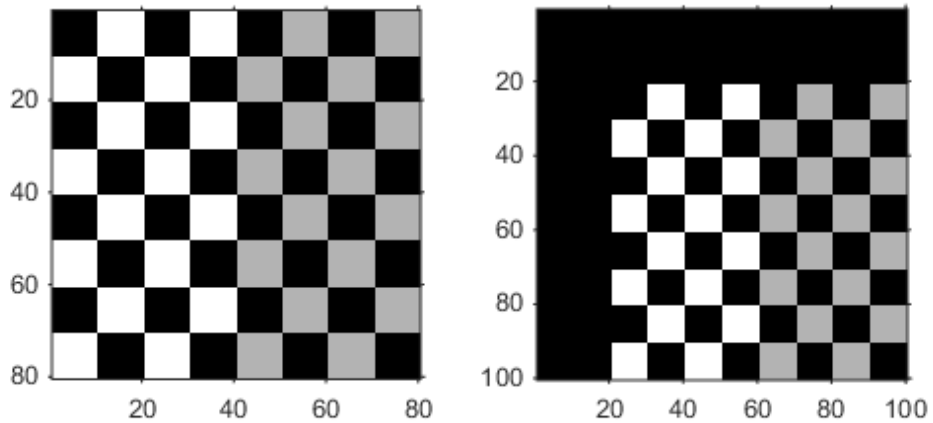
```
imshow(cb,Rcb);  
subplot(1,2,2);  
imshow(out,Rout)
```



To see the entirety of the transformed image in the same relation to the origin of the coordinate space as the original image, use `imwarp` with the `'OutputView'` parameter, specifying a spatial referencing object. The spatial referencing object specifies the size of the output image and how much of the output coordinate space is included in the output image. To do this, the example makes a copy of the spatial referencing object associated with the original image and modifies the world coordinate limits to accommodate the full size of the transformed image. The example sets the limits of the output image in world coordinates to include the origin from the input

```
Rout = Rcb;
```

```
Rout.XWorldLimits(2) = Rout.XWorldLimits(2)+20;  
Rout.YWorldLimits(2) = Rout.YWorldLimits(2)+20;  
  
[out,Rout] = imwarp(cb,tform,'OutputView',Rout);  
  
figure, subplot(1,2,1);  
imshow(cb,Rcb);  
subplot(1,2,2);  
imshow(out,Rout)
```



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, “Perform the Geometric Transformation” on page 6-14, 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.

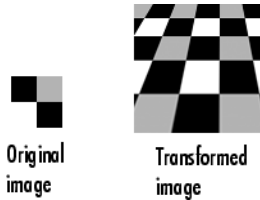
For example, if you are sampling 3-D data on a 2-D slice or manifold, the input array might have a lower dimensionality. The output dimensionality might be higher, for example, if you combine multiple 2-D transformations into a single 2-D to 3-D operation.

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.

Register Two Images Using Spatial Referencing to Enhance Display

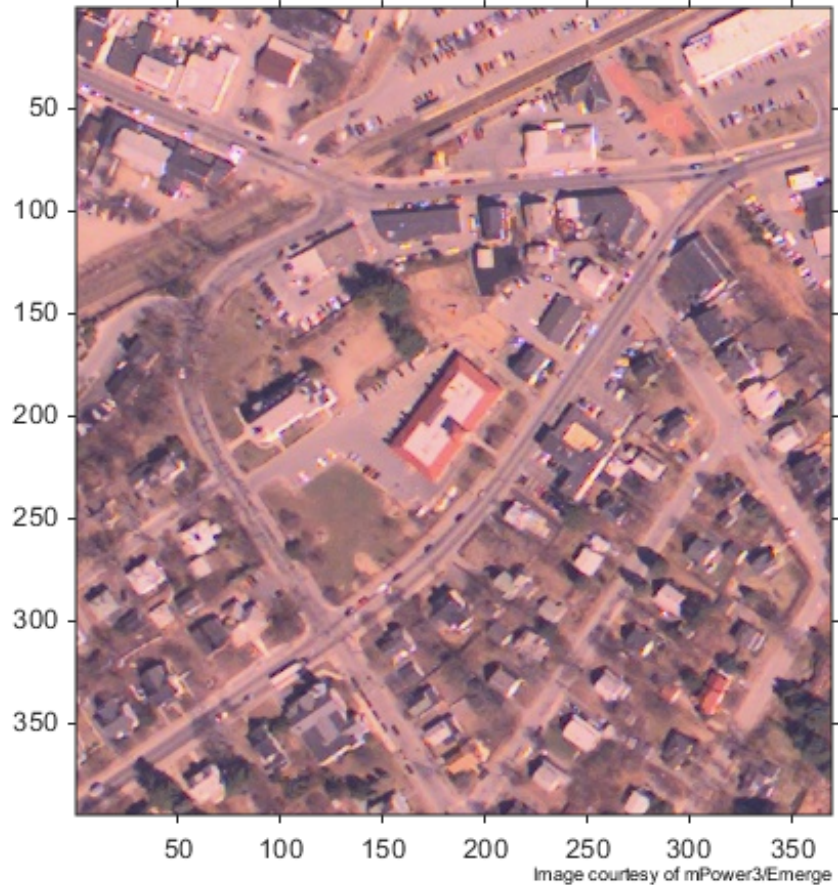
This example shows how to use spatial referencing objects to understand the spatial relationship between two images in image registration and display them effectively. This example brings one of the images, called the `moving` image, into alignment with the other image, called the `fixed` image.

Read the two images of the same scene that are slightly misaligned.

```
fixed = imread('westconcordorthophoto.png');  
moving = imread('westconcordaerial.png');
```

Display the moving (unregistered) image.

```
iptsetpref('ImshowAxesVisible','on')  
imshow(moving)  
text(size(moving,2),size(moving,1)+30, ...  
      'Image courtesy of mPower3/Emerge', ...  
      'FontSize',7,'HorizontalAlignment','right');
```

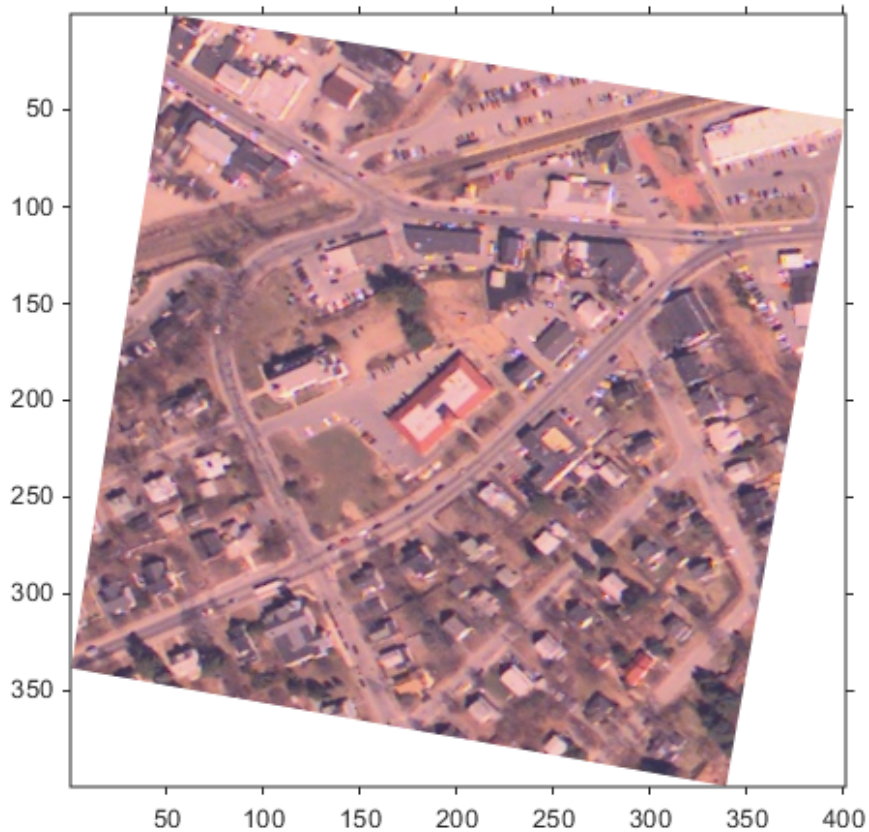


Load a MAT-file that contains preselected control points for the fixed and moving images and create a geometric transformation fit to the control points, using `fitgeotrans` .

```
load westconcordpoints  
tform = fitgeotrans(movingPoints, fixedPoints, 'projective');
```

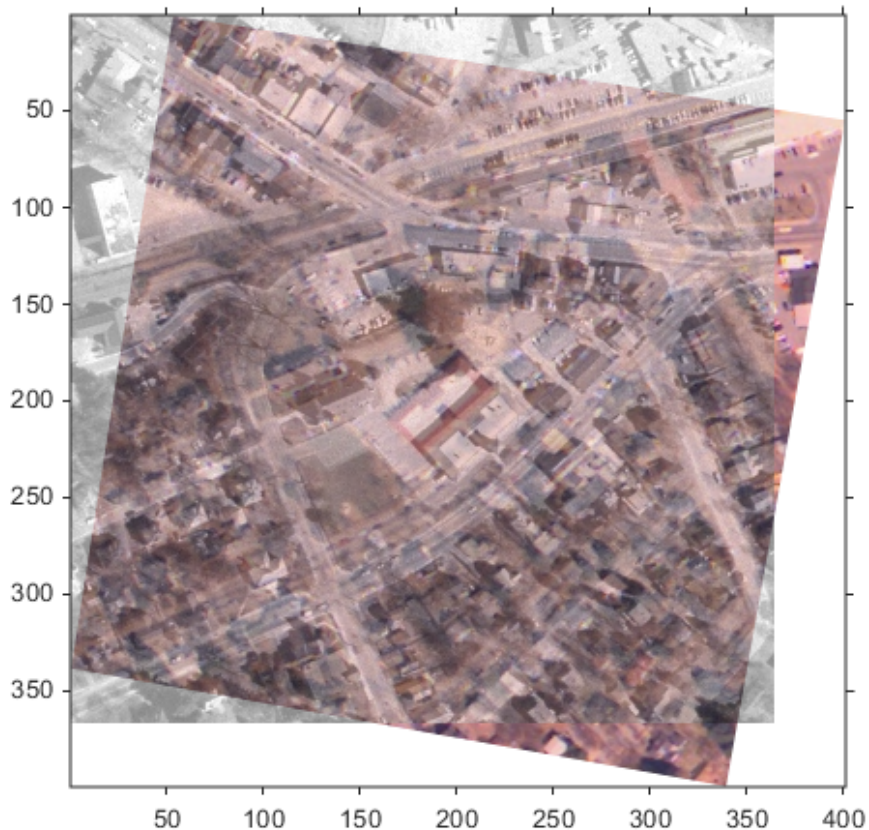
Perform the transformation necessary to register the `moving` image with the `fixed` image, using `imwarp`. This example uses the optional `'FillValues'` parameter to specify a fill value (white), which will help when displaying the `fixed` image over the transformed `moving` image, to check registration. Notice that the full content of the geometrically transformed `moving` image is present, now called `registered`. Also note that there are no blank rows or columns.

```
registered = imwarp(moving, tform, 'FillValues', 255);  
figure, imshow(registered);
```



Overlay the transformed image, `registered`, over the `fixed` image, using `imshowpair`. Notice how the two images appear misregistered. This happens because `imshowpair` assumes that the images are both in the default intrinsic coordinate system. The next steps provide two ways to remedy this display problem.

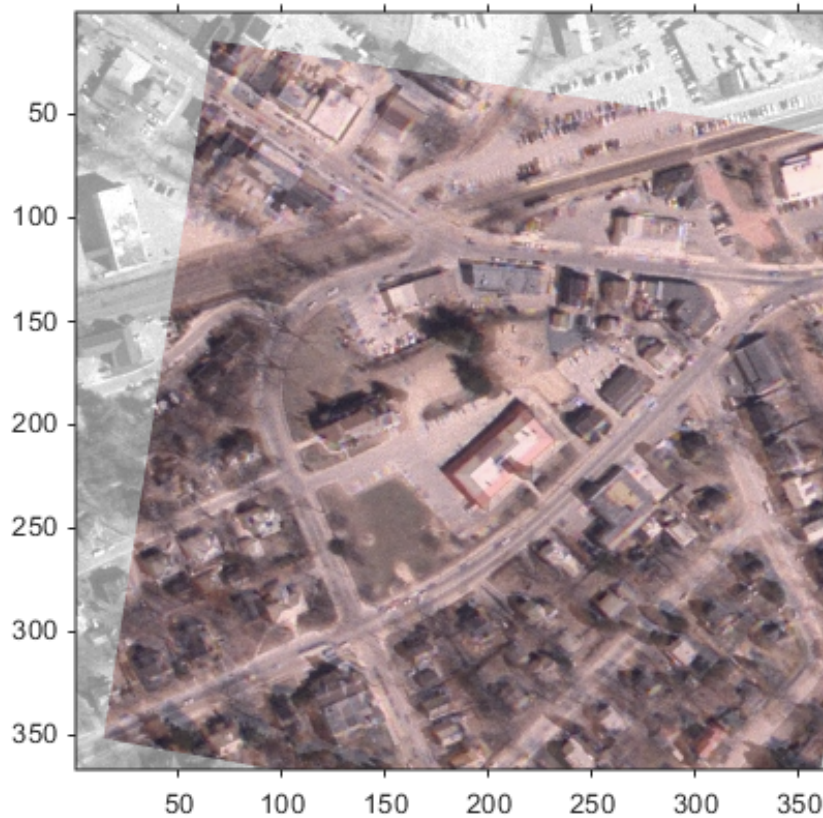
```
figure, imshowpair(fixed,registered,'blend');
```



Constrain the transformed image, `registered`, to the same number of rows and columns, and the same spatial limits as the `fixed` image. This ensures that the registered image appears registered with the fixed image but areas of the registered

image that would extrapolate beyond the extent of the fixed image are discarded. To do this, create a default spatial referencing object that specifies the size and location of the fixed image, and use `imwarp`'s `'OutputView'` parameter to create a constrained resampled image `registered1`. Display the registered image over the fixed image. In this view, the images appear to have been registered, but not all of the unregistered image is visible.

```
Rfixed = imref2d(size(fixed));  
registered1 = imwarp(moving,tform,'FillValues', 255,'OutputView',Rfixed);  
figure, imshowpair(fixed,registered1,'blend');
```



As an alternative, use the optional `imwarp` syntax that returns the output spatial referencing object that indicates the position of the full transformed image in the same default intrinsic coordinate system as the `fixed` image. Display the registered image over the fixed image and note that now the full registered image is visible.

```
[registered2, Rregistered] = imwarp(moving, tform, 'FillValues', 255);  
figure, imshowpair(fixed, Rfixed, registered2, Rregistered, 'blend');
```

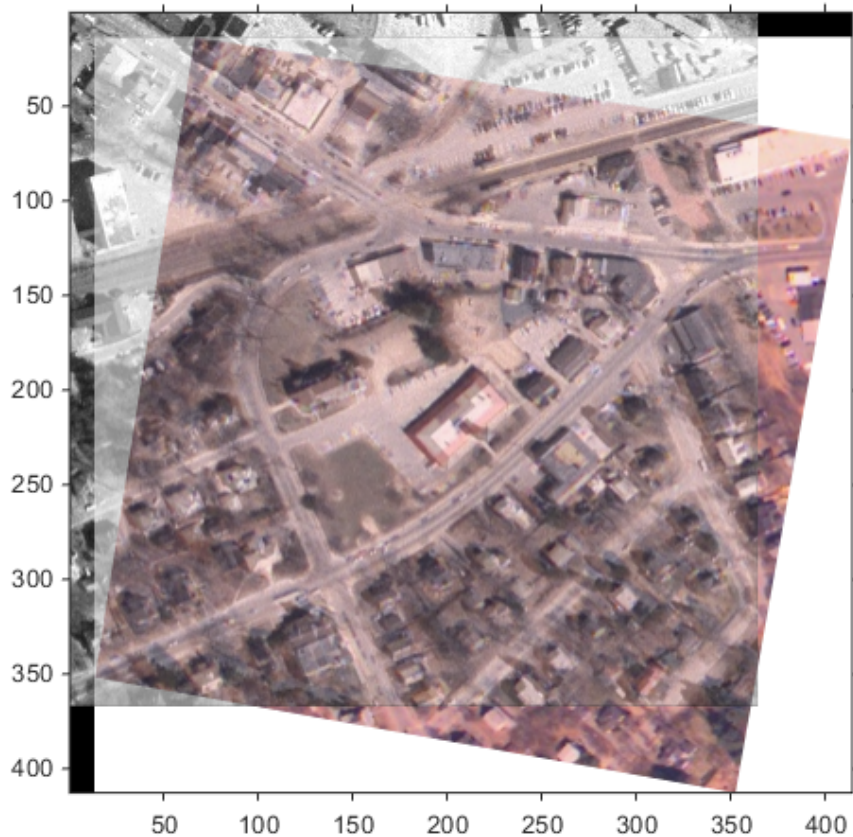


Image Registration

This chapter describes the image registration capabilities of the Image Processing Toolbox software. 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.

- “Image Registration Techniques” on page 7-2
- “Control Point Registration” on page 7-3
- “Using `cpselect` in a Script” on page 7-5
- “Register an Aerial Photograph to a Digital Orthophoto” on page 7-6
- “Geometric Transformation Types” on page 7-11
- “Control Point Selection Procedure” on page 7-12
- “Start the Control Point Selection Tool” on page 7-14
- “Find Visual Elements Common to Both Images” on page 7-16
- “Select Matching Control Point Pairs” on page 7-20
- “Export Control Points to the Workspace” on page 7-27
- “Save Control Point Selection Session” on page 7-28
- “Use Cross-Correlation to Improve Control Point Placement” on page 7-29
- “Use Phase Correlation as Preprocessing Step in Registration” on page 7-30
- “Intensity-Based Automatic Image Registration” on page 7-36
- “Registering Multimodal MRI Images” on page 7-39

Image Registration Techniques

Image registration is the process of aligning two or more images of the same scene. This process involves designating one image as the reference (also called the reference image or the fixed image), and applying geometric transformations to the other images so that they align with the reference. Images can be misaligned for a variety of reasons. Commonly, the images are captured under variable conditions that can change camera perspective. Misalignment can also be the result of lens and sensor distortions or differences between capture devices.

A geometric transformation maps locations in one image to new locations in another image. The step of determining the correct geometric transformation parameters 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 or to align medical images captured with different diagnostic modalities (MRI and SPECT). Image registration allows you to compare common features in different images. For example, you might discover how a river has migrated, how an area became flooded, or whether a tumor is visible in an MRI or SPECT image.

Together, the Image Processing Toolbox and Computer Vision System Toolbox™ offer three image registration solutions:

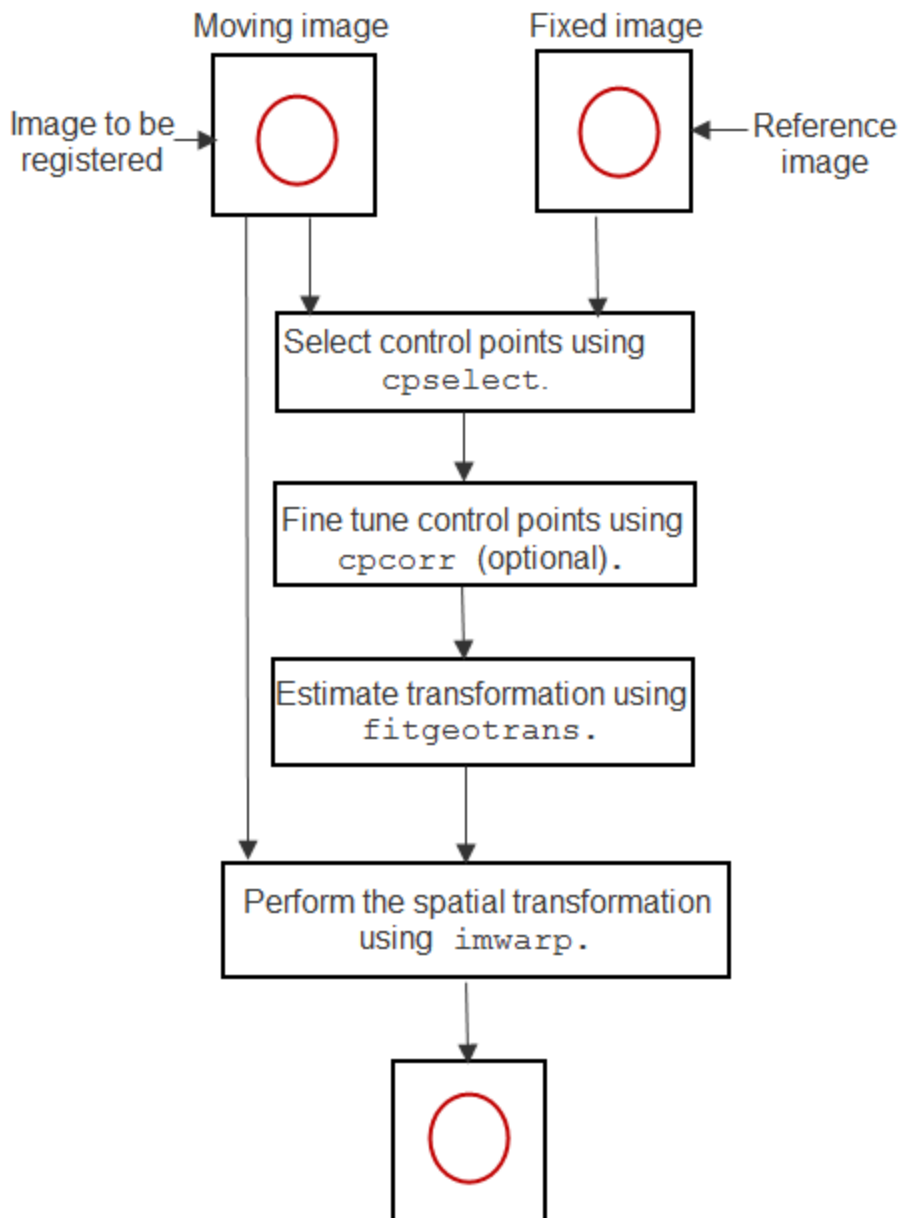
- “Intensity-Based Automatic Image Registration” on page 7-36 maps certain pixels in each image to the same location based on relative intensity patterns. This approach is best suited for workflows that involve a large collection of images or when you require an automated workflow. This functionality resides in the Image Processing Toolbox.
- “Control Point Registration” on page 7-3 allows you to manually select common features in each image to map to the same pixel location. This method of registration is best suited for images that have distinct features. It resides in the Image Processing Toolbox.
- An automated feature-based workflow automatically aligns images by selecting matching features between two images. This workflow includes feature detection, extraction, and matching, followed by transform estimation. Features can be corners or blobs and the distortion can include rotation and scale changes. For more information, see Finding Rotation and Scale of an Image Using Automated Feature Matching. You must have the Computer Vision System Toolbox installed to use this method.

Control Point Registration

The Image Processing Toolbox software 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 geometric 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 “Register an Aerial Photograph to a Digital Orthophoto” on page 7-6 for an extended example.



Using `cpselect` in a Script

If you need to perform the same kind of registration for many images, you can automate the process by putting all the steps in a script. For example, you could create a script that launches the Control Point Selection Tool with the images to be registered. The script could then use the control points selected to create a geometric transformation object and pass this object and the image to be registered to the `imwarp` function, outputting the registered image.

When calling `cpselect` in a script, specify the `'Wait'` option. The `'Wait'` option causes `cpselect` to block the MATLAB command line until control points have been selected and returned. If you do not use the `'Wait'` option, `cpselect` returns control immediately and your script will continue without allowing time for control point selection. In addition, without the `'Wait'` option, `cpselect` does not return the control points as return values. For an example, see the `cpselect` reference page.

Register an Aerial Photograph to a Digital Orthophoto

This example shows how to use point mapping to perform image registration.

Read the sample images and display them.

```
orthophoto = imread('westconcordorthophoto.png');  
figure, imshow(orthophoto)  
unregistered = imread('westconcordaerial.png');  
figure, imshow(unregistered)
```



Aerial Photo Image

Image Courtesy of mPower3/Emerge



Orthophoto Image

Image Courtesy of MassGIS

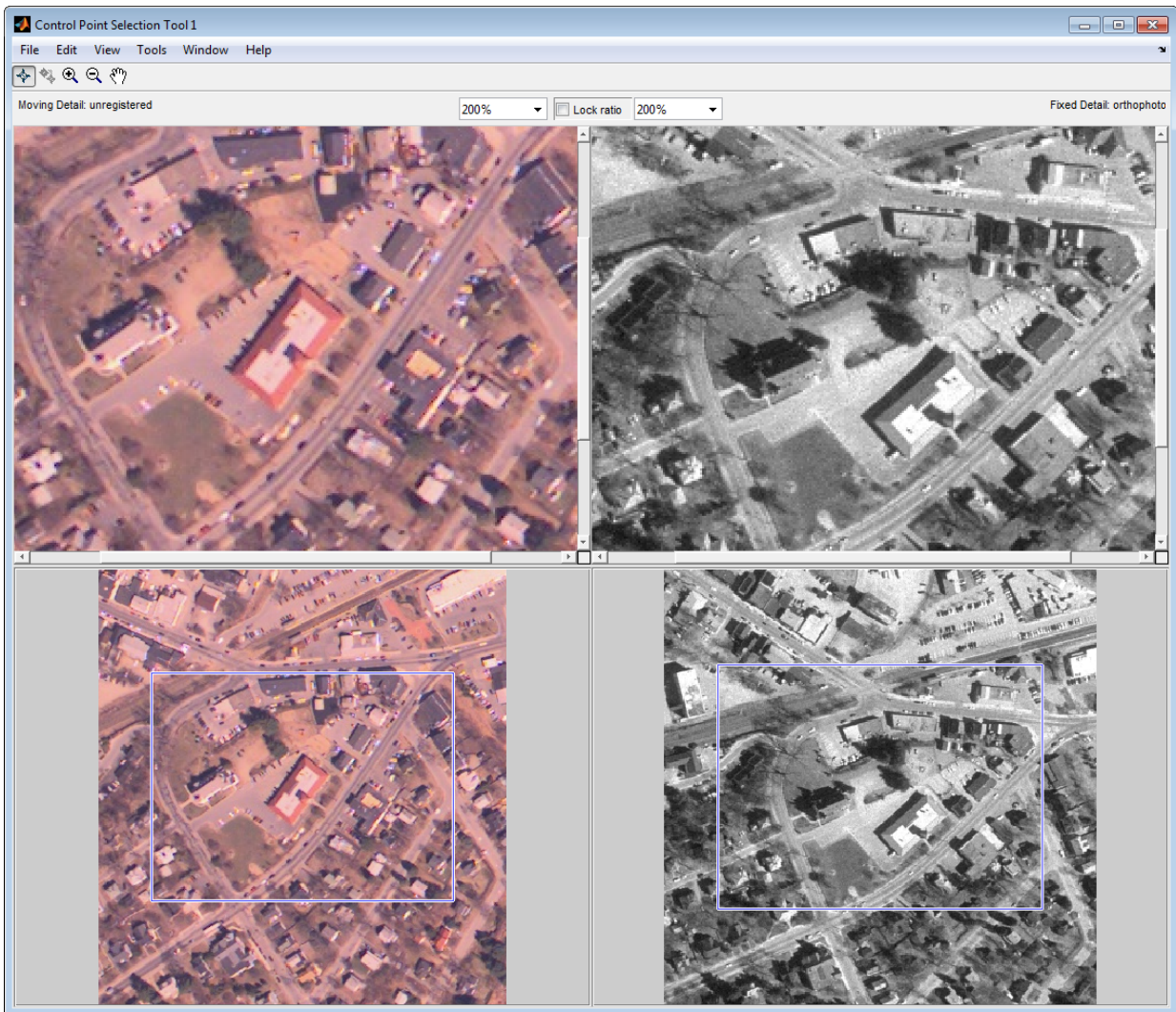
In this example, the fixed image is `westconcordorthophoto.png`, the MassGIS georegistered orthophoto. It is a panchromatic (grayscale) image, supplied by the Massachusetts Geographic Information System (MassGIS), that has been orthorectified to remove camera, perspective, and relief distortions (via a specialized image transformation process). The orthophoto 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. In the orthophoto, each pixel center corresponds to a definite geographic location, and every pixel is 1 meter square in map units.

The image to be registered is `westconcordaerial.png`, a digital aerial photograph supplied by mPower3/Emerge, and is a visible-color RGB image. The aerial image is geometrically uncorrected: it includes camera perspective, terrain and building relief, internal (lens) distortions, and it does not have any particular alignment or registration with respect to the earth.

The example reads both images into the MATLAB workspace but the `cpselect` 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.

Select pairs of corresponding control points in both images, using the Control Point Selection tool. 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`, specifying as arguments the moving and fixed images.

```
cpselect(unregistered, orthophoto)
```



Save the control point pairs to the MATLAB workspace. In the Control Point Selection Tool, click the **File** menu and choose the **Export Points to Workspace** option.

For example, the following set of control points in the moving image represent spatial coordinates; the left column lists x -coordinates, the right column lists y -coordinates.


```

moving_points =

    118.0000    96.0000
    304.0000    87.0000
    358.0000   281.0000
    127.0000   292.0000

```

You can optionally fine-tune the control point pair placement, using the `cpcorr` function.

This is an optional step that uses cross-correlation to adjust the position of the control points you selected with `cpselect`. To use cross-correlation, features in the two images must be at the same scale and have the same orientation. They cannot be rotated relative to each other. Because the Concord image is rotated in relation to the fixed image, `cpcorr` cannot tune the control points.

Specify the type of transformation and infer its parameters, using `fitgeotrans`. `fitgeotrans` is a data-fitting function that determines the transformation needed to bring the image into alignment, based on the geometric relationship of the control points. `fitgeotrans` returns the parameters in a geometric transformation object.

```
mytform = fitgeotrans(movingPoints, fixedPoints, 'projective');
```

```
mytform =
```

```

    projective2d with properties:
        T: [3x3 double]
        Dimensionality: 2

```

When you use `fitgeotrans`, you must specify the type of transformation you want to perform. The `fitgeotrans` function can infer the parameters for several types of transformations. You must choose which transformation will correct the type of distortion present in the moving image. Images can contain more than one type of distortion.

The predominant distortion in the aerial image of West Concord (the moving image) results from the camera perspective. Ignoring terrain relief, which is minor in this area, image registration can correct for camera perspective distortion by using a projective transformation. The projective transformation also rotates the image into alignment with the map coordinate system underlying the fixed digital orthophoto image.

Transform the moving image (`unregistered`) to bring it into alignment with the fixed image. You use `imwarp` to perform the transformation, passing it the moving image and

the geometric transformation object returned by `fitgeotrans`. `imwarp` returns the transformed image.

```
registered = imwarp(unregistered, mytform);
```

The following figure shows the transformed image transparently overlaid on the fixed image to show the results of the registration.



Geometric Transformation Types

The Image Processing Toolbox provides functionality for applying geometric transformations to register images.

For control point registration, the `fitgeotrans` function can infer the parameters for the following types of transformations, listed in order of complexity.

- `'nonreflective similarity'`
- `'affine'`
- `'projective'`
- `'polynomial'` (Order 2, 3, or 4)
- `'piecewise linear'`
- `'lwm'`

The first four transformations, `'nonreflective similarity'`, `'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.

Your choice of transformation type affects the number of control point pairs you need to select. For example, a nonreflective similarity transformation requires at least two control point pairs. A polynomial transformation of order 4 requires 15 control point pairs. For more information about these transformation types, and the special syntaxes they require, see `cpselect`.

For information about the specific transformation types available for intensity-based automatic image registration, see the `imregister` function reference page.

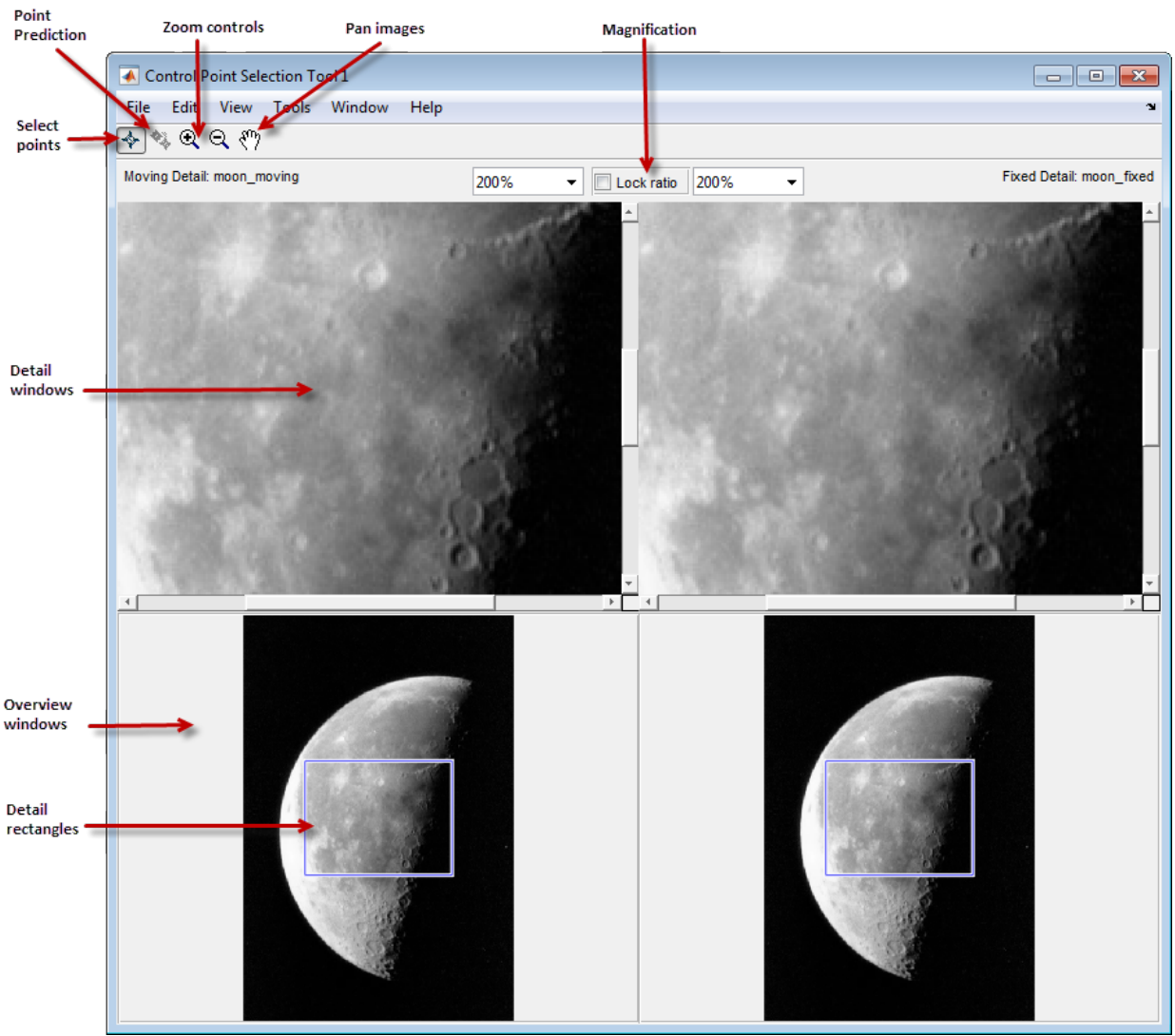
Control Point Selection Procedure

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 *moving* image, next to the image you want to compare it to, called the *fixed* image.

Specifying control points is a four-step process:

- 1** Start the tool, specifying the moving image and the fixed image.
- 2** Use navigation aids to explore the image, 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 moving image and the fixed 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.



Start 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 moving image) and the image you want to compare it to (the fixed image).

For simplicity, this section uses the same image as the moving and the fixed image. To walk through an example of an actual registration, see “Register an Aerial Photograph to a Digital Orthophoto” on page 7-6.

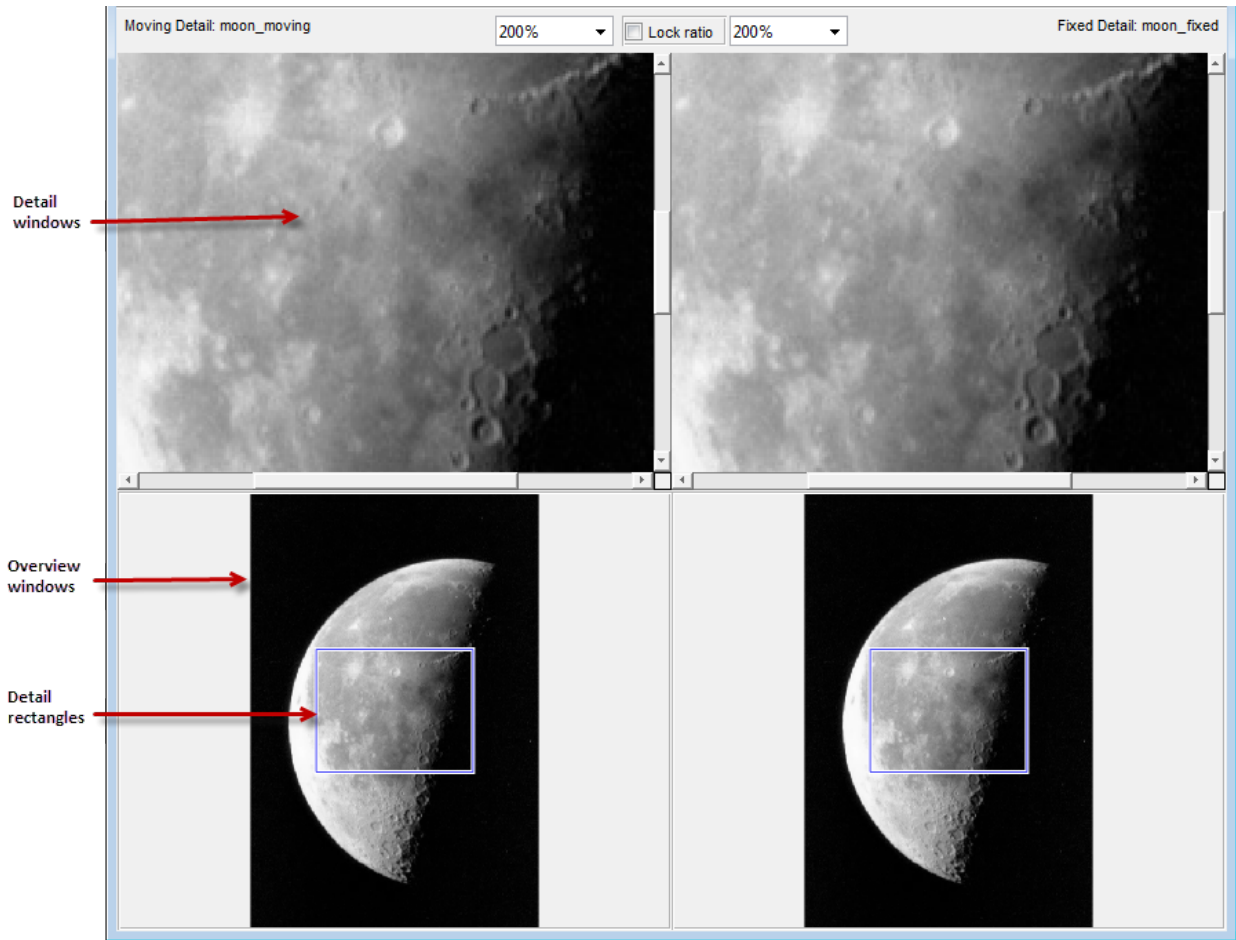
```
moon_fixed = imread('moon.tif');  
moon_moving = moon_fixed;  
cpselect(moon_moving, moon_fixed);
```

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 “Export Control Points to the Workspace” on page 7-27. For complete details, see the `cpselect` reference page.

When the Control Point Selection Tool starts, it contains three primary components:

- **Details windows**—The two windows displayed at the top of the tool are called the Detail windows. These windows show a close-up view of a portion of the images you are working with. The moving image is on the left and the fixed image is on the right.
- **Overview windows**—The two windows displayed at the bottom of the tool are called the Overview windows. These windows show the images in their entirety, at the largest scale that fits the window. The moving image is on the left and the fixed image is on the right. You can control whether the Overview window appears by using the **View** menu.
- **Details rectangle**—Superimposed on the images displayed in the two Overview windows is a rectangle, called the Detail rectangle. This rectangle controls 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. You can move the Detail rectangle to change the portion of the image displayed in the Detail windows.

The following figure shows these components of the Control Point Selection Tool.



Find Visual Elements Common to Both Images

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:

In this section...
“Using Scroll Bars to View Other Parts of an Image” on page 7-16
“Using the Detail Rectangle to Change the View” on page 7-16
“Panning the Image Displayed in the Detail Window” on page 7-17
“Zooming In and Out on an Image” on page 7-17
“Specifying the Magnification of the Images” on page 7-18
“Locking the Relative Magnification of the Moving and Fixed Images” on page 7-19

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 for 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 Move the pointer into the Detail rectangle. The cursor changes to the fleur shape, .
- 2 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 **Pan** button  in the Control Point Selection Tool toolbar or select **Pan** from the Tools menu.
- 2 Move the pointer over the image in the Detail window. The cursor changes to the hand shape, .
- 3 Press and hold the mouse button. The cursor changes to a closed fist shape, . Use the mouse to move the image in the Detail window.

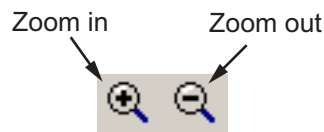
Note As you move the image in the Detail window, the Detail rectangle in the Overview window moves.


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, you can zoom in or zoom out on the images displayed. (You can also zoom in or out on an image by changing the magnification. See “Specifying the Magnification of the Images” on page 7-18 for more information.)

To zoom in or zoom out on the fixed or moving images,

- 1 Click the appropriate magnifying glass button on the Control Point Selection Tool toolbar or select **Zoom In** or **Zoom Out** from the Tools menu.



- 2 Move the pointer over the image in the Detail window that you want to zoom in or out on. The cursor changes to the appropriate magnifying glass shape, such as . 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 7-18 for a list of some of these magnifications.) `cpselect` centers the new view of the image on the spot where you clicked.

Another way to use the Zoom tool to zoom in on an image is to 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 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.

The size of the Detail rectangle in the Overview window changes as you zoom in or out on the image in the Detail window.

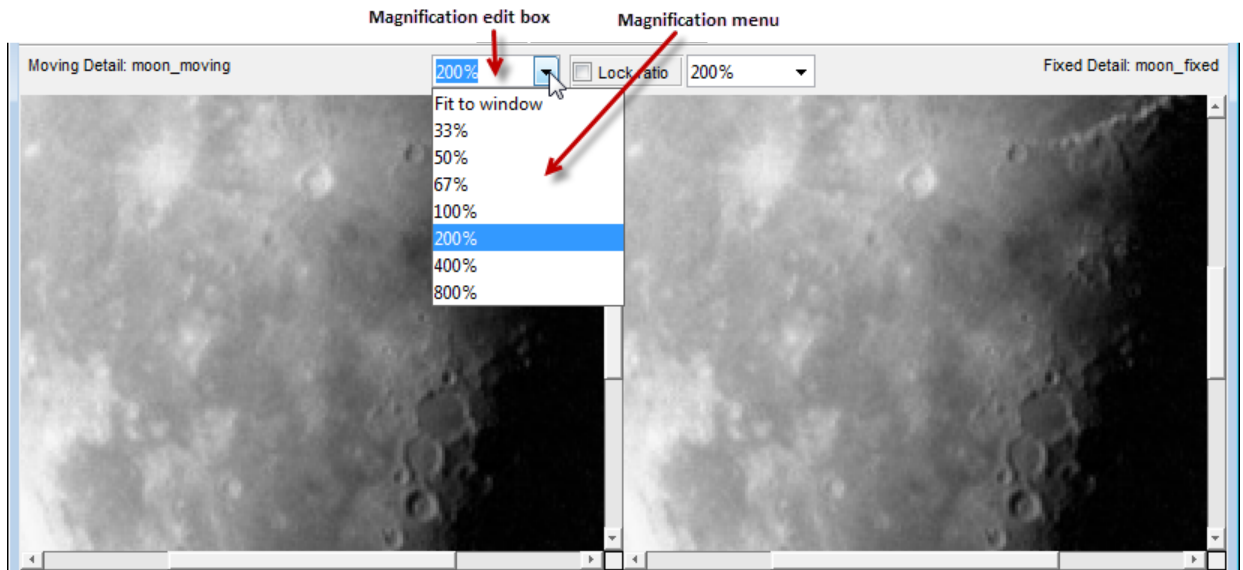
To keep the relative magnifications of the fixed and moving images synchronized as you zoom in or out, click the **Lock ratio** check box. See “Locking the Relative Magnification of the Moving and Fixed Images” on page 7-19 for more information.

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 7-17 for more information.)

To change the magnification of an image,

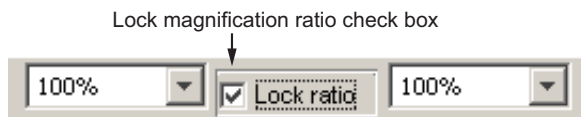
- 1 Move the cursor into the magnification edit box of the window you want to change. The cursor changes to the text entry cursor.
- 2 Type a new value in the magnification edit box and press **Enter**, or click the menu associated with the edit box and choose from a list of preset magnifications. `cpselect` changes the magnification of the image and displays the new view in the appropriate window. To keep the relative magnifications of the fixed and moving images synchronized as you change the magnification, click the **Lock ratio** check box. See “Locking the Relative Magnification of the Moving and Fixed Images” on page 7-19 for more information.



Locking the Relative Magnification of the Moving and Fixed Images

To keep the relative magnification of the moving and fixed images automatically synchronized in the Detail windows, click the **Lock Ratio** check box.

When the **Lock Ratio** check box is selected, *cpselect* changes the magnification of *both* the moving and fixed images when you zoom in or out on either one of the images or specify a magnification value for either of the images.



Select 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 moving image) and the image to which you are comparing it (the fixed image). When you start `cpselect`, point selection is enabled, by default.

You specify control points by pointing and clicking in the moving and fixed images, in either the Detail or the Overview windows. Each point you specify in the moving image must have a match in the fixed image. The following sections describe the ways you can use the Control Point Selection Tool to choose control point pairs:

In this section...

“Picking Control Point Pairs Manually” on page 7-20




“Using Control Point Prediction” on page 7-22

“Moving Control Points” on page 7-25

“Deleting Control Points” on page 7-26

Picking Control Point Pairs Manually

To specify a pair of control points in your images,

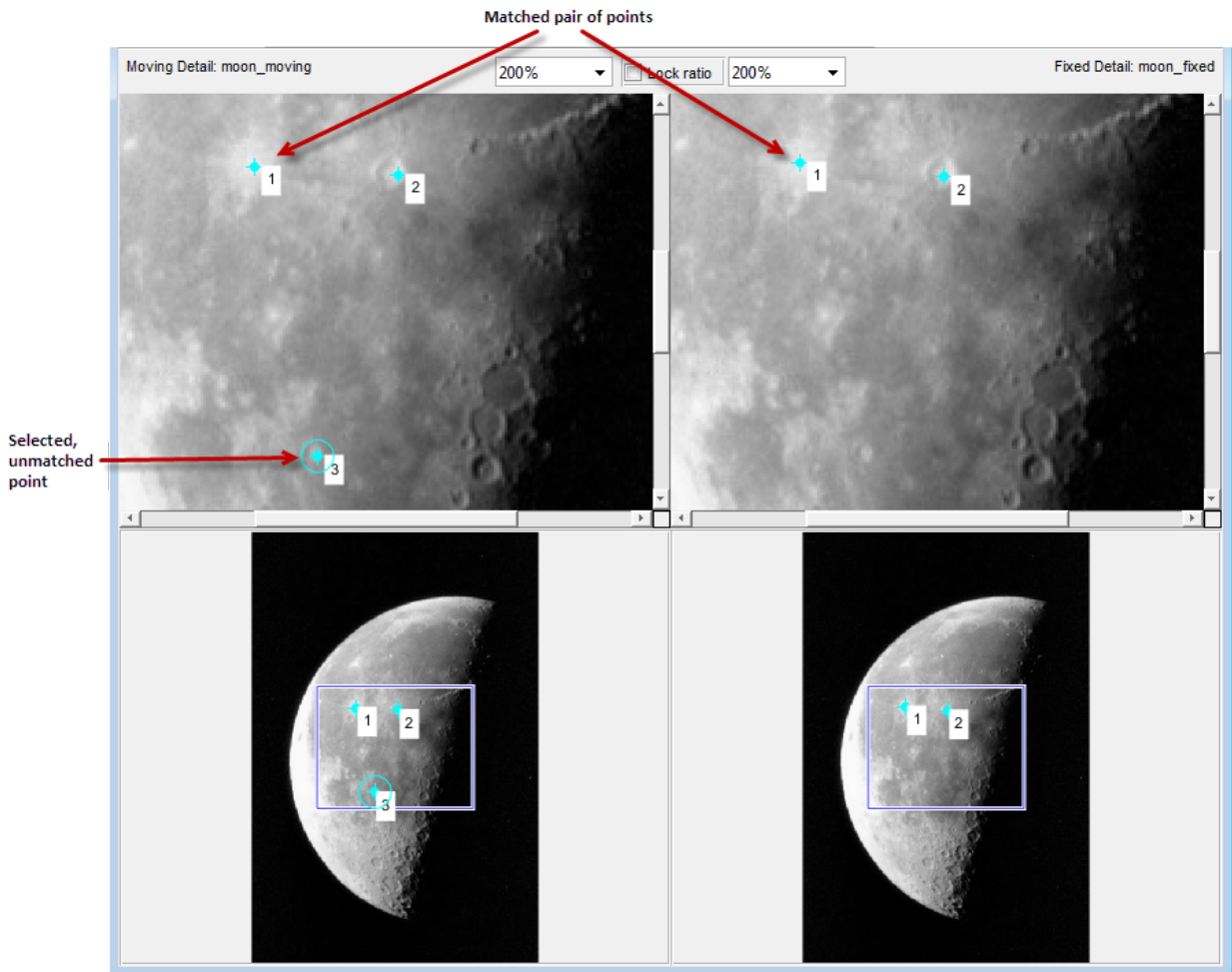
- 1 Click the **Control Point Selection** button  in the Control Point Selection Tool toolbar or select **Add Points** from the Tools menu. (Control point selection mode is active by default.) The cursor changes to a crosshairs shape .
- 2 Position the cursor over a feature you have visually selected in any of the images displayed and click the mouse button. `cpselect` places a control point symbol, , at the position you specified, in both the Detail window and the corresponding Overview window. `cpselect` numbers the points as you select them. The appearance of the control point symbol indicates its current state. The circle around the point indicates that it is the currently selected point. The number identifies control point pairs.

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.

- 3** You can select another point in the same image or you can move to the corresponding image and create a match for the point. To create the match for this control point, position the cursor over the same feature in the corresponding Detail or Overview window and click the mouse button. `cpselect` places a control point symbol at the position you specified, in both the Detail and Overview windows. You can work in either direction: picking control points in either of the Detail windows, moving or fixed, or in either of the Overview windows, moving or fixed.

To match an unmatched control point, select it, and then pick a point in the corresponding window. You can move or delete control points after you create them.

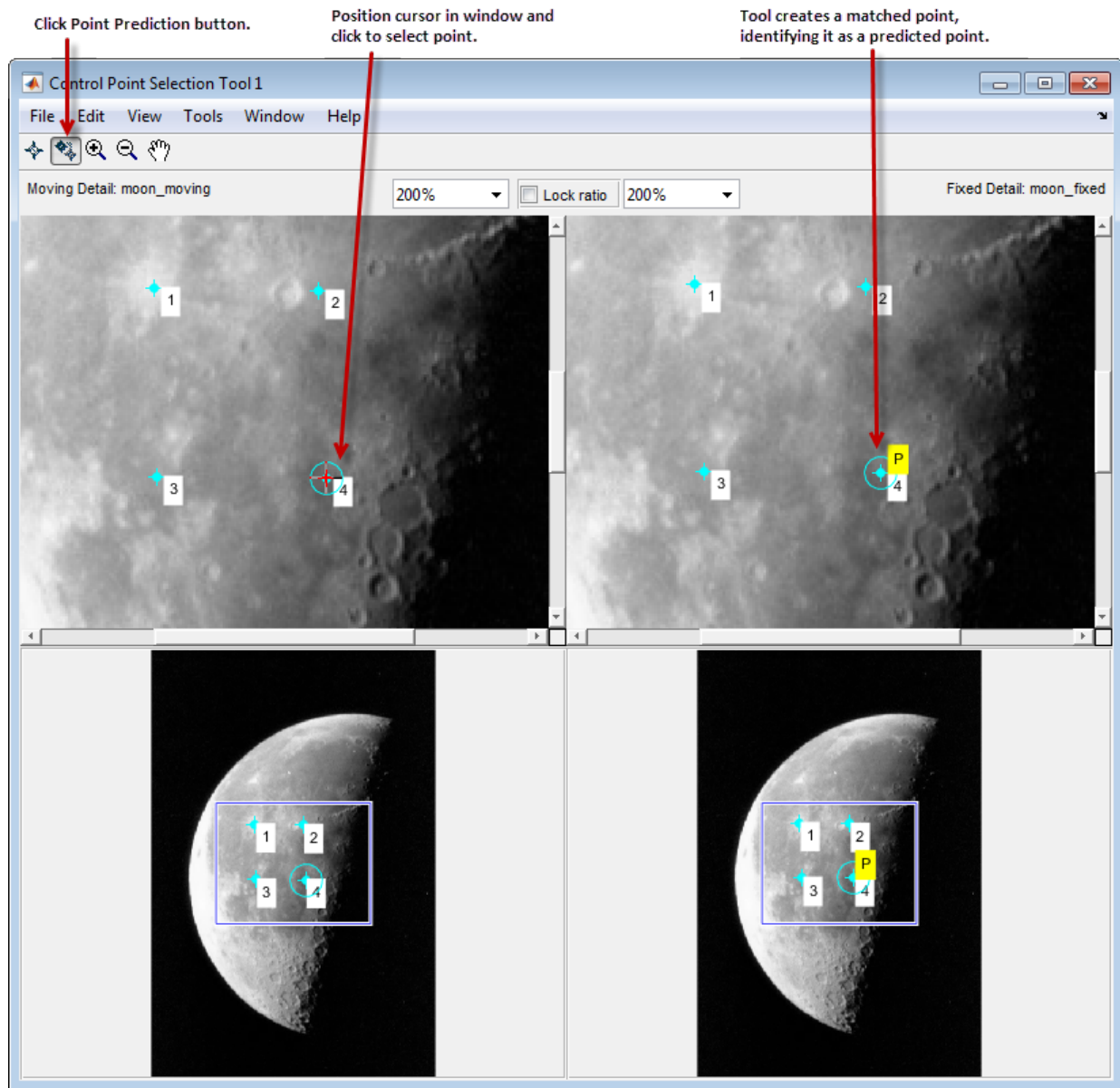
The following figure illustrates control points in several states.



Using Control Point Prediction

Instead of picking matching control points yourself, 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, not on any feature of the underlying images.

To illustrate point prediction, this figure shows four control points selected in the moving image, where the points form the four corners of a square. (The control point 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 By default, the Control Point Selection Tool does not include predicted points in the set of valid control points returned in `movingPoints` or `fixedPoints`. 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 7-25.

To use control point prediction,


- 1 Click the **Control Point Prediction** button .

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.

- 2 Position the cursor anywhere in any of the images displayed. The cursor changes to the crosshairs shape, .

You can pick control points in either of the Detail windows, moving or fixed, or in either of the Overview windows, moving or fixed. You also can work in either direction: moving-to-fixed image or fixed-to-moving 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,

- 4 To accept a predicted point, select it with the cursor and move it. The Control Point Selection Tool removes the P from the point.

Moving Control Points

To move a control point,


- 1 Click the **Control Point Selection** button .

- 2 Position the cursor over the control point you want to move. The cursor changes to the fleur shape, .
- 3 Press and hold the mouse button and drag the control point. The state of the control point changes to selected when you move it.

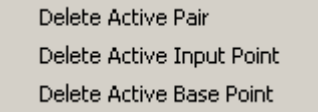
If you move a predicted control point, the state of the control point changes to a regular (nonpredicted) control point.

Deleting Control Points

To delete a control point, and its matching point, if one exists

- 1 Click the **Control Point Selection** button .
- 2 Click the control point you want to delete. Its state changes to selected. 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 moving or fixed images.



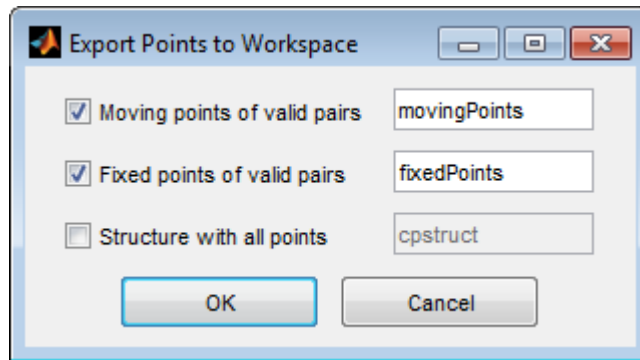
- Delete Active Pair
- Delete Active Input Point
- Delete Active Base Point

Export Control Points to the Workspace

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 `fitgeotrans`.

To save control points to the MATLAB workspace,

- 1 Select **File** on the Control Point Selection Tool menu bar.
- 2 Choose the **Export Points to Workspace** option. The Control Point Selection Tool displays this dialog box:



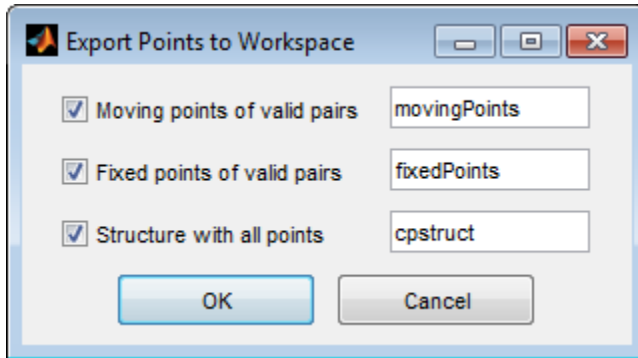
By default, the Control Point Selection Tool saves the x -coordinates and y -coordinates that specify the locations of the control points you selected in two arrays named `movingPoints` and `fixedPoints`, although you can specify other names. These are n -by-2 arrays, where n is the number of valid control point pairs you selected. This example shows the `movingPoints` array containing 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.

```
movingPoints =
    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.

Save Control Point Selection Session

To save the current state of the Control Point Selection Tool, choose the **Export Points to Workspace** option from the File menu. In the **Export Points to Workspace** dialog box, select the **Structure with all points** check box.



This option saves the positions of all the control points you specified and their current states in a `cpstruct` structure.

`cpstruct =`

```

    inputPoints: [2x2 double]
    basePoints: [2x2 double]
inputBasePairs: [2x2 double]
    ids: [2x1 double]
    inputIdPairs: [2x2 double]
    baseIdPairs: [2x2 double]
isInputPredicted: [2x1 double]
isBasePredicted: [2x1 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 `movingPoints` and `fixedPoints` arrays.

To extract the arrays of valid control point coordinates from a `cpstruct`, use the `cpstruct2pairs` function.

Use Cross-Correlation to Improve Control Point Placement

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 moving and fixed images, along with the images themselves, to the `cpcorr` function.

```
moving_pts_adj= cpcorr(movingPoints, fixedPoints, moving, fixed);
```

The `cpcorr` function defines 11-by-11 regions around each control point in the moving image and around the matching control point in the fixed 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.

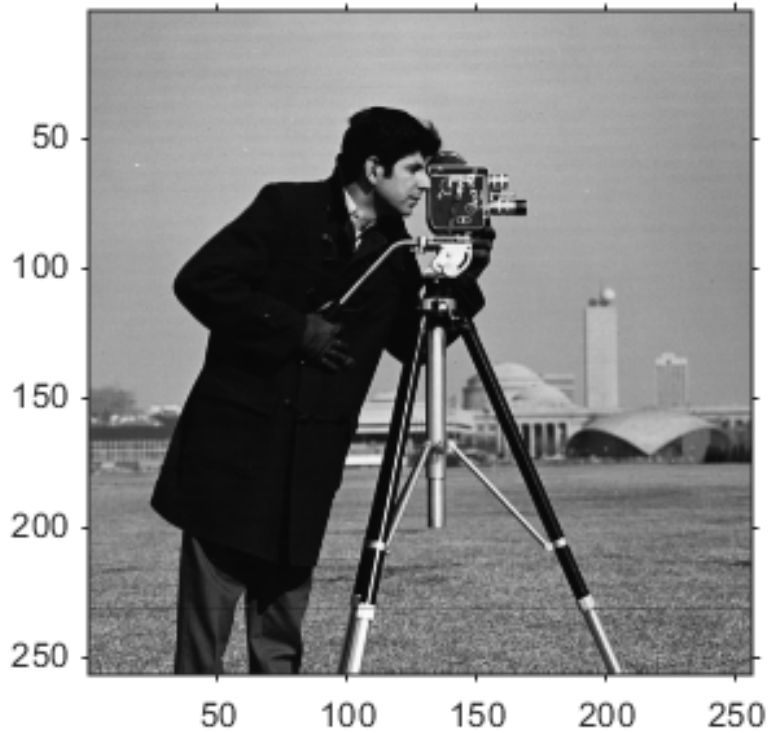
If `cpcorr` cannot correlate some of the control points, it returns their values in `movingPoints` unmodified.

Use Phase Correlation as Preprocessing Step in Registration

This example shows how to use phase correlation as a preliminary step for automatic image registration. In this process, you perform phase correlation, using `imregcorr`, and then pass the result of that registration as the initial condition of an optimization-based registration, using `imregister`. Phase correlation and optimization-based registration are complementary algorithms. Phase correlation is good for finding gross alignment, even for severely misaligned images. Optimization-based registration is good for finding precise alignment, given a good initial conditions.

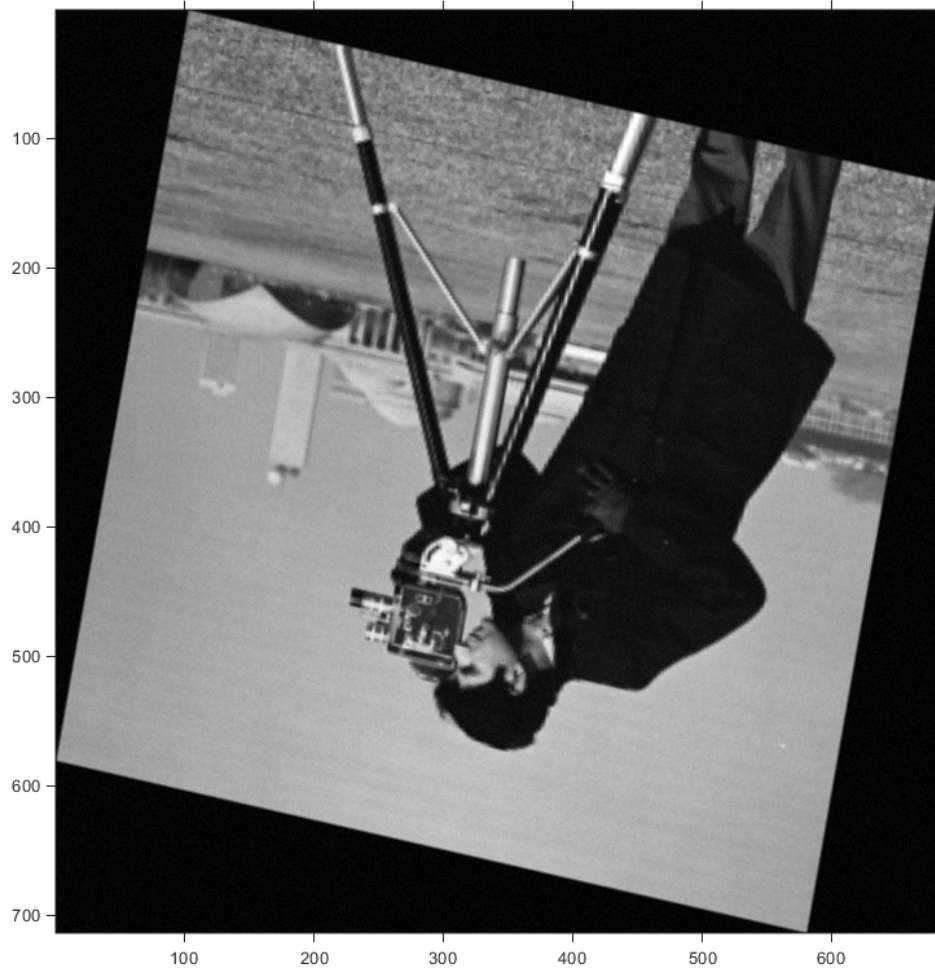
Read image that will be reference image in the registration.

```
fixed = imread('cameraman.tif');  
imshow(fixed);
```



Create an unregistered image by deliberately distorting this image using rotation, scale, and shear. Display the image.

```
theta = 170;  
S = 2.3;  
ShearY = 1.3;  
tform = affine2d([S.*cosd(theta) -S.*ShearY*sind(theta) 0; S.*sind(theta) S.*cosd(theta)  
moving = imwarp(fixed,tform);  
moving = moving + uint8(10*rand(size(moving)));  
figure, imshow(moving)
```



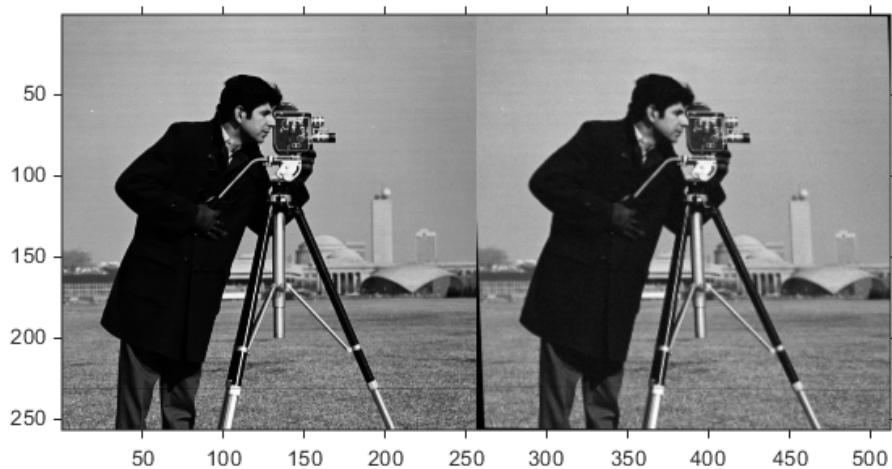
Estimate the registration required to bring these two images into alignment. `imregcorr` returns an `affine2d` object that defines the transformation.

```
tformEstimate = imregcorr(moving, fixed);
```

Apply the estimated geometric transform to the misaligned image. Specify 'OutputView' to make sure the registered image is the same size as the reference

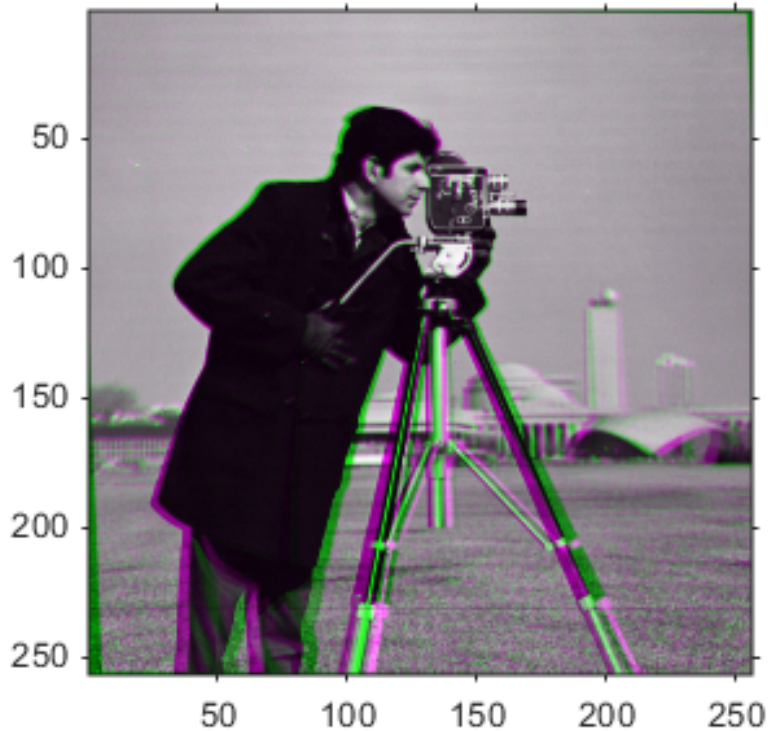
image. Display the original image and the registered image side-by-side. You can see that `imregcorr` has done a good job handling the rotation and scaling differences between the images. The registered image, `movingReg`, is very close to aligned with the original image, `fixed`. But there is some misalignment left. `imregcorr` can handle rotation and scale well, but not shear distortion.

```
Rfixed = imref2d(size(fixed));  
movingReg = imwarp(moving,tformEstimate,'OutputView',Rfixed);  
figure, imshowpair(fixed,movingReg,'montage');
```



View the aligned image overlaid on the original image, using `imshowpair`. In this view, `imshowpair` uses color to highlight areas of misalignment that remain.

```
figure, imshowpair(fixed,movingReg,'falsecolor');
```

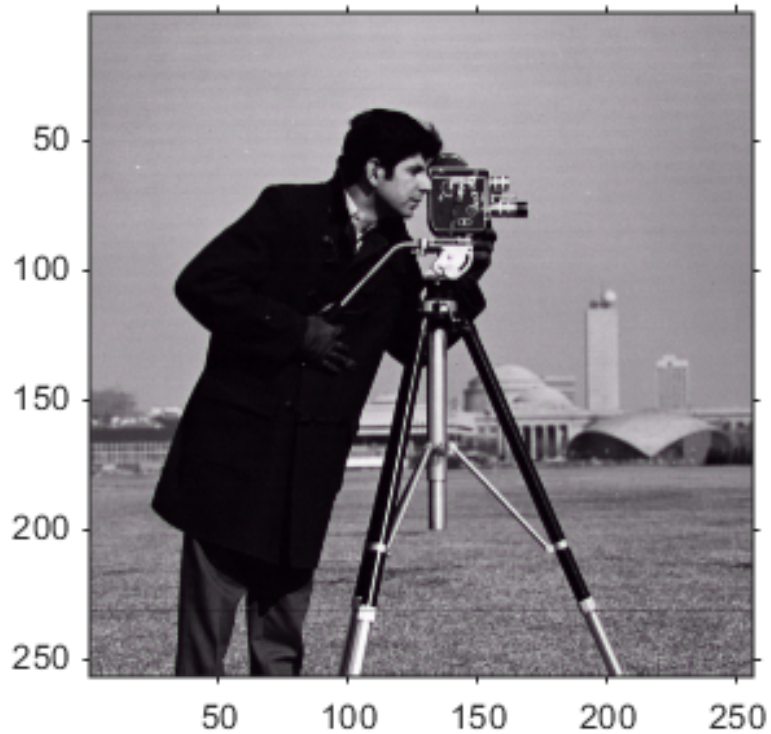


To finish the registration, use `imregister`, passing the estimated transformation returned by `imregcorr` as the initial condition. `imregister` is more effective if the two images are roughly in alignment at the start of the operation. The transformation estimated by `imregcorr` provides this information for `imregister`. The example uses the default optimizer and metric values for a registration of two images taken with the same sensor ('monomodal').

```
[optimizer, metric] = imregconfig('monomodal');  
movingRegistered = imregister(moving, fixed,...  
    'affine', optimizer, metric, 'InitialTransformation', tformEstimate);
```

Display the result of this registration. Note the `imregister` has achieved a very accurate registration, given the the good initial condition provided by `imregcorr` .

```
figure  
imshowpair(fixed, movingRegistered, 'Scaling', 'joint');
```



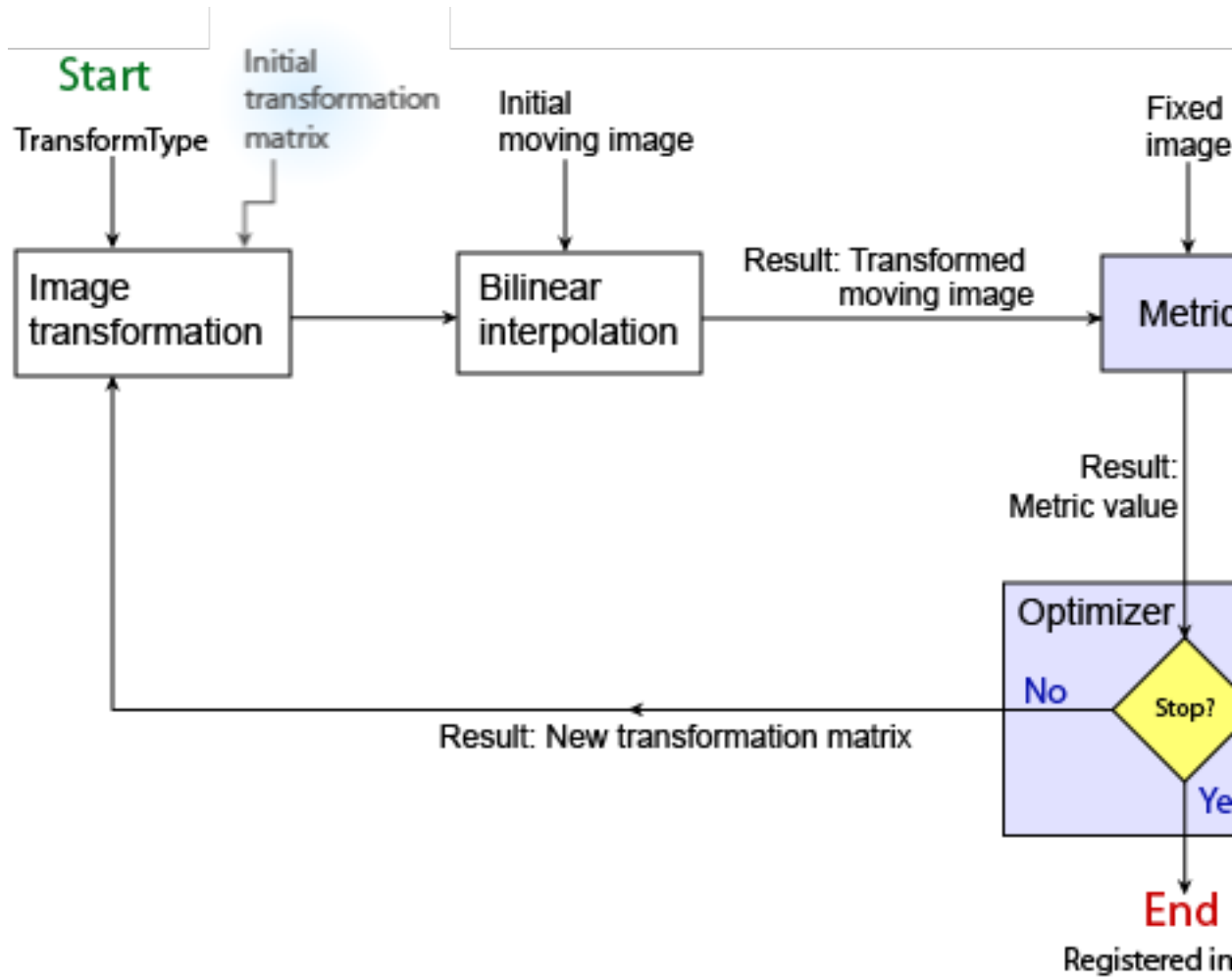
Intensity-Based Automatic Image Registration

Intensity-based automatic image registration is an iterative process. It requires that you specify a pair of images, a metric, an optimizer, and a transformation type. The metric defines the image similarity metric for evaluating the accuracy of the registration. This image similarity metric takes two images and returns a scalar value that describes how similar the images are. The optimizer defines the methodology for minimizing or maximizing the similarity metric. The transformation type defines the type of 2-D transformation that brings the misaligned image (called the *moving image*) into alignment with the reference image (called the *fixed image*).

The process begins with the transform type you specify and an internally determined transformation matrix. Together, they determine the specific image transformation that is applied to the moving image with bilinear interpolation.

Next, the metric compares the transformed moving image to the fixed image and a metric value is computed.

Finally, the optimizer checks for a stop condition. A stop condition is anything that warrants the termination of the process. In most cases, the process has reached a point of diminishing returns or it has reached the specified maximum number of iterations. If there is no stop condition, the optimizer adjusts the transformation matrix to begin the next iteration.



Perform intensity-based image registration with the following steps:

- 1 Read the images into the workspace with `imread`.
- 2 Create the optimizer and metric with `imregconfig`.
- 3 Register the images with `imregister`.
- 4 View the results with `imshowpair` or save a copy of an image showing the results with `imfuse`.

The example, “Registering Multimodal MRI Images” on page 7-39, demonstrates this workflow in detail.

Registering Multimodal MRI Images

This example shows how you can use `imregister` to automatically align two magnetic resonance images (MRI) to a common coordinate system using intensity-based image registration. Unlike some other techniques, it does not find features or use control points. Intensity-based registration is often well-suited for medical and remotely sensed imagery.

Step 1: Load Images

This example uses two magnetic resonance (MRI) images of a knee. The fixed image is a spin echo image, while the moving image is a spin echo image with inversion recovery. The two sagittal slices were acquired at the same time but are slightly out of alignment.

```
fixed = dicomread('knee1.dcm');  
moving = dicomread('knee2.dcm');
```

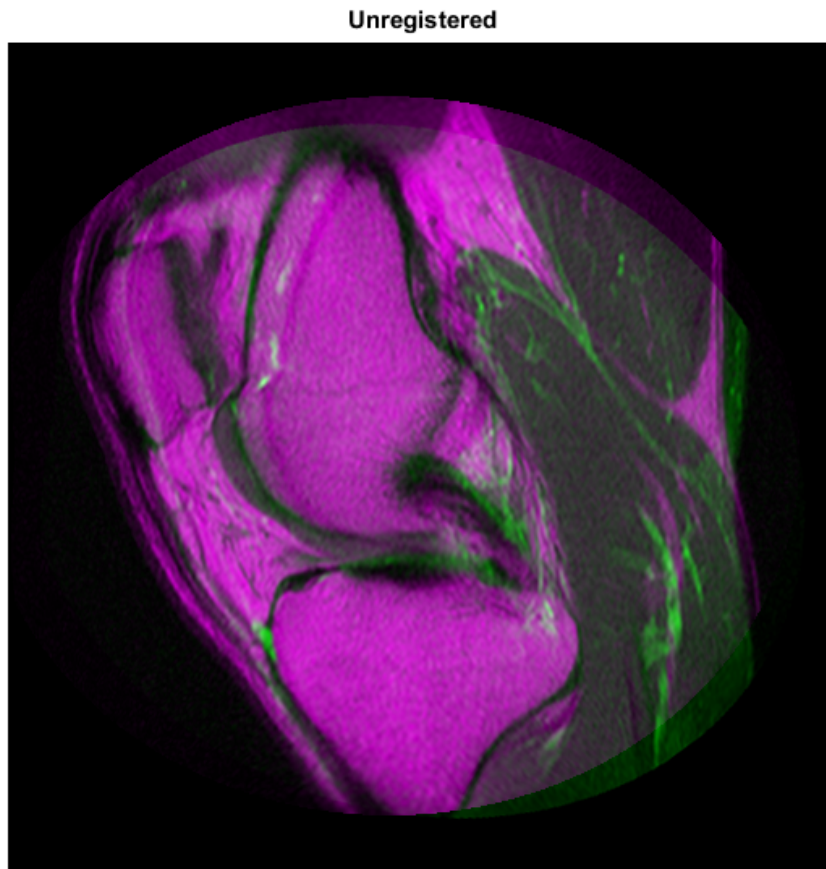
The `imshowpair` function is a useful function for visualizing images during every part of the registration process. Use it to see the two images individually in a montage fashion or display them stacked to show the amount of misregistration.

```
figure, imshowpair(moving, fixed, 'montage')  
title('Unregistered')
```



In the overlapping image from `imshowpair`, gray areas correspond to areas that have similar intensities, while magenta and green areas show places where one image is brighter than the other. In some image pairs, green and magenta areas don't always indicate misregistration, but in this example it's easy to use the color information to see where they do.

```
figure, imshowpair(moving, fixed)  
title('Unregistered')
```



Step 2: Set up the Initial Registration

The `imregconfig` function makes it easy to pick the correct optimizer and metric configuration to use with `imregister`. These two images have different intensity distributions, which suggests a multimodal configuration.

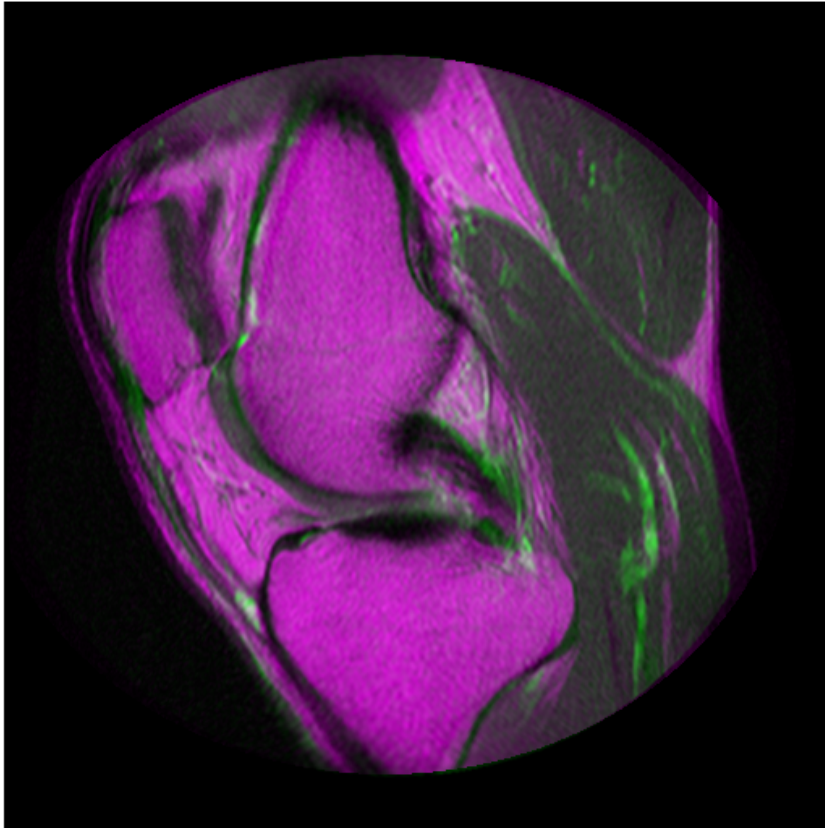
```
[optimizer,metric] = imregconfig('multimodal');
```

The distortion between the two images includes scaling, rotation, and (possibly) shear. Use an affine transformation to register the images.

It's very, very rare that `imregister` will align images perfectly with the default settings. Nevertheless, using them is a useful way to decide which properties to tune first.

```
movingRegisteredDefault = imregister(moving, fixed, 'affine', optimizer, metric);  
figure, imshowpair(movingRegisteredDefault, fixed)  
title('A: Default registration')
```

A: Default registration



Step 3: Improve the Registration

The initial registration is not very good. There are still significant regions of poor alignment, particularly along the right edge. Try to improve the registration by adjusting the optimizer and metric configuration properties.

The optimizer and metric variables are objects whose properties control the registration.

```
disp(optimizer)
```

```
disp(metric)

registration.optimizer.OnePlusOneEvolutionary

Properties:
    GrowthFactor: 1.050000e+00
    Epsilon: 1.500000e-06
    InitialRadius: 6.250000e-03
    MaximumIterations: 100
registration.metric.MattesMutualInformation

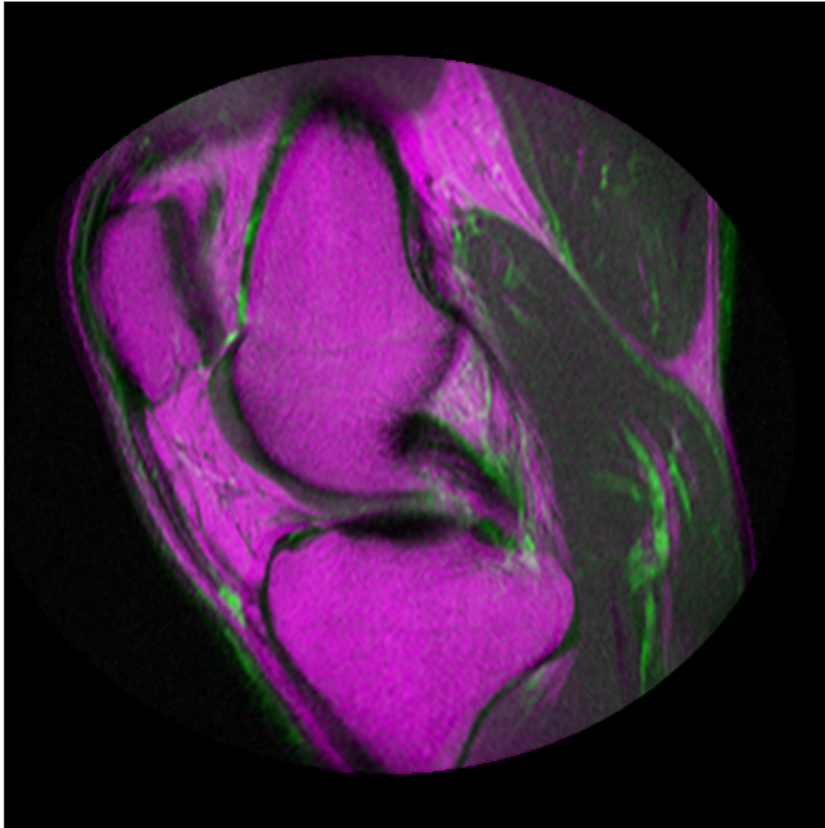
Properties:
    NumberOfSpatialSamples: 500
    NumberOfHistogramBins: 50
    UseAllPixels: 1
```

The `InitialRadius` property of the optimizer controls the initial step size used in parameter space to refine the geometric transformation. When multi-modal registration problems do not converge with the default parameters, the `InitialRadius` is a good first parameter to adjust. Start by reducing the default value of `InitialRadius` by a scale factor of 3.

```
optimizer.InitialRadius = optimizer.InitialRadius/3.5;

movingRegisteredAdjustedInitialRadius = imregister(moving, fixed, 'affine', optimizer,
figure, imshowpair(movingRegisteredAdjustedInitialRadius, fixed)
title('Adjusted InitialRadius')
```

Adjusted InitialRadius

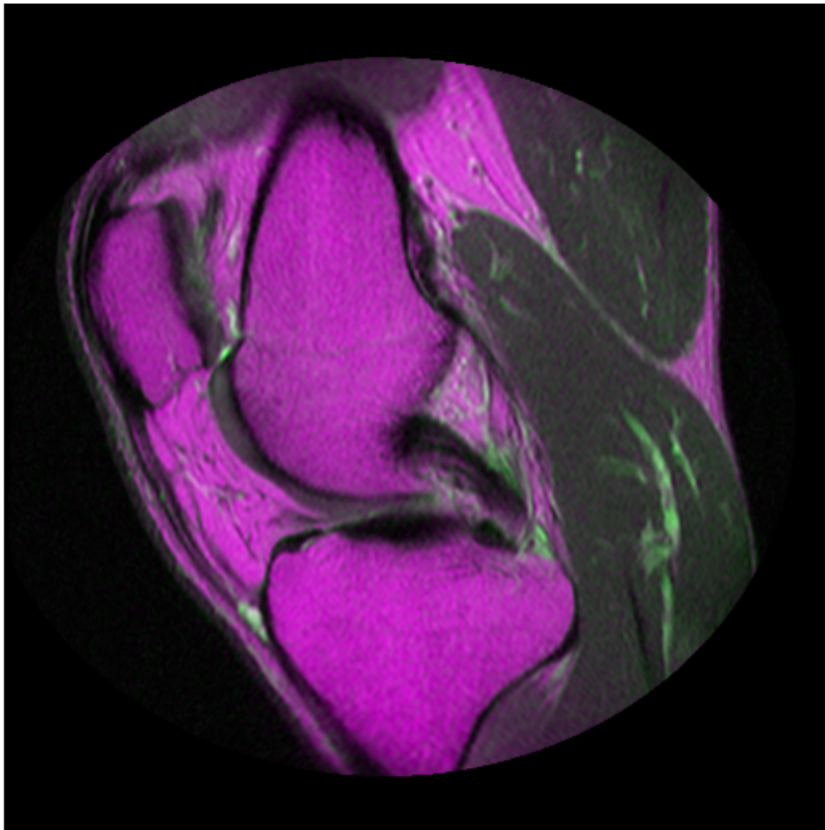


Adjusting the InitialRadius had a positive impact. There is a noticeable improvement in the alignment of the images at the top and right edges.

The MaximumIterations property of the optimizer controls the maximum number of iterations that the optimizer will be allowed to take. Increasing the MaximumIterations allows the registration search to run longer and potentially find better registration results. Does the registration continue to improve if the InitialRadius from the last step is used with a large number of iterations?

```
optimizer.MaximumIterations = 300;  
movingRegisteredAdjustedInitialRadius300 = imregister(moving, fixed, 'affine', optimizer,  
figure, imshowpair(movingRegisteredAdjustedInitialRadius300, fixed)  
title('B: Adjusted InitialRadius, MaximumIterations = 300, Adjusted InitialRadius.')
```

B: Adjusted InitialRadius, MaximumIterations = 300, Adjusted InitialRadius.



Further improvement in registration were achieved by reusing the InitialRadius optimizer setting from the previous registration and allowing the optimizer to take a large number of iterations.

Step 4: Use Initial Conditions to Improve Registration

Optimization based registration works best when a good initial condition can be given for the registration that relates the moving and fixed images. A useful technique for getting improved registration results is to start with more simple transformation types like 'rigid', and then use the resulting transformation as an initial condition for more complicated transformation types like 'affine'.

The function `imregtform` uses the same algorithm as `imregister`, but returns a geometric transformation object as output instead of a registered output image. Use `imregtform` to get an initial transformation estimate based on a 'similarity' model (translation, rotation, and scale).

The previous registration results showed in improvement after modifying the `MaximumIterations` and `InitialRadius` properties of the optimizer. Keep these optimizer settings while using initial conditions while attempting to refine the registration further.

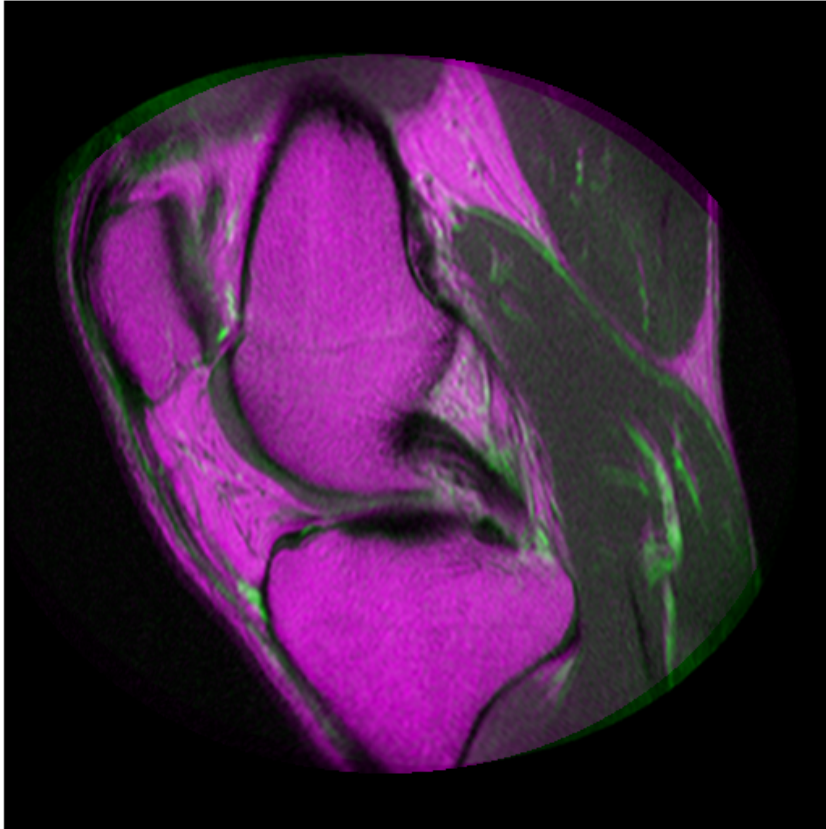
```
tformSimilarity = imregtform(moving, fixed, 'similarity', optimizer, metric);
```

Because the registration is being solved in the default MATLAB coordinate system, also known as the intrinsic coordinate system, obtain the default spatial referencing object that defines the location and resolution of the fixed image.

```
Rfixed = imref2d(size(fixed));
```

Use `imwarp` to apply the geometric transformation output from `imregtform` to the moving image to align it with the fixed image. Use the 'OutputView' option in `imwarp` to specify the world limits and resolution of the output resampled image. Specifying `Rfixed` as the 'OutputView' forces the resampled moving image to have the same resolution and world limits as the fixed image.

```
movingRegisteredRigid = imwarp(moving, tformSimilarity, 'OutputView', Rfixed);  
figure, imshowpair(movingRegisteredRigid, fixed);  
title('C: Registration based on similarity transformation model.');
```

C: Registration based on similarity transformation model.

The "T" property of the output geometric transformation defines the transformation matrix that maps points in moving to corresponding points in fixed.

```
tformSimilarity.T
```

```
ans =
```

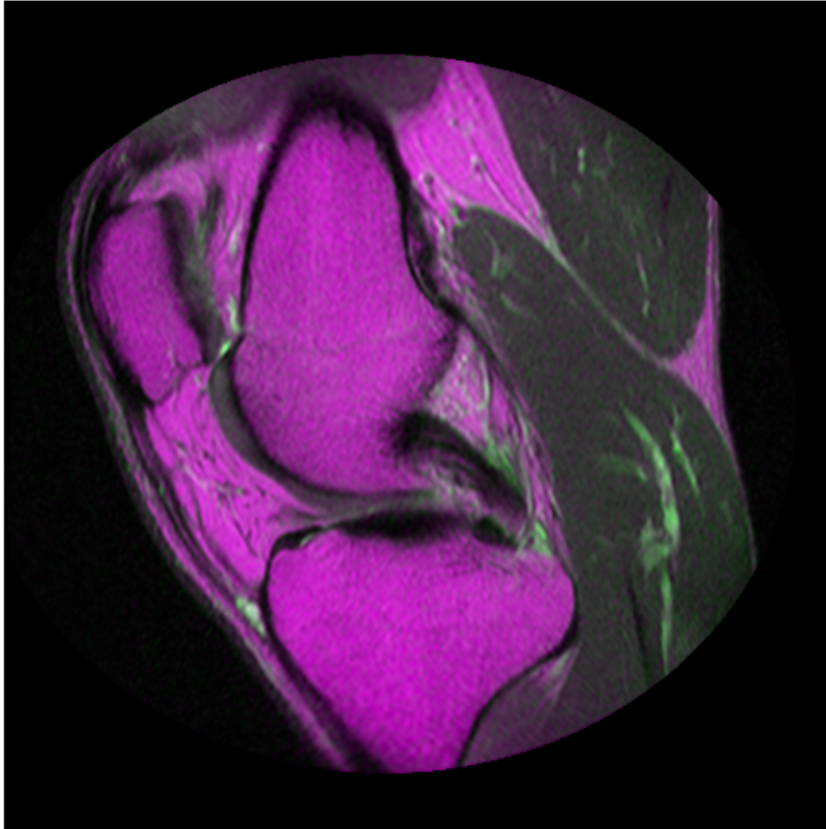
```
1.0331 -0.1110 0
0.1110 1.0331 0
```

-51.1491 6.9891 1.0000

Use the 'InitialTransformation' Name/Value in `imregister` to refine this registration by using an 'affine' transformation model with the 'similarity' results used as an initial condition for the geometric transformation. This refined estimate for the registration includes the possibility of shear.

```
movingRegisteredAffineWithIC = imregister(moving, fixed, 'affine', optimizer, metric, ...  
    'InitialTransformation', tformSimilarity);  
figure, imshowpair(movingRegisteredAffineWithIC, fixed);  
title('D: Registration from affine model based on similarity initial condition.');
```


D: Registration from affine model based on similarity initial condition.



Using the 'InitialTransformation' to refine the 'similarity' result of `imregtform` with a full affine model has also yielded a nice registration result.

Step 5: Deciding When Enough is Enough

Comparing the results of running `imregister` with different configurations and initial conditions, it becomes apparent that there are a large number of input parameters that can be varied in `imregister`, each of which may lead to different registration results.

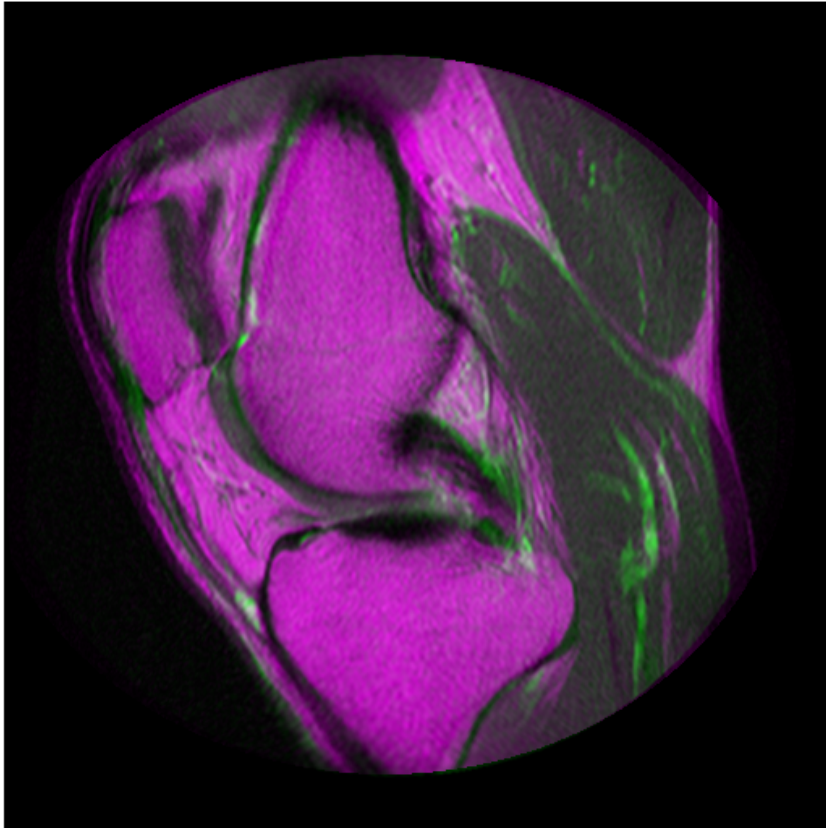
```
figure
imshowpair(movingRegisteredDefault, fixed)
title('A - Default settings.');
```

```
figure
imshowpair(movingRegisteredAdjustedInitialRadius, fixed)
title('B - Adjusted InitialRadius, 100 Iterations.');
```

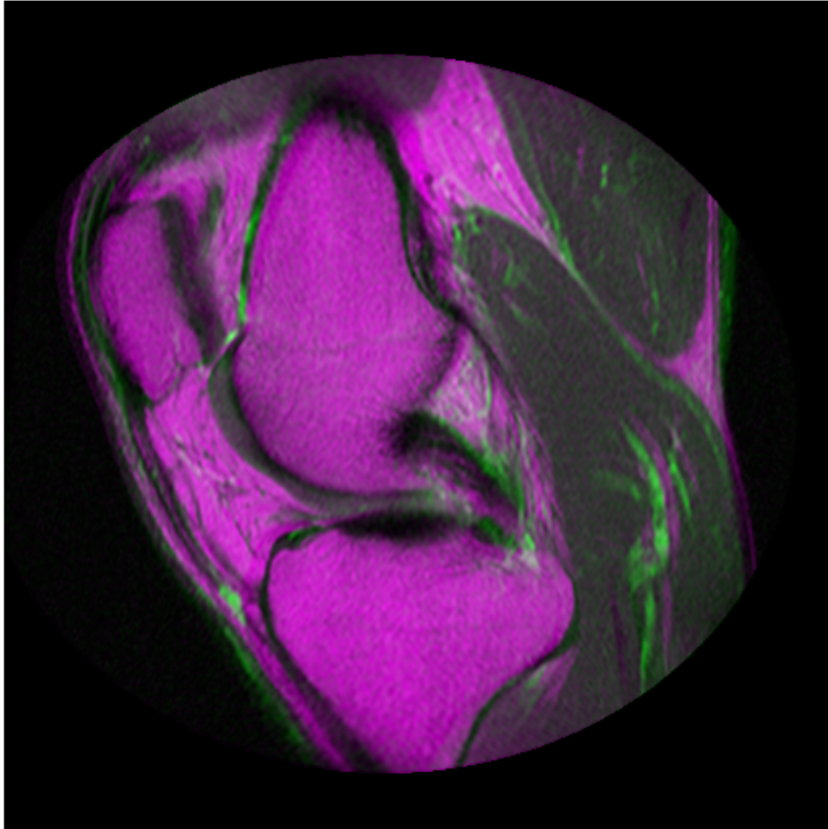
```
figure
imshowpair(movingRegisteredAdjustedInitialRadius300, fixed)
title('C - Adjusted InitialRadius, 300 Iterations.');
```

```
figure
imshowpair(movingRegisteredAffineWithIC, fixed)
title('D - Registration from affine model based on similarity initial condition.');
```

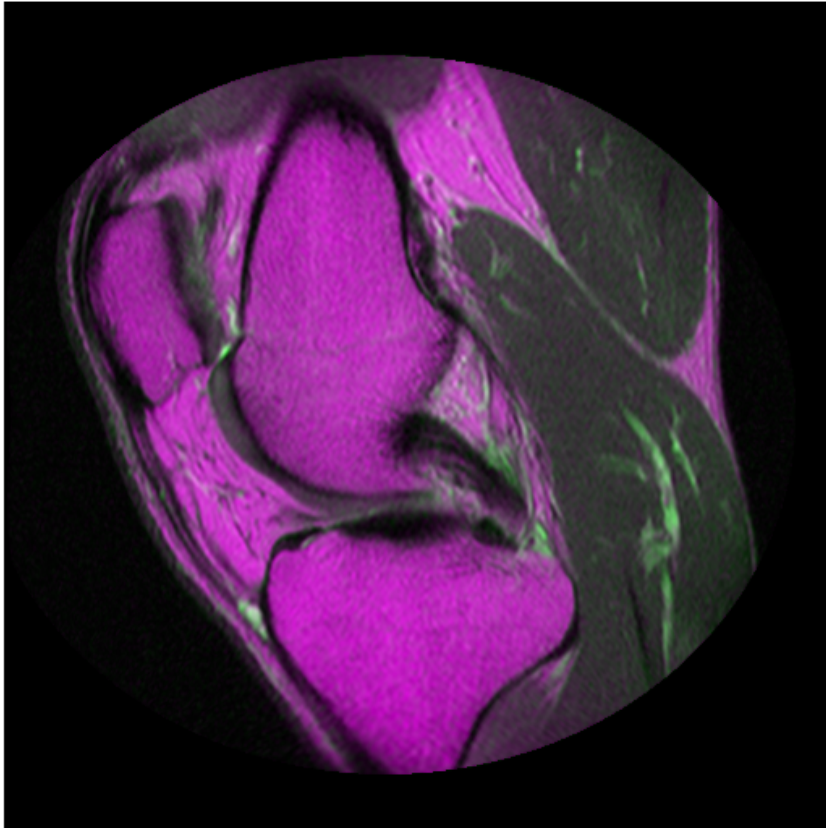
A - Default settings.



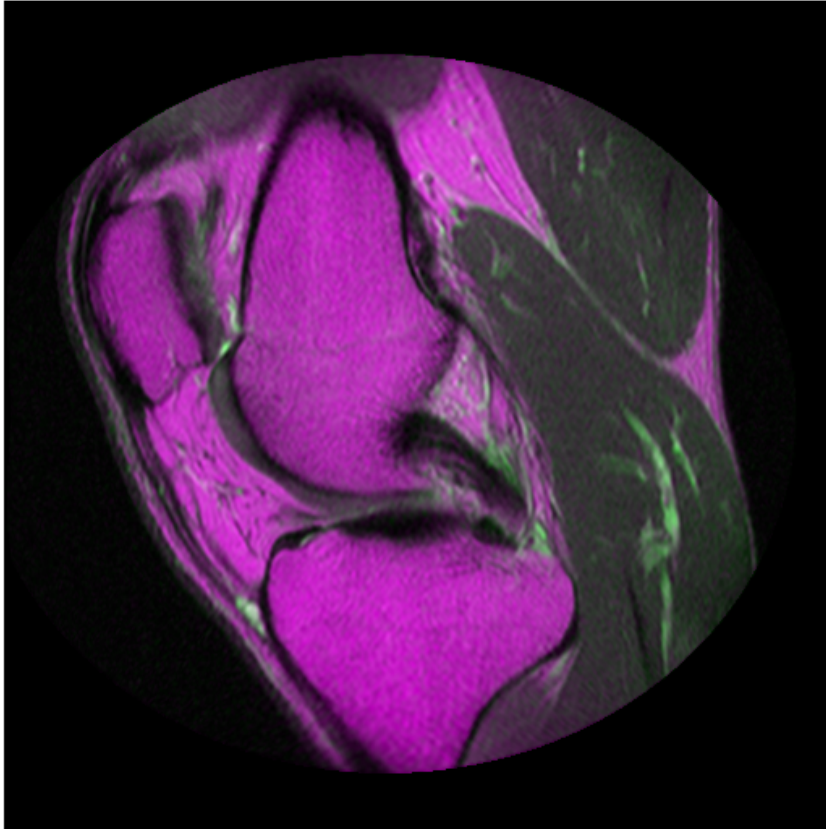
B - Adjusted InitialRadius, 100 Iterations.



C - Adjusted InitialRadius, 300 Iterations.



D - Registration from affine model based on similarity initial condition.



It can be difficult to quantitatively compare registration results because there is no one quality metric that accurately describes the alignment of two images. Often, registration results must be judged qualitatively by visualizing the results. In The results above, the registration results in C) and D) are both very good and are difficult to tell apart visually.

Step 6: Alternate Visualizations

Often as the quality of multimodal registrations improve it becomes more difficult to judge the quality of registration visually. This is because the intensity differences can

obscure areas of misalignment. Sometimes switching to a different display mode for `imshowpair` exposes hidden details. (This is not always the case.)

Designing and Implementing Linear Filters for Image Data

The Image Processing Toolbox software 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.

- “What Is Image Filtering in the Spatial Domain?” on page 8-2
- “Filter Images Using `imfilter`” on page 8-5
- “How `imfilter` Handles Data Types” on page 8-7
- “`imfilter` Boundary Padding Options” on page 8-9
- “Filter Images Using `imfilter` with Convolution” on page 8-13
- “Filter Images Using Predefined Filters” on page 8-14
- “Filter Multidimensional Images with `imfilter`” on page 8-17
- “What is Guided Image Filtering?” on page 8-19
- “Perform Flash/No-flash Denoising with Guided Filter” on page 8-20
- “Designing Linear Filters in the Frequency Domain” on page 8-25
- “Two-Dimensional Finite Impulse Response (FIR) Filters” on page 8-33

What Is Image Filtering in the Spatial Domain?

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 “Neighborhood or Block Processing: An Overview” 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.

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

$$A = \begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

and the convolution kernel is

$$h = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

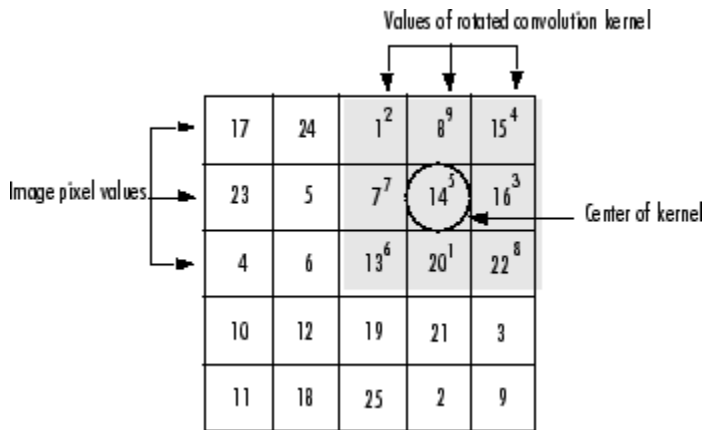
The following figure shows how to compute the (2,4) output pixel using these steps:

- 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

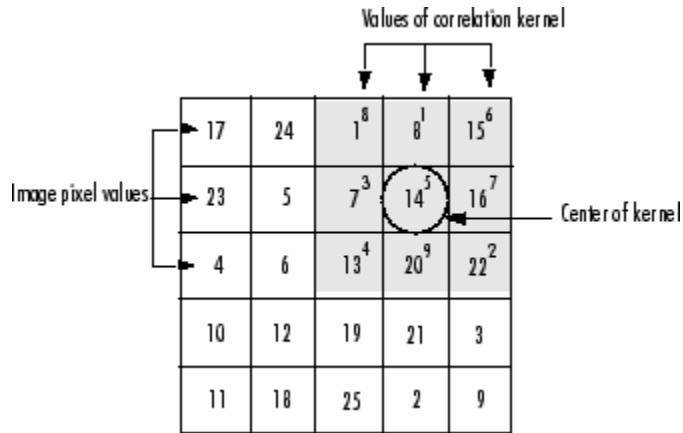
The operation called *correlation* is closely related to convolution. In correlation, the value of an output pixel is also computed as a weighted sum of neighboring pixels. The difference is that the matrix of weights, in this case called the *correlation kernel*, is not rotated during the computation. The Image Processing Toolbox filter design functions 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

Filter Images Using `imfilter`

This example shows how to filter an image with a 5-by-5 filter containing equal weights (often called an *averaging filter*) using `imfilter`.

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.

Read image.

```
I = imread('coins.png');
```

Create filter.

```
h = ones(5,5) / 25;
```

Apply filter to image using `imfilter`.

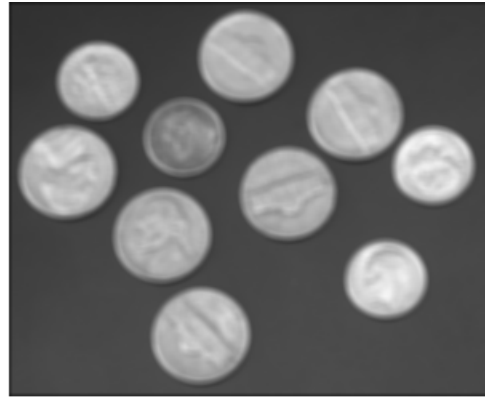
```
I2 = imfilter(I,h);
```

Display original image and filtered image for comparison.

```
imshow(I), title('Original Image');  
figure, imshow(I2), title('Filtered Image')
```



Original Image



Filtered Image

How `imfilter` Handles 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 2-26.

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`.

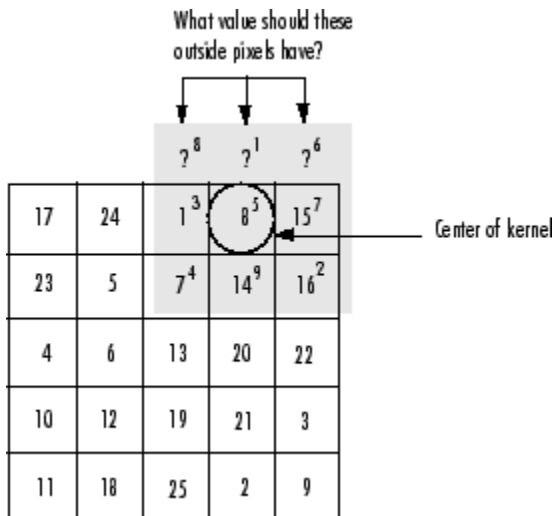
```
A = uint8(magic(5));  
imfilter(A,h)
```

```
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`.

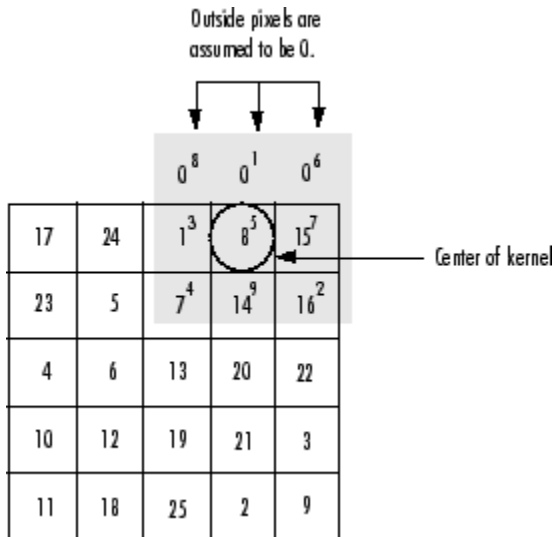
imfilter Boundary Padding Options

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')
```

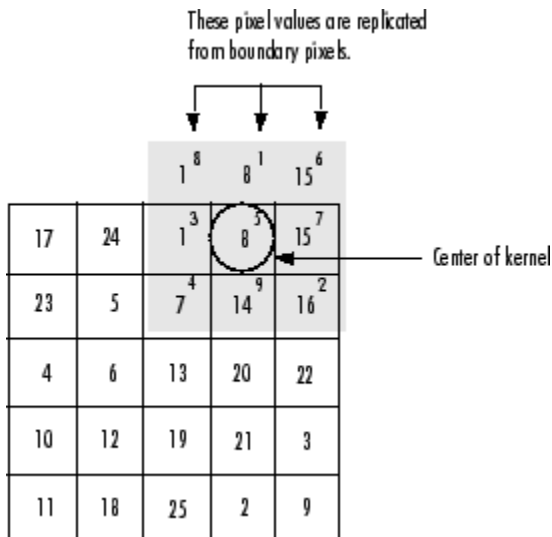


Original Image



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.

Filter Images Using `imfilter` with Convolution

This example shows how to perform image filtering using convolution with `imfilter`. By default, `imfilter` uses correlation because the toolbox filter design functions produce correlation kernels.

Create a sample matrix.

```
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
```

Create a filter.

```
h = [-1 0 1];
```

Filter using correlation, the default.

```
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
```

Filter using convolution, using the parameter.

```
imfilter(A,h,'conv')
```

```
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
```

Filter Images Using Predefined Filters

This example shows how to create filters using the `fspecial` function that can be used with `imfilter`. The `fspecial` function produces several kinds of predefined filters, in the form of correlation kernels. 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.

Read image.

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

Create filter, using `fspecial`.

```
h = fspecial('unsharp')
```

h =

```
-0.1667  -0.6667  -0.1667
-0.6667   4.3333  -0.6667
-0.1667  -0.6667  -0.1667
```

Apply filter to image using `imfilter`.

```
I2 = imfilter(I,h);
```

Display original image and filtered image for comparison.

```
imshow(I)
title('Original Image')
figure
imshow(I2)
title('Filtered Image')
```

Original Image



Filtered Image



Filter Multidimensional Images with `imfilter`

This example shows how to filter a multidimensional truecolor image using `imfilter`. 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.

Read truecolor image.

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



Create a filter.

```
h = ones(5,5)/25
```

```
h =
```

```
0.0400    0.0400    0.0400    0.0400    0.0400  
0.0400    0.0400    0.0400    0.0400    0.0400  
0.0400    0.0400    0.0400    0.0400    0.0400  
0.0400    0.0400    0.0400    0.0400    0.0400  
0.0400    0.0400    0.0400    0.0400    0.0400
```

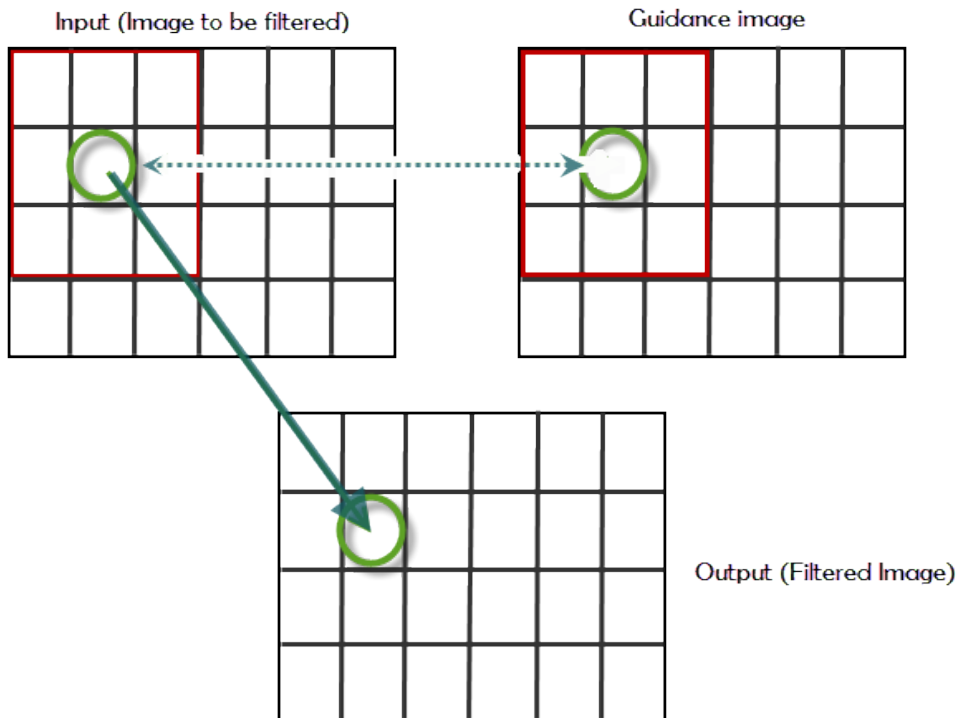
Filter the image using `imfilter` and display it.

```
h = ones(5,5)/25;  
rgb2 = imfilter(rgb,h);  
figure, imshow(rgb2)
```



What is Guided Image Filtering?

The `imguidedfilter` function performs edge-preserving smoothing on an image, using the content of a second image, called a *guidance image*, to influence the filtering. The guidance image can be the image itself, a different version of the image, or a completely different image. Guided image filtering is a neighborhood operation, like other filtering operations, but takes into account the statistics of a region in the corresponding spatial neighborhood in the guidance image when calculating the value of the output pixel.



If the guidance is the same as the image to be filtered, the structures are the same—an edge in original image is the same in the guidance image. If the guidance image is different, structures in the guidance image will impact the filtered image, in effect, imprinting these structures on the original image. This effect is called *structure transference*.

Perform Flash/No-flash Denoising with Guided Filter

This example shows how to use a guided filter to smooth an image, reducing noise, while preserving edges. The example uses two pictures of the same scene, one taken with a flash and the other without a flash. The version without a flash preserves colors but is noisy due to the low-light conditions. This example uses the version taken with a flash as the guidance image.

Read the image that you want to filter. In this example, it's an image of some children's toys taken without a flash, `A`. Because of the low light conditions, the image contains a lot of noise.

```
A = imread('toysnoflash.png');  
figure;  
imshow(A);  
title('Input Image - Camera Flash Off')
```

Input Image - Camera Flash Off



Next, read the image that you want to use as a guidance image. In this example, the guidance image is a picture of the same scene taken with a flash.

```
G = imread('toysflash.png');  
figure;  
imshow(G);  
title('Guidance Image - Camera Flash On')
```

Guidance Image - Camera Flash On



Perform the guided filtering operation. Using the `imguidedfilter` function, you can specify the size of the neighborhood used for filtering. The default is a 5-by-5 square. This example uses a 3-by-3 neighborhood. You can also specify the amount of smoothing performed by the filter. The value can be any positive number. One way to approach this is to use the default first and view the results. If you want less smoothing and more edge preservation, use a lower value for this parameter. For more smoothing, use a higher value. This example sets the value of the smoothing parameter.

```
nhoodSize = 3;
smoothValue = 0.001*diff(getrangefromclass(G)).^2;
B = imguidedfilter(A, G, 'NeighborhoodSize',nhoodSize, 'DegreeOfSmoothing',smoothValue);
```

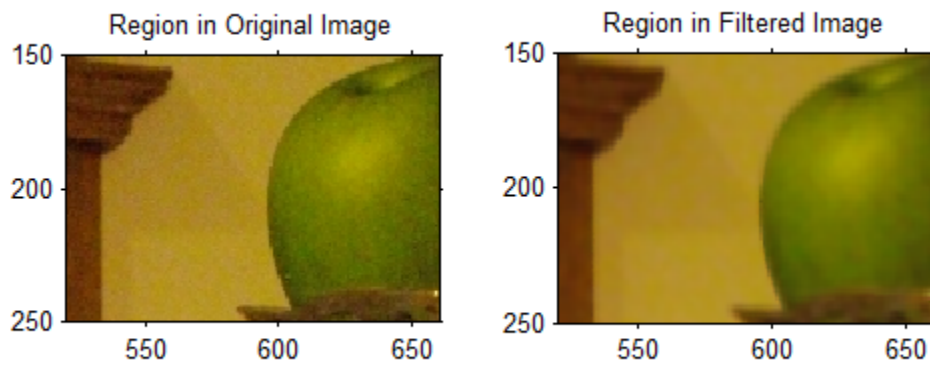


```
figure, imshow(B), title('Filtered Image')
```



Examine a close up of an area of the original image and compare it to the filtered image to see the effect of this edge-preserving smoothing filter.

```
figure;
h1 = subplot(1,2,1);
imshow(A), title('Region in Original Image'), axis on
h2 = subplot(1,2,2);
imshow(B), title('Region in Filtered Image'), axis on
linkaxes([h1 h2])
xlim([520 660])
ylim([150 250])
```



Designing Linear Filters in the Frequency Domain

In this section...

“Transform 1-D FIR Filter to 2-D FIR Filter” on page 8-25

“Frequency Sampling Method” on page 8-28

“Windowing Method” on page 8-29

“Creating the Desired Frequency Response Matrix” on page 8-30

“Computing the Frequency Response of a Filter” on page 8-31

For information about designing linear filters in the spatial domain, see “What Is Image Filtering in the Spatial Domain?” on page 8-2.

Transform 1-D FIR Filter to 2-D FIR Filter

This example shows how to transform a one-dimensional FIR filter into a two-dimensional FIR filter using the `ftrans2` function. This function can be useful because it is easier to design a one-dimensional filter with particular characteristics than a corresponding two-dimensional filter. The frequency transformation method preserves most of the characteristics of the one-dimensional filter, particularly the transition bandwidth and ripple characteristics. The shape of the one-dimensional frequency response is clearly evident in the two-dimensional response.

This function uses *atransformation matrix*, a set of elements that defines the frequency transformation. 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.)

Create 1-D FIR filter.

```
b = remez(10,[0 0.4 0.6 1],[1 1 0 0])
```

```
b =
```

```
Columns 1 through 9
```

```
0.0537 -0.0000 -0.0916 -0.0001 0.3131 0.4999 0.3131 -0.0001 -0.0
```

```
Columns 10 through 11
```

```
-0.0000    0.0537
```

Transform the 1-D filter to a 2-D filter.

```
h = ftrans2(b);
```

```
h =
```

```
Columns 1 through 9
```

```

0.0001    0.0005    0.0024    0.0063    0.0110    0.0132    0.0110    0.0063    0.0001
0.0005    0.0031    0.0068    0.0042   -0.0074   -0.0147   -0.0074    0.0042    0.0001
0.0024    0.0068   -0.0001   -0.0191   -0.0251   -0.0213   -0.0251   -0.0191   -0.0001
0.0063    0.0042   -0.0191   -0.0172    0.0128    0.0259    0.0128   -0.0172   -0.0001
0.0110   -0.0074   -0.0251    0.0128    0.0924    0.1457    0.0924    0.0128   -0.0001
0.0132   -0.0147   -0.0213    0.0259    0.1457    0.2021    0.1457    0.0259   -0.0001
0.0110   -0.0074   -0.0251    0.0128    0.0924    0.1457    0.0924    0.0128   -0.0001
0.0063    0.0042   -0.0191   -0.0172    0.0128    0.0259    0.0128   -0.0172   -0.0001
0.0024    0.0068   -0.0001   -0.0191   -0.0251   -0.0213   -0.0251   -0.0191   -0.0001
0.0005    0.0031    0.0068    0.0042   -0.0074   -0.0147   -0.0074    0.0042    0.0001
0.0001    0.0005    0.0024    0.0063    0.0110    0.0132    0.0110    0.0063    0.0001

```

```
Columns 10 through 11
```

```

0.0005    0.0001
0.0031    0.0005
0.0068    0.0024
0.0042    0.0063
-0.0074    0.0110
-0.0147    0.0132
-0.0074    0.0110
0.0042    0.0063
0.0068    0.0024
0.0031    0.0005
0.0005    0.0001

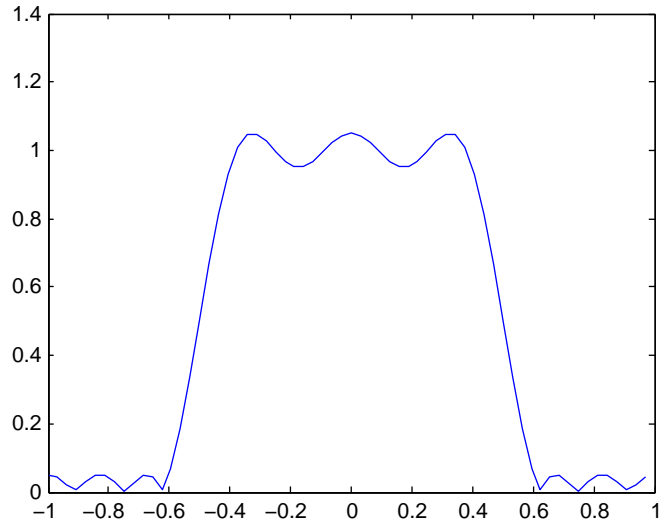
```

View the frequency response of the filters.

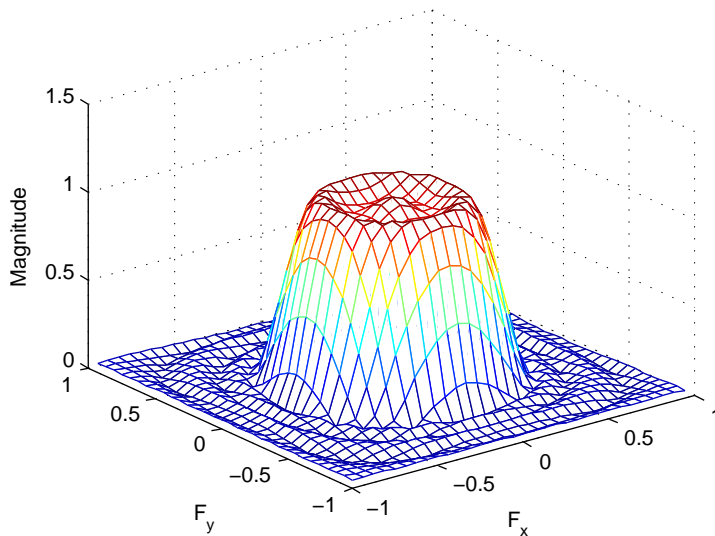
```

[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.)

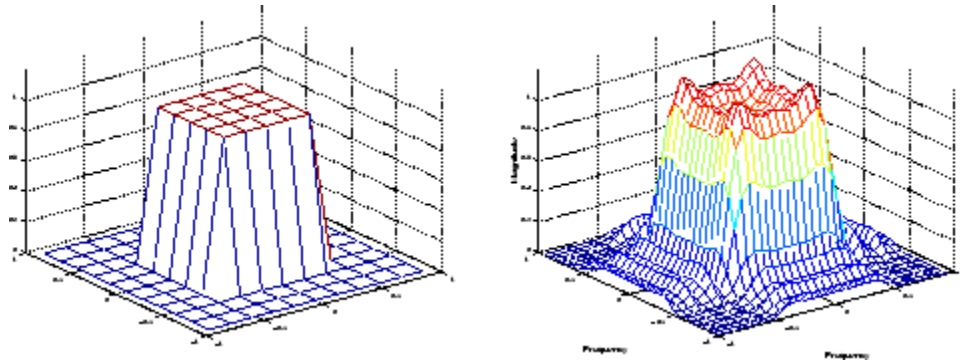
The toolbox function `fsamp2` implements frequency sampling design for two-dimensional FIR filters. `fsamp2` returns a filter `h` with a frequency response that passes through the points in the input matrix `Hd`. The example below creates an 11-by-11 filter using `fsamp2` and plots the frequency response 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 8-31 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.

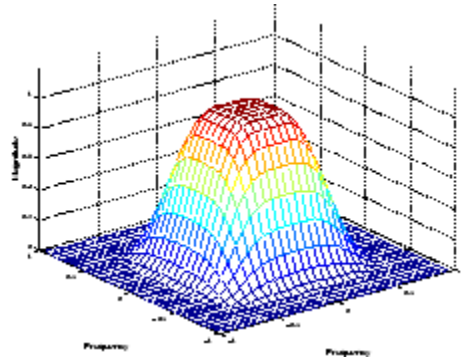
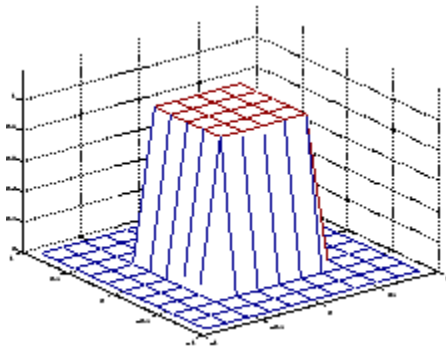
The toolbox provides two functions for window-based filter design, `fwind1` and `fwind2`. `fwind1` designs a two-dimensional filter by using a two-dimensional window that it creates from one or two one-dimensional windows that you specify. `fwind2` designs a two-dimensional filter by using a specified 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`. The example uses the Signal Processing Toolbox `hamming` function 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

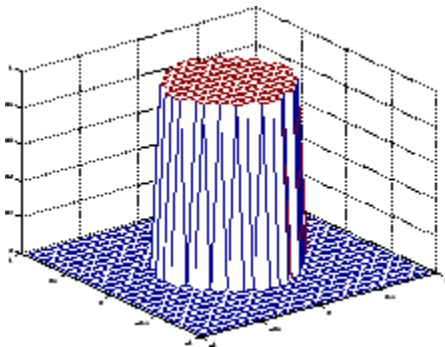
The filter design functions `fsamp2`, `fwind1`, and `fwind2` all create filters based on a desired frequency response magnitude matrix. Frequency response is a mathematical function describing the gain of a filter in response to different input frequencies.

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 ($f_1 = 0$, $f_2 = 0$).

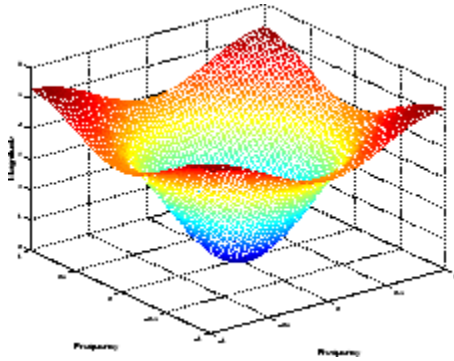
Computing the Frequency Response of a Filter

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 π radians.

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

See “Fourier Transform” on page 9-2 for more information about the fast Fourier transform and its application to linear filtering and filter design.

Two-Dimensional Finite Impulse Response (FIR) Filters

The Image Processing Toolbox software 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 the Image Processing Toolbox filter design functions 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.

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 Signal Processing Toolbox functions. Although this toolbox is not required, you might find it difficult to design filters if you do not have the Signal Processing Toolbox software.

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” on page 9-2
- “Discrete Cosine Transform” on page 9-15
- “Radon Transform” on page 9-19
- “The Inverse Radon Transformation” on page 9-29
- “Fan-Beam Projection Data” on page 9-35

Fourier Transform

In this section...

“Definition of Fourier Transform” on page 9-2

“Discrete Fourier Transform” on page 9-7

“Applications of the Fourier Transform” on page 9-10

Definition of 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.

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

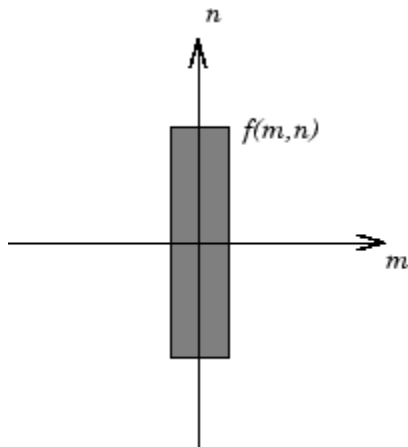
The inverse of a transform is an operation that when performed on a transformed image produces the original image. The inverse two-dimensional Fourier transform is given by

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

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 (ω_1, ω_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.

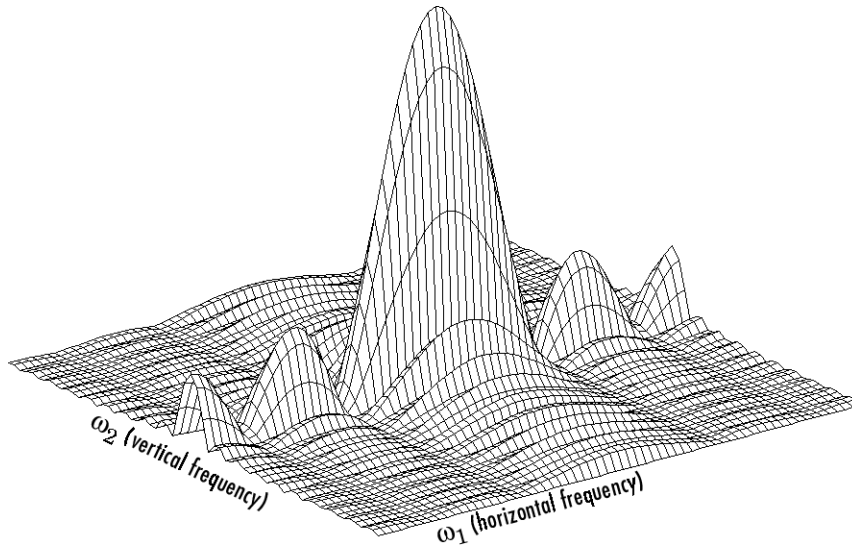


Rectangular Function

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.



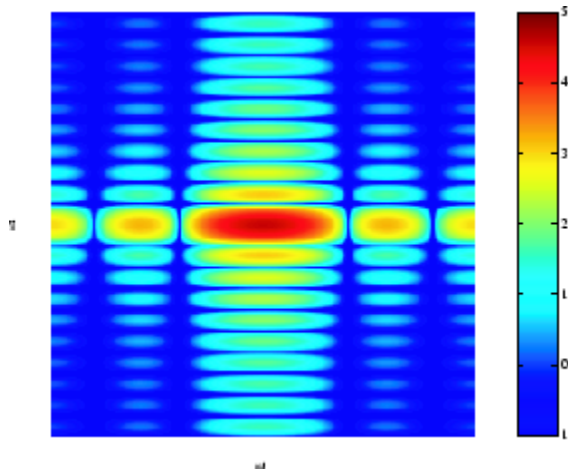
Magnitude Image of a Rectangular Function

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

$$\log |F(\omega_1, \omega_2)|$$

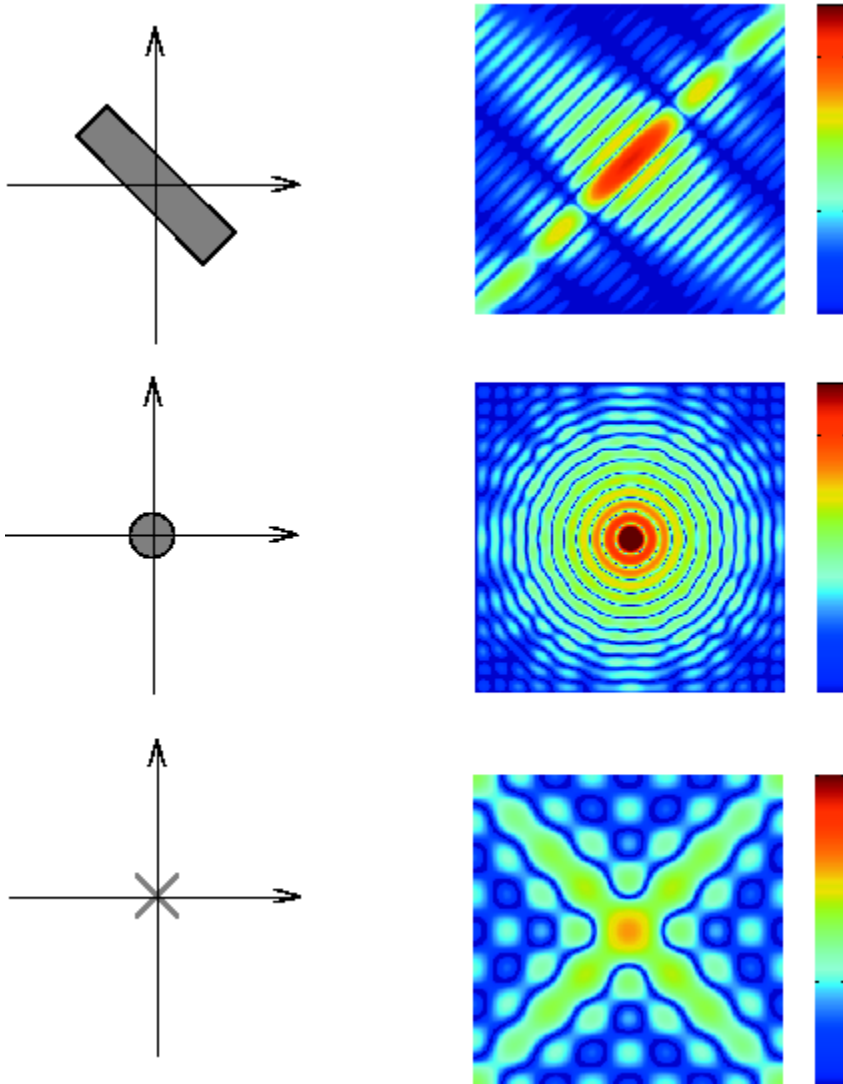
as an image, as shown.



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

$$F(p,q) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) e^{-j2\pi pm/M} e^{-j2\pi qn/N} \quad \begin{array}{l} p = 0, 1, \dots, M-1 \\ q = 0, 1, \dots, N-1 \end{array}$$

and

$$f(m,n) = \frac{1}{MN} \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} F(p,q) e^{j2\pi pm/M} e^{j2\pi qn/N} \quad \begin{array}{l} m = 0, 1, \dots, M-1 \\ n = 0, 1, \dots, N-1 \end{array}$$

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

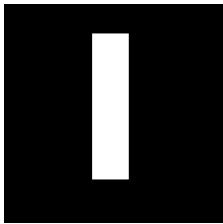
The DFT coefficients $F(p,q)$ are samples of the Fourier transform $F(\omega_1, \omega_2)$.

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

Visualizing the Discrete Fourier Transform

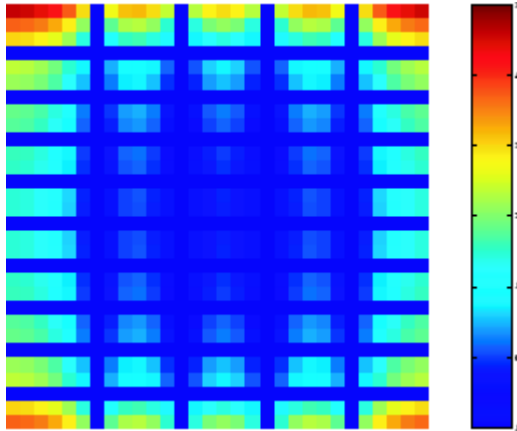
- 1 Construct a matrix `f` that is similar to the function $f(m, n)$ in the example in “Definition of Fourier Transform” on page 9-2. Remember that $f(m, n)$ is equal to 1 within the rectangular region and 0 elsewhere. Use a binary image to represent $f(m, n)$.

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



- 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], 'InitialMagnification', 'fit');  
colormap(jet); colorbar
```



Discrete Fourier Transform Computed Without Padding

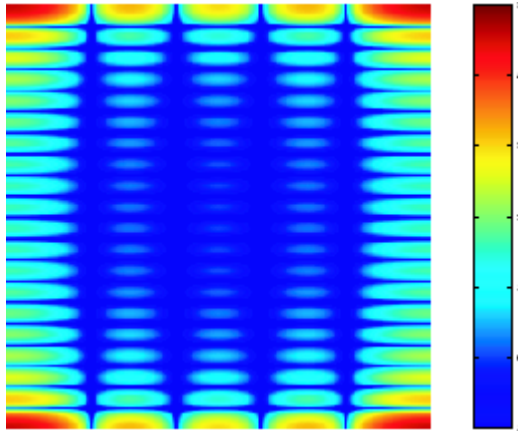
This plot differs from the Fourier transform displayed in “Visualizing the Fourier Transform” on page 9-3. First, the sampling of the Fourier transform is much coarser. Second, the zero-frequency coefficient is displayed in the upper left corner instead of the traditional location in the center.

- 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.

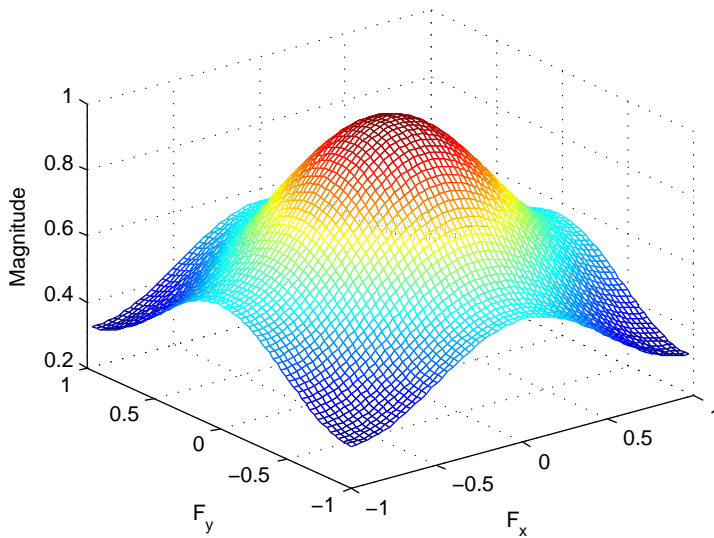
Applications of the Fourier Transform

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 “Designing Linear Filters in the Frequency Domain” on page 8-25 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`, multiply the two DFTs together, and compute the inverse two-dimensional DFT of the result using `ifft2`

```
C = ifft2(fft2(A).*fft2(B));
```

- 4 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)
```

This example produces the following result.

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`.

The following figure shows both the original image and the template.

```
imshow(bw);
figure, imshow(a);
```

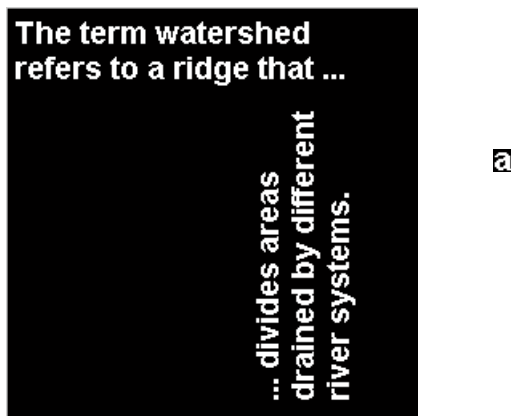


Image (left) and the Template to Correlate (right)

- 3 Compute the correlation of the template image with the original image by rotating the template image by 180° 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° .) 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.

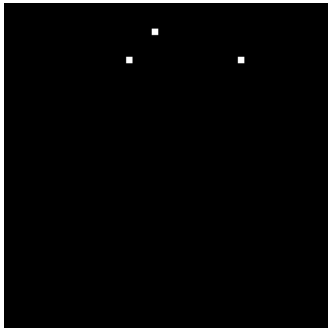
```
figure, imshow(C,[]) % Scale image to appropriate display range.
```



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 pixels over threshold.
```



Correlated, Thresholded Image Showing Template Locations

Discrete Cosine Transform

In this section...

“DCT Definition” on page 9-15

“The DCT Transform Matrix” on page 9-17

“DCT and Image Compression” on page 9-17

DCT Definition

The discrete cosine transform (DCT) represents an image as a sum of sinusoids of varying magnitudes and frequencies. The `dct2` function 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

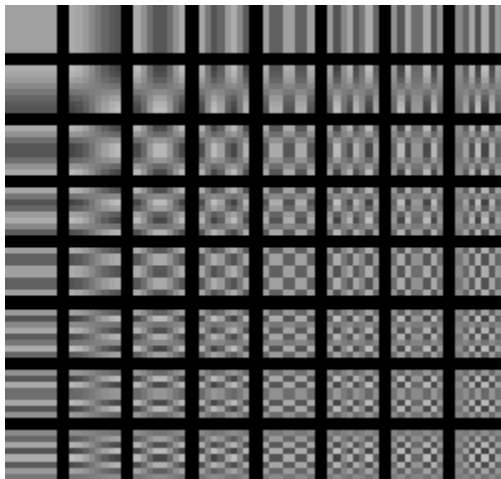
$$A_{mn} = \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} \alpha_p \alpha_q B_{pq} \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N}, \quad \begin{matrix} 0 \leq m \leq M-1 \\ 0 \leq n \leq N-1 \end{matrix}$$

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

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

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

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

There are two ways to compute the DCT using Image Processing Toolbox software. The first method is to use the `dct2` function. `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

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

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);
dct = @(block_struct) T * block_struct.data * T';
B = blockproc(I,[8 8],dct);
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 = blockproc(B,[8 8],@(block_struct) mask .* block_struct.data);
invdct = @(block_struct) T' * block_struct.data * T;
I2 = blockproc(B2,[8 8],invdct);
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.

Radon Transform

In this section...

“Radon Transformation Definition” on page 9-19

“Plotting the Radon Transform” on page 9-22

“Viewing the Radon Transform as an Image” on page 9-24

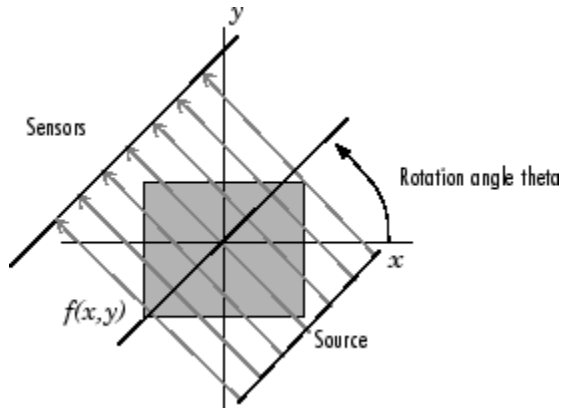
“Detecting Lines Using the Radon Transform” on page 9-25

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.

Radon Transformation Definition

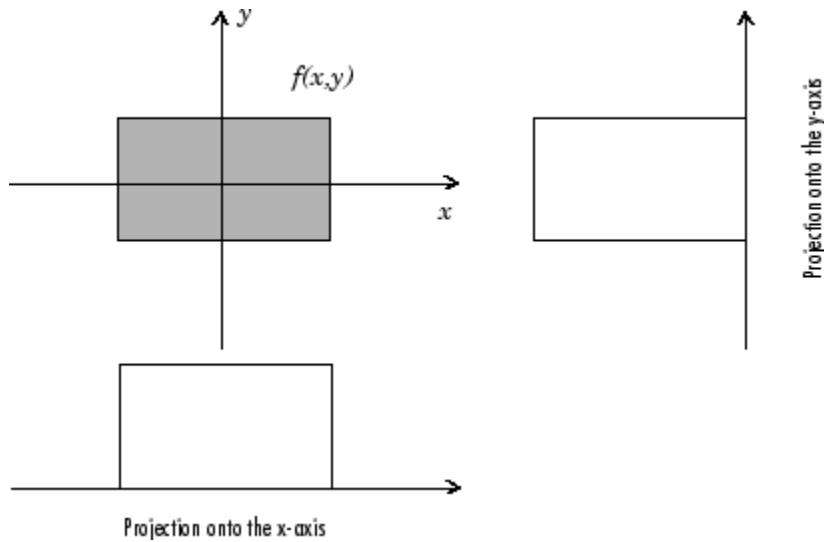
The `radon` function computes *projections* of an image matrix along specified directions.

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

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

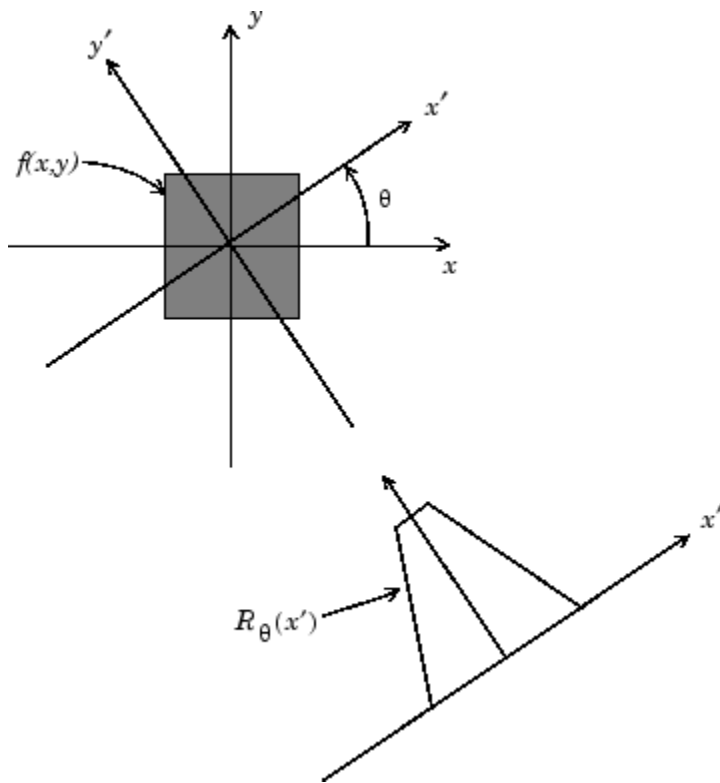
Projections can be computed along any angle $[\theta]$. In general, the Radon transform of $f(x,y)$ is the line integral of f parallel to the y' -axis

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

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

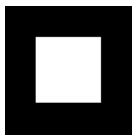
You can compute the Radon transform of an image I for the angles specified in the vector θ using the `radon` function with this syntax.

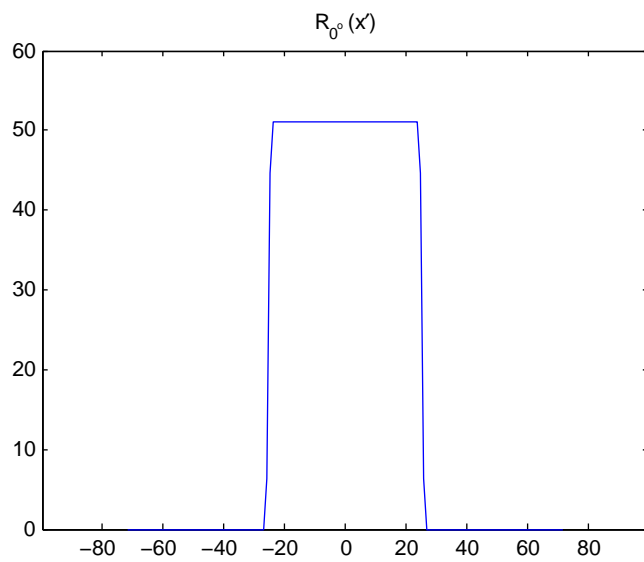
```
[R,xp] = radon(I,theta);
```

The columns of R contain the Radon transform for each angle in θ . The vector x_p contains the corresponding coordinates along the x' -axis. The center pixel of I is defined to be `floor((size(I)+1)/2)`; this is the pixel on the x' -axis corresponding to $x' = 0$.

The commands below compute and plot the Radon transform at 0° and 45° of an image containing a single square object. x_p is the same for all projection angles.

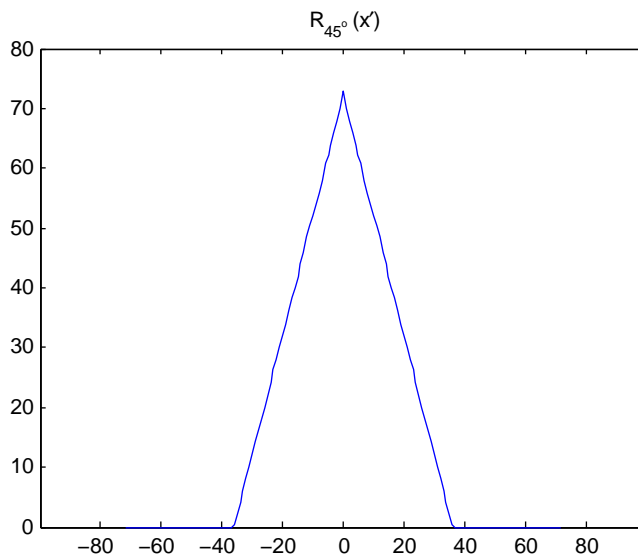
```
I = zeros(100,100);  
I(25:75, 25:75) = 1;  
imshow(I)  
[R,xp] = radon(I,[0 45]);  
figure; plot(xp,R(:,1)); title('R_{0^o} (x\prime)')
```





Radon Transform of a Square Function at 0 Degrees

```
figure; plot(xp,R(:,2)); title('R_{45^\circ}(x\prime)')
```

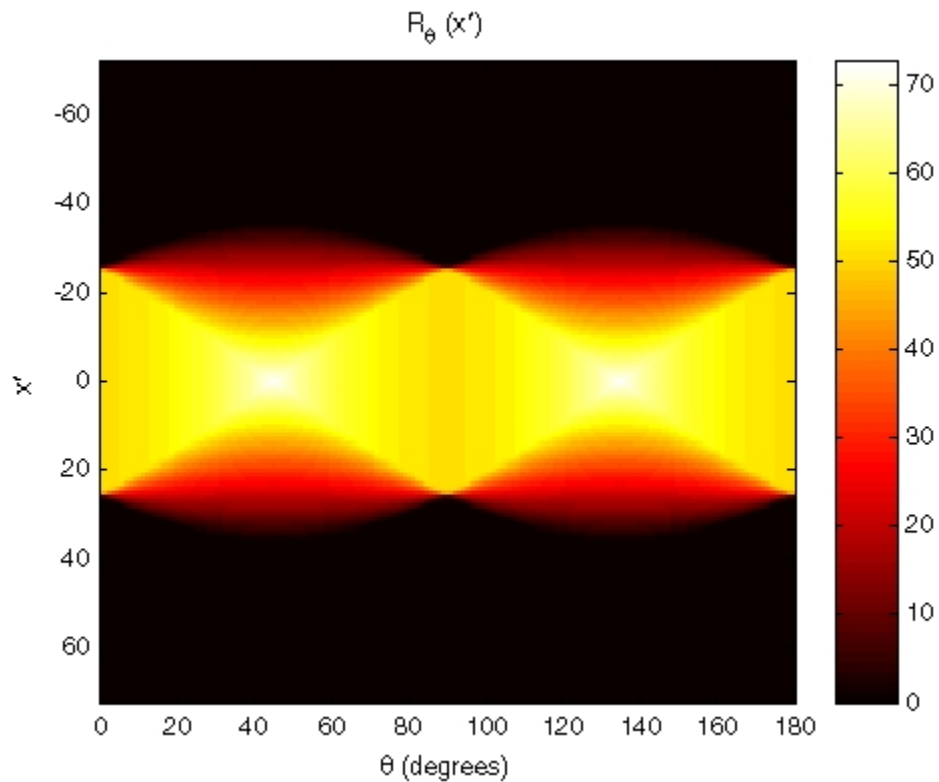


Radon Transform of a Square Function at 45 Degrees

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° to 180° , in 1° increments.

```
theta = 0:180;
[R, xp] = radon(I, theta);
imagesc(theta, xp, R);
title('R_{\theta} (X\prime)');
xlabel('\theta (degrees)');
ylabel('X\prime');
set(gca, 'XTick', 0:20:180);
colormap(hot);
colorbar
```



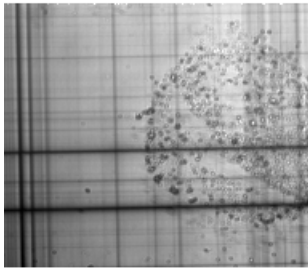
Radon Transform Using 180 Projections

Detecting Lines Using the Radon Transform

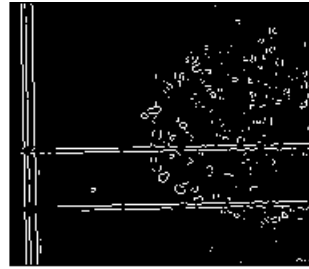
The Radon transform is closely related to a common computer vision operation known as the Hough transform. You can use the `radon` function to implement a form of the Hough transform used to detect straight lines. The steps are

- 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)
```



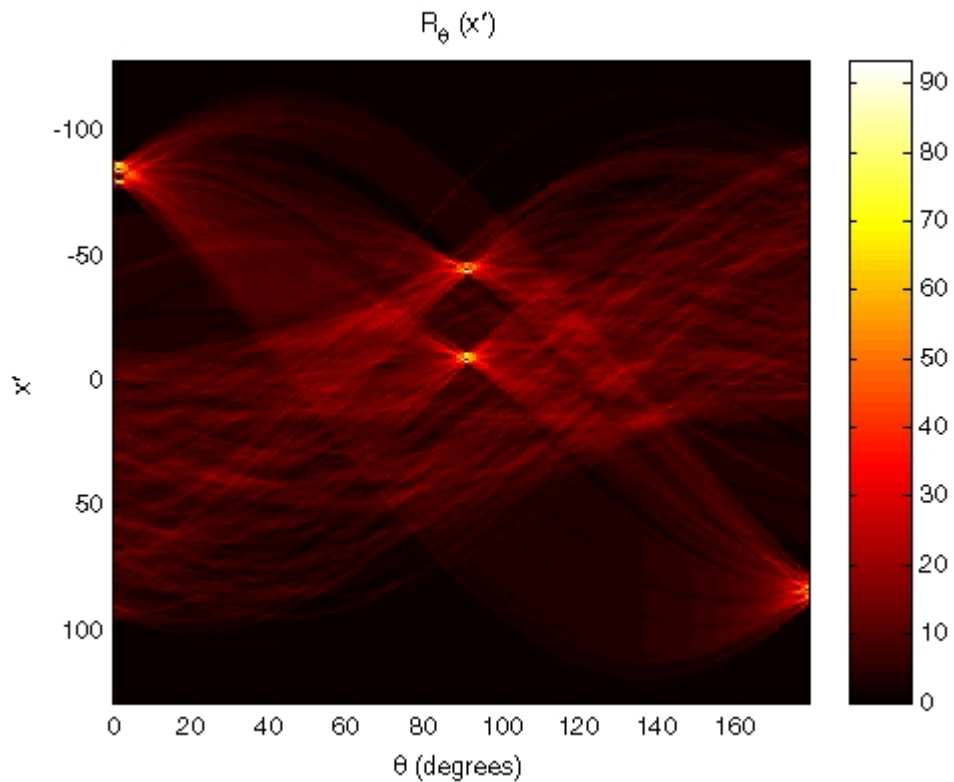
Original Image
Image Courtesy of Ann Walker



Edge Image

- 2 Compute the Radon transform of the edge image.

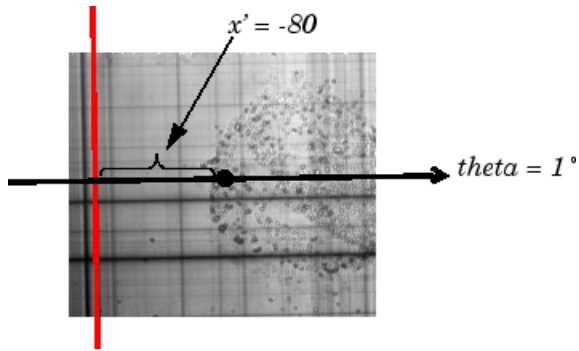
```
theta = 0:179;  
[R,xp] = radon(BW,theta);  
figure, imagesc(theta, xp, R); colormap(hot);  
xlabel('\theta (degrees)'); ylabel('x\prime');  
title('R_{\theta} (x\prime)');  
colorbar
```



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)

The Inverse Radon Transformation

In this section...

“Inverse Radon Transform Definition” on page 9-29

“Reconstructing an Image from Parallel Projection Data” on page 9-31

Inverse Radon Transform Definition

The `iradon` function inverts the Radon transform and can therefore be used to reconstruct images.

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 = radon(I, theta);
```

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

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

In the example above, projections are calculated from the original image `I`.

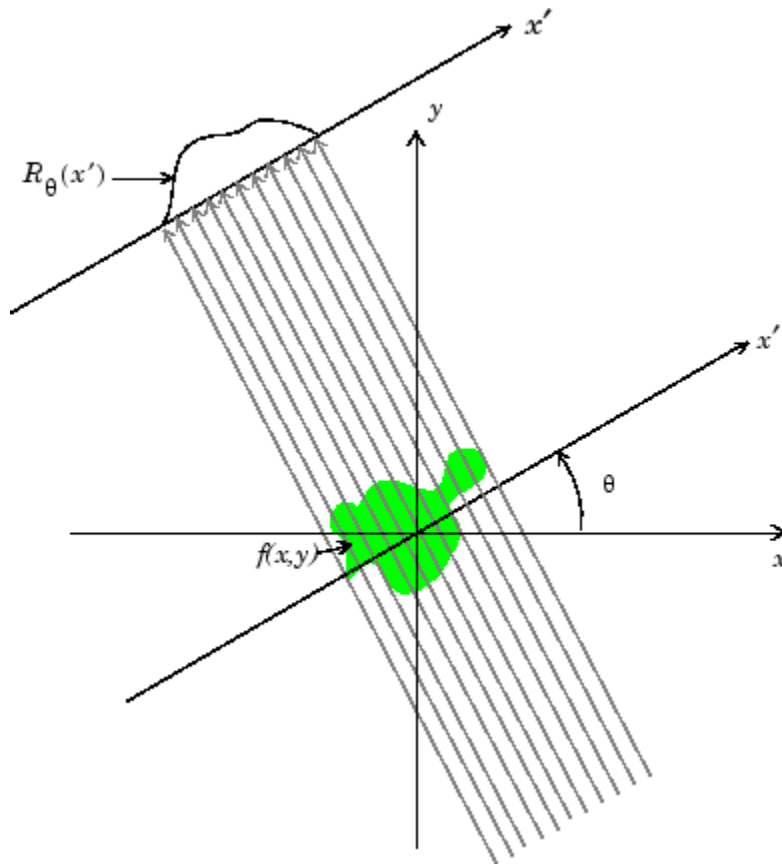
Note, however, that in most application areas, there is no original image from which projections are formed. For example, the inverse Radon transform is commonly used in tomography applications. In X-ray absorption tomography, projections are formed by measuring the attenuation of radiation that passes 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 `Dtheta`. When the scalar `Dtheta` 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. To get unfiltered backprojection data, specify `'none'` for the filter parameter.

```
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);
```

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 using the `phantom` function. 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)
```

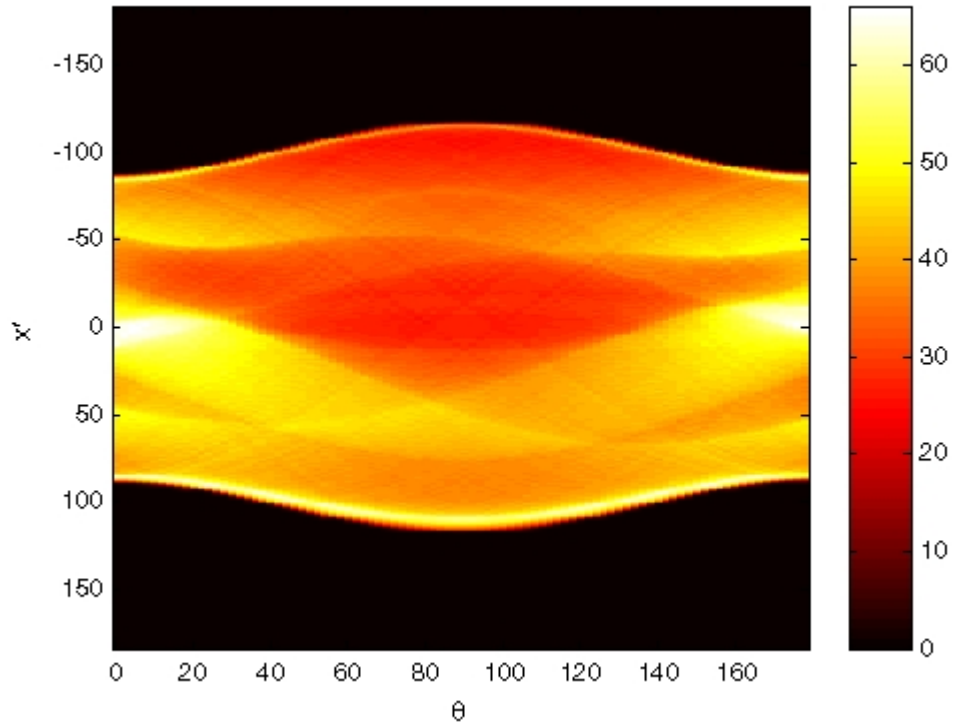


- 2 Compute the Radon transform of the phantom brain for three different sets of theta values. R1 has 18 projections, R2 has 36 projections, and R3 has 90 projections.

```
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

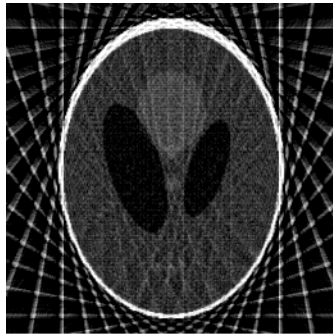
Note how some of the features of the input image appear in this image of the transform. The first column in the Radon transform corresponds to a projection at 0° that is integrating in the vertical direction. The centermost column corresponds to a projection at 90° , which is integrating in the horizontal direction. The projection at 90° has a wider profile than the projection at 0° due to the larger vertical semi-axis of the outermost ellipse of the phantom.

- 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)
```

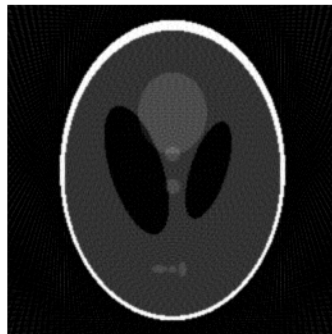
The following figure shows the results of all three reconstructions. Notice how image I1, which was reconstructed from only 18 projections, is the least accurate reconstruction. Image I2, which was reconstructed from 36 projections, is better, but it is still not clear enough to discern clearly the small 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

In this section...

“Fan-Beam Projection Data Definition” on page 9-35

“Computing Fan-Beam Projection Data” on page 9-36

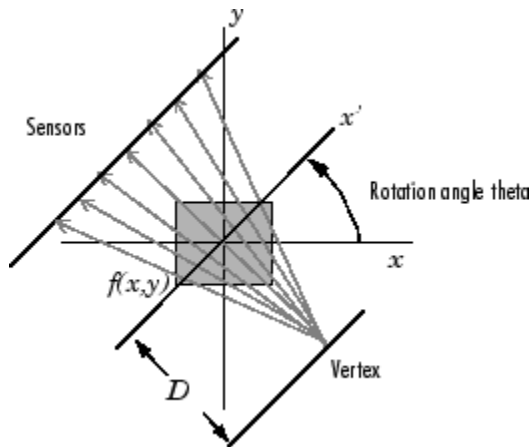
“Reconstructing an Image from Fan-Beam Projection Data” on page 9-38

“Reconstruct Image using Inverse Fanbeam Projection” on page 9-39

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.

Fan-Beam Projection Data Definition

The `fanbeam` function 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

Computing Fan-Beam Projection Data

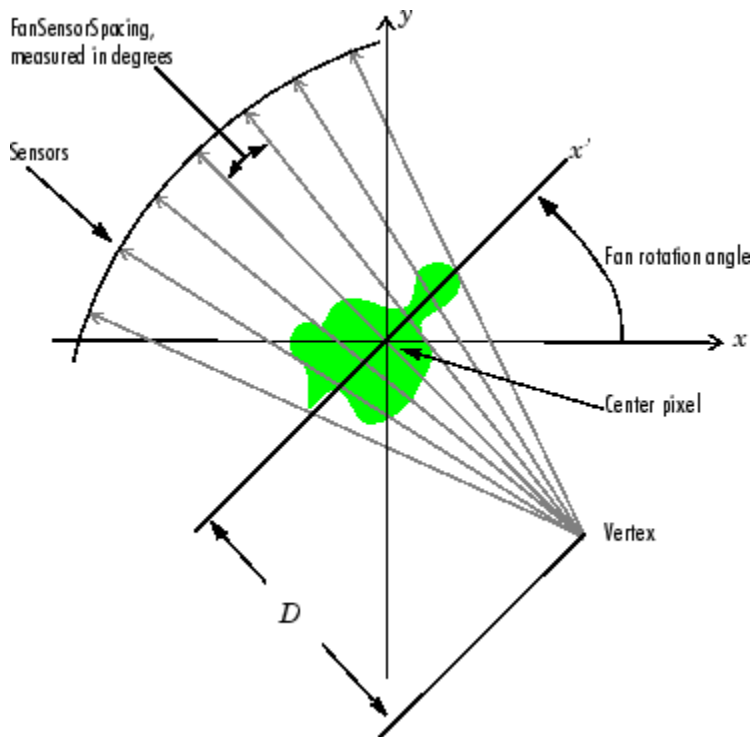
To compute fan-beam projection data, use the `fanbeam` function. You specify as arguments an image and the distance between the vertex of the fan-beam projections and the center of rotation (the center pixel in the image). The `fanbeam` function determines the number of beams, based on the size of the 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.

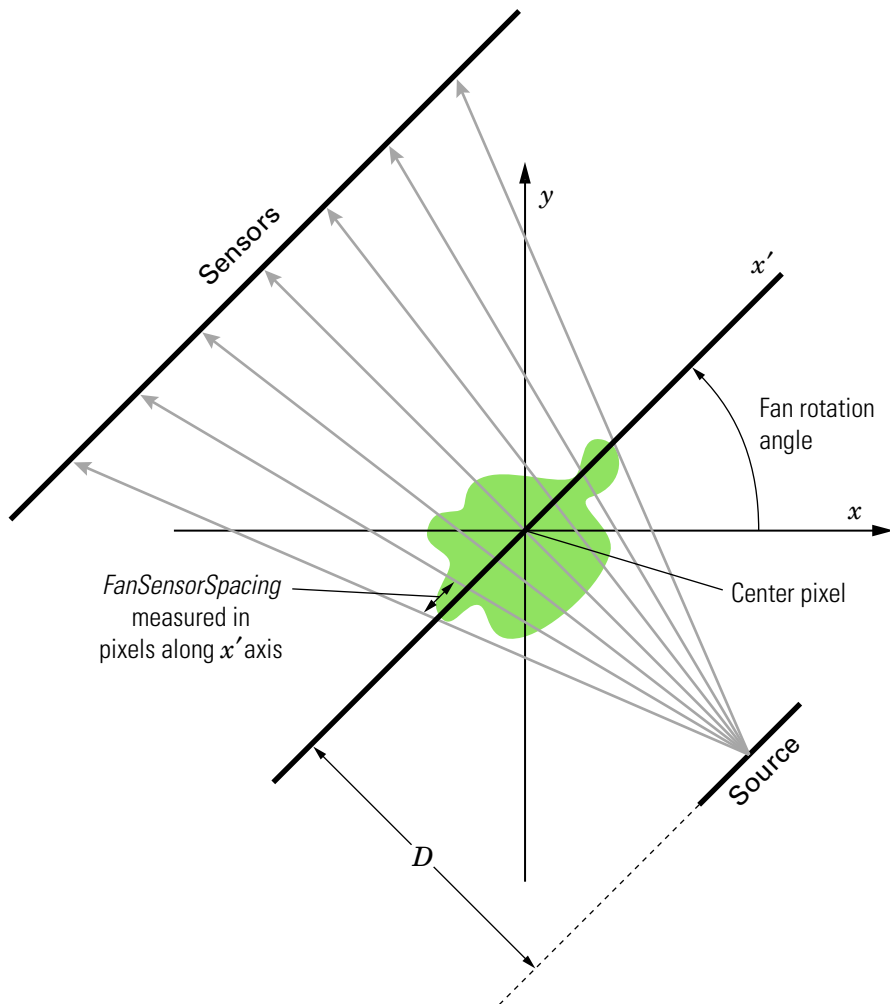
The following figures illustrate both these geometries. The first figure illustrates geometry used by the `fanbeam` function when `FanSensorGeometry` is set to `'arc'` (the

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 “The Inverse Radon Transformation” on page 9-29 for more information about the `iradon` function.

Reconstruct Image using Inverse Fanbeam Projection

This example shows how to use `fanbeam` and `ifanbeam` to form projections from a sample image and then reconstruct the image from the projections.

Generate a test image and display it. The test image is the Shepp-Logan head phantom, which can be generated by the `phantom` function. The phantom image illustrates many of the qualities that are found in real-world tomographic imaging of human heads.

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



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.

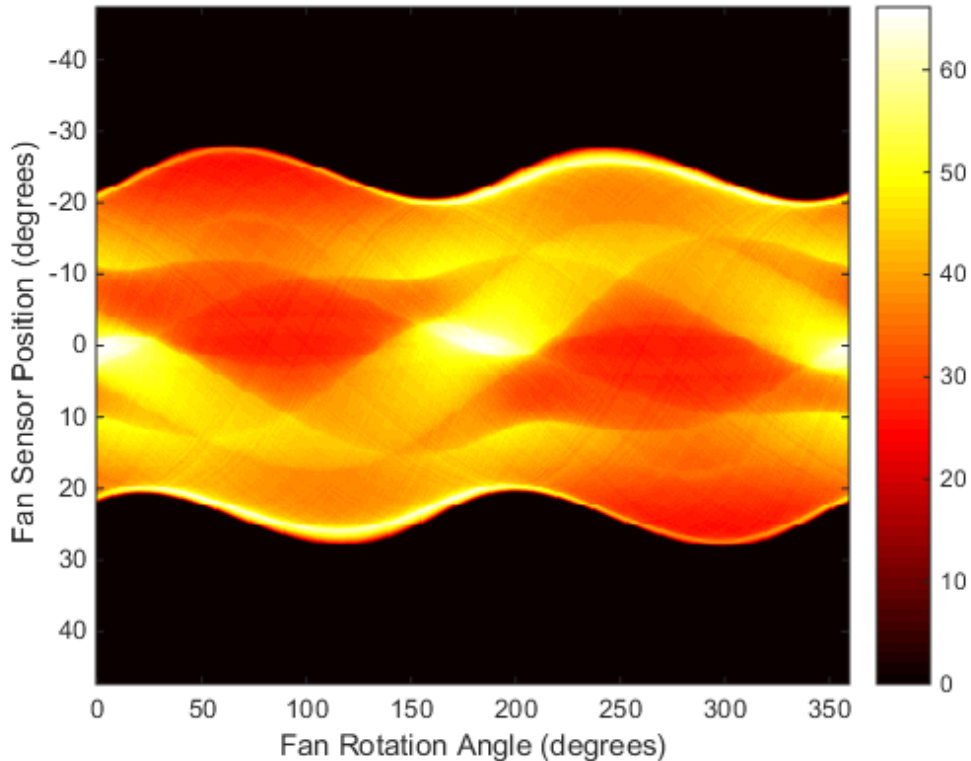
```
D = 250;
```

```
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);
```

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.

```
figure, imagesc(fan_rot_angles3, sensor_pos3, F3)  
colormap(hot); colorbar  
xlabel('Fan Rotation Angle (degrees)')  
ylabel('Fan Sensor Position (degrees)')
```



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 previously. 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`. In the output, note how the quality of the reconstruction gets better as the number of beams in the projection increases. The first image, `Ifan1`, was created using 2 degree spacing of the beams; the second image, `Ifan2`, was created using 1 degree spacing of the beams; the third image, `Ifan3`, was created using 0.25 spacing of the beams.

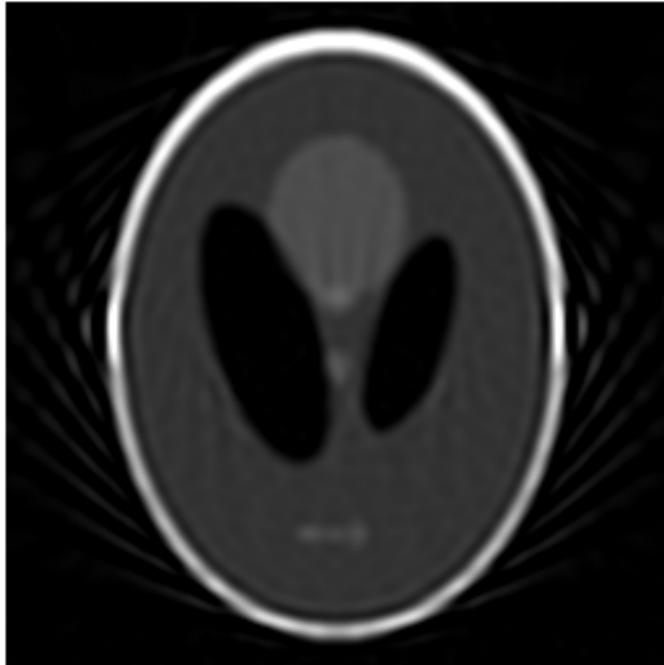
```
output_size = max(size(P));

Ifan1 = ifanbeam(F1,D, ...
    'FanSensorSpacing',dsensor1,'OutputSize',output_size);
figure, imshow(Ifan1)
```

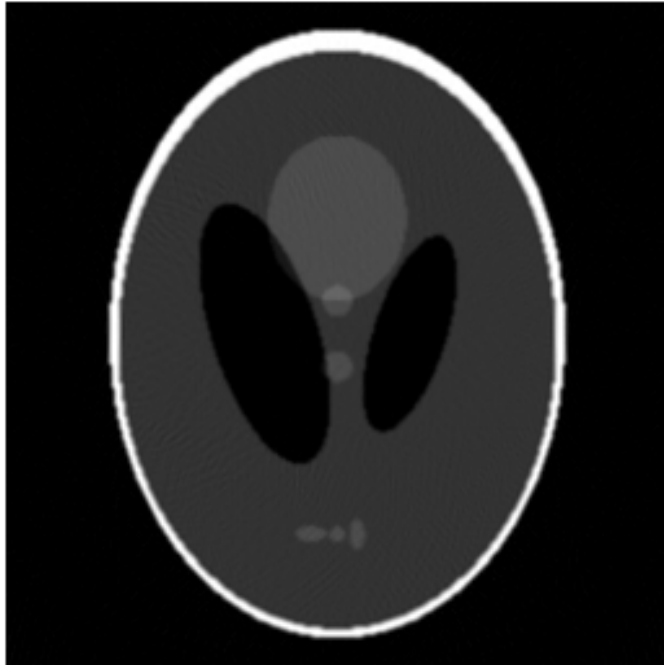
```
title('Ifan1')  
  
Ifan2 = ifanbeam(F2,D, ...  
    'FanSensorSpacing',dsensor2,'OutputSize',output_size);  
figure, imshow(Ifan2)  
title('Ifan2')  
  
Ifan3 = ifanbeam(F3,D, ...  
    'FanSensorSpacing',dsensor3,'OutputSize',output_size);  
figure, imshow(Ifan3)  
title('Ifan3')
```

Ifan1

Ifan2



Ifan3



Morphological Operations

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.

- “Morphology Fundamentals: Dilation and Erosion” on page 10-2
- “Morphological Reconstruction” on page 10-16
- “Distance Transform” on page 10-33
- “Labeling and Measuring Objects in a Binary Image” on page 10-35
- “Lookup Table Operations” on page 10-40

Morphology Fundamentals: Dilation and Erosion

In this section...

- “Understanding Dilation and Erosion” on page 10-2
- “Understanding Structuring Elements” on page 10-5
- “Dilating an Image” on page 10-8
- “Eroding an Image” on page 10-10
- “Combining Dilation and Erosion” on page 10-11
- “Dilation- and Erosion-Based Functions” on page 10-13

To see how morphological processing can solve an image processing problem, view the Image Processing Toolbox watershed segmentation example.

Understanding Dilation and Erosion

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. 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.

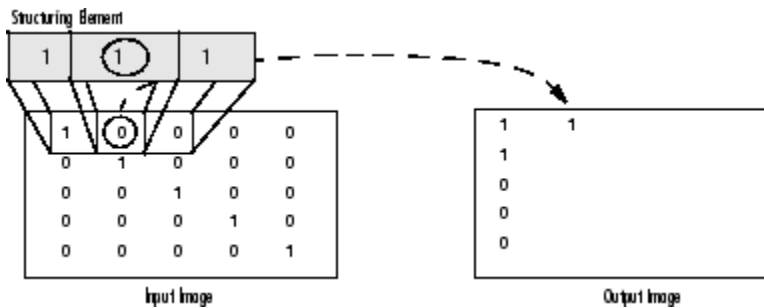
The most basic morphological operations are dilation and erosion. Dilation adds pixels to the boundaries of objects in an image, while erosion removes pixels on object boundaries. The number of pixels added or removed from the objects in an image depends on the size and shape of the *structuring element* used to process the image. 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.

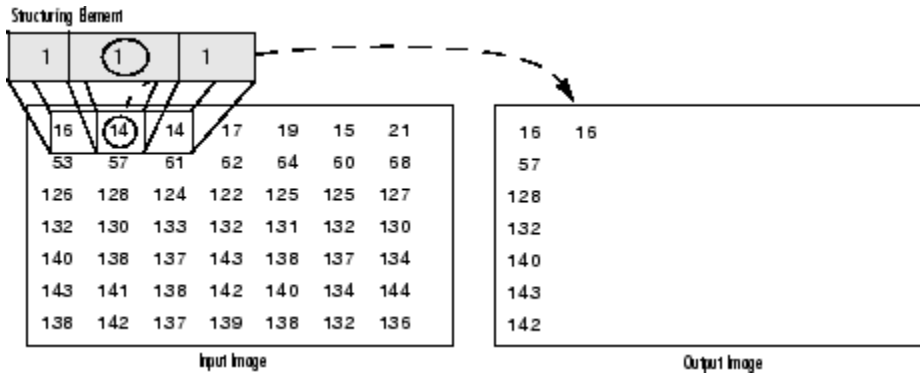
Operation	Rule
Erosion	The value of the output pixel is the <i>minimum</i> value of all the pixels in the input pixel's neighborhood. In a binary image, if any of the pixels is set to 0, the output pixel is set to 0.

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 “Understanding Structuring Elements” on page 10-5 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 <code>uint8</code> 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 <code>uint8</code> 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.

Understanding 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.

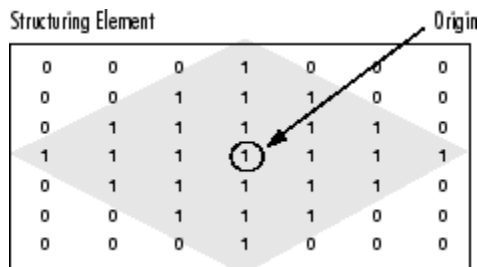
The Origin of a Structuring Element

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 matrix 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: `SHAPE` and `PACKOPT`. The `SHAPE` 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 “Understanding Structuring Elements” on page 10-5.)

```

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: `SHAPE`, `PACKOPT`, and `M`.

The `SHAPE` 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 “Understanding Structuring Elements” on page 10-5.)

```
SE = strel('arbitrary',eye(5));  
SE=
```

Flat STREL object containing 5 neighbors.

Neighborhood:

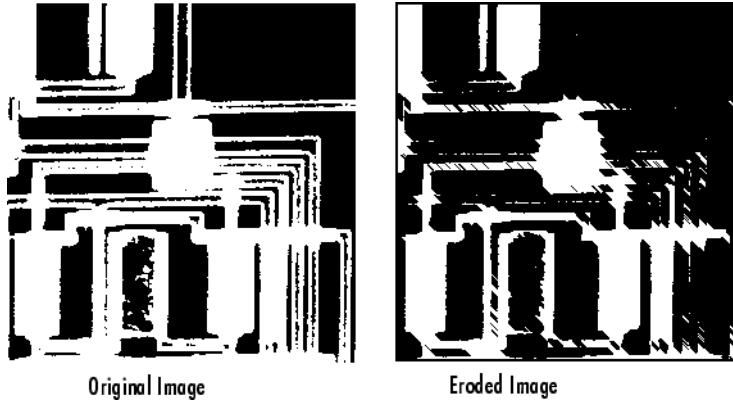
```
    1    0    0    0    0  
    0    1    0    0    0  
    0    0    1    0    0  
    0    0    0    1    0  
    0    0    0    0    1
```

- 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 10-13 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.

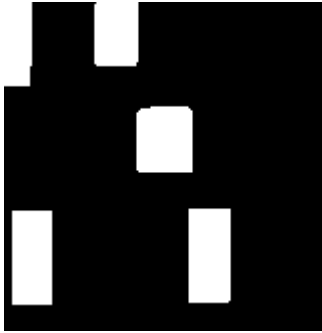
```
BW2 = imerode(BW1,SE);  
imshow(BW2)
```

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.

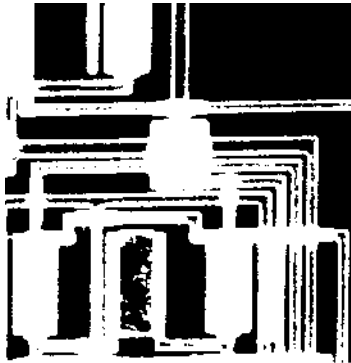
Dilation- and Erosion-Based Functions

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.
<code>imopen</code>	Erodes an image and then dilates the eroded image using the same structuring element for both operations.
<code>imtophat</code>	Subtracts a morphologically opened image from the original image. Can be used to enhance contrast in an image.

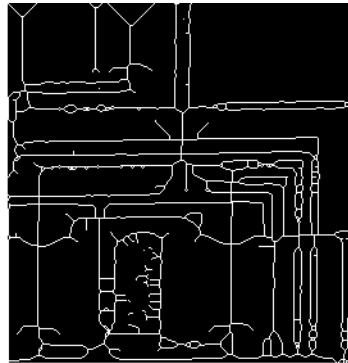
Skeletonization

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

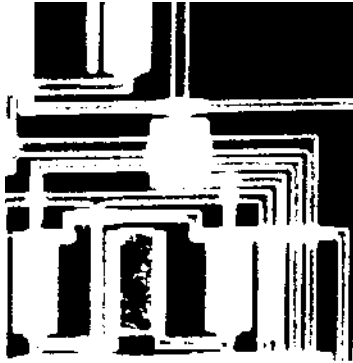
Perimeter Determination

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:

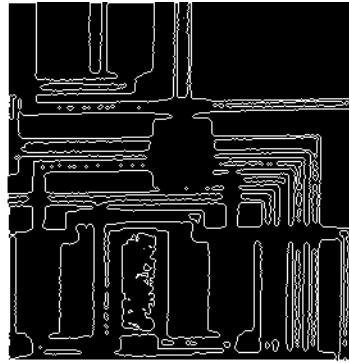
- The pixel is `on`.
- One (or more) of the pixels in its neighborhood is `off`.

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

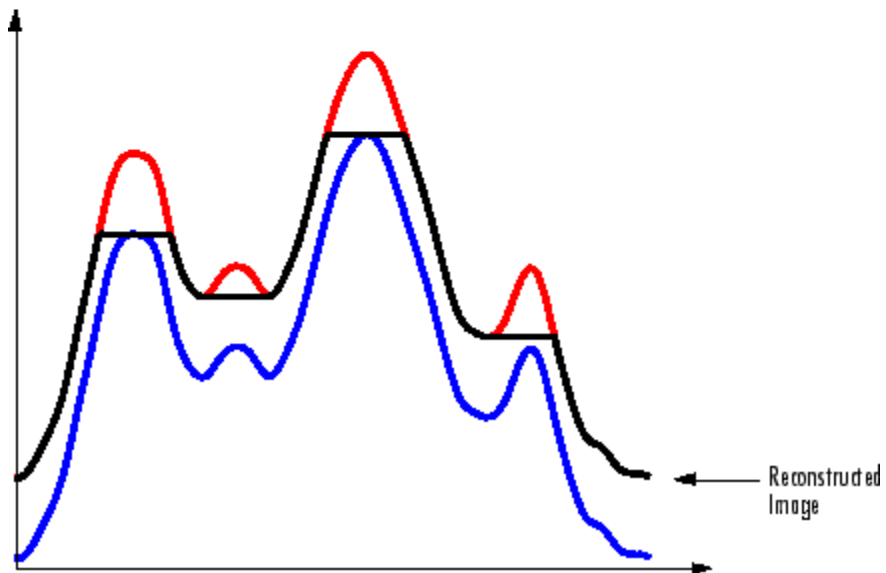
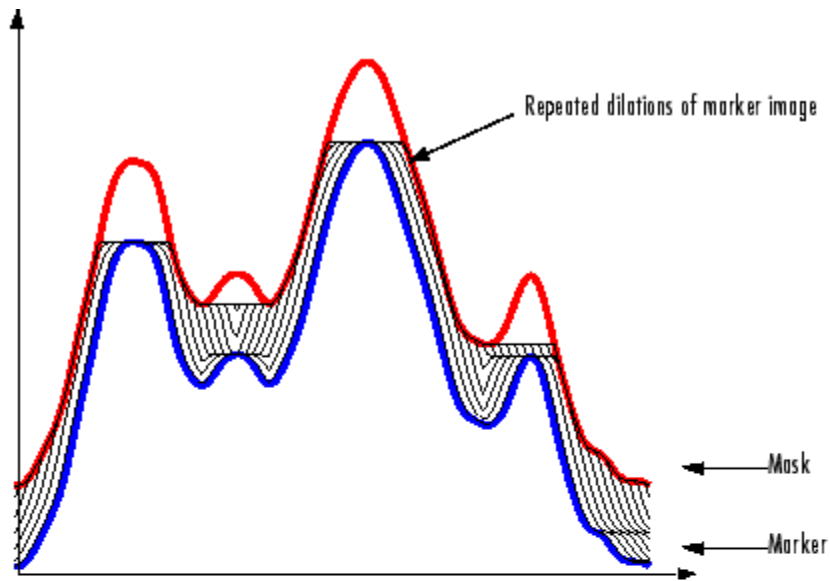
Morphological Reconstruction

In this section...
“Understanding Morphological Reconstruction” on page 10-16
“Understanding the Marker and Mask” on page 10-18
“Pixel Connectivity” on page 10-19
“Flood-Fill Operations” on page 10-22
“Finding Peaks and Valleys” on page 10-25

Understanding Morphological Reconstruction

Morphological reconstruction can be thought of conceptually as repeated dilations of an image, called the *marker image*, until the contour of the marker image fits under a second image, called the *mask image*. In morphological reconstruction, 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

Morphological reconstruction is based on morphological dilation, but note the following unique properties:

- Processing is based on two images, a marker and a mask, rather than one image and a structuring element.
- Processing is based on the concept of connectivity, rather than a structuring element.
- Processing repeats until stability; i.e., the image no longer changes.

Understanding the 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

```

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*.

Determining which pixels create a connected component depends on how pixel connectivity is defined. For example, this binary image contains one foreground object or two, depending on the connectivity. If the foreground is 4-connected, the image is all one object — there is no distinction between a foreground object and the background. However, if the foreground is 8-connected, the pixels set to 1 connect to form a closed loop

and the image has two separate objects: the pixels in the loop and the pixels outside the loop.

```

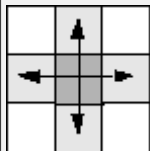
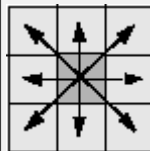
0   0   0   0   0   0   0   0
0   1   1   1   1   1   0   0
0   1   0   0   0   1   0   0
0   1   0   0   0   1   0   0
0   1   0   0   0   1   0   0
0   1   1   1   1   0   0   0
0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0
    
```

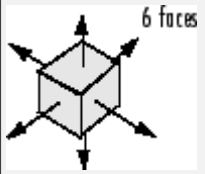
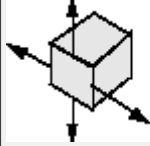
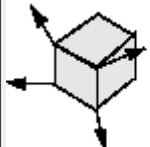
Defining Connectivity in an Image

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 10-21
- “Specifying Custom Connectivities” on page 10-21

Supported Connectivities

Two-Dimensional Connectivities		
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.	
Three-Dimensional Connectivities		

6-connected	Pixels are connected if their faces touch.	 6 faces
18-connected	Pixels are connected if their faces or edges touch.	 6 faces + 12 edges
26-connected	Pixels are connected if their faces, edges, or corners touch.	 6 faces + 12 edges + 8 corners

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 which can be used to break up an image into independent columns.


```

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.

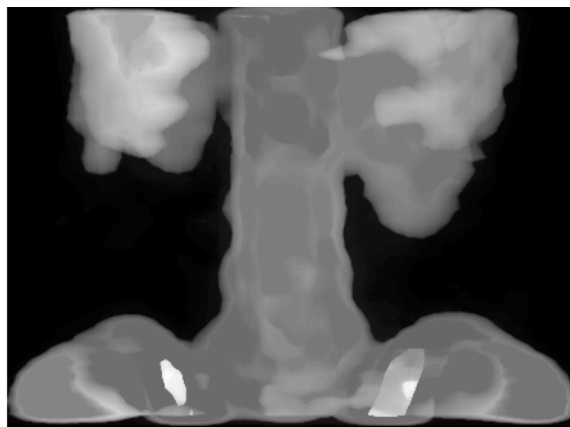
```

[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 10-25
- “Understanding the Maxima and Minima Functions” on page 10-26
- “Finding Areas of High or Low Intensity” on page 10-26
- “Suppressing Minima and Maxima” on page 10-28
- “Imposing a Minimum” on page 10-30

Terminology

This section uses the following terms.

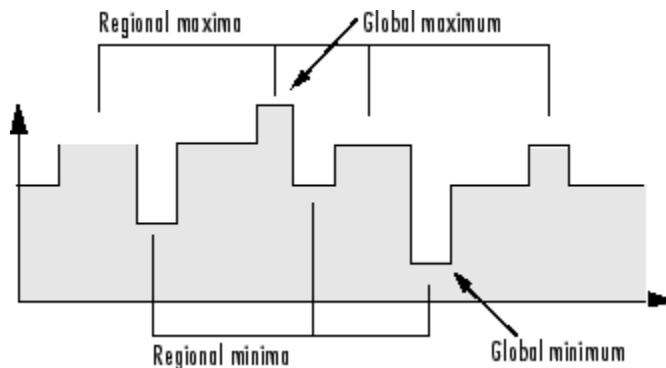
Term	Definition
global maxima	Highest regional maxima in the image. See the entry for regional maxima in this table for more information.
global minima	Lowest regional minima in the image. See the entry for regional minima in this table for more information.
regional maxima	Connected set of pixels of 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 .
regional minima	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

Term	Definition
	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 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.

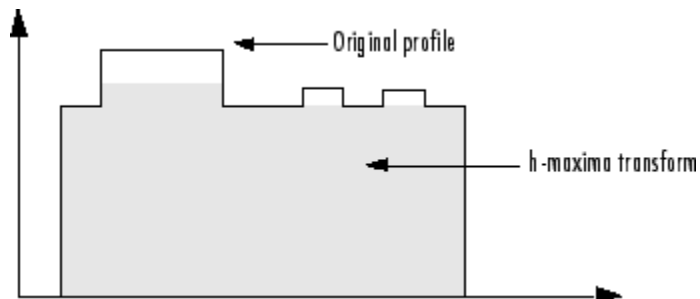
```
B = imhmax(A,2)
```

```

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     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
```

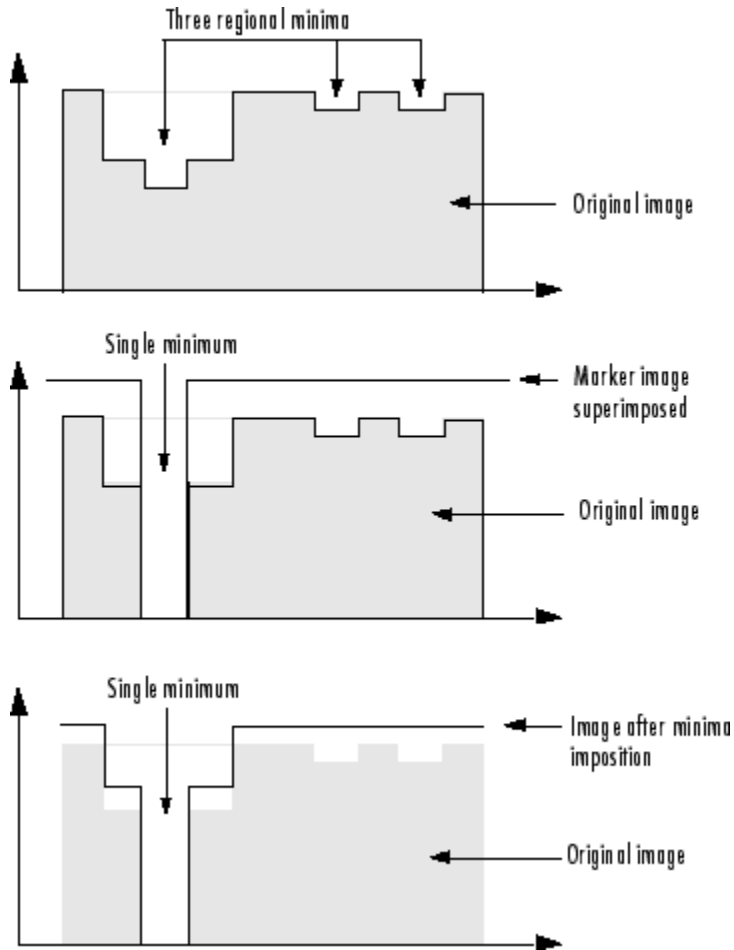
Applying the Marker Image to the Mask

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  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.



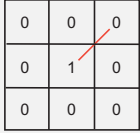

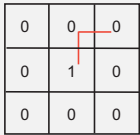

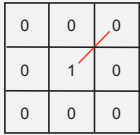

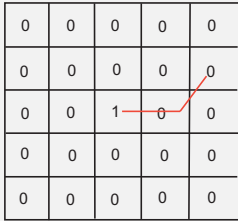

Imposing a Minimum

Distance Transform

The distance transform provides a metric or measure of the separation of points in the image. The `bwdist` function 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.	 Image  Distance Transform
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.	 Image  Distance Transform
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.	 Image  Distance Transform
Quasi-Euclidean	The quasi-Euclidean metric measures the total Euclidean distance along a set of horizontal, vertical, and diagonal line segments.	 Image  Distance Transform

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')
```



Labeling and Measuring Objects in a Binary Image

In this section...

“Understanding Connected-Component Labeling” on page 10-35

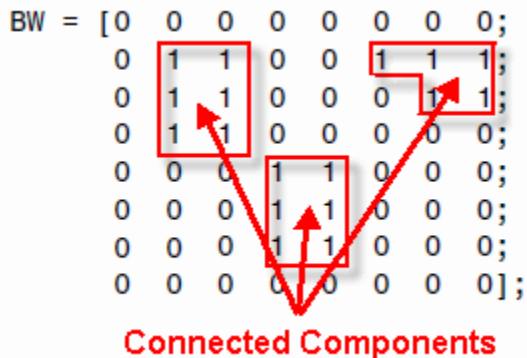
“Selecting Objects in a Binary Image” on page 10-37

“Finding the Area of the Foreground of a Binary Image” on page 10-38

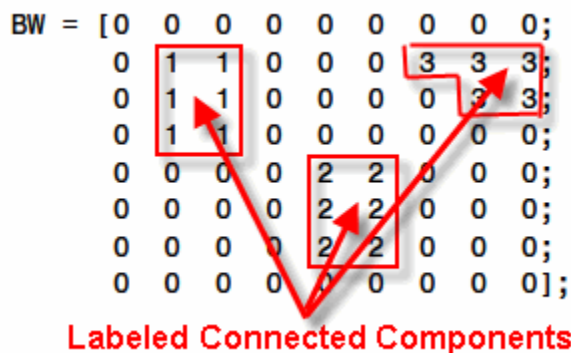
“Finding the Euler Number of a Binary Image” on page 10-38

Understanding Connected-Component Labeling

A connected component in a binary image is a set of pixels that form a connected group. For example, the binary image below has three connected components.



Connected component labeling is the process of identifying the connected components in an image and assigning each one a unique label, like this:



The matrix above is called a *label matrix*.

`bwconncomp` computes connected components, as shown in the example:

```

cc = bwconncomp(BW)
cc =

    Connectivity: 8
    ImageSize: [8 9]
    NumObjects: 3
    PixelIdxList: {[6x1 double] [6x1 double] [5x1 double]}

```

The `PixelIdxList` identifies the list of pixels belonging to each connected component.

For visualizing connected components, it is useful to construct a label matrix. Use the `labelmatrix` function. To inspect the results, display the label matrix as a pseudo-color image using `label2rgb`.

Construct a label matrix:

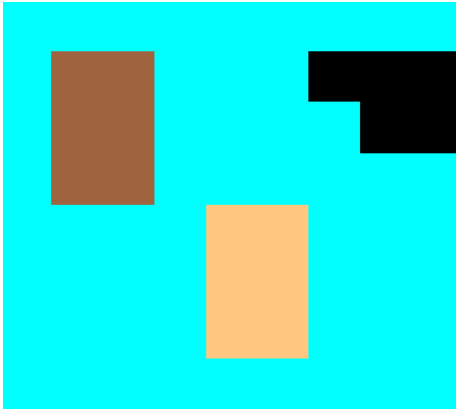
```
labeled = labelmatrix(cc);
```

Create a pseudo-color image, where the label identifying each object in the label matrix maps to a different color in the associated `colormap` matrix. Use `label2rgb` to choose the colormap, the background color, and how objects in the label matrix map to colors in the colormap:

```

RGB_label = label2rgb(labeled, @copper, 'c', 'shuffle');
imshow(RGB_label, 'InitialMagnification', 'fit')

```



Remarks

The functions `bwlabel`, `bwlabeln`, and `bwconncomp` all compute connected components for binary images. `bwconncomp` replaces the use of `bwlabel` and `bwlabeln`. It uses significantly less memory and is sometimes faster than the older functions.

Function	Input Dimension	Output Form	Memory Use	Connectivity
<code>bwlabel</code>	2-D	Double-precision label matrix	High	4 or 8
<code>bwlabeln</code>	N-D	Double-precision label matrix	High	Any
<code>bwconncomp</code>	N-D	CC struct	Low	Any

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

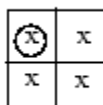
In this section...

“Creating a Lookup Table” on page 10-40

“Using a Lookup Table” on page 10-40

Creating a Lookup Table

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



2-by-2 neighborhood



3-by-3 neighborhood

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

Note `makelut` and `applylut` support only 2-by-2 and 3-by-3 neighborhoods. Lookup tables larger than 3-by-3 neighborhoods are not practical. 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)
```

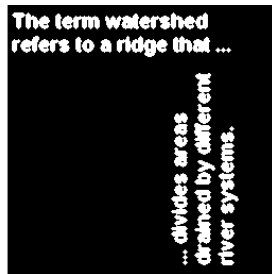
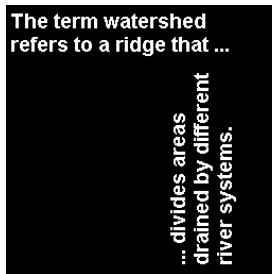


Image Before and After Applying Lookup Table Operation

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 functions that support a range of standard image processing operations for analyzing and enhancing images.

- “Pixel Values” on page 11-2
- “Intensity Profile of Images” on page 11-4
- “Contour Plot of Image Data” on page 11-8
- “Create Image Histogram” on page 11-11
- “Image Mean, Standard Deviation, and Correlation Coefficient” on page 11-14
- “Edge Detection” on page 11-15
- “Corner Detection” on page 11-18
- “Boundary Tracing in Images” on page 11-21
- “Hough Transform” on page 11-28
- “Quadtree Decomposition” on page 11-33
- “Texture Analysis” on page 11-35
- “Gray-Level Co-Occurrence Matrix (GLCM)” on page 11-38
- “Contrast Adjustment” on page 11-43
- “Noise Removal” on page 11-55
- “Image Segmentation” on page 11-61
- “Image Segmentation Using the Color Thesholder App” on page 11-62
- “Image Quality Metrics” on page 11-80
- “Image Segmentation Using the Image Segmenter App” on page 11-85
- “Image Region Properties” on page 11-99
- “Calculate Region Properties Using Image Region Analyzer” on page 11-100
- “Filter Images on Region Properties Using Image Region Analyzer App” on page 11-108

Pixel Values

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 “Get Pixel Information in Image Viewer App” on page 4-30.

Determine Values of Individual Pixels in Images

This example shows how to use `impixel` interactively to get pixel values.

Display an image.

```
imshow canoe.tif
```

Call `impixel`. When called with no input arguments, `impixel` associates itself with the image in the current axes.

```
pixel_values = impixel
```

Select the points you want to examine in the image by clicking the mouse. `impixel` places a star at each point you select.

```
imshow canoe.tif
```



When you are finished selecting points, press **Return**. `impixel` returns the pixel values in an n -by-3 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
```

Intensity Profile of Images

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. 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 or interactively using a mouse. For points that do not fall on the center of a pixel, the intensity values are interpolated. By default, `improfile` uses nearest-neighbor interpolation, but you can specify a different method. For more information, see “Specify the Interpolation Method” on page 6-3.) `improfile` works best with grayscale and truecolor images.

Create an Intensity Profile of an Image

This example shows how to create an intensity profile for an image interactively using `improfile`.

Read an image and display it.

```
I = fitsread('solarspectra.fts');  
imshow(I,[]);
```

Create the intensity profile. Call `improfile` with no arguments. The cursor changes to crosshairs when you move it over the displayed image. Using the mouse, 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**. In the following figure, the line is shown in red.

```
improfile
```

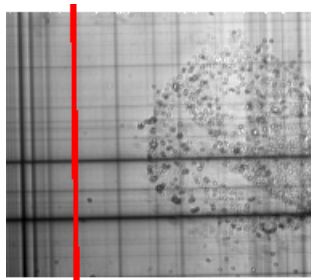
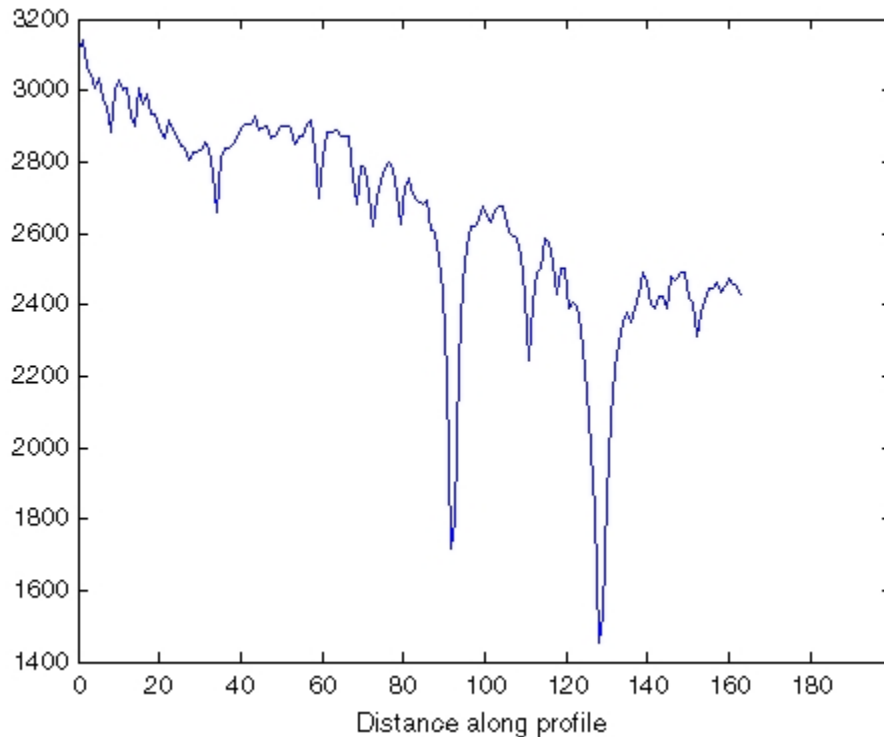


Image Courtesy of Ann Walker

After you finish drawing the line over the image, `improfile` displays a plot of the data along the line. Notice how the peaks and valleys in the plot correspond to the light and dark bands in the image.



Create Intensity Profile of an RGB Image

This example shows how to plot the intensity values in an RGB image. 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.

Display an RGB image using `imshow`.

```
imshow peppers.png
```

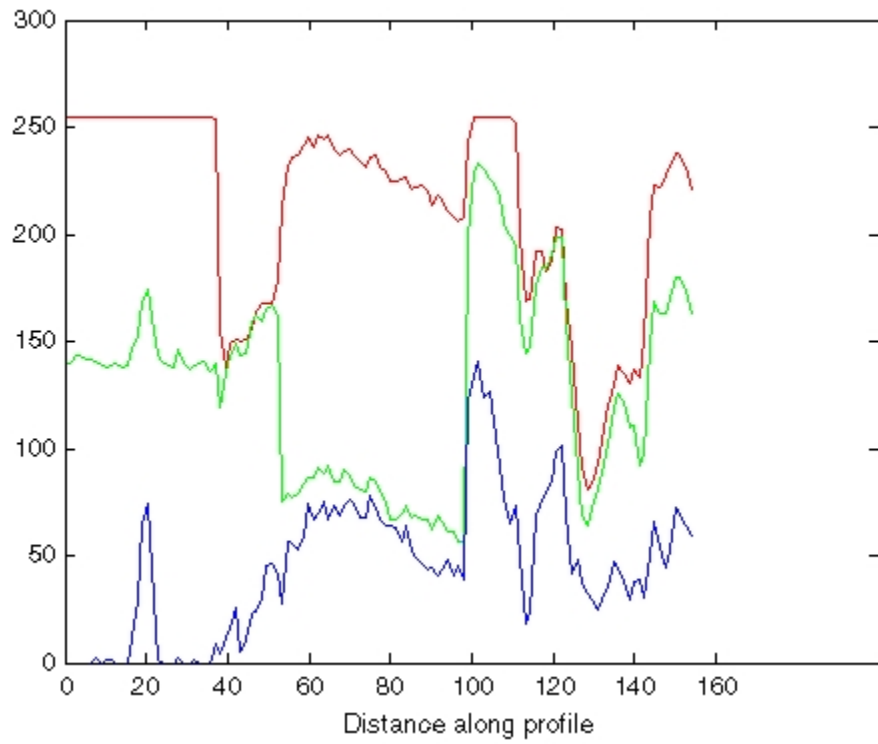
Call `improfile` without any arguments and trace a line segment in the image interactively. In the figure, the black line indicates a line segment drawn from top to bottom. Double-click to end the line segment

```
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

Contour Plot of Image Data

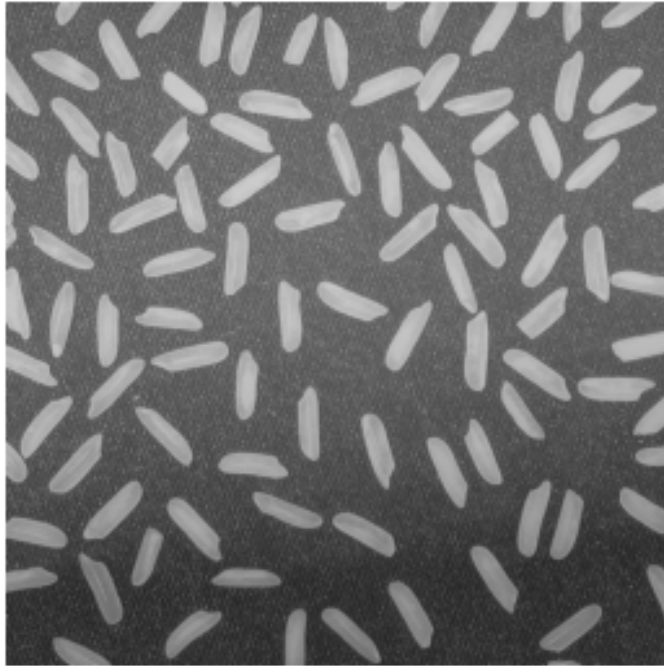
A contour is a path in an image along which the image intensity values are equal to a constant. You can create a contour plot of the data in a grayscale image using `imcontour`. This function is similar to the `contour` function in MATLAB, but it automatically sets up the axes so their orientation and aspect ratio match the image. To label the levels of the contours, use the `clabel` function.

Create Contour Plot of Image Data

This example shows how to create a contour plot of an image.

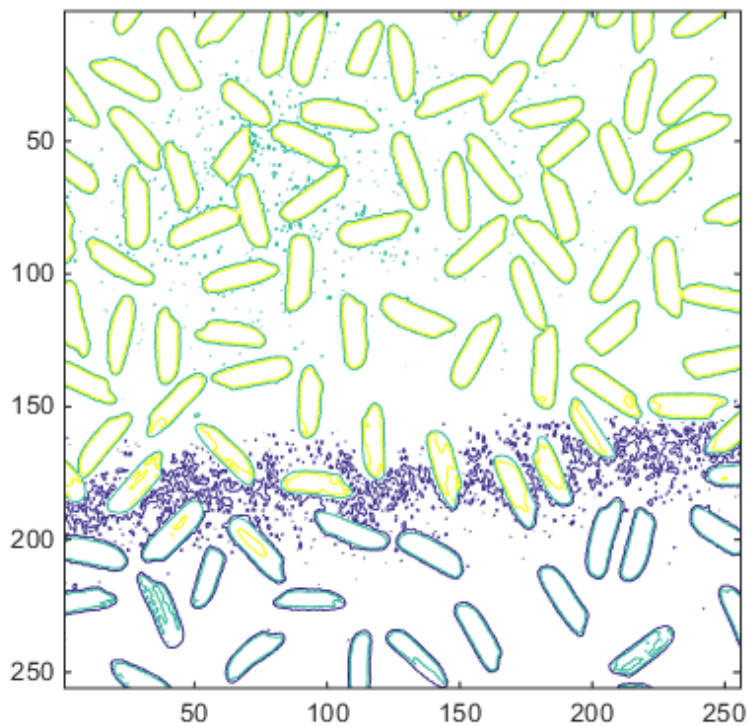
Read grayscale image and display it. The example uses an example image of grains of rice.

```
I = imread('rice.png');  
imshow(I)
```



Create a contour plot of the image using `imcontour` .

```
figure;  
imcontour(I,3)
```

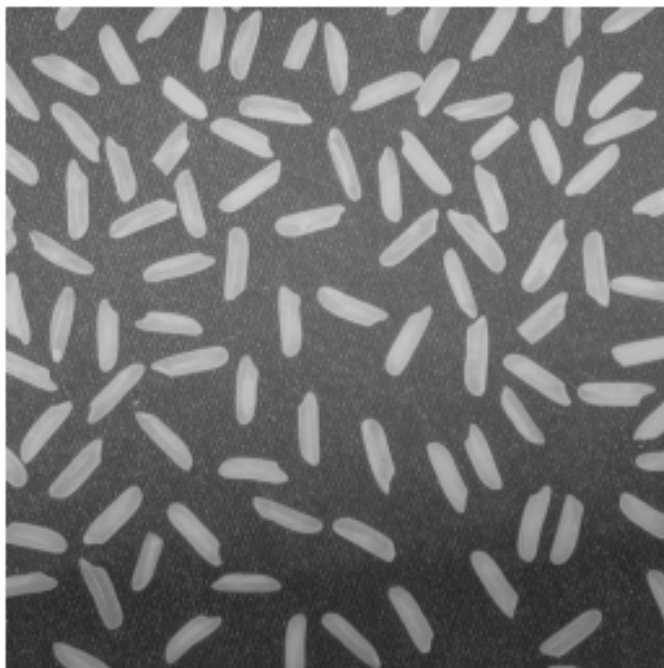


Create Image Histogram

This example shows how to create a histogram for an image using the `imhist` function. An image histogram is a chart that shows the distribution of intensities in an indexed or grayscale image. The `imhist` function creates a histogram plot by defining n equally spaced bins, each representing a range of data values, and then calculating the number of pixels within each range. 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.

Read an image into the workspace and display it.

```
I = imread('rice.png');  
imshow(I)
```



Create the histogram. For the example image, showing grains of rice, `imhist` creates a histogram with 64 bins. The `imhist` function displays the histogram, by default. The histogram shows a peak at around 100, corresponding to the dark gray background in the image.

```
figure;  
imhist(I);
```

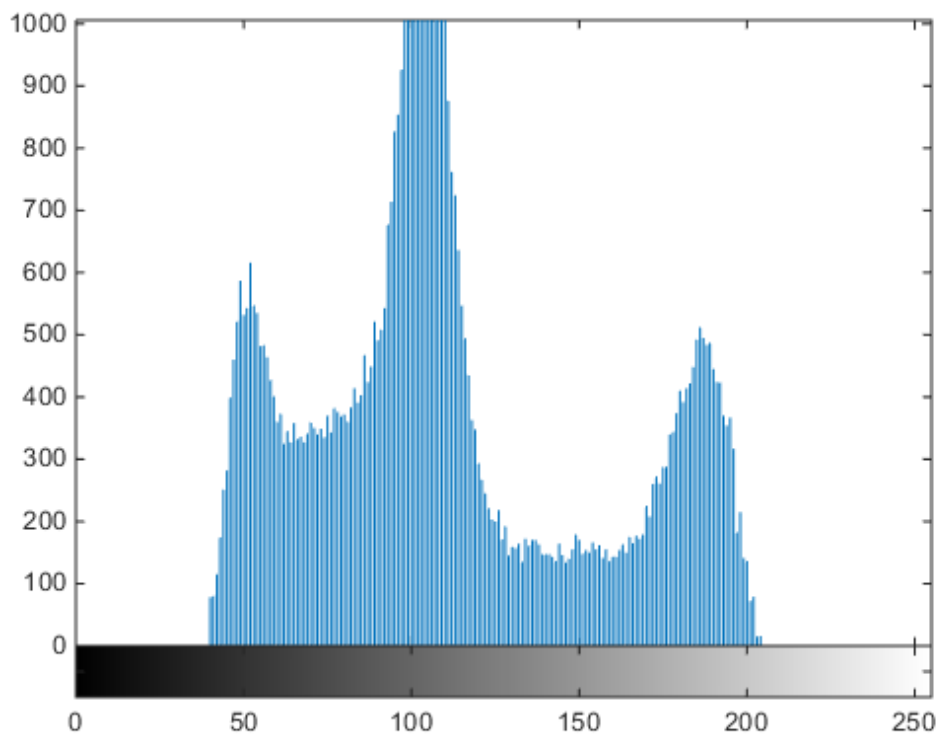


Image Mean, Standard Deviation, and Correlation Coefficient

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.

Edge Detection

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.

Detect Edges in Images

This example shows how to detect edges in an image using both the Canny edge detector and the Sobel edge detector.

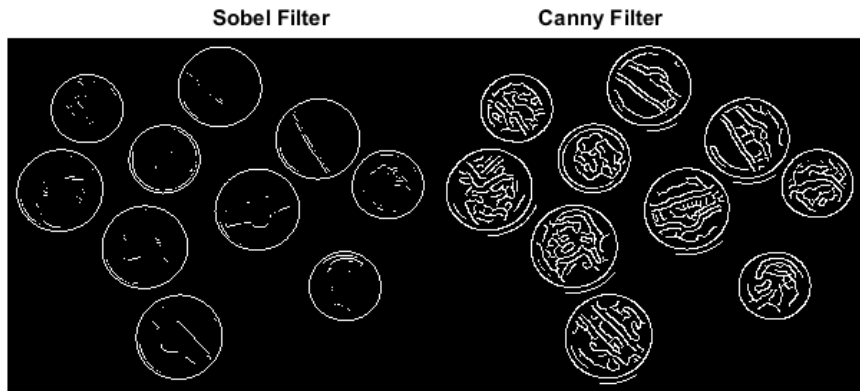
Read image and display it.

```
I = imread('coins.png');  
imshow(I)
```



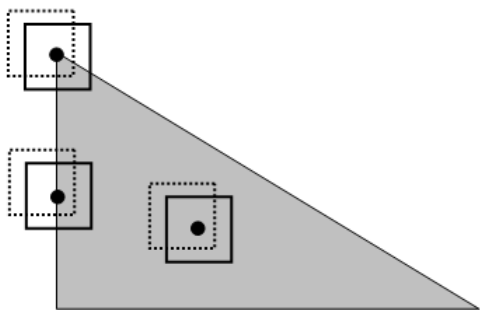
Apply both the Sobel and Canny edge detectors to the image and display them for comparison.

```
BW1 = edge(I, 'sobel');  
BW2 = edge(I, 'canny');  
figure;  
imshowpair(BW1, BW2, 'montage')  
title('Sobel Filter' 'Canny Filter');
```



Corner Detection

Corners are the most reliable feature you can use to find the correspondence between images. The following diagram shows three pixels—one inside the object, one on the edge of the object, and one on the corner. If a pixel is inside an object, its surroundings (solid square) correspond to the surroundings of its neighbor (dotted square). This is true for neighboring pixels in all directions. If a pixel is on the edge of an object, its surroundings differ from the surroundings of its neighbors in one direction, but correspond to the surroundings of its neighbors in the other (perpendicular) direction. A corner pixel has surroundings different from all of its neighbors in all directions.



The `corner` function identifies corners in an image. Two methods are available—the Harris corner detection method (the default) and Shi and Tomasi's minimum eigenvalue method. Both methods use algorithms that depend on the eigenvalues of the summation of the squared difference matrix (SSD). The eigenvalues of an SSD matrix represent the differences between the surroundings of a pixel and the surroundings of its neighbors. The larger the difference between the surroundings of a pixel and those of its neighbors, the larger the eigenvalues. The larger the eigenvalues, the more likely that a pixel appears at a corner.

Find Corners in Images

This example shows how to locate corners with the `corner` function and adjust your results by refining the maximum number of desired corners.

Create a checkerboard image.

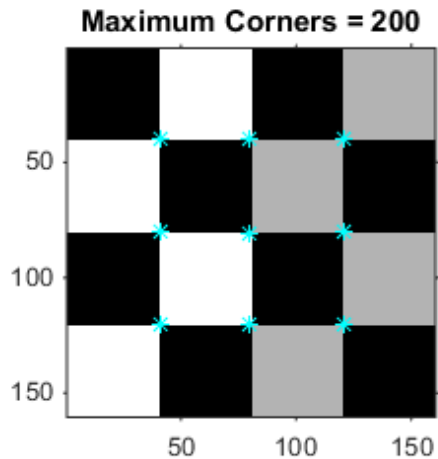
```
I = checkerboard(40,2,2);
```

Find the corners in the image.

```
C = corner(I);
```

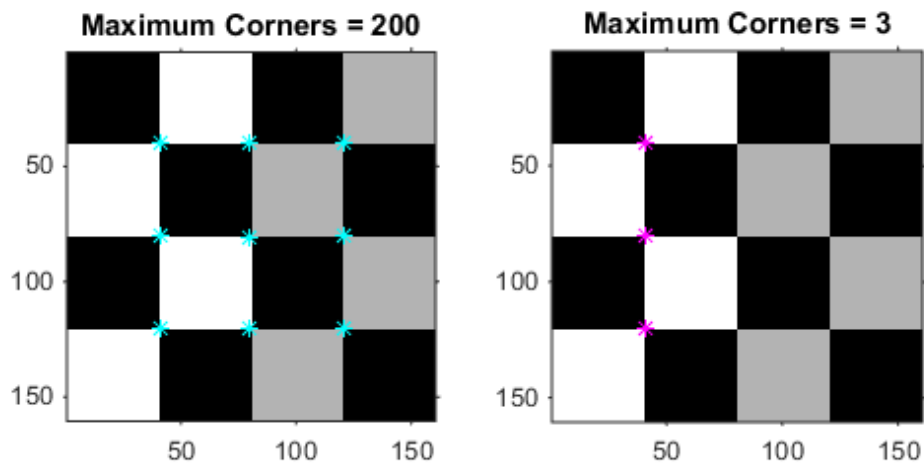
Display the corners when the maximum number of desired corners is the default setting of 200.

```
subplot(1,2,1);  
imshow(I);  
hold on  
plot(C(:,1), C(:,2), '*', 'Color', 'c')  
title('Maximum Corners = 200')  
hold off
```



Display the corners when the maximum number of desired corners is 3.

```
corners_max_specified = corner(I,3);  
subplot(1,2,2);  
imshow(I);  
hold on  
plot(corners_max_specified(:,1), corners_max_specified(:,2), ...  
      '*', 'Color', 'm')  
title('Maximum Corners = 3')  
hold off
```



Boundary Tracing in Images

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.

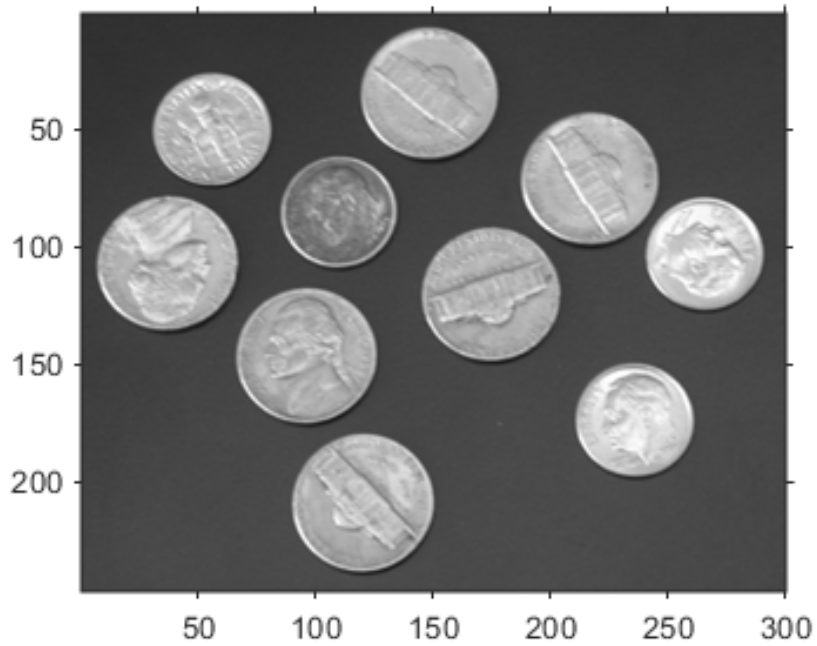
For both functions, the nonzero pixels in the binary image belong to an object, and pixels with the value 0 (zero) constitute the background.

Trace Boundaries of Objects in Images

This example shows how to trace the border of an object in a binary image using `bwtraceboundary`. Then, using `bwboundaries`, the example traces the borders of all the objects in the image.

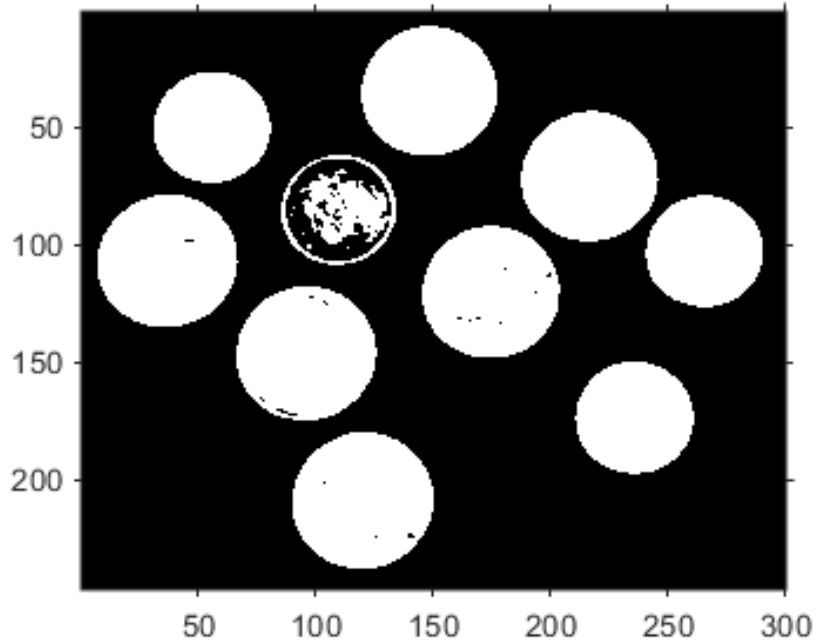
Read image and display it.

```
I = imread('coins.png');  
imshow(I)
```



Convert the image to a binary image. `bwtraceboundary` and `bwboundaries` only work with binary images.

```
BW = im2bw(I);  
imshow(BW)
```

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)))
```

```
dim =
    246    300
```

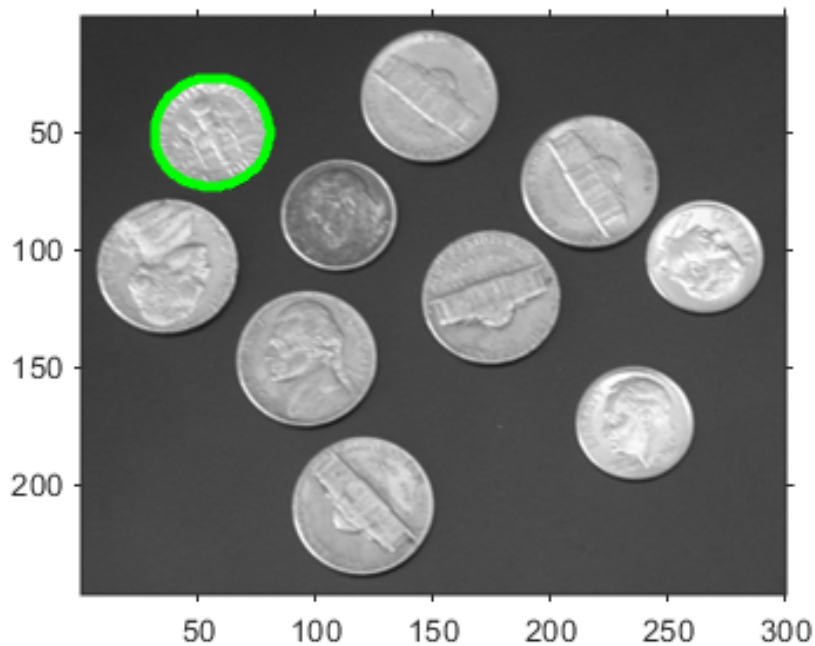
```
row =
    27
```

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').

```
boundary = bwtraceboundary(BW,[row, col], 'N');
```

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);
```

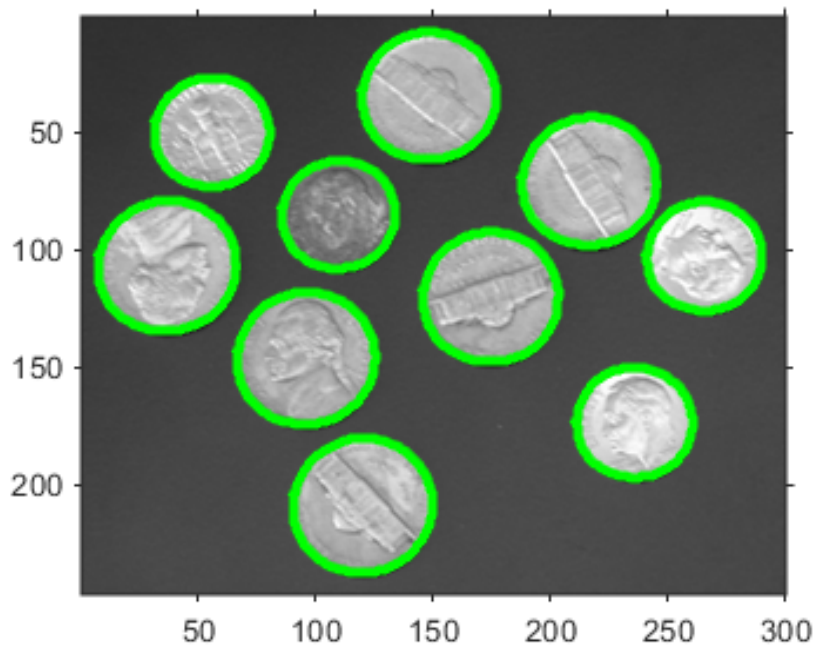


To trace the boundaries of all the coins in the image, use the `bwboundaries` function. By default, `bwboundaries` finds the boundaries of all objects in an image, including objects inside other objects. In the binary image used in this example, some of the coins contain black areas that `bwboundaries` interprets as separate objects. To ensure that `bwboundaries` only traces the coins, use `imfill` to fill the area inside each coin. `bwboundaries` returns a cell array, where each cell contains the row/column coordinates for an object in the image.

```
BW_filled = imfill(BW, 'holes');  
boundaries = bwboundaries(BW_filled);
```

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
```



Select First Step and Direction for Tracing

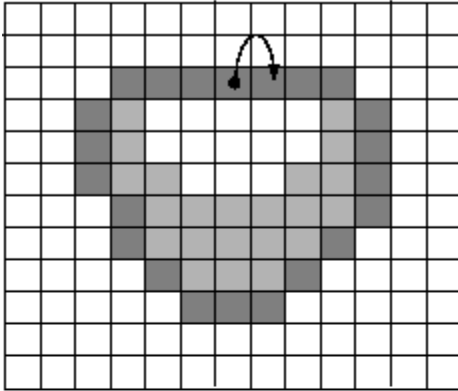
For certain objects, you must take care when selecting the border pixel you choose as the starting point and the direction you choose for the first step parameter (north, south, etc.).

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).

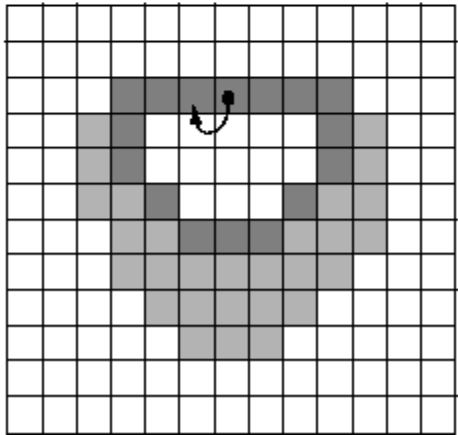
FirstStep = North

Direction = Clockwise

 = Boundary pixel = Starting point

FirstStep = South

Direction = Clockwise

**Impact of First Step and Direction Parameters on Boundary Tracing**

Hough Transform

The Image Processing Toolbox supports functions that enable you to use the Hough transform to detect lines in an image.

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.

After you compute the Hough transform, you can use the `houghpeaks` function to find peak values in the parameter space. These peaks represent potential lines in the input image.

After you identify the peaks in the Hough transform, you can use the `houghlines` function to find the endpoints of the line segments corresponding to peaks in the Hough transform. This function automatically fills in small gaps in the line segments.

Detect Lines in Images

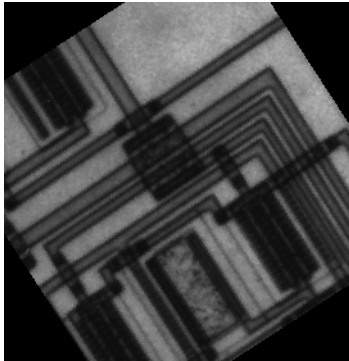
The following example shows how to use the Hough function to detect lines in an image.

- 1 Read an image into the MATLAB workspace.

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

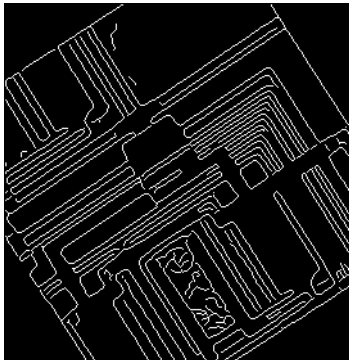
- 2 For this example, rotate and crop the image using the `imrotate` function.

```
rotI = imrotate(I,33,'crop');  
fig1 = imshow(rotI);
```



- 3 Find the edges in the image using the `edge` function.

```
BW = edge(rotI, 'canny');
figure, imshow(BW);
```

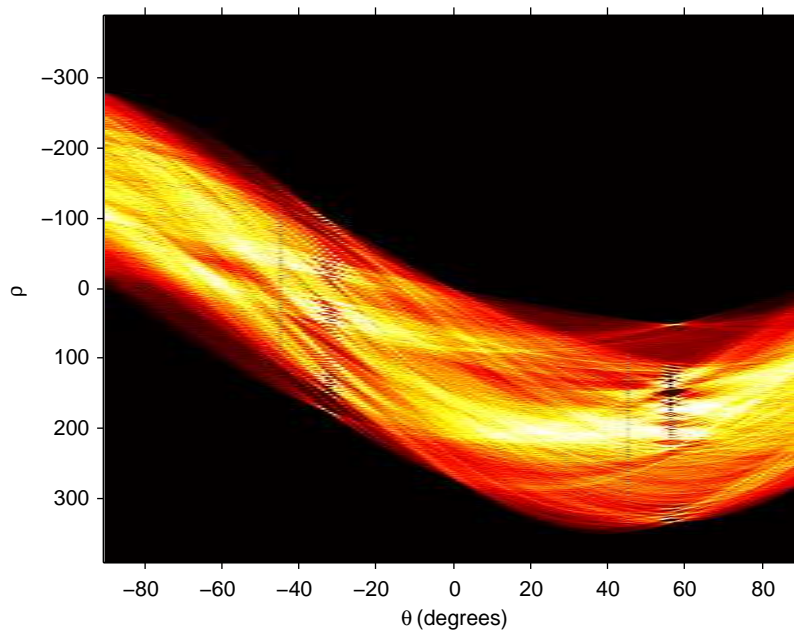


- 4 Compute the Hough transform of the image using the `hough` function.

```
[H,theta,rho] = hough(BW);
```

- 5 Display the transform using the `imshow` function.

```
figure, imshow(imadjust(mat2gray(H)),[], 'XData',theta, 'YData',rho,...
    'InitialMagnification','fit');
xlabel('\theta (degrees)'), ylabel('\rho');
axis on, axis normal, hold on;
colormap(hot)
```

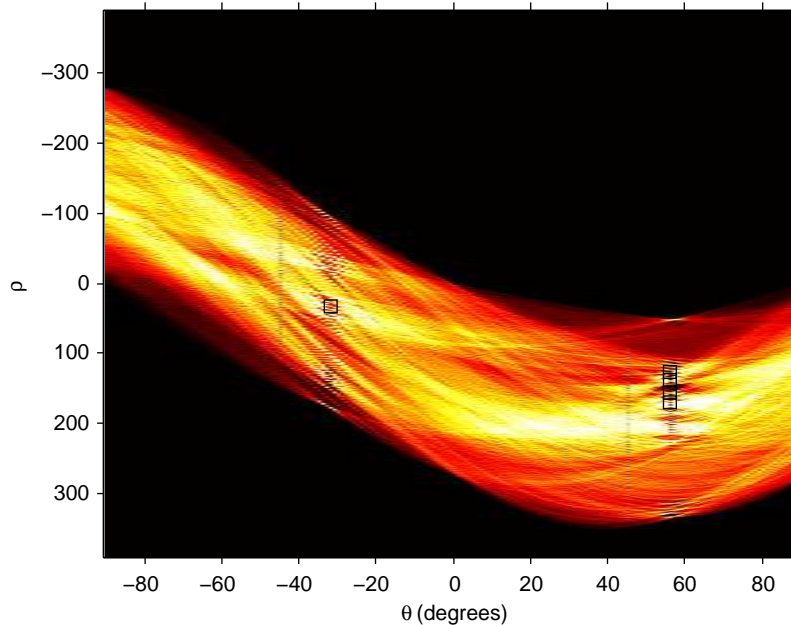


- 6 Find the peaks in the Hough transform matrix, H , using the `houghpeaks` function.

```
P = houghpeaks(H,5,'threshold',ceil(0.3*max(H(:))));
```

- 7 Superimpose a plot on the image of the transform that identifies the peaks.

```
x = theta(P(:,2));  
y = rho(P(:,1));  
plot(x,y,'s','color','black');
```

- 8 Find lines in the image using the `houghlines` function.

```
lines = houghlines(BW,theta,rho,P,'FillGap',5,'MinLength',7);
```

- 9 Create a plot that superimposes the lines on the original image.

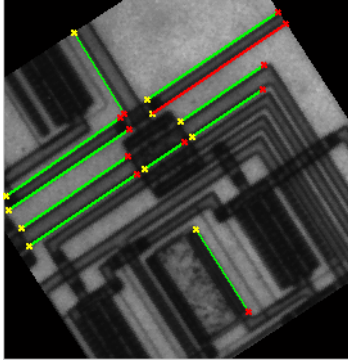
```
figure, imshow(rotI), hold on
max_len = 0;
for k = 1:length(lines)
    xy = [lines(k).point1; lines(k).point2];
    plot(xy(:,1),xy(:,2),'LineWidth',2,'Color','green');

    % 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','red');
```



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.

Determine Image Homogeneity Using Quadtree Decomposition

To illustrate, this example performs quadtree decomposition on a 512-by-512 grayscale image.

- 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(:)) <= 0.27
```

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.

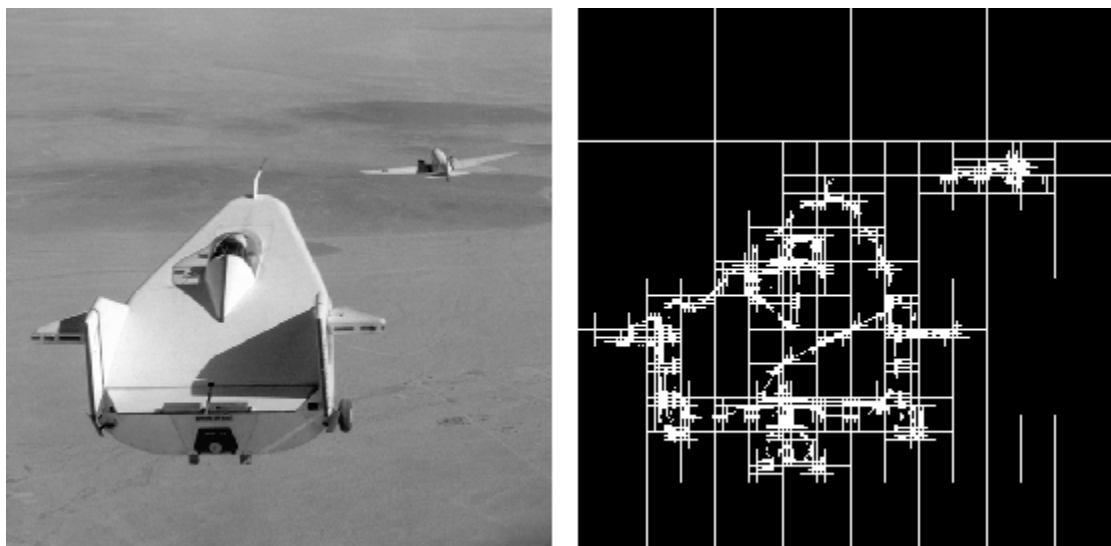


Image Courtesy of NASA

Image and a Representation of Its Quadtree Decomposition

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 includes several texture analysis functions that filter an image using standard statistical measures. 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 table lists these functions.

Function	Description
<code>rangefilt</code>	Calculates the local range of an image.
<code>stdfilt</code>	Calculates the local standard deviation of an image.
<code>entropyfilt</code>	Calculates the local entropy of a grayscale image. Entropy is a statistical measure of randomness.

Understanding How Texture Filter Functions Work

The functions all operate in a similar way: they define a neighborhood around the pixel of interest, calculate the statistic for that neighborhood, and use that value as the value of the pixel of interest in the output image.

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 =
```

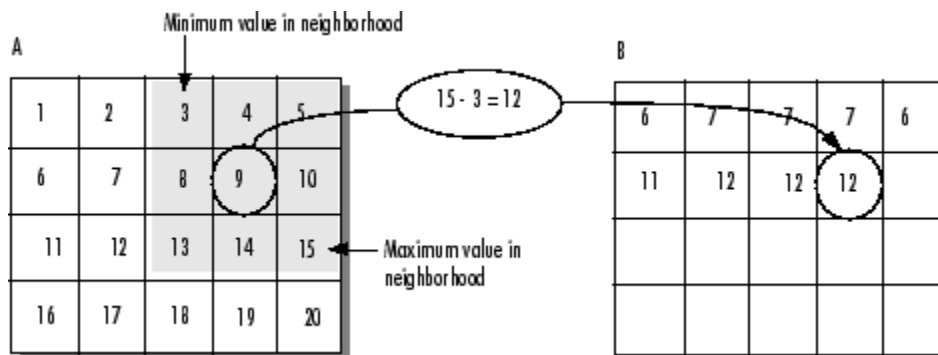
1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

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 of 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.

Detect Regions of Texture in Images

This example shows how to detect regions of texture in an image using the texture filter functions.

Read in the image and display it.

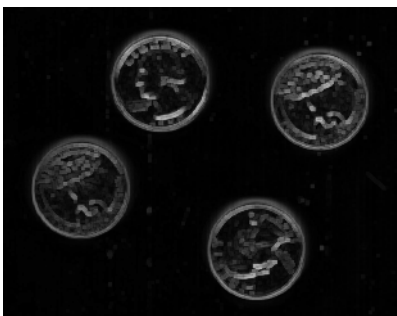
```
I = imread('eight.tif');  
imshow(I)
```



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.

Filter the image with the `rangefilt` function and display the results. Range filtering makes the edges and contours of the coins visible.

```
K = rangefilt(I);  
figure, imshow(K)
```



Gray-Level Co-Occurrence Matrix (GLCM)

A statistical method of examining texture 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 GLCM 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, creating a GLCM, and then extracting statistical measures from this matrix. (The texture filter functions, described in “Texture Analysis” on page 11-35 cannot provide information about shape, i.e., the spatial relationships of pixels in an image.)

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.

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.

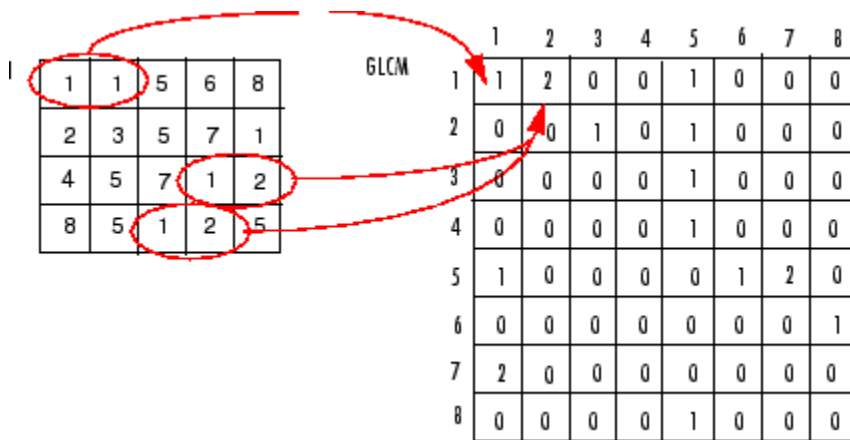
Understanding a Gray-Level Co-Occurrence Matrix

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 “Derive Statistics from GLCM and Plot Correlation” on page 11-40 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

Specify Offset Used in GLCM Calculation

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

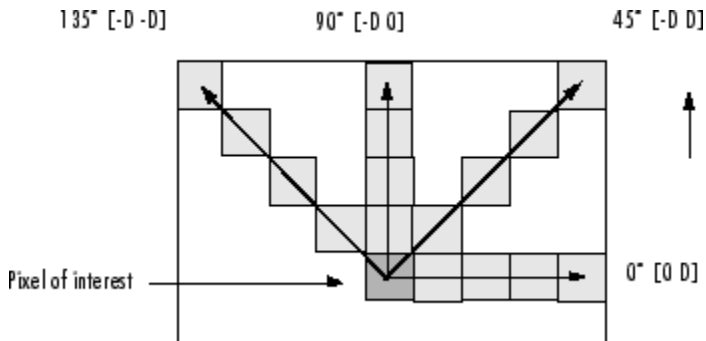
orientation. For this reason, `graycomatrix` can create multiple GLCMs for a single input image.

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.

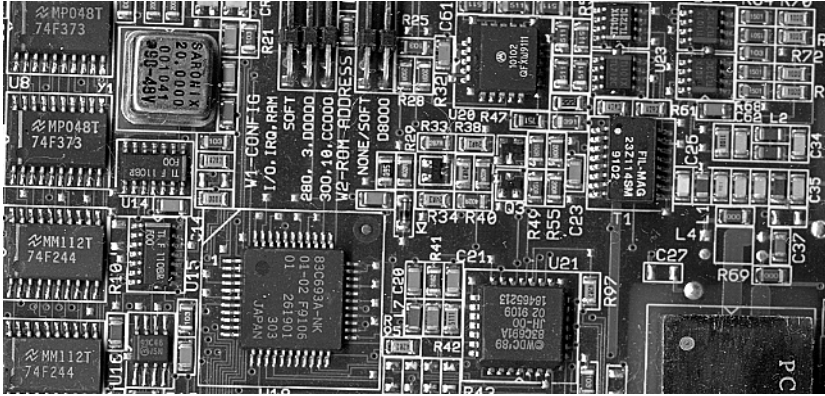


Derive Statistics from GLCM and Plot Correlation

This example shows how to create a set of GLCMs and derive statistics from them. The example also illustrates how the statistics returned by `graycoprops` have a direct relationship to the original input image.

Read 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)
```



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)'];
```

Create the GLCMs. Call the `graycomatrix` function specifying the offsets.

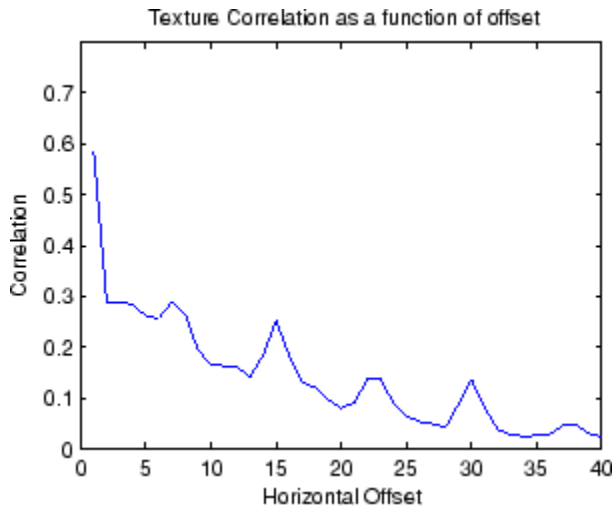
```
glcms = graycomatrix(circuitBoard,'Offset',offsets0)
```

Derive statistics from the GLCMs using the `graycoprops` function. The example calculates the contrast and correlation.

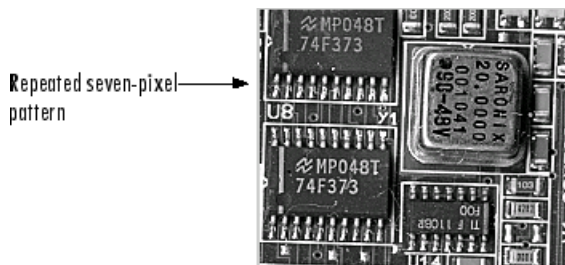
```
stats = graycoprops(glcms,'Contrast Correlation');
```

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.



Contrast Adjustment

In this section...

“Adjust Intensity Values to Specified Range” on page 11-44

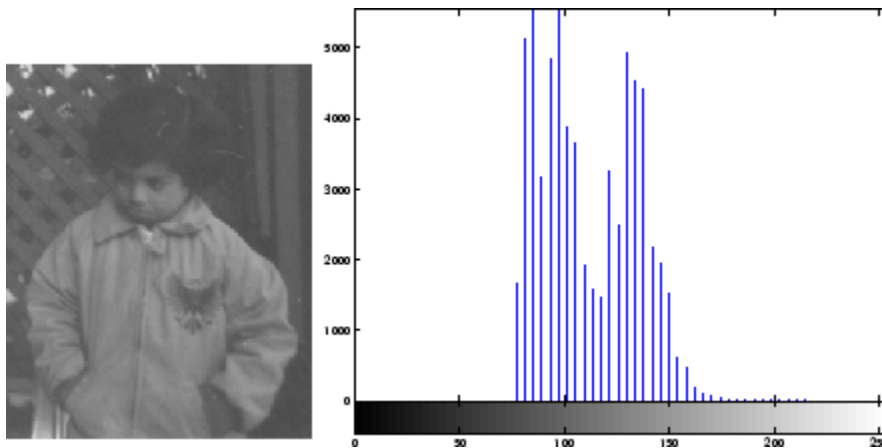
“Adjust Image Intensity Values Using Histogram Equalization” on page 11-48

“Adjust Contrast Using Contrast-Limited Adaptive Histogram Equalization (CLAHE)” on page 11-49

“Enhance Color Separation Using Decorrelation Stretching” on page 11-50

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. If you remap the data values to fill the entire intensity range $[0, 255]$, you can increase the contrast of the image.



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.

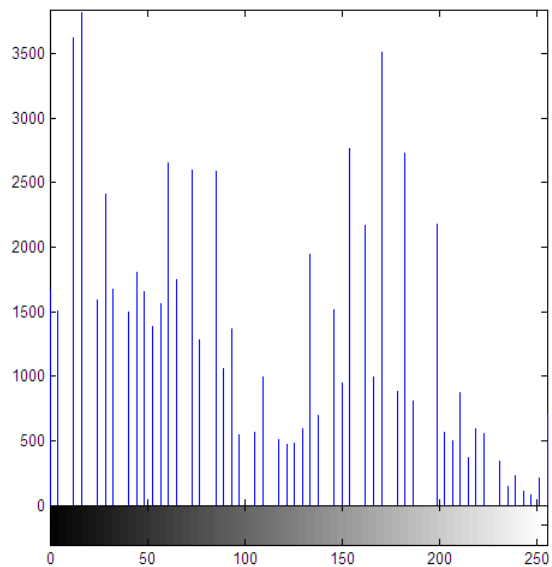
Adjust Intensity Values to Specified Range

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

Specify Adjustment Limits as Range

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 “Set Image Intensity Adjustment Limits Automatically” on page 11-46.

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

Set Image Intensity 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.

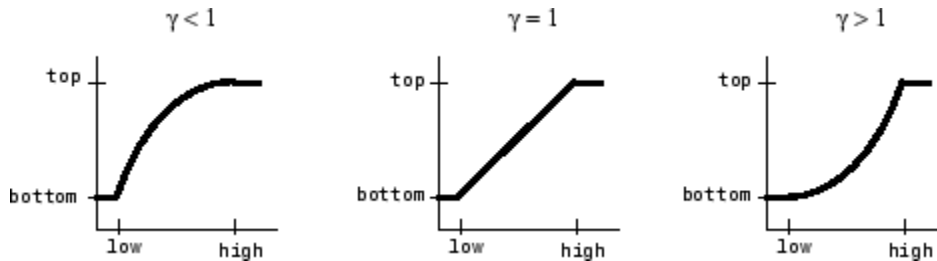
Gamma Correction

`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

Adjust Image Intensity Values Using 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)
```

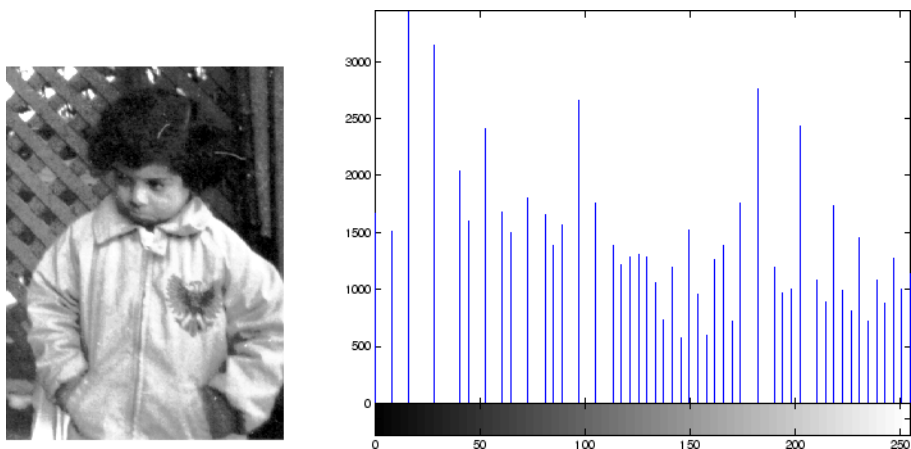
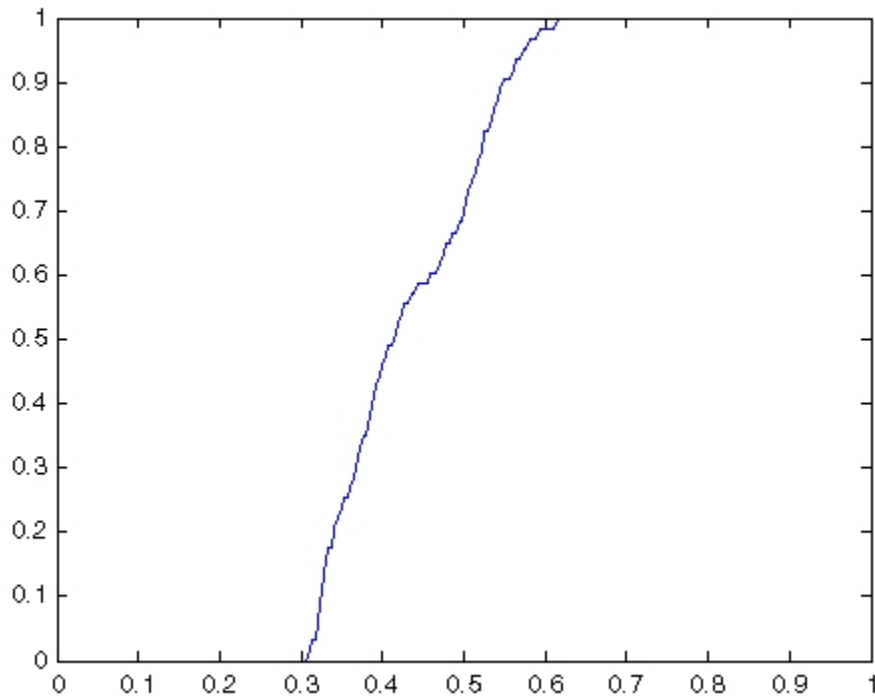


Image After Histogram Equalization with Its Histogram

`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.

Adjust Contrast Using Contrast-Limited Adaptive Histogram Equalization (CLAHE)

As an alternative to using `histeq`, you can perform contrast-limited adaptive histogram equalization (CLAHE) using the `adaphisteq` function. While `histeq` works on the entire image, `adaphisteq` 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(J)  
figure, imhist(J,64)
```

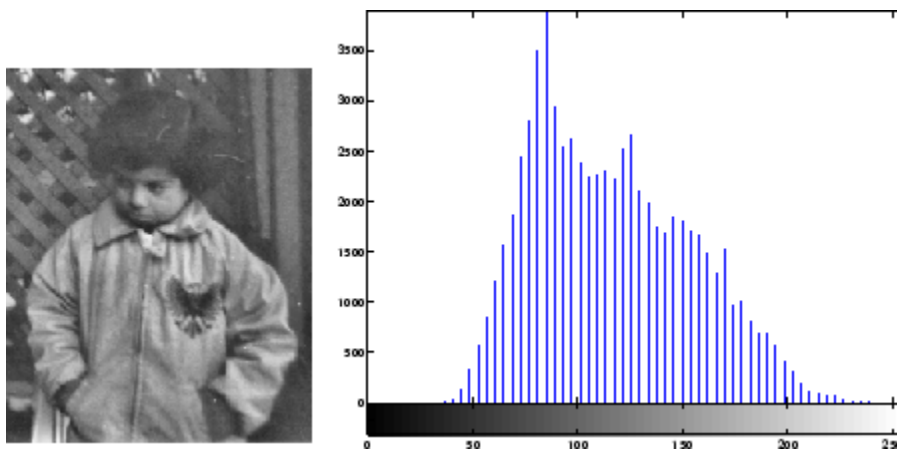


Image After CLAHE Equalization with Its Histogram

Enhance Color Separation Using Decorrelation Stretching

Decorrelation stretching enhances the color separation of an image with significant band-to-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 11-53 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 `imdata` folder. 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]});
```

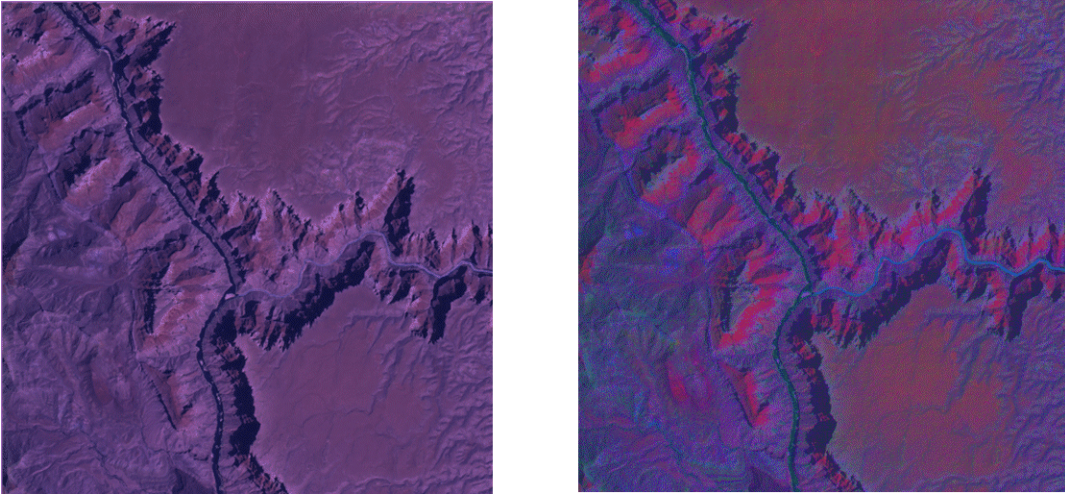
- 2 Then perform the decorrelation stretch:

```
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.

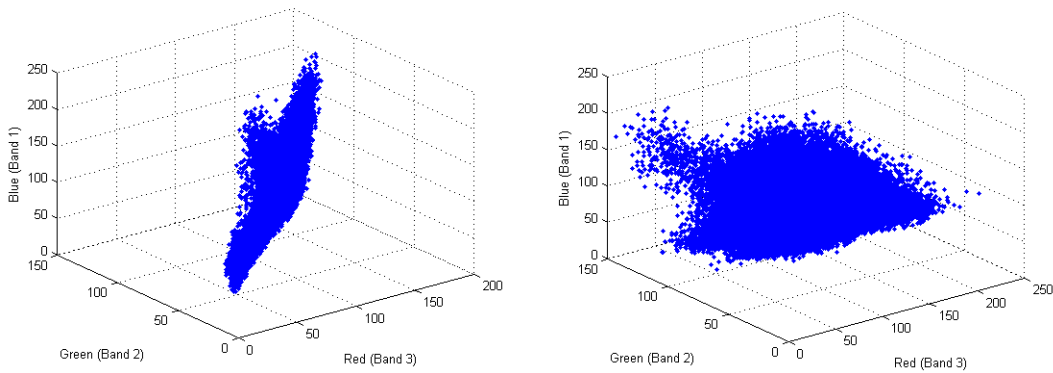


Little Colorado River Before (left) and After (right) Decorrelation Stretch

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)')
```



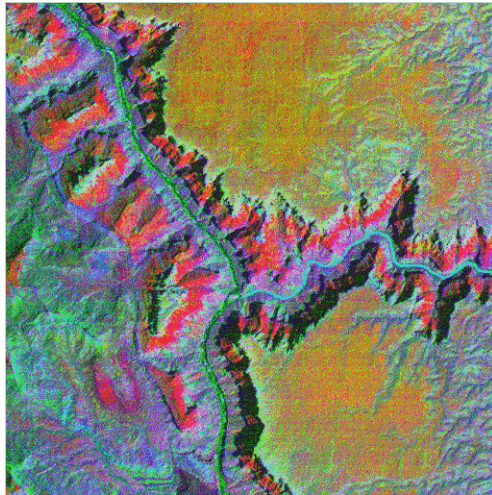
Color Scatterplot Before (left) and After (right) Decorrelation Stretch

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, 'To1', 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

In this section...

“Remove Noise By Linear Filtering” on page 11-55

“Remove Noise Using Median Filtering” on page 11-55

“Remove Noise By Adaptive Filtering” on page 11-58

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.

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.

Remove Noise By 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 “What Is Image Filtering in the Spatial Domain?” for more information about linear filtering using `imfilter`.

Remove Noise Using Median Filtering

This example shows how to remove *salt and pepper* noise from an image using an averaging filter and a median filter (`medfilt2`) to allow comparison of the results.

Median filtering is similar to 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. 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.

Read image and display it.

```
I = imread('eight.tif');  
imshow(I)
```



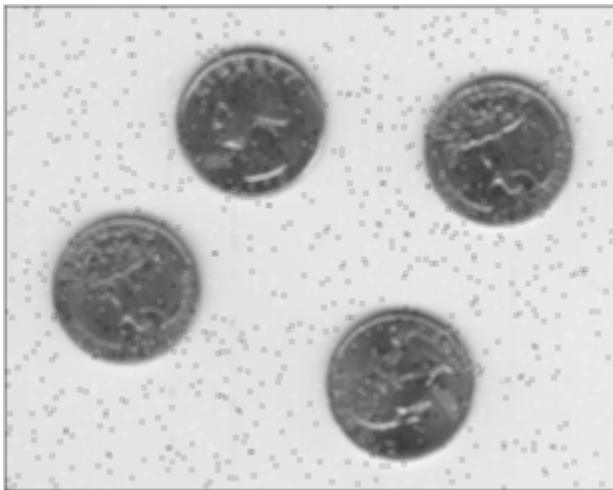
For this example, add salt and pepper noise to the image. This type of noise consists of random pixels being set to black or white (the extremes of the data range).

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



Filter the noisy image with an averaging filter and display the results. The example uses a 3-by-3 neighborhood.

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



Now use a median filter to filter the noisy image and display the results. The example uses a 3-by-3 neighborhood. Notice that `medfilt2` does a better job of removing noise, with less blurring of edges.

```
L = medfilt2(J,[3 3]);  
figure, imshow(L)
```



Remove Noise By 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.

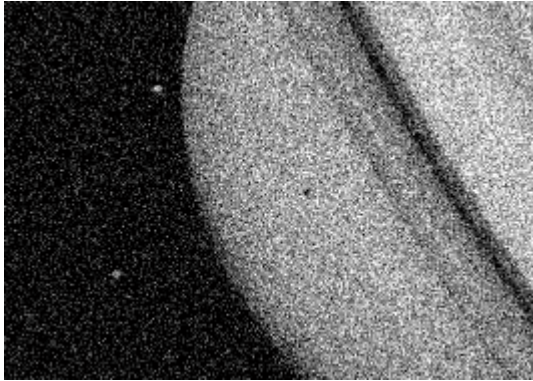
`wiener2` works best when the noise is constant-power (“white”) additive noise, such as Gaussian noise. The example below applies `wiener2` to an image of Saturn that has had Gaussian noise added.

- 1 Read in an image. Because the image is a truecolor image, the example converts it to grayscale.

```
RGB = imread('saturn.png');  
I = rgb2gray(RGB);
```

- 2 The example then add Gaussian noise to the image and then displays the image. Because the image is quite large, the figure only shows a portion of the image.

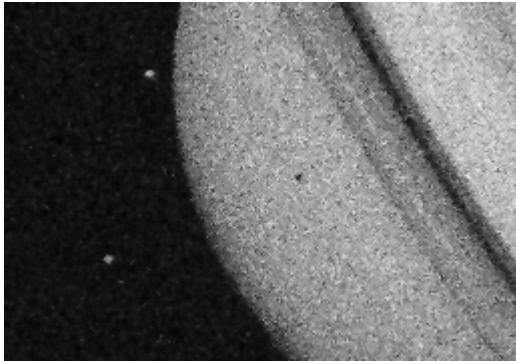
```
J = imnoise(I, 'gaussian', 0, 0.025);  
imshow(J)
```



Portion of the Image with Added Gaussian Noise

- 3 Remove the noise, using the `wiener2` function. Again, the figure only shows a portion of the image

```
K = wiener2(J, [5 5]);  
figure, imshow(K)
```



Portion of the Image with Noise Removed by Wiener Filter

Image Segmentation

Image segmentation is the process of partitioning an image into parts or regions. This division into parts is often based on the characteristics of the pixels in the image. For example, one way to find regions in an image is to look for abrupt discontinuities in pixel values, which typically indicate edges. These edges can define regions. Another method is to divide the image into regions based on color values.

Image Segmentation Using the Color Thesholder App

This example shows how to segment an image to create a binary mask image using the Color Thresholder app. The example has several parts. The first part shows how to open an image in the Color Thresholder. The next part of the example shows how to use the color selection option to segment the image automatically. The next part shows an iterative approach to thresholding using color component controls. In a typical scenario, you perform an initial segmentation using color selection and then refine that segmentation using color component controls. The last part of this example shows what you can do after you complete the segmentation, such as creating a mask image, saving a segmented version of the original image, and getting the MATLAB code used to perform the segmentation.

In this section...

“Open Image in Color Thresholder App” on page 11-62

“Segment Image Using Color Selector” on page 11-67

“Segment Image Using Color Component Controls” on page 11-70

“Create an Image Mask” on page 11-74

Open Image in Color Thresholder App

This example shows how to open an image in the Color Thresholder app. When you first open the app, you must choose the color space to use to represent the color components of the image. Choose the color space where the colors you are interested in segmenting appear near each other in the color model. You can always change the color space you choose later, using **New Color Space**.

Read a color image into the MATLAB workspace and view it.

```
I = imread('peppers.png');
```

```
imshow(I)
```

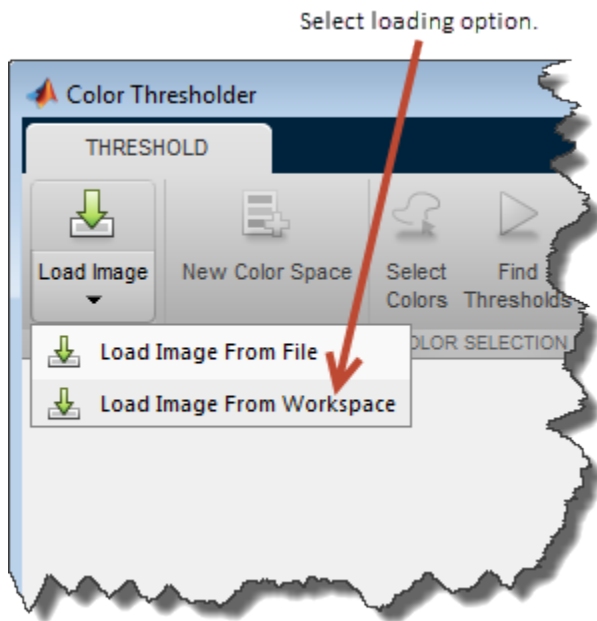



Open the Color Thresholder app. From the MATLAB Toolstrip, open the Apps tab and

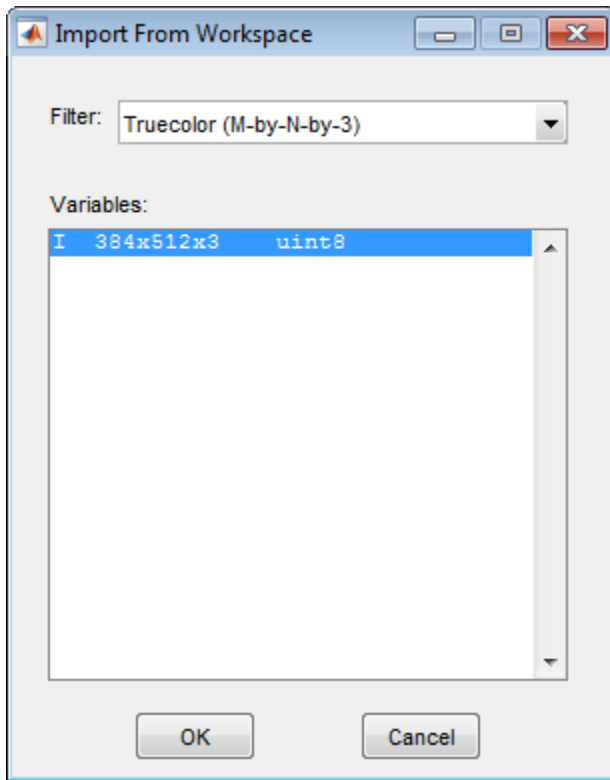


under Image Processing and Computer Vision, click **Color Thresholder**. You can also open the app using the `colorThresholder` command.

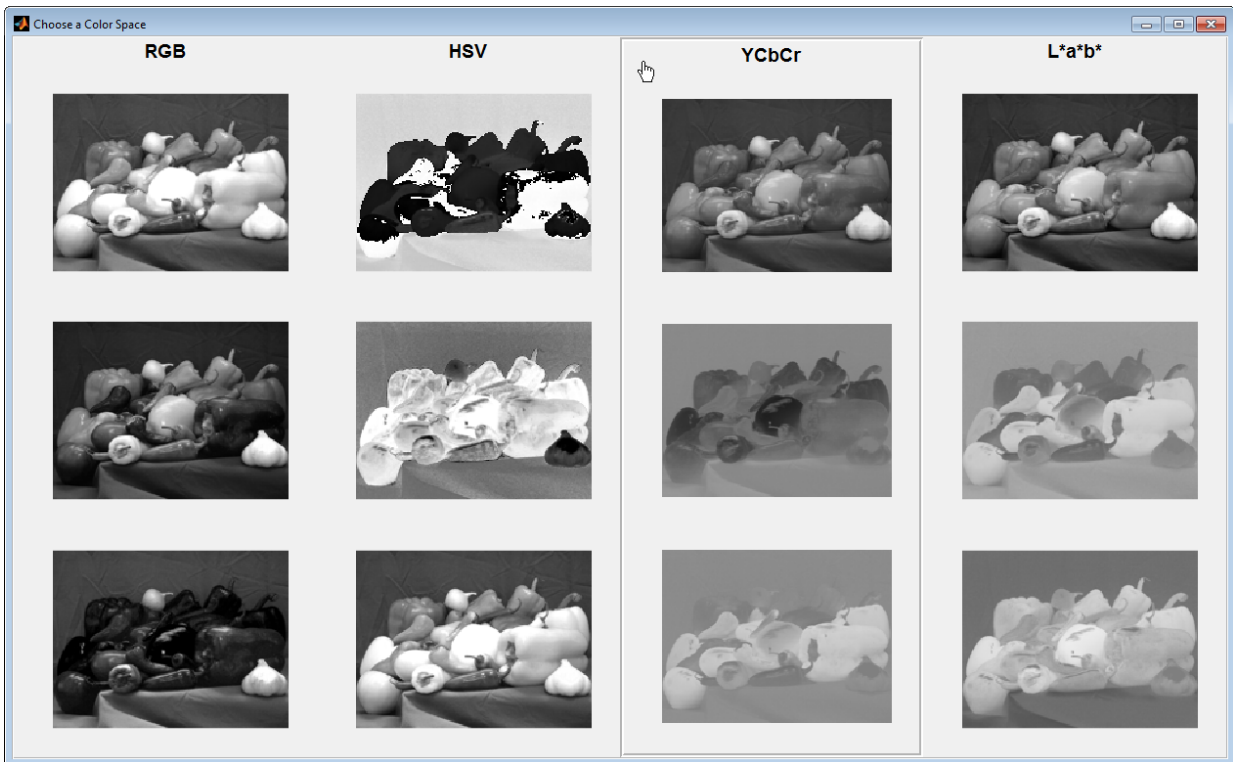
Bring an image into the Color Thresholder app. Click **Load Image**. Since you have already read the image into the workspace, select **Load Image from Workspace**.



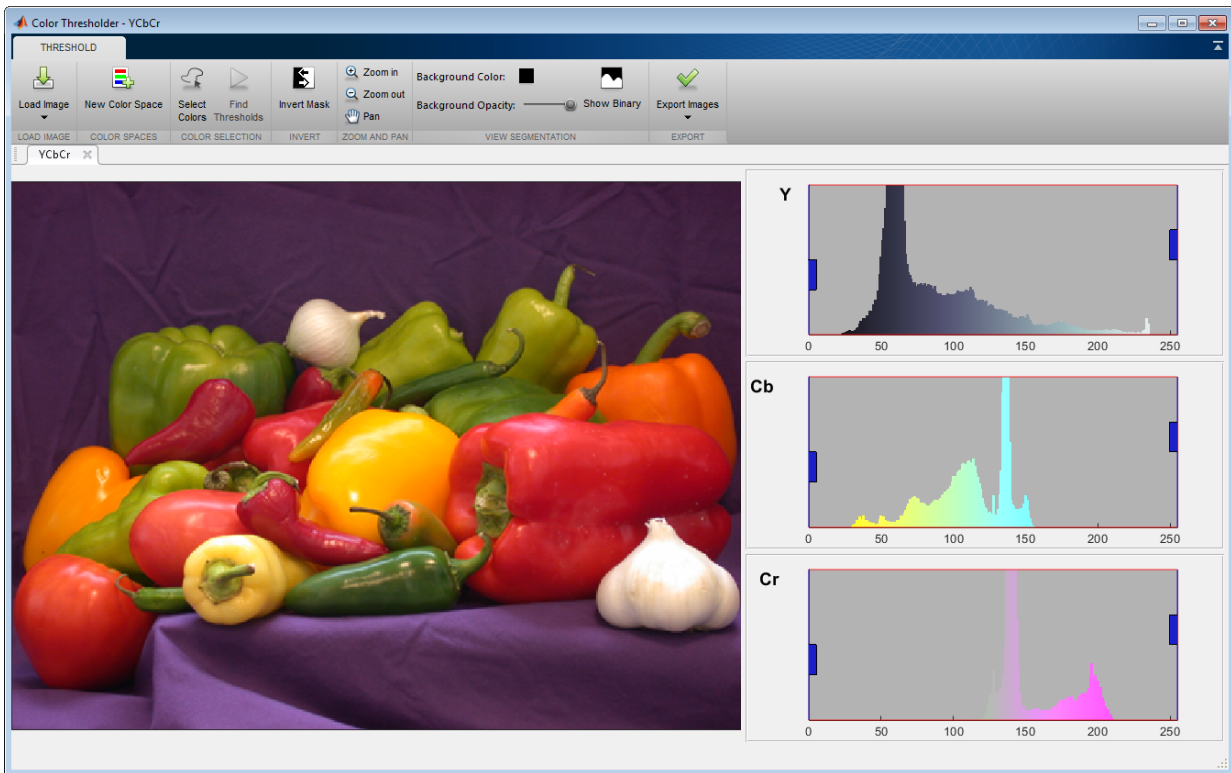
Select the variable that contains the image you read into the workspace in the Import From Workspace dialog box and click **OK**.



Choose the color space you want to represent color components in your image. When it opens, the Color Thresholder app displays the **Choose a color space** dialog box. This dialog box displays your image as it is represented in several popular color spaces: RGB, HSV, YCbCr, and $L^*a^*b^*$. Using the mouse, choose the color space. For this example, choose the YCbCr color space, as shown.



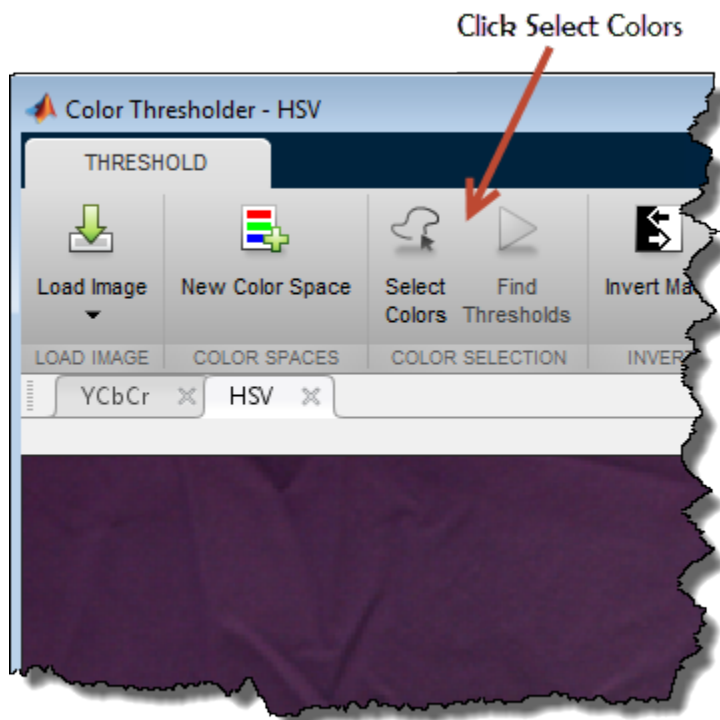
The app opens, displaying the image along with a set of controls for each color component. For the YCbCr color space, the Color Thresholder displays three histograms representing color components of the image. In this color space, the **Y** component represents brightness, the **Cb** component represents the blue-yellow spectrum, and the **Cr** component represents the red-green spectrum. Other color spaces use different types of controls.



Segment Image Using Color Selector

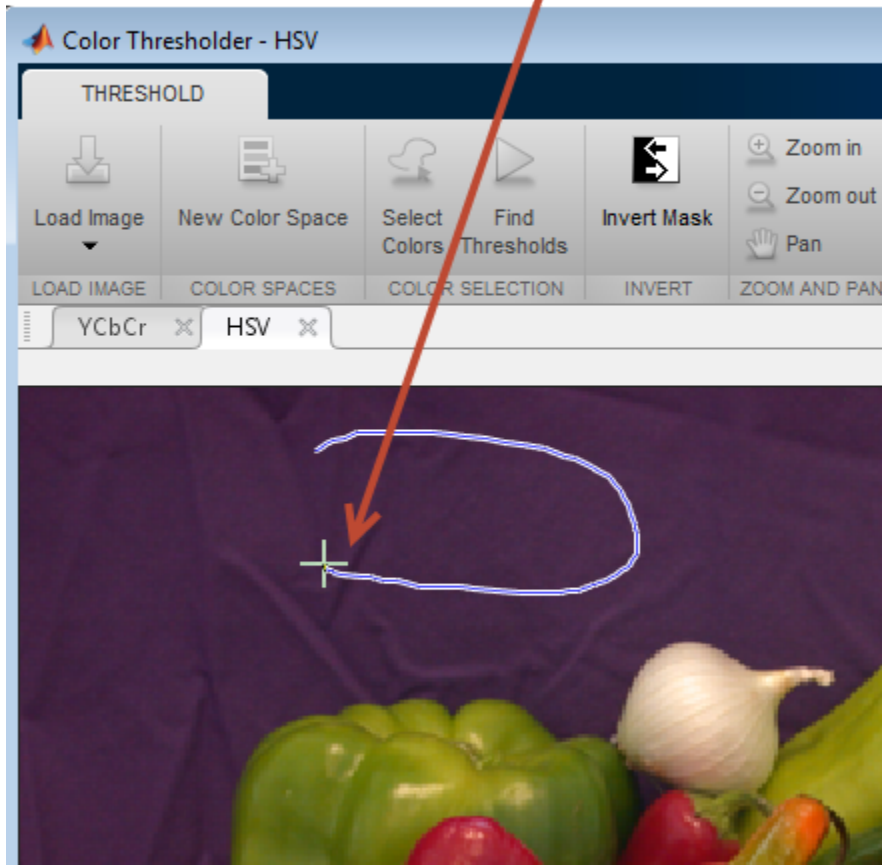
This example shows how to use the Color Selection option to segment an image automatically. With this option, you select a color in the foreground or background by drawing a freehand region. You can draw multiple regions. After you segment your image using color selection, you can refine your result using the individual color component controls. See “Segment Image Using Color Component Controls” on page 11-70.

To segment the image automatically based on a color selection, click **Select Colors**. When you move the cursor of the image, it changes shape to a cross-hairs which you can use to draw regions to specify the colors you want to segment.



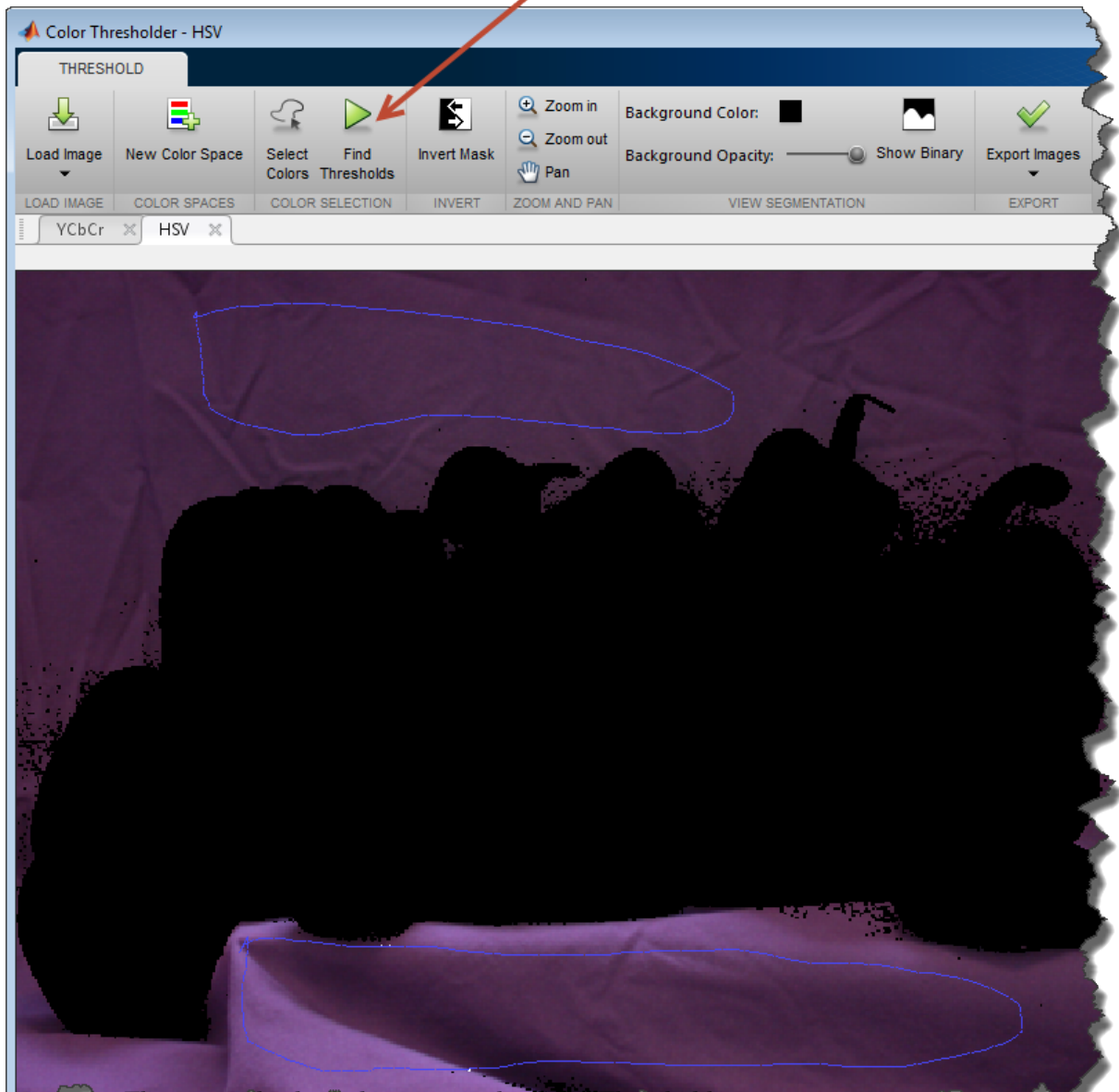
Draw a freehand region using the mouse to select the color you want to use for segmentation. You can draw multiple regions.

Draw a freehand region to select colors.



After drawing the regions, click **Find Thresholds** to threshold the image based on the color in the region you drew. To delete the regions that you drew, right click on the line and select **Delete**.

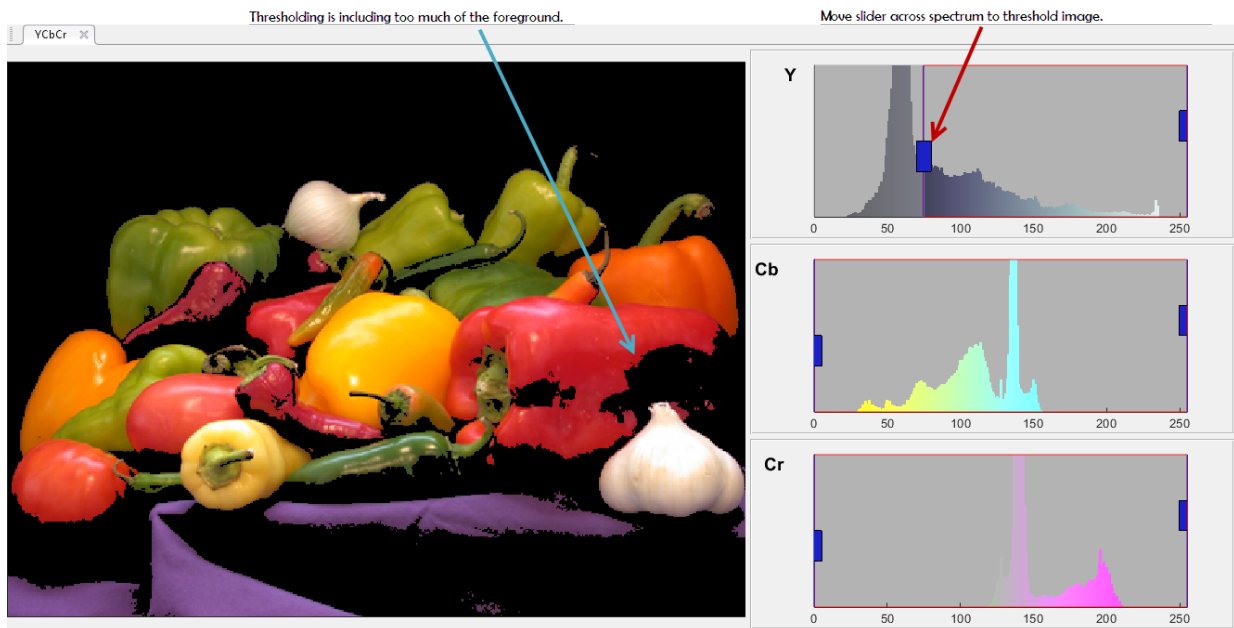
Click Find Thresholds to segment the image.



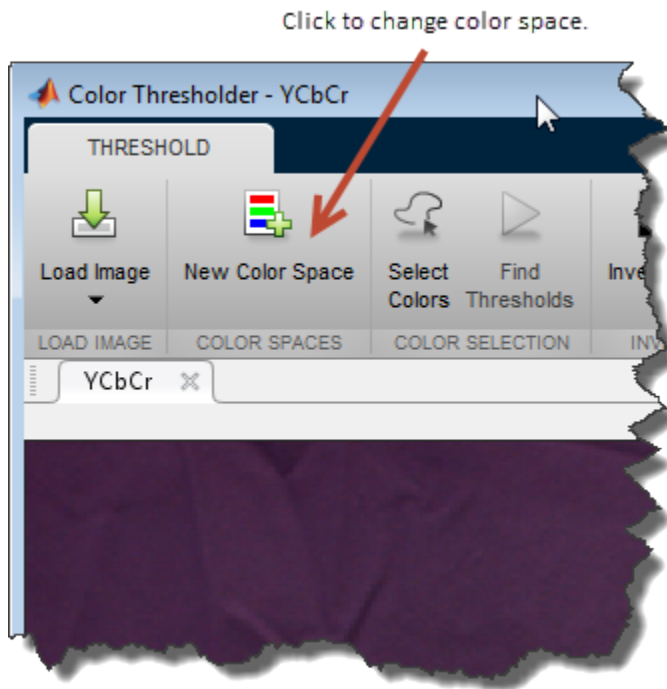
This example shows how to use the Color Thresholder app to segment an image interactively using color component controls. Segmentation using the Color Thresholder

is an iterative process—you might need to try several different color spaces before you achieve a segmentation that meets your needs. You can also perform an initial segmentation automatically using the color selection option and then refine the results using the color component controls. See “Segment Image Using Color Selector” on page 11-67.

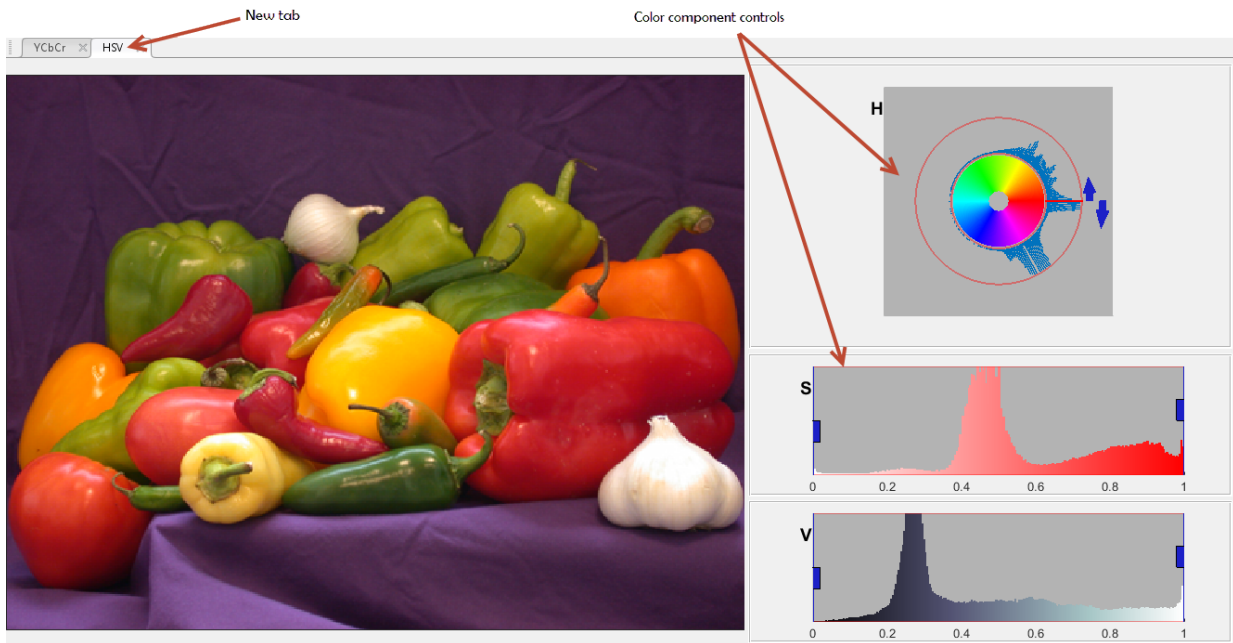
Segment the image interactively using the color component controls. Move the slider associated with each histogram over the colors in the histogram. You can see the segmentation in progress. For this example, moving the slider on the **Y** component has the greatest effect on segmenting the background, but it’s hard to cleanly segment the background without including part of the foreground image, as shown in the following figure. Since this is an iterative process, try another color space.



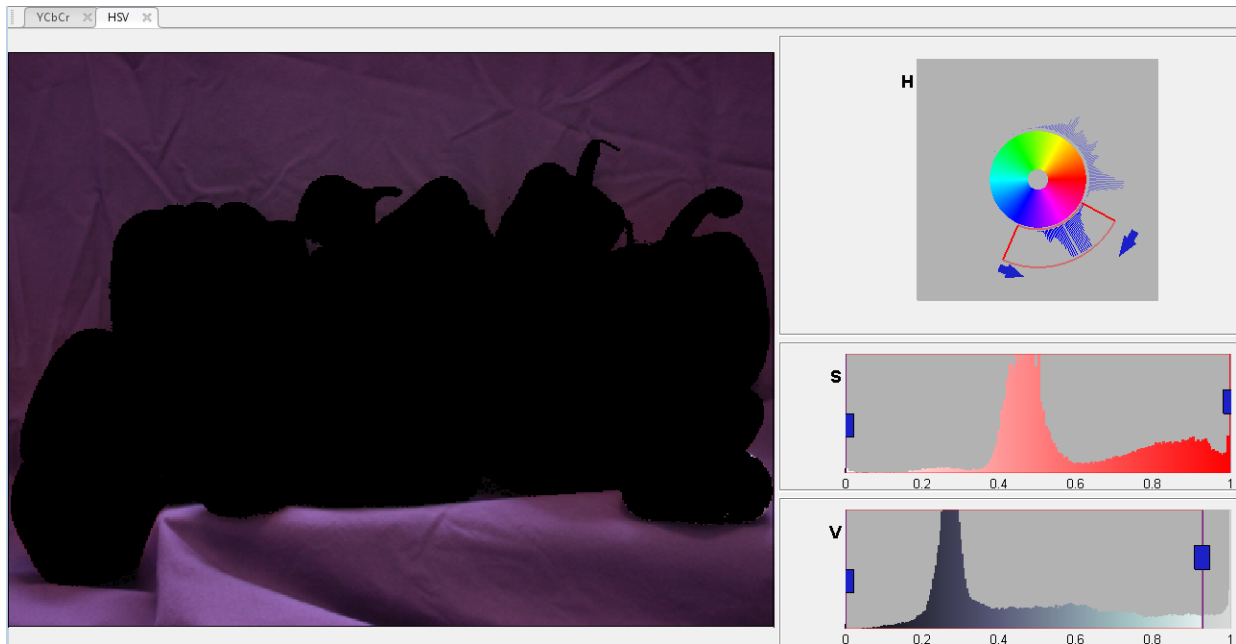
To use another color space, click **New Color Space**. The app displays the **Choose a color space** dialog box again.



Select a new color space in the Choose a Color Space dialog box. For this example, choose the **HSV** color space. The Color Thresholder creates a new tab displaying the image and the color component controls for this color space. The HSV color space uses a dual-direction knob for the **H** component and two histogram sliders for the **S** and **V** components. In this color space, **H** stands for hue, **S** for saturation, and **V** for value.



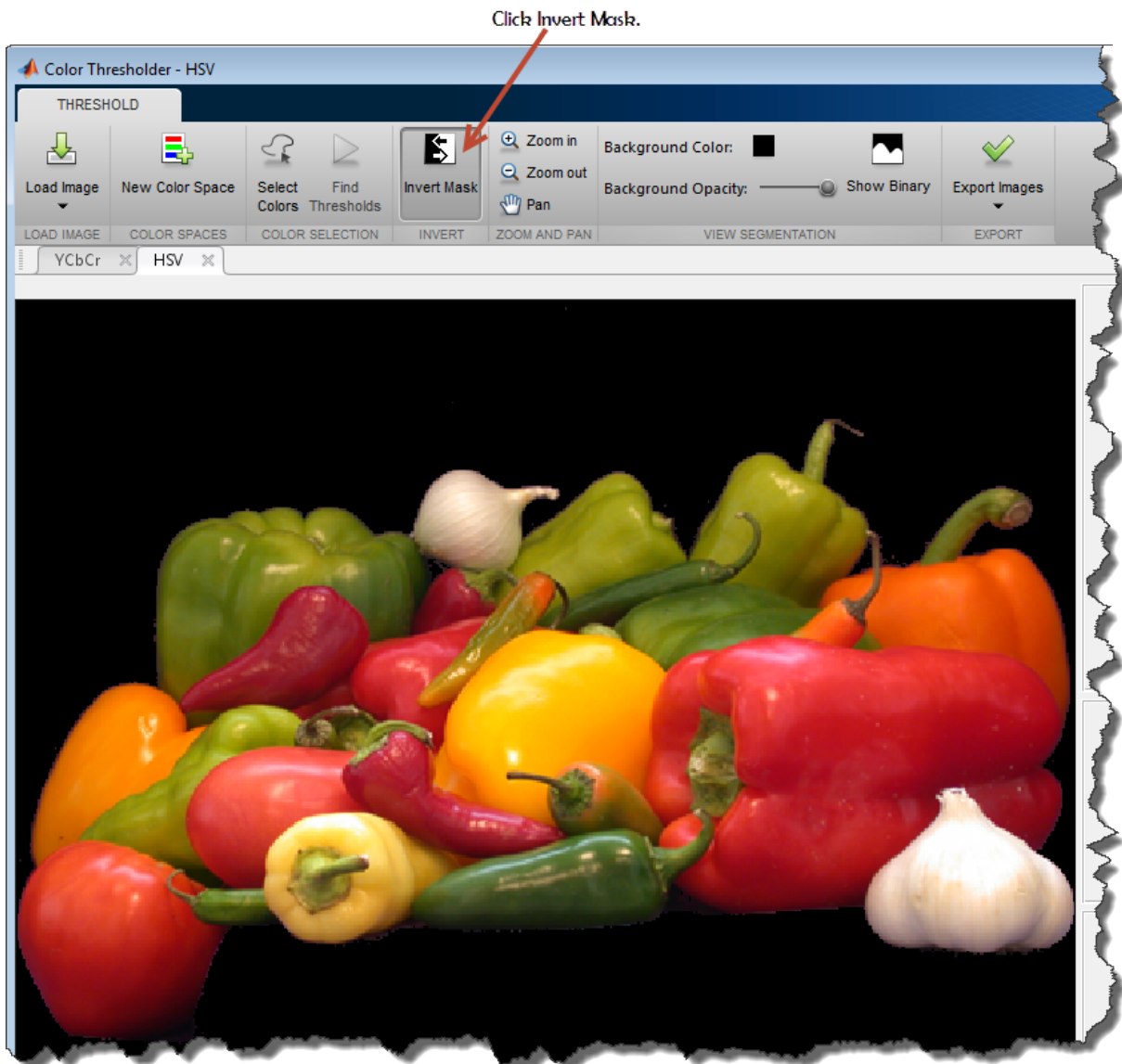
As you did before with the YCbCr color space, use the color component controls to segment the image interactively. As you use the controls, you can see the segmentation in progress. Using the mouse, grab one of the handles on the **H** control and move it in the direction of the arrow. Experiment with the controls until you have a clean separation of the background from the foreground. In this color space, you can achieve a good segmentation using the **H** control, as shown in the following figure. You can clean up small imperfections after you create the mask image using other toolbox functions, such as morphological operators. For information about saving your segmented image, see “Create an Image Mask” on page 11-74.



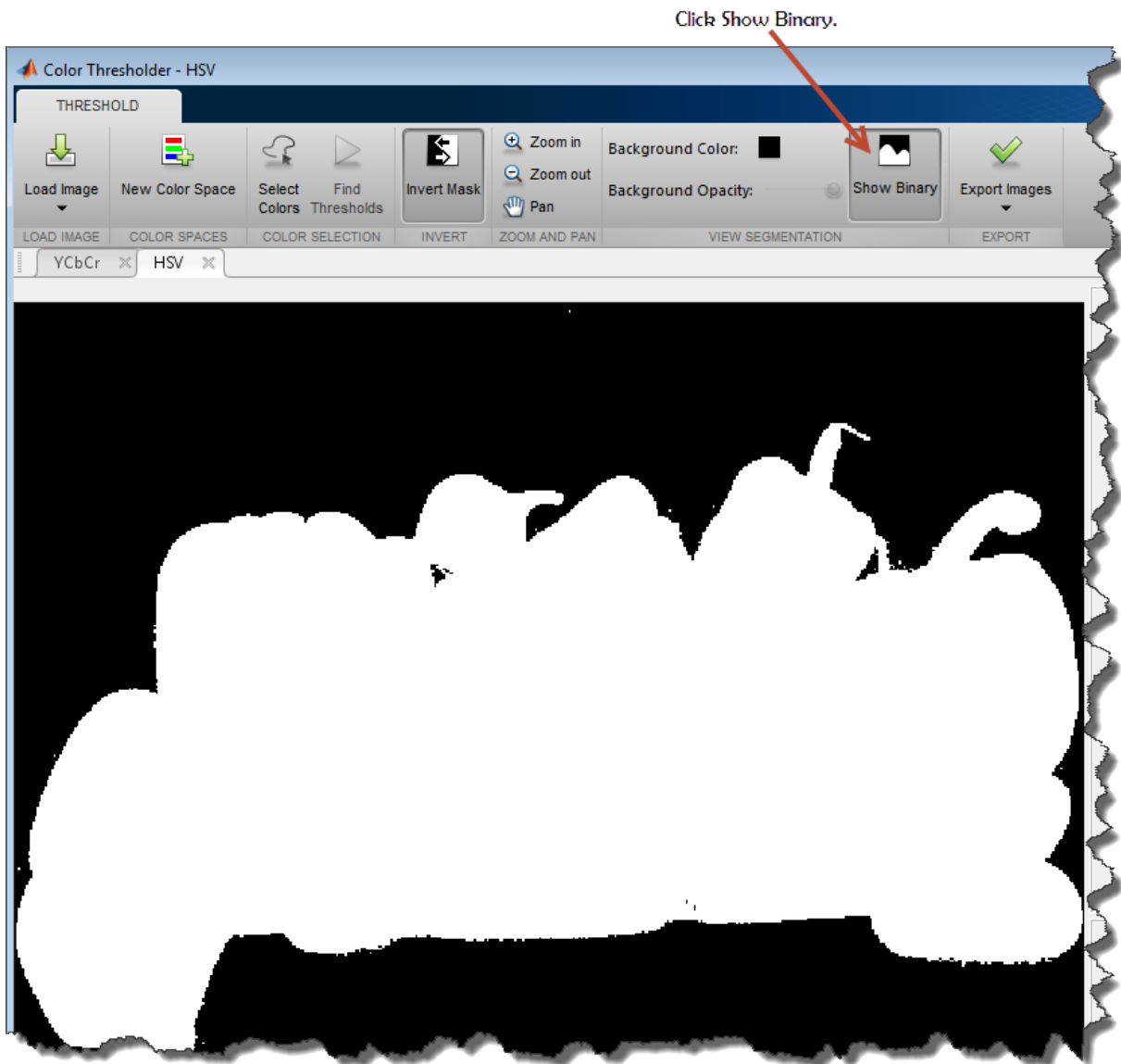
Create an Image Mask

This example shows how to create a mask image after segmentation. You can also get the segmented image and the MATLAB code used to create the mask image.

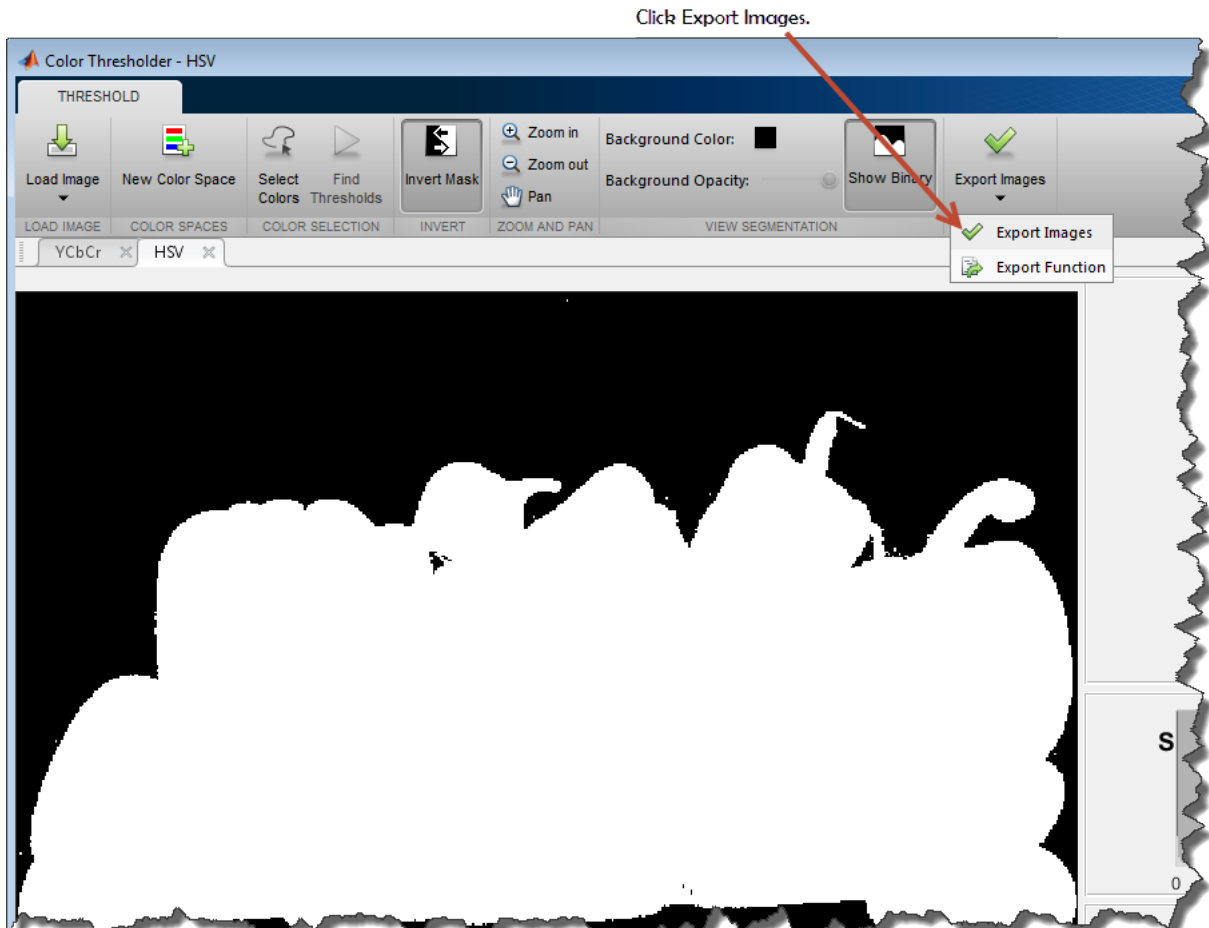
After segmenting the foreground and background, you can swap the foreground and background by clicking **Invert Mask**. Inverting the mask can be helpful when, for example, you can achieve a cleaner segmentation working with the background but you want a mask of the foreground. Perform the segmentation with the background, and then invert the mask.



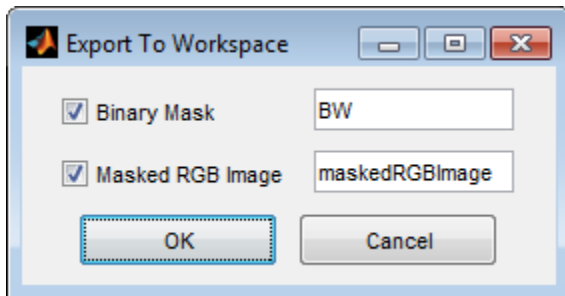
View the binary mask image that you created by clicking **Show Binary**.



When you are satisfied with the segmentation, click **Export Images** and select **Export Images** to save the mask image in the workspace. you can also save the segmented original image.



In the Export to Workspace dialog box, specify the name of the variables for the binary mask image and the segmented version of the original image.



To save the MATLAB code required to recreate the segmentation you just performed, click **Export Images** and select **Export Function**. The Color Thresholder app opens the MATLAB Editor with the code that creates the segmentation. To save the code, click **Save** in the MATLAB Editor. You can run this code, passing it an RGB image, and create the same mask image programmatically.

```
function [BW,maskedRGBImage] = createMask(RGB)
%createMask Threshold RGB image using auto-generated code from colorThresholder app.
% [BW,MASKEDRGBIMAGE] = createMask(RGB) thresholds image RGB using
% auto-generated code from the colorThresholder App. The colorspace and
% minimum/maximum values for each channel of the colorspace were set in the
% App and result in a binary mask BW and a composite image maskedRGBImage,
% which shows the original RGB image values under the mask BW.

% Auto-generated by colorThresholder app on 03-Jun-2014
%-----

% Convert RGB image to chosen color space
I = rgb2hsv(RGB);

% Define thresholds for channel 1 based on histogram settings
channel1Min = 0.739;
channel1Max = 0.940;

% Define thresholds for channel 2 based on histogram settings
channel2Min = 0.328;
channel2Max = 0.671;

% Define thresholds for channel 3 based on histogram settings
channel3Min = 0.122;
channel3Max = 0.803;
```



```
% Create mask based on chosen histogram thresholds
BW = (I(:,:,1) >= channel1Min ) & (I(:,:,1) <= channel1Max) & ...
      (I(:,:,2) >= channel2Min ) & (I(:,:,2) <= channel2Max) & ...
      (I(:,:,3) >= channel3Min ) & (I(:,:,3) <= channel3Max);

% Invert mask
BW = ~BW;

% Initialize output masked image based on input image.
maskedRGBImage = RGB;

% Set background pixels where BW is false to zero.
maskedRGBImage(repmat(~BW,[1 1 3])) = 0;
```

Image Quality Metrics

While the final arbiter of image quality is the human viewer, efforts have been made to create objective measures of quality. This can be useful for many applications. Many objective measures of quality require the existence of a distortion-free copy of an image, called the reference image, that can be used for comparison with the image whose quality is to be measured. The dimensions of the reference image matrix and the dimensions of the degraded image matrix must be identical.

The Image Processing Toolbox provides several function that can be used to measure quality:

- `psnr` — The peak signal-to-noise ratio measure of quality works by first calculating the mean squared error (MSE) and then dividing the maximum range of the data type by the MSE. This measure is simple to calculate but sometimes doesn't align well with perceived quality by humans. For example, the PSNR for a blurred image compared to an unblurred image is quite high, even though the perceived quality is low.
- `ssim` — The Structural Similarity (SSIM) Index measure of quality works by measuring the structural similarity that compares local patterns of pixel intensities that have been normalized for luminance and contrast. This quality metric is based on the principle that the human visual system is good for extracting information based on structure.

In this section...
“Obtain Local Structural Similarity Index” on page 11-80
“Compare Image Quality at Various Compression Levels” on page 11-82

Obtain Local Structural Similarity Index

This example shows how to measure the quality of regions of an image when compared with a reference image. The `ssim` function calculates the structural similarity index for each pixel in an image, based on its relationship to other pixels in an 11-by-11 neighborhood. The function returns this information in an image that is the same size as the image whose quality is being measured. This local, pixel-by-pixel, quality index can be viewed as an image, with proper scaling.

Read an image to use as the reference image.

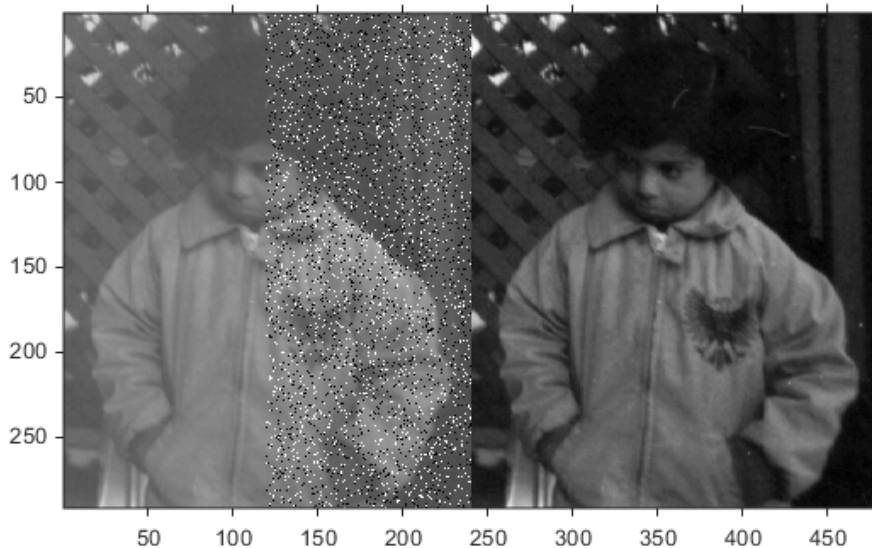
```
ref = imread('pout.tif');
```

Create an image whose quality is to be measured, by making a copy of the reference image and adding noise. To illustrate local similarity, isolate the noise to half of the image. Display the reference image and the noisy image side-by-side.

```
A = ref;
```

```
A(:,ceil(end/2):end) = imnoise(ref(:,ceil(end/2):end),'salt & pepper', 0.1);
```

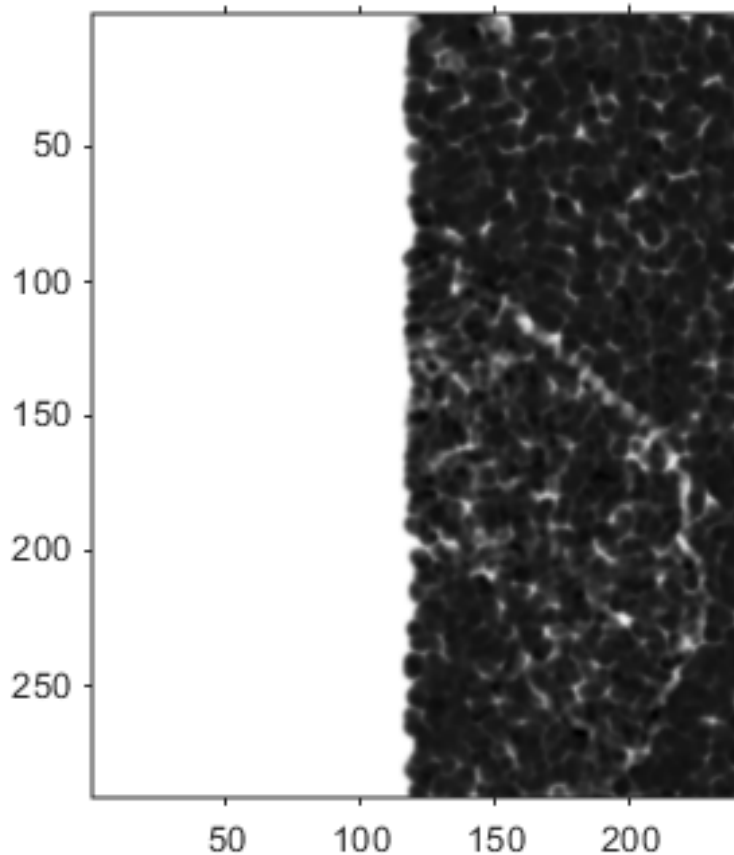
```
figure, imshowpair(A,ref,'montage')
```



Calculate the local Structural Similarity Index for the modified image (A), when compared to the reference image (ref). Visualize the local structural similarity index. Note how left side of the image, which is identical to the reference image displays as white because all the local structural similarity values are 1.

```
[global_sim local_sim] = ssim(A,ref);
```

```
figure, imshow(local_sim,[])
```



Compare Image Quality at Various Compression Levels

This example shows how to test image quality using `ssim`. The example creates images at various compression levels and then plots the quality metrics. To run this example, you must have write permission in your current folder.

Read image.

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

Write the image to a file using various quality values. The JPEG format supports the 'Quality' parameter. Use `ssim` to check the quality of each written image.

```
ssimValues = zeros(1,10);  
qualityFactor = 10:10:100;  
for i = 1:10  
    imwrite(I,'compressedImage.jpg','jpg','quality',qualityFactor(i));  
    ssimValues(i) = ssim(imread('compressedImage.jpg'),I);  
end
```

Plot the results. Note how the image quality score improves as you increase the quality value specified with `imwrite`.

```
plot(qualityFactor,ssimValues,'b-o');  
xlabel('Compression Quality Factor');  
ylabel('SSIM Value');
```

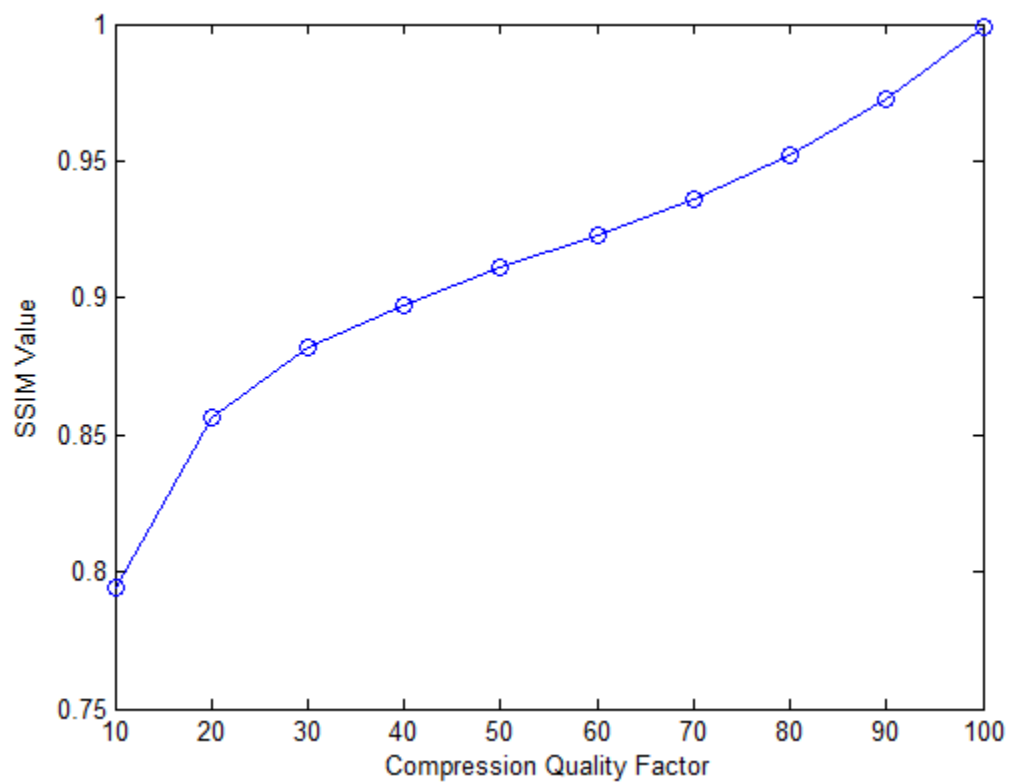


Image Segmentation Using the Image Segmenter App

This example shows how to use the Image Segmenter app to segment an image. The Image Segmenter app uses the active contours algorithm. To segment an image using the app is a three step process.

- Open the app and load an image.
- Create the initial segmentation—This part involves creating a mask image that specifies the initial segmentation from. This initial segmentation defines the starting points where active contour algorithm begins its segmentation of the image. The app provides several ways for you to initialize a segmentation: importing a mask image, using thresholding, using a grid of predefined starting points, or drawing regions in the image.
- Evolve and refine the segmentation—After creating the mask image that defines the segmentation starting points, you run the segmentation. The segmentation algorithm evolves the initial segmentation over the specified number of iterations. The resulting segmentation can then be further refined. The result of the segmentation is a mask image.

In this section...

“Open Image Segmenter App” on page 11-85

“Create the Initial Segmentation” on page 11-88

“Evolve and Refine the Segmentation” on page 11-91

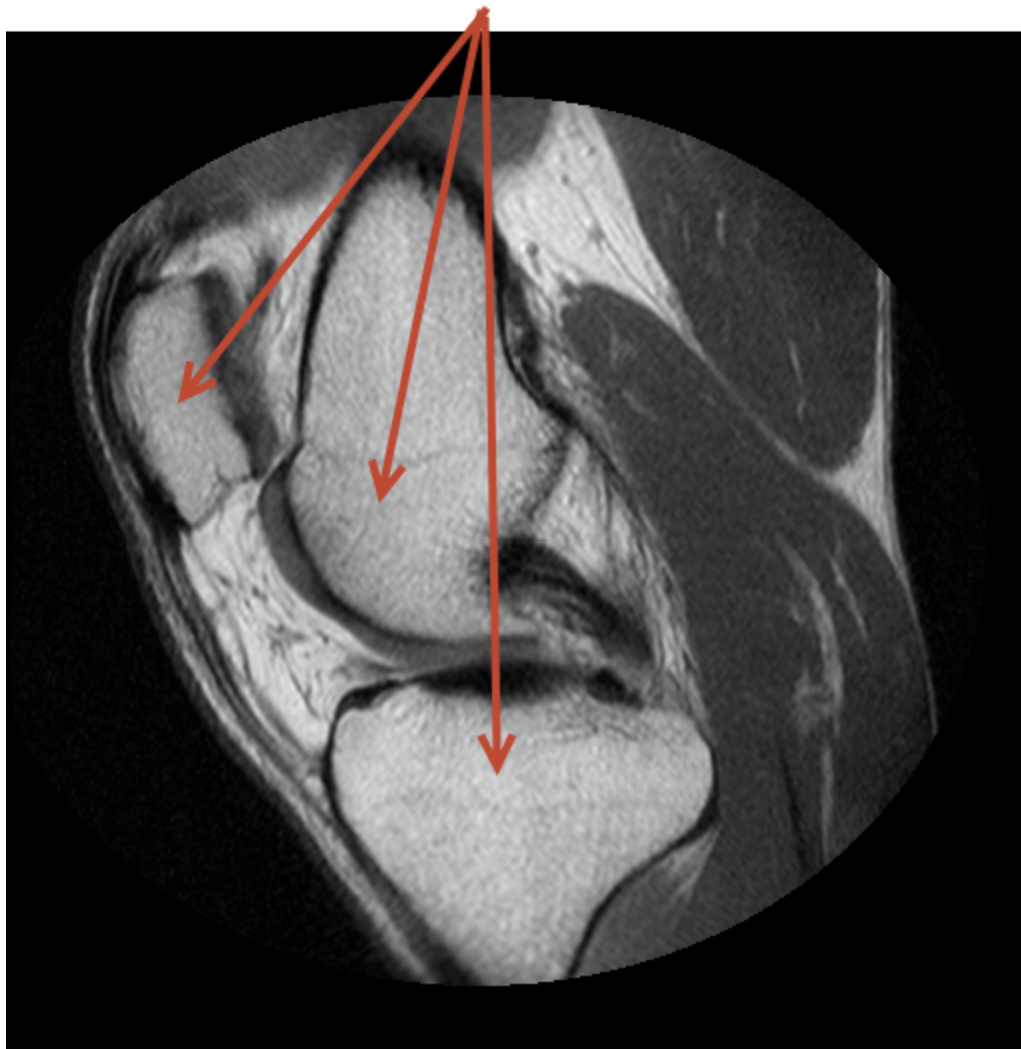
Open Image Segmenter App

This example shows how to open the Image Segmenter app and load an image.


Read an image into the MATLAB workspace. The example uses the `dicomread` function to import MRI data of a knee joint. This example segments the three bony areas of the image from the surrounding soft tissue.

```
I = dicomread('knee1');  
imshow(I, [])
```

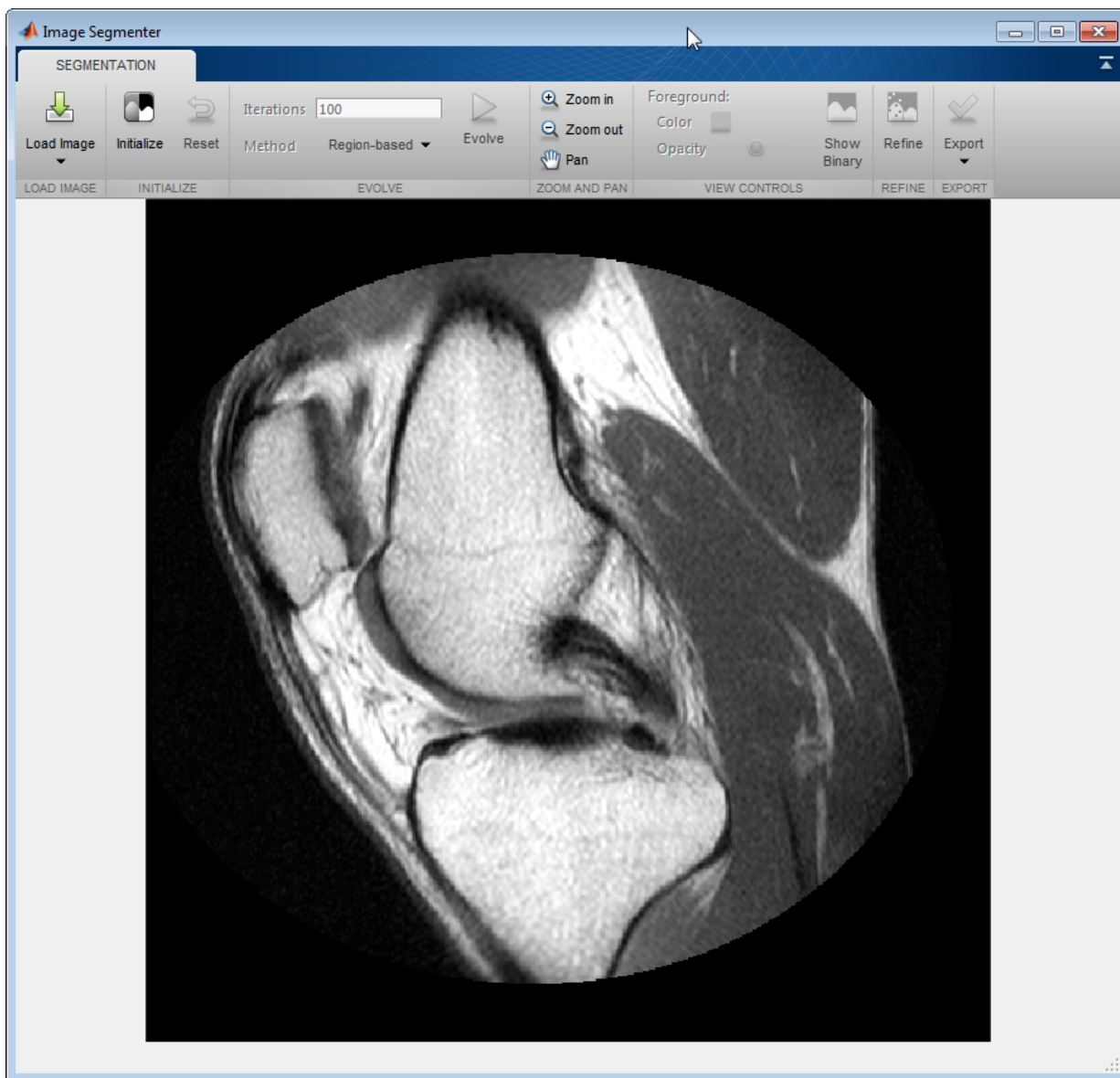
Segment the bones from the soft tissue.



To segment the knee image, open the Image Segmenter app, From the MATLAB Toolstrip, open the Apps tab and under Image Processing and Computer Vision, click

Image Segmenter . Click **Load Image** to bring the knee image into the app. You can also open the app at the command line using the `imageSegmenter` command.

```
imageSegmenter(I)
```

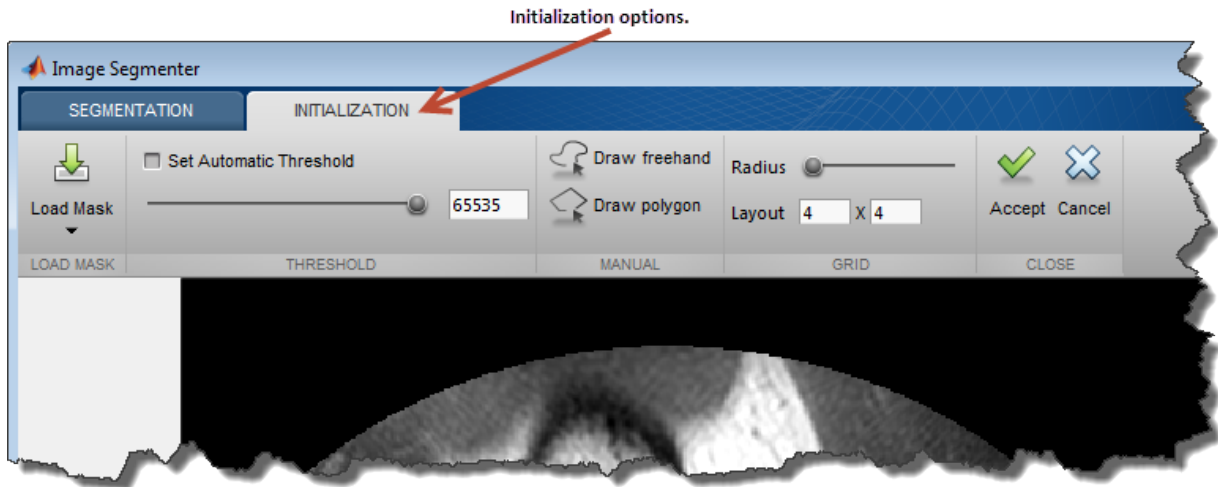


Create the Initial Segmentation

This example shows how to create the initial segmentation. When using the Image Segmenter app, you must create an initial segmentation of the image in which you

identify the objects in the image that you want to segment and define the starting points used by the active contours algorithm when it evolves this segmentation.

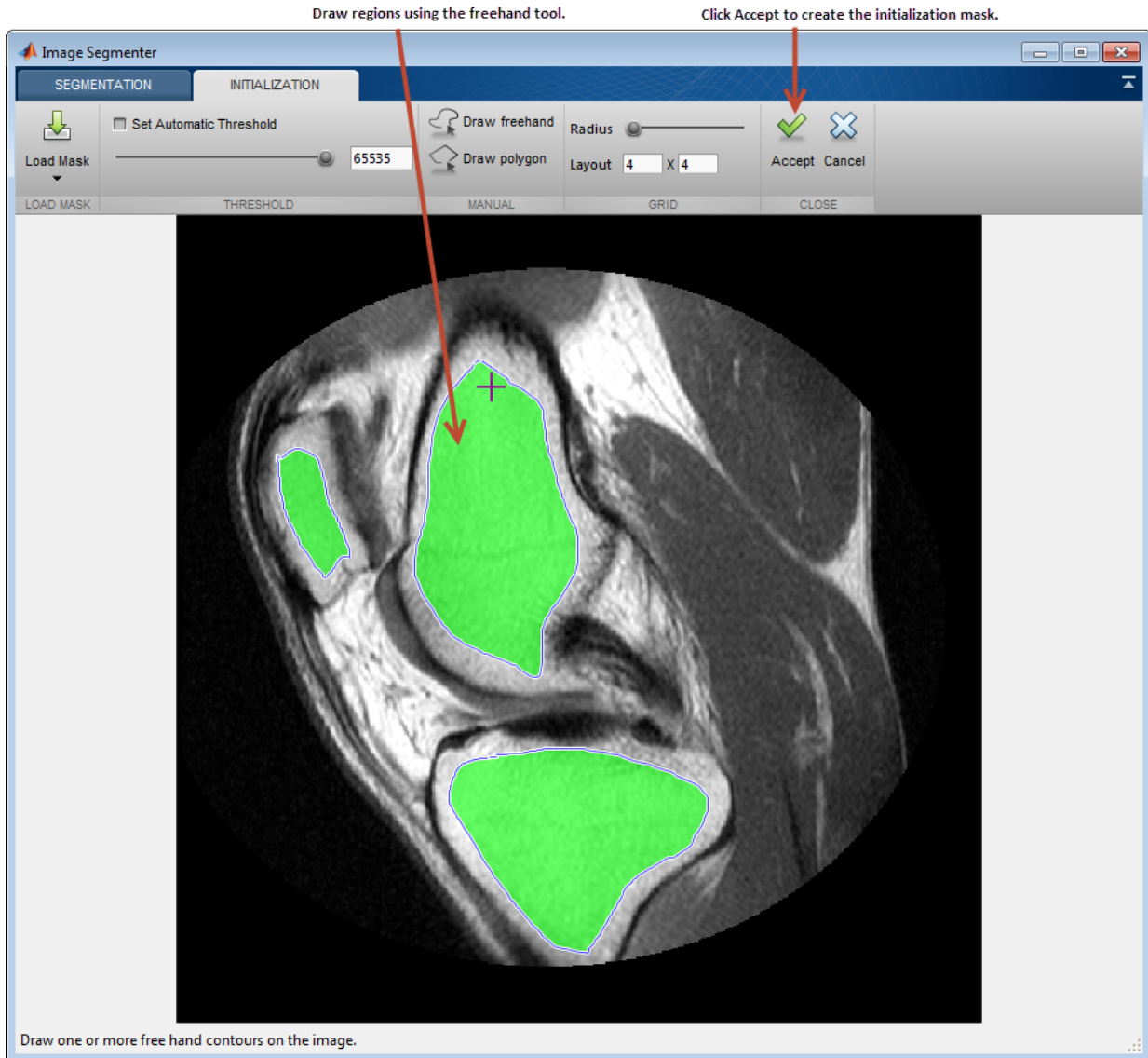
To create the initial segmentation, click **Initialize**. The app opens the **Initialization** tab.



The Initialization tab provides several ways to create the initial segmentation. Pick the one that works best with your image.

- Load a predefined mask image from the workspace or a binary image file. Click **Load Mask**.
- Use thresholding to create a mask. Sometimes, the objects you want to segment in the image have similar pixel intensity values and these values are easily distinguished from other areas of the image, such as the background. You can specify an intensity value, use the slider to select an intensity value interactively, or let the app pick the intensity value for you, by setting **Set Automatic Threshold**.
- Create a mask that is a grid of circular regions that cover the entire image. You control how many elements are in the grid and the size of each element. This method can be useful when the image has many regions to be segmented that are distinct from the background.
- Create a mask interactively by drawing regions in the objects you want to segment. You can draw freehand regions or polygonal regions.

Because the knee image does not have well-defined pixel intensity differences between foreground and background, and contains many other objects besides the bony objects, this example uses the **Draw freehand** option to create the initial segmentation. When you have drawn the regions in the objects, click **Accept**. The app closes the **Initialization** tab and returns you to the Segmentation tab with the initialization mask in place.

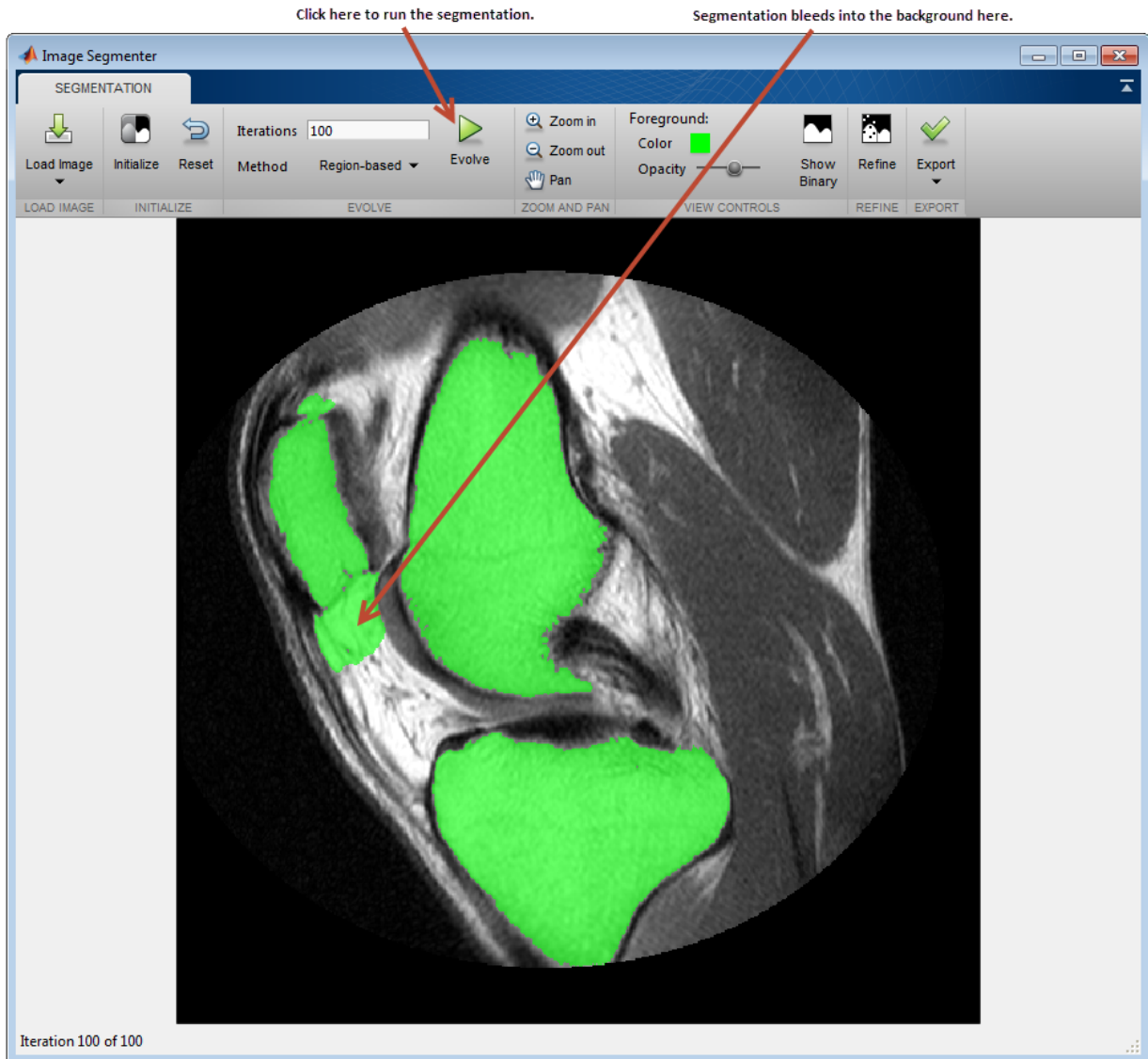


Evolve and Refine the Segmentation

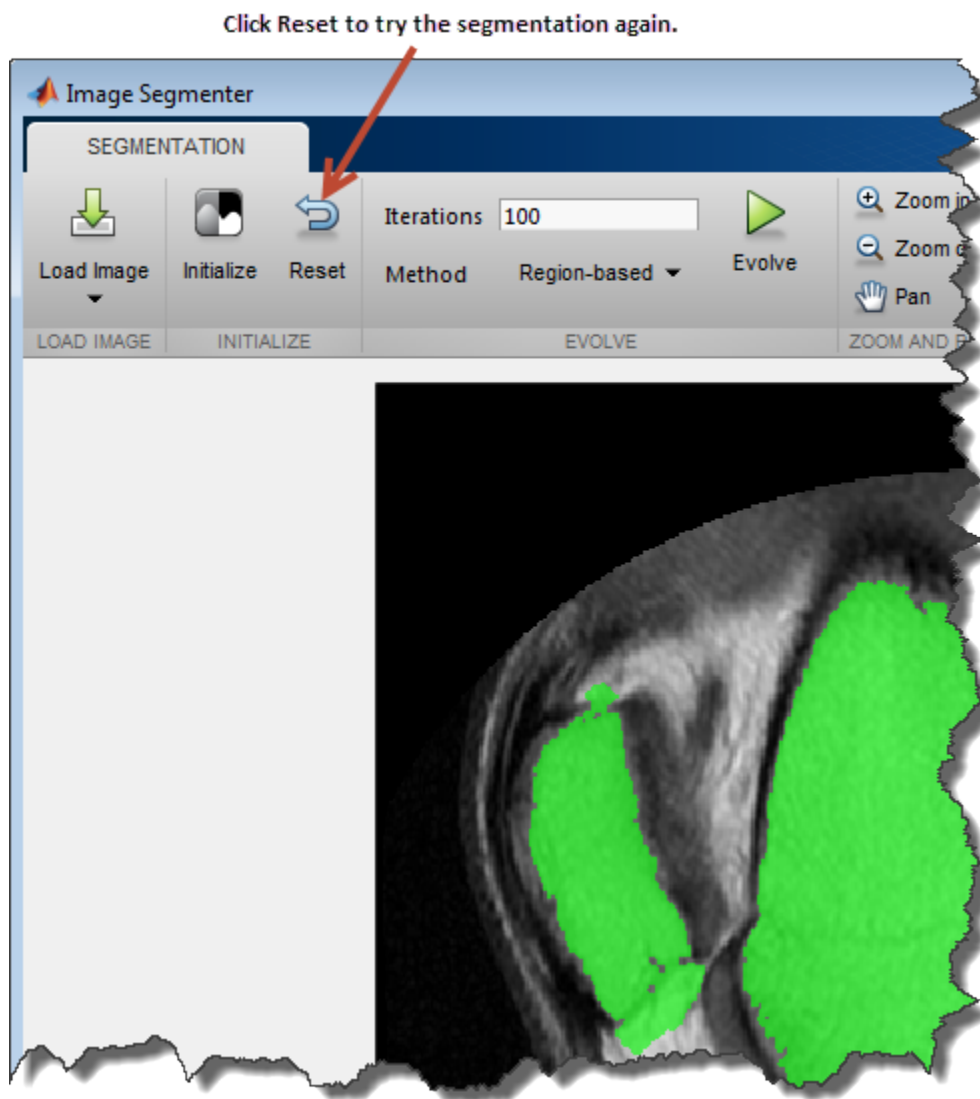
This example shows how to evolve the segmentation of the image using the initial segmentation that you created. The Image Segmenter app uses the active contours

algorithm, which is an iterative algorithm. You can specify the number of iterations and the method used, Region-based or Edge-based. You can experiment with these settings until you achieve the desired segmentation.

Run the segmentation by clicking **Evolve**. For a first pass, the example uses the default method (Region-based) and the default number of iterations. Looking at the results, you can see that this approach worked for two of the three objects but the segmentation bled into the background for one of the objects. The object boundary isn't as well defined in this area.

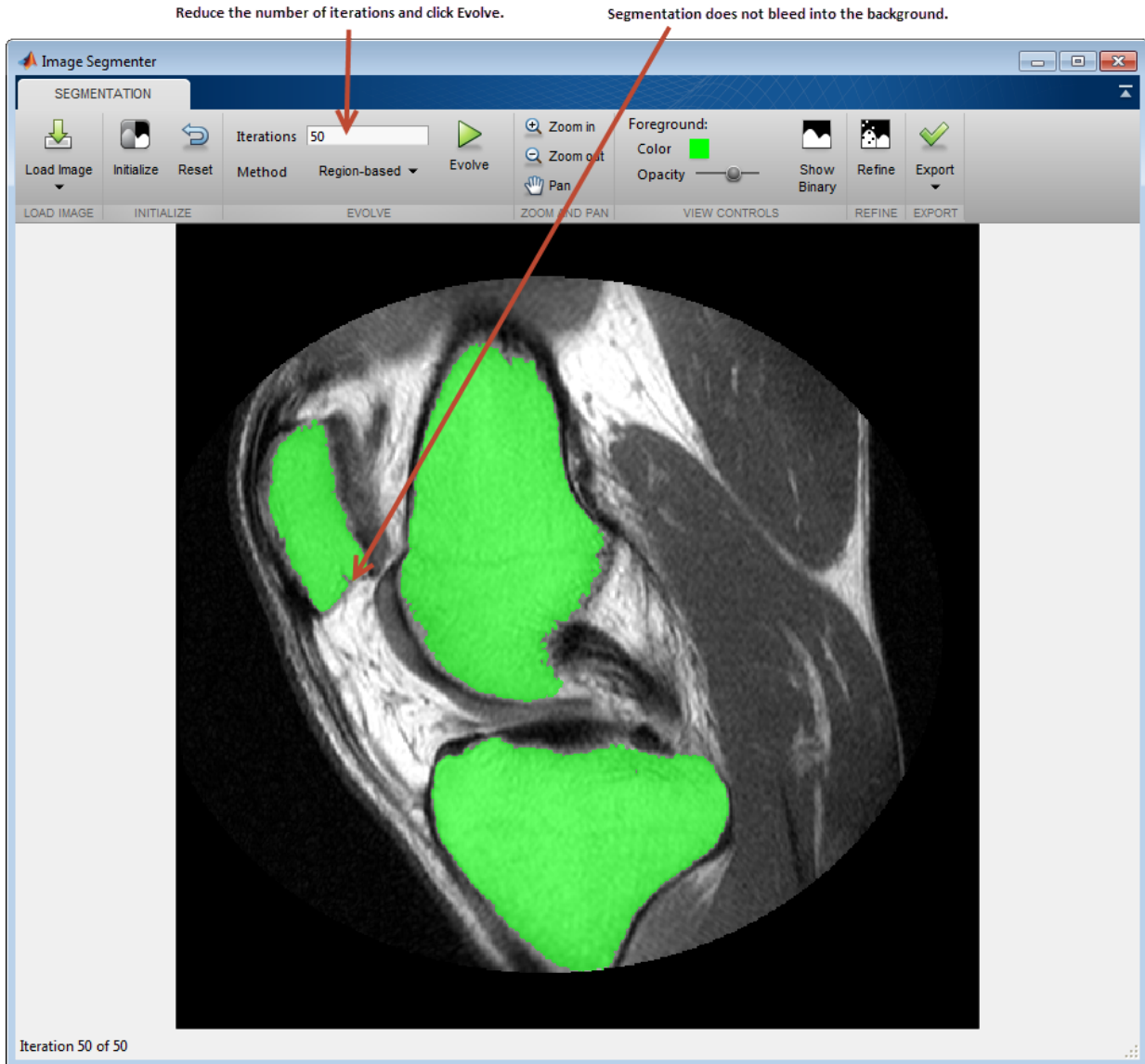


To try the segmentation again, click **Reset**. This option reverts the image back to the initial segmentation. Use **Reset** if you want to experiment with different segmentation options like the number iterations or method used.

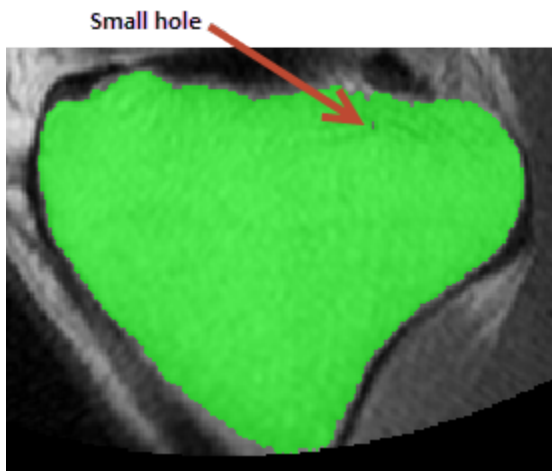


One way to improve this result is to run the segmentation again, this time reducing the number of iterations. By performing fewer iterations, you can stop the segmentation before it goes over the object boundary. On the Segmentation tab, reduce the number of

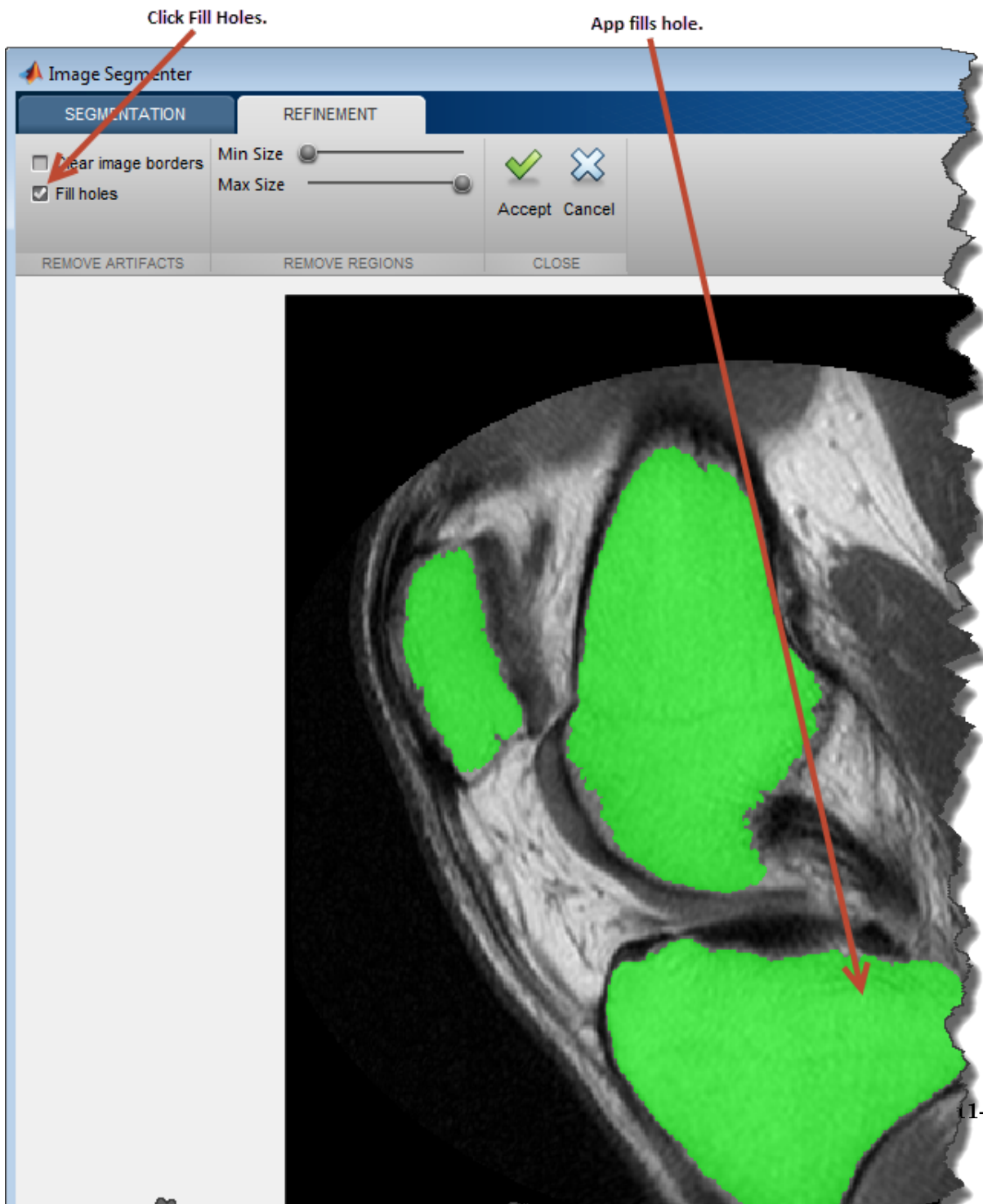
iterations by half (50) and click **Evolve**. The results of this segmentation do not bleed over the object boundary.



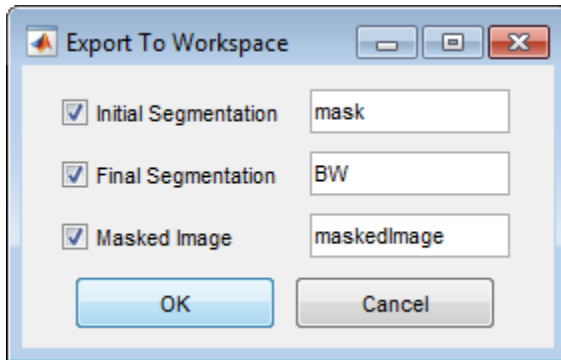
To improve the initialization mask, click **Refine**. On the Refinement tab you can eliminate small or large objects, fill holes, or clear objects that touch the border. For example, in the segmentation, one of the objects has a very small hole.



To fill this hole, click **Fill Holes**. To save the changes you make, click **Accept**.



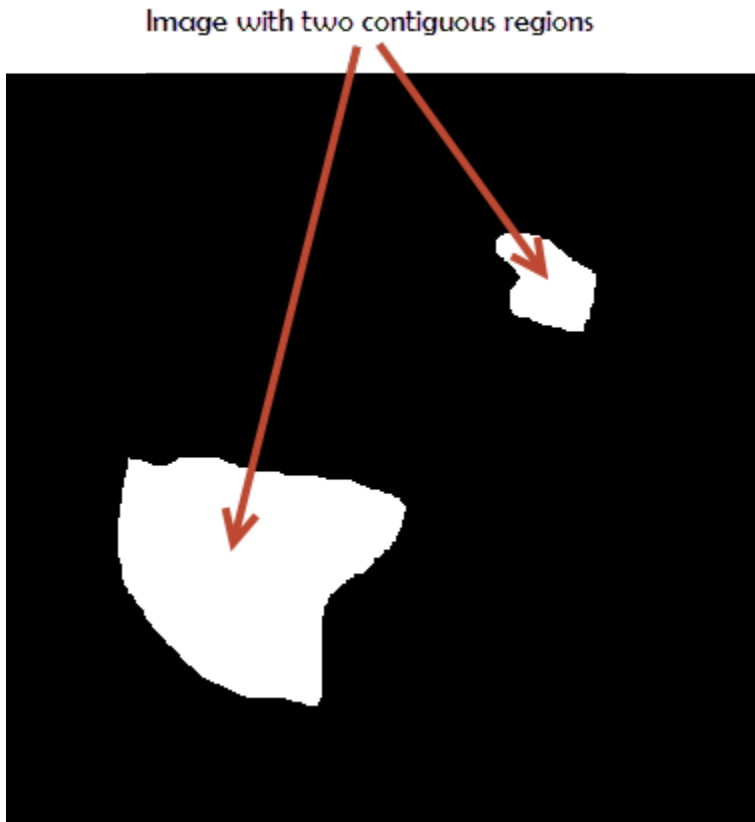
When you achieve the segmentation you desire, you can create a mask image. Click **Export** and select **Export Images**. In the Export to Workspace dialog box, you can assign names to the initial segmentation mask image, the evolved segmentation mask image, or a segmented version of the original image.



To get the MATLAB code the app used to segment the image, click **Export** and select the **Export Function** option. The app opens the MATLAB editor containing a function with the code required to segment the image.

Image Region Properties

Image regions, also called *objects*, *connected components*, or *blobs*, can be contiguous or discontinuous. The following figure shows a binary image with two contiguous regions.



A region in an image can have properties, such as an area, center of mass, orientation, and bounding box. To calculate these properties for regions (and many more) in an image, you can use the Image Region Analyzer app or the `regionprops` function.

Calculate Region Properties Using Image Region Analyzer

This example shows how to calculate the properties of regions in binary images using the Image Region Analyzer app. This example finds the 10 largest regions in the image as measured by their area.

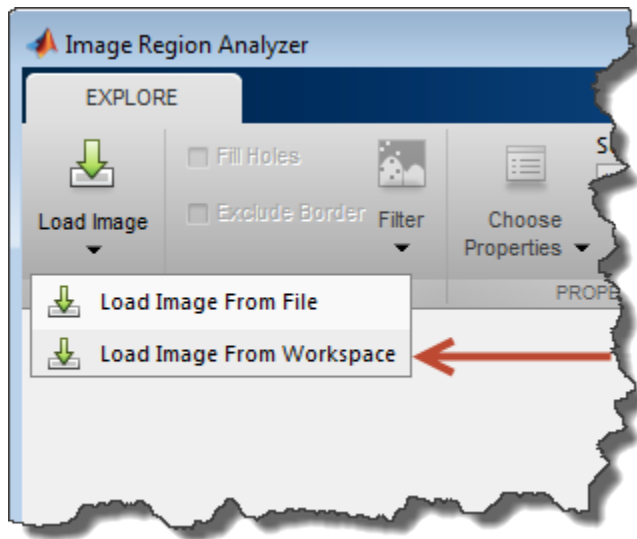
Read a binary image into the MATLAB workspace.

```
BW = imread('text.png');
```

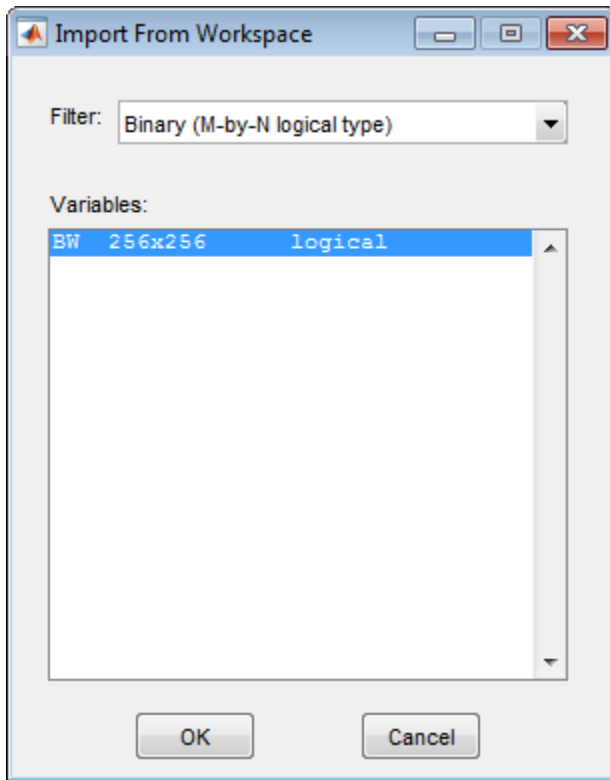
Open the Image Region Analyzer app from the MATLAB Toolstrip. On the Apps tab, in

the Image Processing and Computer Vision group, click .

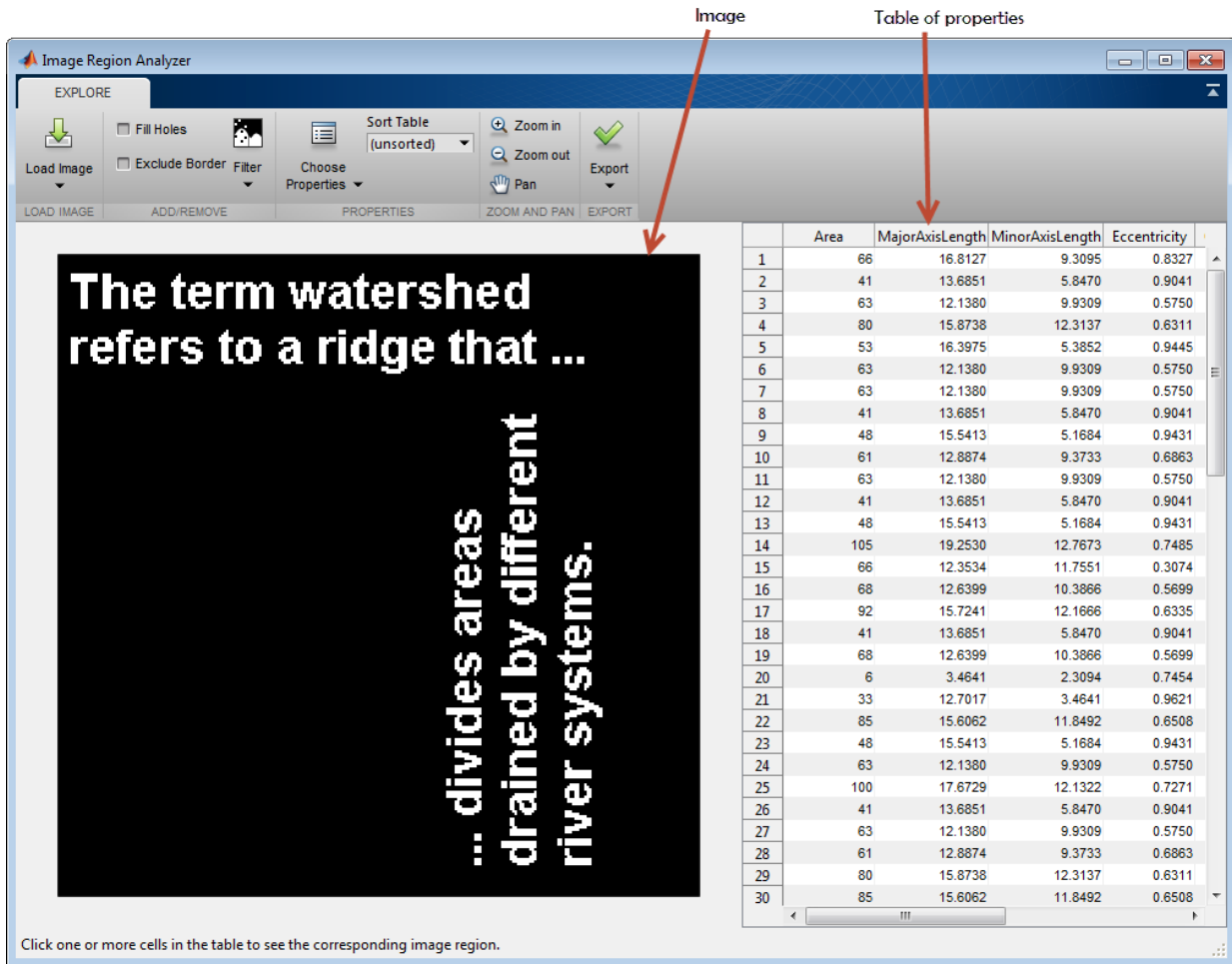
In the Image Region Analyzer app, click **Load Image**, and then select **Load Image from Workspace**, since you have already read the image into the workspace.



In the Import From Workspace dialog box, select the image you read into the workspace, and click **OK**.



The Image Region Analyzer app displays the image you selected next to a table where every row is a region identified in the image and every column is a property of that region, such as the area of the region, perimeter, and orientation. (The Image Region Analyzer app uses `regionprops` to identify regions in the image and calculate properties of these regions.)



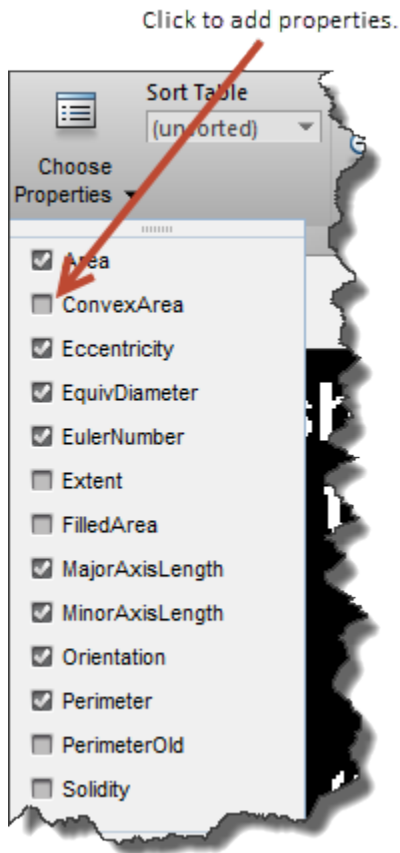
Image

Table of properties

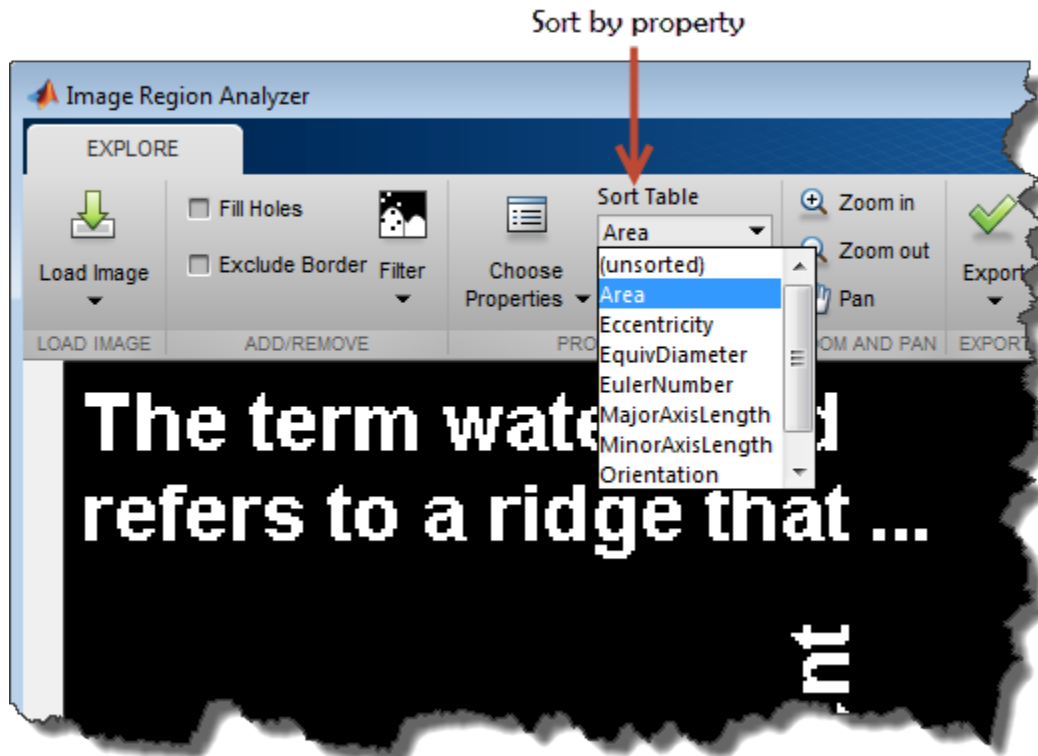
	Area	MajorAxisLength	MinorAxisLength	Eccentricity
1	66	16.8127	9.3095	0.8327
2	41	13.6851	5.8470	0.9041
3	63	12.1380	9.9309	0.5750
4	80	15.8738	12.3137	0.6311
5	53	16.3975	5.3852	0.9445
6	63	12.1380	9.9309	0.5750
7	63	12.1380	9.9309	0.5750
8	41	13.6851	5.8470	0.9041
9	48	15.5413	5.1684	0.9431
10	61	12.8874	9.3733	0.6863
11	63	12.1380	9.9309	0.5750
12	41	13.6851	5.8470	0.9041
13	48	15.5413	5.1684	0.9431
14	105	19.2530	12.7673	0.7485
15	66	12.3534	11.7551	0.3074
16	68	12.6399	10.3866	0.5699
17	92	15.7241	12.1666	0.6335
18	41	13.6851	5.8470	0.9041
19	68	12.6399	10.3866	0.5699
20	6	3.4641	2.3094	0.7454
21	33	12.7017	3.4641	0.9621
22	85	15.6062	11.8492	0.6508
23	48	15.5413	5.1684	0.9431
24	63	12.1380	9.9309	0.5750
25	100	17.6729	12.1322	0.7271
26	41	13.6851	5.8470	0.9041
27	63	12.1380	9.9309	0.5750
28	61	12.8874	9.3733	0.6863
29	80	15.8738	12.3137	0.6311
30	85	15.6062	11.8492	0.6508

Click one or more cells in the table to see the corresponding image region.

The app calculates more properties than are displayed initially. To view more properties in the table, click **Choose Properties** and select the properties you want to view. Properties displayed are marked with a check. The app updates the table automatically, adding a new column to the table.



To explore the image, sort the information in the table. For example, if you sort on Area property, the table lists the regions in order by size. Click the **Sort Table** button in the Properties group and select the property you want to sort on.

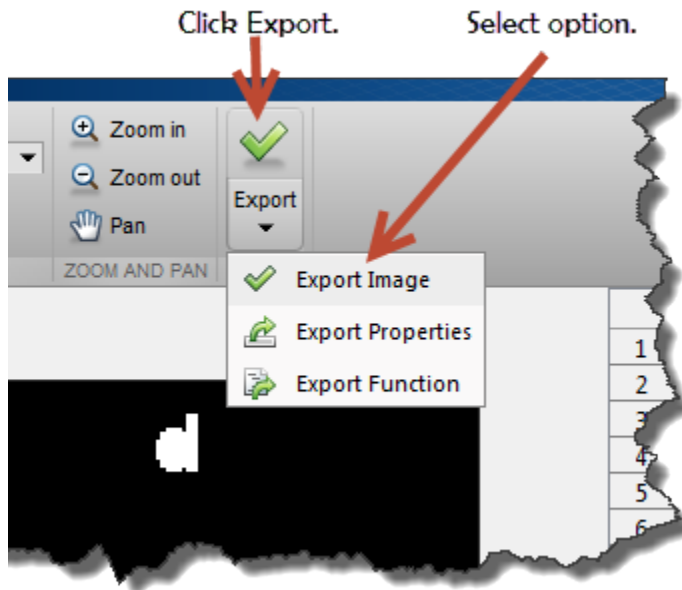


To view the region in the image with the largest area, click the item in the table. The app highlights the corresponding region in the image.

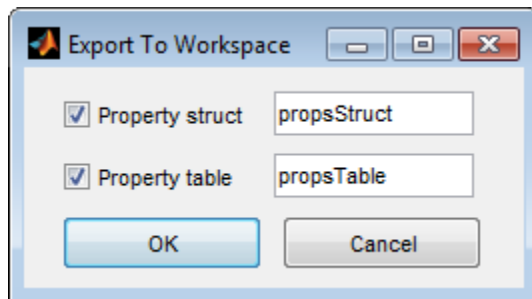
The screenshot shows the 'Image Region Analyzer' application window. The interface includes a toolbar with options like 'Load Image', 'Fill Holes', 'Exclude Border', 'Filter', 'Choose Properties', 'Sort Table' (set to 'Area'), 'Zoom in', 'Zoom out', 'Pan', and 'Export'. A table on the right lists 22 regions with columns for 'Area' and 'MajorAxisLength'. The first row is highlighted in blue. A red arrow points from the text 'Select item to highlight in image.' to the first row of the table. Another red arrow points from the text 'wides areas' to the first row of the table. A large black text overlay on the left reads: 'The term watershed refers to a ridge that ...' and 'wides areas ned by different systems.'

	Area	MajorAxisLength
1	106	16.5
2	105	19.253
3	105	19.2
4	100	17.6
5	92	15.7
6	85	15.60
7	85	15.6062
8	85	15.6
9	85	15.606
10	85	15.6
11	85	15.6
12	85	15.60
13	85	15.60
14	80	15.8738
15	80	15.873
16	80	15.873
17	74	17.2
18	74	17.28
19	72	13.667
20	72	13.6
21	68	12.65
22	68	12.63

To save this data, click **Export** to see options.



If you want to save the table of region property values in a workspace variable, select **Export Properties**. To save the data in both a structure and a table, click **OK**.



The app stores the data in a MATLAB structure, as in the following.

```
propsStruct(1)
```

```
ans =
```

```
Area: 106
MajorAxisLength: 16.5975
MinorAxisLength: 12.8996
Eccentricity: 0.6292
Orientation: -18.7734
EulerNumber: 0
EquivDiameter: 11.6174
Perimeter: 64.7960
```

The app also stores the data in a MATLAB table, as in the following.

```
propsTable(1,1)
```

```
propsTable(1,:)
```

```
ans =
```


Area	MajorAxisLength	MinorAxisLength	Eccentricity	Orientation	EulerNumber
106	16.598	12.9	0.62925	-18.773	0

Filter Images on Region Properties Using Image Region Analyzer App

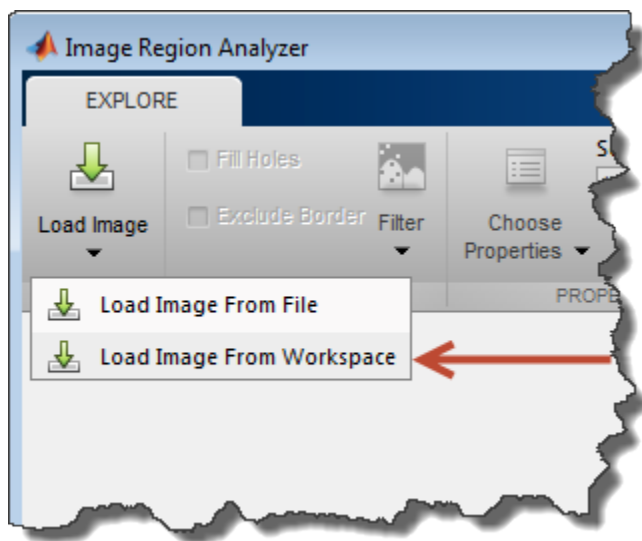
This example shows how to create a new binary image by filtering an existing binary image based on the properties of regions in the image.

Read a binary image into the MATLAB workspace.

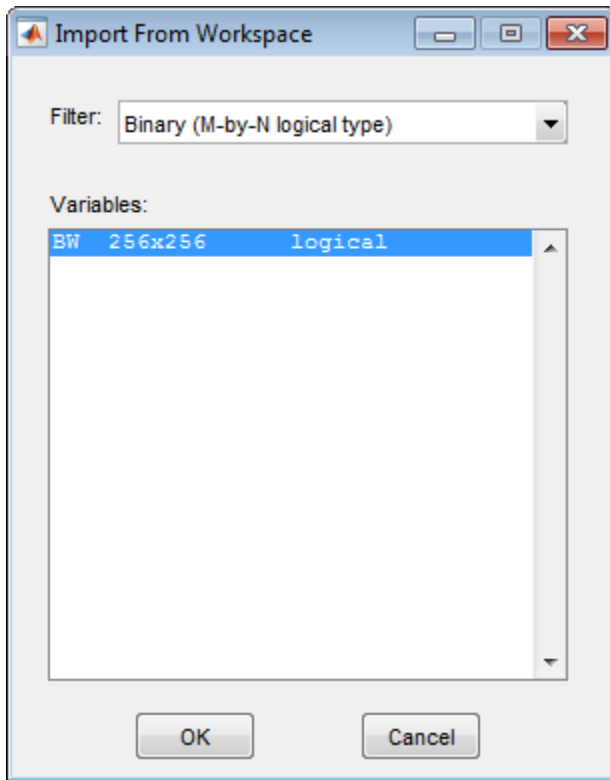
```
BW = imread('text.png');
```

Open the Image Region Analyzer app from the MATLAB Toolstrip. On the Apps tab, in the Image Processing and Computer Vision group, click .

In the Image Region Analyzer app, click **Load Image**, and then select **Load Image from Workspace**, since you have already read the image into the workspace.



In the Import From Workspace dialog box, select the image you read into the workspace, and click **OK**.



The Image Region Analyzer app displays the image you selected next to a table where every row is a region identified in the image and every column is a property of that region, such as the area, perimeter, and orientation. The Image Region Analyzer app uses `regionprops` to identify regions in the image and calculate properties of these regions.

Image Region Analyzer

EXPLORE

Load Image

Fill Holes

Exclude Border

Filter

Choose Properties

Sort Table (unsorted)

Zoom in

Zoom out

Pan

Export

LOAD IMAGE

ADD/REMOVE

PROPERTIES

ZOOM AND PAN

EXPORT

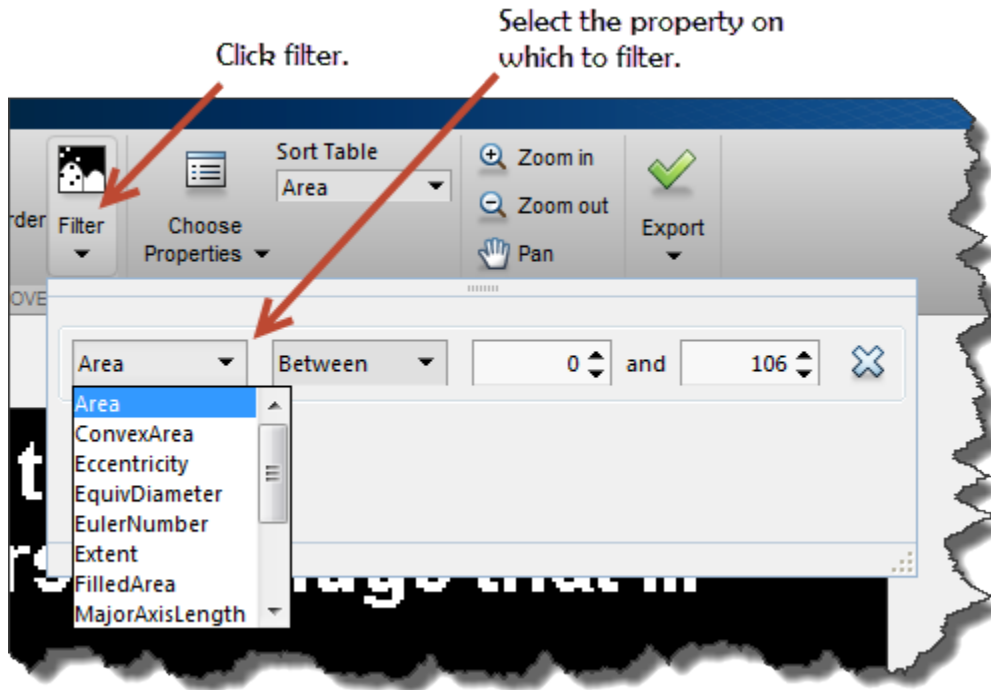
The term watershed refers to a ridge that ...

... divides areas drained by different river systems.

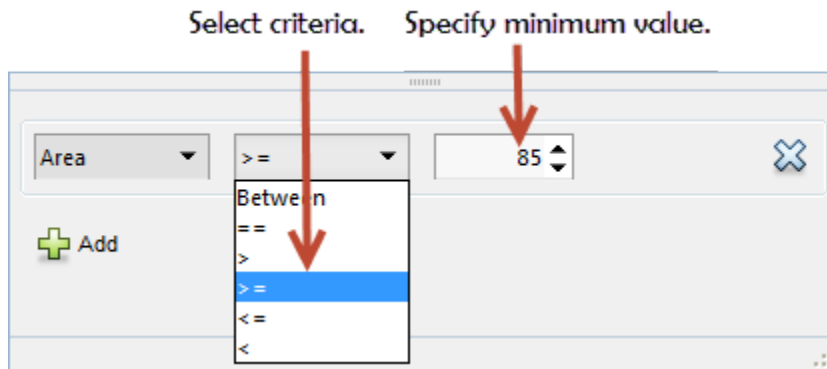
	Area	MajorAxisLength	MinorAxisLength	Eccentricity
1	66	16.8127	9.3095	0.8327
2	41	13.6851	5.8470	0.9041
3	63	12.1380	9.9309	0.5750
4	80	15.8738	12.3137	0.6311
5	53	16.3975	5.3852	0.9445
6	63	12.1380	9.9309	0.5750
7	63	12.1380	9.9309	0.5750
8	41	13.6851	5.8470	0.9041
9	48	15.5413	5.1684	0.9431
10	61	12.8874	9.3733	0.6863
11	63	12.1380	9.9309	0.5750
12	41	13.6851	5.8470	0.9041
13	48	15.5413	5.1684	0.9431
14	105	19.2530	12.7673	0.7485
15	66	12.3534	11.7551	0.3074
16	68	12.6399	10.3866	0.5699
17	92	15.7241	12.1666	0.6335
18	41	13.6851	5.8470	0.9041
19	68	12.6399	10.3866	0.5699
20	6	3.4641	2.3094	0.7454
21	33	12.7017	3.4641	0.9621
22	85	15.6062	11.8492	0.6508
23	48	15.5413	5.1684	0.9431
24	63	12.1380	9.9309	0.5750
25	100	17.6729	12.1322	0.7271
26	41	13.6851	5.8470	0.9041
27	63	12.1380	9.9309	0.5750
28	61	12.8874	9.3733	0.6863
29	80	15.8738	12.3137	0.6311
30	85	15.6062	11.8492	0.6508

Click one or more cells in the table to see the corresponding image region.

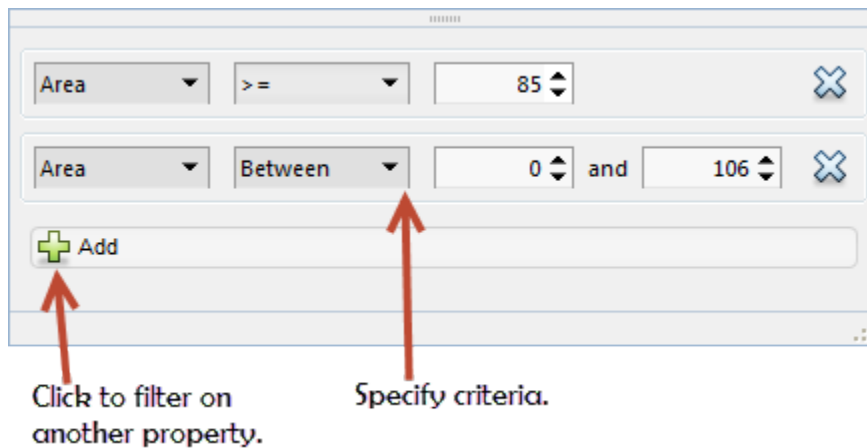
To filter on the value of a region property, click **Filter** and select the property on which you want to filter.



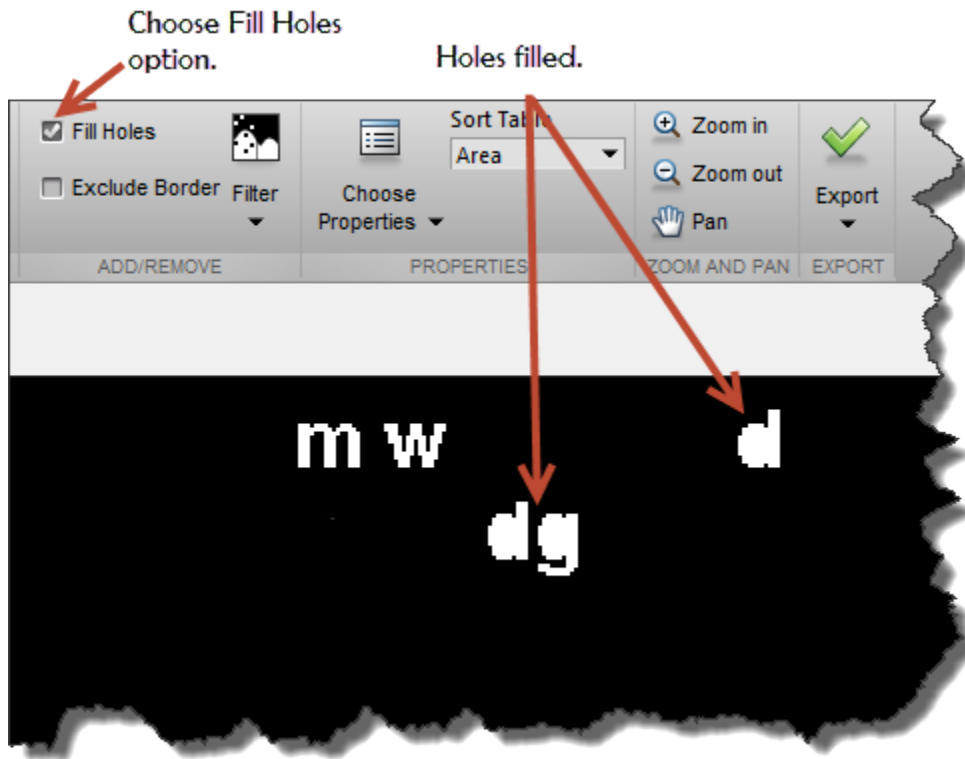
Next, specify the filter criteria. For example, to create an image that removes all but the largest regions, choose the greater than or equal to symbol (\geq) and then specify the minimum value. Sometimes it's useful to sort the values in the table by the property you are interested in to determine what the minimum value should be. The app changes the elements of this dialog box, depending on which criteria you pick. The app uses the `bwpropfilt` and `bwareafilt` function to filter binary images.



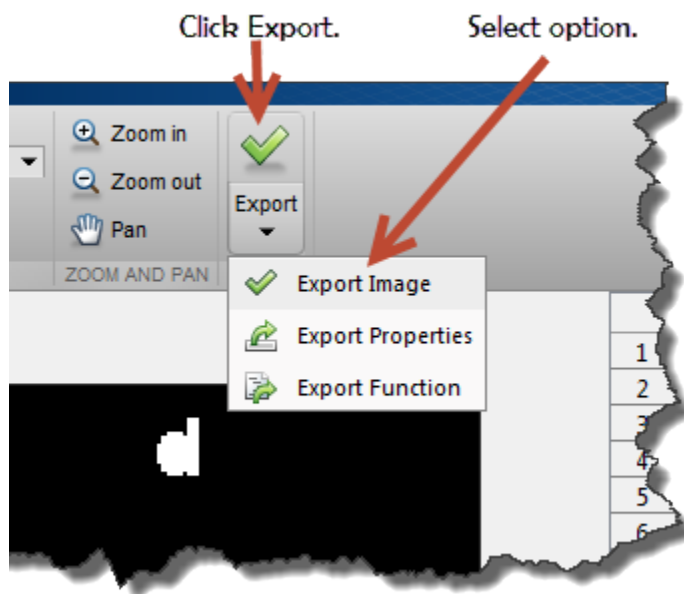
To filter on another property, click **Add**. The app displays another row in which you can select a property and specify filter criteria.



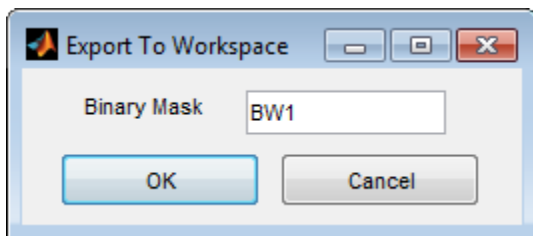
If you are creating a mask image, you can optionally perform some clean up operations on the mask, such as clearing all foreground pixels that touch the border and filling holes in objects.



When you are done filtering the image, you can save it. Click **Export** and select **Export Image**.



In the Export to Workspace dialog box, accept the default name for the mask image, or specify another name, and click **OK**.



ROI-Based Processing

This chapter describes how to define a region of interest (ROI) and perform processing on the ROI you define.

- “Specifying a Region of Interest (ROI)” on page 12-2
- “Filtering an ROI” on page 12-5
- “Filling an ROI” on page 12-8

Specifying a Region of Interest (ROI)

In this section...

“Overview of ROI Processing” on page 12-2

“Creating a Binary Mask” on page 12-2

“Creating an ROI Without an Associated Image” on page 12-3

“Creating an ROI Based on Color Values” on page 12-4

Overview of ROI Processing

A *region of interest* (ROI) is a portion of an image that you want to filter or perform some other operation on. You define an ROI by creating a *binary mask*, which is a binary image that is the same size as the image you want to process with pixels that define the ROI set to 1 and all other pixels set to 0.

You can define more than one ROI 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.

Using Binary Images as a Mask

This section describes how to create binary masks to define ROIs. However, any binary image can be used 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)`.

Creating a Binary Mask

You can use the `createMask` method of the `imroi` base class to create a binary mask for any type of ROI object — `impoint`, `imline`, `imrect`, `imellipse`, `impoly`, or `imfreehand`. The `createMask` method returns a binary image the same size as the input image, containing 1s inside the ROI and 0s everywhere else.

The example below illustrates how to use the `createMask` method:

```
img = imread('pout.tif');  
h_img = imshow(img);
```

```
e = imellipse(gca,[55 10 120 120]);
BW = createMask(e,h_im);
```



You can reposition the mask by dragging it with the mouse. Right click, select **copy position**, and call `createMask` again.

```
BW = createMask(e,h_im);
```

The mask behaves like a cookie cutter and can be repositioned repeatedly to select new ROIs.

Creating an ROI Without an Associated Image

Using the `createMask` method of ROI objects you can create a binary mask that defines an ROI associated with a particular image. To create a binary mask without having an associated image, use the `poly2mask` function. Unlike the `createMask` method, `poly2mask` does not require an input image. You specify the vertices of the ROI in two vectors and specify the size of the binary mask returned. For example, the following creates a binary mask that can be used to filter an ROI in the `pout.tif` image.

```
c = [123 123 170 170];
r = [160 210 210 160];
```

```
m = 291; % height of pout image
n = 240; % width of pout image
BW = poly2mask(c,r,m,n);
figure, imshow(BW)
```

Creating an ROI Based on Color Values

You can use the `roicolor` function to define an ROI based on color or intensity range. For more information, see the reference page for `roicolor`.

Filtering an ROI

In this section...

“Overview of ROI Filtering” on page 12-5

“Filtering a Region in an Image” on page 12-5

“Specifying the Filtering Operation” on page 12-6

Overview of ROI Filtering

Filtering a region of interest (ROI) is the process of applying a filter to a region 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 an ROI in an image, use the `roifilt2` function. When you call `roifilt2`, you specify:

- Input grayscale image to be filtered
- Binary mask image that defines the ROI
- Filter (either a 2-D filter or function)

`roifilt2` filters the input image and returns an image that consists of filtered values for pixels where the binary mask contains 1s and unfiltered values for pixels where the binary mask contains 0s. This type of operation is called *masked filtering*.

`roifilt2` is best suited for 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`.

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 by the `createMask` method in the section “Creating a Binary Mask” on page 12-2. The region of interest specified is the child's face.

- 3 Use `fspecial` to create the filter:

```
h = fspecial('unsharp');
```

- 4 Call `roifilt2`, specifying the filter, the image to be filtered, and the mask:

```
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 ROI. 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 function you want to use as a 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)
```

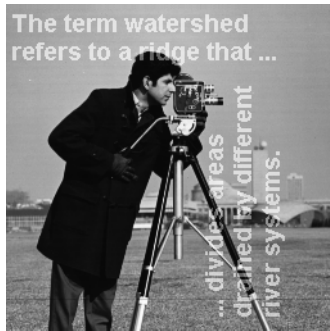


Image Brightened Using a Binary Mask Containing Text

Filling an ROI

Filling is a process that fills a region of interest (ROI) 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 an ROI, 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.

Select the region of interest with the mouse. When you complete the selection, `roifill` returns an image with the selected ROI filled in.

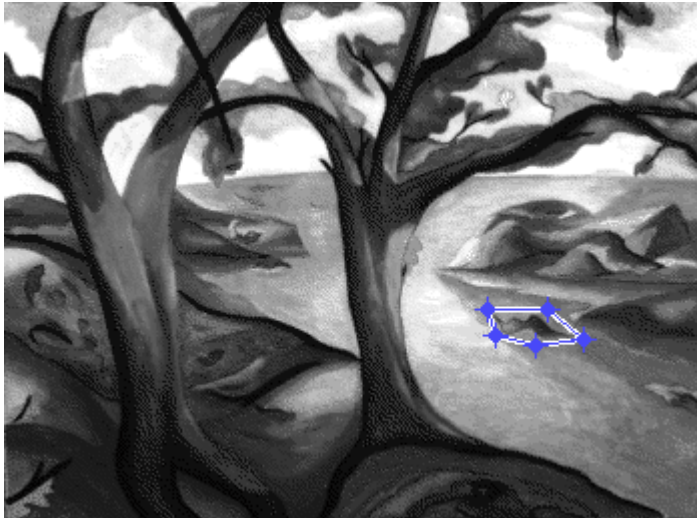
This example shows how to use `roifill` to fill an ROI in an image.

- 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` to specify the ROI you want to fill. When you move the pointer over the image, the pointer shape changes to cross hairs . Define the ROI by clicking the mouse to specify the vertices of a polygon. You can use the mouse to adjust the size and position of the ROI:

```
I2 = roifill;
```



- 3 Perform the fill operation. Double-click inside the ROI or right-click and select **Fill Area**. `roifill` returns the modified image in `I2`. View the output image to see how `roifill` filled in the area defined by the ROI:

```
imshow(I2)
```



Image Deblurring

This chapter describes how to deblur an image using the toolbox deblurring functions.

- “Understanding Deblurring” on page 13-2
- “Deblurring with the Wiener Filter” on page 13-6
- “Deblurring with a Regularized Filter” on page 13-7
- “Deblurring with the Lucy-Richardson Algorithm” on page 13-9
- “Deblurring with the Blind Deconvolution Algorithm” on page 13-14
- “Creating Your Own Deblurring Functions” on page 13-20
- “Avoiding Ringing in Deblurred Images” on page 13-21

Understanding Deblurring

In this section...

“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. All the functions accept a PSF and the blurred image as their primary arguments.

<code>deconvwnr</code>	Implements a least squares solution. You should provide some information about the noise to reduce possible noise amplification during deblurring. See “Deblurring with the Wiener Filter” on page 13-6 for more information.
<code>deconvreg</code>	Implements a constrained least squares solution, where you can place constraints on the output image (the smoothness requirement is the default). You should provide some information about the noise to reduce possible noise amplification during deblurring. See “Deblurring with a Regularized Filter” on page 13-7 for more information.
<code>deconvlucy</code>	Implements an accelerated, damped Lucy-Richardson algorithm. This function performs multiple iterations, using optimization techniques and Poisson statistics. You do not need to provide information about the additive noise in the corrupted image. See “Deblurring with the Lucy-Richardson Algorithm” on page 13-9 for more information.
<code>deconvblind</code>	Implements the blind deconvolution algorithm, which performs deblurring without knowledge of the PSF. You pass as an argument your initial guess at the PSF. The <code>deconvblind</code> function returns a restored PSF in addition to the restored image. The implementation uses the same damping and iterative model as the <code>deconvlucy</code> function. See “Deblurring with the Blind Deconvolution Algorithm” on page 13-14 for more information.

When using the deblurring functions, note the following:

- Deblurring is an iterative process. You might need to repeat the deblurring process multiple times, 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.
- 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 13-21 for more information.
- For information about creating your own deblurring functions, see "Creating Your Own Deblurring Functions" on page 13-20.

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. See the `deconvwnr` reference page for an example of deblurring an image with the Wiener filter.

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 examples.

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')
```

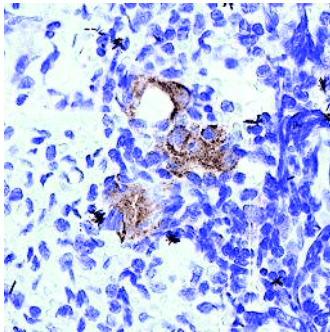
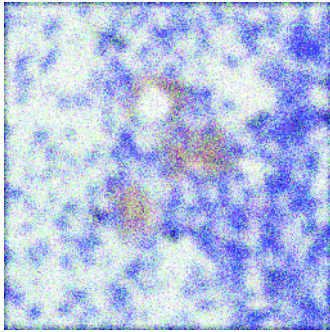


Image Courtesy Alan W. Partin

- 2 Create the PSF.
- 3 Create a simulated blur in the image and add noise.

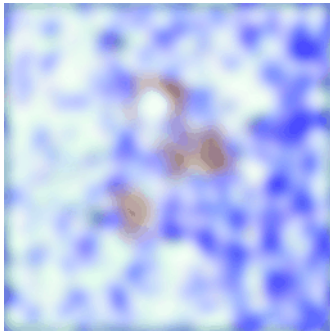
```
Blurred = imfilter(I,PSF,'conv');

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 examples.

Deblurring with the Lucy-Richardson Algorithm

In this section...

“Overview” on page 13-9

“Reducing the Effect of Noise Amplification” on page 13-9

“Accounting for Nonuniform Image Quality” on page 13-10

“Handling Camera Read-Out Noise” on page 13-10

“Handling Undersampled Images” on page 13-10

“Example: Using the deconvlucy Function to Deblur an Image” on page 13-11

“Refining the Result” on page 13-13

Overview

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.

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 13-21 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.

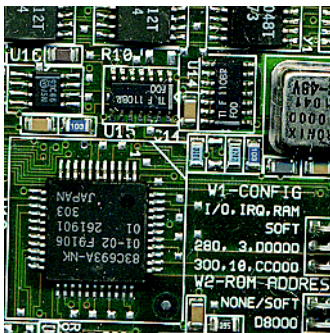
If the undersampled data is the result of camera pixel binning during image acquisition, the PSF observed at each pixel rate can serve as a finer grid PSF. Otherwise, the PSF can be obtained via observations taken at subpixel offsets or via optical modeling techniques. This method is especially effective 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')
```

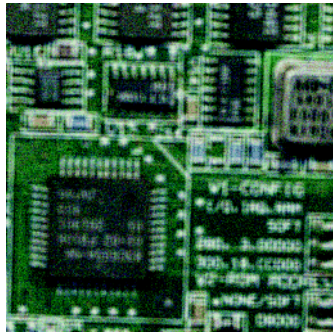


- 2 Create the PSF.


```
PSF = fspecial('gaussian',5,5);
```
- 3 Create a simulated blur in the image and add noise.


```
Blurred = imfilter(I,PSF,'symmetric','conv');
```

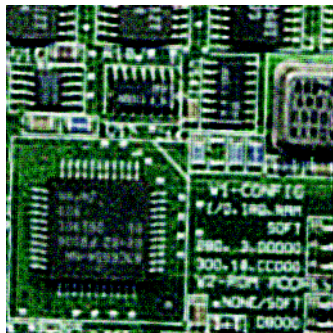
```
V = .002;  
BlurredNoisy = imnoise(Blurred,'gaussian',0,V);  
figure, imshow(BlurredNoisy)  
title('Blurred and Noisy Image')
```



- 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 examples.

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 13-9. 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.

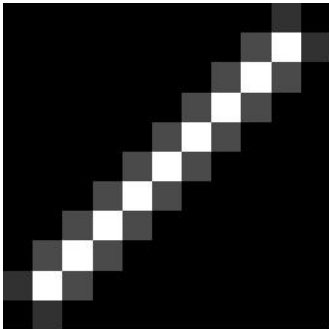
```
I = imread('cameraman.tif');  
figure, imshow(I)  
title('Original Image')
```



Image Courtesy of MIT

- 2 Create the PSF.

```
PSF = fspecial('motion',13,45);  
figure, imshow(PSF,[],'InitialMagnification','fit')  
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')
```

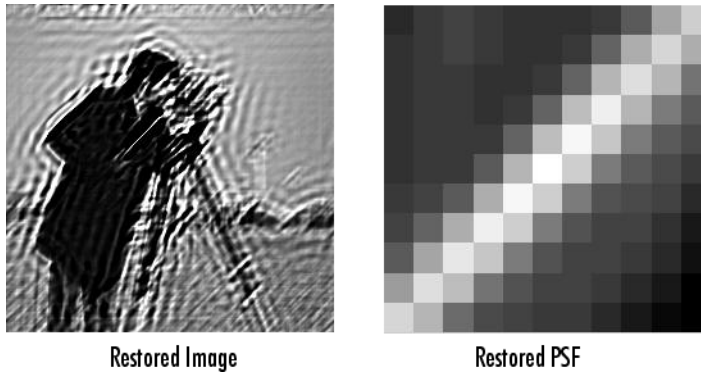


- 4 Deblur the image, making an initial guess at the size of the PSF.

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 example.

```
INITPSF = ones(size(PSF));  
[J P]= deconvblind(Blurred,INITPSF,30);  
figure, imshow(J)  
title('Restored Image')  
figure, imshow(P,[],'InitialMagnification','fit')  
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 “Accounting for Nonuniform Image Quality” on page 13-10 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 “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;

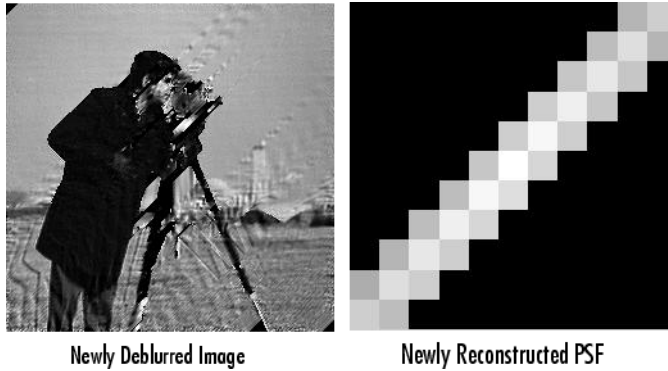
```

- 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,[],'InitialMagnification','fit')
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.

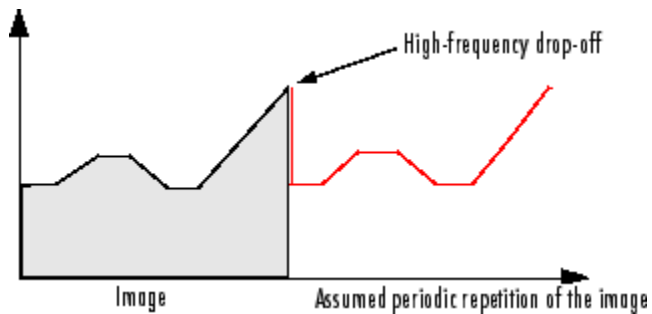
The `deconvblind` function supports several other optional arguments you can use to achieve the best possible result, such as specifying a damping parameter to handle additive noise in the blurred image. To see the impact of these optional arguments, as well as the functional option that allows you to place additional constraints on the PSF reconstruction, see the Image Processing Toolbox deblurring examples.

Creating Your Own Deblurring Functions

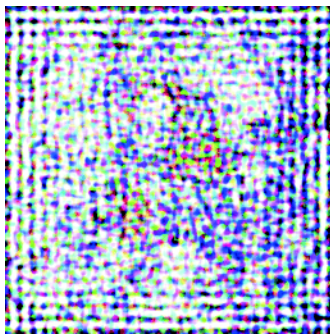
All the toolbox deblurring functions perform deconvolution in the frequency domain, where the process becomes a simple matrix multiplication. To work in the frequency domain, the deblurring functions must convert the PSF you provide into an optical transfer function (OTF), using the `psf2otf` function. 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.

- “Displaying Colors” on page 14-2
- “Reducing the Number of Colors in an Image” on page 14-4
- “Convert from YIQ to RGB Color Space” on page 14-12
- “Convert from YCbCr to RGB Color Space” on page 14-13
- “Convert from HSV to RGB Color Space” on page 14-14
- “Profile-Based Color Space Conversions” on page 14-17
- “Device-Independent Color Spaces” on page 14-21
- “Understanding Color Spaces and Color Space Conversion” on page 14-24
- “Determine if L*a*b* value is in RGB gamut” on page 14-25

Displaying Colors

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.

Most computer displays use 8, 16, or 24 bits per screen pixel. Depending on your system, you might be able to choose the screen bit depth you want to use. 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.

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}).

Value	Screen Bit Depth
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 “Mapping Data to Transparency — Alpha Data” in the MATLAB 3-D Visualization documentation.

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 information about reducing the number of colors used by an image, see “Reducing the Number of Colors in an Image” on page 14-4.

Reducing the Number of Colors in an Image

In this section...

“Reducing Colors Using Color Approximation” on page 14-4

“Reducing Colors Using `imapprox`” on page 14-9

“Dithering” on page 14-10

Reducing Colors Using Color Approximation

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.

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.

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 (described in “Quantization” on page 14-5)
 - Uniform quantization
 - Minimum variance quantization
- Colormap mapping (described in “Colormap Mapping” on page 14-9)

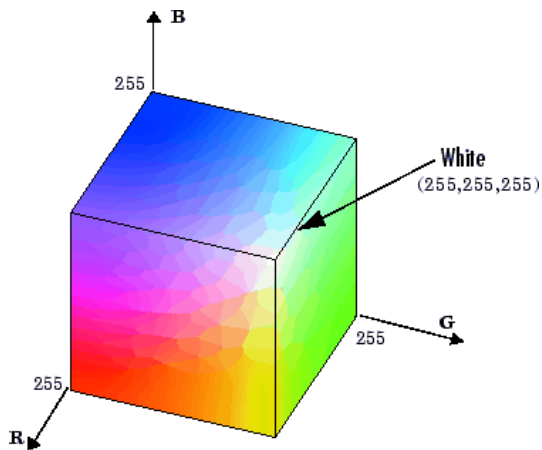
The quality of the resulting image depends on the approximation method you use, the range of colors in the input image, and whether or not you use dithering. Note that different methods work better for different images. See “Dithering” on page 14-10 for a description of dithering and how to enable or 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*. 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

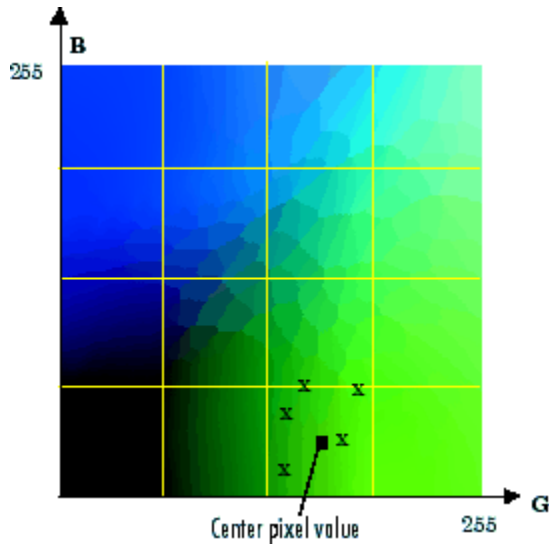
To perform uniform quantization, call `rgb2ind` and specify a *tolerance*. The tolerance determines the size of the cube-shaped boxes into which the RGB color cube is divided. The allowable range for a tolerance setting is [0,1]. For example, if you specify a tolerance of 0.1, the edges of the boxes 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);
```

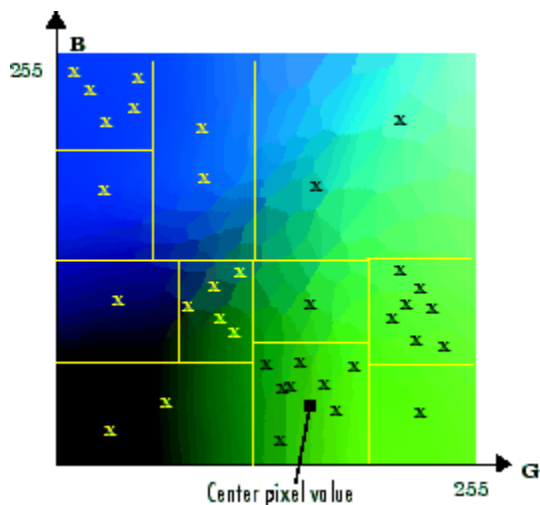
Minimum variance quantization works by associating pixels into groups based on the variance between their pixel values. For example, a set of blue pixels might be grouped together because they have a small variance from the center pixel of the group.

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 4-9 or the reference page for `subimage`.

Reducing Colors Using `imapprox`

Use `imapprox` when you need to reduce the number of colors in an indexed image. `imapprox` is based on `rgb2ind` and uses the same approximation methods. Essentially, `imapprox` first calls `ind2rgb` to convert the image to RGB format, and then calls `rgb2ind` to return a new indexed image with fewer colors.

For example, these commands create a version of the `trees` image with 64 colors, rather than the original 128.

```
load trees
[Y,newmap] = imapprox(X,map,64);
imshow(Y, newmap);
```

The quality of the resulting image depends on the approximation method you use, the range of colors in the input image, and whether or not you use dithering. Note that different methods work better for different images. See “Dithering” on page 14-10 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, dithering 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. To illustrate dithering, the following example loads a 24-bit truecolor image, and then uses `rgb2ind` to create an indexed image with just eight colors. The first example does not use dithering, the second does use dithering.

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. 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.

```
[X_dither,map]=rgb2ind(rgb,8,'dither');  
figure, imshow(X_dither,map);
```



Convert from YIQ to RGB 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.

Convert from YCbCr to RGB 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);
```

Convert from HSV to RGB 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.

Note: MATLAB and the Image Processing Toolbox software do not support the HSI color space (Hue, Saturation, Intensity). However, if you want to work with color data in terms of hue, saturation, and intensity, the HSV color space is very similar. Another option is to use the LCH color space (Luminosity, Chroma, and Hue), which is a polar transformation of the CIE $L^*a^*b^*$ color space — see “Device-Independent Color Spaces” on page 14-21.

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.

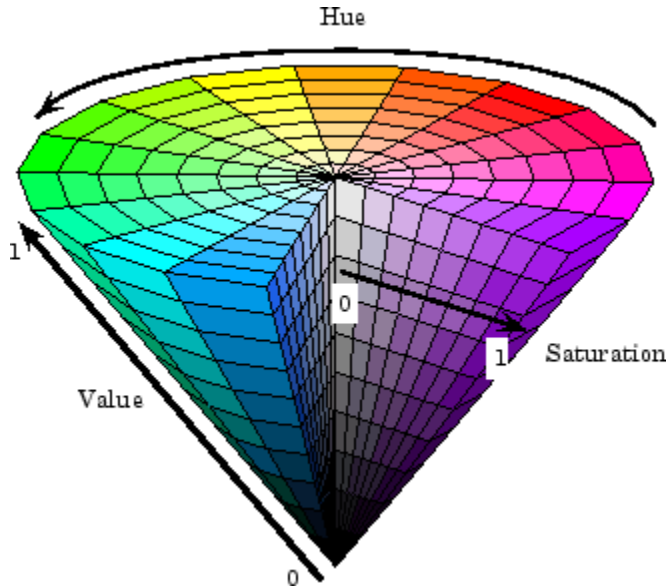


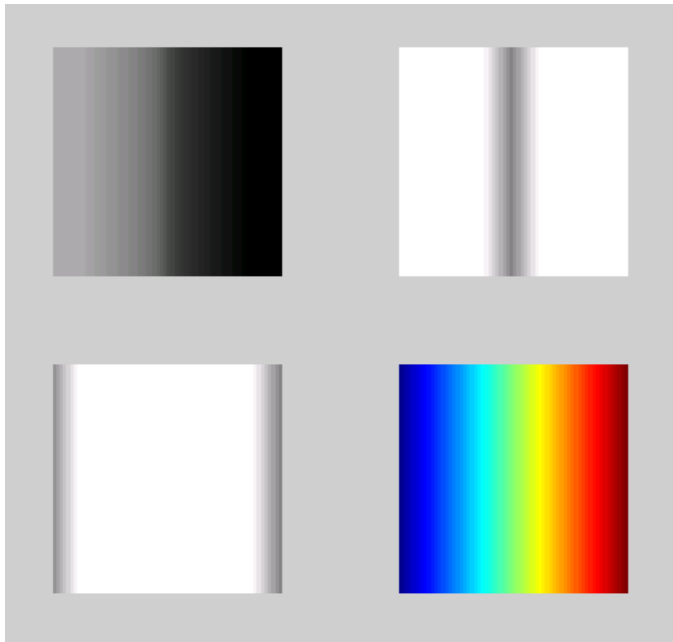
Illustration of the HSV Color Space

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.

Profile-Based Color Space Conversions

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.

Read 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 this structure, 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.

Write ICC 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.

When it creates the output file, `iccwrite` checks the validity of the input profile structure. If any required fields are missing, `iccwrite` returns an error message. For more information about the writing ICC profile data to a file, see the `iccwrite` function reference page. 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.

Convert RGB to CMYK Using ICC Profiles

This example shows how to convert color data from the RGB color space used by a monitor to the CMYK color space used by 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:

Import RGB color space data. This example imports an RGB color image into the MATLAB workspace.

```
I_rgb = imread('peppers.png');
```

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.)

```
outprof = iccread('USSheetfedCoated.icc');
```

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. This example creates a color transformation structure that defines a conversion from RGB color data to CMYK color data. 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.

```
C = makecform('icc',inprof,outprof);
```

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);
```

Write the converted data to a file. To export the CMYK data, use the `imwrite` function, specifying the format as TIF. If the format is TIF 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'

What is Rendering Intent in Profile-Based Conversions?

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.

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.

Convert Between Device-Independent Color Spaces

Image Processing 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 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 , uwl , $u'v'L$, 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
uwl	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 $L^*a^*b^*$

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	<code>uint16</code> or <code>double</code>
xyY	<code>double</code>
uvL	<code>double</code>
u'v'L	<code>double</code>
$L^*a^*b^*$	<code>uint8</code> , <code>uint16</code> , or <code>double</code>
L^*ch	<code>double</code>
RGB	<code>double</code> <code>uint8</code> <code>uint16</code>

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. To change the encoding of XYZ data, use these functions:

- `xyz2double`

- `xyz2uint16`

To change the encoding of L*a*b* data, use these functions:

- `lab2double`
- `lab2uint8`
- `lab2uint16`

To change the encoding of RGB data, use these functions:

- `im2double`
- `im2uint8`
- `im2uint16`

Understanding Color Spaces and Color Space Conversion

The Image Processing Toolbox software 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.

Determine if L*a*b* value is in RGB gamut

This example shows how to use color space conversion to determine if an L*a*b* value is in the RGB gamut. The set of colors that can be represented using a particular color space is called its *gamut*. Some L*a*b* color values may be out-of-gamut when converted to RGB.

Convert an L*a*b* value to RGB. The negative values returned demonstrate that the L*a*b* color [80 -130 85] is not in the gamut of the sRGB color space, which is the default RGB color space used by `lab2rgb`. An RGB color is out of gamut when any of its component values are less than 0 or greater than 1.

```
lab = [80 -130 85];  
lab2rgb(lab)
```

```
ans =
```

```
-0.6210    0.9537   -0.1926
```

Convert the L*a*b* value to RGB, this time specifying a different RGB colorspace, the Adobe RGB (1998) color space. The Adobe RGB (1998) has a larger gamut than sRGB. Use the 'ColorSpace' name-value pair. Because the output values are between 0.0 and 1.0 (inclusive), you can conclude that the L*a*b* color [80 -130 85] is inside the Adobe RGB (1998) gamut.

```
lab2rgb(lab, 'ColorSpace', 'adobe-rgb-1998')
```

```
ans =
```

```
0.1234    0.9522    0.1073
```


Neighborhood and Block Operations

This chapter discusses these generic block processing functions. Topics covered include

- “Neighborhood or Block Processing: An Overview” on page 15-2
- “Performing Sliding Neighborhood Operations” on page 15-3
- “Performing Distinct Block Operations” on page 15-7
- “Using Columnwise Processing to Speed Up Sliding Neighborhood or Distinct Block Operations” on page 15-23

Neighborhood or Block Processing: An Overview

Certain image processing operations involve processing an image in sections, called *blocks* or *neighborhoods*, rather than processing the entire image at once. Several functions in the toolbox, such as linear filtering and morphological functions, use this approach.

The toolbox includes several functions that you can use to implement image processing algorithms as a block or neighborhood operation. These functions break the input image into blocks or neighborhoods, call the specified function to process each block or neighborhood, and then reassemble the results into an output image. The following table summarizes these functions.

Function	Description
<code>nlfilter</code>	Implements sliding neighborhood operations that you can use to process an input image in a pixelwise fashion. For each pixel in the input image, the function performs the operation you specify on a block of neighboring pixels to determine the value of the corresponding pixel in the output image. For more information, see “Performing Sliding Neighborhood Operations” on page 15-3
<code>blockproc</code>	Implements distinct block operations that you can use to process an input image a block at a time. The function divides the image into rectangular blocks, and performs the operation you specify on each individual block to determine the values of the pixels in the corresponding block of the output image. For more information, see “Performing Distinct Block Operations” on page 15-7
<code>colfilt</code>	Implements <i>columnwise processing operations</i> which provide a way of speeding up neighborhood or block operations by rearranging blocks into matrix columns. For more information, see “Using Columnwise Processing to Speed Up Sliding Neighborhood or Distinct Block Operations” on page 15-23.

Performing Sliding Neighborhood Operations

In this section...

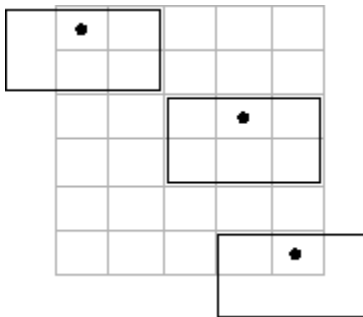
“Understanding Sliding Neighborhood Processing” on page 15-3

“Implementing Linear and Nonlinear Filtering as Sliding Neighborhood Operations” on page 15-5

Understanding Sliding Neighborhood Processing

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. (To operate on an image a block at a time, rather than a pixel at a time, use the distinct block processing function. See “Performing Distinct Block Operations” on page 15-7 for more information.)

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. For information about how the center pixel is determined, see “Determining the Center Pixel” on page 15-4.



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.

General Algorithm of Sliding Neighborhood Operations

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 in Sliding Neighborhood Operations

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 as Sliding Neighborhood Operations

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 “What Is Image Filtering in the Spatial Domain?” 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.

To implement a variety of sliding neighborhood operations, use the `nlfilter` function. `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. `nlfilter` calculates the value of each pixel in the output image by passing the corresponding input pixel's neighborhood to the function.

Note: Many operations that `nlfilter` can implement run much faster if the computations are performed on matrix columns rather than rectangular neighborhoods. For information about this approach, see “Using Columnwise Processing to Speed Up Sliding Neighborhood or Distinct Block Operations” on page 15-23.

For example, this code 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 code 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 = nlfiter(I,[2 3],@myfun);
```

If you prefer not to write code to implement a specific function, 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

Performing Distinct Block Operations

In this section...

“Understanding Distinct Block Processing” on page 15-7

“Implementing Block Processing Using the blockproc Function” on page 15-8

“Applying Padding” on page 15-9

“Block Size and Performance” on page 15-11

“Using Parallel Block Processing on large Image Files” on page 15-13

“Working with Data in Unsupported Formats” on page 15-15

Understanding Distinct Block Processing

In *distinct block* processing, you divide a matrix into m -by- n sections. These sections, or distinct blocks, overlay the image matrix starting in the upper left corner, with no overlap. If the blocks do not fit exactly over the image, you can add padding to the image or work with partial blocks on the right or bottom edges of the image. The following figure shows a 15-by-30 matrix divided into 4-by-8 blocks. The right and bottom edges have partial blocks. You can process partial blocks as is, or you can pad the image so that the resulting size is 16-by-32. For more information, see “Applying Padding” on page 15-9. (To operate on an image a pixel at a time, rather than a block at a time, use the sliding neighborhood processing function. For more information, see “Performing Sliding Neighborhood Operations” on page 15-3.)

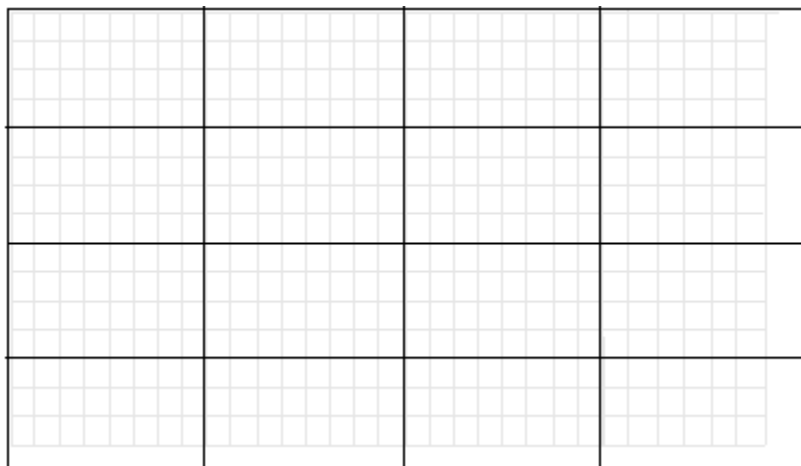


Image Divided into Distinct Blocks

Implementing Block Processing Using the `blockproc` Function

To perform distinct block operations, use the `blockproc` function. The `blockproc` function extracts each distinct block from an image and passes it to a function you specify for processing. The `blockproc` function assembles the returned blocks to create an output image.

For example, the commands below process image `I` in 25-by-25 blocks with the function `myfun`. In this case, the `myfun` function resizes the blocks to make a thumbnail. (For more information about using function handles and anonymous functions, see `function_handle` in the MATLAB Function Reference documentation.)

```
myfun = @(block_struct) imresize(block_struct.data,0.15);  
I = imread('tire.tif');  
I2 = blockproc(I,[25 25],myfun);
```

Note: Due to block edge effects, resizing an image using `blockproc` does not produce the same results as resizing the entire image at once.

The example below uses the `blockproc` function to set every pixel in each 32-by-32 block of an image to the average of the elements in that block. The anonymous function

computes the mean of the block, and then multiplies the result by a matrix of ones, so that the output block is the same size as the input block. As a result, the output image is the same size as the input image. The `blockproc` function does not require that the images be the same size. If this is the result you want, make sure that the function you specify returns blocks of the appropriate size:

```
myfun = @(block_struct) ...
    uint8(mean2(block_struct.data)*...
        ones(size(block_struct.data)));
I2 = blockproc('moon.tif',[32 32],myfun);
figure;
imshow('moon.tif');
figure;
imshow(I2,[]);
```



Original Image



Image with Pixels Set to Average Value

Note: Many operations that `blockproc` can implement run much faster if the computations are performed on matrix columns rather than rectangular blocks. For information about this approach, see “Using Columnwise Processing to Speed Up Sliding Neighborhood or Distinct Block Operations” on page 15-23.

Applying Padding

When processing an image in blocks, you may wish to add padding for two reasons:

- To address the issue of partial blocks
- To create overlapping borders

As described in “Understanding Distinct Block Processing” on page 15-7, if blocks do not fit exactly over an image, partial blocks occur along the bottom and right edges of the image. By default, these partial blocks are processed as is, with no additional padding. Set the 'PadPartialBlocks' parameter to `true` to pad the right or bottom edges of the image and make the blocks full-sized.

You can also add borders to each block. Use the 'BorderSize' parameter to specify extra rows and columns of pixels outside the block whose values are taken into account when processing the block. When there is a border, `blockproc` passes the expanded block, including the border, to the specified function.

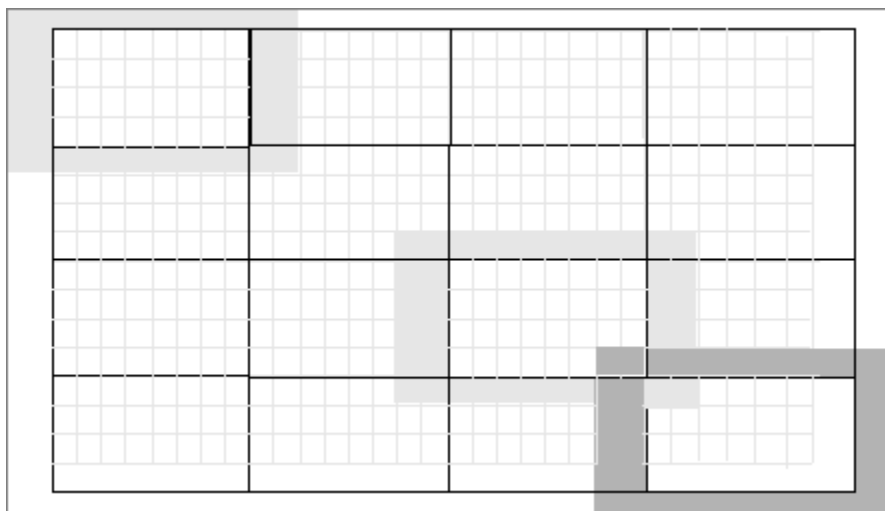


Image Divided into Distinct Blocks with Specified Borders

To process the blocks in the figure above with the function handle `myfun`, the call is:

```
B = blockproc(A,[4 8],myfun,'BorderSize',[1 2], ...  
    'PadPartialBlocks',true)
```

Both padding of partial blocks and block borders add to the overall size of the image, as you can see in the figure. The original 15-by-30 matrix becomes a 16-by-32 matrix due to padding of partial blocks. Also, each block in the image is processed with a 1-

by-2 pixel border—one additional pixel on the top and bottom edges and two pixels along the left and right edges. Blocks along the image edges, expanded to include the border, extend beyond the bounds of the original image. The border pixels along the image edges increase the final size of the input matrix to 18-by-36. The outermost rectangle in the figure delineates the new boundaries of the image after all padding has been added.

By default, `blockproc` pads the image with zeros. If you need a different type of padding, use the `blockproc` function's `'PadMethod'` parameter.

Block Size and Performance

When using the `blockproc` function to either read or write image files, the number of times the file is accessed can significantly affect performance. In general, selecting larger block sizes reduces the number of times `blockproc` has to access the disk, at the cost of using more memory to process each block. Knowing the file format layout on disk can help you select block sizes that minimize the number of times the disk is accessed. See the `blockproc` reference page for more information about file formats.

TIFF Image Characteristics

TIFF images organize their data on disk in one of two ways: in tiles or in strips. A tiled TIFF image stores rectangular blocks of data contiguously in the file. Each tile is read and written as a single unit. TIFF images with strip layout have data stored in strips; each strip spans the entire width of the image and is one or more rows in height. Like a tile, each strip is stored, read, and written as a single unit.

When selecting an appropriate block size for TIFF image processing, understanding the organization of your TIFF image is important. To find out whether your image is organized in tiles or strips, use the `imfinfo` function.

The struct returned by `imfinfo` for TIFF images contains the fields `TileWidth` and `TileLength`. If these fields have valid (nonempty) values, then the image is a tiled TIFF, and these fields define the size of each tile. If these fields contain values of empty (`[]`), then the TIFF is organized in strips. For TIFFs with strip layout, refer to the struct field `RowsPerStrip`, which defines the size of each strip of data.

When reading TIFF images, the minimum amount of data that can be read is a single tile or a single strip, depending on the type of TIFF. To optimize the performance of `blockproc`, select block sizes that correspond closely with how your TIFF image is organized on disk. In this way, you can avoid rereading the same pixels multiple times.

Choosing Block Size

The following three cases demonstrate the influence of block size on the performance of `blockproc`. In each of these cases, the total number of pixels in each block is approximately the same; only the size of the blocks is different.

First, read in an image file and convert it to a TIFF.

```
imageA = imread('concordorthophoto.png','PNG');  
imwrite(imageA,'concordorthophoto.tif','TIFF');
```

Use `imfinfo` to determine whether `concordorthophoto.tif` is organized in strips or tiles.

```
imfinfo concordorthophoto.tif
```

Select fields from the struct appear below:

```
      Filename: 'concordorthophoto.tif'  
      FileSize: 6595038  
      Format: 'tif'  
      Width: 2956  
      Height: 2215  
      BitDepth: 8  
      ColorType: 'grayscale'  
      BitsPerSample: 8  
      StripOffsets: [1108x1 double]  
      SamplesPerPixel: 1  
      RowsPerStrip: 2  
      PlanarConfiguration: 'Chunky'  
      TileWidth: []  
      TileLength: []
```

The value 2 in `RowsPerStrip` indicates that this TIFF image is organized in strips with two rows per strip. Each strip spans the width of the image (2956 pixels) and is two pixels tall. The following three cases illustrate how choosing an appropriate block size can improve performance.

Case 1: Typical Case — Square Block

First try a square block of size `[500 500]`. Each time the `blockproc` function accesses the disk it reads in an entire strip and discards any part of the strip not included in the current block. With two rows per strip and 500 rows per block, the `blockproc` function

accesses the disk 250 times for each block. The image is 2956 pixels wide and 500 rows wide, or approximately six blocks wide ($2956/500 = 5.912$). The `blockproc` function reads the same strip over and over again for each block that includes pixels contained in that strip. Since the image is six blocks wide, `blockproc` reads every strip of the file six times.

```
tic, im = blockproc('concordorthophoto.tif',[500 500],@(s) s.data);  
toc
```

Elapsed time is 17.806605 seconds.

Case 2: Worst Case — Column-Shaped Block

The file layout on the disk is in rows. (Stripped TIFF images are always organized in rows, never in columns.) Try choosing blocks shaped like columns of size [2215 111]. Now the block is shaped exactly opposite the actual file layout on disk.

The image is over 26 blocks wide ($2956/111 = 26.631$). Every strip must be read for every block. The `blockproc` function reads the entire image from disk 26 times. The amount of time it takes to process the image with the column-shaped blocks is proportional to the number of disk reads. With about four times as many disk reads in Case 2, as compared to Case 1, the elapsed time is about four times as long.

```
tic, im = blockproc('concordorthophoto.tif',[2215 111],@(s) s.data);  
toc
```

Elapsed time is 60.766139 seconds.

Case 3: Best Case — Row-Shaped Block

Finally, choose a block that aligns with the TIFF strips, a block of size [84 2956]. Each block spans the width of the image. Each strip is read exactly one time, and all data for a particular block is stored contiguously on disk.

```
tic, im = blockproc('concordorthophoto.tif',[84 2956],@(s) s.data);  
toc
```

Elapsed time is 4.610911 seconds.

Using Parallel Block Processing on large Image Files

If you have a Parallel Computing Toolbox™ license, you can take advantage of multiple processor cores on your machine by specifying the `blockproc` setting `'UseParallel'`

as `true`. Doing so divides the block processing among all available MATLAB sessions to potentially improve the performance of `blockproc`.

What is Parallel Block Processing?

Parallel block processing allows you to process many blocks simultaneously by distributing task computations to a collection of MATLAB sessions, called *workers*. The MATLAB session with which you interact is called the *client*. The client reserves a collection of workers, called a *MATLAB pool*. Then the client MATLAB session divides the input image and sends sections to the worker MATLAB sessions. Each worker processes a subset of blocks and sends the results back to the client. The client MATLAB collects the results into an output variable.

When you set `'UseParallel'` to `true`, `blockproc` uses all available workers in the MATLAB pool to process the input image in parallel.

To read more about parallel computing, see “*Key Problems Addressed by Parallel Computing*” and “*Introduction to Parallel Solutions*” in the Parallel Computing Toolbox User’s Guide.

When to Use Parallel Block Processing

When processing small images, serial mode is expected to be faster than parallel mode. For larger images, however, you may see significant performance gains from parallel processing. The performance of parallel block processing depends on three factors:

- Function used for processing
- Image size
- Block size

In general, using larger blocks while block processing an image results in faster performance than completing the same task using smaller blocks. However, sometimes the task or algorithm you are applying to your image requires a certain block size, and you must use smaller blocks. When block processing using smaller blocks, parallel block processing is typically faster than regular (serial) block processing, often by a large margin. If you are using larger blocks, however, you might need to experiment to determine whether parallel block processing saves computing time.

How to Use Parallel Block Processing

You must meet two conditions to use parallel block processing:

- The source image is not specified as an `ImageAdapter` class.
- A Parallel Computing Toolbox license exists in the MATLAB installation.

If you meet these conditions, you can enable parallel block processing by opening a MATLAB pool:

```
parpool(4)
```

Here, 4 represents the number of workers in your pool. (See the `parpool` reference page for more details.)

After opening a MATLAB pool, you can invoke parallel processing in `blockproc` by specifying `'UseParallel'` as `true`. In the following example, compute a discrete cosine transform for each 8 x 8 block of an image in parallel:

```
blockFun = @(block_struct) dct2(block_struct.data);  
result = blockproc(input_image,[8 8], blockFun, ...  
    'UseParallel',true);
```

Working with Data in Unsupported Formats

In addition to reading TIFF or JPEG2000 files and writing TIFF files, the `blockproc` function can read and write other formats. To work with image data in another file format, you must construct a class that inherits from the `ImageAdapter` class. The `ImageAdapter` class is an “abstract class” that is part of the Image Processing Toolbox software. It defines the signature for methods that `blockproc` uses for file I/O with images on disk. You can associate instances of an Image Adapter class with a file and use them as arguments to `blockproc` for file-based block processing.

This section demonstrates the process of writing an Image Adapter class by discussing an example class (the `LanAdapter` class). The `LanAdapter` class is part of the toolbox. Use this simple, read-only class to process arbitrarily large `uint8` LAN files with `blockproc`.

Learning More About the LAN File Format

To understand how the `LanAdapter` class works, you must first know about the LAN file format. Landsat thematic mapper imagery is stored in the Erdas LAN file format. Erdas LAN files contain a 128-byte header followed by one or more spectral bands of data, band-interleaved-by-line (BIL), in order of increasing band number. The data is stored in little-endian byte order. The header contains several pieces of important information about the file, including size, data type, and number of bands of imagery contained in

the file. The LAN file format specification defines the first 24 bytes of the file header as shown in the table.

File Header Content

Bytes	Data Type	Content
1–6	6-byte character string	'HEADER' or 'HEAD74'
7–8	16-bit integer	Pack type of the file (indicating bit depth)
9–10	16-bit integer	Number of bands of data
11–16	6 bytes	Unused
17–20	32-bit integer	Number of columns of data
21–24	32-bit integer	Number of rows of data

The remaining 104 bytes contain various other properties of the file, which this example does not use.

Parsing the Header

Typically, when working with LAN files, the first step is to learn more about the file by parsing the header. The following code shows how to parse the header of the `rio.lan` file:

```

1  Open the file:

    file_name = 'rio.lan';
    fid = fopen(file_name, 'r');
2  Read the header string:

    header_str = fread(fid, 6, 'uint8=>char');
    fprintf('Header String: %s\n', header_str);
3  Read the pack type:

    pack_type = fread(fid, 1, 'uint16', 0, 'ieee-le');
    fprintf('Pack Type: %d\n', pack_type);
4  Read the number of spectral bands:

    num_bands = fread(fid, 1, 'uint16', 0, 'ieee-le');
    fprintf('Number of Bands: %d\n', num_bands);
5  Read the image width and height:

    unused_bytes = fread(fid, 6, 'uint8', 0, 'ieee-le');
```

```

width = fread(fid,1,'uint32',0,'ieee-le');
height = fread(fid,1,'uint32',0,'ieee-le');
fprintf('Image Size (w x h): %d x %d\n',width,height);
6 Close the file:

fclose(fid);

```

The output appears as follows:

```

Header String: HEAD74
Pack Type: 0
Number of Bands: 7
Image Size (w x h): 512 x 512

```

The `rio.lan` file is a 512 x 512, 7-band image. The pack type of 0 indicates that each sample is an 8-bit, unsigned integer (`uint8` data type).

Reading the File

In a typical, in-memory workflow, you would read this LAN file with the `multibandread` function. The LAN format stores the RGB data from the visible spectrum in bands 3, 2, and 1, respectively. You could create a truecolor image for further processing.

```

truecolor = multibandread('rio.lan', [512, 512, 7],...
    'uint8=>uint8', 128,'bil', 'ieee-le', {'Band','Direct',[3 2 1]});

```

For very large LAN files, however, reading and processing the entire image in memory using `multibandread` can be impractical, depending on your system capabilities. To avoid memory limitations, use the `blockproc` function. With `blockproc`, you can process images with a file-based workflow. You can read, process, and then write the results, one block at a time.

The `blockproc` function only supports reading and writing certain file formats, but it is extensible via the `ImageAdapter` class. To write an Image Adapter class for a particular file format, you must be able to:

- Query the size of the file on disk
- Read a rectangular block of data from the file

If you meet these two conditions, you can write an Image Adapter class for LAN files. You can parse the image header to query the file size, and you can modify the call to

`multibandread` to read a particular block of data. You can encapsulate the code for these two objectives in an Image Adapter class structure, and then operate directly on large LAN files with the `blockproc` function. The `LanAdapter` class is an Image Adapter class for LAN files, and is part of the Image Processing Toolbox software.

Examining the `LanAdapter` Class

This section describes the constructor, properties, and methods of the `LanAdapter` class. Studying the `LanAdapter` class helps prepare you for writing your own Image Adapter class. If you are new to object-oriented programming, see “Developing Classes—Typical Workflow” for general information on writing classes in the MATLAB documentation.

Open `LanAdapter.m` and look at the implementation of the `LanAdapter` class.

Classdef

The `LanAdapter` class begins with the keyword `classdef`. The `classdef` section defines the class name and indicates that `LanAdapter` inherits from the `ImageAdapter` superclass. Inheriting from `ImageAdapter` allows the new class to:

- Interact with `blockproc`
- Define common `ImageAdapter` properties
- Define the interface that `blockproc` uses to read and write to LAN files

Properties

Following the `classdef` section, the `LanAdapter` class contains two blocks of class properties. The first block contains properties that are publicly visible, but not publicly modifiable. The second block contains fully public properties. The `LanAdapter` class stores some information from the file header as class properties. Other classes that also inherit from `ImageAdapter`, but that support different file formats, can have different properties.

```
classdef LanAdapter < ImageAdapter
    properties(GetAccess = public, SetAccess = private)
        Filename
        NumBands
    end

    properties(Access = public)
        SelectedBands
    end
```


In addition to the properties defined in `LanAdapter.m`, the class inherits the `ImageSize` property from the `ImageAdapter` superclass. The new class sets the `ImageSize` property in the constructor.

Methods: Class Constructor

The class constructor initializes the `LanAdapter` object. The `LanAdapter` constructor parses the LAN file header information and sets the class properties. Implement the constructor, a class method, inside a `methods` block.

The constructor contains much of the same code used to parse the LAN file header. The `LanAdapter` class only supports `uint8` data type files, so the constructor validates the pack type of the LAN file, as well as the header string. The class properties store the remaining information. The method responsible for reading pixel data uses these properties. The `SelectedBands` property allows you to read a subset of the bands, with the default set to read all bands.

`methods`

```
function obj = LanAdapter(fname)
    % LanAdapter constructor for LanAdapter class.
    % When creating a new LanAdapter object, read the file
    % header to validate the file as well as save some image
    % properties for later use.

    % Open the file.
    obj.Filename = fname;
    fid = fopen(fname,'r');

    % Verify that the file begins with the string 'HEADER' or
    % 'HEAD74', as per the Erdas LAN file specification.
    header_str = fread(fid,6,'uint8=>char');
    if ~(strcmp(header_str,'HEADER') || strcmp(header_str,...
        'HEAD74'))
        error('Invalid LAN file header.');
```

end

```

    % Read the data type from the header.
    pack_type = fread(fid,1,'uint16',0,'ieee-le');
    if ~isequal(pack_type,0)
        error('Unsupported pack type. The LanAdapter example only...
            supports reading uint8 data.');
```

end

```
% Provide band information.
obj.NumBands = fread(fid,1,'uint16',0,'ieee-le');
% By default, return all bands of data
obj.SelectedBands = 1:obj.NumBands;

% Specify image width and height.
unused_field = fread(fid,6,'uint8',0,'ieee-le'); %#ok<NASGU>
width = fread(fid,1,'uint32',0,'ieee-le');
height = fread(fid,1,'uint32',0,'ieee-le');
obj.ImageSize = [height width];

% Close the file handle
fclose(fid);

end % LanAdapter
```

Methods: Required

Adapter classes have two required methods defined in the abstract superclass, `ImageAdapter`. All Image Adapter classes must implement these methods. The `blockproc` function uses the first method, `readRegion`, to read blocks of data from files on disk. The second method, `close`, performs any necessary cleanup of the Image Adapter object.

```
function data = readRegion(obj, region_start, region_size)
% readRegion reads a rectangular block of data from the file.

% Prepare various arguments to MULTIBANDREAD.
header_size = 128;
rows = region_start(1):(region_start(1) + region_size(1) - 1);
cols = region_start(2):(region_start(2) + region_size(2) - 1);

% Call MULTIBANDREAD to get data.
full_size = [obj.ImageSize obj.NumBands];
data = multibandread(obj.Filename, full_size,...
'uint8=>uint8', header_size, 'bil', 'ieee-le',...
    {'Row', 'Direct', rows},...
    {'Column','Direct', cols},...
    {'Band', 'Direct', obj.SelectedBands});

end % readRegion
```

`readRegion` has two input arguments, `region_start` and `region_size`. The `region_start` argument, a two-element vector in the form `[row col]`, defines the first

pixel in the request block of data. The `region_size` argument, a two-element vector in the form `[num_rows num_cols]`, defines the size of the requested block of data. The `readRegion` method uses these input arguments to read and return the requested block of data from the image.

The `readRegion` method is implemented differently for different file formats, depending on what tools are available for reading the specific files. The `readRegion` method for the `LanAdapter` class uses the input arguments to prepare custom input for `multibandread`. For LAN files, `multibandread` provides a convenient way to read specific subsections of an image.

The other required method is `close`. The `close` method of the `LanAdapter` class appears as follows:

```
function close(obj) %#ok<MANU>
% Close the LanAdapter object. This method is a part
% of the ImageAdapter interface and is required.
% Since the readRegion method is "atomic", there are
% no open file handles to close, so this method is empty.

end

end % public methods

end % LanAdapter
```

As the comments indicate, the `close` method for `LanAdapter` has nothing to do, so `close` is empty. The `multibandread` function does not require maintenance of open file handles, so the `close` method has no handles to clean up. Image Adapter classes for other file formats may have more substantial `close` methods including closing file handles and performing other class clean-up responsibilities.

Methods (Optional)

As written, the `LanAdapter` class can only read LAN files, not write them. If you want to write output to a LAN format file, or another file with a format that `blockproc` does not support, implement the optional `writeRegion` method. Then, you can specify your class as a 'Destination' parameter in `blockproc` and write output to a file of your chosen format.

The signature of the `writeRegion` method is as follows:

```
function [] = writeRegion(obj, region_start, region_data)
```

The first argument, `region_start`, indicates the first pixel of the block that the `writeRegion` method writes. The second argument, `region_data`, contains the new data that the method writes to the file.

Classes that implement the `writeRegion` method can be more complex than `LanAdapter`. When creating a writable Image Adapter object, classes often have the additional responsibility of creating new files in the class constructor. This file creation requires a more complex syntax in the constructor, where you potentially need to specify the size and data type of a new file you want to create. Constructors that create new files can also encounter other issues, such as operating system file permissions or potentially difficult file-creation code.

Using the `LanAdapter` Class with `blockproc`

Now that you understand how the `LanAdapter` class works, you can use it to enhance the visible bands of a LAN file. Run the Computing Statistics for Large Images (`BlockProcessStatisticsExample`) example to see how the `blockproc` function works with the `LanAdapter` class.

Using Columnwise Processing to Speed Up Sliding Neighborhood or Distinct Block Operations

In this section...

“Understanding Columnwise Processing” on page 15-23

“Using Column Processing with Sliding Neighborhood Operations” on page 15-23

“Using Column Processing with Distinct Block Operations” on page 15-24

Understanding Columnwise Processing

Performing sliding neighborhood and distinct block operations columnwise, when possible, 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.

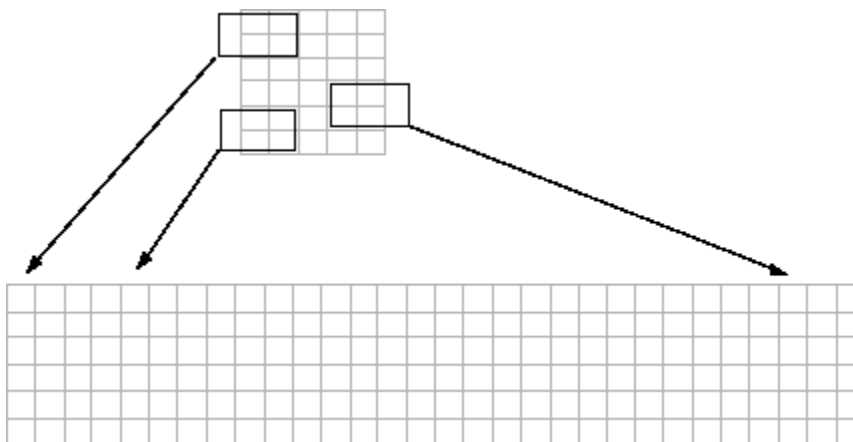
To use column processing, use the `colfilt` function . This function

- 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

Using Column Processing with Sliding Neighborhood Operations

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.

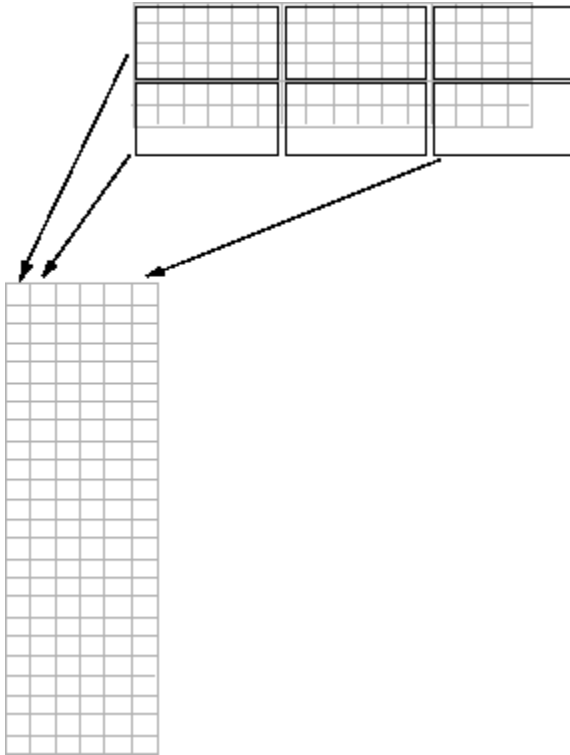
`colfilt` can produce the same results as `nlfilter` with faster execution time; however, it might use more memory. The example below sets each output pixel to the maximum value in the input pixel's neighborhood, producing the same result as the `nlfilter` example shown in “Implementing Linear and Nonlinear Filtering as Sliding Neighborhood Operations” on page 15-5.

```
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.

```
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 `blockproc` performs. However, `colfilt` has certain restrictions that `blockproc` 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 `blockproc`.

Code Generation for Image Processing Toolbox Functions

- “Code Generation for Image Processing” on page 16-2
- “List of Supported Functions with Usage Notes” on page 16-3
- “Generate Code from Application Containing Image Processing Functions” on page 16-11
- “Code Generation Using a Shared Library” on page 16-25

Code Generation for Image Processing

Certain Image Processing Toolbox functions have been enabled to generate C code using MATLAB Coder™. To use code generation with image processing functions, follow these steps:

- Write your MATLAB function or application as you would normally, using functions from the Image Processing Toolbox.
- Add the `%#codegen` compiler directive to your MATLAB code.
- Open the MATLAB Coder app, create a project, and add your file to the project. Once in MATLAB Coder, you can check the readiness of your code for code generation. For example, your code may contain functions that are not enabled for code generation. Make any modifications required for code generation.
- Generate code by clicking **Build** on the Build tab of the MATLAB Coder app. You can choose to build a MEX file, a shared library, a dynamic library, or an executable.

Even if you addressed all readiness issues identified by MATLAB Coder, you might still encounter build issues. The readiness check only looks at function dependencies. When you try to generate code, MATLAB Coder might discover coding patterns that are not supported for code generation. View the error report and modify your MATLAB code until you get a successful build.

For more information about code generation, see the MATLAB Coder documentation. To see an example of using code generation, see “Generate Code from Application Containing Image Processing Functions” on page 16-11.

Note: To generate code from MATLAB code that contains image processing functions, you must have the MATLAB Coder software.

When working with generated code, note the following:

- For some Image Processing Toolbox functions, code generation depends on a precompiled, platform-specific shared library.

List of Supported Functions with Usage Notes

The following table lists the Image Processing Toolbox functions that have been enabled for code generation. You must have the MATLAB Coder software installed to generate C code from MATLAB for these functions.

Image Processing Toolbox provides three types of code generation support:

- Functions that generate C code.
- Functions that generate C code that depends on a platform-specific shared library (.dll, .so, or .dylib). Use of a shared library preserves performance optimizations in these functions, but this limits the target platforms for which you can generate code. For more information, see “Code Generation for Image Processing”.
- Functions that generate C code or C code that depends on a shared library, depending on which target platform you specify in MATLAB Coder. If you specify the generic **MATLAB Host Computer** target platform, these functions generate C code that depends on a shared library. If you specify any other target platform, these functions generate C code.

In generated code, each supported toolbox function has the same name, arguments, and functionality as its Image Processing Toolbox counterpart. However, some functions have limitations. The following table includes information about code generation limitations that might exist for each function. In the following table, all the functions generate C code. The table identifies those functions that generate C code that depends on a shared library, and those functions that can do both, depending on which target platform you choose.

Function	Remarks/Limitations
<code>affine2d</code>	When generating code, you can only specify single objects—arrays of objects are not supported.
<code>bwdist</code>	The <code>method</code> argument must be a compile-time constant. Input images must have fewer than 2^{32} pixels. Generated code for this function uses a precompiled, “platform-specific shared library”.
<code>bwlookup</code>	For best results, specify an input image of class <code>logical</code> . If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, “platform-specific shared library”.

Function	Remarks/Limitations
<code>bwmorph</code>	<p>The text string specifying the operation must be a constant and, for best results, specify an input image of class <code>logical</code>.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
<code>bwpack</code>	Generated code for this function uses a precompiled “platform-specific shared library”.
<code>bwselect</code>	<p>Supports only the 3 and 4 input argument syntaxes: <code>BW2 = bwselect(BW,c,r)</code> and <code>BW2 = bwselect(BW,c,r,n)</code>. The optional fourth input argument, <code>n</code>, must be a compile-time constant. In addition, with code generation, <code>bwselect</code> only supports only the 1 and 2 output argument syntaxes: <code>BW2 = bwselect(____)</code> or <code>[BW2, idx] = bwselect(____)</code>.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>bwtraceboundary</code>	The <code>dir</code> , <code>fstep</code> , and <code>conn</code> arguments must be compile-time constants.
<code>bwunpack</code>	Generated code for this function uses a precompiled “platform-specific shared library”.
<code>conndef</code>	Input arguments must be compile-time constants.
<code>edge</code>	<p>The <code>method</code>, <code>direction</code>, and <code>sigma</code> arguments must be a compile-time constants. In addition, nonprogrammatic syntaxes are not supported. For example, the syntax <code>edge(im)</code>, where <code>edge</code> does not return a value but displays an image instead, is not supported.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>fitgeotrans</code>	<p>The <code>transformtype</code> argument must be a compile-time constant. The function supports the following transformation types: <code>'nonreflectivesimilarity'</code>, <code>'similarity'</code>, <code>'affine'</code>, or <code>'projective'</code>.</p>
<code>fspecial</code>	Inputs must be compile-time constants. Expressions or variables are allowed if their values do not change.
<code>getrangefromclass</code>	—

Function	Remarks/Limitations
<code>histeq</code>	<p>All the syntaxes that include indexed images are not supported. This includes all syntaxes that accept <code>map</code> as input and return <code>newmap</code>.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>im2uint8</code>	Generated code for this function uses a precompiled “platform-specific shared library”.
<code>im2uint16</code>	Generated code for this function uses a precompiled “platform-specific shared library”.
<code>im2int16</code>	Generated code for this function uses a precompiled “platform-specific shared library”.
<code>im2single</code>	—
<code>im2double</code>	—
<code>imadjust</code>	<p>Does not support syntaxes that include indexed images. This includes all syntaxes that accept <code>map</code> as input and return <code>newmap</code>.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>imbothat</code>	<p>The input image <code>IM</code> must be either 2-D or 3-D image. The structuring element input argument <code>SE</code> must be a compile-time constant.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
<code>imclearborder</code>	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant. Supports only up to 3-D inputs.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>imclose</code>	<p>The input image <code>IM</code> must be either 2-D or 3-D image. The structuring element input argument <code>SE</code> must be a compile-time constant.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
<code>imcomplement</code>	Does not support <code>int64</code> and <code>uint64</code> data types.

Function	Remarks/Limitations
imdilata	<p>The input image IM must be either 2-D or 3-D image. The SE, PACKOPT, and SHAPE input arguments must be a compile-time constant. The structuring element argument SE must be a single element—arrays of structuring elements are not supported. To obtain the same result as that obtained using an array of structuring elements, call the function sequentially.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
imerode	<p>The input image IM must be either 2-D or 3-D image. The SE, PACKOPT, and SHAPE input arguments must be a compile-time constant. The structuring element argument SE must be a single element—arrays of structuring elements are not supported. To obtain the same result as that obtained using an array of structuring elements, call the function sequentially.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
imextendedmax	<p>The optional third input argument, conn, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
imextendedmin	<p>The optional third input argument, conn, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>

Function	Remarks/Limitations
imfill	<p>The optional input connectivity, <code>conn</code> and the string 'holes' must be compile-time constants.</p> <p>Supports only up to 3-D inputs.</p> <p>The interactive mode to select points, <code>imfill(BW,0,CONN)</code> is not supported in code generation.</p> <p><code>locations</code> can be a P-by-1 vector, in which case it contains the linear indices of the starting locations. <code>locations</code> can also be a P-by-<code>ndims(I)</code> matrix, in which case each row contains the array indices of one of the starting locations. Once you select a format at compile-time, you cannot change it at run time. However, the number of points in <code>locations</code> can be varied at run time.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
imfilter	<p>The input image can be either 2-D or 3-D. The value of the input argument, <code>options</code>, must be a compile-time constant.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
imhist	<p>The optional second input argument, <code>n</code>, must be a compile-time constant. In addition, nonprogrammatic syntaxes are not supported. For example, the syntaxes where <code>imhist</code> displays the histogram are not supported.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
imhmax	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>

Function	Remarks/Limitations
imhmin	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
imlincomb	<p>The <code>output_class</code> argument must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
imopen	<p>The input image <code>IM</code> must be either 2-D or 3-D image. The structuring element input argument <code>SE</code> must be a compile-time constant.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
imquantize	—
imreconstruct	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
imref2d	<p>The <code>XWorldLimits</code>, <code>YWorldLimits</code> and <code>ImageSize</code> properties can be set only during object construction. When generating code, you can only specify single objects—arrays of objects are not supported.</p>
imref3d	<p>The <code>XWorldLimits</code>, <code>YWorldLimits</code>, <code>ZWorldLimits</code> and <code>ImageSize</code> properties can be set only during object construction. When generating code, you can only specify single objects—arrays of objects are not supported.</p>
imregionalmax	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>

Function	Remarks/Limitations
<code>imregionalmin</code>	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>imtophat</code>	<p>The input image <code>IM</code> must be either 2-D or 3-D image. The structuring element input argument <code>SE</code> must be a compile-time constant.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
<code>imwarp</code>	<p>The geometric transformation object input, <code>tform</code>, must be either <code>affine2d</code> or <code>projective2d</code>. Additionally, the interpolation method and optional parameter names must be string constants.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>intlut</code>	Generated code for this function uses a precompiled “platform-specific shared library”.
<code>iptcheckconn</code>	Input arguments must be compile-time constants.
<code>iptcheckmap</code>	—
<code>label2rgb</code>	<p>Referring to the standard syntax:</p> <pre>RGB = label2rgb(L, map, zerocolor, order)</pre> <ul style="list-style-type: none"> • Submit at least two input arguments: the label matrix, <code>L</code>, and the colormap matrix, <code>map</code>. • <code>map</code> must be an <code>n-by-3, double</code>, colormap matrix. You cannot use a string containing the name of a MATLAB colormap function or a function handle of a colormap function. • If you set the boundary color <code>zerocolor</code> to the same color as one of the regions, <code>label2rgb</code> will not issue a warning. • If you supply a value for <code>order</code>, it must be <code>'noshuffle'</code>.
<code>mean2</code>	—

Function	Remarks/Limitations
medfilt2	The <code>padopt</code> argument must be a compile-time constant. Generated code for this function uses a precompiled “platform-specific shared library”.
multithresh	—
ordfilt2	The <code>padopt</code> argument must be a compile-time constant. Generated code for this function uses a precompiled “platform-specific shared library”.
padarray	Support only up to 3-D inputs. Input arguments, <code>padval</code> and <code>direction</code> are expected to be compile-time constants.
projective2d	When generating code, you can only specify single objects—arrays of objects are not supported.
rgb2ycbcr	—
strel	Input arguments must be compile-time constants. The following methods are not supported for code generation: <code>getsequence</code> , <code>reflect</code> , <code>translate</code> , <code>disp</code> , <code>display</code> , <code>loadobj</code> . When generating code, you can only specify single objects—arrays of objects are not supported.
stretchlim	Generated code for this function uses a precompiled “platform-specific shared library”.
ycbcr2rgb	—

More About

- “MATLAB Coder”

Generate Code from Application Containing Image Processing Functions

This example shows how to generate C code using MATLAB Coder from MATLAB applications that use Image Processing Toolbox functions. The example describes how to setup your MATLAB environment, prepare your MATLAB code for code generation, and work around any issues that you might encounter in your MATLAB code that prevent code generation.

In this section...

“Setup Your Compiler” on page 16-11

“Prepare Your MATLAB Code for Code Generation” on page 16-12

“Generate Code” on page 16-18

Setup Your Compiler

This example shows how to specify which C/C++ compiler you want to use with MATLAB Coder to generate code.

Use the `mex` function with the `-setup` option to specify the C/C++ compiler you want to use with MATLAB Coder.

```
mex -setup
```

```
MEX configured to use 'Microsoft Visual C++ 2010 (C)' for C language compilation.  
Warning: The MATLAB C and Fortran API has changed to support MATLAB variables with more
```

```
To choose a different C compiler, select one from the following:  
Microsoft Visual C++ 2010 (C) mex -setup:C:\matlab\bin\win64\mexopts\msvc2010.xml C  
Intel Visual C++ 12 with Microsoft Visual Studio 2008 (C) mex -setup:C:\matlab\bin\win  
Intel Visual C++ 12 with Microsoft Visual Studio 2010 (C) mex -setup:C:\matlab\bin\win
```

```
To choose a different language, select one from the following:  
mex -setup C++
```

```
mex -setup FORTRAN
```

Prepare Your MATLAB Code for Code Generation

This example shows how to use the MATLAB Coder app to evaluate your code's readiness for code generation. For example, the code may use functions that are not enabled for code generation.

For this example, copy the following code into a file and save it, giving it the name `cellDetectionMATLAB.m`. This code is a modified version of the Image Processing Toolbox example `Detecting a Cell Using Image Segmentation (Detecting a Cell Using Image Segmentation)`. To illustrate code generation, this version of the example removes the display code.

```
function BWfinal = cellDetectionMATLAB(I)
%cellDetectionMATLAB - detect cells using image segmentation.

[~, threshold] = edge(I, 'sobel');
fudgeFactor = .5;
BWs = edge(I, 'sobel', threshold * fudgeFactor);
figure
imshow(BWs)
title('binary gradient mask');

se90 = strel('line', 3, 90);
se0 = strel('line', 3, 0);

BWsdil = imdilate(BWs, [se90 se0]);
figure
imshow(BWsdil)
title('dilated gradient mask');

BWdfill = imfill(BWsdil, 'holes');
figure
imshow(BWdfill);
title('binary image with filled holes');

BWnobord = imclearborder(BWdfill, 4);
figure
imshow(BWnobord)
title('cleared border image');

seD = strel('diamond',1);
```

```

BWfinal = imerode(BWnobord,seD);
BWfinal = imerode(BWfinal,seD);
figure
imshow(BWfinal)
title('segmented image');

```

Test the example code with a sample image. For this example, use the `cell.tif` file, included with the Image Processing Toolbox (`matlab\toolbox\images\imdata\cell.tif`).

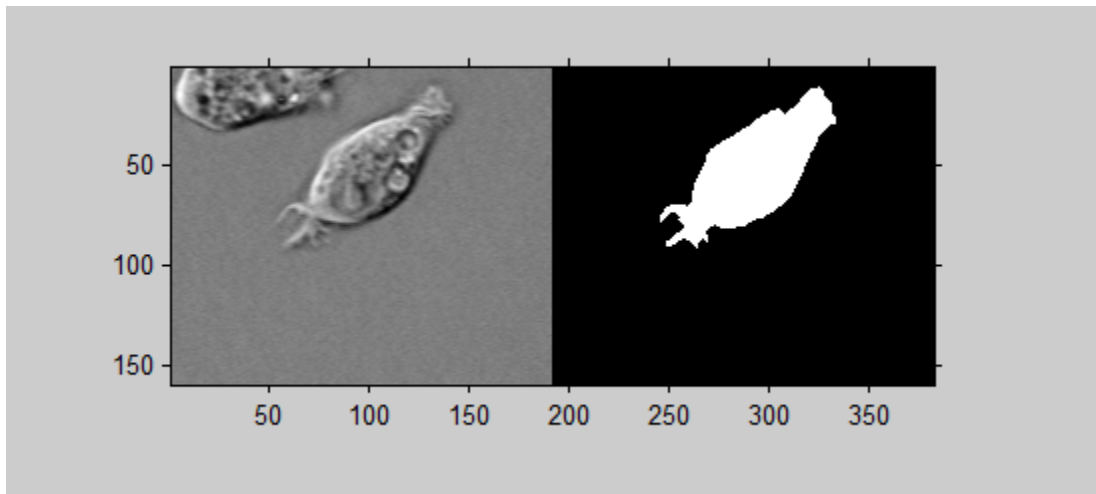
```

I = imread('cell.tif');

Iseg = cellDetectionMATLAB(I);

% Display the original image and the segmented image side-by-side.
imshowpair(I,Iseg,'montage')

```



Create a copy of the MATLAB function for which you want to generate C code. Since you modify this code for code generation, it is good to work with a copy.

```

copyfile('cellDetectionMATLAB.m','cellDetectionCodeGeneration.m');

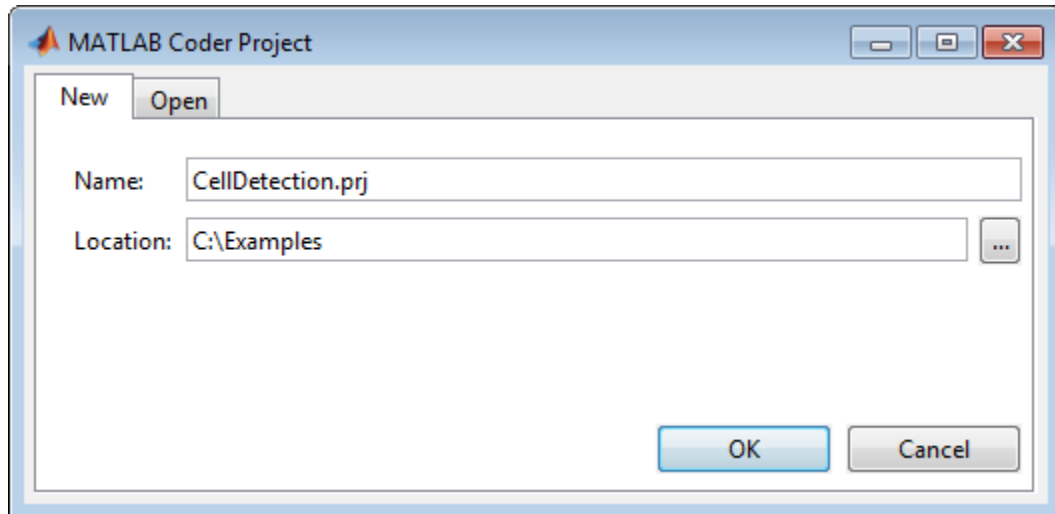
```

As a first step, change the name of the function in the function signature to match the file name and add the MATLAB Coder compilation directive `%%codegen` at the end of the function signature. This directive instructs the MATLAB code analyzer to diagnose issues that would prohibit successful code generation.

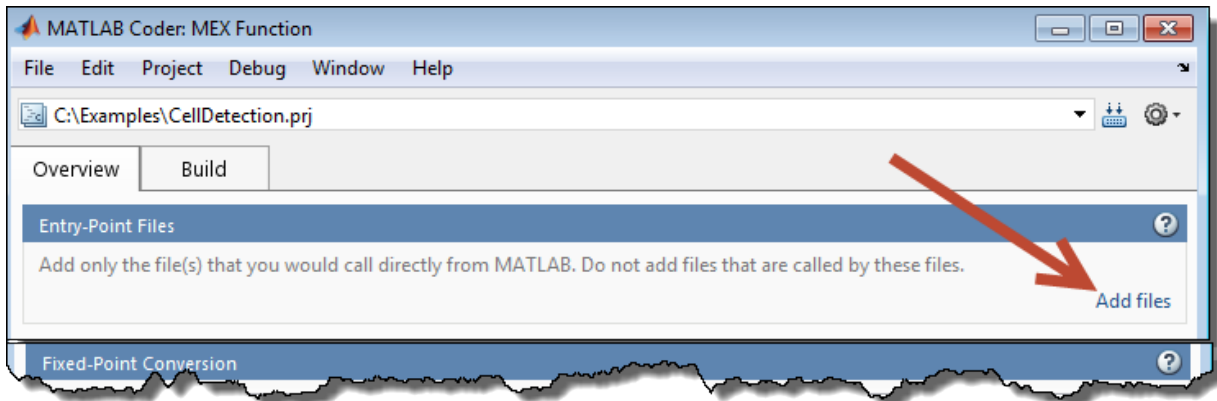
```
function BWfinal = cellDetectionCodeGeneration(I) %#codegen
.  
.  
.
```

Open the MATLAB Coder app. In MATLAB, select the Apps tab, navigate to Code Generation and click the MATLAB Coder app. (Alternatively, you can enter `coder` at the MATLAB command prompt.)

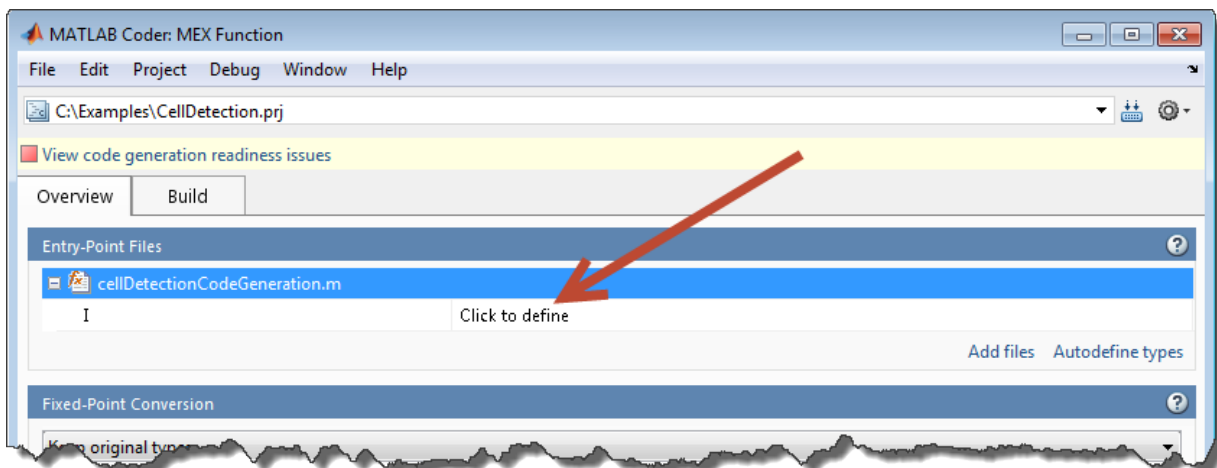
In the MATLAB Coder Project dialog, create a new code generation project. This example names the project `cellDetection.prj`. Click **OK** to create the project and open the MATLAB Coder dialog box.



Add your MATLAB code file to the project. Click **Add files** and, for this example, select `cellDetectionCodeGeneration.m`.

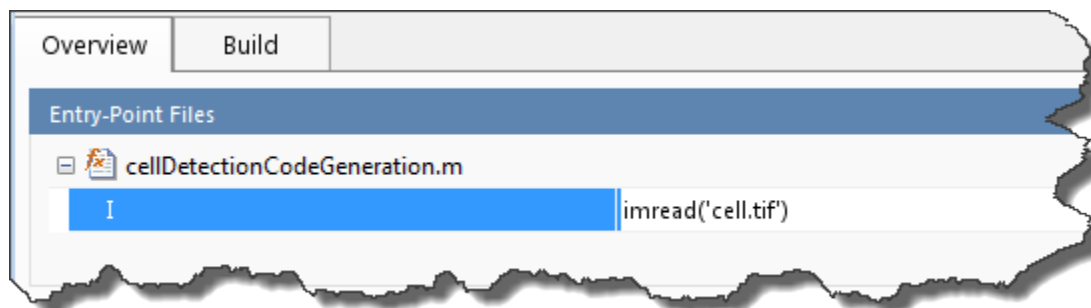


Define the size and data type of the inputs to your function. When you add the file, MATLAB Coder determines the input arguments required by your file and populates the entry-point area. Every input must be specified to be of fixed size, variable size or a constant. For this example, the function accepts one input—an image **I**. To define the size of this input image, click the **Click to define** label.

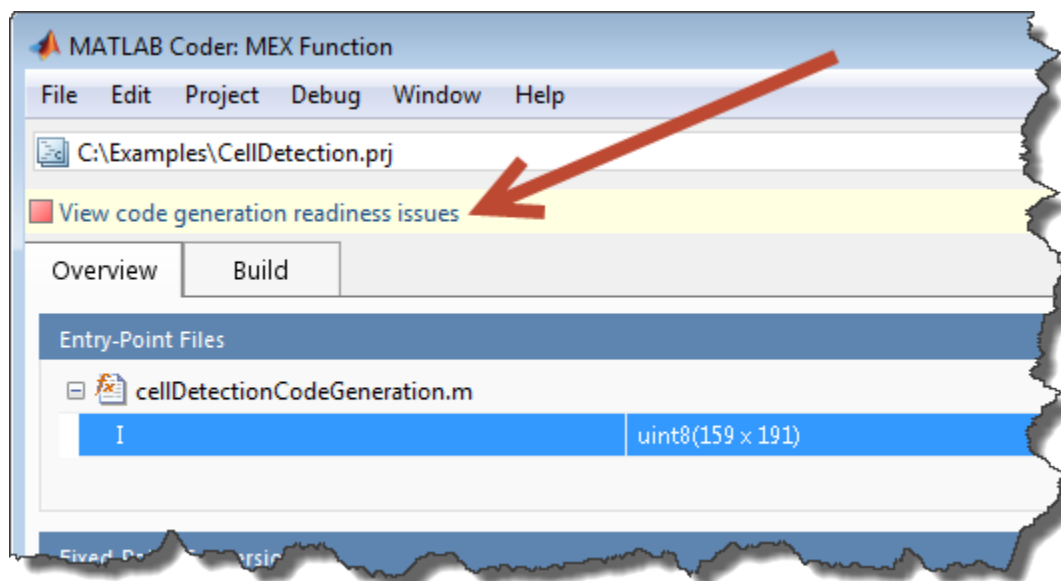


There are several ways to specify the size of your input argument. For fixed size inputs, choose the class and size of the input. For variable-sized inputs, select the class and specify the upper bound on the size. For an unbounded input, select the unbounded

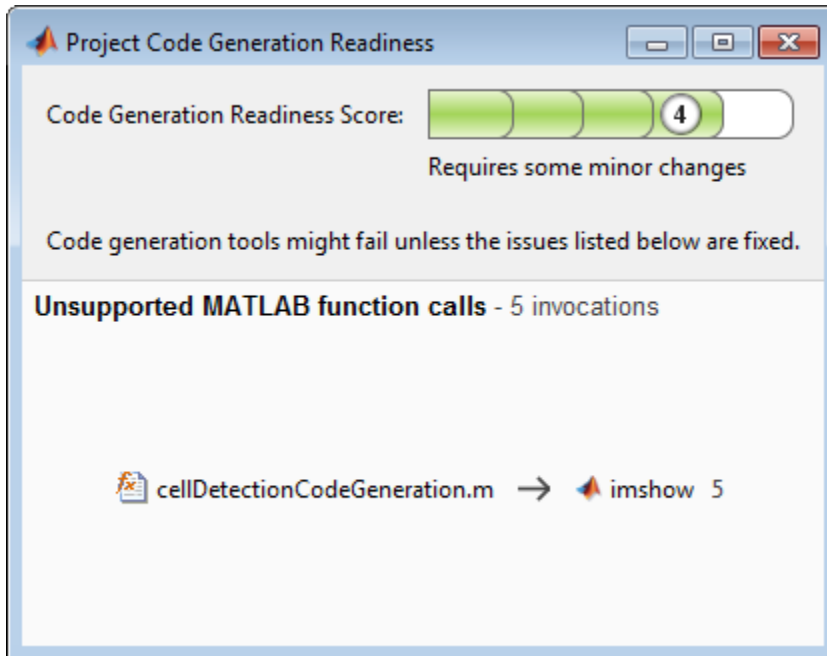
option (: Inf). To define a constant, choose the **Define Constant** option. This example specifies the size and class of the input by providing MATLAB Coder with an example of the input. Click **Define by Example** and specify the example `imread('cell.tif')`. (For more information about defining inputs, see the MATLAB Coder documentation.)



After you specify the size and class of the input, MATLAB Coder displays the **View code generation readiness issues** option, if it finds issues that need addressing. Click this option to view issues that prevent you from generating code.



MATLAB Coder opens the Project Code Generation Readiness dialog, providing a readiness score and a list of issues that must be addressed. For this example, MATLAB Coder highlights one issue: the example code uses a function (`imshow`) that has not been enabled for code generation.



To address this readiness issue, you must remove the calls to `imshow`. This function isn't necessary for the segmentation operation. Modify the example code file, `cellDetectionCodeGeneration.m`, removing all the calls to `imshow` and related functions.

```
function BWfinal = cellDetectionCodeGeneration(I) %#codegen
%cellDetectionMATLAB - detect cells using image segmentation.
```

```
[~, threshold] = edge(I, 'sobel');
fudgeFactor = .5;
BWs = edge(I, 'sobel', threshold * fudgeFactor);
```

```
se90 = strel('line', 3, 90);
se0 = strel('line', 3, 0);
```

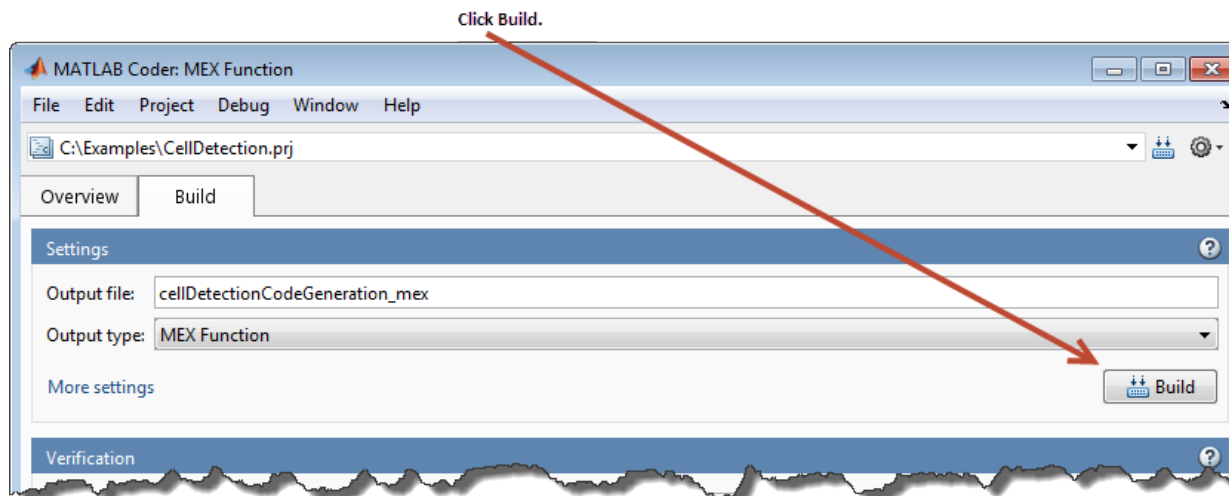
```
BWsdil = imdilate(BWs, [se90 se0]);  
BWdfill = imfill(BWsdil, 'holes');  
BWnobord = imclearborder(BWdfill, 4);  
  
seD = strel('diamond',1);  
BWfinal = imerode(BWnobord,seD);  
BWfinal = imerode(BWfinal,seD);
```

When you have addressed all readiness issues, the MATLAB Coder dialog no longer displays the **View code generation readiness issues** link.

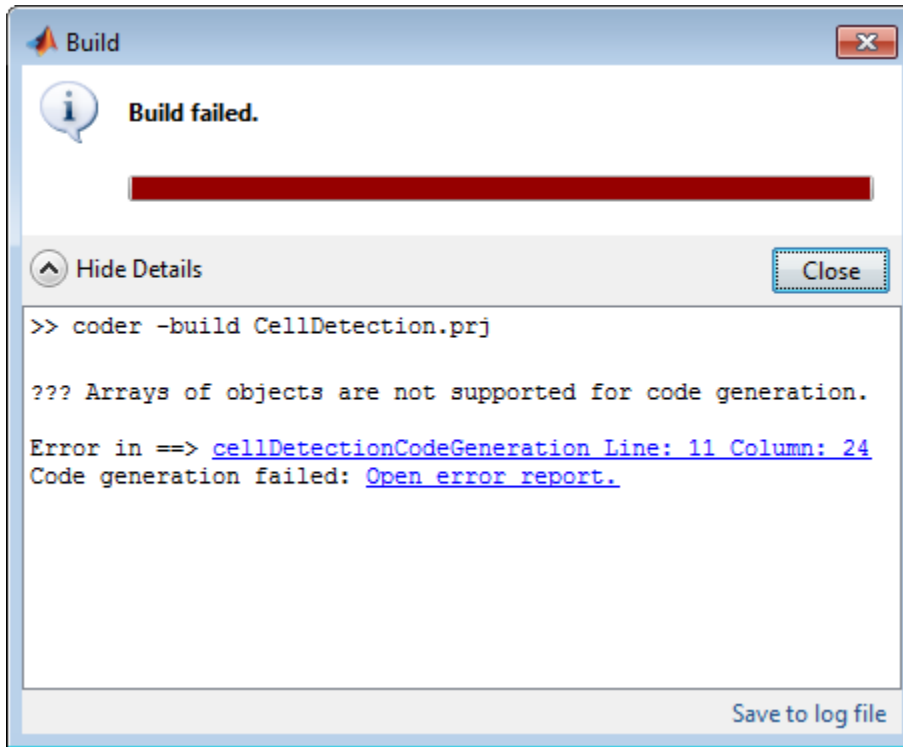
Generate Code

This example shows how to generate C code from MATLAB code, using MATLAB Coder. Note that, even though you performed the MATLAB Coder readiness checks, additional issues might arise during the build process that can prevent code generation. While the readiness checks look at function dependencies to determine readiness, the build process examines coding patterns. This example illustrates how to address issues that arise in the build process that prevent successful code generation.

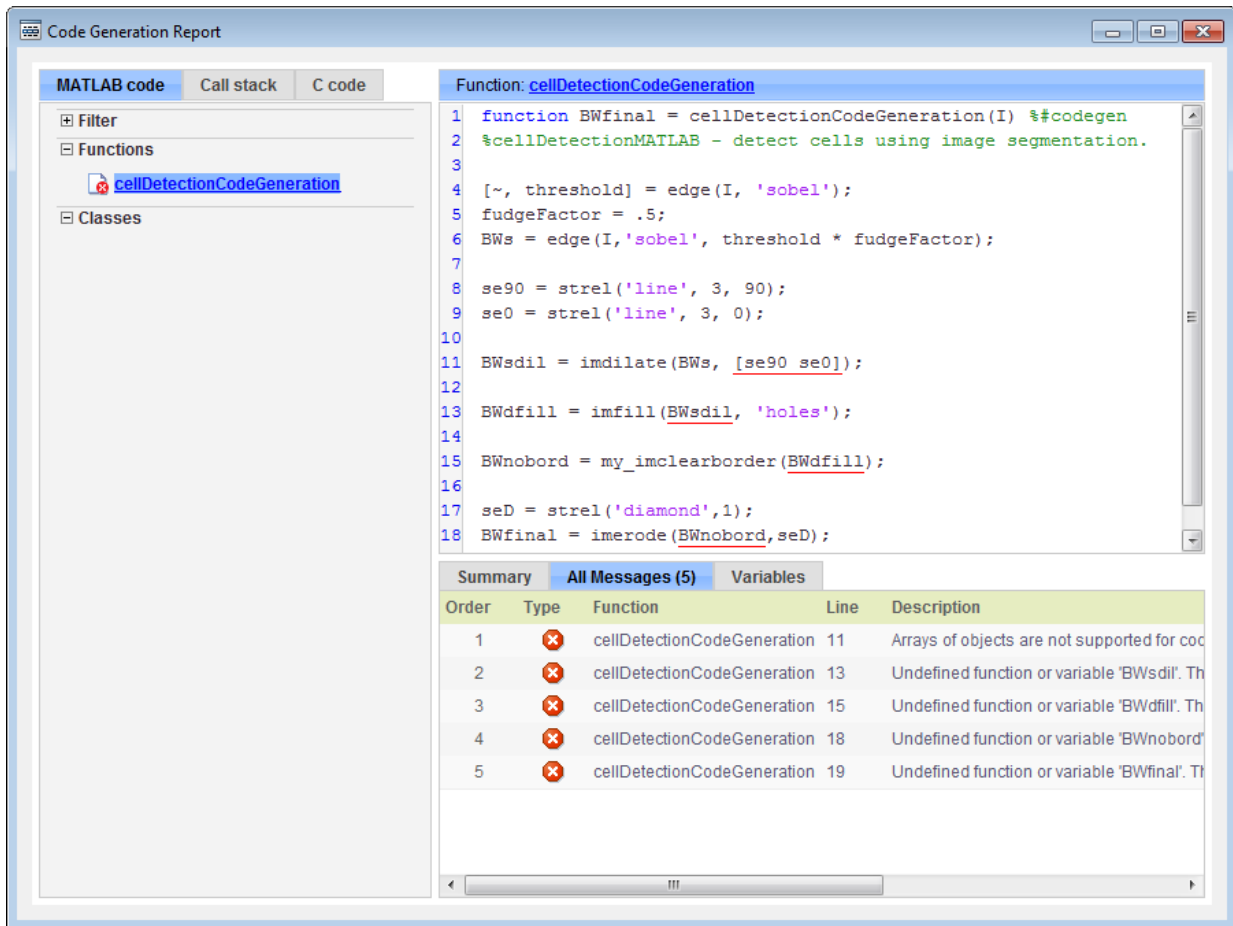
Select the Build tab and click **Build**.



Initially, the build fails.



Click **Open Error Report** to get details of the errors. The report lists the errors and uses red underlining to highlight the location of errors in the source code. For this example, the first error indicates that on line 11 the code passes in an array of objects and code generation does not support arrays of objects passed as arguments. The remaining errors are a result of this first error.



The screenshot shows the 'Code Generation Report' window. The left sidebar contains a tree view with 'MATLAB code' selected, showing a filter and a list of functions including 'cellDetectionCodeGeneration'. The main area displays the MATLAB code for the function 'cellDetectionCodeGeneration' with line numbers 1 through 18. The code includes comments and function calls like 'edge', 'strel', 'imdilate', 'imfill', 'my_imclearborder', and 'imerode'. Below the code is a 'Summary' table with the following data:

Order	Type	Function	Line	Description
1	✘	cellDetectionCodeGeneration	11	Arrays of objects are not supported for code generation.
2	✘	cellDetectionCodeGeneration	13	Undefined function or variable 'BWsdil'. The function was not found in the search path.
3	✘	cellDetectionCodeGeneration	15	Undefined function or variable 'BWdfill'. The function was not found in the search path.
4	✘	cellDetectionCodeGeneration	18	Undefined function or variable 'BWnobord'. The function was not found in the search path.
5	✘	cellDetectionCodeGeneration	19	Undefined function or variable 'BWfinal'. The function was not found in the search path.

Address issues identified in the report. For this example, you must modify the call to `imdilate` to avoid passing an array of strel objects. This is accomplished by replacing the single call to `imdilate`, where the example passes an array of strel objects, with two separate calls to `imdilate`, highlighted in the following figure.

```
function BWfinal = cellDetectionCodeGeneration(I) %#codegen
%cellDetectionMATLAB - detect cells using image segmentation.

[~, threshold] = edge(I, 'sobel');
fudgeFactor = .5;
BWs = edge(I, 'sobel', threshold * fudgeFactor);

se90 = strel('line', 3, 90);
se0 = strel('line', 3, 0);

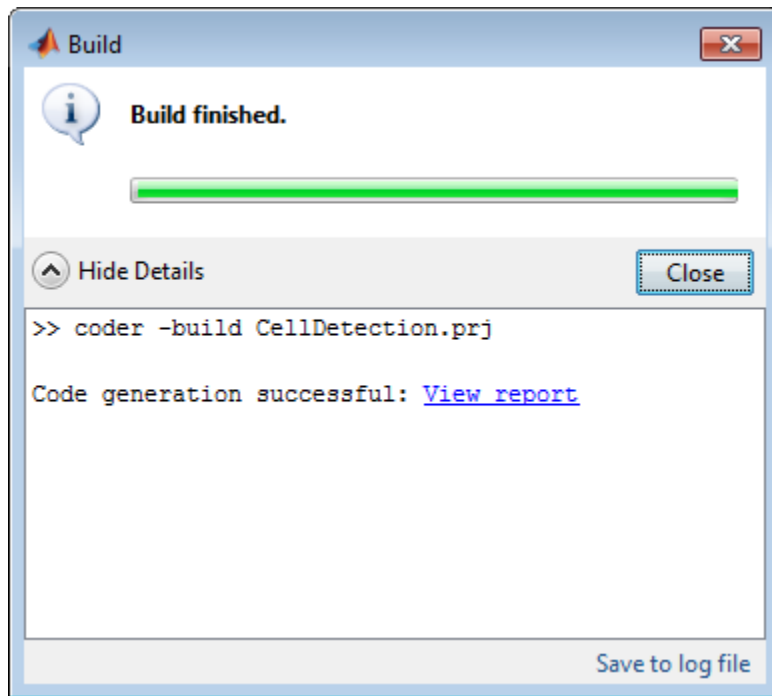
BWsdil = imdilate(BWs, se90);
BWsdil = imdilate(BWsdil, se0);

BWdfill = imfill(BWsdil, 'holes');

BWnobord = imclearborder(BWdfill, 4);

seD = strel('diamond', 1);
BWfinal = imerode(BWnobord, seD);
BWfinal = imerode(BWfinal, seD);
```

With that change, the build process succeeds and MATLAB Coder generates code.



By default, MATLAB Coder generates a MEX function with the name `cellDetectionCodeGeneration_mex`, but you can choose to generate other types of code. Use the **Output type** menu to generate a C/C++ static library, dynamic library, or standalone executable. MATLAB Coder creates a `codegen` subfolder in your work folder that contains the generated output. For more information about generating code, see the MATLAB Coder documentation.

Test your code. Run the `mex` version of your function and then compare the output with the regular MATLAB code.

```
im = imread('cell.tif');  
out = cellDetectionCodeGeneration_mex(im);  
imshow(out);  
  
isequal(out,cellDetectionMATLAB(im))  
  
ans =  
  
1
```

If you chose the C/C++ Dynamic Library option, MATLAB Coder generates the following interface. This interface can be used to integrate with an external application:

```
type codegen\dll\cellDetectionCodeGeneration\cellDetectionCodeGeneration.h
```

```
/*
 * cellDetectionCodeGeneration.h
 *
 * Code generation for function 'cellDetectionCodeGeneration'
 *
 */

#ifndef __CELLDETECTIONCODEGENERATION_H__
#define __CELLDETECTIONCODEGENERATION_H__
/* Include files */
#include <math.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>

#include "rtwtypes.h"
#include "cellDetectionCodeGeneration_types.h"

/* Function Declarations */
#ifdef __cplusplus
extern "C" {
#endif
extern void cellDetectionCodeGeneration(const unsigned char I[30369], boolean_T BWfinal);
#ifdef __cplusplus
}
#endif
#endif
/* End of code generation (cellDetectionCodeGeneration.h) */
```

The function signature exposed indicates there are two inputs. The first input **I** is expected to be an `unsigned char` with 30369 elements. This is used to pass the input image on which cell detection is to be performed. The type of the input image in MATLAB (`uint8`) is represented as an `unsigned char`. The size of the image is 159 x 191, i.e. 30369 elements. The number of elements is specified through the interface since the input was defined to be fixed size. The second input `BWfinal` is expected to be a `boolean_T` with 30369 elements. This is used to pass a pointer to the output image.

Note that there is a difference in the way MATLAB and C store arrays in memory. MATLAB stores arrays in column-major order and C stores arrays in row-major order. To illustrate, consider the following array:

```
[ 1  2  
  3  4 ]
```

MATLAB stores the array in memory as [1 3 2 4] where C stores the elements as [1 2 3 4]. You must account for this difference when working with arrays.

You can also generate a standalone executable using the "C/C++ Executable" option. In this case a main function that invokes the generated code must be specified. Note that there are no ready-made utilities for importing data. For an example, refer to the Using Dynamic Memory Allocation for an "Atoms" Simulation in the MATLAB Coder documentation.

More About

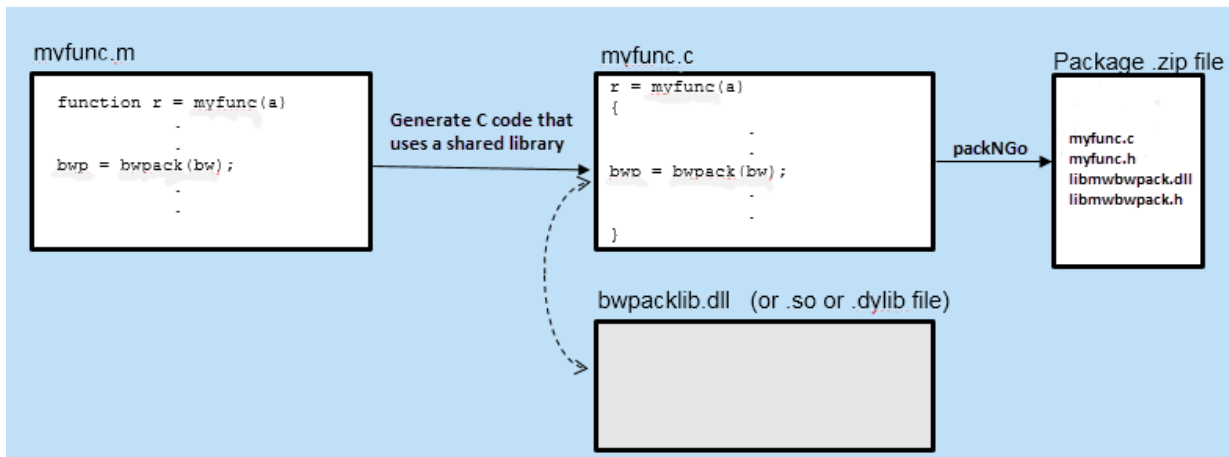
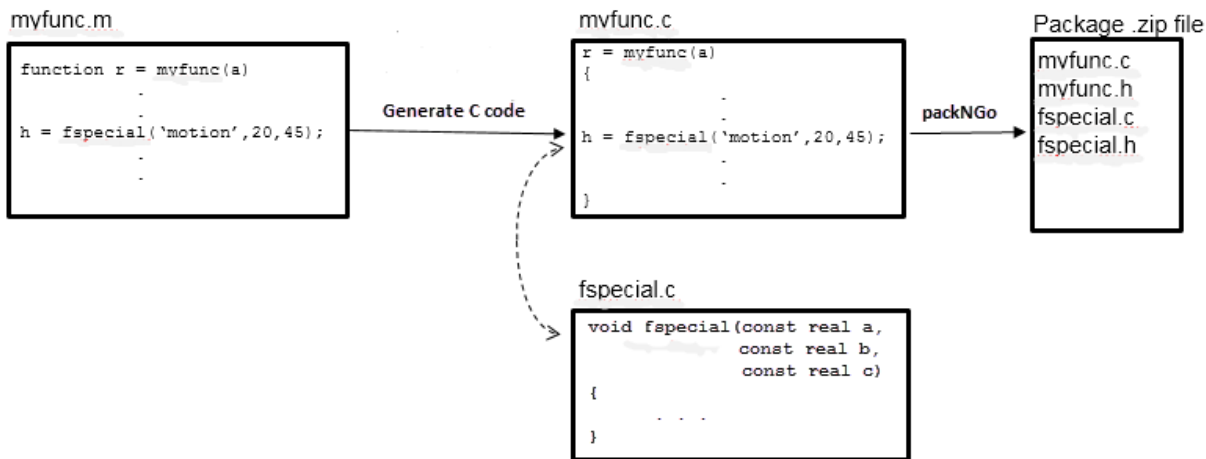
- "Code Generation Workflow"

Code Generation Using a Shared Library

The Image Processing Toolbox includes many functions that support the generation of efficient, C/C++ code using MATLAB Coder. These functions support code generation in several ways:

- Some functions generate standalone C/C++ code that can be incorporated into applications that run on many platforms, such as ARM processors.
- Some functions generate C/C++ code that use a platform-specific shared library. The Image Processing Toolbox uses this shared library approach to preserve performance optimizations, but this limits the platforms on which you can run this code to only platforms that can host MATLAB. To view a list of host platforms, see system requirements.
- Some functions can generate either standalone C/C++ code or code that depends on a shared library, depending upon which target you choose in the MATLAB Coder configuration settings. If you choose the generic **MATLAB Host Computer** option, these functions deliver code that uses a shared library. If you choose any other platform option, these functions deliver portable C/C++ code.

The following diagram illustrates the difference between generating portable C/C++ code and generating code that uses a shared library.



For a complete list of Image Processing Toolbox that support code generation, with information about the type of code generated, see “List of Supported Functions with Limitations and Other Notes”

GPU Computing with Image Processing Toolbox Functions

- “Image Processing on a GPU” on page 17-2
- “List of Supported Functions with Limitations and Other Notes” on page 17-4
- “Perform Thresholding and Morphological Operations on a GPU” on page 17-7
- “Perform Element-wise Operations on a GPU” on page 17-11

Image Processing on a GPU

To take advantage of the performance benefits offered by a modern Graphics Processing Unit (GPU), certain Image Processing Toolbox functions have been enabled to perform image processing operations on a GPU. This can provide GPU acceleration for complicated image processing workflows. These techniques can be implemented exclusively or in combination to satisfy design requirements and performance goals. To perform an image processing operation on a GPU, follow these steps:

- Move the data from the CPU to the GPU. You do this by creating an object of type `gpuArray`, using the `gpuArray` function.
- Perform the image processing operation on the GPU. Any toolbox function that accepts a `gpuArray` object as an input can work on a GPU. For example, you can pass a `gpuArray` to the `imfilter` function to perform the filtering operation on a GPU. For a list of all the toolbox functions that have been GPU-enabled, see “List of Supported Functions with Limitations and Other Notes” on page 17-4.
- Move the data back onto the CPU from the GPU. Applications typically move the data from the GPU back to the CPU after processing, using the `gather` function.

For more information:

- For general information about using GPUs, see “Transfer Arrays Between Workspace and GPU”.
- To see an example of performing image processing operations on a GPU, see “Perform Thresholding and Morphological Operations on a GPU” on page 17-7.
- To see an example of using the MATLAB `arrayfun` function to perform element-wise or pixel based operations on a GPU, see “Perform Element-wise Operations on a GPU” on page 17-11.
- To learn about integrating custom CUDA kernels directly into MATLAB to accelerate complex algorithms, see “Run CUDA or PTX Code on GPU”.

Note: To run image processing code on a graphics processing unit (GPU), you must have the Parallel Computing Toolbox software.

When working with a GPU, note the following:

- Performance improvements can depend on the GPU device.
- There may be small differences in the results returned on a GPU from those returned on a CPU.

List of Supported Functions with Limitations and Other Notes

The following table lists all the Image Processing Toolbox functions that have been enabled to run on a GPU. In most cases, the functions support the same syntaxes and operate the same, but in some cases there are certain differences. This table lists these limitations, if any.

Function	Remarks/Limitations
<code>bwdist</code>	Inputs must be 2-D and have less than 2^{32-1} elements. Euclidean is the only distance metric supported.
<code>bwlabel</code>	—
<code>bwlookup</code>	—
<code>bwmorph</code>	—
<code>corr2</code>	—
<code>edge</code>	Canny method is not supported on the GPU.
<code>histeq</code>	—
<code>im2double</code>	—
<code>im2int16</code>	—
<code>im2single</code>	—
<code>im2uint8</code>	—
<code>im2uint16</code>	—
<code>imabsdiff</code>	Only <code>single</code> and <code>double</code> are supported
<code>imadjust</code>	—
<code>imbothat</code>	<code>gpuArray</code> input must be of type <code>uint8</code> or <code>logical</code> and the structuring element must be flat and two-dimensional.
<code>imclose</code>	<code>gpuArray</code> input must be of type <code>uint8</code> or <code>logical</code> and the structuring element must be flat and two-dimensional.
<code>imcomplement</code>	—
<code>imdilate</code>	<code>gpuArray</code> input must be of type <code>uint8</code> or <code>logical</code> and the structuring element must be flat and two-dimensional The <code>PACKOPT</code> syntaxes are not supported on the GPU.

Function	Remarks/Limitations
<code>imerode</code>	<code>gpuArray</code> input must be of type <code>uint8</code> or <code>logical</code> and the structuring element must be flat and two-dimensional The <code>PACKOPT</code> syntaxes are not supported on the GPU.
<code>imfill</code>	Inputs must be 2-D, supporting only the 2-D connectivities, 4 and 8. Does not support the interactive hole filling syntax.
<code>imfilter</code>	Input kernel must be 2-D
<code>imgradient</code>	—
<code>imgradientxy</code>	—
<code>imhist</code>	When running on a GPU, <code>imhist</code> does not display the histogram. To display the histogram, use <code>stem(X,counts)</code> .
<code>imlincomb</code>	—
<code>imnoise</code>	—
<code>imopen</code>	<code>gpuArray</code> input must be of type <code>uint8</code> or <code>logical</code> and the structuring element must be flat and two-dimensional.
<code>imreconstruct</code>	—
<code>imregdemons</code>	The parameter ' <code>PyramidLevels</code> ' is not supported on the GPU.
<code>imresize</code>	Only cubic interpolation is supported on GPU and function always performs antialiasing.
<code>imrotate</code>	The ' <code>bicubic</code> ' interpolation mode used in the GPU implementation of this function differs from the default (CPU) <code>bicubic</code> mode. The GPU and CPU versions of this function are expected to give slightly different results.
<code>imshow</code>	—
<code>imtophat</code>	<code>gpuArray</code> input must be of type <code>uint8</code> or <code>logical</code> and the structuring element must be flat and two-dimensional.
<code>iradon</code>	The GPU implementation of this function supports only Nearest-neighbor and linear interpolation methods.
<code>mat2gray</code>	—
<code>mean2</code>	—
<code>medfilt2</code>	Padding options are not supported on the GPU

Function	Remarks/Limitations
normxcorr2	—
padarray	—
radon	—
rgb2gray	—
rgb2ycbcr	—
std2	—
stdfilt	—
stretchlim	—
ycbcr2rgb	—

Perform Thresholding and Morphological Operations on a GPU

This example shows how to perform image processing operations on a GPU. The example uses filtering to highlight the watery areas in a large aerial photograph.

Read an image into the workspace.

```
imCPU = imread('concordaerial.png');
```

Move the image to the GPU by creating a `gpuArray` object.

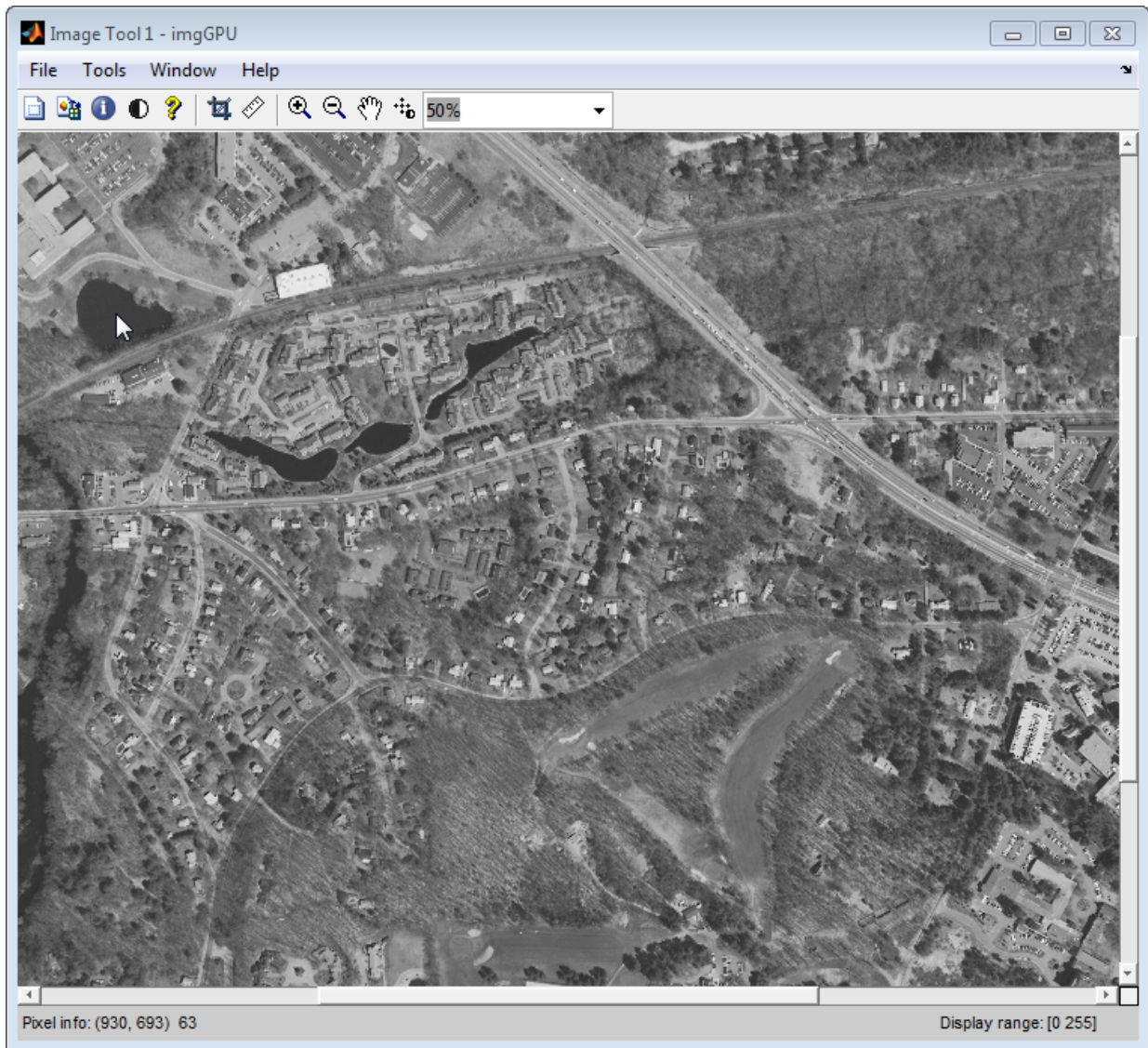
```
imGPU = gpuArray(imCPU);
```

As a preprocessing step, change the RGB image to a grayscale image. Because you are passing it a `gpuArray`, `rgb2gray` performs the conversion operation on a GPU. If you pass a `gpuArray` as an argument, a function that has been GPU-enabled performs the operation on the GPU.

```
imGPUgray = rgb2gray(imGPU);
```

View the image in the Image Viewer and inspect the pixel values to find the value of watery areas. Note that you must bring the image data back onto the CPU, using the `gather` function, to use the Image Viewer. As you move the mouse over the image, you can view the value of the pixel under the cursor at the bottom of the Image Viewer. In the image, areas of water have pixel values less than 70.

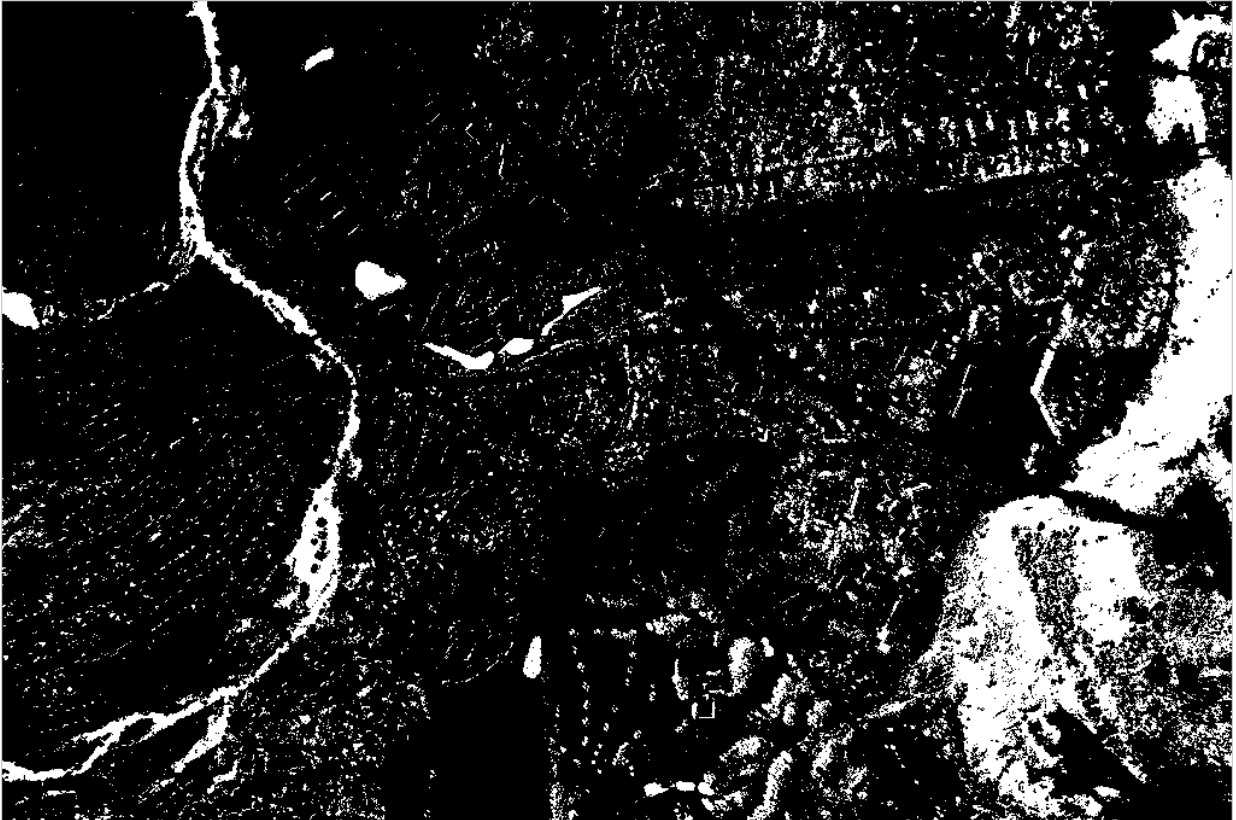
```
imtool(gather(imGPUgray));
```



Filter the image on the GPU to get a new image that contains only the pixels with values of 70 or less and view it.

```
imWaterGPU = imGPUgray<70;
```

```
figure;imshow(imWaterGPU);
```



Using morphological operators that are supported on the GPU, clean up the mask image, removing points that do not represent water.

```
imWaterMask = imopen(imWaterGPU, strel('disk',4));  
imWaterMask = bwmorph(imWaterMask, 'erode',3);
```

Blur the mask image, using `imfilter`.

```
blurH = fspecial('gaussian',20,5);  
imWaterMask = imfilter(single(imWaterMask)*10, blurH);
```

Boost the blue channel to identify the watery areas.


```
blueChannel = imGPU(:,:,3);  
blueChannel = imlincomb(1, blueChannel,6, uint8(imWaterMask));  
imGPU(:,:,3) = blueChannel;
```

View the result. The `imshow` function can work with images on the GPU.

```
figure;imshow(imGPU);
```



After filtering the image on the GPU, move the data back to the CPU., using the `gather` function, and write the modified image to a file.

```
outCPU      = gather(imGPU);  
imwrite(outCPU, 'concordwater.png');
```

Perform Element-wise Operations on a GPU

This example shows how to perform element-wise, or pixel-based, operations on a GPU by using functions that send both the data and operations to the GPU for processing. This method is most effective for element-wise operations that require 2 or more data sets.

Move the data from the CPU to the GPU by creating a `gpuArray`

```
I = imread('concordaerial.png');  
Igpu = gpuArray(I);
```

Create a custom function that performs element-wise operations. This example creates a custom grayscale conversion function using weighted RGB data.

```
function gray = rgb2gray_custom(r,g,b)  
gray = 0.5*r + 0.25*g + 0.25*b;
```

Perform the operation on the GPU. Use `arrayfun` to pass the handle to the custom function and data object to the GPU for evaluation.

```
Igray_gpu = arrayfun(@rgb2gray_custom,Igpu(:,:,1),Igpu(:,:,2),Igpu(:,:,3));
```

Move the data back to the CPU from the GPU, using the `gather` function.

```
I_gpuresult = gather(Igray_gpu);
```

